# CampusQuest

Exploring Graph Theory and the Shortest Route Problem in Java

Ian Melnick

May 5, 2006

# Contents

## 1 Abstract

A Java application was created to use Dijkstra's shortest path algorithm to determine the best route between any two locations in the Science and Engineering Complex. Output resembles online driving direction services like MapQuest. A user selects start and destination rooms, and is provided with a map outlining the best walking path. Text-based turn-by-turn directions are additionally provided, and include estimated travel distance. The program works both stand-alone and as a web browser applet, and can accommodate other buildings and outdoor locations on campus. To allow for ease of expansion, a built-in graphical interface is included to provide new maps and label significant points. Distances are calculated automatically based on map scale size, but manual distance specification is also possible for special cases like stairwells and elevators.

## 2 Acknowledgements

Maps for this project were provided by Loren Rucinski and Fred Puliafico of the Union College Facilities Services Department. Special thanks to David Hemmendinger and John Spinelli for their guidance and support.

# 3 Introduction and Background

This project's objective is to create path-finding software that would allow guests, students and faculty to more easily navigate the buildings on Union's campus. Users would be able to select starting and destination points, and the program would return step-by-step instructions. Like the driving directions service at *MapQuest.com*, a graphical route would also be displayed. Ideally, the directions determined by the software would be the shortest possible walkable route between the two locations.

Finding the shortest route is a lengthy task. For a person, determining the shortest route between two places requires the discovery of every possible way of getting to the destination. Determining a single valid route may enough of a challenge. Then the distances for each of the routes must be measured, and conclusions are drawn. In a very large environment, this process can become so time consuming that it may be realistically impossible to complete. When employing a computer, the process works similarly, only the computer has the advantage of knowing where every reachable location is from the start. With this knowledge, the shortest route can be found much faster—it's simply a matter of determining the relationships between all the available locations.

A lot of modern technology exists to help people more easily navigate the world around them. As mentioned earlier, online services like *MapQuest.com* and *Google Maps* provide driving directions services for most parts of the United States. This allows people to quickly find out how to get somewhere without guessing or using a paper map. Typically in these web-based services, a user inputs the starting and destination locations, and the website returns text-based turn-by-turn directions and a map depicting a graphical route. Also common in the modern navigation technologies are *Global Positioning System* devices. These GPS units use satellite signals to determine where a person is situated on the Earth. GPS has become so common and advanced that it is now available both in cars and as handheld units. Like the online services, GPS are able to provide outdoor driving and walking directions.

Nearly all the technologies that deal with displaying and using geographical data use special software known as a *Geographic Information System.* A GIS is a sophisticated database that, in simplest terms, describes where things are and what they look like. They can be used to input and display roads and rivers, demographics and trends, and practically any other geographically-related information. While this project does not employ such advanced technology, it does implement basic concepts found in these systems to meet the project objectives. (Wikipedia.org, 2006)

Perhaps the most important aspect in a computer-driven geographic system is the way in which information is represented internally. Data must be stored in such a way that it can both be accessed and manipulated by the computer, and be translated into a visual representation for the user. Additionally, systems that represent many different kinds of data must be able to correctly relate them together. A simple example is relating the physical geographic information to the specific data associated with it. Geographic data may be stored as a typical raster image (i.e., a JPEG), and descriptive data (such as city names) is stored in a separate location. The descriptive data must contain information that describes where it exists relative to the map image, such that it can be superimposed correctly. Think about drawing city names on a map—if the locations of where the names belong are unknown, the names are meaningless. The more complex alternative is a "vector" format, where information describing the geographic data is stored, rather than the image itself. This makes it easier for large-scale GIS systems to accurately depict data and internally relate many types of information together. Rather than having to draw metadata on top of an image, the metadata can be drawn within the image. (Wikipedia.org, 2006)

For this project, emphasis is placed most on determining shortest routes within buildings. Therefore, a simple raster-based map works fine for representing floor plans, with descriptive (route) information being superimposed. This allows the data representation mechanism to be focused solely around the project objectives. In this case, a location can simply be considered a coordinate on a map. The path connecting the locations can simply be a line.

In computer science, the graph data type fits this requirement perfectly. A graph consists of nodes (a.k.a points, vertices) and edges (lines), and unlike a tree, a graph has no defined structure to conform to. This allows a graph to be used for representing locations and the paths that connect them. Graphs are so often used for this purpose that the study of this topic has a name—graph theory. Of course, graphs are not limited to geographical modeling, and also are commonly used for describing computer networks. Various properties can be applied to edges to make the network model more meaningful. For example, edges that have an assigned length are known as weighted edges, and a graph with weighted edges is known as a weighted graph. Weighted edges are useful for indicating distance between points. In certain cases, a path may only be valid in one direction—think of a one-way street. This property is known as directed edges, and a graph that is made of directed edges is called a directed graph. Similarly, a graph which does not have directed edges is called an undirected graph. Weighted and directed edges can be combined to represent complicated networks. (Wikipedia.org, 2005)

Normally, a program must do more than simply store a graph—most applications require the ability to step through it in a meaningful order. Graph traversal is the process of "walking" from node to node via the edges. Common traversal techniques include breadth-first search (BFS) and depth-first search (DFS). Topological sort, though a sorting technique, can also be considered a traversal method since it walks from node to node within the graph. This approach requires a directed graph, and nodes are traversed based on what other nodes point to them. The breadth- and depth-first searches require neither weighted nor directed graphs. Think of a depth-first search as the process of exploring the length of a corridor without attempting to visit every single room along the way, coming back to the passed-by rooms later. A breadth-first search is the opposite—visiting each nearby room before advancing further down the corridor.

A simple graph traversal will not solve the shortest path problem on its own. Single-source shortest path algorithms like Bellman-Ford and Dijkstra use these traversal methods to determine the shortest paths between two points. The shortest route also may not always be the fastest route, and in this report, whenever “best route” is used, it is used to mean “shortest route”. The strengths and weaknesses of these various algorithms will be discussed later, but keep in mind that Dijkstra’s algorithm, a BFS-based solution, is key to reaching the project’s goal.

Though it may not be powerful enough to compete with online driving directions services, this project is still quite useful. It has the potential to be another item in Union College’s toolbox of things to make visitors’ lives easier. The project is able to support any building on campus and any outdoor area, provided basic floor plans are available. Though the Science and Engineering Complex has been used as a proof-of-concept, the program has the potential to extend to the entire campus, and provide a truly useful service to the Union community. It could also be used in malls, Chicago’s McCormick Place, or other large areas where an interactive map would be useful. Additionally, it could be used within other similar projects to make an even more powerful tool. The disadvantage to these types of technologies, however, is that people become less dependent on directional instincts and more dependent on potentially buggy software and outdated maps. Most companies pride themselves however on map accuracy, and overall, the ability to be able to get immediate directions is really priceless.

The remainder of the report will discuss project-specific requirements, alternatives chosen for the implementation, and a detailed description of the implementation itself. A program user’s guide and maintenance manual is also included at the end.

## 4 Design Requirements

The program needs to be able to present a graphical interface that would allow the user to specify start and destination points. This display would include a map of the current floor (within the current building), which can be panned using the mouse. Available locations would be drawn on top of the map. Both text-based step-by-step directions and an outlined graphical route must be returned. As the user steps through the given directions, the map should update to display the currently described position.

To prepare the program for use, the maintainer must obtain floor plans of the building and manually specify accessible locations and their relationships. This data entry should be performed through a similar graphical interface, in which the input is achieved via click-and-drag.

The program must be able to write the location information to a file, and be able to read the JPEG image format for floor map display. An appropriate shortest path algorithm must be implemented to determine the best route between two points. This need not be the fastest route as well; therefore, tracking time between points is not a requirement, but an option. The route-finding logic must be able to function between multiple floors, and be designed such that obtaining routes between buildings is also possible. To avoid over-complexity in the first version of the software, accommodations for special needs is not required. This primarily involves users who are limited to elevator use for travel between floors.

## 5 Design Alternatives

Java is an appropriate programming language choice, as it is write-once-run-anywhere, and thus very portable. It also can be used to create stand-alone applications as well as "applets" that are executed via a web browser. Additionally, by using a language that is widely used and well-known, the potential for ability to integrate the software into other future projects increases. Java 1.5 (JDK 5.0) is the most recent public release of the language, and it includes new features like generics and autoboxing which were useful for development. Since it is backwards-compatible with older Java conventions, it made sense to develop the project using JDK 5.0.

Though I considered using the Java OpenGL graphics libraries for rendering output, they introduce another variable in the development process. OpenGL would need to be studied and learned, and could impede development of the primary components. Since the standard Java AWT library meets the needs of the project, it was chosen for use. Similarly, Java's built-in windowing toolkit (Swing/AWT) was chosen for building the interactive components, since it is powerful enough for the requirements of the project.

