CS351 Fall 2011, Project 1

`Baconator`

a.k.a., The Six Degrees Machine

MondoSoft, Inc.

STATUS Specification and requirements document

VERSION 1.0

DATE Aug 23, 2011

0 Changelog

Version 1.0 Initial release. Aug 23, 2011.

Version 1.01 Sep 14, 2011.

- Fixed typo in EDGE SET definition (Section 5).
- Fixed error in cross-reference mechanism that yielded reference to "Appendix 9", rather than "Appendix A".
- Added `BaconParseActor.java` and `BaconParseDirector.java` to the deliverables of Section 8.4 to clarify that all three MUST be turned in.
- Clarified that `BaconLexer` MUST be a generic class, while `BaconParseActress`, `BaconParseActor`, and `BaconParseDirector` MUST be type-concrete (Section 6.7).
- Removed deprecated reference to statistics report from Rollout deadline (Section 9).

1 Summary

In the wave of social media and Web 2.x, the great minds of MondoSoft (your employer) have decided to leverage the company's core data-handling competencies by forging a bold path into the world of user content-driven media analytics.[1] As a prototype project, they have assigned you to build a tool for analyzing the *movie social graph*: the graph of actors, actresses, directors, and movies and who-worked-with-whom-on-what relationships. The goal is to be able to answer

[1]That is, movie data are worth big $$$, and the big bosses are too lazy to type in all the movie data themselves.

business value-added queries about the movie social graph in realtime for high-revenue-generating customers.[2]

You have caught the lucky job of designing and building the `Baconator`, a prototype data loader and query tool. This tool will be able to read conveniently-formatted flat text file data files describing the actresses, directors, actors, movies, and their relationships, and it will provide a state of the art[3] text-mode command-line interface. Finally, in order to maximize revenues[4], the big bosses have decreed that you will implement the underlying graph data structure and analysis routines as separate, generic modules that can be reused in other products or sold as a standalone library.

[2]I.e., the big bosses want to be able to win at the "Six degrees from Kevin Bacon" game at their cocktail parties.

[3]State of the art for 1975, anyway.

[4]Sell your blood, sweat, and tears twice.

Contents

2 Group/Individual Effort

This project is an *individual* programming project. Refer to the CS351 Fall 2011 Syllabus and FAQ for details on what this means.

3 Java Language Level

You MUST write this project in Java 1.5.x (a.k.a., "the Java 5 platform") or a later version. Your code MUST support generic classes where required by the required project interfaces.

4 Libraries and Support Code

DEVELOPERs MAY use any objects or functions available through the Java 1.5.x or 1.6.x SE JDK.

DEVELOPERs MUST use a number of objects and functions provided by the instructors in the `cs351_p1_support_code.v1.0.jar` library file. The specific objects/functions will be discussed throughout this document.

DEVELOPERs MUST use the JUnit 4 (or later) library for unit testing.

DEVELOPERs MUST NOT use any other libraries or code, beyond those listed here, without prior permission from the CS351 instructors.

5 Definitions

The following definitions will be used in this document:

ACTOR A person named in the `actors.list` file. A type of ENTITY.

ACTRESS A person named in the `actresses.list` file. A type of ENTITY.

CLOSED SET The set of NODEs that have already been examined in a graph search such as a breadth-first or depth-first search. Each node generated by the search (including the start node) is added to the OPEN QUEUE if, and only if, it is not already in the CLOSED SET. This prevents cycling by ensuring that each NODE is enqueued on OPEN only once.

CLI Command-line interpreter; a simple, text-mode user interface to the `Baconator` program.

DEVELOPER The project designer, programmer, tester, and documenter. You.

DIRECTOR A person named in the `directors.list` file. A type of ENTITY.

EDGE An arc (or pointer) between two NODEs in a GRAPH. In the `Baconator` database GRAPH, an EDGE is represented via an "acted in/contained actor—actress" or "directed/was directed by" relation in one of the supported IMDB data files. Each edge has an origin (a.k.a., parent, top, or, from) NODE and a destination (a.k.a., child, bottom, or to) NODE. An EDGE is allowed to start and end at the same NODE — this is called a self-EDGE. The destination (to) of an EDGE is considered to be the NEIGHBOR of the origin (from).

Note: In the IMDB data, self-edges should be impossible, as people can only act in/direct movies, while movies cannot act in/direct themselves. Therefore, the IMDB data graph is a *bipartite* graph — its nodes can be divided into two classes (people and movies) and all edges pass between these classes.

EDGE SET The collection of all EDGEs in a graph. Formally written $E = \{(v_i, v_j) \in V \times V : v_i \text{ is connected to } v_j\}$. The size of the EDGE SET, $|E|$, is the total number of EDGEs in the GRAPH.

EDGEs may be either directed or undirected. In the `Baconator` project, the underlying `Graph` data structure is a directed graph data structure – every EDGE as a "start" and an "end" NODE. However, the IMDB data relations are considered to be undirected — if person A acted in movie B, then movie B contains actor/actress A. This should be represented by adding both edges (A, B) and (B, A) via two calls to `Graph.addEdge()`.

ENTITY A logical entity (a "thing") in the IMDB data domain. ENTITIES form the NODEs of an undirected relationship GRAPH whose edges have semantics like "acted in", "directed", etc. `Baconator` MUST support at least the ENTITIES ACTRESS, ACTOR, DIRECTOR, and MOVIE. It MAY also support additional entity types (such as FX company), at the DEVELOPER's discretion.

GRAPH An abstract data structure containing two types of entities: NODEs (a.k.a., vertices) and EDGEs (a.k.a., arcs). A GRAPH is a generalization of a list or a tree, that allows NODEs to be connected arbitrarily. The IMDB data is a moderately large GRAPH, composed of ENTITIES and acted-in/directed EDGEs. Formally, a GRAPH is often written $G = \langle V, E\rangle$, where V is the NODE SET (or vertex set) and E is the EDGE SET.

Note: In the IMDB data, all edges are considered to be bidirectional — if person A acted in movie B, then both (A, B) and (B, A) are EDGEs in the GRAPH. However, the underlying `graph` data structure is a directed graph data structure, so this will require two `Graph.addEdge()` calls — one to add the (A, B) EDGE and one to add the (B, A) EDGE.

EXTRA CREDIT Denotes an opportunity for optional extra credit. The amount of extra credit assigned will depend on the scope and challenge of the implemented feature. However, be sure to see the class policy on extra credit in the CS351 FAQ.

All EXTRA CREDIT features introduced by the DEVELOPER MUST be appropriately documented in USER and INTERNAL DOCUMENTATION.

Note: Opportunities for EXTRA CREDIT noted in this document are not exhaustive — we are open to other extra credit ideas.

