Spatially-Aware Information Retrieval on the Internet

SPIRIT is funded by EU IST Programme
Contract Number: IST-2001-35047

A Working Searching System

Deliverable number:	D10 2101
Deliverable type:	P
Contributing WP:	WP 2
Contractual date of delivery:	1st January 2004
Actual date of delivery:	31st December 2003
Authors:	Joho, H., Clough, P., and Sanderson, M.
Keywords:	Information retrieval system

Abstract: This report describes the functionality of the text searching system, the architecture of a PC cluster to handle a terabyte size of collection, and finally the data format for the integration with other closely related components.

Contents

Executive Summary

This report describes the functionality of the text searching system, the architecture of the PC cluster to handle a terabyte size of collection, and finally the data format for the integration with other closely related components.

D10 2101

A Working Searching System

1. Introduction

The searching system presented in this report is designed to provide an infrastructure for other components in the project (e.g. the ontology, spatial indexing, and relevance ranking) to achieve spatially-aware searches.

This report first describes the basic concepts and functionality of the searching system along with some examples in the context of Glass system which provides the core applications of indexing and searching.

The report then shows the extension of the basic system to a terabyte size of collection by presenting the architecture of the PC cluster that has been built at Sheffield, followed by a summary of data format used in the transaction between closely related components.

The appendix includes a sample document, the stopwords list, statistics of SPIRIT collection, and the manual of Glass system to support the understanding of this report.

2. Information retrieval system

2.1. Basic concepts

Salton and McGill (1983) describe an Information Retrieval system as: *a system used to store items or information that need to be processed, searched, retrieved and disseminated to various user populations*. Some of the core components in our searching system are based on *Glass* which provides the necessary modules to build such a system and handle the processing involved with indexing and retrieving information in a textual format. Several C programs provide core functionality including basic index and search routines. The motivation behind Glass is not one of providing a packaged IR system (e.g. like INQUERY (Callan, et al., 1992) or Okapi (Robertson, et al., 1997), but rather to provide a modular set of components that programmers and researchers alike can combine to create an IR environment. This enables users to perform controlled experiments and create prototypical IR systems.

Glass runs from the command-line and does not come with a user-interface. However, researchers at Sheffield University have developed a Web-based interface using CGI scripts based on the command-line programs. Researchers are also developing a Simple Object Access Protocol (SOAP)[1] interface to Glass which enables retrieval services to be called from a remote machine to that on which Glass is loaded. The flexibility of the system can be seen through projects which have adopted Glass as their information retrieval system, including

[1] SOAP: a protocol specification for invoking methods on servers, services, components and objects and uses XML and HTTP as the method invocation mechanism: http://www.develop.com/soap/.

CiQuest (CiQuest Project, 2003), EuroVision (Eurovision Project, 2003), and MIND (MIND Project, 2004).

Documents are stored in an inverted index and user information needs are expressed in the form of queries to retrieve documents from the store. Queries are matched to documents using a similarity measure based upon term co-occurrence. Any document containing at least one or more terms from the query is retrieved from the index and a similarity score computed for that document-query pair. Documents containing any number of query terms are retrieved (creating an OR'ing effect) to create the relevance set. Within this, documents are ranked in descending order of similarity under the assumption that those nearer the top of the ranked list are more relevant to the user than those nearer the bottom.

The similarity measure used in Glass is the BM25 weighting operator which can also be found in the Okapi probabilistic IR system. The BM25 function estimates term frequency as Poisson in distribution, and takes into account inverse document frequency and document length. BM25 is based on three sources of weighting which have empirically shown to be useful for different retrieval tasks:

- **Document frequency** - terms occurring in only a few documents are likely to more useful than terms appearing in many.
- **Term frequency** - the more frequent a term appears in a document the more important it is likely to be for that document.
- **Document length** - a term occurring the same number of times in a short document than a longer one is likely to be more important in the shorter one.

All three of these components are present in the BM25 weighting function:

$$\sum_{T \in Q} w * \frac{(k_1+1)tf}{K+tf} \frac{(k_3+1)qtf}{k_3+qtf} + k_2 * |Q| * \frac{avdl-dl}{avdl+dl} \qquad (1)$$

where:

Q is a query contain terms T

w is the Robertson/Spark Jones weight of T in Q

K is $k_1 * \left((1-b)+b*\frac{dl}{avdl}\right)$

k_1, b, k_2, k_3, and k_4 are parameters which depend upon the nature of queries and possibly the database. We use the parameters from TREC-7 experiments (Robertson, et al. 1998): $k_1 = 1.2$, $b = 0.75$, $k_2 = 0$, and $k_3 = 1000$.

tf is the frequency of occurrence of term within a specific document

qtf is the frequency of occurrence of the term within the topic from which Q was derived

dl and $avdl$ are the document length and average document length (arbitrary units), respectively.

The last component of the BM25 operator is added when longer queries are used (called the Query Frequency, or QF). In TREC experiments with Okapi, it is common to find $k_2 = 0$ (as used in Robertson, et al. (1998)), enabling Equation 1 to be re-written as:

$$\sum_{T \in Q} w * \frac{(k_1+1)tf}{K+tf} \frac{(k_3+1)qtf}{k_3+qtf} \quad (2)$$

In both Equations 1 and 2, the weighting function w (the Inverse Document Frequency or IDF) reduces to the following with no relevance feedback (when R=r=0):

$$w = \log \frac{N-n+0.5}{n+0.5} \quad (3)$$

where:

N is the number of documents in the collection

n is the number of documents containing the term

Within the probabilistic retrieval model, this weighting is used to predict which documents are probably relevant to a user query. Two major advantages of Glass over other systems are: 1) its simplicity and 2) its modularity which make this a useful tool for experimenting with, and teaching the rudimentary of information retrieval. Being an experimental platform for IR, Glass also includes evaluation modules that allow comparison with existing experiments on the TREC collections, and modules exist to ease the conversion between custom and TREC-specific data formats.

2.2. Installation

Given that Glass is installed under a *unix environment (with C shell in the example), two things must be done before Glass is run:

- Insert the following two lines in your `.cshrc`:
  - `setenv GLASS_PRIVATE private dir`
  - `source ~/GLASS/GLASS_vars`
- In the file `GLASS_vars`, set `$GLASS_ROOT` to the directory location Glass is in (e.g. `/usr/local/GLASS`).

Finally make the files with the commands `GLASS_make clean` and `GLASS_make all`. The `$GLASS_ROOT/bin/i686/` directory contains executables which perform basic IR functions such as word tokenisation, word stemming, stopword removal and indexing. Another directory which contains useful Shell scripts is the `$GLASS_ROOT/sh_scr/` directory. These scripts use the executable programs to create more high-level tasks such as indexing, retrieval and relevance feedback.

2.3. Basic utilities

Glass contains a large collection of program executables and Shell scripts, given in Appendices. However, out of these there are four scripts which perform the most used functions:

- **index**: this script takes a document collection in TREC format (after tokenisation) and creates a Glass inverted index.
- **retrieve**: this script takes a set of query words (given by a Glass tokeniser) and searches the Glass index for relevant documents. The script returns a ranked list of results sorted in descending order of query-document similarity.

Further functions exist to provide relevance feedback, evaluate the results and convert between TREC and non-TREC data formats. These and other functions are explained more fully and with examples in the next section.

2.4. Tokenisation

An important stage of both the indexing and searching of documents is word tokenisation. The isolation of word-type units from, or tokenisation of, an input character stream is a problem common to many language engineering areas. Typically tokenisation comes before more linguistically-intensive processes such as morphological analysis, but is itself a non-trivial task as one must decide what constitutes a word from potentially large volumes of running text (Grefenstette & Tapanainen, 1994), e.g. should *it's* be treated as two words or one? The simplest approach is to tokenise on whitespace and assume that words are separated by whitespace. Using Perl regular expressions, one could split a sentence into words using the following code:

```
@words = split(/s+/, $sentence);
```

Glass provides a number of tokenisers for different input formats, e.g. TREC and UTF-8, and these can be combined with other programs, such as word stemmers and case conversion routines. Glass tokenisers read from standard input. To tokenise a character stream into words, the following command can be used:

```
% echo "Isn't it ashame." | toke_simple_words b
```

The program `toke_simple_words` takes the standard input and tokenises the text to produce a list of words of block type `a' in the following form.

```
b      Isn't
b      it
b      ashame.
```

Note, the program does not remove punctuation and keeps the original case. The command can be combined with other programs to perform functions such as stemming and case conversion. For example

```
% echo "Isn't it ashame." | toke_simple_words b | word_lower b |
word_stem_porter b
```

produces

```
b      isn't
b      it
b      asham
```

2.5. Indexing a custom collection

Glass assumes the collection to be indexed comes in the same format as the TREC collections of data. A set of documents must first be gathered into one text file, which we call a *data file*. Documents within this single file are separated from each other using textual annotations and assigning them a unique identifier. Annotations come in the form of `<BEGIN>` and `</END>` tags using the SGML markup language. Using annotation allows the specification of sub-elements within documents such as title, author and summary, as well as the main body of text. Documents are divided using the `<DOC>` tags and must be given a unique identification code as captured with the `<DOCNO>` tag. Before the main body of text which is demarcated using the <TEXT> tag, a headline must be given using the `<HEADLINE>` tag. For example in the SPIRIT collection, an entry in the data file looks Figure 1.