Serializing the graph data to an XML-based storage format would allow certain data to be modified using a text editor. Beginning with Java 1.4, a built-in XML object encoder/decoder has been offered through the java.beans package. However, it is limited as to what kinds of objects it can serialize, and it was found during testing that it is unreliable for serializing linked data structures. Free third-party XML serialization libraries are available, such as Xstream, XMLWriter and XOM. In terms of usage, these libraries do not quite function the same way as Sun's built-in serialization techniques. If in the future a different serialization format was desired (or these libraries had unforeseen problems), much code would need to be modified to adapt to the change. While a wrapper class could be produced to allow for an easy transition, it was again decided to focus more on the development of the primary functional components. Therefore, Java's traditional object serialization strategy was chosen, regardless that it is a binary format. Though minor data touch-ups will be impossible, it does force the data entry/editing portion of the project to be fully functional.

The program must be able to represent specific locations as well as the paths between them. A graph type data structure fits this requirement very well. In a building, paths between any two points are always positive, and have constant distance. Therefore, the data can be represented using an undirected, positively-weighted graph. While Java provides many different abstract data types, a Graph class was not found in the Java 1.5.0 class library. Therefore, it was decided that a graph-like class would be created and tailored to meet the specific requirements of the project. Array-based storage for a very large number of elements might become inefficient if/when dynamic resizing is necessary. And if a classic two-dimensional array is used to represent the graph, given the graphs are undirected, this would introduce large space inefficiency as well. Therefore, nodes and edges will be stored using linked lists.

Of the three common shortest path algorithms, Dijkstra's algorithm is the best fit. All paths in any building have a positive distance, so the Bellman-Ford algorithm is unneeded. Floyd-Warshall is also ruled out since the shortest route is required between only two specific points at any given time. Since the project's specifications do not include any caching method for route results, algorithm efficiency is also important. Therefore, Dijkstra's algorithm was chosen. (Skiena, 1997)

# 6 Final Design and Implementation

I wrote the software in Java 1.5, using Dijkstra's shortest path algorithm for determining routes. Java's AWT libraries were used for graphical display, and the built-in object stream methods were used for storing and retrieving graph data. First I'll describe how the program operates, and later provide a more detailed explanation of its classes.

Application mode, map, and screen dimensions are specified via commandline arguments or applet parameters. If no arguments are provided, default settings are assumed. Available modes are "Editor" and "Router", where the former provides a data entry environment, and the latter is the end-user route finding interface. In both modes, a map of the requested floor is drawn in a window, with directional information drawn on top of it. The editing mode is used to store this information, while the routing mode uses this data to show the user where to go.

Locations are represented by "nodes" which store coordinates relative to a certain position on the map. Accessible nodes are represented by green circles, while invisible nodes are represented by yellow circles (editing mode only). Invisible nodes are used for corners, turns and other locations which are not significant destinations. They are necessary in order to provide an accurate description of a walking path. Each node contains a list of "edges", or paths, that it is directly connected to. Therefore, relationships between nodes are determined by edges that nodes have in common. On the map, edges are displayed with black lines. The program automatically determines edge length by calculating the distance between the nodes it relates. By using the screen and print resolutions of the base map image, the program is able to report these distances in terms of feet.

Since the map is very large, only a portion of it is displayed on the screen at any given time. It can be panned using click-and-drag and mouse wheel scrolling. Clicking the mouse wheel changes scroll direction. In editing mode, the map's position can also be manipulated using sliders.

Routing mode provides the user with drop-down menus that list source and destination rooms. When valid locations are selected, a text box with a list of directions is returned. Next and previous buttons are provided at the top of this window. Within the list, the current position shown on the map is indicated using arrows. As the user clicks through the list, the map centers itself around the currently described position. This makes it easy for the user to see where to go. If the next position involves a different floor, the current map is replaced with the appropriate one.

In editing mode, the maintainer may create nodes by double-clicking over the position of the desired spot. A node may be moved by clicking and dragging it to a new location. Edges are created by clicking and dragging from one node to another. A menu for modifying a node is provided when the node is double-clicked on. This menu allows the maintainer to change the node's description, modify its edges, and link it to other floors. Typically, only nodes representing stairwells and elevators are linked to other floors. Changes to the graph are saved using the Save button, and are loaded via the Load button.

Saving to and loading from different configuration files is not supported; the graph data is saved to a DAT file with the same file name as the respective map image. Graph data files are saved to and loaded from the home directory of the current user. Map images must follow a specific naming convention: map_building_floor.jpg, where building specifies the building, and floor specifies the floor. Note that png- and gif-format map images are also permissible, though only jpg-type images have been used for this project.

There are thirteen public classes: `Main`, `Editor`, `Router`, `MapComponent`, `MapItem`, `NodeItem`, `EdgeItem`, `Dijkstra`, `NodeEditor`, `FloorLinker`, `DirectionsFrame`, `Utility` and `Errors`.

## 6.1 Main, Editor and Router

`Main` is responsible for launching the program. It extends `javax.swing.JApplet` so that it can be run directly as an applet, or encapsulated within a `JFrame` for stand-alone operation. It calls `Utility.getArgs` to parse the parameters sent to the program either via commandline or HTML. Bruce Eckel's Console class is used for the applet-to-frame encapsulation. (Eckel, 2002)

`Editor` and `Router` are the two main interactive classes, and are in charge of running the rest of the program. Only one is used at a time. They each set up the appropriate windowing environment, and contain specific callback methods that are invoked when the user performs an action. For example, all click and drag-and-drop functionality is defined in `Editor`'s `MouseMethods` class. When the mouse is clicked, `mouseClicked()` is called. If the first or third mouse buttons have been double-clicked over an empty space, a new node is created there; if a double-click occurs over an existing node, the `NodeEditor` is called on it. As another example, `mouseReleased()` is called whenever a mouse button has been released. It checks to make sure that the mouse was dragged first, and then does one of three things: move a node, create an edge, or move the map. All of the callback functions extend or implement Java's mouse-action classes.

`Router` has similar mouse and button-related callback functions, though they're not as complex as `Editor`'s. `Router` is mostly concerned with properly displaying menus that allow the user to select start and destination locations. The `setNodeList()` method is used to generate a list of all possible places from every available map, and this list is later loaded into the `JComboBox` menus that the user selects from. The `setMap()` and `update()` methods are used to set and display `Router`'s map component, so that different maps can be loaded within the same session. This is required when traveling between multiple floors. Management of the map itself, however, is left to the `MapComponent` class.

## 6.2 MapComponent

`MapComponent` is responsible for handling and displaying all map-related graphical information. This includes drawing the base map as well as the nodes and edges on top of it. As a `JComponent`, `MapComponent` can be used in any Swing application, and is completely self-contained. The `paintComponent()` method is called by Java every time a `JComponent` needs to be rendered. Therefore, the main display logic for the `MapComponent` resides within this method. The base map is opened as an `ImageIcon`, which is then painted on to the `MapComponent`.

Again, since the map image is larger than the available screen area, not all of the map can be shown at once. Panning occurs by shifting the painted location of the `ImageIcon`. Experiments with these painting routines found that Java considers the top-left corner of the component as the origin, with the bottom-right corner being the maximum. This is illustrated in Figure 1. Painting of the image begins at the origin of the component. Therefore, to view image area beyond the boundaries of the component, the image must be shifted upwards and leftwards. This forces the origin of the painted image to be negative relative to the origin of the component. Thus, when drawing the map, coordinates are nearly always negative.

When the component becomes mouse-aware, coordinate issues are further complicated. Mouse coordinates are always positive, as they are relative to the component, and only apply within the component's boundaries. In other words, two coordinate systems are used within the `MapComponent`—one that specifies the origin of the map, and the other that tracks the location of the mouse. The former always is a negative pair, while the latter is always a positive pair. Therefore, the MapManager class deals with map manipulation, and map-mouse relationships.

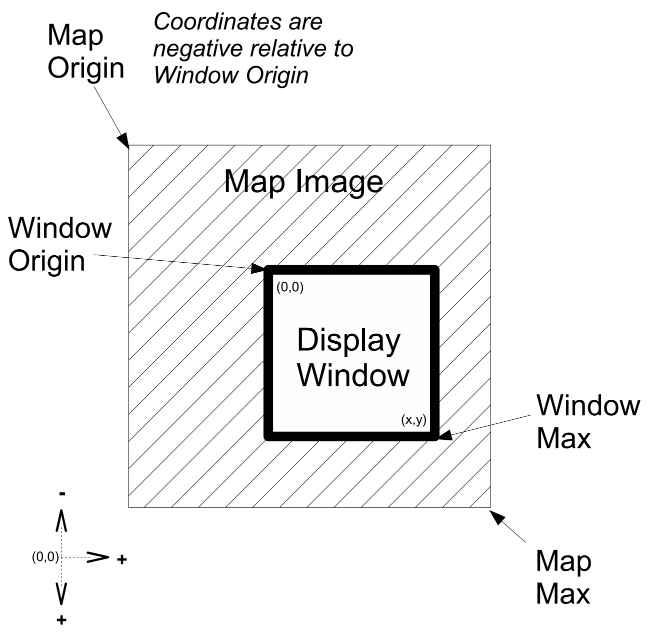


Figure 1: Map Coordinates Diagram

Some examples of methods found in `MapManager` are `centerAroundPoint()` and `mouseCoordsToMapCoo`
The former is designed to center the map on a point relative to the visible component. The latter converts mouse coordinates (relative to component) to the appropriate location on the map.