INDEGREE The INDEGREE of a NODE, v, in a GRAPH is defined to be the total number of EDGEs that *end* at v. That is, the number of EDGEs for which v is a child. Formally, $\text{indegree}(v) = |\{e = (v_i, v) : e \in E\}|$. Note that the INDEGREE *does* count self-EDGEs.

INTERNAL DOCUMENTATION Documentation of the design, structure, behavior, and public/protected interface of the program code. Includes internal code comments, Javadoc API documentation, and external, human-level design documentation.

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.)

MOVIE A movie ENTITY listed in one of the `actors.list`, `actresses.list`, or `directors.list` data files.

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.

NEIGHBOR A neighbor of NODE v_i in the GRAPH is a NODE v_j that is connected to v_i by an EDGE starting at v_i. That is, v_j is a NEIGHBOR of v_i iff there is an edge from v_i to v_j: v_j is neighbor of $v_i \Leftrightarrow (v_i, v_j) \in E$.

For example, in the IMDB data, if person A acted in movie B, then B is a NEIGHBOR of A.

NEIGHBOR SET The NEIGHBOR SET of a NODE v_i is the set of all NEIGHBOR nodes to v_i: $\text{NeighborSet}(v_i) = \{v_j : (v_i, v_j) \in E\}$

NODE A vertex in a GRAPH. In the `Baconator` data graph, NODEs are represented by ENTITIES. A NODE has zero or more outgoing EDGEs and zero or more incoming EDGEs. (See: INDEGREE and OUTDEGREE.)

OPEN LIST The ordered set of NODEs that have been generated, but not yet examined, by a graph search such as a breadth-first or depth-first search. The OPEN LIST represents the set of "upcoming" NODEs that need to be examined later in the search, sorted by the priority that they'll be looked at.

Note: While this is commonly referred to as "the open list", it need not *literally* be a list data structure. It should be whatever data structure is appropriate to the search in question. See Appendix C.

OUTDEGREE The OUTDEGREE of a NODE, v, in a GRAPH is defined to be the total number of EDGEs that *start* at v. That is, the number of EDGEs for which v is a parent. Formally, $\text{outdegree}(v) = |\{e = (v, v_i) : e \in E\}|$.

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.

RELATIONSHIP An interaction between exactly two ENTITIES. In the IMDB data, RELATIONSHIPs may be "acted in", "directed", "contained actor/actress", or "was directed by". The RELATIONSHIPs are represented by EDGEs in a GRAPH data structure.

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

SUBMISSION ARCHIVE The archive file that the CONTRACTOR turns in for each milestone. The SUBMISSION ARCHIVE MUST be a `.jar` file. The SUBMISSION ARCHIVE file MUST be named `lastname_p1[milestone](version).[jar]` where `[milestone]` indicates which milestone is being submitted (`m1`, `m2`, `m3`, `r`). The `(version)` string is optional; if present, it indicates which version of a milestone is being submitted, in case the CONTRACTOR wishes to submit a revised version of the milestone.

Examples of valid SUBMISSION ARCHIVE names are `lane_p1m1.jar` and `lane_p1r_v2.jar`.

See Section 6.2 for details on the directory structure of the `.jar` file.

UNRECOVERABLE ERROR An error condition from which recovery is impossible. The program MUST produce an error message describing the condition and then cleanly halt. The program MUST NOT crash, core-dump, display a stack trace, dump a raw exception to the screen, or corrupt any stored data.

USER DOCUMENTATION One or more human-readable documents that describe the use of the `Baconator` program, in terms comprehensible by non-experts and non-programmers. USER DOCUMENTATION MUST include the file `user_doc.extension` (for an appropriate value of `extension`), and MAY include a `README.TXT`, `CHANGELOG.TXT`, or `BUGS.TXT` file. Where appropriate, USER DOCUMENTATION also includes in-program help features (such as the `help` command of Section 6.8).

USER DOCUMENTATION is as opposed to INTERNAL DOCUMENTATION.

6 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 8.

The `Baconator` project comprises one main program (the `Baconator` program itself) as well as necessary supporting and test files. In addition to the "performance" code of this project, you will also develop a complete set of test cases and test code for all elements of the project. Some of this MUST be in `JUnit`, but it will probably prove necessary to have external (e.g., shell script) test suites as well.

6.1 Package and File Names

The designer MUST use at least the following packages

`unm.cs351.p1`	All program class files specified in this document (`Baconator.java`, `MovieGraph.java`, etc.)
`unm.cs351.p1.test`	All `JUnit` test files

The designer MAY offer additional packages. Any additional packages MUST be sub-packages of `unm.cs351.p1`, and all such packages MUST be documented in `README.TXT`.

6.2 Format of the SUBMISSION ARCHIVE

The SUBMISSION ARCHIVE MUST be a `jar` file containing the following top-level directories:

`src/` Root of all source code (package) directories.

`bin/` Root of all compiled `.class` file directories.

`doc/` Contains all USER DOCUMENTATION and all human-readable INTERNAL DOCUMENTATION (such a design documents).

`api-doc/` Contains all INTERNAL DOCUMENTATION generated from Javadoc (i.e., documentation of the code itself).

The SUBMISSION ARCHIVE MAY, in addition, contain any of the following:

`data/` Directory containing any data used by test scripts. However, the submitted archive MUST NOT contain more than 1 Mb of data. (The DEVELOPER is free to use as much test data as she/he pleases on personal machines, but MUST NOT turn in entire, raw IMDB data files.)

`scripts/` Any non-java scripts or supporting configuration files. Examples include `ant` scripts or Eclipse wizard `xml` configuration files.

The SUBMISSION ARCHIVE MAY contain other top-level directories, at the DEVELOPER's discretion, but the purpose of all such directories MUST be documented in the USER DOCUMENTATION (e.g., in the `README.TXT` file.)

The SUBMISSION ARCHIVE MUST NOT contain any of the following:

- The Eclipse `.classpath` or `.project` files.
- Any Subversion, CVS, git, SCCS, or other revision control system meta-data directories or files.
- Any of the content of the `cs351_p1_support_code.v1.0.jar` (or the entire `cs351_p1_support_code.v1.0.jar` itself). DEVELOPERs may assume that their code is being run in an environment in which the `cs351_p1_support_code.v1.0.jar` is on the Java `CLASSPATH`.

  Hint: To configure Eclipse to access a `jar` file (such as `cs351_p1_support_code.v1.0.jar`) from a project, select the project name in the package explorer, pick "`Properties`" from the context menu, go to `Java Build Path`→`Libraries`→`Add External Jars`, and browse to the location where you have stored the `cs351_p1_support_code.v1.0.jar` file.

