

CS351 Spring 2005, Project 1

Moogle

MondoSoft, Inc.

STATUS Specification and requirements document

VERSION 1.2

DATE Feb 14, 2005

0 Changelog

Version 1.0 Initial release. Jan 19, 2005.

Version 1.1 Update 1. Jan 26, 2005.

- Packages (Section 4)
- `EntrySet` (Section 4.1)
- `AbstractCollection` (Section 4.1)
- `String.split()` (Section 5)

Version 1.2 Update 2. Feb 14, 2005.

- Corrected dates to 2005. Oops.
- Added Query Trace File requirement (Section 4.3.4)

1 Summary

With the recent boom in success of Google (TM), MondoSoft¹ has decided to get a slice of the search engine technology market. Upper management's position is: "If Microsoft can play that game, what's stopping us?" In order to differentiate their product from everybody else, MondoSoft has proposed to produce a standalone search client rather than providing search over the web. As a prototype effort, you've been assigned to build a general-purpose web spider and a standalone Swing-based search client. In a brilliant stroke of marketing "genius", the marketing department has decided to name this product `Moogle` (micro-Google).² Furthermore, MondoSoft

¹Your employer.

²Upper management continues to be mystified by generally poor sales and low stock price. Go figure.

has recognized the business opportunity to remarket a sub-component of this program as a high-performance standalone software module. Therefore, this project also includes `MondoHashMap`, a fully functional hashtable implementation of the `java.util.Map` interface. To improve marketability and demonstrate these programs' superior performances, both components (the hashtable and the full `MoogLe` suite) will also be accompanied by rigorous scientific empirical performance evaluations³.

³Unfortunately, because MondoSoft's VCs pulled out during the bubble burst, MondoSoft is a bit strapped for cash to cover independent laboratory certification, so you get stuck with this job too.

Contents

0	Changelog	1
1	Summary	1
2	Group/Individual Effort	4
3	Definitions	4
4	Requirements	5
4.1	MondoHashMap	6
4.2	MSpider	7
4.2.1	Reverse Index and Web Database	7
4.2.2	Durable State	8
4.2.3	Support Libraries	8
4.2.4	Spider Options	9
4.3	The MoogLe Client Program	9
4.3.1	User Interface	9
4.3.2	Query Parser	10
4.3.3	Relevance Sort	11
4.3.4	Query Trace File	11
5	Quantitative Requirements	13
6	Deliverables	15
6.1	Milestone 1: MondoHashMap	15
6.2	Milestone 2: MSpider	16
6.3	Rollout: Complete MoogLe Suite	17
7	Due Dates	18
8	Grading Levels	18
A	The TF/IDF Scoring Function	19

2 Group/Individual Effort

This project is an *individual* programming project, though you are welcome (and encouraged) to do design work with your group (or anybody in the class). Specifically:

1. All code must be your own. Copying code from anybody (in the class or elsewhere) is explicitly prohibited and is considered cheating. Similarly, looking up source to Java standard library classes and/or disassembling existing `.class` or `.jar` files is disallowed. All elements of the class “Statement of Policy”, from the class syllabus, are included here by reference.
2. You are allowed (and encouraged) to discuss design with your group or anybody in the class. Your internal documentation **MUST** indicate which parts of your design were developed collaboratively.
3. You are allowed to ask other members of your group or the class to assist with debugging.
4. You are allowed to discuss library functions and library code documentation with other members of the class.

3 Definitions

The following definitions will be used in this document:

AND/OR The QUERY language for the `MoogLe` client. Allows a sequence of WORDs joined by AND and OR conjunctives, as well as parenthetical grouping. See Section 4.3.2 for details.

CRAWL-MAX The maximum number of PAGEs that the spider engine can download in a single run (including restarts). The spider **MUST NOT** crawl more PAGEs than allowed by CRAWL-MAX.

MAY A requirement that the product can choose to implement if desired. Can also indicate a choice among acceptable alternatives (e.g., “The program **MAY** do x, y, or z.” indicates that the choice of behavior x, y, or z is up to the designer.)

MUST A requirement that the product must implement for full credit.

MUST NOT A behavior or assumption that must not be violated. Violating a **MUST NOT** restriction will result in a penalty on the assignment.

PAGE The content document pointed to by a URL. This project **MUST** support “text/html” documents, but the system **MAY** support additional document types, at the designer’s discretion.

RECOVERABLE ERROR An error condition that the software can ignore, correct, or otherwise recover from. The program **MUST** produce a warning message and then cleanly continue with no corruption or loss of valid data.

REVERSE INDEX A mapping that records which PAGEs each WORD occurs in.

SHOULD A requirement that is recommended, but not required. The designer may violate a SHOULD requirement, but should be prepared to explain why.

PUNCTUATION Punctuation characters. For the purposes of this project, the punctuation characters are considered to be any characters other than WHITESPACE, letters, digits, or parentheses.