Originally the methods contained in `MapManager` were simply methods of `MapComponent`. However, `MapComponent` is also responsible for drawing nodes and edges. Early on, node management methods were also plain methods of `MapComponent`. Later, the map-related and node-related methods were separated, to allow `MapComponent` to be more easily understood and maintained. Thus, `MapComponent` contains another class of helper methods, called `NodeManager`. `NodeManager` is essentially the specialized graph-like class that was discussed in the previous section. Anything related to node manipulation is performed through it. `NodeManager` deals with storing nodes, tracking their relationships, and saving and loading this data to and from a file. Like `MapManager`, however, `NodeManager` depends on the `MapComponent` environment—it is a part of `MapComponent`'s core functionality—and therefore, it is inappropriate to separate `NodeManager` into an entirely separate class.

The `MapComponent.initMap()` method exists to allow for the `NodeManager` data to be accessible without loading the respective map image. While most cases require both the map and nodes to be loaded simultaneously, in certain situations, it is only appropriate to load the node data. More on this later.

## 6.3 MapItem, NodeItem and EdgeItem

While `NodeManager` should be considered the graph-like data structure, the data objects it stores are `NodeItem`s. Therefore, the `NodeItem` class represents nodes on the map. `NodeItem` inherits from `MapItem`, which is a generic class that describes any object to be drawn on a map. `EdgeItem` is another `MapItem`, which represents an edge. Of these `MapItem`s, `NodeItem` has the most responsibilities. A complete node representation includes its description, the map it belongs to, and its coordinate location on the corresponding map. It also stores a list of edges that it is connected to, and a list of floors it is linked to. This makes it very easy for the `NodeManger` to determine node relationships—it simply must find nodes that share a common edge. A very similar technique is used for finding floor-linked (vertically-linked) nodes. Nodes which represent a stairwell or elevator contain a list of identification values of the corresponding nodes on the other floors. For the Dijkstra algorithm to work correctly, the `NodeItem` must contain some additional algorithm-specific fields. Using a separate data structure for holding algorithm-related findings would dramatically complicate the shortest-path logic.

`EdgeItems` are also important—they are responsible for storing distance and descriptive information. If nodes simply stored a list of references to other nodes (and `EdgeItem`s didn't exist), then distance information would continually need to be recalculated. Descriptive information would also be lost. Finally, without edges, the object model wouldn't as closely match the rooms and hallways that are being represented. Node ID references are used for floor links, however, because there is no significant horizontal change between floors.

## 6.4 Dijkstra

The `Dijkstra` class contains the logic for finding the shortest route between two points. *Data Networks* contains the definition of the algorithm. Rather than re-defining it here, a basic description of how the logic works is included instead. The class does a bit more than simply implement Dijkstra's algorithm—it also includes functionality for working between floors, and obtaining the desired route based on the algorithm's findings.

As far as the algorithm itself is concerned, there are three major functions. The first, `setConnectedDistances()`, marks the distance values on all nodes connected to the specified one. This is analogous to what algorithm definitions refer to as "labeling" the node's distance from the requested source. Additionally, each of these nodes have their source fields set to the specified node, so that determining a route is possible. Each of these newly touched nodes are dumped into the $T$ list, which is for temporarily labeled nodes. This is where the second function comes into play—`closestNodeInT()` iterates through $T$ to find the node with the minimum distance to the requested source. Following the algorithm's definition, the next step would be to append the result of `closestNodeInT()` to the $P$ list—that is, the list of permanently-labeled nodes.

The `updateLabels()` method ties these processes together, continuing to build $P$. According to the definition, the algorithm is run until $P$ contains all nodes. However, for this task's specific purpose, knowing the shortest distance to every node in the graph is not needed. Therefore, to save execution time, `updateLabels()` runs until $P$ contains the requested destination node.

Before continuing the discussion on the Dijkstra class's responsibilities, a brief note on part of the algorithm implementation. $T$ is not explicitly included in the algorithm definition, but here it used for the "estimation" process, which is a requirement. $T$ should have been implemented as a priority queue, rather than a set, as this would have eliminated the need for the `closestNodeInT()` function. While writing this part of the code, I didn't know what a priority queue was. Now that I do, I recognize that it might be useful here.

However, there was another issue worth mentioning. T is not just any set, it is of type `CopyOnWriteArraySet`. When a normal `Set` was used, `updateLabels()` ran into some concurrency issues with $T$, since `updateLabels()` is recursive. Therefore, Java's `PriorityQue` may not have worked here anyway. While it may be possible to simply pass $T$ through as a parameter to `updateLabels()` and avoid making it a class field, the current implementation most closely follows the pseudocode I wrote to code it, and in my opinion, this form makes it easiest to understand what is happening.

The remainder of the `Dijkstra` class mostly deals with accommodating multiple floors, and returning the shortest route between source and destination. While the main algorithm runs, as shortest paths between nodes are determined, they are labeled with the identification number of the previous node. After the algorithm finishes, the shortest route is determined by the `getRoute()` method. This method generates a list of nodes to travel to, in order from start to finish. It works by starting at the destination node, and adding its previous node to the front of the list. This continues until the starting node is contained by the list. A stack-like data type would have worked best here, however, at the time of writing this method, I didn't know what a stack was.

To handle routes between floors, the `Dijkstra` class contains the `linksToFloor()` and `nearestFloorLink()` methods. These are private methods which are used respectively to determine if a node links to another floor, and to find the closest floor-linking node from the given node. As of this writing, `nearestFloorLink()` does not work exactly as described. It returns the first floor-linking node found, but this may not actually be the physically closest floor-link available. Even worse, this means that the floor-linking node returned may actually be in a non-optimal direction relative to the route as a whole. While writing this method, the goal was to first ensure that the `Dijkstra` class could be used to work correctly with multiple floors. The `nearestFloorLink()` bug will be fixed in the future.

Nearly all times the `Dijkstra` class is used, the `run()` method must be invoked after class initialization. Before `Dijkstra` handled multiple floors, this method simply invoked `updateLabels()` on the source (starting) node. With multiple floor support, the `run()` method now has a larger responsibility. If the start and destination nodes are on separate floors, the `run()` method uses `nearestFloorLink()` to find a node which can be used to connect the floors together. It then splits the shortest-route problem into two separate pieces. The main `Dijkstra` instance is reset to determine the shortest route between the source node and the floor linking node. A new `Dijkstra` instance is created within it to determine the shortest route between the corresponding floor linking node and the final destination node. When `getRoute()` is called in the main instance, after the route between the source and the floor link is determined, the internal `Dijkstra` instance's route information is loaded into the remainder of the route list. Both routes merged together appear as one contiguous route to the user. This approach required no changes to the Dijkstra algorithm itself, and therefore, no further testing of the algorithm's validity was required.

## 6.5 NodeEditor and FloorLinker

The remaining classes provide helper functions and secondary graphical interfaces for `Editor` and `Router`. `NodeEditor` and `FloorLinker` are both classes used by `Editor` for managing node information and relationships. The `NodeEditor` is called whenever the user double-clicks a node in editing mode. In the *Node* tab, additional details about the node can be modified, such as its description and its visibility. A node may also be deleted from the map in `NodeEditor`'s *Node* tab. Two other tabs are displayed—one for modifying edge details and another for managing floor links for nodes that are stairwells or elevators. In the *Edge* tab, the edges that the node is connected to are listed. When an edge from this list is selected, its length may be changed, and it can also be deleted. Note that when nodes or edges are deleted, all related objects involving that particular item are updated, too.

I chose to modify edges through the `NodeEditor` rather than create a separate frame just for edges, because it's difficult for the user to precisely double-click on an edge—edges are very narrow. Since edges also overlap with nodes, additional logic would also be necessary for the program to distinguish between where a user has double-clicked on an edge, and where a user has double-clicked on a node. Having the edge-modification interface within the `NodeEditor` may not be the most intuitive choice as far as the end-user is concerned, but it avoids overcomplicating the action-listening code within the `Editor` class.