For rollout (Section 8.5), the SUBMISSION ARCHIVE SHOULD be an executable archive. That is, running

```
java -jar lastname_p1r.jar
```

SHOULD run the `Baconator` program.

6.3 The IMDB Movie Data

Important: IMDB Intellectual Property Rights Requirements

For this project, we will be using data made freely available by the Internet Movie Database (IMDB). Access to the data is predicated on a number of terms and conditions specified by IMDB at:
`http://www.imdb.com/help/show_leaf?usedatasoftware`
Additional terms MAY be attached to the header/footer info of a given IMDB data file.
Some key terms are reproduced here, however, all students MUST obey all IMDB-specified terms and conditions. The full terms and conditions given at the master IMDB site, above, supersede any given here; any discrepancies between them MUST be resolved in favor of IMDB.

- All data used in this project and all software developed as part of this project are **NON-COMMERCIAL**. You MUST NOT sell, rent, republish, or repost any of IMDB's data or any software arising from this project.
- You MUST NOT redistribute the software you develop for this project outside of this class. (See also the CS351 FAQ in the course syllabus.)
- You MUST NOT robotically access IMDB's site or its data. All IMDB files MUST be individually manually downloaded.
- If you are in any way uncomfortable using IMDB's data, you are not required to use it. On request, the course instructors will make synthetic, non-IMDB data available to you. You MUST make such a request at least five days before the due date of Milestone 3.

The raw text movie data files are available via FTP download from one of a number of FTP sites. The FTP sites are listed on IMDB's "Alternative Interfaces" page, under "Plain Text Data Files":
`http://www.imdb.com/interfaces#plain`

The `Baconator` program MUST support at least the following files, or correctly structured extracts of them:

- `actors.list`
- `actresses.list`
- `directors.list`

Extra Credit: *Support additional IMDB file types. The magnitude of the extra credit will be related to the complexity of such files, which fields are extracted from them, etc. All additional file types MUST be documented in the USER DOCUMENTATION and via the* `help` *command (Section 6.8).*

For each of these files, the `Baconator` program MUST be capable of parsing the entire file, according to the grammar given in Appendix A. These files are rather complex, so it is not required that `Baconator` extract *all* fields from them. At a minimum, `Baconator` MUST extract ENTITY (ACTRESS, ACTOR, DIRECTOR, MOVIE) names from the file, and the relationships among them (acted-in, directed, contained-actor/actress, was-directed-by).

Extra Credit: *Extract and store additional fields from the supported file types. How* `Baconator` *chooses to handle such fields is up to the DEVELOPER, however to receive the extra credit, they must be stored in the database as separate info (i.e., not simply string concatenated on to a person or movie name) and the CLI (Section 6.8) MUST provide some way to access/use them. Any such field extensions MUST be documented in the USER DOCUMENTATION.*

Each of these files starts with a header and ends with a footer. While those sections contain a great deal of information that is useful to humans, it is all irrelevant to the `Baconator`. `Baconator` MUST be capable of ignoring the content of the file headers and footers and still correctly parsing and interpreting the data content of the files.

`Baconator` MAY treat a file lacking a header and/or footer as malformed. If so, it MUST treat this case as a RECOVERABLE ERROR. At the DEVELOPER's discretion, `Baconator` MAY treat such files as syntactically correct (so long as the rest of the data is correctly formatted).

6.4 The Movie Graph Data Structure

`Baconator`'s primary jobs are to maintain a data structure representing ENTITIES and their RELATIONSHIPs, extracted from the IMDB, and to provide an interface allowing a user to query that data.

Conceptually, the RELATIONSHIP data forms a GRAPH — a mathematical/algorithmic data structure that encodes ENTITIES (ACTORs, ACTRESSes, DIRECTORs, MOVIEs) as GRAPH NODES and their RELATIONSHIPs as GRAPH EDGEs. The supported RELATIONSHIPs are those captured in the supported IMDB text data files (Section 6.3). Specifically, `Baconator` MUST recognize and encode:

- ACTRESS *acted-in* MOVIE
- MOVIE *contained* ACTRESS
- ACTOR *acted-in* MOVIE
- MOVIE *contained* ACTOR
- DIRECTOR *directed* MOVIE
- MOVIE *was-directed-by* DIRECTOR

Extra Credit: `Baconator` *MAY support other ENTITIES and/or RELATIONSHIPs. For example,* `Baconator` *MAY support special effects companies as ENTITIES or* produced *as a RELATIONSHIP. Any such additions MUST be supported at the data structure level, MUST be queryable at the CLI level, and MUST be documented in the USER DOCUMENTATION and appropriate INTERNAL DOCUMENTATION.*

While the RELATIONSHIPs are given unidirectionally in the IMDB files (director A *directed* movie B), `Baconator` MUST treat them as bidirectional. That is, whenever `Baconator` encounters a RELATIONSHIP of the form A *related-to* B, it MUST insert both (A, B) and (B, A) into its EDGE SET. (Effectively, `Baconator` is treating the data as an undirected graph, even though it is described as a directed graph.)

Extra Credit: *`Baconator` MAY support EDGE and/or NODE annotations. These are additional, meta-data attached to each EDGE and/or NODE. For example, the `actresses.list` and `actors.list` files often provide "role" notations (e.g., "Hamill, Mark (I)" appeared in "Star Wars (1977)" as "[Luke Skywalker]".) These could be encoded as meta-data attached to EDGES. Any such additions MUST be supported through the CLI (Section 6.8) and documented through appropriate USER and INTERNAL DOCUMENTATION.*

The DEVLOPER's version of `Baconator` MUST include a `MovieGraph .java` file implementing the `Graph<T>` interface from the `cs351_p1_support_code.v1.0.jar` library. `Baconator` MAY include additional `.java` files supporting the `MovieGraph` code.

The `Graph<T>` interface is generic — a `Graph` is a container (or collection, in Java terminology) that can contain an arbitrary type of content object. The `MovieGraph` implementation MUST remain generic.

Note that the `Graph<T>` interface does not include any ways to explicitly represent the types of NODES (ENTITIES) or EDGES (RELATIONSHIPS) – the `Graph<T>` treats NODES and EDGES as homogeneous and untyped. To record information about ENTITY type (ACTRESS vs ACTOR etc.) or RELATIONSHIP type (acted-in, directed, etc.), `Baconator` will use a Movie Database class or classes (Section 6.5).

The `Graph<T>` will be `Baconator`'s internal representation of ENTITY and RELATIONSHIP data. The design of `MovieGraph` is at the DEVELOPER's DISCRETION, however it MUST obey all constraints documented in the `Graph<T>` API. (E.g., it must obey all asymptotic time and space constraints.)