QUERY A user's request for relevant PAGES, expressed as a sequence of WORDs joined by an AND/OR query language. See Section 4.3.2 for details.

TF/IDF A scoring function that attempts to assess the relevance of a PAGE with respect to a QUERY. See Appendix A for details.

UNRECOVERABLE ERROR An error condition from which recovery is impossible. The program MUST produce an error message describing the condition and then cleanly halt.

URL A pointer to a PAGE, represented in html with a fully qualified universal resource locator. Because each URL maps one-to-one onto a single PAGE, this specification will often use the two interchangeably. Note that to ensure the one-to-one mapping, URLs will have to be canonicalized.

WEB DATABASE The complete database of information necessary for the MoogLe client to do its job. This includes the REVERSE INDEX, but will also include additional information to, for example, support crawl restarts and TF/IDF result sorting.

WHITESPACE Non-printable characters including (but not limited to) space, horizontal and vertical tabs, newlines, and carriage returns. See the Java JDK API call `Character.isWhitespace()`.

WORD The smallest unit of parsing (for the MSpider engine) or querying (for the MoogLe client). In this project, WORDs are considered to be sequences of letters and digits, excluding the two reserved words AND and OR. WORDs SHOULD be treated as case-insensitive (with the exception of AND and OR), but MAY be treated as case-sensitive. Either way, case (in)sensitivity MUST be documented in the user manual.

4 Requirements

This section describes the elements that MUST be developed as part of this project. The designer MAY also choose to implement additional Java source files, programs, and/or shell scripts in support of the following items. This section only describes the general performance requirements for each element; for specific deliverable requirements, please refer to Section 6.

The MoogLe suite comprises two primary programs: MSpider and MoogLe. The MSpider program is responsible for crawling the web to retrieve PAGES, parse them into individual WORDs, and build the underlying web database. Essentially, MSpider caches the web content and precomputes the REVERSE INDEX so that the user interface client can very quickly locate relevant pages without having to make web requests itself. MSpider is also responsible for accumulating the statistics that will be used by the MoogLe UI client program for search and result retrieval. The MSpider program will be discussed in Section 4.2.

The `MoogLe` user interface client program is responsible for providing the user an ergonomic interface to search the cached `REVERSE INDEX` and then present the results to the user, sorted in relevance order. Important components of the `MoogLe` client are the user interface itself, a parser for understanding user queries, and the search/sort mechanism that retrieves and presents the query response. The `MoogLe` client program will be discussed in Section 4.3.

A core component employed by both programs is the `REVERSE INDEX`. The `REVERSE INDEX`, in turn, can be built around a hash map (i.e., hash table) to provide high-performance access and retrieval of `WORD`→`PAGE` relationships. The hash map implementation will be the first deliverable for this project. The hash map will be discussed in Section 4.1.

The designer `MAY` choose any package naming convention (including the default package) for this project. If a root package other than the default is chosen, it `SHOULD` be `unm.cs351.p1`. If the choice of package affects how the programs are invoked, the `README.TXT` document `MUST` specify this.

4.1 `MondoHashMap`

The fundamental unit of search is a single `WORD`, and the critical relationship to track is the binding between a `WORD` and the `PAGE` it occurs in. To track the mapping between `WORDS` and the `PAGES` that they occur in, the `MoogLe` suite will use a hash mapping, specifically, a from-scratch implementation of the `java.util.Map` interface, as documented in the Java 1.4.2 API specification, backed by a hash table implementation. The developer `MAY` instead choose to implement the generic version of `java.util.Map` as documented in the Java 1.5.0 API.

This module `MUST` be named `MondoHashMap.java` and `MUST` support the complete `java.util.Map` interface and contract specification, including operations marked as “optional” in the JDK documentation. The `MondoHashMap` implementation `MUST NOT` use, access, refer to, or rely on the `AbstractMap` or any other implementation of the `Map` interface. The `MondoHashMap` implementation `MAY` employ the `java.util.AbstractSet` and/or `java.util.Ab` implementations to support the `Map.keySet()`, `Map.values()`, and/or `Map.entrySet()` operations.

As part of the project deliverables, the developer `MUST` demonstrate the performance of the `MondoHashMap` and show that it meets the quantitative requirements given in Section 5. To do so, it will probably be necessary to provide additional data members, methods, or subclasses to track quantities such as number of allocations and reallocations, number of accesses, wall clock time, etc. The choice of which data/methods/subclasses to provide is up to the developer, but all such entities `MUST` be documented in the API documentation (c.f., Section 6.1).

The `MondoHashMap` will form the core of the first milestone submission; refer to Section 6.1 for details on the full submission requirements.