The *Floors* tab is the last tab in `NodeEditor`. While the `Edges` tab manages horizontal links, the *Floors* tab handles vertical links. When the *Links floors* checkbox is selected, the user can press an *Add New Link* button. This loads the `FloorLinker` class initialized to the next available neighboring floor.

`FloorLinker` is a watered-down `Editor`-like window, which loads the other floor's map centered on the approximate location of the corresponding stairwell/elevator. This neighboring floor must already have had nodes created for it. The user double-clicks the correct corresponding node, and references in both nodes are updated.

`FloorLinker` also has a `deleteConnection()` method, which is designed for use when a user would like to break the vertical connection between two nodes. This method is called from `NodeEditor`'s *Floors* tab when the *Remove Selection* button is clicked.

## 6.6 DirectionsFrame

`DirectionsFrame` is called by `Router` when the user has selected valid start and destination points. The step-by-step route directions are listed in a `JTextArea` within this frame, with the current position indicated by arrows. *Previous* and *Next* buttons are provided at the top of the frame, such that the user is able to scroll back and forth between positions in the route. The `DirectionsFrame` is called with a reference to the `Router` that called it, so that it can send messages back to the `Router` to realign the map (or load a new one) depending on which step it is on. The `showDirections()` method is responsible for displaying the directions, and it is called by the `ButtonMethods` functions whenever the user clicks *Previous* or *Next*. While `showDirections()` will determine when a new map needs to be loaded, the `ButtonMethods` functions are responsible for telling `Router` where to center the current map.

## 6.7 Utility and Errors

`Utility` contains static methods that don't belong in any of the other classes. Its two methods are `getArgs()` and `map_images()`. The former is responsible for parsing command-line and applet arguments and storing them in a uniform way that the rest of the program can use (`Utility.Args` class defines a common structure for argument storage). The latter obtains a list of available map images. The list can be filtered to match only certain names, so that the result can be limited to a specific building. This method is used by `Router.setNodeList()` when extracting available locations, but is general enough to be able to be used by any method that might need it. The list was originally generated by simply listing the directory that contains all images. Unfortunately when the program is distributed in a JAR file, a normal directory list query cannot be made. Therefore, the function determines whether or not the images are stored within a JAR, and if so, it uses Java's JAR functions to extract the list of map images.

The `Errors` class is used for debugging statements. Rather than manually specifying the location of code while outputting debug-lines, the `Errors.debug()` function uses `Errors.source()` to trace the location of the code, so that this information can be output automatically. It makes maintaining debugging statements much easier.

# 7 User's Manual

This section discusses how to use and maintain the software.

## 7.1 Usage

As a stand-alone application, the program is designed to be invoked from the commandline using specific arguments. This can be done using the following basic form: `java -jar Project01.jar <arguments>`. The first argument is either *Editor* or *Router* to indicate which mode the program should run. The former will open a data editing environment, while the latter will provide the shortest-path-finding display. The next arguments are the map to use, and the width and height of the program. By convention, the map names go by *building_floorlevel*. In the case of this project, the only available building is known as *scieng*, and the available floorlevels are *000*, *100*, *200* and *300*. Therefore, if the basement of the Science and Engineering Complex is desired, the map argument would be *scieng_000*. The width and height are specified in pixels, as plain integers. To obtain a help screen, invoke the program with the *-h* argument. If no arguments are provided, the program defaults to *Router* mode, with *scieng_000* as the base map. This default behavior can be changed in the `Utility` class (see below for further development information).

## 7.2 As an applet

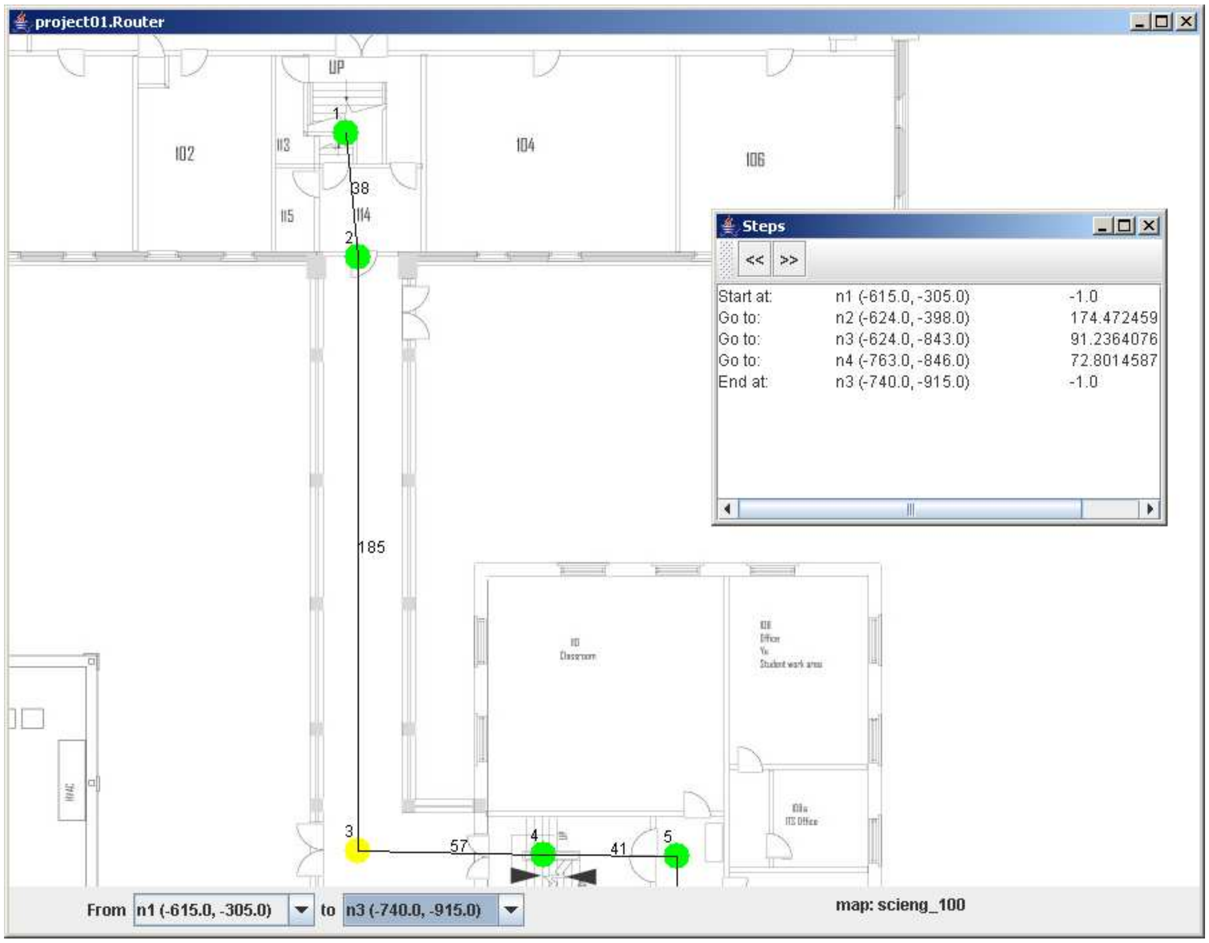
To run the program within a browser as an applet, place the HTML shown in Figure 2 in a file within the same directory as the JAR. JavaScript can be used to determine appropriate width and height values, though that's not shown here.

```
<applet code="project01/Main.class" archive="Project01.jar"
        width="1024" height="768">
  <param name="m" value="scieng_000">
</applet>
```

Figure 2: Example applet-calling HTML

## 7.3 Router

Depending on platform and CPU speed, it may take a moment for the program to completely load—please be patient. Select the starting point from the first menu and the destination point from the second menu. The destination may be on a different floor, and the floors are indicated within the menu. As soon as a valid destination has been selected, a directions window will open displaying the correct directions. The arrow buttons at the top of this



```
Start at:   n1 (-615.0, -305.0)   -1.0
Go to:      n2 (-624.0, -398.0)   174.472459
Go to:      n3 (-624.0, -843.0)   91.2364076
Go to:      n4 (-763.0, -846.0)   72.8014587
End at:     n3 (-740.0, -915.0)   -1.0
```

Figure 3: Router Mode

window may be used to focus the map on the indicated direction. The map may also be moved using the mouse via click-and-drag. If the directions involve multiple floors, the map of the new floor is loaded. The name of the map is shown in the bottom right-hand corner of the main window. The screen capture shown in Figure 3 illustrates what the Router mode looks like. Please note that this screen capture is of test data, and actual points are labeled with meaningful names in the final release of the project.

## 7.4 Editing

Unlike *Routing* mode, the nodes are not displayed immediately when the program is opened. Therefore, the *Load* button is used to load and display the nodes for the respective map.

Similarly, the *Save* button is used to store the node data in a file. Only one data file is stored per map, and it assumes the same name as the map image file that it corresponds to. The data file is stored in the root of the user's home directory. On Unix-like platforms this is typically found in `/home/username`, while on Windows NT-based machines this is found in `\Documents and Settings\username`. A screen capture of this mode is shown in Figure 4.

