6.170 Laboratory in Software Engineering
Fall 2005
Problem Set 5: Stata Center Navigation and Analysis
Due Date: Oct 20, 2005

Quick links:

We recommend that you read the entire problem set before you begin work.

Purpose

The purpose of this problem set is to exercise your design and implementation skills. At the end of the problem set, you will have designed and implemented a complete interactive program, from start to finish. You will also gain some experience with component reuse, and its pros and cons.

Part 1: Exercise

Draw a module dependency diagram for the code that your wrote in the last problem set (the graph datatype and subway route finder). You'll need to review your code very carefully to make sure you haven't omitted any dependences.

After that, answer the following 3 questions:

- 1) Briefly comment on the quality of your design as measured by the presence of dependences.

- 2) Are any of the dependences undesirable?
- 3) If so, how might the code be changed to eliminate them?

To hand in:

- 1. Type your answer to these 3 questions in `doc/exercise.txt` then add it to CVS.
- 2. You can draw the diagram using any tool you like. Just make sure you export it as a GIF or PNG file, then add it to your CVS as `doc/exercise.gif` or `doc/exercise.png`. Make sure you add the picture in BINARY mode, or else the picture will be corrupted.

To ensure that your pictures are added in binary mode, go to the menu on top and click on **Window**. Then click on **Preferences...**. Expand the **Team** option on the left. Then click on the **CVS** option under **Team**. Make sure you check the checkbox that says *Treat all new files as binary*. You can then click **OK** to close the window.

Running `validate6170 ps5` will verify whether your picture and text files are correctly added to the CVS repository.

Part 2: Design

As some of you may have discovered, getting round Stata isn't always easy. In this problem set, you'll build a program that generates directions to help you find your way around.

Your program should take as input a starting and ending point, specified by MIT room numbers (eg. "123", "D430", "G519", etc), and should be able to generate both a minimum time and a minimum distance path. It should handle starting and ending points that are on different floors.

We will supply you with a map of (most of) the building in textual form, as well as images showing the room layout of individual floors. How you use these resources is up to you. You will probably want to display the recommended directions graphically, but you **DO NOT** need to build it as a graphical program if you provide a well-designed textual output instead.

The core of your program algorithmically will be the graph datatype that you built in your previous problem set. Below you will find some hints on how to conduct a search in a graph. There are several challenges to overcome beyond the problem of search in a graph, however. At the very least, you will need to figure out how to show directions that cross floors, and how to handle ill-formed input.

The focus of this problem is designing a collection of modules that work well together to provide the necessary functionality, but which interact via simple and clean specifications. A measure of the quality of your design is how easy it is to extend the function of the program. As part of your submission, you should explain how you would handle each of the following enhancements. You are not required to implement them (but if you have time, you may well want to).

- Find optimal paths during a fire alarm (no elevators), and for handicapped (no stairs).
- Find paths that do not pass in front of your TA's office in Stata.
- Find shortest path that also goes through places of a particular type. (e.g. past a bathroom or a printer)

Deliverables

- 1. An object model diagram describing the problem domain: rooms, names, floors, paths, etc. In particular, your model should explain how elevators and stairs are modelled. Please save this picture in `doc/pom.png` or `doc/pom.gif`
- 2. An object model diagram describing the heap state of the running code. Please save this picture in `doc/com.png` or `doc/com.gif`
- 3. A module dependency diagram for your program. Please save this picture in `doc/mdd.png` or `doc/mdd.gif`
- 4. Source code with an appropriate level of commenting and Javadoc specification, and appropriate JUnit tests for your modules. These should be in `src/ps5`
- 5. Think about the 3 potential enhancements listed earlier, then answer how you might implement them. Please type your answer into `doc/enhancements.txt` (Note: you are NOT required to implement these 3 enhancements. We just want you think about how someone could implement these 3 enhancements)

As in part 1, please make sure all the picture files are added in binary mode. Running `validate6170 ps5` will verify whether your picture and text files are correctly added to the CVS repository.

In your grading meeting, you will have an opportunity to demonstrate your program to your TA. You can bring your laptop computer, but you do not have to. If you don't have a laptop, **tell your TA** well ahead of time, so that your TA will try to arrange to have a computer present at the grading meeting.

Resources

Partial Solution to PS4

If you absolutely do not wish to use your code from PS4 in this problem set, please email, and we will give you a minimum graph implementation that should suffice for this problem set.