6.5 The Movie Database

The `Baconator` program MUST track the type of all ENTITIES. That is, it MUST distinguish between ACTRESSes, ACTORs, DIRECTORs, and MOVIEs. This is a level conceptually above the `Graph` data structure (Section 6.4) and MUST NOT be explicitly represented in the `MovieGraph` data structure code. The `Baconator` program MUST store this information in auxiliary data structures. Those auxiliary data structures MAY be a separate class that encapsulates or references the `Graph`, or they MAY be represented via the type `T` that is stored in the `Graph`.

At the minimum, `Baconator` MUST be able to report the number of unique ACTORs, ACTRESSes, DIRECTORs, and MOVIEs that it encounters.

Extra Credit: *`Baconator` MAY provide additional functionality using the type information. For example, it MAY allow search queries to be restricted by type (Section 6.8).*

6.6 The `MovieGraphAnalyzer` Class

The ultimate objective of `Baconator` is to be able to answer queries like, "find the shortest path between ENTITIES A and B" or "extract the set of all ENTITIES within 3 hops of ENTITY A". Rather than clutter the `MovieGraph` class with this functionality, the necessary analysis functions are split out into the `GraphAnalyzer<T>` interface.

Like `Graph<T>`, `GraphAnalyzer<T>` is a generic interface — its concrete implementations operate on the same type of object contained in the `Graph<T>`. Since `GraphAnalyzer` operates on the *structure* of the graph, rather than its *contents*, it does not require any additional knowledge about the type `T`.

`Baconator` MUST provide a `MovieGraphAnalyzer .java` class implementing `GraphAnalyzer<T>`. The `MovieGraphAnalyzer` concrete class MUST remain generic. `MovieGraphAnalyzer` MUST implement all functions in the `GraphAnalyzer` interface, though it may implement additional functions, at the DEVELOPER's discretion. The DEVELOPER's `MovieGraphAnalyzer` implementation MUST obey all asymptotic time and space complexity constraints specified in the `GraphAnalyzer<T>` interface documentation.

Extra Credit: *Provide additional graph analysis functionality. The magnitude of any extra credit will depend on the scope, complexity, and correctness of the implemented functionality. Any such additions MUST be appropriately documented in USER and INTERNAL DOCUMENTATION.*

6.7 The Lexical Analyzer and Parser

In order to populate the movie database, `Baconator` needs to be able to read data out of the IMDB text data files (Section 6.3), break them into ENTITIES, and infer the RELATIONSHIPs between those entities. The process of decomposing raw text into grammatical (structural) relationships is called *parsing*, and it uses as a subroutine *lexical analysis* (or simply *lexing*).

`Baconator` MUST provide a lexical analyzer that implements the `LexicalAnalyzer<T>` interface provided in the `cs351_p1_support_code.v1.0.jar` library. The lexer MUST be named `BaconLexer.java`. The `BaconLexer` MUST generate tokens conforming to the `Token` interface of `cs351_p1_support_code.v1.0.jar`, though the specific `Token` implementation is up to the DEVELOPER. The lexer MUST remain a generic class (retaining type parameter `T`) – that is, the `BaconLexer` implementation MUST NOT commit to a specific token type. The declaration of `BaconLexer` MUST be:

```
public class BaconLexer<T extends Enum<T> & HasPattern>
  implements LexicalAnalyzer<T>
```

`Baconator` MUST provide at least three parsers implementing the `Parser<S,T>` interface of the `cs351_p1_support_code.v1.0.jar` library:

`BaconParseActress` Parse a complete `actresses.list` data file, or properly formed fragment thereof, including handling (ignoring) headers and footers.

`BaconParseActor` Parse a complete `actors.list` data file, or properly formed fragment thereof, including handling (ignoring) headers and footers.

`BaconParseDirector` Parse a complete `directors.list` data file, or properly formed fragment thereof, including handling (ignoring) headers and footers.

All `Parser<S,T>` implementations MUST provide a single-argument constructor of the signature:

```
    /*
     * Initialize parser to read from specified file.
     */
    public BaconParseFoo(String fileName) throws IOException,
      ParseException;
```

where "`Foo`" is one of `Actress`, `Actor`, or `Director`.

`Baconator` MAY provide additional parsers and/or additional classes to support the parser implementations, at the DEVELOPER's discretion.

The three concrete parsers MUST be type-concrete; that is, they MUST NOT retain the generic type signature of `Parser<S, T extends Enum<T> & HasPattern>`. They MUST commit to specific types for `S` and `T`. The specific `S` and `T` classes are at the DEVELOPER's discretion, but the DEVELOPER MUST provide (at least skeleton) implementions for both. (Also see Section 8.4 for details on token sets.)

6.8 The Command-Line Interpreter (CLI)

The user interacts with the `Baconator` program and its database via a simple command-line interpreter (CLI). Much like a UNIX or Windows terminal (shell) program, the CLI issues a command prompt and waits for the user to input a command. The CLI then interprets and executes the command and displays any results of the command (possibly issuing a help message or an error message). The whole prompt-command-interpret-response procedure takes place in a loop that continues until the user issues a `quit` command.

The `Baconator` CLI MUST support at least the following commands. It MAY support additional commands, or extensions to these commands, at the DEVLOPER's discretion. Any such extensions MUST be documented in the USER DOCUMENTATION.

`help` Issue a concise summary of commands supported by the `Baconator` CLI and their respective syntaxes.

Syntax `help`

Input arguments None

Effect on database None

Output/display Display a help message, listing the usage message for every command.

`load` Load one or more files. Capable of loading either a saved binary file, or one of the `Baconator`-supported text files.

Extra Credit: *(small) Support loading from compressed files.*

Syntax `load fileType file [file]*`

Input arguments `fileType` is one of the following:

- **`actor`** Load an `actors.list`, or compatible fragment of one.
- **`actress`** Load an `actresses.list`, or compatible fragment of one.
- **`director`** Load an `directors.list`, or compatible fragment of one.
- **`saved`** Load a pre-compiled, binary-format movie database file that was previously saved with the `save` command.

`file` is the relative or absolute pathname of a data file to be loaded. (Note: `Baconator` MUST support both relative and absolute path references to files.)

Effect on database Add the ENTITIES and RELATIONSHIPs found in the specified file(s) to the internal database.

On Error An error in loading a file is considered a RECOVERABLE ERROR. `Baconator` MUST issue an appropriate warning and continue (return to the CLI prompt), without corrupting the database. A file MAY be partially loaded, but all ENTITIES and RELATIONSHIPS loaded MUST be correct. (E.g., `Baconator` MUST NOT store a fragment of a name in a NODE or leave one end of an EDGE dangling.) In no case must the database be corrupted by partially loaded data, nor must any existing data be lost.

