

Maze Documentation

I. User Manual

Maze is a small Java command line application which populates a graph data structure from a textual description.

Maze is primarily of interest to developers and students and is often used as an embedded library. Maze is delivered as source code. See the Developer documentation for more technical details.

Installation

Since Maze is supplied only as source code, there is no installer. Maze is distributed as a single zip archive file. The name of this file may differ between providers and versions. If you cannot determine the appropriate file to download, please contact your provider.

If you are a developer, you may prefer to follow the installation advice in the Developer Documentation and add the source to a project in your IDE.

To install the application as a user:

1. Download the zip file into a folder (directory) of your choice.
2. Use your local system zip utility to unpack the file. Depending on your operating, this utility may be named unzip, gunzip, zip, 7-zip, or something similar. On many systems you can simply right click on the file in your file browser and select “unzip” or “extract here”.
3. Once you have unpacked the archive, you should browse into the new file hierarchy. In the first two or three levels you should find a folder named “src”.
4. When you have found this “src” directory, open a command line shell and cd into the src directory. (If this operation is unfamiliar, ask a developer friend for help.
5. You now need to compile the code. From the command-line type:

```
javac -classpath . edu/iastate/cs228/hw5/shared/Maze.java
```

NOTE: your system may require “\” in place of “/”.

6. If the compile operation completes without any messages indicating an error, then you should now have an executable version of MAZE. To test your new installation, type the command:

```
java edu/iastate/cs228/hw5/shared/Maze.
```

You should see the following “demo” response:

```
Warning: no file parameter, proceeding with default
file../mazes/maze.10.10.50.txt
findPath was called with start=0, dest=9
0: (0, 7: 2)
1: (1, 5: 5)
2:
3: (3, 2: 3) (3, 3: 3)
4:
5: (5, 3: 0)
6:
7: (7, 1: 9) (7, 9: 8)
8:
9:
```

Note: if you have received a Version 2 pre-release, the output may also include these lines, generated by a temporary stub of the currently under development shortest path feature.

```
Now you need to give it a real implementation.
Path length: No Path
Path cost:    No Path
Path No Path
```

Note: Maze can be easier to use from the command-line if you find the file “Maze.java” at the location referenced in the earlier “javac” command and copy it to the “src” directory and then recompile with:

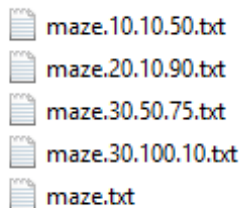
```
javac -classpath . Maze.java
```

Once you have done this, you will be able to invoke maze from the src directory using only

```
Maze <filename>
```

Building the Demo Graphs

Maze is shipped with a small number of demonstration graph description files. You can find these graphs in “mazes” directory, which should be located in src’s parent directory. Maze is shipped with these demonstration files:



```
maze.10.10.50.txt
maze.20.10.90.txt
maze.30.50.75.txt
maze.30.100.10.txt
maze.txt
```

To build one of these graphs (from a command shell positioned at “src”), type a command of the form:

```
java Maze <file name>
```

For example to run the maze.10.10.50.txt file use the command:

```
java Maze maze.10.10.50.txt
```

This command should generate a report similar to the “demonstration” report, but without the warning about a missing file parameter.

Creating New Graph Descriptions

Maze graph descriptions are simple text files, thus you can create your own mazes in any simple text editor. (Maze graph descriptions *must be* .txt files. Maze does not understand word processor or document files, such as .doc or .pdf.)

A Maze graph description consists of the graph vertex count on the first line, followed by individual vertex descriptions, one per line. For example, the Maze description for the graph in maze.10.10.50.txt is:

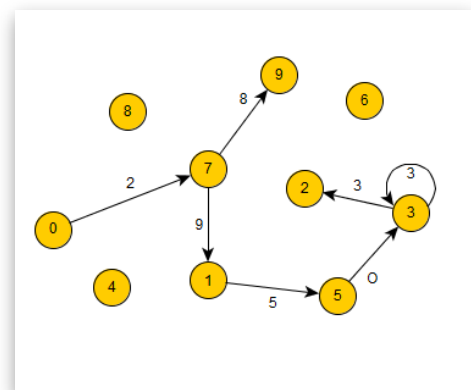
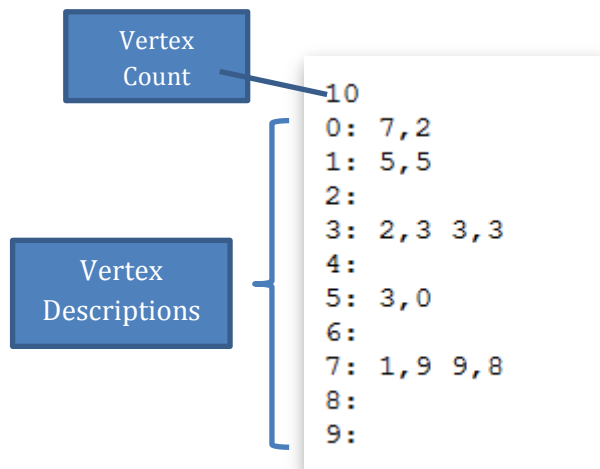