4.2 MSpider

Important: MSpider Safety Requirements

This class will be producing many spiders, many of which will be running and tested at the same time. These conditions can easily lead to swamped networks, overloaded web servers, and pissed-off users and sysadmins (not to mention professors). In order to “play nice” and not disrupt other user’s access to web pages, your programs **MUST** adhere to the following rules:

1. The MSpider program **MUST NOT** spider or make web requests outside the `www.cs.unm.edu` web hierarchy. All pointers to content outside this domain **MUST** be discarded. All other programs in the `MoogLe` suite **MUST** obey this restriction as well. Essentially, all programs in this project **MUST NOT** generate *any* network traffic outside `www.cs.unm.edu`.
2. The MSpider program **MUST** accept a non-negative integer that sets an upper bound on the number of pages that the spider will crawl. This parameter **MUST** be user settable. The spider **MUST NOT** crawl more than this number of pages. If the spider is run with an existing web database (i.e., from a previous crawl), the count of crawled pages **MUST** include the number of pages already found in the previous crawl. (E.g., if the previous crawl found 10 pages and the `CRAWL-MAX` parameter for this run is 20, the spider **MUST NOT** crawl more than 10 additional pages. If the current `CRAWL-MAX` parameter is 5, then the spider must halt without crawling any pages.)
3. The MSpider engine **MUST** pause for *at least* one second between each page request. The pause time **MAY** be a user-configurable parameter, but MSpider **MUST NOT** allow any delay less than one second.
4. The MSpider engine **MUST** obey the `robots.txt` file and any meta robots tags.
5. The MSpider engine **MUST NOT** retrieve the same page more than once in any given crawl.

Violation of any of these requirements will result in reduced letter grade and may, in cases of serious abuse, result in failure of project 1 altogether.

The job of MSpider is to crawl the web domain (for this project, just `www.cs.unm.edu`) and compile the REVERSE INDEX and WEB DATABASE. Regarding the web domain as a graph, where PAGES are nodes and URLs are arcs, the crawl will be a *breadth-first search* with cycle detection. Note that the web domain may be an arbitrary, directed graph, so cycle detection will be necessary to prevent the spider from looping over the same set of pages indefinitely.

4.2.1 Reverse Index and Web Database

The heart of the web search engine is a REVERSE INDEX that stores a mapping from WORDs to the PAGES in which those WORDs occur. In addition, to support document scoring and TF/IDF ranking, the REVERSE INDEX will have to store the count of the number of times each word occurs in each document and the total number of documents in which each word has been seen. Note that the REVERSE INDEX will have a Map at its core, but it captures additional functionality

and will be more complex than a basic Map.

The WEB DATABASE will contain the REVERSE INDEX, but will also need to include additional information beyond the basic data in the REVERSE INDEX. To track max-crawl and implement cycle-detection, the WEB DATABASE will also have to store a list of all PAGES that have been seen. Finally, to implement durable state and restartability, the WEB DATABASE will have to contain a list of the outstanding URLs that have not yet been examined.

4.2.2 Durable State

The MSpider program MUST be capable of storing its WEB DATABASE (including REVERSE INDEX) durably on the drive beyond the execution of the program.⁴ The save files SHOULD use a .moog extension. Furthermore, the MSpider engine MUST be capable of loading a previously existing .moog save file and continuing the crawl from where it left off. When the MSpider is initialized with a saved state, it MUST NOT repeat any of the previous work. Specifically, it MUST NOT request any PAGES that were downloaded on its previous run.

The designer MAY use the Java serialization mechanism to save and retrieve the WEB DATABASE and REVERSE INDEX and any other necessary program state.

4.2.3 Support Libraries

A number of library tools are available to help construct the MSpider program. Network access and document retrieval is provided by the `java.net` package via the `java.net.URL` and `java.net.HttpURLConnection` classes. Elements of the `java.io` package will also be necessary to support network activities.

Parsing and analysis of HTML documents can be accomplished with the `javax.swing.text.html` tools. Specifically, the `javax.swing.text.html.parser.ParserDelegator` class provides parsing infrastructure, though it will be necessary for the designer to provide a specialization of the `javax.swing.text.html.HTMLEditorKit.ParserCallback` class to implement the necessary functionality.

Other elements of the Java JDK library may also be useful (e.g., `java.util`). Most elements of the JDK library are available for use in the MoogLe suite, but check Section 5 for restrictions.

The `gnu.getopt.Getopt` is a useful tool for handling command-line options, though the designer MAY choose to handle options in a different way. Note, however, that in the past students who have “rolled their own” command-line handling have reported that using `Getopt` would ultimately have saved time, in spite of the learning curve.

While class lecture, exercises, and labs will describe *some* of these libraries, there is not enough time to convey all important information in the necessary detail. *Therefore it is the designer’s responsibility to read and understand the necessary documentation.* In addition to the class text, the online Java API specifications and a number of online tutorial documents will be useful in completing this project.

⁴Indeed, it would be useless to the MoogLe client if it couldn’t store its state.

4.2.4 Spider Options

MSpider MUST be capable of being run from the command line in batch-mode and MUST NOT require a graphical user interface. Essentially, the user MUST be able to run the spider client with a command line similar to the following:

```
java MSpider [options] >&! logfile &
```

and then coming back some later time to view the results.