Maps

In order to make your job easier, Dr. Larry Rudolph and his group have generously supplied maps of each floor. These maps are in the `PNG` format which can be viewed by most of the popular image editors (GIMP, Windows Picture Viewer, etc). Notice that the scale varies from map to map, and is noted (in pixel/feet) on the top right corner of the image. Also note that the non-EECS floors are missing (6-9D); don't worry about filling these in. These `PNG` files are located in the `input/maps` subdirectory of your project.

Graph Files

Larry Rudolph also supplied a graph file for each of the maps. These graph files contain information about how one can move around in the Stata Center. The 6.170 staff has modified these files for completeness and to make your life easier, and placed them in the `input/graphs` subdirectory of your project.

Each graph file can contain five different types of statements. A statement consists of a keyword and a number of arguments, where the keyword and its arguments are all separated by commas.

These are the statements used by the graph files:

- `mapname` - indicates the name of this map; also indicates corresponding `PNG` file. This statement should appear only once within a file. Requires 1 argument :
  - `name` -the name of the map
- `scale` - indicates scale of the `PNG` file. This statement should appear only once within a file. Requires 1 argument :
  - `scale` - a double, which is the number of pixels per feet.
- `vertex` - A vertex represents a location in the Stata Center. This statement requires three arguments, with two optional extra arguments.
  - `number` - a unique identifier for this vertex within the file. (required)
  - `x` - the x coordinate (in pixels) of the location represented by this vertex in the corresponding `PNG` map file. (required)
  - `y` - the y coordinate (in pixels) of the location represented by this vertex in the corresponding `PNG` map file. (required)
  - `name` - a human readable name for this vertex (e.g. "G710","west elevatorbay")
  - `type` - the type of this vertex's location (e.g. "office","bathroom"). The (name,type) pair of each vertex is unique in its file. For your convenience, if two vertices from different files should have an edge connecting them, their (name,type) pairs are the same.

- `edge` - indicates that one can easily go from the first vertex to the second vertex. This statement requires two arguments, with two optional extra arguments.
  - `number 1` - the number of the first vertex (required).
  - `number 2` - the number of the second vertex (required).
  - `cost` - a numeric quantity representing the cost of traveling along this edge. If missing the cost is assumed to be the pixel distance divided by the scale.
  - `cost unit` - the unit of cost (in the graphs we give you, this is either "s" for seconds or "ft" for feet)
- `external_vertex` - indicates a reference to a vertex in another map. This statement requires 3 arguments:
  - `number` - a unique identifier for this vertex within the file.
  - `name` - the name of the other map
  - `external number` - the unique identifier for this vertex in the other map

If you have been to the Stata Center, you might notice that the graph files we provide you with might not match reality very closely. You are not responsible for making sure they do.

Also, note that you will probably be able to reuse parts of your `MetroMapParser.java` in your parser for these graph files.

Sample Code

We provide some sample code which should allow you to build a GUI without knowledge of the Java library's user interface classes (which you'll learn about in the next problem set). This code is located in `src/ps5/GUIExample.java`. To run it, just right click on the file in Eclipse then click "Run As", then select "Java Application". Playing with this code may give you ideas about what sort of things you can do for user input and output. If you would like to write your own GUI, that's fine too.

But, once again, you **DO NOT** need to build it as a graphical program if you provide a well-designed textual output instead.

The code we give you constructs a window with two tabs and a combo box. Each tab displays an image file and some other small drawing. The combo box lets you choose between a number of options. If you click on any of the images, or select a an option on the combo box, you will notice that the title of one of the tabs changes. The sample code we give you does this just to exemplify the use of *Listeners*, which help you respond to user input. You can find two classes that implement the two types of listeners you will most likely need at the end of the `GUIExample.java` file.

Dijkstra's Algorithm

Most shortest path algorithms are written for graphs in which the weights (costs) are associated with edges; an example is Dijkstra's algorithm, which you can find in any

algorithms textbook (for instance, Introduction to Algorithms by Cormen, Leiserson, and Rivest).

