

Assignment 5: Dijkstra's Algorithm for Finding Least-Cost Paths

Due at 11:50 pm on Friday, April 21

Introduction

For this project you will implement Dijkstra's shortest path (least cost) algorithm and use it with two different applications:

- **Maze:** a text-based application from last year's HW5, which constructs a graph from a textual description and then invokes your solver to find the shortest path from start to finish.
- **Explore:** a hex-grid terrain map (with provided viewer). The same solver that works with Maze will also work with Explore. To complete Explore, in addition to the solver, you will need to complete a class which loads information from a simple, textual terrain description file. We supply some special classes to simplify this effort.

We supply a complete implementation of $\text{Graph}\langle E \rangle$, which is used by both applications. Graph is a typical adjacency list representation, which can have a generic data element attached to each edge. Maze includes a graph builder which can construct an instance of Graph from a simple text file. You can use Maze to reliably construct test graphs to use while developing and testing your solver.

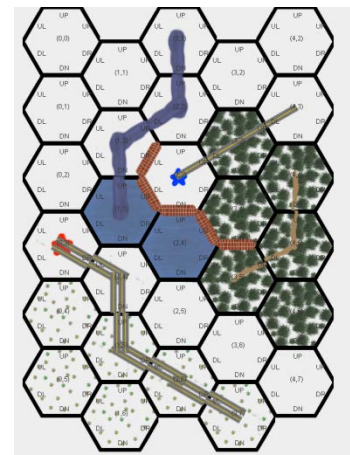


Figure 1 - Example terrain map before solving.

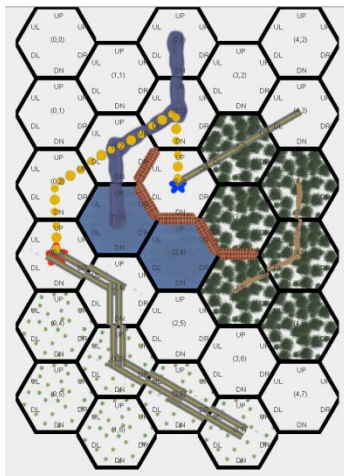


Figure 2 – The map with solution (the dotted yellow line.)

The Explore application uses a much more complex TerrainGraph to represent maps like Figure 1. However, because TerrainGraph is derived from Graph and, once built, exports the same simple interface for analysis as does Maze, you can use the exact same solver to produce shortest path solutions for both Maze and Explore!

You need to supply your solver and some input parsing 'glue' to complete Explore. We provide a high-level, simple-to-use construction methods on TerrainGraph so that you only need to parse a simple text file format and make appropriate calls to TerrainGraph to create a map for display in Explore. Once that map is constructed, you can access its base Graph "personality" with your solver and then display,

graphically, both the full-terrain map along with your solution path, as in Figure 2.

Suggested Work Plan

You should approach this project as five distinct activities:

1. Get Maze running without a solver.
2. Build a solver using Maze as your development harness.
3. Complete the TerrainLoader for Explore. Use the supplied Junit driver as your development harness) while working on TerrainLoader.
4. Attach your solver and the TerrainViewer.
5. Copy the working guts from the Junit driver and add some code to create a main() for Explore.

The following sections will give background information, requirements, and guidance for each of these steps.

Maze

Maze consists of these classes and interfaces:

Maze: the main app. This class interacts with the command line (via a single filename argument), takes responsibility for handling file-related exceptions, constructs a special Reader from the file input stream, creates an instance of the app and then invokes the components which will use input from the Reader to build a graph, use the graph to compute a solution, and print the result.

MazeBuilder: the input parser. MazeBuilder.buildGraph() reads the input file and constructs the graph described there.

Graph: the main graph. This is a simple, adjacency list representation of a weighted graph. All vertices are kept in a single ArrayList and are identified by their index within the list. Each Vertex owns a set of Edges, where it keeps references to its neighbors. MazeBuilder will create this structure for you and while computing a solution, you will interact with it *only* through the “Bare” graph interfaces: BareG, BareE, and BareV (see below.)