The map can be moved around on the screen using the sliders on the top and left parts of the screen, or using click-and-drag, or using the mouse scroll wheel. When the wheel is clicked, the map will pan about the opposite axis.

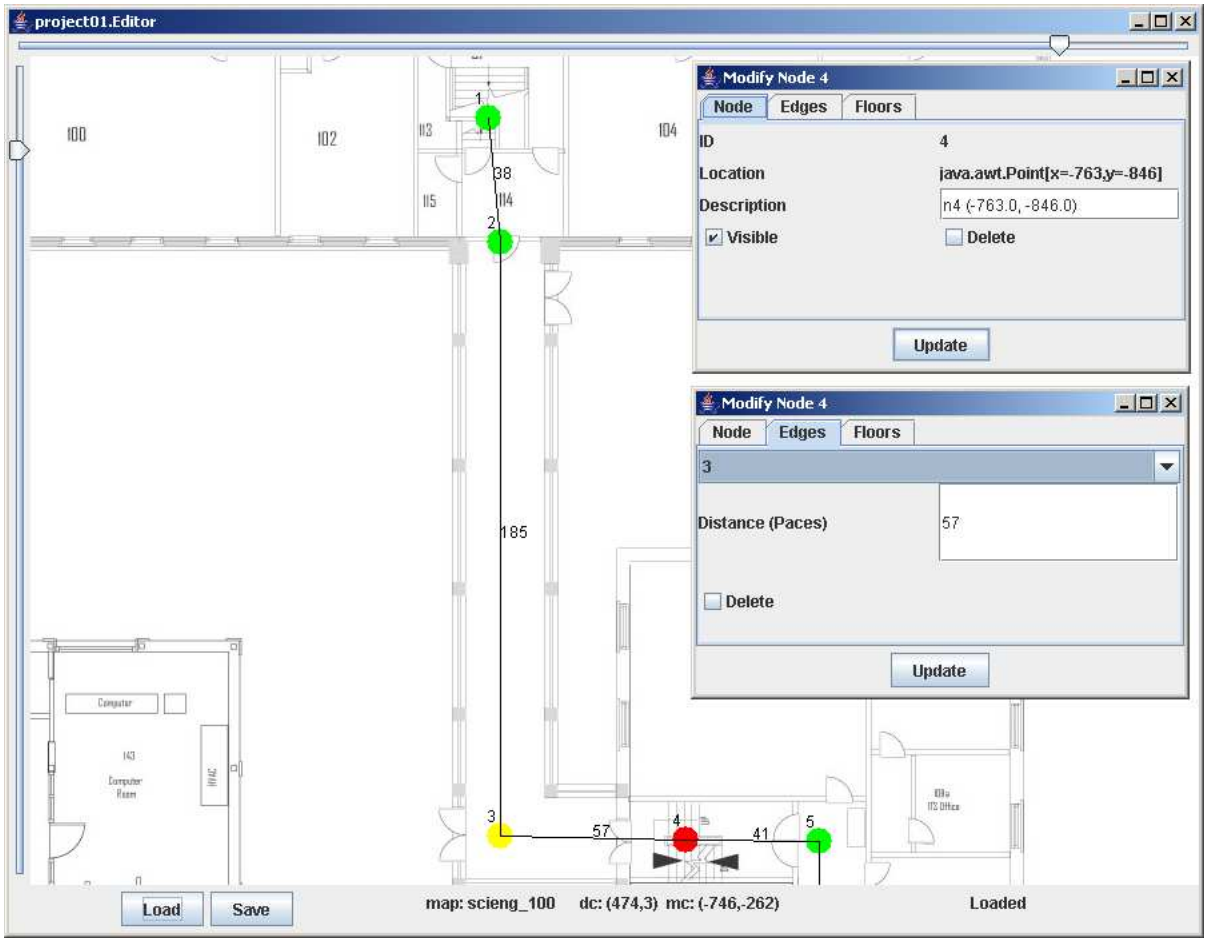
A node can be created by double-clicking over the desired location. It can be moved via click-and-drag. To modify the node's properties, double-click it. The node's name can be changed in the properties window that opens. If desired, the node can also be deleted through this window.

To create edges, click-and-drag between two existing nodes. To delete the edge or modify its distance, double-click either of the nodes that it connects, go to the Edges tab, and select the correct edge.

To create a floor link (i.e., represent a stairwell or elevator), double-click the node at the correct stairwell, and go to the *Floors* tab. Select *Links Floors* and click *Add New Link.* Double-click the correct corresponding node in the new window that opens. Click *Add New Link* again to set the link on the other corresponding floor. A screen capture of the floor link feature in action is shown in Figure 5.

## 7.5 New Maps

Every map image file must be in JPEG (`.jpg`) format, and begin with the name "map_" in the project's images directory. Since all maps are stored within the JAR, when maps are added to the program they must be included into the JAR file. The easiest way to do this is to simply add the files into the correct directory, and recreate the JAR using ant. Note that the map images must have print resolution of 96dpi. See section 8.



Modify Node 4

Node | Edges | Floors

| | |
|---|---|
| ID | 4 |
| Location | java.awt.Point[x=-763,y=-846] |
| Description | n4 (-763.0, -846.0) |
| ✓ Visible | ☐ Delete |

Update

Modify Node 4

Node | Edges | Floors

3

| | |
|---|---|
| Distance (Paces) | 57 |

☐ Delete

Update

map: scieng_100 dc: (474,3) mc: (-746,-262) Loaded

Figure 4: Editor Mode

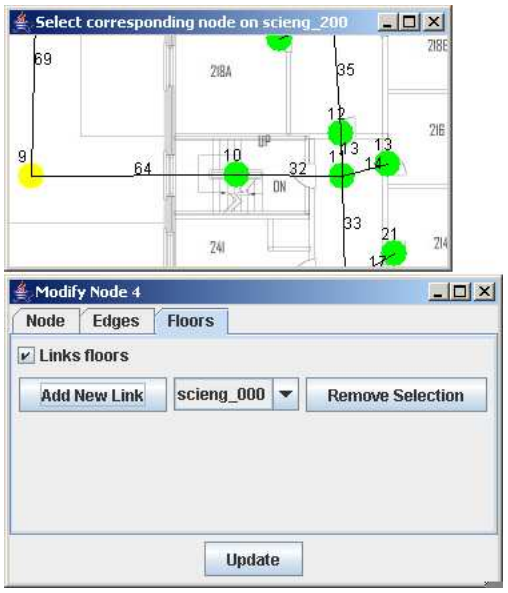


Figure 5: Floor Linking in Editor Mode

## 7.6 Development / Class Reference

This software was developed using Java 1.5/JDK 5.0 Standard Edition with the NetBeans 4.1 IDE. Please refer to the appendix for the source code.

# 8 Known Bugs

One of the goals of the project was to be able to provide the shortest path between two locations. While this was achieved for any two locations on a specific floor, as of this writing I have not yet proven that the shortest route is also correctly obtained between two floors. As mentioned above, within the `Dijkstra` class, `nearestFloorLink()` returns the first floor-linking node found, but this may not actually be the physically closest floor-link available. It may in fact actually be in a non-optimal direction relative to the route as a whole. Occasionally a floor link may not even be found, and currently this causes the program to crash. I've also discovered that a stack overflow occurs when going between three floors. I am working hard at fixing these specific problems. Again, the first concern was to ensure that the `Dijkstra` class could be used to work with multiple floors, and these bugs in `nearestFloorLink()` will be fixed as soon as possible.

A requirement for the returned directions is that they are "step-by-step". While the program currently does this, it would be more useful to provide turning directions (i.e., left and right) when turning is required. It would also be more usable if points along a straight path were not returned, i.e., points between the beginning and end of a hallway were not presented in the directions if the user was required to walk the complete length of the hallway. Both of these problems can be resolved using an angle-based approach, where the angle determined between three points describes the direction that the user must follow. At the time of this writing, this directions enhancement is in the process of being completed.

To correctly determine the physical distance between two points, both the physical dots-per-inch resolution and the screen resolution need to be determined. Unfortunately, not all JPEGs have dpi information listed in their JFIF metadata, and when they do, it's not always in the same field. Two different techniques were used to extract dpi information from the maps used, and both failed (`ImageInfo` class from `schmidt.devlib.org` and Sun's `JPEGImageDecoder` class). Therefore, the dpi for the map files was hardcoded into `MapComponent`. Eventually, a method should be provided such that the program can dynamically determine dpi of any map image, so that different print resolutions can be used.

In *Router* mode, the map loaded via the commandline arguments must contain the point of the desired starting location. If a point on a different map is selected, then the program will not function correctly.