The following version of Dijkstra's algorithm has been written up in a pseudo-code fashion which, while not executable, should be easy to read, understand, and translate to Java. The notation [a,b,c] stands for the three-element sequence consisting of a, b, and c; here, we use it to represent a path. When applied to paths, "+" means concatenation; for instance, [a,b] + [c,d] = [a,b,c,d]. If m is a map and k is a key in m, then m(k) represents the value associated with k in m.

```
   // return a shortest path from start to goal
 1 Algorithm Dijkstra(start,goal) {
 2
 3     // active contains paths
 4     PriorityQueue active = { [start] }
 5
 6     // finished contains nodes that we know the shortest path to
 7     Set finished = { }
 8
 9     while active is non-empty do {
10         // minPath is the shortest path of active
11         minPath = active.extractMin()
12         minNode = last node of minPath
13
14         if (minNode equals goal) return minPath
15
16         if (minNode in finished) continue
17
18         // iterate over edges in minNode.edges
19         for each child z of minNode {
20             zpath = minPath + [z]
21
22             // avoid going into loops
23             if (z not in finished)
24                 // you can check whether active already has a path
to z
25                 // for efficiency if you'd like
26                 add zpath to active
27         }
28         remove minPath from active
29         insert minNode in finished
30     }
31     print "path not found"
32 }
```

Clarification:

In line 11, the "extractMin(active)" method already finds the minimum path from the "active" set and then removes it. Therefore, line 28 can be deleted since minPath is already removed from "active".

Statistics

After extensive measurement, we've determined the following fictional average usage statistics for the Stata Center. You may or may not want to take these numbers into consideration when designing your system. If you make a design decision (for example, that climbing stairs through 1 section is always slower than another section or something) that is not listed here, then please document it.

- In average, one has to wait 30 seconds for Stata Center's elevators.
- An elevator takes 5 seconds to go from one floor to the next.
- The average Stata Center visitor walks at 4.5 feet/second.
- The average Stata Center visitor takes 15 seconds to go from a floor to the next using the stairs.
- The closest nodes on the images `4D.png` and `4G.png` are about 40 feet apart.
- The closest nodes on the images `5D.png` and `5G.png` are about 80 feet apart.

Mechanics

In your repository

In your CVS repository you will find the following files:

- `src/ps5/GUIExample.java` : the sample GUI code.
- `input/samples/` : resources used by the sample GUI code.
- `input/graphs/` : the graph files.
- `input/maps/` : the maps of the Stata Center.

Checking in

You should place all your source code in the `src/ps5/` directory and all your documentation in the `doc/` directory. In particular, you must have the following files in your `doc/` directory:

- `doc/exercise.txt`
- `doc/exercise.png` or `doc/exercise.gif`
- `doc/pom.png` or `doc/pom.gif`
- `doc/com.png` or `doc/com.gif`
- `doc/mdd.png` or `doc/mdd.gif`
- `doc/enhancements.txt`

Please submit object models or module dependency diagrams in either `PNG` or `GIF` format. As described in part 1 of this problem set, you must make sure the picture files are added to CVS in Binary Mode (or else the pictures may be corrupted). You can run `validate6170 ps5` to verify whether the required text files and picture files are in CVS or not.

Expectations

What you need to know before you start:

You should be familiar with the basic concepts of module dependency, data abstraction, and what comprises a good abstract type; with the notions of representation invariant and abstraction function; how to express structure as object models and MDDs; how to write specifications of methods; and you should have by now some experience writing Java code.

What you should expect to learn:

You will learn about the challenges of designing a piece of software from start to finish. You should also expect to gain experience with the process of designing flexible and robust software.

How you will be evaluated:

- 10 points for the exercise in part 1
- 20 points for a good underlying model, as expressed in the POM and COM diagrams
- 20 points for a good module design, as expressed in the MDD
- 10 points for a good user interface design
- 20 points for clear, clean and well-documented code
- 10 points for appropriate unit tests
- 10 points for a good analysis of the enhancements

Errata

Working with Files (Oct 19, 2005):

When working with files in computer applications, it is generally bad form to have absolute paths in your code because it will not run on any other systems but your own. (C:\Documents and Settings\User\6.170\ps5... does not exist on Athena).

One solution is to use relative path names. Here, you still have platform dependencies. For example, path names on Windows use the backslash character as a separator whereas Unix-based systems use the forward slash.

A better solution uses java.net.URL (a Universal Resource Locator) to find resources in your program. If a file sample.txt is in the same directory as class file Foo.class, then you can do the following:

```
URL url = Foo.class.getResource("sample.txt"); // Note - No file paths!
```

Use url.openStream() to get an InputStream for the contents of the file (throws IOException).

Clarification (Oct 17, 2005):

In line 11 of the Dijkstra algorithm, the "extractMin(active)" method already finds the minimum path from the "active" set and then removes it. Therefore, line 28 can be deleted since minPath is already removed from "active".