Extra Credit: *Support* transactional semantics for load*: load a file completely or not at all.*

Output/display Messages at the beginning and completion of loading, indicating success. Appropriate warning messages in the case of loading failure.

`save` Save internal database to a binary-format file, to be retrieved later via `load`.

Extra Credit: *(small) Support loading from compressed files.*

Syntax `save fileName`

Input arguments `fileName`: a relative or absolute path to the destination save file. (Note: `Baconator` MUST support both relative and absolute path references to files.)

Effect on database None

On Error An error in saving a file is considered a RECOVERABLE ERROR. `Baconator` MUST issue an appropriate warning and continue (return to the CLI prompt), without corrupting the database. `Baconator` MUST NOT produce a corrupt or partial output file – any such mangled outputs MUST be deleted.

Output/display Create the specified file, containing a binary image of the internal database.

Display at least messages at the beginning and completion of saving, indicating success. Appropriate warning messages in the case of failure.

`pwd` Print the current working directory.

Syntax `pwd`

Input arguments None

Effect on database None

Output/display Display the current working directory: the directory with respect to which all relative paths in `load` or `save` commands are resolved.

`cd` Change current working directory.

Syntax `cd newdir`

Input arguments `newdir`: a relative or absolute path to change to.

Effect on database No change to the database, but change `Baconator`'s notion of the current working directory to the specified directory. After a successful `cd` command, all relative pathnames MUST resolve with respect to the newly specified directory.

On Error Issue appropriate warning to the user and leave the working directory unchanged.

Output/display None, if successful.

`ls` List contents of current working directory.

Syntax `ls`

Input arguments None

Effect on database None

Output/display Print a sorted list of the contents of the current working directory, sorted by name. Every entry SHOULD be printed with either the string '`dir`', to indicate a subdirectory, or an indication of the size of the file.

`stats` Print a summary of the contents of the internal database.

Syntax `stats`

Input arguments None

Effect on database None

Output/display Print at least the following information about the database:

- Count of the total number of unique ACTRESSes in the database.
- Count of the total number of unique ACTORs in the database.
- Count of the total number of unique DIRECTORs in the database.
- Count of the total number of unique MOVIEs in the database.
- Count of the total number of unique EDGEs in the database.

`diam` Report the diameter of the graph (the maximum, finite, distance between any pair of reachable nodes). If the graph has multiple, disconnected components, report the diameter of the largest diameter component.

Syntax `diam`

Input arguments None

Effect on database None, though will compute auxiliary data structures in the process of computing the graph diameter.

On error Errors in the diameter search are RECOVERABLE ERRORs. Print an appropriate warning message and return to the CLI prompt.

Output/display Print the diameter of the (largest diameter component) of the graph.

`Baconator` SHOULD issue a warning if the `diam` computation is expected to take an unduly long time.

`path_length` Print the length of the shortest path between two ENTITIEs.

Syntax `path_length from to`

Input arguments

`from` Start node of the path search.

`to` End node of the path search.

Effect on database None, though will compute auxiliary data structures in the process of computing the shortest path.

On error Errors in the shortest path search are RECOVERABLE ERRORs. Print an appropriate warning message and return to the CLI prompt.

Output/display Length of the shortest path between `from` and `to`. Special cases:

- If `from==to`, then the path length is 0.
- If `to` is not reachable from `from`, then the distance is infinite. Display a message indicating that `to` is not reachable from `from`.

`path` Display all steps of the shortest path between two nodes.

Syntax `path from to`

Input arguments

`from` Start node of the path search.

`to` End node of the path search.

Effect on database None, though will compute auxiliary data structures in the process of computing the shortest path.

On error Errors in the shortest path search are RECOVERABLE ERRORs. Print an appropriate warning message and return to the CLI prompt.

Output/display Display all ENTITIES along the shortest path starting at `from` and ending at `to`, one per line. Note that both `from` and `to` are included in the shortest path, and should be displayed. Special cases:

- If `from==to`, then the path is the singleton path containing only the ENTITY `from`. Print this single element, once only.
- If `to` is not reachable from `from`, then the distance is infinite. Display a message indicating that `to` is not reachable from `from`.

`sphere` Extract and save a sub-graph of the complete movie graph, consisting of all ENTITIES within a fixed number of EDGEs of a specified "center" ENTITY.

Syntax `sphere c r outFile`

Input arguments

`c` An ENTITY to use as the center of the sphere.

`r` A non-negative integer specifying the radius of the sphere. (Note: Allowed to be 0.)

`outFile` A relative or absolute path name specifying a destination file for the subgraph. (Note: `Baconator` MUST support both relative and absolute path references to files.)

The semantics are that this constructs a new, secondary database, consisting of a subgraph of the original movie database. It then writes this secondary database to disk, as a binary database image file that can later be loaded via the `load` command, and then flushes the secondary database from memory.

The specific subgraph to be constructed is a "sphere" – the set of all ENTITIES within `r` hops (EDGEs) of the specified start ("center") ENTITY `c`. (That is, all nodes n such that $distance(\texttt{c}, n) \leq \texttt{r}$.) If `r==0`, then the only NODE of the sphere is `c` itself.

Effect on database None, though this constructs a new database containing a subgraph of the original graph.

On error Errors during graph traversal (to construct the sphere) or during file I/O are considered to be RECOVERABLE ERRORS — `Baconator` MUST issue an appropriate warning, terminate the `sphere` computation or I/O, and return to the CLI prompt. It MAY flush the intermediate secondary database in this case, but it MUST NOT corrupt any of the primary database. In the case of I/O errors, `Baconator` MUST NOT produce a corrupt or partial output file – any such mangled outputs MUST be deleted.

Output/display Write the sphere subgraph to disk. Display messages indicating start and end of processing.

`quit` Exit the `Baconator` program.

Syntax `quit`

Input arguments None

Effect on database Flush the database. (Possibly simply by allowing the Java garbage collector to reclaim the memory as the program exits.)

Output/display A message indicating that the program is terminating.

Extra Credit: *(small) Prompt the user to save the database, if there are unsaved changes.*

7 Quantitative Requirements

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

1. All of the elements of the IMDB Intellectual Property Rights requirements (Section 6.3) are included here, by reference.

2. All behavior and performance specifications of any interfaces provided to the designer (e.g., `Graph<T>`) are included here, by reference.

3. All programs MUST NOT crash, core dump, dump a stack trace, or throw an exception on any input.

4. 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).

5. 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.

6. In the case of any ERROR, a program MUST NOT delete, corrupt, or damage existing MDB files or any other "stateful" files employed by the program suite.

7. 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.

8. 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.)

9. 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.