Note that the sample document in Figure 1 contains only visible texts in the original HTML document (which has been formatted in a SGML scheme in order to be compatible with the collection used in TREC Web Track, See Appendix I). The extraction of visible texts was carried out in part by a text-based web browser called *lynx* which is a standard application in *unix environment[2].

```
<DOC>
<DOCNO>SPRT-009-060-045-0022108</DOCNO>
<HEADLINE>
 British Council Education Information Service
</HEADLINE>
<TEXT>

   Education Information Service Welcome

      Welcome to the British Council's Education Information
Service.

              British Council Home Page To the Virtual Campus

                 Produced in Britain by the British Council
                          The British Council 2000

   The British Council, registered in England as a charity no.
209131, is
                            the United Kingdom's
   international network for education, culture and
development services.
</TEXT>
</DOC>
```

Figure 1 Sample document (SPRT-009-060-045-0022108)

Once the document collection has been transformed, it can be indexed. Given a data file called `foo.dat`, the collection is indexed using the following command (which one might typically redirect to `/dev/null`):

```
% toke_parsed_generic_TREC foo.dat | index PRE -d
```

The index command has a few options (which can be seen by typing index at the command prompt) and PRE is a prefix for the index files which can be the name of the collection. The option "-d" uses the default setting of converting characters to lowercase, removing stopwords (see Appendix II for the list of stopwords) and removing word suffixes using the Porter stemming algorithm (Porter, 1980). The index command creates an inverted index of the text contained in the document body, and a separate index is created for the text contained in the headline (called PRE_head). Given a PRE name, e.g. test, the following files are generated under the current directory (not including the headline index which provides the same information):

[2] http://lynx.browser.org/

- **test.dii**: this file contains a summary of the data file, e.g. number of lines indexed.
- **test.dol**: this file contains each term and lists all documents containing that term and term frequency.
- **test.ti**: this file contains the term index, a list of terms with term and document frequencies.
- **test.dsi**: this contains information about each document in the collection, e.g. average document length, document length and byte offset in the data file.

2.6. Searching the index

The role of the index files can be summarised in Figure 2. In short, for each word in a query, `test.ti` file looks up the byte offset of document lists for the word in `test.dol`, then `test.dol` looks up the byte offset of document internal IDs mapping for a document, and finally `test.dsi` looks up the actual document file (external document ID) based on the internal document ID. `test.dii` is looked up when the summary data is required (not very often). The detail of these index files can be found in Appendix IV.

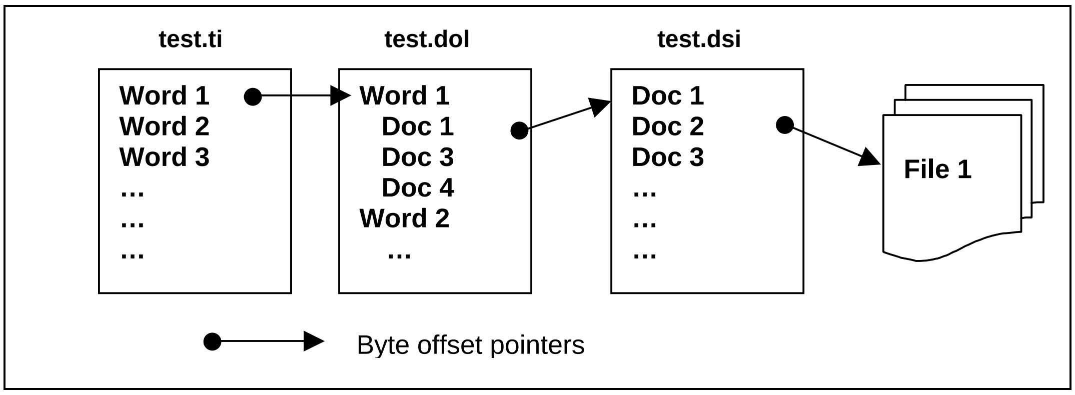

Figure 2 Overview of Index files

Once the index has been created, it can be searched using the `retrieve` Shell script. This expects the name of the index (PRE), some parameters (can use default “-d”), and a query from the standard input. Core modules in Glass use a specific data format when passing results between them. This format must be adherred to by custom scripts and programs, but to ease this the Shell scripts provide this wrapper. For example, the `retrieve` script expects the input to have come from a tokenisation program first, therefore to query a collection using the retrieve script, the following command could be used:

```
% echo 'word_1 word_2 word_3' | toke_simple_words b  | retrieve
PRE –d
```

The tokenised output gets passed to the retrieve script which outputs a list of documents matching the query, ranked in descending order of similarity. For example if we issue the query “hotels in Paris” to one of the SPIRIT indexes, we obtain the following results:

```
r    SPRT-021-039-083-0041034   2002928 355    UNPRCSSD      1
r    SPRT-021-040-056-0027681   2042216 352    UNPRCSSD      2
r    SPRT-021-013-040-0019878   690573  351    UNPRCSSD      3
r    SPRT-021-041-012-0005604   2066192 349    UNPRCSSD      4
r    SPRT-021-040-037-0018420   2032955 348    UNPRCSSD      5
```

where the columns indicate (from left):R token, External Doc ID, Internal Doc ID, BM25 score, Relevance Information, and Rank.

In order to fetch the text of a document, the following command could be used:

```
% index_docs_get_doc PRE Internal_Doc_ID
```
or
```
% index_docs_get_doc_external PRE External_Doc_ID
```

2.7. Some functionality specific to the project

Tokenisation

The current default parameters for the tokenisation used both in indexing and retrieving of SPIRIT collection are:

- Case sensitivity: Off (i.e. all words are stored in lower cases)
- Stopword removal: Off
- Stemming: Off

The main motivation for this setting (despite the stopword removal and stemming would reduce the size of index files) is that, first, many stopwords such as ‘in’, ‘inside’, or ‘of’ are potentially useful for spatially-aware searches, and, second, the suffix stripping by a stemming can eliminate the difference between of words that can be useful for, again, spatially-aware searches. For example, the porter stemming will produce “London” from both “London” and “Londoner”. However, this setting should be revised when the integration with other components is progressed.

Query matching

Although the default probabilistic model are not designed to perform the AND’ing operation in the query-document matching process, the current system is configured to retrieve the documents in which all query words appear (i.e. AND’ing is on).

Query disambiguation and relevance feedback

The final SPIRIT system will handle both freetext and spatially-aware queries. Take “hotels in paris” for instance, the freetext mode will treat the query words in the way described above. In the spatially-aware mode, the place name, Paris, will be disambiguated by our ontology (See Deliverable D5 3101) and the footprint will be given.

Similarly, the relevance feedback, which is usually carried out by the analysis of word frequency in a set of retrieved documents, will be performed in the combination of the ontology and relevance ranking.

3. Extension to a terabyte collection

Now that the functionality of the searching system has been described in the context of Glass, and some examples of how the system works have been described. This section will describe the extension of the system into a terabyte size of collection, followed by the format of data that are used in the transaction between other search components.

3.1. Summary of SPIRIT collection

The SPIRIT collection consists of approx. 94 million web pages (1.2 terabyte). The data set used of the collection was originally crawled by Clarke and his colleagues at Department of Computer Science, University of Waterloo (Clarke, et al., 2002) in Mid-2001. It started from a set of pre-defined URLs in educational domains. The data was then brought and distributed to a cluster of 24 PCs at the Department of Information Studies, University of Sheffield. Approximately 50-60GB of data was distributed to each PC[3]. Each page was modified to a format used in TREC Web Track (see Appendix I) and saved as a single file. Table 1 shows the number of documents across the cluster.

Server	No. of docs	Server	No. of docs
SPIRIT-2	4,158,622	SPIRIT-14	3,905,786
SPIRIT-3	3,912,743	SPIRIT-15	3,593,654
SPIRIT-4	3,793,784	SPIRIT-16	3,881,621
SPIRIT-5	3,762,365	SPIRIT-17	4,148,390
SPIRIT-6	4,103,188	SPIRIT-18	4,150,275
SPIRIT-7	4,040,795	SPIRIT-19	4,039,572
SPIRIT-8	4,028,474	SPIRIT-20	4,181,966
SPIRIT-9	4,003,355	SPIRIT-21	3,464,087
SPIRIT-10	4,170,146	SPIRIT-22	3,845,657
SPIRIT-11	3,602,087	SPIRIT-23	3,842,650
SPIRIT-12	4,225,347	SPIRIT-24	3,951,296

Table 1 Number of documents across the cluster PCs

Other statistics of the collection can be found in Appendix III.

3.2. Architecture of the cluster

Currently one of the cluster PCs (SPIRIT-1) is served as the control machine and it distributes a submitted query to the rest of machines (SPIRIT-2 to 25) where each member contains a set of indexes of a portion of the collection (See Figure 3). When the retrieval of all members is completed, the control machine merges the results returned from them.