Figure 1 - the graph described by demonstration file maze.10.10.50.txt

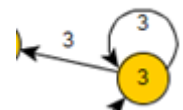
Each vertex description has this structure:

<vertex identifier> : <edge description> [<edge description> ... <edge description>]

each edge description is of the form

<destination vertex identifier> <edge weight>

Thus, the line “3: 2,3, 3,3” from above describes this part of the graph in Figure 1:



Graph descriptions must have as many vertex descriptions as there are lines and the vertices must appear in order by their identifying number.

NOTE: if you are upgrading Maze from an earlier version, you may need to revise your existing graph description files. Earlier versions allowed vertex positions to be skipped and did not require a vertex count at the beginning.

If You Need Help

Check our discussion board. Other users have probably encountered similar problems, so there is a good chance you will find the answer there. If you don't find an answer to your question, you can always pose a new question on the discussion board.

If you need additional support, please contact the provider from whom you acquired the zip.

II. Developer Notes

Overview

Maze constructs adjacency list graph representations from simple textual descriptions. The resulting graph is an instance of generic class `Graph<E>`, where the type `E` is a client programmer chosen object attached to a graph edge *after* construction.

The primary reusable classes in Maze are:

- **Maze**: houses the application's command-line user interface.
- **MazeBuilder**: a Graph factory. It processes textual input and produces a corresponding Graph instance. MazeBuilder includes a utility method to locate start and end nodes in a graph.
- **Graph**: an adjacency list representation of a directed, weighted graph.
- **Heap**: a generic heap implemented using Java ArrayList.
- **LinkedStack**: a generic Stack, implemented as a proprietary singly-linked list.

Maze also defines these interfaces:

- **BareG, BareV, BareE**: graph access interfaces that hide Graph's generic<E> edge object. These interfaces simplify the code of graph algorithms that aren't interested in the generic edge-related information.
- **PureStack<E>** a classic stack interface.
- **PurePriorityQueue<E>** a classic Min-Priority interface

See the source or the Javadoc for API details.

Architecture and Design Notes

- Maze stores vertices in an ArrayList, and uses a vertex's index in that list as the vertex's "natural key". To retrieve a vertex, you must know its integer identifier.
- Since Maze is typically used as a library, we keep all of its code in a "shared" package. You should not create code in any other non-test package without specific clearance from the engineering manager or technical lead.
- The new Pathfinder solution should access the graph *only* via the Bare interfaces.

Identified Technical Debt:

- We need a way to tell Maze where to look for the Mazes directory.
- we should move `getEndVertex()` from MazeBuilder to Graph. The method is tightly coupled to the graph definition and could be useful, even in applications that don't build their graph using the Maze file format.
- `getEndVertex()` would be better as `getFirst()` and `getLast()`.
- The public `lastCost` member in Pathfinder is a shameless hack. It could be avoided if `findPath()` returned a proper Path object.
- Since Graph is already dependent upon ArrayList (meaning the code already required JCF), it is unclear what benefit derives from having proprietary Heap and Stack implementations. We should plan to replace these with standard implementations.

- The use of EE to quiet the compiler seems an awkward hack. We should at least look for a way to hide it, perhaps by having Graph implement method overrides that only use the bare interfaces?
- We need to add a proper logging implementation to support dynamic control of debug trace and field error reporting.
- While using a linear index to identify a Graph vertex is simple and efficient, it requires many applications for which the integer is *not* a natural key to add a Map or other mechanism to map the client's preferred vertex identifier to and from the Graph identifier. We should consider absorbing responsibility for this mapping. A subclass of Graph could expose an api that took Objects as the vertex key and internally mapped that to the appropriate integer when new vertices were created.
- We should develop a set of graph descriptions for use in the autoTests that are designed to assure conditional coverage in the graph construction.

Version and other Historical Information

- To satisfy needs of Explore, in version 2, we made Graph generic on an edge information type<E>. The E type parameter can be used to attach arbitrary extra information to a graph edge. Unfortunately this change caused compatibility issues with Maze and added significant syntactic ugliness for applications that don't need the generic capability.

Rather than force all applications to be generic aware (and to reduce future compatibility issues), we further modified Graph to export a set of non-generic access interfaces. These "Bare" interfaces support most common graph operations while hiding the details of the generic edge object capability.

- In Version 2, graph description files must begin with a vertex count.
- To support applications that need to be able to reliably map from other coordinate systems to the implicit integer identifier of Maze components, we changed the Vertex construction policy. Whereas earlier versions allowed slots in the graph vertex to be 'unused' (by assigning null to the element), starting with Version 2, all slots in within the range of vertex identifiers used in the input file, will have a vertex object assigned. This may lead to the creation of "phantom" disconnected vertices in some client applications.
- Client programmers are free to subtype the Vertex class and add a mark to hide unwanted Vertices.