8 Deliverables

This section describes the content to be delivered at each stage of the project (three 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 `MovieGraph` was not fully functional at Milestone 2, a revised version MAY be submitted at Milestone 3 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. Similarly, more test cases can be submitted in Milestone 2 or later, etc.

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 9.

For each submission, the designer MAY assume that the submitted code will be executed in an environment with the following items freely available (not requiring further reference or configuration):

- The complete, standard, Javasoft Java JDK, version 1.5.x, including the `java`, `jar`, and `javadoc` executables, and the complete JDK 1.5.x library suite.

- The `JUnit` library, version 1.4.x.

- The instructor-provided CS351-P1 Support Code `.jar` file (`cs351_p1_support_code.v1.0.jar`).
- Standard shell-scripting interpreters, including `sh`, `bash`, `perl`, and `python`.

The designer MAY assume that all of the above are already available on the appropriate `PATH` and `CLASSPATH` variables.

Any other programs, support libraries, or configurations necessary to run/test the submitted code MUST be approved in advance and documented in the submission. All submissions must be capable of being compiled and run standalone. In particular, the submissions MUST NOT assume that the code is being loaded and run in `Eclipse`, `NetBeans`, or any other IDE. The submission also MUST NOT assume that any instructor code *other* than the Support Code `.jar` file is available at runtime.

8.1 Turnin Process

All milestones and the final rollout are to be packaged in SUBMISSION ARCHIVEs (Sec 6.2) and submitted via the `turnin` program on the UNM CS department's computer systems. Students MAY log in to any of the `moons.cs.unm.edu` machines to do the turn in.

The syntax of the `turnin` command is:

```
turnin ASSIGNMENT-NAME file
```

where `ASSIGNMENT-NAME` is one of the following strings:

Deliverable	`ASSIGNMENT-NAME`
Milestone 1	`cs351.p1m1`
Milestone 2	`cs351.p1m2`
Milestone 3	`cs351.p1m3`
Rollout	`cs351.p1r`

and `file` is the SUBMISSION ARCHIVE to be submitted.

Students MAY use the `-ls` option with `turnin` to verify that files have been submitted properly.

Later `turnin` submissions to a given `ASSIGNMENT-NAME` will overwrite previous submissions, up to the deliverable deadline. Thereafter, `turnin` will create new archives for "late day" submissions, so students can turn in revised versions of their projects.

The delivery date/time of a submission will be the timestamp on the file as it appears in the `turnin` database archive. A wise hacker will allow padding time when submitting a file to compensate for clock skew, `turnin` glitches, power outages, etc.

NOTE: The file to be turned in MUST be world readable (mode `o+r`) when `turnin` is run on it, `turnin` MUST be run from the directory in which that file resides, and that directory MUST be world readable and executable (mode `o+rx`). Students are still responsible for exercising caution with their files and not allowing them to be copied or read by others. See Question 5.6 of the CS351 FAQ for suggested strategies.

8.2 Milestone 1: `JUnit` Test Suites

The first project component due is a set of `JUnit` tests for the `MovieGraph <T>` and `MovieGraphAnalyzer <T>` classes, implementing the `Graph<T>` and `GraphAnalyzer<T>`, respectively. These tests MUST cover all methods in both classes and include both reasonable, unreasonable, and erroneous inputs. Edge cases and corner cases MUST also be examined. Every test MUST be fully documented and describe what is being tested, how, and expected behavior. For Milestone 1, it is *not* necessary to provide tests of asymptotic time and space performance.

Note that implementations for `MovieGraph` and `MovieGraphAnalyzer` are NOT required for this milestone.

The deliverables for this milestone are:

`TestMovieGraph.java` The `JUnit` test suite for the `MovieGraph` class.

`TestGraphAnalyzer.java` The `JUnit` test suite for the `MovieGraphAnalyzer` class.

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

API documentation The handin MUST also include the full, compiled Javadoc documentation for all Java source files. This documentation MUST include full descriptions of every public or protected method, field, sub-class, enclosed class, or constructor employed by any of these Java files. This documentation MUST be included in the `api-doc/` directory of the SUBMISSION ARCHIVE, as described in Section 6.2.

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. This file is considered to be part of the USER DOCUMENTATION and should be placed in the `doc/` directory, as described in Section 6.2.

`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. This file is considered to be part of the USER DOCUMENTATION and should be placed in the `doc/` directory, as described in Section 6.2..

All materials for Milestone 1 MUST be packaged in a SUBMISSION ARCHIVE (see Section 6.2 for details and layout) and submitted according to the `turnin` process (Section 8.1).

8.3 Milestone 2: `MovieGraph` and `MovieGraphAnalyzer` Classes

The second component of the project is the complete implementation of the `MovieGraph` and `MovieGraphAnalyzer` classes. These MUST be complete and functional Java classes that `implement` the `Graph<T>` and `GraphAnalyzer<T>` interfaces, respectively. These classes MUST implement all methods specified in their respective interfaces and MUST support all non-method functionality described in the interface documentation (e.g., serializability). Further, these classes MUST meet all specified asymptotic time and space performance bounds.

These classes MAY, in addition, implement additional methods or functionalities not specified by the interfaces. Any such functionality MUST, however, be fully documented in INTERNAL DOCUMENTATION.

The deliverables for this milestone are:

`MovieGraph.java` The Java code file for the `Graph<T>` implementation.

`MovieGraphAnalyzer.java` The Java code file for the `GraphAnalyzer` implementation.

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

`JUnit` source files Test code files for all of the above. This set of files MUST contain at least the tests submitted in Milestone 1, but it MAY contain additional tests. The original tests MAY be revised to improve them or make them more complete.

API documentation The handin MUST also include the full, compiled Javadoc documentation for all Java source files. This documentation MUST include full descriptions of every public or protected method, field, sub-class, enclosed class, or constructor employed by any of these Java files. This documentation MUST be included in the `api-doc/` directory of the SUBMISSION ARCHIVE, as described in Section 6.2.

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

`BUGS.TXT` As in Section 8.2.

`README.TXT` As in Section 8.2.

`CHANGELOG.TXT` A concise list of all substantial changes to Milestone 1 deliverables that have been made by this milestone. This file is considered part of USER DOCUMENTATION and should be stored in the `doc/` directory, as described in Section 6.2.

All materials for Milestone 2 MUST be packaged in a SUBMISSION ARCHIVE (see Section 6.2 for details and layout) and submitted according to the `turnin` process (Section 8.1).

8.4 Milestone 3: Lexical Analysis and Parsing

The third milestone includes the lexical analyzer `BaconLexer`, implementing `LexicalAnalyzer`, and the three parser classes, `BaconParseActress`, `BaconParseActor`, and `BaconParseDirector`, implementing the `Parser` interface. These classes MUST implement all functionality specified in the `LexicalAnalyzer` and `Parser` interfaces.