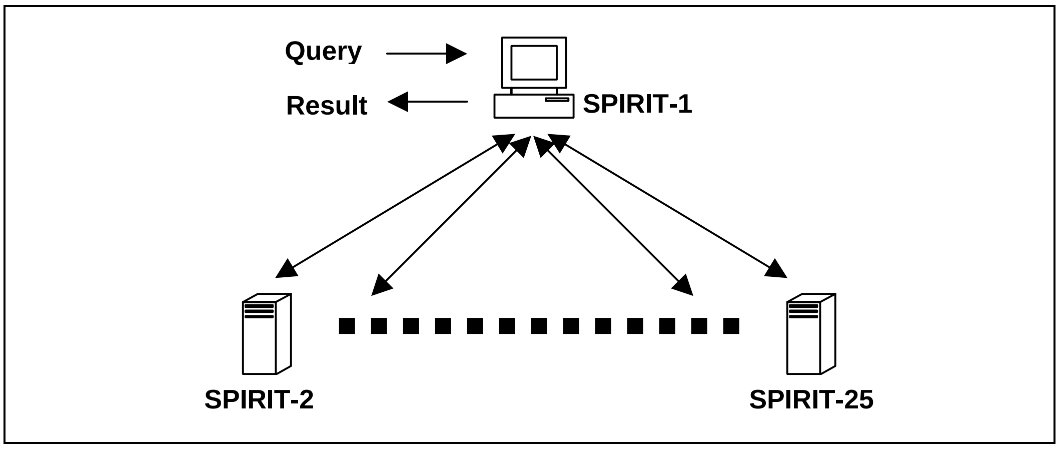

Figure 3 Cluster architecture

SPIRIT-2 to 25 mount the Glass directory on SPIRIT-1 using NFS (Network File System[4]) which allows the machines to access to the Glass directory as if one of the local directories.

[3] Equipped with P4 2.0GHz, 250GB HDD, and 500MB RAM. OS: Linux 2.4.
[4] http://nfs.sourceforge.net/

The retrieval commands and their results between SPIRIT-1 and SPIRIT-2 to 25 are transferred through a SSH connection for a security purpose.

The merging of the results is currently carried out by a so-called “Round-Robin” approach, which takes the top document ranked by each machine, and the second, third, and so forth. The reason why the BM25 score is not directly used for the merging is that the BM25 measure is in part based on the total number of documents in a collection (which differs in each machine). However, it is possible to improve the merging process by incorporating with a score given by the relevance ranking (see Deliverable D14 5201), and which is one of the aims of the project.

3.3. SOAP API

SOAP API allows a remote user to carry out a retrieval on the SPIRIT collection. Figure 4 shows a sample request to our SOAP server (submitting a query “hotels in paris” and requesting to retrieve the top five documents), and Figure 5 shows a part of the response.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
SOAP-ENV: encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<namesp1:doSearch xmlns:namesp1="urn:SPIRITSearch">
<c-gensym3 xsi:type="xsd:string">hotels in paris</c-gensym3>
<c-gensym5 xsi:type="xsd:int">0</c-gensym5>
<c-gensym7 xsi:type="xsd:int">5</c-gensym7>
</namesp1:doSearch>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 4 Sample SOAP request

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:namesp2="http://namespaces.soaplite.com/perl"
xmlns:namesp3="http://xml.apache.org/xml-soap"
SOAP-ENV:   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<namesp1:doSearchResponse xmlns:namesp1="urn:SPIRITSearch">
<return xsi:type="namesp2:doSearchResult">
<searchQuery xsi:type="xsd:string">hotels in paris </searchQuery>
<startIndex xsi:type="xsd:int">0</startIndex>
<resultElements
SOAP-ENC:arrayType="namesp3:SOAPStruct[5]" xsi:type="SOAP-ENC:Array">
<item xsi:type="namesp3:SOAPStruct">
<url xsi:type="xsd:string"
>http://www.paris.org/Kiosque/apr98/champs.elysees.1948.html</url>
<score xsi:type="xsd:int">57121</score>
<title xsi:type="xsd:string">Paris Kiosque - The Champs-Elysses - March
1998</title>
<size xsi:type="xsd:string">21k</size>
<docid xsi:type="xsd:string">SPRT-025-035-083-0041151</docid>
<rank xsi:type="xsd:int">1</rank></item>
…
</resultElements>
<endIndex xsi:type="xsd:int">5</endIndex>
<estimatedTotalResultCount xsi:type="xsd:int">5</estimatedTotalResultCount>
</return>
</namesp1:doSearchResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 5 Sample SOAP response

3.4. Indexing and retrieval performance

Although the accurate time spent for indexing the collection was not recorded, the extraction of visible texts from HTML documents appeared to be completed over night (i.e. 12 to 18 hours) for each machine, and the indexing of the texts was estimated to take roughly two days to complete.

The retrieval performance in terms of speed is given for a set of queries in Table 2. The test was carried out using the SOAP API and the entire index of the collection was used, and the top 100 documents were retrieved along with the title, URL, and size.

Query	**Time (sec)**
Hotels in Paris	4
Pubs in Cardiff	4
Student accommodation in Sheffield	13

Table 2. Retrieval performance (Speed)

The current plan to reduce the retrieval speed includes:

- An equalization of collection size in each machine;
- An optimization of word matching process;
- Use of spatial indexing.

4. Integration with other search components

The searching system presented in this report is planned to be integrated with other components that have been developed in the project. This section shows the data format that will be used in the transaction between two closely related components, namely, spatial indexing and relevance ranking. The details of these two components are given in Deliverable D12 2201 and D14 5201, respectively.

4.1. Overview

The information retrieval process in the project can be summarised in the following steps.

1. A user sends a query to the interface component;
2. A geographical term in the query will then disambiguated by the ontology and the footprint is given for the query;
3. The disambiguated query will be sent to the search engine component (which consists of three sub components such as the core searching system presented in this report, spatial indexing, and relevance ranking);
4. Given the query, the spatial indexing component will then determine which index to search based on the query footprint and carry out the retrieval;
5. The set of documents retrieved by the spatial indexing component will be given the footprints based on the geographical terms that appear in every document;
6. The query and documents (with footprints) will then be sent to the relevance ranking component which rerank the documents based on the BM25 scores and the distance between the query footprint and document footprints;
7. The result of the relevance ranking component will be returned to the search engine component;
8. The search engine component will extract the document information such as the title, URL, size, or snippets, and send the final results to the interface component.

4.2. Search engine component to Spatial indexing component

This is the data format used as an input to the spatial indexing component:

```
t  Integer:RANKINGTYPE
c  String:CONNECTOR
q  String:QUERY
n  Integer:NO_OF_FOOTPRINT
f  String:UID Float:X1 Float:Y1 .. Float:Xn Float:Yn
```

The coordinate in the F line will be repeated for the number of points used in the footprint (i.e. 1 for the central point, 2 for the minimal bounding box, and many for the polygon). This is true for the rest of formats.

4.3. Spatial indexing component to Relevance ranking component

The following is the data format used as an input to the relevance ranking component (i.e. the output of spatial indexing component):

```
t  Integer:RANKINGTYPE
```

```
c  String:CONNECTOR
q  String:QUERY
n  Integer:NO_OF_FP
f  String:UID Float:X1 Float:Y1 .. Float:Xn Float:Yn
n Integer:NO_OF_DOCS
r  String:EXTDOCID Double:INTDOCID Float:BM25SCORE String:RELINFO
Integer:RANK Integer:NO_OF_FP (Note: this is one line from r)
f  String:UID Float:X1 Float:Y1 .. Float:Xn Float:Yn
(repeat the last two lines for the number of docs)
```

4.4. Relevance ranking component to Search engine component

And, finally the following is the output of the relevance ranking component which is returned to the search engine component:

```
n Integer:NO_OF_DOCS
r  String:EXTDOCID Double:INTDOCID Float:BM25SCORE String:RELINFO
Integer:RANK  Integer:NO_OF_FP  Float:RR_SCORE  (Note:  this  is  one
line from r)
f  String:UID Float:X1 Float:Y1 .. Float:Xn Float:Yn
(repeat the last two lines for the number of docs)
```

The sample output is as follows (the figures are not accurate):

```
n  2
r  SPRT-002-031-016-0007557  1937314  480  UNPRCSSD  1  2  100
f  UK315 53.3833  -1.5000  53.3833  -1.5000
f  UK315 53.3833  -1.5000  53.3833  -1.5000
r  SPRT-002-031-016-0007558  1937315  470  UNPRCSSD  2  1  99
f  UK315 53.3833  -1.5000  53.3833  -1.5000
```

5. References

Callan, J.P., Croft, B.W. & Harding, S.M. (1992). "The INQUERY Retrieval System". *Proceedings of the Third International Conference on Database and Expert Systems Applications*, Valencia, Spain. pp. 78-83. Springer-Verlag, Valencia, Spain.

CiQuest Project (2003). http://ciquest.shef.ac.uk/.

Clarke, C. L. A., Cormack, G. V., Laszlo, M., Lynam, T. R., and Terra, E. L. (2002) “The impact of Corpus Size on Question Answering Performance”. In: Proceedings of the 25th Annual ACM Conference on Research and Development in Information Retrieval, 369-370, Tampere, Finland: ACM.

Eurovision Project (2003). http://ir.shef.ac.uk/eurovision/index.html.