PathFinder: This class house a single method *findPath()* which you must complete and which must use Dijkstra’s algorithm. To simplify your work, we supply **Heap** which you can use as a priority queue, and **LinkedStack** (handy when recovering the solution path.)

The “Bare” Graph Interfaces: To help you focus only on what matters, 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 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 Maze Input File Format

Graph input for Maze consists of a Vertex_Count (a single integer) followed by an arbitrary number of Vertex descriptions, where each description is of the form

```
vertex_id : [dest, weight]*
```

Vertex_ids must start with zero and be consecutive numbers. The example in Figure 4 is from the file *maze.10.10.50.txt*, which is supplied with the templates. The left column of (somewhat gray) numbers are editor supplied line numbers, *not* part of the file. The vertex count *must* appear in the first line. Line 2 of the example describes vertex zero, which is connected to one neighbor (vertex 7) by an edge with weight 2.

```
1 10
2 0: 7,2
3 1: 5,5
4 2:
5 3: 2,3 3,3
6 4:
7 5: 3,0
8 6:
9 7: 1,9 9,8
10 8:
11 9:
```

Figure 4 – A Maze graph input file.

Maze assumes it will find graph input files in the directory mazes. If you are working in eclipse, *mazes* will “live” at the very root of your project (at the same level as the src folder). If you are working at the command line, the relevant directory may be *inside* the src directory. You may need to experiment.

(Remember: to see directories like mazes in Eclipse, you will need to switch from package explorer to Navigator view.)

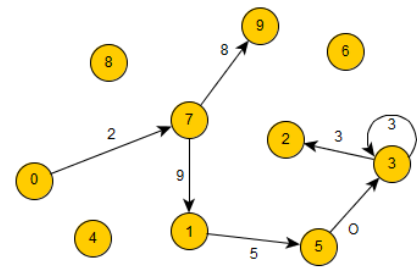


Figure 3 – A drawing of the graph represented by Figure 4

The temporary template implementation of *findpath()* just prints a message saying it was called. When run with no arguments, Maze will load the default file in Figure 4. Thus, if your installation is correct, you should be able to launch just by right-clicking on the class in the package explorer and selecting “run as application”. Thus, without changing any code, you should be able to make Maze generate a report showing the graph it built, and the “called with” report from *findpath()*, as in Figure 5.

The *findpath()* method will be called with references that satisfy the “Bare” interfaces and you should not need to cast these references or make any direct references to Graph, Vertex, or Edge. In fact, your life will definitely be easier if you do not. If you try to “reach into” Graph without going through the Bare interfaces, your solver implementation will most likely not work with Explore.

```
A graph representing a maze
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:

Find Path called with 0, 9
A shortest path in the maze
[]
```

Figure 5—The report Maze will generate with only template code.

Explore

Unlike Maze, Explore will not run until you supply code to read feature information from a terrain description file. As background for that task, you need to be familiar with:

- The coordinate systems (plural) that are used,
- The supported Terrain features,
- The input file format used to describe these features,
- The “construction methods” you should call in TerrainGraph,
- And the helper classes we have supplied to make this task easier.

Coordinates: getting from linear to hex

In Explore, Terrain features are described using a particular Hex coordinate system. (There are at least three different systems one could use, so don't assume you already know how this works.)

Explore uses a hex cell with horizontal top and bottom, so the coordinates are described by vertical lines down and (in our case), diagonal lines running right and up, as in Figure 6.

These are the coordinates you will use when describing terrain features to TerrainGraph. However, the vertices in Explore each have multiple identities. When mentioned in a feature description to TerrainGraph, we identify the vertex by it's hex coordinate. However, the solver you created for Maze doesn't know anything about hex coordinates, it wants to identify vertices by their “serial number” – i.e., by a linear 1-dimensional coordinate system.

This amounts to the same problem compiler developers encounter when trying to create two (or multiple) dimensional arrays in a linear memory system. Explore uses essentially the same solution. If we know the dimensions of our grid before creating the graph, then we can map sequential positions to the grid in row-major order, as in Figure 7. If we combine the solid red line segments in Figure 7 with the vertical green lines in the other figures, we have the basis for assigning rectangular coordinates to each cell. For example, linear cell 7 is also rectangular cell 2,1.

