MGPP:
A Search engine for XML documents
User guide

Katherine Don
Department of Computer Science
University of Waikato
Hamilton
New Zealand
kjdon@cs.waikato.ac.nz

MGPP (or MG++) is a reimplementation of MG, the 'Managing Gigabytes' text compression and indexing system. MG is described in the book with the same name, 'Managing Gigabytes', by Ian Witten, Alistair Moffat and Timothy C. Bell, and is the default indexing tool used in the Greenstone digital library software. MG provides document level indexes, and compression of the source documents. MGPP extends MG to also provide word-level indexes, enabling proximity and fielded searching. A small change to the configuration file for a collection is all that is required to use MGPP . Once the collection is online, Greenstone provides form based search interfaces (accessed through the preferences page) enabling fielded search on any specified metadata.

This documentation describes how to use MGPP , both as part of Greenstone, and as a stand-alone indexer.

1 Where to get MGPP

There is no stand-alone distribution for MGPP . However, it can be obtained in two ways: from a Greenstone distribution, or from CVS.[1]

1. Greenstone distributions contain both MG and MGPP , and provide both the source code and binaries. Greenstone is available from

`http://www.greenstone.org/english/download.html`.

The source code for MGPP is located in `gsdl/src/mgpp` , and the executables will be in `gsdl/bin/linux` or `gsdl/bin/windows` , depending on the distribution. The configure scripts and Makefiles have been adapted for use within Greenstone , and therefore may require some modification if MGPP is to be used on its own.

[1] Concurrent Versions System, `http://www.cvshome.org`

2. Alternatively, the source code for MGPP by itself can be obtained through the Greenstone CVS server in the same way that Greenstone can. The configure scripts and Makefiles are set up for use of MGPP as a stand-alone system.

The CVS package is called mgpp, and can be obtained through a CVS checkout using anonymous access:

```
cvs -d :pserver:cvs_anon@cvs.scms.waikato.ac.nz:2402/usr/local/global-cvs/gsdl-src co mgpp
```

2 Using MGPP in Greenstone

There are two parts to using MGPP within Greenstone : collection building, and collection querying.

2.1 Collection building

Greenstone can build collections using MG or MGPP . The default is MG, but you can use MGPP by editing the collection configuration file (`collect.cfg` , found in the `etc` directory of a collection). Figure 1 shows a sample configuration file, with the changes required to use MGPP shown in the box.

First, add the line

```
buildtype mgpp
```

Second, the way indexes are described is different. MG uses a line like:

```
indexes document:text section:text section:Title
```

This specifies three indexes, a document level index containing the text of each document, and two section level indexes, one containing the text, the other the Title of each section. The document and section parts determine the granularity of the searching and of the items retrieved. The first index returns a list of document numbers while the second and third indexes return section numbers.

MGPP does things differently. By default it builds a word level index, with Document level granularity, i.e. searching and retrieval are done at Document level. Section level can also be specified, enabling searching and retrieval of sections as well as documents. To specify sections, add the following line:

```
levels Section
```

To specify what goes into the index, an indexes line is used, similar to that for MG but without the level information (as this is specified separately by the levels info). For example:

```
indexes text
```

This will index all the text at word level. Metadata fields can also be specified in the index, for example:

```
indexes text,Title,Subject
indexes text,metadata
```

The first one builds one index, with tagged entries for Title and Subject metadata. Unlike levels, metadata names can be anything - though obviously they

creator me@myaddress.com
maintainer me@myaddress.com
public true

	buildtype mgpp
indexes section:text section:Title document:text defaultindex section:text	indexes text,Title defaultindex text,Title levels Section
collectionmeta .section:text “sections” collectionmeta .document:text “entire documents” collectionmeta .section:Title “section titles”	collectionmeta .text,Title “documents”

plugin GAPlug
plugin TEXTplug
plugin HTMLPlug
...
plugin RecPlug

classify AZList -metadata Title
classify AZList -metadata Source

collectionmeta collectionname “my collection”
collectionmeta collectionextra “This is a sample collection.”

Figure 1: A sample collection configuration file, with differences between MG (left) and MGPP (right) shown in the box.

should match the names in your documents.

The second one builds one index with tagged entries for all the metadata it finds—this is useful if you don't know in advance what metadata is available, or want all of it indexed anyway.

After the building has finished, the `build.cfg` file in the `building` directory of the collection has a list of what metadata it has found and indexed, for example

```
indexfields Subject TextOnly Title
indexfieldmap TextOnly->TX Subject->SU Title->TI
```

The metadata names are passed to MGPP during building as two letter codes - indexfieldmap specifies what codes were used. These are the codes that need to be used when carrying out fielded search on the collection.