Grefenstette, G. & Tapanainen, P. (1994). "What is a word, What is a sentence? Problems of Tokenisation". In: Kiefer, F.K., Kiss, G. & Pajzs, J. (eds.), *Proceedings of the 3rd Conference on Computational Lexicography and Text Research (COMPLEX'94)*, Budapest, Hungary. pp. 79-87. Hungarian Academy of Sciences, Budapest, Hungary.

MIND project (2003). http://mind.cis.strath.ac.uk/.

Porter, M.F. (1980). "An algorithm for suffix stripping". *Program*, **14**, 130-137.

Robertson, S.E., Walker, S. & Beaulieu, M. (1997). "Laboratory Experiments with Okapi: Participation in the TREC Programme". *Journal of Documentation*, **53** (1), 20-34.

Robertson, S.E., Walker, S. & Beaulieu, M. (1998). "Okapi at TREC-7: automatic ad hoc, filtering VLC and interactive track". In: Voorheer, E.M. & Harman, D.K. (eds.), *NIST Special Publication 500-242: The Seventh Text REtrieval Conference (TREC-7)*, Gaithersburg, MD. pp. 253--264. NIST, Gaithersburg, MD.

Salton, G. & McGill, J. (1983). *Introduction to Modern Information Retrieval.* McGraw-Hill.

6. Appendix I. SPRT-009-060-045-0022108.sgml

```
 <DOC>
 <DOCNO>SPRT-009-060-045-0022108</DOCNO>
 <DOCURL>http://www.britcoun.org.uk/eis/index.htm</DOCURL>
 <DOCHDR>
 http://www.britcoun.org.uk/eis/index.htm      1670
 HTTP/1.1 200 OK
 Server: Microsoft-IIS/4.0
 Date: Mon, 08 Oct 2001 17:33:48 GMT
 Content-Type: text/html
 Accept-Ranges: bytes
 Last-Modified: Wed, 25 Jul 2001 11:00:34 GMT
 ETag: "0f507f914c11:12bfc"
 Content-Length: 1444
 </DOCHDR>

 <html>

 <head>
 <link REV="made" HREF="mailto:webmaster@<span class="hl0>british</span><span
class="hl1">council</span>.org">
 <!--Filename: index.htm -->

 <!--Page added PR 31.05.96 -->
 <!--Updated 18.07.96 GCT 16/7/97 -->
 <!--Update by -->
 <!--Delete by -->
 <title><span class="hl0">British</span> <span class="hl1">Council</span>  Education
Information Service</title>
 <!-- Following Stylesheet link is added by SPIRIT -->
 <link rel="stylesheet" href="/search/showit.css" type="text/css">
 </head>

 <body BGCOLOR="#FFFFFF" TEXT="#000000" LINK="#0000FF" VLINK="#400040"  ALINK="#FF0000">

 <p><table BORDER="0"><tr>

 <td NOWRAP><img WIDTH="327" HEIGHT="75" BORDER="0" SRC="./graphics/hdlbceis.gif"
ALT="Education Information Service"><img WIDTH="248" HEIGHT="75" BORDER="0"  ALT="Welcome"
SRC="./graphics/hdrwelcm.gif"></td>
 </tr></table></p>

 <center>

 <p><img WIDTH="575" HEIGHT="241" BORDER="0" ALT="Welcome to the <span
class="hl0>British</span> <span class="hl1">Council</span>'s Education   Information
Service." SRC="./graphics/gdewelcm.gif"></p>

 <p><table BORDER="0"><tr>
 <td NOWRAP><a HREF="./../index.htm"><img WIDTH="173" HEIGHT="26" BORDER="0"  ALT="<span
class="hl0>British</span> <span class="hl1">Council</span> Home Page"
SRC="./graphics/sgnbchm.gif"></a><a HREF="./campus.htm"><img WIDTH="173"  HEIGHT="26"
BORDER="0" ALT="To the Virtual Campus"  SRC="./graphics/sgnvcamp.gif"></a></td>

 </tr></table></p>

 <font SIZE="1"><p>Produced in Britain by the <span class="hl0">British</span>  <span
class="hl1">Council</span><br>
 &copy; The <span class="hl0">British</span> <span class="hl1">Council</span>  2000</p>
 <p>The <span class="hl0">British</span> <span class="hl1">Council</span>, registered in
England as a charity no. 209131, is the United Kingdom's<br>

 international network for education, culture and development services.</p></font>

 </center>
 </body>
 </html>
 </DOC>
```

Appendix II. Stopword list

a	becoming	empty	hereby	most	own	their	via
about	been	enough	herein	mostly	part	them	was
above	before	etc	hereupon	move	per	themselves	we
across	beforehand	even	hers	much	perhaps	then	well
after	behind	ever	herself	must	please	thence	were
afterwards	being	every	him	my	put	there	what
again	below	everyone	himself	myself	rather	thereafter	whatever
against	beside	everything	his	name	re	thereby	when
all	besides	everywhere	how	namely	same	therefore	whence
almost	between	except	however	neither	see	therein	whenever
alone	beyond	few	hundred	never	seem	thereupon	where
along	bill	fifteen	i	nevertheless	seemed	these	whereafter
already	both	fify	ie	next	seeming	they	whereas
also	bottom	fill	if	nine	seems	thick	whereby
although	but	find	in	no	serious	thin	wherein
always	by	fire	inc	nobody	several	third	whereupon
am	call	first	indeed	none	she	this	wherever
among	can	five	interest	noone	should	those	whether
amongst	cannot	for	into	nor	show	though	which
amoungst	cant	former	is	not	side	three	while
amount	co	formerly	it	nothing	since	through	whither
an	con	forty	its	now	sincere	throughout	who
and	could	found	itself	nowhere	six	thru	whoever
another	couldnt	four	keep	of	sixty	thus	whole
any	cry	from	last	off	so	to	whom
anyhow	de	front	latter	often	some	together	whose
anyone	describe	full	latterly	on	somehow	too	why
anything	detail	further	least	once	someone	top	will
anyway	do	get	less	one	something	toward	with
anywhere	done	give	ltd	only	sometime	towards	within
are	down	go	made	onto	sometimes	twelve	without
around	due	had	many	or	somewhere	twenty	would
as	during	has	may	other	still	two	yet
at	each	hasnt	me	others	such	un	you
back	eg	have	meanwhile	otherwise	system	under	your
be	eight	he	might	our	take	until	yours
became	either	hence	mill	ours	ten	up	yourself
because	eleven	her	mine	ourselves	than	upon	yourselves
become	else	here	more	out	that	us	
becomes	elsewhere	hereafter	moreover	over	the	very	

Appendix III. Statistics of SPIRIT Collection

This section contains some statistics obtained from the documents in the SPIRIT collection.

Distribution of domains

Table 3 compares the distribution of top domains in SPIRIT collection with the data provided by ISC (REF) in July 2001 (169 million pages), which is a similar period to SPIRIT data, and January 2003 (238 million pages), which is the latest one at the time of writing this report.

21 European domains were found in the top 50 domains. The European domains consist of over 9.6 million web pages which is approximately 10.24% of the whole collection.

Table 3 Distribution of top domains (top 20) compared with the data from ISC in Jul 2001 and Jan 2003 (Source: Internet Software Consortium (http://www.isc.org/))

SPIRIT Mid-01		ISC Jul 01		ISC Jan 03	
com	42.29%	com	36.12%	com	35.34%
edu	17.05%	net	24.76%	net	27.09%
org	10.17%	edu	4.38%	jp	3.94%
net	4.82%	jp	3.54%	edu	3.23%
us	3.05%	ca	1.87%	it	1.64%
uk	2.75%	uk	1.68%	ca	1.64%
gov	2.42%	de	1.46%	de	1.43%
de	2.35%	us	1.32%	au	1.26%
jp	2.15%	it	1.21%	uk	1.10%
ca	2.00%	mil	1.21%	nl	1.34%
au	1.30%	au	1.13%	br	1.04%
fr	0.78%	nl	1.07%	tw	0.97%
it	0.60%	org	0.86%	mil	0.94%
nl	0.47%	fr	0.84%	fr	0.86%
ru	0.46%	tw	0.77%	us	0.97%
se	0.43%	br	0.63%	es	0.76%
ch	0.42%	se	0.62%	se	0.72%
es	0.33%	es	0.56%	org	0.52%
mil	0.32%	gov	0.52%	dk	0.49%
cn	0.32%	fi	0.52%	fi	0.48%

Table 4 Distribution of second domains (top 20) compared with the data from ISC in Jul 2001 and Jan 2003 (Source: Internet Software Consortium (http://www.isc.org/))

SPIRIT Mid-01		ISC Jul 01		ISC Jan 03	
co.uk	1.44%	lucent.com	3.20%	ne.jp	2.46%
ac.uk	0.95%	aol.com	2.65%	uu.net	2.10%
co.jp	0.70%	uu.net	2.61%	rr.com	1.88%
edu.au	0.47%	ne.jp	1.97%	aol.com	1.80%
rootsweb.com	0.45%	home.com	1.86%	attbi.com	1.47%
ne.jp	0.44%	rr.com	1.36%	nortelnetworks.com	1.27%
com.au	0.43%	pacbell.net	0.84%	t-dialin.net	1.09%
amazon.com	0.42%	t-dialin.net	0.81%	pacbell.net	1.08%
yahoo.com	0.40%	avaya.com	0.80%	qwest.net	1.08%
lycos.com	0.38%	qwest.net	0.73%	comcast.net	0.95%

ac.jp	0.37%	co.uk	0.68%	hinet.net	0.74%
or.jp	0.36%	genuity2.net	0.64%	cox.net	0.73%
sun.com	0.32%	dialsprint.net	0.60%	rima-tde.net	0.70%
geocities.com	0.32%	hinet.net	0.60%	interbusiness.it	0.69%
tripod.com	0.32%	ac.uk	0.58%	net.tw	0.68%
nasa.gov	0.27%	att.net	0.53%	dialsprint.net	0.60%
amazon.de	0.27%	splitrock.net	0.51%	co.uk	0.59%
ca.us	0.25%	net.tw	0.48%	ad.jp	0.53%
aol.com	0.24%	ad.jp	0.45%	genuity2.net	0.51%
uiuc.edu	0.22%	com.br	0.45%	optonline.net	0.51%

Document Size

The distribution of documents both in byte and character is given in Table 3 and 4. As can be seen, most documents are distributed between 5K and 50K in byte, and 200 and 1000 in words.