The rectangular coordinates are actually more convenient for computing screen display coordinates for each cell. They are also the most convenient starting point for computing a cell's hex coordinate. Thus, you can visualize what happens as we construct the Explore graph thus: all vertices are created left to right along a single line. That line is then ‘folded’ (or segmented) (based on the grid width) to create a stack of rows which are linked left and right and up and down.

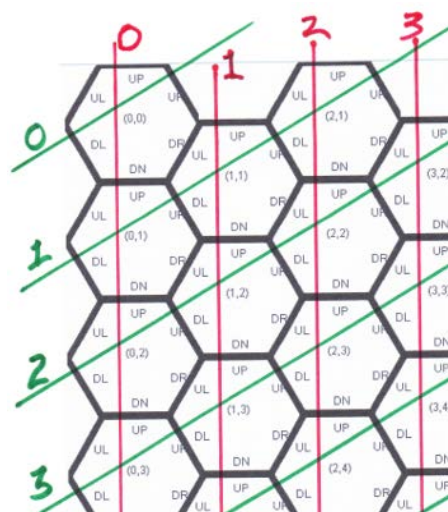


Figure 6 – The hex coordinate system used in Explore. Red lines represent x coordinates and green lines represent y coordinates.



Figure 7 – The red line shows a row-major scan through a 5-wide grid. The green numbers are the linear address associated with the cell.

That however, creates a grid where each cell/vertex has links to four neighbors. In a hex grid, there should be six links (neighbors) per cell.

If we linked neighbor cells up and down along the up-right diagonal (along the green lines in Figure 6), then we would have the requisite six links per cell – we just need to figure out how to compute the appropriate hex coordinate.

If you examine the hex coordinates for cells at the same rectangular y value, you'll see that the hex "y" changes at half the rate. The second coordinate of the cells in rectangular row 0 form the sequence 0, 1, 1, 2, 2, 3, 3 ... The hexY starts at the same value as the rectangular Y and then increases once for every two rectangular columns.

Class Coordinate

At various times we need to be able to reference a Vertex or a grid cell by each of these coordinate systems, but trying to keep track of which system produced a particular x,y pair and getting all the calculations to convert from one to another correct is tiring and error prone. Thus we have created a class named Coordinate that represents both the "identity" of the cell and its associated vertex, and knows how to produce the right value in any of the coordinate systems on demand.

A Coordinate object can be constructed from any of the graph coordinate values.

- `Coordinate(int i)` constructs a coordinate starting from a vertex's linear index.
- `Coordinate(int xHex, int yHex)` starts from a hex coordinate pair.
- `Coordinate(int x, int y, boolean true)` starts from a rectangular x,y pair.

Once created, a Coordinate instance can produce a convenient to use value in any of the systems. Single-valued (linear) coordinates are ints. Two-valued coordinates are returned as a Pair object which includes an external flag ('B', 'R', 'S') indicating whether it belongs to the Board (hex), Rectangular, or Screen coordinate system.

- `public Pair getBoard()` returns a hex pair,
- `public Pair getRect()` returns a rectangular pair, and
- `public int getLinear()` returns the linear value.

Coordinate objects also support various utility and convenience functions. For example:

- `Boolean isValid(Pair coord)` will tell you if the coordinate is within the grid or its graph.
- `Pair getDelta(Coordinate c)` computes the hex-valued difference between two coordinates.
- `Coordinate get(HexDir dir)` returns the coordinate of the board neighbor at direction dir.

Several functions test whether or not Coordinates and Pairs represent neighbors.

You MUST set the geometry!

Coordinate is also the static authority for board geometry. `Coordinate.setGeometry(cols, rows)` must be call before any coordinate operation are possible and before any vertices are created.