The spider client MUST support at least the following command-line options. The designer MAY provide additional options, but all such options MUST be fully described in the user documentation.

- c # Set the CRAWL-MAX parameter to # for this run. Note that different CRAWL-MAXs may be used on different runs.
- i iPage Set the initial page from which to begin the crawl.
- m modelFileName Load/save spider database to/from modelFileName.
- d dumpFileName Dump statistics on the REVERSE INDEX into the specified file and then exit.

In addition, the system SHOULD provide the following option:

- h Print a help message and then exit.

4.3 The MoogLe Client Program

The MoogLe client program provides a user interface to the REVERSE INDEX generated by the MSpider, including search and ranking mechanisms and a primitive browsing capability.

4.3.1 User Interface

The specific layout of the user interface is up to the designer, but it MUST contain at least the following elements:

1. Entry field allowing the user to enter a query.
2. “Go” button to initiate query processing. The system SHOULD bind “return” to the go function and possibly also a menu item, but a “Go” button MUST be present. The actual text of the “Go” button is up to the designer, so long as the meaning is clear.
3. Presentation pane that displays results. The system MUST display each URL that matches the entire query, but it MAY also display additional information (such as a sample of the contents of the URL’s targeted PAGE or the title of the PAGE). The system MAY simply display all relevant results in a single pane or it MAY show only a subset of them, but in the latter case it MUST offer a mechanism to page through *all* relevant results.
4. “Reset” button that resets the presentation pane to empty (or the initial greeting message). The system SHOULD also provide a menu option to achieve this.

5. “Quit” button that shuts down the `MoogLe` client and exits. The system **SHOULD** also provide a “Quit” menu option. When the program exits through the “Quit” function, the program **MUST** return a 0 exit status.
6. Minimal browser function that allows the user to click through any of the returned results to see the `PAGE` it points to. The level of HTML presentation supported by `javax.swing.JEditorPane` is sufficient.
7. An “I’m feeling random” button, or equivalent, that immediately loads a random `PAGE` from the set in the `REVERSE INDEX`.

The designer **MAY** implement additional UI functionality as well.

4.3.2 Query Parser

The query language supported by the `MoogLe` UI is a simple “AND/OR” query language. `MoogLe` **MUST** recognize the following syntax:

```

QUERY := WORD* |
        QUERY "AND" QUERY |
        QUERY "OR" QUERY |
        "( " QUERY " )"
WORD  := "[a-zA-Z0-9]+"

```

Note that there are only five kinds of tokens in this language: `WORDS`, `AND`, `OR`, `(`, and `)`. All tokens are `WHITESPACE` separated, but `WHITESPACE` is otherwise discarded. `PUNCTUATION` is discarded. `PUNCTUATION` falling in the middle of a word (e.g., `hyper-cool` or `TF/IDF`) **MAY** be treated as a token separator (`hyper-cool` becomes `hyper` and `cool`) or **MAY** be dropped and the token parts conjoined (`TF/IDF` becomes `TFIDF`).

The semantics of this language are reasonably natural:

1. A query consisting of a single `WORD` should return the set of documents in the `REVERSE INDEX` matching that word.
2. A query consisting of a sequence of `WORDS`, with no conjunctions specified (e.g., `word1 word2 word3 word4`) should be treated as if the `AND` conjunctive were specified (`word1 AND word2 AND word3 AND word4`).
3. The `AND` of two queries should return the conjunction of those queries – the set of documents that match *both* sub-queries.
4. The `OR` of two queries should return the disjunction of those queries – the set of documents that match *either* sub-query.
5. Precedence is left-to-right, unless delimited by parentheses. E.g.,

```
w1 AND w2 OR w3 AND w4
```

should parse as

((w1 AND w2) OR w3) AND w4)

while

(w1 AND w2) OR (w3 AND w4)

should parse as written.

6. A query that returns no documents, including the empty query, is syntactically valid and should display no URLs and/or print a message indicating that no matching documents were found.
7. Queries SHOULD be treated as case-insensitive but they MAY be treated as case-sensitive. The exceptions are the conjunctives (AND and OR), which SHOULD be treated as case-sensitive. Either way, case (in)sensitivity MUST be documented in the user manual.

The designer MAY choose to offer additional query language syntax and functionality. For example, support for a NOT modifier or quoted phrases would be useful extensions.

4.3.3 Relevance Sort

The search pass MUST retrieve all URLs that match the entire query. Afterward, however, it is necessary to decide how to order the results for presentation to the user. The system MUST support the TF/IDF scoring function, $s_{TF/IDF}(d, q)$, which assigns a real-valued score to document d with respect to query q (see Appendix A for information on the TF/IDF scoring function). The system MUST sort results in decreasing order according to this scoring function before presentation to the user. The system MAY also offer an alternate scoring criterion, but if it does so it MUST offer some user interface mechanism (such as radio buttons) to allow the user to pick which scoring function to use.