Finally, the applet functionality needs to be fixed. Currently the program does not execute correctly when invoked as an applet.

# 9 Discussion, Conclusions, Recommendations

Since I became aware of MapQuest, I had always wondered how driving directions systems worked. It seemed pretty phenomenal to be able to instantaneously obtain directions to anywhere that I wanted to go. Considering the complexity of the Science and Engineering building, it seemed like it would be useful to be able to have a similar directions service for locations on Union's campus. Creating my own program to do this, however, seemed unrealistic—some companies are entirely based around on collecting geographic data and producing GIS technology. John and David helped me understand some of the core concepts behind the technology, so that I could adapt it to our campus buildings. By using existing floor plans, I didn't have to worry about creating my own geographical data. This allowed me to focus strictly on the shortest path problem (using Dijkstra's algorithm) and creating a usable graphical interface. It was still quite a challenging project, and there was much I needed to learn through doing.

I'd never had much exposure to GUI development before, and I'd never done any graphics programming at all. Fortunately, Sun provides great documentation for Java. I hadn't yet taken the CSC-140/Data Structures course at Union, so many complex data structures in the program were represented with linked lists. This stemmed from the recommendation to use linked lists as a basis for the graph object. Since they worked so well for the graph object, I ended up using them all over the place. At the time, I wasn't aware of the other possible built-in data structures, and looking back on it now, there are many places in the program that other data types would have been more appropriate. Thanks to Java's inheritance, polymorphism and generics, it shouldn't be too hard to go back and make these updates. Had I taken CSC-140 just one term earlier, I would have been able to write code that better reflected some of the data that I needed to represent.

Solely using linked lists was the least of my worries during development, though. There were quite a few obstacles that we hadn't considered during planning. Some of the highlights included finding out how to draw the map image onto the screen, dealing with multiple coordinate systems, implementing the Dijkstra algorithm correctly, and reworking other internal functionality issues. For the map drawing, Aaron Cass suggested using a custom `JComponent` (`MapComponent`), which worked out very well. The AWT drawing tools naturally drew the nodes and edges right over the image data, and as a `JComponent`, it was simple integrating it into Swing frames.

The separate coordinate systems relating the mouse to the map were a bit of a challenge, considering that the map coordinates make little sense without a clear understanding of how Java handles components. Implementing Dijkstra's algorithm took much longer than I expected. The first attempt yielded working code, but it was extremely difficult to follow and didn't accurately reflect the algorithm's definition. The current Dijkstra class uses a much cleaner approach.

By the time I got to linking floors together, I realized that including `NodeManager` within `MapComponent` may not have been the best design choice. Since they're so closely related, it does make sense to do this, however, when only the nodes need to be loaded, the memory consumed by the base map goes to waste. When linking floors, the data of both floors must be loaded, linked correctly, and saved. Considering Java's limited heap, the size of these map images, and the image-loading overhead, I found that loading two floors simultaneously lead to out-of-memory errors. Though it may have been better to separate the NodeManager from the start, too much of the program was already dependent on the all-in-one approach. Therefore, I added the `initMap()` method to MapManager to indicate whether or not it should load the corresponding map image. This has shown me how important top-down design is, and how planning ahead really makes a difference.

I'm still working on problems as mentioned in the previous section, related to directional advice using angles, and correctly locating the best stairwell/elevator for travel between floors on a route. In the future I would like to create a more aesthetically-pleasing interface, and expand the project to include both outdoor and indoor locations on campus.

This was a really fun project! I learned quite a bit about graphics and GUI programming, and shortest-path algorithms. “Getting it right” took much revising, and I understand now more than ever before why well- structured code is so important. Finally, I pleased that the program can be used immediately by Union College to help guests and new students navigate our buildings.

## References

Bertsekas, D., & Gallager, R. (1992). *Data Networks* (2 ed.). Englewood Cliffs, NJ: Prentice Hall.

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2001). *Introduction to Algorithms* (2 ed.). Cambridge, MA: The MIT Press; McGraw-Hill Book Company.

Eckel, B. (2002). *Thinking In Java* (3 ed.). Englewood Cliffs, NJ: Prentice Hall.

Greenberg, H. (1999). *Dijkstra's Shortest Path Algorithm* [Java Applet]. `http://carbon.cudenver.edu /~hgreenbe/sessions/dijkst%ra/DijkstraApplet.html`.

*JDK 5.0 Documentation.* (2004). `http://java.sun.com/j2se/1.5.0/docs/`.

Skiena, S. (1997). The Algorithm Design Manual. In (chap. Shortest Path). `http://www2.toki.or.id/book/ AlgDesignManual/BOOK/BOO%K4/NODE162.HTM`: SUNY/Stony Brook, NY; Springer-Verlag.

Wikipedia.org. (2005, December). Graph theory. *http://en.wikipedia.org/wiki/Graph_theory*.

Wikipedia.org. (2006, March). Geographic information system. *http://en.wikipedia.org/wiki/Gis*.

# A Source Code

This code is free for academic use at Union College, but credit must be given if it is incorporated into other projects. It cannot be used in products to be sold without the author's written consent.

## A.1 Console.java

```
//: com:bruceeckel:swing:Console.java
// Tool for running Swing demos from the
// console, both applets and JFrames.
// From 'Thinking in Java, 3rd ed.' (c) Bruce Eckel 2002
// www.BruceEckel.com. See copyright notice in CopyRight.txt.
package com.bruceeckel.swing;
import javax.swing.*;
import java.awt.event.*;

public class Console {
  // Create a title string from the class name:
  public static String title(Object o) {
    String t = o.getClass().toString();
    // Remove the word "class":
    if(t.indexOf("class") != -1)
      t = t.substring(6);
    return t;
  }
  public static void
  run(JFrame frame, int width, int height) {
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(width, height);
    frame.setVisible(true);
  }
  public static void
  run(JApplet applet, int width, int height) {
    JFrame frame = new JFrame(title(applet));
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(applet);
    frame.setSize(width, height);
    applet.init();
    applet.start();
    frame.setVisible(true);
    //System.out.println(frame.getSize());
  }
  public static void
  run(JPanel panel, int width, int height) {
    JFrame frame = new JFrame(title(panel));
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(panel);
    frame.setSize(width, height);
    frame.setVisible(true);
  }
} ///:~
```