HexDir

In Explore, edges between vertices correspond to directions on the board, so it is convenient to have a way to identify, represent, and manipulate these directions. The complex enum HexDir

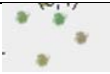




(complex because it supports many mapping and characterization functions) is an important companion to Pair and Coordinate. HexDir not only gives you convenient direction names and convenient iteration through directions, it also knows how to convert Pair instances (e.g., delta's between two Coordinates) into conveniently tested HexDir values. Of particular use in this project:

- `HexDir reverse()` returns the HexDir for 180 degrees from this direction.
- `HexDir fromAbbrev(String d)` returns the HexDir corresponding to the two letter border segment direction abbreviation (see the discussion of BorderSegments in the section on Explore's file format.



Terrain Features




Explore terrain maps are constructed from terrain types and connector types. These attributes are explicitly represented in Enum TerrainType and Enum ConnType. (In both cases, the enums also are authority for the feature weight and certain other attributes.

Each cell's "background" terrain can be one of these:

Name	Graphic	Represents	Weight
default	none	Relatively flat, easily traversed open land.	165
brush		Like default, but with low obstacles.	180
forest		Dense trees. Difficult for vehicular traffic	195
water		Open, relatively still water, such as a lake or pond. Not necessarily suitable for high speed craft.	190
flag1		The start cell	0
flag2		The finish point.	0

Additionally, Travel times can be improved between adjacent cells when certain connectors are present. Strings corresponding to each of the terrain and connector types are defined at the top of class TerrainLoader. You should use the constants defined there, when doing your work to complete TerrainLoader. The supported Connector types are:

Names	Graphic	Represents	Weight
dividedhwy, Hw4		An interstate or turnpike quality high speed trafficway.	110
hwy, Hw2		A two-lane, two-way (e.g., state or county) highway.	130

unpaved, dirt		An unpaved, dirt or gravel road or path	143
river		Moving water, but may only be navigable by certain craft.	185
barrierwall, wall		A barrier that would require explosives and subsequent clearing activity before vehicular traffic could pass.	900

While you are not responsible for setting these values (they are known implicitly when you add the feature to the map, you may need to know the weights to be able to confirm that the map exported by Explore is correct and to confirm that your solver is also handling Explore graphs appropriately.

The effective weight assigned to an edge (and visible to your solver) is computed thus:

- if the Connector between the two cells is a barrier, the weight for that edge is 900.
- if there is no connector between two cells (there will always be an edge in the graph your solver operates on, even if that edge doesn't have an associated terrain connector), the weight is the maximum of the terrain weight in the two cells. For example, an edge between forrest and brush would have an effective weight of 195.
- if there is a non-barrier connector between two cells, the edge weight will be $\min(\text{conn_weight}, \max(\text{cell1_terrainWeight}, \text{cell2_terrainWeight}))$.