Table 5 Distribution of document size in byte

Size (byte)		Size (byte)	
10	2.7309%	10,000	19.3494%
20	0.0038%	20,000	18.9109%
50	0.0189%	50,000	14.5617%
100	0.1532%	100,000	2.2913%
200	1.4795%	200,000	0.4989%
500	9.6051%	500,000	0.1751%
1,000	4.1247%	1,000,000	0.0389%
2,000	7.3258%	2,000,000	0.0166%
5,000	18.7087%	5,000,000	0.0065%

Table 6 Distribution of document size in word

Visible words		Visible words	
10<	0.0000%	5,000	3.4789%
10	6.6160%	10,000	0.9379%
20	8.4310%	20,000	0.3178%
50	9.1817%	50,000	0.1221%
100	9.3134%	100,000	0.0254%
200	16.0338%	200,000	0.0104%
500	25.1390%	500,000	0.0030%
1,000	13.7170%	1,000,000	0.0003%
2,000	6.6723%	>1,000,000	0.0000%

URL Length

A length of URLs is the number of characters without URIs (e.g. http://). For example, the length of x.com is 5, www.google.com is 14, and dis.shef.ac.uk/mark/cv/publications/papers/ is 43.

Length (char)		Length (char)	
5<	0.0000%	55	8.9599%
5	0.0002%	60	6.4870%

10	0.0732%	65	4.4197%
15	0.9477%	70	2.8489%
20	2.1855%	75	1.9178%
25	4.2286%	80	1.3666%
30	8.3106%	85	1.0131%
35	12.6004%	90	0.6561%
40	15.1254%	95	0.4518%
45	14.5864%	100	0.3209%
50	12.0536%	>100	1.4463%

Appendix IV. Glass manual

The GLASS Information Retrieval System (DRAFT version:1.0)

Paul Clough, Hideo Joho and Mark Sanderson

Department of Information Studies
University of Sheffield
United Kingdom

Abstract

GLASS is a simple Information Retrieval (IR) system which provides a number of useful utilities for indexing, searching and evaluation. The system can be compiled for any UNIX or Linux configuration and consists of C programs, Shell scripts and Perl programs. The C programs provide the core functionality of GLASS and can be used to form a pipeline through which data can be passed and processes carried out. This provides a modular and flexible approach to building an IR system, enabling seemless integration of additional functionality.

1 Introduction

Salton and McGill ([4]) describe an Information Retrieval system as: "a system used to store items or information that need to be processed, searched, retrieved and disseminated to various user populations." Glass[1] provides the necessary modules to build such a system and handle the processing involved with indexing and retrieving information in a textual format. Several C programs provide core functionality including basic index and search routines. The motivation behind Glass is not one of providing a packaged IR system (e.g. like Inquery or Okapi), but rather to provide a modular set of components that programmers and researchers alike can combine to create an IR environment. This enables users to perform controlled experiments and create prototypical IR systems.

Glass runs from the command-line and does not come with a user-interface. However, researchers at Sheffield University have developed a Web-based interface using CGI scripts based on the command-line programs. Researchers

[1] Yet another information retrieval system designed to be as transparent as glass.

are also developing a Simple Object Access Protocol (SOAP)[2] interface to Glass which enables retrieval services to be called from a remote machine to that on which Glass is loaded. The flexibility of the system can be seen through projects which have adopted Glass as their information retrieval system, including CiQuest, MIND and EuroVision.

Documents are stored in an inverted index and user information needs are expressed in the form of queries to retrieve documents from the store. Queries are matched to documents using a similarity measure based upon term co-occurrence. Any document containing at least one or more terms from the query is retrieved from the index and a similarity score computed for that document:query pair. Documents containing any number of query terms are retrieved (creating an OR'ing effect) to create the relevance set. Within this, documents are ranked in descending order of similarity under the assumption that those nearer the top of the ranked list are more relevant to the user than those nearer the bottom.

The similarity measure used in Glass is the BM25 weighting operator which can also be found in the Okapi probabilistic IR system (see, e.g. [3]). The BM25 function estimates term frequency as Poisson in distribution, and takes into account inverse document frequency and document length. BM25 is based on three sources of weighting which have empirically shown to be useful for different retrieval tasks:

(1) **Collection frequency** - terms occurring in only a few documents are likely to more useful than terms appearing in many.
(2) **Term frequency** - the more frequent a term appears in a document the more important it is likely to be for that document.
(3) **Document length** - a term occurring the same number of times in a short document than a longer one is likely to be more important in the shorter one.

All three of these components are present in the BM25 weighting function:

$$\sum_{T \in Q} w^{(1)} \frac{(k_1 + 1)tf}{K + tf} \frac{(k_3 + 1)qtf}{k_3 + qtf} + k_2 . \mid Q \mid . \frac{avdl - dl}{avdl + dl} \tag{1}$$

where:

[2] SOAP: a protocol specification for invoking methods on servers, services, components and objects and uses XML and HTTP as the method invocation mechanism: http://www.develop.com/soap/

Q is a query contain terms T

$w^{(1)}$ is the Robertson/Spark Jones weight of T in Q

K is $k_1((1-b)+b.dl/avdl)$

k_1, b, k_2, k_3 and k_4 are parameters which depend upon the nature of queries and possibly the database. We use the parameters from TREC-7 experiments [3]: $k_1 = 1.2, b = 0.75, k_2 = 0$ and $k_3 = 0...1000$

tf is the frequency of occurrence of term within a specific document

qtf is the frequency of the term within the topic from which Q was derived

dl and avdl are the document length and average document length (arbitrary units) respectively

The last component of the BM25 operator is added when longer queries are used (called the Query Freqency, or QF). In TREC experiments with Okapi, it is common to find $k_2 = 0$ (as used in [3]), enabling Equation 1 to be re-written as:

$$\sum_{T \in Q} w^{(1)} \frac{(k_1+1)tf}{K+tf} \frac{(k_3+1)qtf}{k_3+qtf} \tag{2}$$

In both Equations 1 and 2, the weighting function $w^{(1)}$ (the Inverse Collection Frequency or ICF) reduces to the following with no relevance feedback (when R=r=0):

$$w^{(1)} = log \frac{N-n+0.5}{n+0.5} \tag{3}$$

where:

N = number of documents in the collection

n = the number of documents containing the term

Within the probabilistic retrieval model, this weighting is used to predict which documents are probably relevant to a user query. Two major advantages of Glass over other systems are: 1) its simplicity and 2) its modularity which make this a useful tool for experimenting with, and teaching the rudimentaries of infomation retrieval. Being an experimental platform for IR, Glass also includes evaluation modules that allow comparison with existing experiments on the TREC collections, and modules exist to ease the conversion between custom and TREC-specific data formats.

2 Installation

Given the Glass installation, two things must be done before Glass is run:

(1) Insert the lines: `setenv GLASS_PRIVATE private dir` and `source ~/GLASS/GLASS_vars` in your `.cshrc` file, and
(2) In the file GLASS_vars, set $GLASS_ROOT to the directory location Glass is in.

Finally make the files with the commands `GLASS_make clean` and `GLASS_make all`. The `bin/i686/` directory contains executables which perform basic IR functions such as word tokenisation, word stemming, stopword removal and indexing. Another directoy which contains useful Shell scripts is the `sh_scr` directory. These scripts use the executable programs to create more high-level tasks such as indexing, retrieval and relevance feedback.

3 Basic GLASS utilities

Glass contains a large collection of program executables and Shell scripts, given in Appendices A and B respectively. However, out of these there are four scripts which perform the most used functions:

(1) **index**: this script takes a document collection in TREC format (after tokenisation) and creates a Glass inverted index.
(2) **retrieve**: this script takes a set of query words (given by a Glass tokeniser) and searches the Glass index for relevant documents. The script returns a ranked list of results sorted in descending order of query:document similarity.

Further functions exist to provide relevance feedback, evaluate the results and convert between TREC and non-TREC data formats. These and other functions are explained more fully and with examples in Section 4.