## A.2 Dijkstra.java

```
/*
 * Dijkstra.java
 *
 * // TODO: URGENT. The reset() method likely will not reset ALL touched nodes.
 * // Should create a new Set to contain ALL touched nodes throughout process
 * // to ensure everything gets reset on request.
 *
 *
 * This is where the shortest path algorithm goes to work.
 *
 * Created on January 28, 2006, 7:14 PM
 *
 * http://carbon.cudenver.edu/~hgreenbe/sessions/dijkstra/DijkstraApplet.html
 *   used as influence, but no source used.
 *
 * === Semi-Pseudo ============================
 * setConnectedDistances(node):
 *    For each n connected to node, set n's distance values and n's source info
 *    Return the list of connected nodes
 * closestNodeInT():
 *    Return the node in T with the lowest distance value (that's not in P)
 *
 * given start
 * given destination
 * P = {start}          // permanent labels
 * T = {}               // temporary labels
 * next(node) {
```

```
 *   T.addAll(setConnectedDistances(node));
 *   n = closestNodeInT();
 *   T.remove(n);
 *   P.add(n);
 *   if(!p.contains(destination))
 *     next(n);
 * }
 */

package project01;

import java.util.*;
import java.util.concurrent.CopyOnWriteArraySet;

/**
 *
 * author Ian Melnick
*/
public class Dijkstra {
    private static final boolean DEBUG = false;
    private MapComponent.NodeManager nm;
    private NodeItem src, dest;
    private LinkedList<NodeItem> p;
    private CopyOnWriteArraySet<NodeItem> t; // avoids duplicates and potential concurrency problems
    private Dijkstra intermediate;

    /** Creates a new instance of Dijkstra */
    public Dijkstra(MapComponent.NodeManager nm, NodeItem src, NodeItem dest) {
        this.nm   = nm;                           // node manager
        this.src  = src;                          // source node
        this.dest = dest;                         // destination node
        this.p    = new LinkedList<NodeItem>(); // permanently labeled nodes
        this.t    = new CopyOnWriteArraySet<NodeItem>(); // temporarily labeled nodes
        this.intermediate = null;                 // use this for multi-floor searches

        this.src.setDistToMe(0);
        this.p.add(src);
    }

    /** Returns true if source and destination nodes are on different floors */
    private boolean multiFloor() {
        return (src.map().compareTo(dest.map())!=0);
    }

    /** Returns the LinkedList<NodeItem> as a string */
    private String listToString(Collection<NodeItem> L) {
        Iterator<NodeItem> I = L.iterator();
        String ret = "{";
        while(I.hasNext()) {
            ret += (I.next().id());
            if(I.hasNext())
                ret += ",";
        }
        return ret += "}";
    }

    /** Return the closest node in T that's not in P */
    private NodeItem closestNodeInT() {
        Iterator<NodeItem> i = t.iterator();
        NodeItem candidate = null;
        NodeItem next = null;

        try {

            // Set candidate to the first node in T not in P
            do {
                candidate = i.next();
            } while (i.hasNext() && p.contains(candidate));

        } catch(NoSuchElementException err) {
            if(nearestFloorLink(dest.map())==null)
                Errors.debug("No path exists from " + src + " to " + dest);

            p.add(dest);
            return dest;
            //System.exit(1);
        }

        // If anyone else in T has a shorter dist, reset candidate
        while(i.hasNext()) {
            next = i.next();
            if(next.getDistToMe()<candidate.getDistToMe() && !p.contains(next))
                candidate = next;
        }

        return candidate;
    }

    /**
     * Set the Dijkstra-specific distance information on all nodes
     * connected to n, and return the list containing all these connected nodes.
     */
    private LinkedList<NodeItem> setConnectedDistances(NodeItem node) {
        int d;                          // store node distance value
        NodeItem n;                     // temp node
        EdgeItem e;                     // temp edge
        LinkedList<NodeItem> c; // list for nodes to set
        ListIterator<NodeItem> i; // iterates over t

        c = nm.connectedNodes(node);

        //if(DEBUG) System.err.println("Next nodes: " + c);
        i = c.listIterator();
        while(i.hasNext()) {
            n = i.next();
            e = nm.sharedEdge(node,n);

            d = node.getDistToMe() + e.getDistance();

            /*
             * If node's distance hasn't been set yet, OR if the current way
             * is shorter than what it had set before, then set the Dijkstra
             * info in this node to the current best option.
             */
            if(n.getDistToMe() < 0 || d < n.getDistToMe()) {
                n.setDistToMe(d);
                n.setSrcEdge(e);
                n.setSrcNodeId(node.id());
            }
        }

        return c;
    }

    /**
     * Work through nodes and set/update their labels,
     * adding to P when we're sure.
     */
    private void updateLabels(NodeItem node) {
        if(DEBUG) Errors.debug("src:"+src+", dest:"+dest);
        if(DEBUG) Errors.debug("P contains: " + listToString(p));
        NodeItem n = null;
        t.addAll(setConnectedDistances(node));
        t.removeAll(p); // remove nodes already in p
        if(DEBUG) Errors.debug("T contains: " + listToString(t));
        n = closestNodeInT();
        p.add(n);
        //while(t.remove(n)) continue;
        t.remove(n);
        if(!p.contains(dest))
            updateLabels(n);
    }

    /**
     * Return route list, ordered from start to destination.
     * Note that to generate the list we start from the destination and
     * work our way backwards, so if there's a path inconsistency, the
     * returned list will indicate that the user should start at the
     * destination.
     */
    public LinkedList<NodeItem> getRoute() {
        /* Pull nodes from P starting from end, trace back by edges.
         * Loop:
         *   while src not in r,      (contains())
         *     myDistNum = lastDistNum - lastEdgeLength
         *     myNode = look for node in P which has myDistNum
         *     r.addFirst(myNode)
         */
        LinkedList<NodeItem> r = new LinkedList<NodeItem>();
        r.add(this.dest);

        int distNum;
        NodeItem ln, tn;

        // Single floor—normal op
        while(!r.contains(src)) {
            ln = r.getFirst();
            //distNum = (ln.getDistToMe()) - (ln.getSrcEdge().getDistance());
            tn = nm.getNodeById(ln.getSrcNodeId());
            if(tn!=null)
                r.addFirst(tn);
            else {
                Errors.debug("no source for node " + ln);
                break;
            }
        }

        // If intermediate exists, must add in intermediate's route first
        if(intermediate!=null)
            r.addAll(intermediate.getRoute());

        return r;
    }

    public LinkedList<NodeItem> getP() { return this.p; }
    public void run() {

        // First check if source and destination floors are different
        // If they aren't the same, get route details from floor link
        // to destination, which will be used later when returning the
        // correct route.
        if(multiFloor()) {
            NodeItem localLink = nearestFloorLink(dest.map());
            NodeItem otherLink = localLink.floorLink(dest.map());
            MapComponent intmc = new MapComponent(dest.map());
            intmc.nm.loadNodes();
            NodeItem otherSrc = intmc.nm.getNodeById(otherLink.id());
            NodeItem otherDest = intmc.nm.getNodeById(dest.id());
            intermediate = new Dijkstra(intmc.nm, otherSrc, otherDest);
            intermediate.run();
            dest = localLink;
        }

        // Then run normal algorithm
        updateLabels(src);

    }

    public void reset() {
        NodeItem tn;
        ListIterator<NodeItem> tI;
        tI = p.listIterator();
        while(tI.hasNext()) {
            (tI.next()).resetDijkstraValues();
        }
    }


    /* FLOOR LINK SEARCHING */

    /**
     * Returns true if link either directly links to destMap
     * or links to another floor which ultimately links to destMap.
     */
    private boolean linksToFloor(NodeItem link, String destMap) {
        if(!link.isFloorLink())
            return false;
        if(link.hasFloorLink(destMap))
            return true;
        else {
            Set<String> floors = link.connectedFloors();
            Iterator<String> f = floors.iterator();
            while(f.hasNext())
                return linksToFloor(link.floorLink(f.next()), destMap);
        }
        return false;
    }

    /**
     * Returns the closest floor linking node
     * from the source node that can provide a connection
     * path to the destination's floor (i.e., through other
     * floor links).
     */
    private NodeItem nearestFloorLink(String destMap) {
        LinkedList<NodeItem> nodesLookedAt = new LinkedList<NodeItem>();
        return nearestFloorLink(src, destMap, nodesLookedAt);
    }
    private NodeItem nearestFloorLink(NodeItem source, String destMap, LinkedList<NodeItem> nodesLookedAt) {
        if(linksToFloor(source, destMap))
            return source;

        nodesLookedAt.add(source);
        Iterator<NodeItem> i = nm.connectedNodes(source).iterator();
        NodeItem n;
        while(i.hasNext()) {
            n = i.next();
            if(!nodesLookedAt.contains(n)) {
                nodesLookedAt.add(n);
                return nearestFloorLink(n, destMap, nodesLookedAt);
            }
        }
        return null;
    }

}
```

## A.3 DirectionsFrame.java

```
/*
 * DirectionsFrame.java
 *
 * Created on January 8, 2006, 5:45 PM
 */

//http://java.sun.com/developer/techDocs/hi/repository/TBG_Navigation.html
//http://java.sun.com/docs/books/tutorial/uiswing/components/toolbar.html
//http://java.sun.com/docs/books/tutorial/uiswing/components/scrollpane.html

package project01;

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;


/**
 * Frame responsible for stepping user through point-to-point directions.
 *
 * author Ian Melnick
*/
public class DirectionsFrame {
    JFrame window;
    JTextArea directionBox;
    int curDirection;
    String curMap;
    JScrollPane scrollPane;
    JToolBar toolBar;
    JButton bPrevious, bNext;
    Dijkstra d;
    MapComponent.MapManager mm;
    LinkedList<NodeItem> r;
    Router router;

    /** Creates a new instance of DirectionsFrame */
    public DirectionsFrame(Router router, Dijkstra d) {
        this.d = d;
        this.router = router;
        this.mm = router.getMapManager();
        ButtonMethods bm = new ButtonMethods();
        curDirection = 0;
        curMap = router.getMap();

        window = new JFrame("Steps");
        window.setLayout(new BorderLayout());
        window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        bPrevious = new JButton();
            bPrevious.setActionCommand("previous");
            bPrevious.setToolTipText("Previous Instruction");
            bPrevious.addActionListener(bm);
            bPrevious.setText("<<");

        bNext = new JButton();
            bNext.setActionCommand("next");
            bNext.setToolTipText("Next Instruction");
            bNext.addActionListener(bm);
            bNext.setText(">>");

        toolBar = new JToolBar("Navigation");
        toolBar.add(bPrevious);
        toolBar.add(bNext);

        directionBox = new JTextArea(); //10, 30
        directionBox.setEditable(false);
        scrollPane = new JScrollPane(directionBox);

        window.add(toolBar, BorderLayout.NORTH);
        window.add(scrollPane, BorderLayout.SOUTH);


        d.run();
        r = d.getRoute();
        showDirections(curDirection);
        d.reset();


        window.pack();
        window.setVisible(true);
    }


    private void showDirections(int curDirectionNumber) {
        directionBox.setText(null);
        NodeItem prvNode, curNode, nxtNode;
        for(int i=0; i<r.size(); i++) {
            if(i==0)             directionBox.append("Start at:\t");
            else if(i==r.size()−1) directionBox.append("End at:\t");
            else                 directionBox.append("Go to:\t");
            if(i<r.size()-1)      nxtNode = r.get(i+1);
            else                  nxtNode = null;
            if(i>0)               prvNode = r.get(i-1);
            else                  prvNode = null;
            curNode = r.get(i);

            double angle = -1;
            if(nxtNode!=null && prvNode!=null) {
                Point c = prvNode.location();
                Point p = curNode.location();
                Point n = nxtNode.location();

                Point u, v; //line vectors
                u = new Point((int)(c.getX()-p.getX()),(int)(c.getY()-p.getY()));
                v = new Point((int)(n.getX()-p.getX()),(int)(n.getY()-p.getY()));

                double dp, mu, mv; //dot product, magnitude u, magnitude v
                dp = (u.getX()*v.getX()) + (u.getY()*v.getY());
                mu = Math.sqrt(Math.pow(u.getX(),2)+Math.pow(u.getY(),2));
                mv = Math.sqrt(Math.pow(v.getX(),2)+Math.pow(v.getY(),2));

                angle = Math.toDegrees(Math.acos( dp / (mu*mv) ));
            }

            // TODO: Angle isn't enough to determine turning direction.

            //directionBox.append(String.valueOf(r.get(i).id()));
            /*if(angle > 85 && angle < 95) {
                directionBox.append("Continue\n");
                continue;
            }*/
            directionBox.append(curNode.description()); // + "\t"+angle
            if(i==curDirectionNumber) {
                directionBox.append("\t<<<");
                if(curNode.map().compareTo(curMap)!=0) {
                    curMap = curNode.map();
                    router.setMap(curMap);
                    router.update();
                    mm = router.getMapManager();
                }
            }
            directionBox.append("\n");
        }
    }


    /**
     * Listener methods for button actions
     */
    private class ButtonMethods implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if(e.getSource() == bNext) {
                if(curDirection<r.size()-1)
                    curDirection++;
                showDirections(curDirection);
                mm.centerAroundPoint(r.get(curDirection).location());
            }
            if(e.getSource() == bPrevious) {
                if(curDirection>0)
                    curDirection--;
                showDirections(curDirection);
                mm.centerAroundPoint(r.get(curDirection).location());
            }
        }
    }

}
```