You can see the effect of this computation on this diagnostic dump of the base graph for *terrain.txt*.

```

0: (0, 1: 165) (0, 5: 165)
1: (1, 2: 165) (1, 0: 165) (1, 7: 165) (1, 5: 165) (1, 6: 165)
2: (2, 7: 165) (2, 3: 165) (2, 1: 165)
3: (3, 7: 165) (3, 9: 165) (3, 2: 165) (3, 4: 165) (3, 8: 195)
4: (4, 9: 165) (4, 3: 165)
5: (5, 0: 165) (5, 6: 165) (5, 1: 165) (5, 10: 165)
6: (6, 10: 165) (6, 7: 165) (6, 11: 185) (6, 12: 900) (6, 1: 165) (6, 5: 165)
7: (7, 8: 195) (7, 12: 165) (7, 2: 165) (7, 3: 165) (7, 1: 165) (7, 6: 165)
8: (8, 7: 195) (8, 9: 130) (8, 12: 130) (8, 13: 195) (8, 3: 195) (8, 14: 195)
9: (9, 8: 130) (9, 4: 165) (9, 14: 195) (9, 3: 165)
10: (10, 15: 165) (10, 6: 165) (10, 11: 190) (10, 5: 165)
11: (11, 16: 190) (11, 15: 190) (11, 17: 190) (11, 10: 190) (11, 12: 900) (11, 6: 185)
12: (12, 11: 900) (12, 17: 900) (12, 13: 195) (12, 6: 900) (12, 7: 165) (12, 8: 130)
13: (13, 8: 195) (13, 17: 900) (13, 14: 195) (13, 19: 195) (13, 12: 195) (13, 18: 900)
14: (14, 13: 195) (14, 9: 195) (14, 8: 195) (14, 19: 143)
15: (15, 20: 180) (15, 10: 165) (15, 11: 190) (15, 16: 110)
16: (16, 20: 180) (16, 22: 165) (16, 15: 110) (16, 11: 190) (16, 17: 190) (16, 21: 110)
17: (17, 16: 190) (17, 13: 900) (17, 12: 900) (17, 18: 195) (17, 22: 190) (17, 11: 190)
18: (18, 19: 143) (18, 24: 195) (18, 17: 195) (18, 13: 900) (18, 23: 195) (18, 22: 195)
19: (19, 13: 195) (19, 24: 195) (19, 18: 143) (19, 14: 143)
20: (20, 15: 180) (20, 21: 180) (20, 25: 180) (20, 16: 180)
21: (21, 20: 180) (21, 22: 180) (21, 27: 110) (21, 16: 110) (21, 25: 180) (21, 26: 180)
22: (22, 27: 180) (22, 17: 190) (22, 18: 195) (22, 23: 165) (22, 16: 165) (22, 21: 180)
23: (23, 29: 165) (23, 24: 195) (23, 27: 180) (23, 18: 195) (23, 22: 165) (23, 28: 180)
24: (24, 18: 195) (24, 19: 195) (24, 29: 195) (24, 23: 195)
25: (25, 21: 180) (25, 26: 180) (25, 20: 180)
26: (26, 25: 180) (26, 27: 180) (26, 21: 180)
27: (27, 22: 180) (27, 28: 110) (27, 21: 110) (27, 23: 180) (27, 26: 180)
28: (28, 23: 180) (28, 29: 180) (28, 27: 110)
29: (29, 24: 195) (29, 23: 165) (29, 28: 180)

```


This is generated on stdout once the terrain map is constructed. The format is vertexId: (edge) (edge) ... where each edge is (fromId, toId: weight).

Terrain File Format

Figure 8 shows an example terrain description file which includes an instance of all available features and connectors.

The general structure of the file is

<file>:: geometry <pair> <feature>+

<feature>:: <flags>?|<terrain>*|<connector>*| <barrier>*

<flags>:: <coord><coord>

<terrain> :: (water|brush|forest) <coord>+

<connector>::(hwy|dividedhwy|unpaved|river) <path>

<path> :: <start coord><adj coord>*<end coord>

<barrier>:: barrierwall <border segment>+

<border segment> :: <coord> <dir list>

Additional information:

- The file *must* start with a geometry section, so that the Coordinate geometry can be set before processing any terrain features.
- Geometry and flags may only appear once. Geometry is required. The flags section is optional. If it is not present, you should not attempt to run your solver.
- The cells listed in terrain sections do not need to be contiguous. However it might make it easier for you to check the file against your terrain plan if you use a separate section for each contiguous region. Two regions of forest would be described in two different “forest” sections.
- Each connector section must list at least two coordinates (a start and an end). Each pair of connectors in a section list must correspond to adjacent cells. Thus each continuous run of highway, for example, should be described in its own section and the coordinates should name the cells in the order they are crossed. Unconnected (and crossing) highways should each have their own section. The same path-related restrictions apply to all connector types except barrierwall.

Barriers are described by naming a cell and a set of cell sides to be blocked. The sides to be blocked are identified by their direction from the center of the cell. The barriers in Figure 9 are created by the border segment “2,4 up ur”. The order of the directions in the border segment is

not important. Since hex sides are usually shared by two cells, there are

```
geometry 5 6

flags
  2 3
  5 5

water 1 3 2 4

brush 0 4 0 5 1 5 1 6 2 6 3 7

forest 3 3 3 4 3 5 4 4 4 5 4 6

hwy 2 3 3 3 4 3

unpaved 3 5 4 5 4 4

dividedhwy 0 3 1 4 1 5 2 6 3 7

river 2 1 2 2 1 2 1 3

barrierwall
  2 3 ul dl dn
  3 4 dl dn
```

Figure 8 – An example terrain description file.