The implemented `BaconLexer` and `BaconParse*` classes MAY, in addition, implement additional methods or functionalities not specified by their respective interfaces. Any such functionality MUST, however, be fully documented in INTERNAL DOCUMENTATION.

This milestone also includes unit tests and necessary test data for these classes. The DEVELOPER MUST provide a thorough suite of unit tests demonstrating correct functionality of the four required classes. Note that there is not a separate milestone for the unit tests, so the development schedule for them is up to the DEVELOPER, but they MUST be submitted with this milestone.

The exact test data is up to the DEVELOPER, but it MUST thoroughly test all four required classes for this milestone. For the lexical analyzer, the DEVELOPER MAY test any token sets that she or he wishes, but the milestone MUST include tests of at least one of the token sets described in Appendix A as well as at least one token set not specified in this document. The novel token set MUST include at least ten distinct token types.
The deliverables for this milestone are:

`BaconLexer.java` The Java code file for the `BaconLexer` implementation.

`BaconParseActress.java`, `BaconParseActor.java`, `BaconParseDirector.java` The Java code files for the various parser implementations.

Other Java source files Any other supporting code files necessary to compile, load, and use the `BaconLexer.java` and `BaconParse*.java` files. Also included are any token type `enum`s necessary for the unit tests.

`JUnit` source files Test code files for all of the above.

Test data All data necessary to run the unit tests. Test data MUST be placed in the `data/` directory, as specified in Section 6.2.

API documentation The handin MUST also include the full, compiled Javadoc documentation for all Java source files. This documentation MUST include full descriptions of every public or protected method, field, sub-class, enclosed class, or constructor employed by any of these Java files. This documentation MUST be included in the `api-doc/` directory of the SUBMISSION ARCHIVE, as described in Section 6.2.

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

`BUGS.TXT` As in Section 8.2.

`README.TXT` As in Section 8.2.

`CHANGELOG.TXT` A concise list of all substantial changes to Milestone 1–2 deliverables that have been made by this milestone, as in Section 8.3.

All materials for Milestone 3 MUST be packaged in a SUBMISSION ARCHIVE (see Section 6.2 for details and layout) and submitted according to the `turnin` process (Section 8.1).

8.5 Final Rollout: Full `Baconator` Suite

The final delivery of this project is the complete implementation of the the `Baconator` suite, including the CLI (Section 6.8), tests for the CLI, all previously developed code and tests, and full USER DOCUMENTATION and INTERNAL DOCUMENTATION for `Baconator`.

At this milestone, the DEVELOPER MUST also provide integration tests of the complete functionality of the `Baconator` suite. These tests MUST demonstrate all required behaviors as well as the suite's response to erroneous or improperly formatted data and user input. The exact form of these tests is at the DEVELOPER's discretion, but they MUST demonstrate comprehensive testing of functionality and robustness. They SHOULD include as many automated tests as possible (e.g., via scripting or test interfaces), but a small number of manual tests is acceptable. For manual tests, all test inputs and expected outputs MUST be documented, and a log of test executions MUST be provided.

If the designer has made changes to code already submitted in Milestones 1–3, those changes MUST be documented in the `CHANGELOG.TXT` file.
The deliverables for this milestone are:

`Baconator.java` The Java source code for the `Baconator` program.

Other Java source files Any other supporting code files necessary to compile, load, and use the complete `Baconator` suite.

`JUnit` source files Test code files for all of the above. This set of files MUST contain the tests submitted in Milestones 1–3, as well as tests for the new code developed for Rollout. It MAY also contain additional tests for Milestones 1–3 code.

Integration tests Automated and manual tests of the complete, integrated `Baconator` suite and Java and/or script code supporting those tests.

API documentation The handin MUST also include the full, compiled Javadoc documentation for all Java source files. This documentation MUST include full descriptions of every public or protected method, field, sub-class, enclosed class, or constructor employed by any of these Java files. This documentation MUST be included in the `api-doc/` directory of the SUBMISSION ARCHIVE, as described in Section 6.2.

`Baconator` suite USER DOCUMENTATION Documentation on how to compile, install, and run the `Baconator` program and any additional programs provided by the DEVELOPER. This MUST include full descriptions of all implemented capabilities

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

`BUGS.TXT` As in Section 8.2.

`README.TXT` As in Section 8.2.

`CHANGELOG.TXT` A concise list of all substantial changes to Milestone 1–3 deliverables that have been made by this milestone, as in Section 8.3.

All materials for Rollout MUST be packaged in a SUBMISSION ARCHIVE (see Section 6.2 for details and layout) and submitted according to the `turnin` process (Section 8.1).

9 Due Dates

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

Aug 23, 11:00 AM, MDT P1 specification v. 1.0 handed out.

Sep 1, 10:00 AM, MDT Milestone 1 (`JUnit` tests) due. Crit.

Sep 8, 10:00 AM, MDT Milestone 2 (`MovieGraph` and `MovieGraphAnalyzer`) due. Crit.

Sep 15, 10:00 AM, MDT Milestone (`BaconLexer` and `BaconParse*`) due. Crit.

Sep 22, 10:00 AM, MDT Final Rollout (full `Baconator` suite) due. Crit.

Appendix A: Files Grammars

A.1 IMDB File Grammars

`Baconator` MUST be able to parse at least three different files: `actresses.list`, `actors.list`, and `directors.list`. The format of those files is given by the following grammar:

```
<actress_file>      := <actress_header> <person_record>* <actress_footer>
<actor_file>        := <actor_header> <person_record>* <actor_footer>
<director_file>     := <director_header> <person_record>* <director_footer>
<person_record>     := <name> TAB+ <credit> BARE_WORD* NEWLINE
                       (TAB+ <credit> BARE_WORD* NEWLINE)*
<name>              := (PAREN_STRING|BARE_WORD)+
<credit>            := (PAREN_STRING|BARE_WORD)+ YEAR
```

The format of the `header` and `footer` fields is not specified and is up to the DEVELOPER to identify and document. *Hint:* examine the last lines before the data in each file, and the first lines following the data, and consider the use of the `consumeUntil` methods.

Terminals:

BARE_WORD A sequence of one or more non-whitespace characters that is not also any of the following token types.

NEWLINE Either the character '`\n`' or the character '`\r`'.

PAREN_STRING An open parenthesis character, followed by zero or more characters (other than close parenthesis), followed by a close parenthesis character.

TAB The single character '`\t`'

YEAR An open parenthesis followed by a four-digit number, followed by a close parenthesis. Note that a YEAR is a specific type of PAREN_STRING.

Notes

1. The YEAR field *is* considered to be part of a MOVIE name.
2. Any identity qualifiers, such as "`(I)`", "`(IV)`", etc., *are* considered to be part of people names.