4.3.4 Query Trace File

For grading purposes the MoogLe client program MUST provide a query trace file mechanism. This mechanism will write the query, parse tree, and query result to a file for later manual and automatic analysis. This information MUST be recorded for every query issued by the user. This mechanism is *in addition* to the UI display of results described previously. The query trace file is meant primarily for machine consumption, rather than human, so the file format is designed to be simple for I/O rather than simple to read.

The MoogLe client program MAY allow the user to select a query trace file or MAY use a fixed file name. If a fixed file is used, it MUST be named “./[yourname].moogQuery.dat”, where [yourname] stands for the string “your last name”+“your first initial”. E.g., the course TA’s program might produce a file named ./brownj.moogQuery.dat.

While the designer is free to choose many factors about the UI, the MoogLe client MUST produce the following format *exactly*.

The format of the query trace file is:

```
===== BEGIN QUERY =====
```

```
[query info]
```

```
===== END QUERY =====
```

One "QUERY" entry MUST be present for each query issued by the user.

Each QUERY entry is formatted as follows:

```
=== BEGIN QUERY TEXT ===
```

```
[query text string, as typed in by user]
```

```
=== END QUERY TEXT ===
```

```
=== BEGIN QUERY PARSE ===
```

```
[query parse]
```

```
=== END QUERY PARSE ===
```

```
=== QUERY RESULTS ===
```

```
[query results]
```

```
=== END QUERY RESULTS ===
```

The QUERY TEXT STRING field MUST report the text string *exactly* as typed by the user. No whitespace, punctuation, etc. should be trimmed or added.

The QUERY PARSE MUST be formatted according to the following rules:

1. A single WORD MUST be formatted as:

```
"word"
```

with no additional whitespace. The WORD string MUST be printed in lower-case.

2. An AND conjunction MUST be formatted as:

```
(AND [first query term] [second query term])
```

with no whitespace beyond the single spaces separating AND and the first query term and between the first and second query term. Note that the query terms themselves may be WORDs or may be additional parenthesized phrases.

3. An OR conjunction MUST be formatted as:

```
(OR [first query term] [second query term])
```

with no whitespace beyond the single spaces separating OR and the first query term and between the first and second query term. Note that the query terms themselves may be WORDs or may be additional parenthesized phrases.

The QUERY RESULTS field MUST report each URL returned in response to the query, in order sorted by TF/IDF score, as well as the score itself. One URL/score pair MUST be reported per line. Each line is formatted as follows:

```
[url string] [TF/IDF score, to 2 digits of precision]
```

An example of such a file is available at http://www.cs.unm.edu/~terran/classes/cs351-s05/projects/p1/query_trace_example.dat

5 Quantitative Requirements

This section describes the performance and behavior requirements for the MoogLe software suite.

1. All of the elements of the MSpider Safety Requirements (Section 4.2) are included here, by reference.
2. All programs MUST NOT crash, core dump, dump a stack trace, or throw an exception on any input.
3. In the case of a RECOVERABLE ERROR, a program MUST issue a warning statement and continue processing. The program MAY choose to issue the warning statement to standard error, to a log file, or to a user interface element. If the warning is issued to a log file, the log file name and location MUST be a user-specifiable parameter to the program (either by command-line command or via a configuration menu).
4. In the case of an UNRECOVERABLE ERROR, a program MUST issue an error statement and terminate with a non-zero error condition. The program MAY use different exit codes to indicate different error conditions, but such codes MUST be documented in the user manual. The error message MUST be logged to the same destination that warning messages (from RECOVERABLE ERRORS) are.
5. In the case of any ERROR, a program MUST NOT delete, corrupt, or damage existing web database files or any other “stateful” files employed by the program suite.
6. The MondoHashMap.java module MUST NOT use or reference the Hashtable, HashMap, AbstractMap, HashSet, TreeSet, or any of their subclasses.
7. For (substantially) reduced credit, the MoogLe suite MAY use the HashMap class in place of MondoHashMap. Note that this requirement exists only as an aid in case the designer has difficulty getting MondoHashMap to work properly; for full credit the entire MoogLe suite MUST employ MondoHashMap and MUST NOT employ or refer to any of the classes listed in the previous bullet point.
8. The entire program suite MUST NOT employ or refer to the StreamTokenizer class or any element of java.util.regex, including indirect references to it via, e.g., String.split().
9. The programs MAY provide additional output for debugging purposes, but such output must be disabled by default. Any program MAY provide a command-line switch or a user-interface configuration utility to enable debugging support when desired.
10. The MoogLe suite MAY use the gnu.getopt.Getopt and gnu.getopt.LongOpt classes to assist in handling command-line options.
11. The designer MAY ask permission of the instructor or the TA to use any classes outside the JDK that have not already been mentioned. The final programs MUST NOT use any class outside the JDK that have not been explicitly allowed.