Code Details

- The new shortest path feature is currently stubbed. All of the wiring for PathFinder.findPath() is complete, but findPath() doesn't do anything except print out it's calling parameters. Unfortunately, this message goes to stdout and corrupts normal output in the beta release. (Yet another reason we need a proper logging framework.)

The existing drivers and auto tests may need to be updated before release to support and cover the shortest path feature .

- The file handling code assumes it can find the mazes directory relative to the current working directory. Since different IDEs and different launch configurations may use different policies for setting the CWD, this assumption may not be valid in your IDE. The

code at the beginning of `main()` looks in two different places for mazes. This is typically adequate for Eclipse and command-line environments, but may not be adequate for your particular IDE. If you find it necessary to look elsewhere for your graph files.

The Bare Interfaces

The “Bare” Graph Interfaces: Graph implements three minimal “Bare” interfaces. These interfaces expose everything you need to know about Graph to be able to find a solution. The Interfaces are:

```
public interface BareG{  
    // does vertex exist in the graph?  
    boolean checkVertex(BareV vertex);  
    /** retrieve a vertex using its index  
     * (or linear identity). Returns null  
     * if the index is outOfBounds.  
     */  
    BareV getVertex(int index);  
}
```

`BareG`'s `getVertex()` is your primary access to the graph. Since the vertices are stored in an `ArrayList`, if you need to step through all the vertices, you can just loop through the integers until `getVertex()` returns `null`.

```
public interface BareV {  
    //returns an iterable collection of  
    //edges corresponding to this node's neighbors.  
    Iterable<BareE> getBareEdges();  
  
    //returns the index (or linear identity)  
    //of this node.  
    int getIndex();  
}
```

`BareV`, a vertex, exposes the unique integer identifier associated with this vertex and an iterator you can use to explore its edges. (Note: you can wrap the integer identifier in an `Integer` object and use it as a key to index the vertex in a map: `Map<Integer, BareV>`.)

```
public interface BareE {  
    /** returns the neighbor (to) vertex  
     * associated with this edge. */  
    BareV getToVertex();  
  
    /** The weight associated with this edge. */  
    int getWeight();  
}
```

`BareE` is an edge. Each `BareE` represents a connection *from* the vertex where it was discovered (via `getBareEdges()`), *toward* the vertex available via `getToVertex()`. `BareE`'s `getWeight()` returns the cost associated with this edge.

The New Solver

`PathFinder` houses a single public method, for which you must supply the implementation.

```
public static List<Integer> findPath(BareG g, BareV source, BareV dest) {
```

which should use Dijkstra's method to find the least cost path in g from *source* to *dest*. It returns a list of Integers corresponding to the vertex indexes, in order from *source* to *dest*. There is additional documentation in the template Javadoc. The new solver must not reference Graph. Instead it should use only the Bare interfaces.

Developer Infrastructure

- Drivers: See the drivers package in the test source folder. Presently there is one driver, which can be switched to read different files by editing the value assigned to "selected".
- Automated Tests: See the autoTests package in the test source folder. Presently there is a parameterized test that runs the builder against all demo files.
- Installation: Easiest is to create a new project in your IDE, then copy the src and test hierarchies from the unpacked archive into the appropriate place in the new project. Because Maze can be invoked with zero arguments, you can test a new installation without a run configuration (at least in Eclipse). Just right-click on Maze (in project explorer) and select "run as application."
- Build: The application does not require any extra libraries or configuration.

III. Background Material

Extracting source code from jar files.

Jar files *are* zip files. They can be created and unpacked using any standard zip utility. If your zip utility doesn't like the .jar extension, you can just rename it to .zip and then unpack it.

Jar files use a different file extension as a way of declaring that their content conforms to Java standard conventions for file hierarchy. One of these conventions is that source files are organized in directories corresponding to the package name. However, jar files can also contain “special” files that contain possibly extensive meta information about the program, about how it is to be deployed, about who created it, etc.

Running main() directly from Eclipse

Applications that do not require arguments at the command line can be invoked *without a run configuration* by selecting the main() function, right-clicking, and selecting “run as application.” Doing this actually creates a default run configuration, so you can rerun the application by selecting



one of these icons in the main eclipse toolbar.

Technical Debt

Technical debt refers to code features that imply an unnecessary, on-going carrying cost -- just like financial debts impose an ongoing interest-based carrying cost. Generally if such code could be revised, future maintenance would involve less effort.

Developers are usually aware of decisions they make that create technical debt, just as one is aware when they have taken a loan or created new credit card debt. In both cases, the decision to assume the debt is a tradeoff between short term resources and outcomes (like time, staff, money, expertise, and meeting a near deadline) and longer-term costs and outcomes.