By default, only the text is compressed, not the metadata. To change this, you can add a line to the configuration file like

```
textcompress text,Title
```

This will add Title metadata to the text that gets passed to the compressor.

2.2 Collection querying

A collection built with MGPP can be searched in the usual way through Greenstone. Search terms can be combined with boolean operators, and phrases are specified using double quotes ("..."). Because it uses a word level index, it has some extended searching capability over MG. If metadata has been specified in the index, fielded search can also be done.

The current query syntax involves the following:

Boolean operators:

`& AND | OR ! NOT`, with () for precedence

Term modifiers:

`#icus /x` —this is stemming, casefolding and weighting

`#i`	case insensitive
`#c`	case sensitive
`#u`	unstemmed
`#s`	stemmed
`/x`	term weight (default = 1)

Proximity searching:

`''..''` —phrase searching, a form of very strict proximity matching where the query terms must be in the exact order specified by the phrase.

`NEARx` — this is used to specify the maximum distance apart (x) two query terms must be for a document to match. `NEAR` by itself defaults to 20.

Fielded searching:

`[terms]:Field` —specifies searching within a particular field or metadata element of the document. The field name needs to be the name of a metadata element in the collection. If the collection was built with Greenstone , metadata is named

with the two letter codes found in the build.cfg file. Multiple terms inside the [] are ANDed together. Boolean operators cannot be used inside the []. However, the term modifiers (`!#icus/x`) can be used.

Some example queries are:

`computer#is`	search for `computer` , stemmed and casefolded
`Bob & Dylan/10`	search for `Bob` and `Dylan` , giving `Dylan` a 10-fold weighting compared to Bob (used in ranking the result list)
`dog NEAR4 cat`	search for cat within 4 words either side of dog
`[Witten]:CR`	search for `Witten` in the `CR` (creator) field
`[snail farming]:TI`	search for `snail` and `farming` in the `TI` (title) field
`[Witten]:CR & [Gigabytes]:TI`	search for `Witten` in `CR` and `Gigabytes` in `TI`

This syntax can be entered into the standard Greenstone search box. However, there are additional query pages using forms. These can be accessed through the preferences page - select form query, then simple/ advanced. These provide fielded searching without the user needing to learn a new query syntax.

3 Using MGPP outside Greenstone: Indexing

The programs for building are named mgpp_*, for example, mgpp_passes, mgpp_invf_dict etc. They have man pages with them, named mgpp_passes.1 etc. mgpp_passes is the main program used for building and its man page gives an example of how to build a collection.

Here is a simple bash script that can be used to build a collection,

```
#! /bin/bash

# The arguments on the command line specify the
#   source of the text
source=$@

# This is the name of the collection
text=demo

# Create *.text.stats, *.invf.dict, *.invf.level
#   *.invf.chunk and *.invf.chunks.trans
cat ${source} | mgpp_passes -T1 -I1 -f ${text}

# Create *.text.dict
mgpp_compression_dict -f ${text}

# Create *.invf.dict.hash
mgpp_perf_hash_build -f ${text}

# Create *.text, *.text.idx, *.text.level
#   *.invf and *.invf.idx
```

```
cat ${source} | mgpp_passes -T2 -I2 -f ${text}

# Create *.text.weight and *.weight.approx
mgpp_weights_build -f ${text}

# Create *.invf.dict.blocked
mgpp_invf_dict -f ${text}

# Create *.invf.dict.blocked.1
mgpp_stem_idx -s 1 -f ${text}

# Create *.invf.dict.blocked.2
mgpp_stem_idx -s 2 -f ${text}

# Create *.invf.dict.blocked.3
mgpp_stem_idx -s 3 -f ${text}
```

This builds a basic collection, using `Document` as the document level tag, with a word level index, and three stemmed indexes (stem, casefold, stem and casefold). Each of the programs used in the compression and indexing process has different options that can be specified. The man pages have details about what each program does, and the options available.

3.1 Document format

The input to MGPP requires a certain format. A sample document looks like the following:

```
<Document>
<Section>
<Title>Snail Farming</Title><Creator>Joe Bloggs</Creator>
Snail farming is a productive and interesting pastime.
</Section>
<Section>
<Title>Chapter 1</Title>
There are many types of snails. Make sure you only farm ones
which are safe to eat.
</Section>
</Document>
```

There must be a document level tag thats starts (and optionally ends) each document. The default is `Document` . Text outside these tags is ignored. To change the document tag that MGPP looks for, add `-J <tagname>` to the mgpp_passes commands. There can only be one `-J` option.