12. The `MoogLe` suite **MAY** assume that all valid user input is standard ASCII text in the range $(\text{char})0 - (\text{char})127$, inclusive. If a program encounters a character outside this range, it **MAY** treat it as a **RECOVERABLE** or **UNRECOVERABLE ERROR** or silently ignore it. If such characters are treated as **RECOVERABLE** or ignored, they **MUST NOT** disrupt the otherwise normal functioning of the program.
13. All programs **MUST NOT** assume that all user input is validly structured search statements. If a program encounters syntactically erroneous input (e.g., punctuation, multiple “AND” or “OR”s in sequence) it **MAY** produce a **RECOVERABLE** or **UNRECOVERABLE ERROR**, but it **MUST NOT** crash, corrupt the web database file, etc.
14. All programs **MUST NOT** assume that all web references point to syntactically valid HTML pages. The designer **MAY** use the tools in `java.net` and `java.swing.text.html` to help determine which pages are both HTML and are syntactically valid. All non-HTML content **MAY** be ignored by `MoogLe`, though the designer **MAY** choose to handle it in some reasonable way. All syntactically invalid HTML content **MAY** also be silently ignored, though the designer **MAY** choose to attempt to recover from a failed parse and continue to analyze this page.
15. The `MSPider` engine **MUST** require no more than amortized $O(n)$ time to analyze a single web page of size n **WORDS**. It **MUST** also require no more than $O(k)$ HTTP queries to retrieve a web hierarchy of k documents.
16. The `MoogLe` client **MUST** run in $O(m \cdot n)$ time for a query of m words each of which is associated with n web documents.
17. The `MondoHashMap` **MUST** support `get()`, `put()`, `remove()`, `size()`, and `isEmpty()` in amortized $O(1)$ time. The table **MAY** support key/value iteration in time proportional to the *capacity* of the table. For extra credit, it **MAY** support key/value iteration in time proportional to the number of keys/values (respectively). To receive the extra credit, the designer must demonstrate this convincingly in the performance documentation.
18. The `MondoHashMap` **MUST NOT** consume more than $O(n \cdot s)$ memory for n distinct keys, where s represents the combined size of a key/value pair.
19. The `MondoHashMap` **MUST** support the `keySet()` and `values()` operations with only $O(1)$ space above that required by the hashtable itself. Specifically, these operations **MUST NOT** replicate the underlying hashtable, nor duplicate any keys or values.
20. All user documentation **MUST** be grammatically correct and include correct spelling and usage. (You *will* be graded, in part, on the quality of your writing.)
21. The designer **MUST** document any areas in which her or his software suite does not meet this specification. **WARNING!** The grade penalty will be higher if the instructors discover an undocumented program shortcoming or bug than if it is documented up front.

6 Deliverables

This section describes the content to be delivered at each stage of the project (two milestones and a final rollout). Each deliverable is a superset of the previous one – it **MUST** include all of the materials from the previous deliverable as well as the new materials. Milestone 2 and Rollout **MAY** contain updates to the previous deliverables. E.g., if `MondoHashMap` was not fully functional at Milestone 1, a revised version **MAY** be submitted at Milestone 2 or Rollout. Doing so will not change the grade on the previous delivery, but it will contribute to a better grade on the current deliverable. If updates to a past deliverable are included, the `README.TXT` file **MUST** indicate which components (including code, documentation, tests, etc.) have been modified. For the deadlines of these stages, please refer to Section 7.

6.1 Milestone 1: `MondoHashMap`

The first project component due is the `MondoHashMap` implementation. The deliverables for this milestone are:

`MondoHashMap.java` The main class file for the `MondoHashMap` implementation.

Other Java source files Any other supporting code files necessary to compile, load, and use the `MondoHashMap` module.

API documentation The handin **MUST** also include the full, compiled JavaDoc documentation for the `MondoHashMap` implementation. This documentation **MUST** include full descriptions of every public or protected method, field, sub-class, enclosed class, or constructor employed by `MondoHashMap`. This documentation hierarchy **MUST** be included in a sub-directory named `documentation/` within the submission tarball package.

Performance documentation The handin submission **MUST** include a document describing the performance of the hashtable implementation and demonstrating (via empirical experiments) that it meets the quantitative performance goals established in Section 5 of this document. The designer **MAY** choose any tests that he or she desires to establish the performance of his/her `MondoHashMap`, but **MUST** describe all tests and why they lead to the stated conclusions about performance. This document **MUST** be named `PERFORMANCE.extension`, but it **MAY** be a plain text, HTML, PDF, or PostScript document (with the appropriate `extension`). It **MUST NOT** be a Microsoft Word or other nonportable format document.

Test cases The submission tarball **MUST** include a subdirectory named `tests/` that includes all of the test data used to demonstrate the performance of the hashtable implementation.

At the designer's option, this submission **MAY** also include:

`BUGS.TXT` This file documents any known outstanding bugs, missing features, performance problems, or failures to meet specifications of your submission. Note that the penalty for such problems will be smaller if they're fully documented here than if the instructors discover them independently.