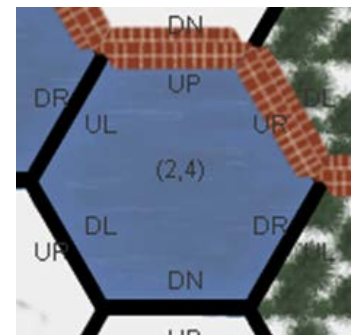


Figure 9 – a cell with barriers in two directions: up and ur.

always two ways to specify a particular segment of barrier. Choose whichever seems most convenient; both result in the same graph structure.

Parsing the Terrain File

We have added several classes to make it easier to parse the terrain file:

A specialized scanner, *TerrainScanner*, knows how to recognize not just Java types, but also Coordinates, direction abbreviations, and BorderSegments. Thus, once you have recognized the *dividedhwy* keyword, you can just loop over the coordinate elements like this:

```
Path path = new Path(c1, scan.nextCoord());
while (scan.hasNextCoord()){
    path.add(scan.nextCoord());
}
```

The same strategy works for the more complex BorderSegment elements:

```
List<BorderSegment> segs = new ArrayList<BorderSegment>();
while (scan.hasNextBorderSegment()) {
    BorderSegment seg = scan.nextBorderSegment();
    segs.add(seg);
}
```

We have also included a filter reader *NoiseFilterReader* which will replace all punctuation and linefeeds with space characters. You must use this reader if you want to use Terrain Scanner. More importantly, though, this allows you to use whatever punctuation you find helpful when authoring a terrain description file. All of that noise, while it certainly helps readability, can greatly complicate parsing. The *NoiseFilterReader* makes it all go away.

To process a string, e.g., to test your parsing strings, the setup would be:

```
String testData = " ..... ";
NoiseFilterReader readr = new NoiseFilterReader(new StringReader(testData));
TerrainScanner scan = new TerrainScanner(readr);
```

A similar sequence, but chaining *File* and *FileReader* in place of *StringReader* can be used to strip punctuation from your terrain descriptions.

The *NoiseFilterReader* is an example of specialization through delegation (instead of through inheritance). If you look at the definition, you will see a lot of code, but with the exception of three or four methods, all of that code was generated using the eclipse source>generate>delegate.

Finally, we have supplied many little data transfer objects which the construction methods in *TerrainGraph* will want as inputs. Aside from *Coordinate*, these include *Path*, *Border Structure*.

Completing Terrain Loader

You should use the tools above to complete the wiring between *TerrainLoader* and *TerrainGraph*. We have marked each of the methods which you need to complete with *//TODO*: The comments also include hints about where each method should connect. Most of these methods readXXX where “XXX” is some structure (like a path or border segment) in the description file. Each read method constructs an appropriate set of data transfer objects and “pushes” them to the relevant construction method in *TerrainGraph*, listed below:

- *setFlags(Coordinate flag1, Coordinate flag2)*. Saves the flag coordinates both so that the flags can be displayed and so that you can later retrieve those coordinates from TerrainGraph when you are ready to launch your solver.
- *setConnectedPath(Path path, ConnType type)*. Lays down a connected section of highway, or other connector.
- *setTerrain(List<Coordinate> coords, TerrainType type)*. Marks the cells listed in coords as being *type* terrain. Used for water, forrest, and brush. Note that default terrain is never marked nor indicated explicitly in the terrain description.

It is important to notice that there is not yet an instance of TerrainGraph when you enter TerrainLoader. It is your responsibility to create one by calling the constructor TerrainGraph(cols, rows), as soon as you have finished processing the geometry specification.

You should finish TerrainGraph *before* you attempt to develop the main function in Explore. We have supplied a driver in the test source folder (look for TestLoader) that will allow you to load your file from static text, construct TerrainGraph and call the viewer without a mainline. When you have the loader working, then you can add the call to your solver, directly in the junit driver. Finally, you can use the code from the driver as a start on your code in the mainline.