Smaller granularity is added using level tags. These must enclose all the text enclosed by the document tags. They can be specified to mgpp_passes by `-K <level tag>` . In the example above, `Section` would be a suitable level tag. There can be many `-K` options. However, they cannot be overlapping. You cannot have the following:

```
<Section>...<Paragraph>...</Section>...</Paragraph>
```

The smallest level of granularity (default is word level) can be changed by adding `-L <index level>` to mgpp_passes. This can be no larger than the smallest level that has been specified.

Metadata or tagged fields are specified like `<Title>the text of the title </Title>` . These are all automatically indexed and can be searched as fields.

The output files are placed in the directory in which the commands were run. To change where they go, use `-d <directory name>` with all of the commands.

There are four stages to mgpp_passes, specified by T1 and T2 (for text compression) and I1 and I2 (for indexing). T1 and I1 must be done before T2 and I2, respectively. As shown above, T1/I1 and T2/I2 can be run together, reducing the number of passes through the documents to 2. But all output files get placed in the same directory, and the same text is passed to the indexer and the compressor.

In Greenstone , we pass only the text to the compressor, but metadata and text to the indexer, and text files and index files are put in separate directories. In this case, mgpp_passes is run four times, like:

```
cat <text src> | mgpp_passes -d <text_dir> -T1 ...
cat <text src> | mgpp_passes -d <text_dir> -T2 ...
cat <index src> | mgpp_passes -d <index_dir> -I1 ...
cat <index src> | mgpp_passes -d <index_dir> -I2 ...
```

4 Using MGPP outside Greenstone: Querying

Queryer is a stand-alone program enabling search and retrieval of documents in an MGPP collection. It is written in C++, and is the equivalent to mgquery for MG. Even when using Greenstone , it is useful for testing collections. Queryer can be run using the following command

```
Queryer -f <index files> -t <text files>
```

`<index files>` and `<text files>` are the paths to these files, including the collection name. For example, if the text files are demo/index/text/demo.t, demo/index/text/demo.ti etc, then the text path is demo/index/text/demo, and similarly for the index files.

There is also a Java version of the Queryer: org.greenstone.mgpp.Queryer. It uses JNI (Java Native Interface) wrappers to talk to the C++ code. It is run using:

```
java org.greenstone.mgpp.Queryer <basedir> <indexdir> <textdir>
```

The two Queryer programs have a similar set of commands available:

.q	quit
.h	print the help message
.i	change the search level
.l	change the result level
.b	full text browse (enter a word or fragment at the prompt)
.r0/.r1	ranking off/on
.t0/.t1	query type some/all
.c0/.c1	casefolding off/on
.s0/.s1	stemming off/on
.o0/.o1	short output off/on
.p	print a document
query	a text query

Queries can be entered straight at the prompt. The query syntax is that described in Section 2.2.

.i and .l are used for changing the search and retrieval levels, respectively. The search level controls where two or more query terms must occur for a document or section to match, for example they must occur within a Section, or a Document. The retrieval level specifies what granularity of document/section numbers to return. These two levels need not be the same. MGPP can return document numbers where all of the query terms occur within a section. Set the search level to Section, and the retrieval level to Document.

.c and .s are used for setting global casefolding and stemming options. These settings will be applied to all of the query terms. Specifying stem and case for individual terms using the `#icus` syntax overrides these global settings.

.t specifies whether the query is a 'some' or 'all' query, i.e. specifies what the default boolean operator is. A 'some' query looks for documents containing any of the terms, corresponding to boolean OR, an 'all' query looks for documents containing all of the terms, corresponding to boolean AND. This can be overridden by specifying operators in the query. A 'some' query is most useful when the results are ranked.

.r specifies whether to return the results in ranked or natural order. Natural order returns the document numbers in increasing order. Ranked returns document numbers in order of relevance to the query—the more terms that feature in a document, the greater the relevance to the query.

.p prints a document. Document numbers are related to the retrieval level. If the retrieval level is set to Document, then 2 will specify the second document. If the level is set to Section, then 2 will specify the second section.

.b is used for full-text browsing. A word, or word fragment, is entered at the prompt, and the program returns a list of words from the index that occur next to the entered word or its closest match.

.o is used to change the output from a more descriptive result to a shorter one that doesn’t list all the document numbers, just the total number of documents matched.