README.TXT This file includes any other notes or documentation necessary to use or understand the submission that are not already included in other documentation.

The submission directory **MUST** be named `lastname_p1m1` and the submission tarball **MUST** be named `lastname_p1m1.tar.gz`.

6.2 Milestone 2: MSpider

The second milestone is the web spider engine. Required deliverable components include the following. **Note:** The WEB DATABASE that is produced by MSpider **MUST NOT** be submitted, but the submission **MUST** include statistics summary files.

MSpider.java The primary source file for the MoogLe web crawler tool.

Other Java source files Any other supporting code files necessary to compile, load, and use the `MSpider.java`. *Note:* if this program depend on external library code other than the Java JDK or the `gnu.getopt` suite, the submission tarball **MUST** either include the library whole or provide easy and explicit instructions on how and where to access such libraries. This documentation **MUST** be provided in the `README.TXT` file. *The designer is responsible for ensuring that all copyright and distribution conditions are adhered to.*

README.TXT This file **MUST** describe how to compile, configure, and install the MSpider engine. It **MUST** also list any dependencies on additional software support libraries. Finally, it **MUST** list any updates to Milestone 1 deliverables that are being included in this delivery.

Internal documentation The handin **MUST** also include the full, compiled JavaDoc documentation for all Java source files in the submission tarball. This documentation **MUST** include full descriptions of every public or protected method, field, sub-class, enclosed class, or constructor employed by the code. This documentation hierarchy **MUST** be included in a sub-directory named `documentation/` within the submission tarball package.

User documentation The handin submission **MUST** include complete user-level documentation for the MSpider engine. This documentation **MUST** include instructions on how to use MSpider including the functionality of all command-line options. The documentation **MUST** also describe the function and use of any additional programs included in the submission. User documentation **MUST** include information on the expected inputs and outputs of all programs, how to read and interpret the output, and information on all status and error messages that the programs could produce. This documentation **MUST** also include at least one example of how to run each program and how to interpret the output. This document **MUST** be named `USERDOC.extension`, but it **MAY** be a plain text, HTML, PDF, or PostScript document (with the appropriate *extension*). It **MUST NOT** be a Microsoft Word or other nonportable format document.

Performance documentation The handin submission **MUST** include a document describing the performance of the MSpider engine, including demonstrations that each PAGE is accessed only once, that the REVERSE INDEX is built correctly, that the web graph is fully examined (up to the max-crawl limit), etc. The designer **MAY** choose any tests that she or he desires

to establish the performance of her/his `MSpider` engine, but **MUST** describe all tests and why they lead to the stated conclusions about performance. This document **MUST** be named `PERFORMANCE.extension`, but it **MAY** be a plain text, HTML, PDF, or PostScript document (with the appropriate extension). It **MUST NOT** be a Microsoft Word or other nonportable format document.

Test cases The submission tarball **MUST** include a subdirectory named `tests/` that includes all of the test data used to demonstrate the performance of the `MSpider` engine.

At the designer's option, this submission **MAY** also include:

BUGS.TXT This file documents any known outstanding bugs, missing features, performance problems, or failures to meet specifications of your submission. Note that the penalty for such problems will be smaller if they're fully documented here than if the instructors discover them independently.

6.3 Rollout: Complete MoogLe Suite

The final deliverable is the complete `MoogLe` suite.

MoogLe.java and MSpider.java The primary source files for the `MoogLe` client and web crawler engine.

Other Java source files Any other supporting code files necessary to compile, load, and use `MoogLe.java` and `MSpider.java`. *Note:* if these programs depend on external library code other than the Java JDK or the `gnu.getopt` suite, the submission tarball **MUST** either include the library whole or provide easy and explicit instructions on how and where to access such libraries. This documentation **MUST** be provided in the `README.TXT` file. *The designer is responsible for ensuring that all copyright and distribution conditions are adhered to.*

README.TXT This file **MUST** describe how to compile, configure, and install the `MoogLe` suite (both programs). It **MUST** also list any dependencies on additional software support libraries. Finally, it **MUST** list any updates to Milestones 1 or 2 that are being included in this delivery.

Internal documentation The handin **MUST** also include the full, compiled JavaDoc documentation for all Java source files in the submission tarball. This documentation **MUST** include full descriptions of every public or protected method, field, sub-class, enclosed class, or constructor employed by the code. This documentation hierarchy **MUST** be included in a sub-directory named `documentation/` within the submission tarball package.

User documentation The handin submission **MUST** include complete user-level documentation for the `MoogLe` suite. This documentation **MUST** include instructions on how to use `MoogLe` and `MSpider`, including the functionality of all command-line options. The documentation **MUST** also describe the function and use of any additional programs included in the submission. User documentation **MUST** include information on the expected inputs and outputs of all programs, how to read and interpret the output, and information on all

status and error messages that the programs could produce. This documentation **MUST** also include at least one example of how to run each program and how to interpret the output. This document **MUST** be named `USERDOC.extension`, but it **MAY** be a plain text, HTML, PDF, or PostScript document (with the appropriate *extension*). It **MUST NOT** be a Microsoft Word or other nonportable format document.