4 Using Glass

4.1 Tokenisation

An important stage of both the indexing and searching of documents is word tokenisation. The isolation of word-type units from, or tokenisation of, an input character stream is a problem common to many language engineering areas. Typically tokenisation comes before more linguistically-intensive processes such as morphological analysis, but is itself a non-trivial task as one must decide what constitutes a word from potentially large volumes of running text [1], e.g. should "it's" be treated as two words or one? The simplest approach is to tokenise on whitespace and assume that words are separated by whitespace. Using Perl regular expressions, one could split a sentence into words using the following code:

```
@words = split(/s+/, $sentence);
```

Glass provides a number of tokenisers for different input formats, e.g. TREC and UTF-8, and these can be combined with other programs, such as word stemmers and case conversion routines. Glass tokenisers read from standard input. To tokenise a character stream into words, the following command can be used:

```
%> echo "Isn't it ashame." | toke_simple_words a
```

The program `toke_simple_words` takes the standard input and tokenises the text to produce a list of words of block type 'a' in the following form.

```
a Isn't
a it
a ashame.
```

Note, the program does not remove punctuation and keeps the original case. The command can be combined with other programs to perform functions such as stemming and case conversion. For example

```
 %> echo "Isn't it ashame" | toke_simple_words a
        | word_lower a | word_stem_porter a
```

produces

```
a isn't
a it
a asham
```

4.2 Indexing a custom collection

Glass assumes the collection to be indexed comes in the same format as the TREC collections of data. A set of documents must first be gathered into one text file, which we call a *data file*. Documents within this single file are separated from each other using textual annotations and assigning them a unique identifier. Annotations come in the form of `<BEGIN>` and `</END>` tags using the SGML markup language. Using annotation allows the specification of sub-elements within documents such as title, author and summary, as well as the main body of text. Documents are divided using the `<DOC>` tags and must be given a unique identification code as captured with the <DOCNO> tag. Before the main body of text which is demarcated using the <TEXT> tag, a headline must be given using the <HEADLINE> tag. For example in the EuroVision collection, an entry in the data file looks like the following.

```
<DOC>
<DOCNO>2099/2099_0.txt</DOCNO>
<HEADLINE>Euphyia Inctuata (white banded moth).</HEADLINE>
<TEXT>
<RECORD_NO>RMA-H.0100606</RECORD_NO>
White banded moth.
Winged insect on white paper with ruler below to indicate size.
14 September 1953
Robert Moyes Adam.
Inverness-shire, Scotland
RMA-H10606
<IMG>2099/2099_0.jpg</IMG>
</TEXT>
</DOC>
```

Once the document collection has been transformed, it can be indexed. Given a data file called `foo.dat`, the collection is indexed using the following command (which one might typically redirect to `/dev/null`):

```
%> toke_parsed_generic_TREC foo.dat | index PRE -d
```

The index command has a few options (which can be seen by typing index at the command prompt) and PRE is a prefix for the index files which can be the name of the collection. The option "-d" uses the default setting of converting characters to lowercase, removing stopwords and removing word suffixes using the Porter stemming algorithm [2]. The index command creates an inverted index of the text contained in the document body, and a separate index is created for the text contained in the headline (called PRE_head). Given a PRE name, e.g. test, the following files are generated under the current directory (not including the headline index which provides the same information):

- **test.dii**: this file contains a summary of the data file, e.g. number of lines indexed.
- **test.dol**: this file contains each term and lists all documents containing that term and term frequency.
- **test.ti**: this file contains the term index, a list of terms with term and document frequencies.
- **test.dsi**: this contains information about each document in the collection, e.g. average document length, document length and byte offset in the data file.

4.3 Indexing the TREC collection

Indexing the whole or part of the TREC collection is slightly easier than a custom one because the data is already in a format suitable for Glass. To create the index, the TREC files have to first be opened, and the contents tokenised. A program exists to do just this called `file_output` which will apply a tokeniser to a given set of files. For example, to use the generic TREC tokeniser (`toke_parsed_generic_TREC` on the files `file1`, `file2` and `file3`, and then index the data, the following command can be used:

```
%> file|ouindex PRE-parsed_generic_TREC file1 file2 file3
```

Wildcards can be used to index a directory containing files, but we have found that problems can arise if the number of files is large (e.g. all files in the TREC collection). Therefore, given three directories to index, `dir1`, `dir2` and `dir3`, we recommend the command is issued in the following way:

```
%> ( fi|fileoutputPREtokeparsed_generic_TREC dir3/*.txt)
```

4.4 Searching the index

Once the index has been created, it can be searched using the `retrieve` Shell script. This expects the name of the index (PRE), some parameters (can use default "-d"), and a query from the standard input. Core modules in Glass use a specific data format when passing results between them. This format must be adhered to by custom scripts and programs, but to ease this the Shell scripts provide this wrapper. For example, the `retrieve` script expects the input to have come from a tokenisation program first, therefore to query a collection using the retrieve script, the following command could be used:

```
%> ec|retrieve_1-sPRE2-word_3' | toke_simple_words b
```

The tokenised output gets passed to the retrieve script which ouputs a list of documents matching the query, ranked in descending order of similarity. For example if we issue the query "ipswich" to the METER index, we obtain the following results:

```
Indexing in a one-er, no blocking.
Gathered term info.
r times-03042000-8 75  165 UNPRCSSD 1
r pa-03042000-38 72  154 UNPRCSSD 2
r independent-03042000-6 78   151 UNPRCSSD 3
r guardian-03042000-7 77   150 UNPRCSSD 4
r telegraph-03042000-6 76   144 UNPRCSSD 5
r mail-03042000-4 74   129 UNPRCSSD 6
r mirror-03042000-3 73 109 UNPRCSSD 7
```

Sometimes, one might want to use the TREC example information requests (called *topics*) to search the index, enabling standard TREC evaluations to be performed on the results. TREC topics contain three fields: title (very short), description (short) and narrative (long), which can be used individually or collectively to form a retrieval request. The Glass distribution contains most TREC topics (and relevance judgements) in the `test_collections/trec/topics` directory, which can be used to compare a Glass configuration against existing TREC results. For an explanation of the topic fields, see a TREC overview paper, e.g. [6]. Glass provides a parser/tokeniser called `toke_parsed_ TREC.qry` to extract text from a set of given topics, and its output can be fed straight into the `retrieve` program to search the index.

For example, to extract only the *title* field from topics 1 to 50, the following command can be used:

```
%> zcat topics.1-50.gz | toke_parsed_TREC.qry -t
```

which produces an output like this:

```
a 001 -1 0 -1
b International
b Economics
b Antitrust
b Cases
b Pending
b Document
b discusses
...
b a
b merger
h 001 -1 1540 -1
a 002 -1 1541 -1
b International
b Economics
...
Valid queries 50
Invalid queries 0
```

Without specifying certain parameters to retrieve, Glass normally expects a single query. However, in this case we are using 50 queries from which we want to perform 50 searches (or *runs*), enabling evaluation of the 50 runs against the output from other systems. The `retrieve` script can perform multiple searches by giving it the '-s' parameter. For example to retrieve documents from the index defined by `PRE` using default settings, the following command is used:

```
%> zcat topics.1-50.gz | toke_parsed_TREC.qry -t
   | retrieve -s PRE -d
```

which produces an output like this:

```
a 001 -1 0 -1
r CR93H-1598 281987 13472 UNPRCSSD 1
r FR941019-2-00116 254314 13175 UNPRCSSD 2
r FR940304-2-00054 218673 12699 UNPRCSSD 3
...
r FT942-12858 167131 4088 UNPRCSSD 1000
a 002 -1 1541 -1
r LA110789-0095 535507 25334 UNPRCSSD 1
r FR941227-2-00128 264966 24354 UNPRCSSD 2
```

The results from this command can be compared with TREC relevance judgements using the `eval_evl_from_rel` script as explained in Section 4.7.

4.6 Relevance feedback

Glass includes a script for relevance feedback, thereby enabling the initial query to be improved through query expansion and term reweighting. The idea of query expansion is to increase the number of query terms beyond it's initial set by adding new terms from relevant documents and most likely improving retrieval because of increased term co-occurrances between the query and documents in the collection documents. Approaches include feedback from the user, using information derived from the set of documents initially retrieved (also called *psuedo relevance feedback*) and information derived from the whole document collection.

The modification of term weights based on the user relevance judgements, or term reweighting, can also be used to improve retrieval and most systems in use today enable information about relevant documents to be taken into account in the retrieval process.

4.7 Evaluating GLASS using trec_eval

Using TREC data as the test collection, and TREC topics as retrieval tasks for a user-defined system enables the evaluation of retrieval success as determined by TREC relevance judgements. In the Glass home directory (e.g. /usr/local/GLASS), the relevance judgements can be found under `test/collections/trec/qrels`. Relevance judgements can be found for most of the TREC retrieval tasks (called *tracks*) including the adhoc and filtering tasks. Files exist for each query and contain all documents judged to be relevant for that task using a NIST-defined process (see, e.g. [6] for more details).

There are four steps involved in evaluating the results from a Glass configuration or a TREC results file. Although the procedure is broken into four separate stages, it is possible to pipe all the commands together to create a single command.