A.2 Command Line Interpreter Grammar

The `Baconator` CLI (Section 6.8) MUST support at least the following grammar. It MAY support a superset of this language, at the DEVELOPER's discretion, but any such additional constructs MUST be documented in the USER DOCUMENTATION.

```
<command_line>   := <command> <argument>* NEWLINE
<command>        := ( HELP | STATS | DIAMETER | LOAD | SAVE |
                    PWD | CD | LS | PATH_LENGTH | PATH | SPHERE )
<argument>       := ( BRACE_STRING | BARE_WORD )
```

Terminals: Whitespace, other than NEWLINEs or whitespace occurring within a `BRACE_STRING`, is not syntactic in the CLI grammar, and should be discarded. (I.e., whitespace serves only to delimit tokens and commands, except when it occurs within a brace string, in which case it is preserved verbatim.)

The `<command>` terminals (`HELP`, `STATS`, etc.) correspond to the literal string command names given in Section 6.8 ("`help`", "`stats`", etc.).

The other terminals are as follows:

BARE_WORD A sequence of one or more non-whitespace characters that is not also any of the following token types.

BRACE_STRING An open curly-brace character ('{') followed by zero or more characters that are not close-braces or NEWLINEs, followed by a close-curly brace character ('}').

NEWLINE Either the character '`\n`' or the character '`\r`'.

Appendix B: The Floyd-Warshall Algorithm

A key step in a number of graph algorithms is computing the all-pairs shortest path distance. That is, for each pair of nodes in the graph, find the minimum number of edges that lead from the first node to the second. Clearly, for a graph with $|V|$ nodes, the output of this function will be a $|V|^2$ numbers — one for each pair. We can think of this as $|V| \times |V|$ matrix, d, where the entry d_{ij} gives the shortest path distance starting at node i and ending at node j. Note that we are only interested in the distance itself, at this point — we are not concerned with the paths themselves. That is, we care about what the distance between i and j is, but we don't care how you actually get between them.

For those who are familiar with Dijkstra's single-source shortest paths algorithm, it might appear that a solution to this problem is to run Dijkstra's algorithm once from each node in the graph. This will, in fact, work, but it is not the most efficient solution in general. Instead, we will use an algorithm designed to solve the all-pairs shortest paths problem: the *Floyd-Warshall* algorithm.

The Floyd-Warshall ultimately returns the square matrix, d, as described above. It begins by initializing it as follows:

$$d_{ij} = \begin{cases} 0 & \text{if} \quad i == j \\ 1 & \text{if} \quad (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$$

After initialization, it runs the analysis given in Algorithm 1. Clearly, this runs in $\Theta(V^3)$ time. More efficient algorithms for the all-pairs shortest paths algorithms exist (especially for sparse graphs, as we expect the IMDB GRAPH to be), but they are *much* more complicated to implement. While this algorithm is not at all complicated to implement, understanding why it works takes a bit more work. Those interested in the mechanics of why this works, and a proof of what it is doing, should refer to a standard algorithms textbook such as Cormen, Leiserson, & Rivest.

Algorithm 1 The Floyd-Warshall all-pairs shortest paths algorithm

```
$n \leftarrow |V|$
for $k = 1$ to $n$ do
  for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
      $d_{ij} \leftarrow \min(d_{ij}, d_{ik} + d_{kj})$
    end for
  end for
end for
return $d$
```

Note a number of important features of this algorithm:

- It is not defined for empty graphs. Behavior on an empty graph is implementation-dependent.
- If node j is not reachable from node i, then $d_{ij} = \infty$ at termination.
- $d_{ii} = 0$ for all nodes $i \in V$.

Appendix C: Single-Source Graph Searches

While the Floyd-Warshall Algorithm is useful for *all pairs* shortest-paths searches, for graphs of even medium size (say, a few thousand nodes), its cubic runtime becomes impractical. When we are interested in a more limited result – say, the shortest distance/shortest path between just two nodes, then it is possible to be much more efficient. For example, you need not touch all the nodes/edges of the graph – when the "destination" is found, the search can stop.

For the case of unweighted graphs (as in IMDB), this leads to two particularly simple graph search algorithms: breadth-first search (BFS; Algorithm 2) and depth-first search (DFS; Algorithm 3).

BFS is well-suited to finding shortest paths: in unweighted graphs (absent additional information) it is an optimal algorithm for finding a shortest path between two nodes. It is guaranteed that the first time BFS visits ("opens") a node n, that it has found the shortest path to that n.

Algorithm 2 The Breadth-First Search (BFS) Algorithm

```
Input: Graph $G = \langle V, E \rangle$; start node from $\in V$; end node to $\in V$
OPEN $\leftarrow$ from
CLOSED $\leftarrow$ from
$done \leftarrow$ false
while $!done$ AND !isEmpty(OPEN) do
    $here \leftarrow$ OPEN.removeFromFront()
    for $n \in G$.neighborSet($here$) do
        if $n \notin$ CLOSED then
            CLOSED.add($n$)
            OPEN.addToEnd($n$)
            if $n ==$ to then
                $done \leftarrow true$
            end if
        end if
    end for
end while
```

DFS is not good for finding shortest paths, but it has other uses, such as detecting cycles, finding connected components, or for solving certain games. In such problems, the termination condition is usually not "found the terminal node," but we express it that way here for symmetry with the BFS description.

Important notes on these algorithms:

- DFS and BFS differ only by the semantics of the "OPEN list" data structure – whether it is treated as FIFO (queue-like) or LIFO (stack-like).
- In worst case, both BFS and DFS touch every node and every edge in the graph, leading to $O(V + E)$ runtime.
- In worst case, both BFS and DFS have to record every node in the graph in the CLOSED set, leading to $O(V)$ space use.

Algorithm 3 The Depth-First Search (DFS) Algorithm

```
Input: Graph $G = \langle V, E \rangle$; start node from $\in V$; end node to $\in V$
OPEN $\leftarrow$ from
CLOSED $\leftarrow$ from
$done \leftarrow$ false
while $!done$ AND !isEmpty(OPEN) do
    $here \leftarrow$ OPEN.removeFromFront()
    for $n \in G$.neighborSet($here$) do
        if $n \notin$ CLOSED then
            CLOSED.add($n$)
            OPEN.addToFront($n$)
            if $n ==$ to then
                $done \leftarrow true$
            end if
        end if
    end for
end while
```

- In practice, both will halt when they reach a terminal condition, such as finding a target node.
- BFS is optimal for finding shortest paths only for *unweighted* graphs. For graphs with edge-weights (e.g., a road distance graph), a more general algorithm, such as Dijkstra's algorithm or A* is more appropriate.