Performance documentation The handin submission **MUST** include a document describing the performance of the `MoogLe` client, including assessment of the quality of the document sort provided by TF/IDF, average time to parse and retrieve each query, etc. The designer **MAY** choose any tests that she or he desires to establish the performance of her/his `MoogLe` client, but **MUST** describe all tests and why they lead to the stated conclusions about performance. This document **MUST** be named `PERFORMANCE.extension`, but it **MAY** be a plain text, HTML, PDF, or PostScript document (with the appropriate *extension*). It **MUST NOT** be a Microsoft Word or other nonportable format document.

Test cases The submission tarball **MUST** include a subdirectory named `tests/` that includes all of the test data used to demonstrate the performance of the `MoogLe` engine.

At the designer's option, this submission **MAY** also include:

BUGS.TXT This file documents any known outstanding bugs, missing features, performance problems, or failures to meet specifications of your submission. Note that the penalty for such problems will be smaller if they're fully documented here than if the instructors discover them independently.

7 Due Dates

Please refer to the class syllabus for the policy on late handins.

Jan 19, 3:00 PM P1 specification handed out.

Jan 31, 3:00 PM M1: `MondoHashMap` component due.

Feb 9, 3:00 PM M2: `MSpider` component due.

Feb 23, 3:00 PM Rollout: complete `MoogLe` suite due.

8 Grading Levels

The following constitutes an *approximate* guide to the credit associated with different levels of implementation. These guidelines are *not* absolute and do not guarantee any specific grade. Superficial achievement of the requirements for one level may still yield a lower grade if, e.g., code is poorly designed, coded, or tested or if documentation is incomplete, poorly written, or incoherent. "C" level is considered to be the minimal "functional" level of implementation.

C level MondoHashMap; full web crawling ability (including cycle detection and full adherence to the MSpider Safety Requirements); storage and recall of REVERSE INDEX; MSpider web crawl statistics report; basic user interface (fixed file load, single-word searches, no browsing capability, no reset or random capability); partial documentation (API, and user).

B level Everything included in level C, plus: handling of AND/OR queries; ability to load user-specified files into MoogLe client; reset and random capability; browser functionality; complete documentation.

A level Complete “MUST” functionality, as specified in this document (including TF/IDF document scoring).

A+ level All of A level, plus significant and innovative extensions (e.g., some of the MAY extensions in this document, plus at least one innovative idea).

Appendix A: The TF/IDF Scoring Function

It is relatively straightforward to make a search engine that retrieves documents in random order, but it’s much harder to make it return documents in a *relevant* order. The real difference between Google and all of the search engines that lost is that Google had a much more effective sorting criterion. In this project, you’re not going to implement Google’s sort criterion (the “PageRank” algorithm), but you *will* implement a reasonably effective scoring function, the “term frequency/inverse document frequency” (TF/IDF) function.

Let a *query* be a sequence of m words, $q = \langle q_1, q_2, \dots, q_m \rangle$ and the set of all possible queries be Q . Note that, with respect to querying, the AND/OR query language and the parse tree (Section 4.3.2) is irrelevant – the set of documents matching the entire AND/OR query has already been selected. We can regard the terms of the AND/OR query as a simple sequence of words. Now, given a set of N documents, $D = \{d_1, d_2, \dots, d_N\}$, a *scoring function* $s()$ assigns a real-valued score to each document/query pair: $s : D \times Q \rightarrow \mathbb{R}$. Given such a score, it is straightforward to sort the document set into decreasing order by score. If the scoring function is good, then the documents that the user finds most relevant will be returned at the top of the list.

Suppose that there are a total of k different unique words, t_1, \dots, t_k , found in all documents. Let the number of occurrences of word i in document j be w_{ij} . Further, let the number documents in which word i is found be c_i . Then the TF/IDF score assigned to document d_j with respect to a query $q = \langle q_1, \dots, q_m \rangle$, is defined to be:

$$s_{\text{TF/IDF}}(d_j, \mathbf{Q}) = \sum_{i=1}^m w_{ij} \ln \left(\frac{N}{c_i} \right) \quad (1)$$

Essentially, Equation 1 says that a document is scored more highly if it contains many examples of a query term (the w_{ij} factor), while it is penalized slightly if word i is a common term (the log fraction factor). The intuition is that very common terms (e.g., “the” or “computer”) give you very little indication of the true relevance of the document, while rare words should indicate very important documents. Note that, in principle, you could skip the query search phase and simply apply TF/IDF directly to *all* PAGES in the WEB DATABASE and pick off the top candidates. This would, however, be highly expensive and is well beyond the scope of this project.

Now you have enough mathematical background to implement the search engine. The rest is Java...