4.7.1 Step 1: Creating the results file

If the results from a previous TREC retrieval are being used for evaluation, this step is unnecessary. If the results file does not yet exist, the first step is to create one from the TREC collection and topics. For example, to perform the adhoc retrieval task in TREC6, disc4 and disc5 of the TREC collection must be indexed, and topics 201 to 250 used for retrieval. The results file can be generated using the following command:

```
%> zcat topics.201-250.gz | toke_parsed_TREC.qry -t
   | retrieve -s PRE -d  >results
```

If using a results file from a system submitted to TREC, this must first be passed through the `trec2glass.pl` Perl script to transform the TREC format into a form that Glass can parse:

```
%> zcat adhoc/input.AntHoc01.gz | perl trec2glass.pl > results
```

TREC results are limited to the top 1000 documents in the ranked list.

4.7.2 Step 2: Tagging relevant documents

Given a set of results based upon TREC topics, the next step is to indicate in the results file which of the documents retrieved are relevant. The `eval_tag_rel` program does this and creates a file with the documents in the ranked list marked as either relevant or not relevant. Given a results file in Glass format, documents can be tagged using the following command:

```
%> more results | eval_tag_rel  -s qrels/adhoc > results.rel
```

The "-s" option is used to tell the script to expect results from more than one query. The script expects a directory or file containing the relevance judgements (in the previous example the program is pointed to all topic relevance results). The output from the Shell script will be similar to this:

```
a 301 -1 0 -1
r FBIS3-21961 -1 -1 RELEVANT 1
r FBIS3-19646 -1 -1 RELEVANT 2
...
r FBIS4-15318 -1 -1 NOTREL 1000
a 302 -1 0 -1
r LA043090-0036 -1 -1 RELEVANT 1
r CR93E-5666 -1 -1 RELEVANT 2
...
```

This script has the option "-mr" that is used to include any missing relevant documents at the end of the ranking. Documents can be missing because the curt-off point for relevant documents is 1000 in the ranked list. Documents below this point are assumed to be non-relevant and therefore not included in the TREC results (top 1000 results used).

4.7.3 Step 3: Create the evaluation list

The third stage is to create a list of rank positions for the relevant documents in the results list which is used by the `trec_eval` program to compute the retrieval performance of the system. The `eval_evl_from_rel` script does this task:

```
%> more results.rel | eval_evl_from_rel  > results.evl
```

The resulting file will look similar to the following where the topic identifier is given first, followed by the number of relevant document and their rank positions:

```
301
   120    1  2  4  8  9  11  13  16  30  31
...
302
   77    1  2  3  5  6  8  9
...
```

4.7.4 Step 4: Running `trec_eval`

The final stage is to run the `trec_eval` program which calculates the performance of the retrieval system based upon the rank positions of relevant documents as given in the file generated by the `eval_evl_from_rel` command. The version of `trec_eval` used in Glass is a slightly modified version as supplied by UMASS, which includes a number of tests for statistical significance between the results, e.g. the sign test, t-test and Wilcoxon rank test. The program can be run as follows:

```
%>  trec_eval results.evl
```

The program can be fed multiple results files (enabling comparison between systems) simply by including these as parameters. For example, to compare four results files, the following command can be used:

```
%>  trec_eval results1.evl results2.evl results3.evl results4.evl
```

The default output includes interpolated recall-precision, precision at 1, 5, 10, 15, 20, 30, 100, 200, 500, 1000 documents, and average precision. For example, the output will appear similar to the following for comparison between several systems:

```
 1. city6at.evl
 2. city6at_mr.evl

Queryid (Num):  |     50 |     50
Total number of documents over all queries
    Retrieved:  |  50000 |  50000
    Relevant:   |   2560 |   4611
    Rel_ret:    |   2560 |   2560
Interpolated Recall - Precision
```

```
    at 0.00     | 0.6745 | 0.6745
    at 0.10     | 0.5930 | 0.5428
    at 0.20     | 0.5165 | 0.4586
    at 0.30     | 0.4342 | 0.3705
    at 0.40     | 0.3952 | 0.3300
    at 0.50     | 0.3568 | 0.2894
    at 0.60     | 0.3214 | 0.2481
    at 0.70     | 0.2496 | 0.1755
    at 0.80     | 0.2059 | 0.1297
    at 0.90     | 0.1599 | 0.0885
    at 1.00     | 0.1022 | 0.0400
Average precision (non-interpolated) over all rel docs
                | 0.3499 | 0.2876
Precision:
  At    1 docs: | 0.5400 | 0.5400
  At    5 docs: | 0.4800 | 0.4800
  At   10 docs: | 0.4380 | 0.4380
  At   15 docs: | 0.3933 | 0.3933
  At   20 docs: | 0.3670 | 0.3670
  At   30 docs: | 0.3200 | 0.3200
  At  100 docs: | 0.2196 | 0.2196
  At  200 docs: | 0.1542 | 0.1542
  At  500 docs: | 0.0852 | 0.0852
  At 1000 docs: | 0.0512 | 0.0512
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:      | 0.3548 | 0.3220
```

A Perl program called `TREC_plot_results.pl` can be used to produce a postscript graph from the default output of `trec_eval` using `gnuplot` which can be included in Latex documents (see Figure 1).

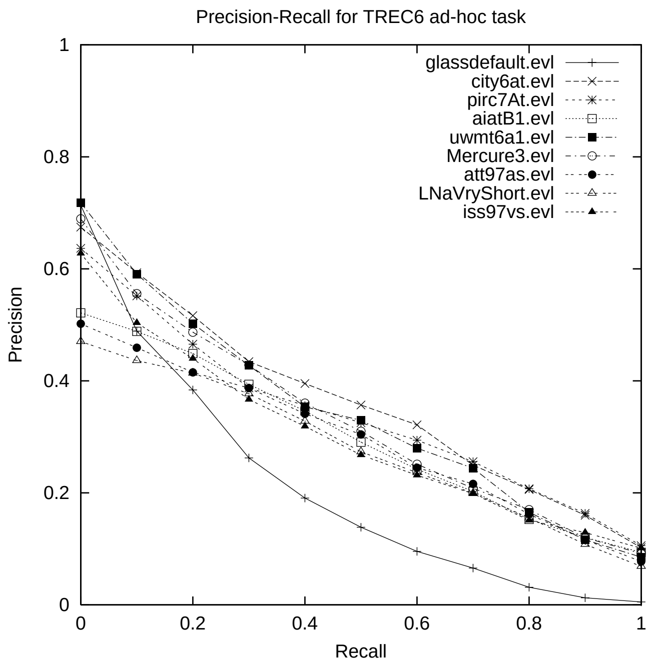

Fig. 1. Example graph created from `trec_eval` results

5 GLASS SOAP API

This section describes the implementation of SOAP API for GLASS system and show some example scripts of SOAP clients to use the GLASS commands. While the SOAP server is implemented using Perl's SOAP::Lite module, the clients can be written by any computer languages such as Perl, Java, .Net, C, C++, etc.

Although the following two sections will describe the overview of SOAP and the GLASS SOAP server, the main focus in this section will be on how to use GLASS via SOAP, that is, how to write a client program. Please refer to the previous sections to understand how GLASS works, and references for further information about SOAP in general.

5.1 Overview of SOAP

Programming Web Services with SOAP [5] is a good introductory book for SOAP.

5.2 GLASS SOAP Clients

5.2.1 Using Perl

An advantage of using Perl to write a SOAP client is its simplicity while you need to know some trick to see the actually XML inputs/outputs if you want.

5.2.1.1 Preparation In order to write a SOAP client in Perl, your system needs to have a perl module called SOAP::Lite installed. You can check if your system has a perl module in the following way.

```
%> perl -e 'use SOAP::Lite'
```

if no error message is shown, then your system has the module. If some error message is shown, then you need to install it. You can install a perl module in Linux/Unix environments (you need to be *root*):

```
%> perl -MCPAN -e 'install SOAP::Lite'
```

You can isntall a perl module in Windows environments using *Command Prompt* program:

```
%> ppm install SOAP::Lite
```

If these approaches don't work for you, then you need to go `http://www.cpan.org` to find the module, download, and install manually. It shouldn't be too hard.

5.2.1.2 Sample clients

The following is a part of one of the SOAP clients written in Perl, and this script (*returnQuery_client.cgi*) returns query terms that will actually be used in GLASS.

```
use SOAP::Lite;

# Query terms
my $query = shift;
# Output option
my $xmloutput = shift;
# Location of SOAP API (CGI)
my $API = 'http://spirit1.shef.ac.uk/glass/GlassSearch.cgi';
# Pointer to resulted hash
my $result;

# Output is xml
if($xmloutput) {
        $result = SOAP::Lite
                ->       uri('urn:GlassSearch')
                ->       proxy($API)
                ->       outputxml(1)
                ->       returnQuery($query);
# Output is string
} else {
        $result = SOAP::Lite
                ->       uri('urn:GlassSearch')
                ->       proxy($API)
                ->       returnQuery($query)
                ->       result;
}

print $result . "\n";
```

This script can be used in the following way.

```
%> perl returnQuery_client.cgi "International Organized Crime" 0
```

The first argument is a set of query terms and the second is to set the output format in string (0) or XML (1). The actual XML strings that are sent to the server can be displayed by changing the line 1:

```
use SOAP::Lite;
```

to

```
use SOAP::Lite +trace => qw(debug);
```

Output of the above example script would be:

```
%> intern organ crime
```

As you can see, the query terms that are actually used in GLASS are lower-cased and stemmed. Stopwords such as *of*, *he*, or *its* would also be removed in this setting. The XML output would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/
soap/encoding/" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/
soap/encoding/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>
    <namesp1:returnQueryResponse xmlns:namesp1="urn:GlassSearch">
      <q xsi:type="xsd:string">
      intern organ crime
      </q>
    </namesp1:returnQueryResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Reasonably formatted simple output of *doGlassSearch.cgi* would be:

```
i       searchQuery     = intern organ crime
i       startIndex      = 0
i       endIndex        = 2
i       estimatedTotalResultCount       = 15155
a       rank    = 1
r       score   = 232
r       internalDocID   = 415602
r       externalDocID   = LA022790-0156
r       title   =
r       snippet =
r       date    =
a       rank    = 2
r       score   = 144
r       internalDocID   = 243757
r       externalDocID   = FT943-16234
r       title   =
r       snippet =
r       date    =
```

References

[1] G. Grefenstette and P. Tapanainen. What is a word, what is a sentence? problems of tokenisation. In *Proceedings of the 3rd Conference on Computational Lexicography and Text Research (COMPLX'94)*, 1994.

[2] M.F. Porter. An algorithm for suffix stripping. *Program*, Vol. 14(3):130–137, 1996.

[3] S. E. Robertson, S. Walker, and M. M. Beaulieu. Okapi at trec-7: automatic ad hoc, filtering vlc and interactive track. In *NIST Special Publication 500-242: The Seventh Text REtrieval Conference (TREC-7)*, pages 253–264, 1998.

[4] G. Salton and J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Computer Science Series, 1983.

[5] J. Snell, D. Tidwell, and P. Kulchenko. *Programming Web Services with SOAP.* O'Reilly., 2002.

[6] E. M. Voorhees and D. Harman. Overview of the sixth text retrieval conference (trec-6). In *Overview of the Sixth Text REtrieval Conference (TREC-6)*, pages 1–25, 1997.

A Glass executables

```
document_ex_to_in
document_prune
document_score_bm25
document_tf_len
eval_evl_from_rel
eval_tag_rel
index_docs
index_docs_get_doc
index_docs_insert_doc_info
index_print_header
index_terms_1
index_terms_2
index_terms_insert_ti
index_terms_merge
index_terms_occs
index_terms_weight_idf
query_rank_terms
toke_find_tags
toke_parsed_TREC.qry
toke_parsed_cran
toke_parsed_generic_TREC
toke_parsed_generic_UTF-8
toke_parsed_generic_dot
toke_parsed_ltt
toke_parsed_npl
toke_parsed_npl.query
toke_parsed_srt
toke_process_distribute
toke_process_filter
toke_simple_per_line
toke_simple_regexp
toke_simple_words
toke_strip
word_clean
word_lower
word_stem_porter
word_stem_wordnet
word_stop_tag
word_stopper
```

B Glass shell and Perl scripts

```
calc_emim
co-occ_terms
eval_average_from_std11
eval_median_from_std11
eval_rel_docs_from_TREC
eval_stnd_interp_from_all_rp
experiment_TREC_generic
f_max
file_output
gawkt
index
index_docs_get_doc_external
index_terms
index_terms_dol_sort
make_ttbl
number_index
percent
```

```
porter_stem_table
relevance_feedback
retr
retrieve
retrieve_set
sortt
srt_prob_scale
srt_prob_strip
toke_process_fold

toke_process_passage
toke_process_sort
toke_process_strip_fpos
toke_process_swap
toke_simple_whole_file
trec2glass.pl
url_ascii_to_form
url_escape_to_ascii
word_process
word_stem_inflectional
```

C Summary of index files

C.1 Overview

GLASS *index* command requires a prefix to index a set of documents, and produces six index files by default. The following is a set of index files listed along with a brief description (prefix is SAMPLE):

A *term* is a word indexed with some pre-processing described later.

C.1.1 Document information

SAMPLE.dii contains a summary of the data file, e.g. number of lines indexed.

SAMPLE.dsi contains the information about each document in the collection, e.g. average document length, document IDs, document length, and byte offset in the data file.

SAMPLE_head.dii is same as SAMPLE.dii but for a header section (if any)

SAMPLE_head.dsi is same as SAMPLE.dsi but for a header section (if any).

C.1.2 Word information

SAMPLE.dol contains each term and lists all documents containing that term and term frequency. This file is so-called an *inverted file.*

SAMPLE.ti contains the term index, a list of terms with term and document frequencies.

C.1.3 Other notes

Information about the data contained in each file will be given in the following sections. Note that the examples being used in the following sections were generated by indexing two files containing 375 and 346 documents, respectively, with the total size of 2MB. Also note that a file contains more than one documents in this setting.

Options for the indexing were lower cased, stopwords removed, and stemmed by the porter stemming.

C.2 .dii

A .dii file looks like Table C.1 and the description of each field of *g* entry (i.e. lines starting with a *g* token) is given in Table C.2.

A GLASS index file often consists of two general parts: *header* part and *body* part. In the case of a .dii file, *e* and *c* entries are the header, and *g* entries are the body.

Note that fields in index files are delimited with a tab character ('\t') unless otherwise mentioned, and this is true for the rest of index files.

There is a sequence of dots at the end of *g* entries in actual index files to keep the length of each line equal to a given size (i.e. 150 bytes), but they are *not* shown in all examples to avoid confusion.

```
e   **No File**
c   150     Byte size of each line
c   2       Number of objects in table
c   0       Number of extra objects in header
g   /home/hideo/writing/glass/indexes/FT911_1    1     375
g   /home/hideo/writing/glass/indexes/FT911_2    376   72
```

Table C.1
.dii file

Field	Description
1	token (g)
2	File location
3	Start of internal Doc ID of a file
4	End of internal Doc ID of a file

Table C.2
Description of the fields in .dii (g entries)

C.3 .dsi

A .dsi file contains the detail information about individual documents. The first *e* entry points a .dii file so that it can seek the beginning of a document effeciently. The main aim of .dsi file is to show the range of document IDs stored in each file.

```
e   SAMPLE.dii
c   150    Byte size of each line
c   721    Number of objects in table
c   1      Number of extra objects in header
c   239    Average document length
g   1    FT911-1    0       1388    129
g   2    FT911-2    1389    1682    138
```

Table C.3
.dsi file

Field	Description
1	token (g)
2	Internal Doc ID
3	External Doc ID
4	Byte offset of the start of a document
5	Length of a document in bytes
6	Length of a document in terms/tokens

Table C.4
Description of the fields in .dsi (g entries)

C.4 _head.dii

A _head.dii is identical to .dii.

```
e   **No File**
c   150     Byte size of each line
c   2       Number of objects in table
c   0       Number of extra objects in header
g   /home/hideo/writing/glass/indexes/FT911_1    1     375
g   /home/hideo/writing/glass/indexes/FT911_2    376   721
```

Table C.5
_head.dii file

Field	Description
1	token (g)
2	File location
3	start of internal Doc ID of a file
4	End of internal Doc ID of a file
5	Record length controller

Table C.6
Description of the fields in _head.dii (g entries)

C.5 _head.dsi

A _head.dsi is also identical to .dsi except the difference in the seeking points. Fouth and fifth fields record the beginning and end of a header section in a document.

```
e   SAMPLE.dii
c   150    Byte size of each line
c   721    Number of objects in table
c   1      Number of extra objects in header
c   239    Average document length
g   1    FT911-1    94      82    129
g   2    FT911-2    1483    91    138
```

Table C.7
_head.dsi file

Field	Description
1	token (g)
2	Internal Doc ID
3	External Doc ID
4	Byte offset of the start of a document
5	Length of a document in bytes
6	Length of a document in terms/tokens

Table C.8
Description of the fields in _head.dsi (g entries)

C.6 .dol

An *a* entry contains only a token and a term. The *a* entry is followed by *b* entries which contain the information about the documents in which the term occurs. For example, a stemmed term *zulu* appeared in two documents and occurred twice in the document 286 and once in 571.

A .dol file also contains the document length in the 4th field for the convenience of GLASS weighting function.

```
a    zubin
b    384    1    191
a    zulu
b    286    2    235
b    571    1    158
```

Table C.9
.dol file

Field	Description
1	token (b)
2	Internal Doc ID in which a term occurs
3	Frequency of occurrence in a document
4	Length of a document in bytes

Table C.10
Description of the fields in .dol (b entries)

C.7 .ti

A .ti file contains the detail information about terms. Like the relationship between .dii and .dol, the first *e* entry points .dol file to tell which file to seek.

The 5th and 6th field show the length of *a* entries (term) and *b* entries (documents), respectively.

```
e   SAMPLE.dol
c   78      Byte size of each line
c   15096   Number of objects in table
c   2       Number of extra objects in header
c   721     Number of documents
c   110382  Total number of term occurrences
g   zubin    1     1444153   8       12    98
g   zulu     2     1444173   7       24    98
```

Table C.11
.ti file

Field	Description
1	token (g)
2	Term
3	Document frequency of the term
4	Byte offset in .dol
5	Length of *a* entry in .dol in bytes
6	Length of *b* entry in .dol in bytes
7	Score based on IDF ranging between 0 and 100

Table C.12
Description of the fields in .ti (g entries)