## A.4 EdgeItem.java

```
/*
 * EdgeItem.java
 *
 * Created on December 12, 2005, 5:04 PM
 */

package project01;


import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

/**
 * Describes an edge on a map.
 *
 * author Ian Melnick
*/
public class EdgeItem extends MapItem implements Serializable {
    private int distance; // distance in "paces"

    /** Creates a new instance of EdgeItem */
    public EdgeItem(String map, int id) {
        this.map      = map;
        this.id       = id;
        this.distance = 0;
    }

    public int getDistance()      { return this.distance;       }
    public void setDistance(int paces) { this.distance = paces; }
}
```

## A.5 Editor.java

```
/*
 * Editor.java
 *
 * Created on September 14, 2005, 11:27 PM
 *
 * To change this template, choose Tools | Options and locate the template under
 * the Source Creation and Management node. Right-click the template and choose
 * Open. You can then make changes to the template in the Source Editor.
 */
// <applet code='Editor' width='200' height='50'></applet>

package project01;

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;
//import com.bruceeckel.swing.*;
//import java.io.*;


/**
 * Sets up the environment and runs the Map Data Editor program;
 * contains listener methods that work with the {link MapComponent}.
 *
 * author Ian Melnick
 */
public class Editor extends Main {

    private JButton bLoad, bSave;
    private JSlider sPanX, sPanY;
    private JPanel bottom, bottomL, bottomC, bottomR;
    private JLabel status, message;
    private MapComponent imc;

    public Editor(Args a) {
        //a      = Utility.getArgs(args, this);
        this.a = a;

        bLoad = new JButton("Load");
        bSave = new JButton("Save");
        imc   = new MapComponent(a.map);
        imc.initMap(true);
        sPanX = new JSlider(JSlider.HORIZONTAL, −(int)imc.getMapSize().getWidth(), 0, (int)imc.getCoordinates().getX());
        sPanY = new JSlider(JSlider.VERTICAL, −(int)imc.getMapSize().getHeight(), 0, (int)imc.getCoordinates().getY());
        bottom = new JPanel(new GridLayout(1,3));
        bottomL = new JPanel(new FlowLayout());
        bottomC = new JPanel(new FlowLayout());
        bottomR = new JPanel(new FlowLayout());
        status = new JLabel(" ");
        message = new JLabel(" ");
    }


    /**
     * Listener methods for mouse actions
     */
    private class MouseMethods extends MouseInputAdapter implements MouseWheelListener {
        private Point mouseClickPoint, mouseReleasePoint, newNode;
        private boolean mouseDragged, scrollDirection; // scroll direction = false, use y, true use x
        public void mouseMoved(MouseEvent e) {
            status.setText("map: " + a.map + " dc: (" +
                    (int)e.getPoint().getX()+","+(int)e.getPoint().getY() +
                    ")" + " mc: (" +
                    (int)imc.mm.mouseCoordsToMapCoords(e.getPoint()).getX()+","+
                    (int)imc.mm.mouseCoordsToMapCoords(e.getPoint()).getY() + ")"
            );
        }
        public void mouseClicked(MouseEvent e) {
            if((e.getButton()==1 || e.getButton()==3) && e.getClickCount()==2) {
                // Check if it was pressed over a visible node
                // Loop thru nodes and verify if mouse coords are within boundaries of any of the visible sections.
                //System.out.println("Pressed at: " + e.getPoint());

                // If double-click over a node, edit it
                // If no node, create node at click point
                NodeItem selNode = imc.nm.getNodeAtCoords(e.getPoint());
                try {
                    //System.out.println("Tapped node: " + selNode.location());
                    selNode.select();
                    imc.repaint();
                    if(selNode.selected()) {
                        NodeEditor npm = new NodeEditor(selNode, imc);
                    }
                } catch (java.lang.NullPointerException error) {
                    //System.out.println("The third button is reserved for node operations.");
                    // Create node at specified point.
                    // Point needs to be offset from the current window coords so it's relative to map instead
                    newNode = new Point(
                        (int)(imc.getCoordinates().getX() − e.getPoint().getX()),
                        (int)(imc.getCoordinates().getY() − e.getPoint().getY())
                    );
                    setMessage("Created Node");
                    imc.nm.addNode(newNode);
                }
            }
        }
        public void mousePressed(MouseEvent e) {
            this.mouseClickPoint = e.getPoint();
            if(e.getButton()==2)
                this.scrollDirection = !this.scrollDirection;
        }
        public void mouseDragged(MouseEvent e) {
            this.mouseDragged = true;
        }
        public void mouseReleased(MouseEvent e) {
            this.mouseReleasePoint = e.getPoint();
            if(this.mouseDragged && !this.mouseReleasePoint.equals(this.mouseClickPoint)) {
                // If dragged over node, create edge
                // If dragged over nothing, move node
                if(e.getButton()==1 || e.getButton()==3) {
                    NodeItem selNode = imc.nm.getNodeAtCoords(this.mouseClickPoint);
                    NodeItem relNode = imc.nm.getNodeAtCoords(this.mouseReleasePoint);
                    EdgeItem edge = null;
                    try {
                        if(relNode!=null && !imc.nm.connected(selNode, relNode)) {
                            edge = new EdgeItem(imc.map(), ++imc.edgeId);
                            edge.setDistance(imc.calcDistance(selNode,relNode));
                            selNode.addEdge(edge);
                            relNode.addEdge(edge);
                            setMessage("Created Edge " + edge.id());
                        }
                        else { // Move node
                            selNode.setLocation(imc.mm.mouseCoordsToMapCoords(this.mouseReleasePoint));
                            NodeItem otherNode = null;
                            ListIterator<NodeItem> i = imc.nm.connectedNodes(selNode).listIterator();
                            while(i.hasNext()) {
                                otherNode = i.next();
                                edge = imc.nm.sharedEdge(otherNode, selNode);
                                edge.setDistance(imc.calcDistance(otherNode, selNode));
                            }
                        }
                        imc.repaint();
                    } catch (java.lang.NullPointerException error) {
                        // move map when no node available
                        imc.mm.offsetByMouse(this.mouseClickPoint, this.mouseReleasePoint);
                        sPanY.setValue((int)imc.getCoordinates().getY());
                        sPanX.setValue((int)imc.getCoordinates().getX());
                    }
                }

                this.mouseDragged=false;
            }
        }
        public void mouseWheelMoved(MouseWheelEvent e) {
            Point newCoords = new Point();
            if(this.scrollDirection) {
                newCoords.setLocation( imc.getCoordinates().getX()−(e.getWheelRotation()*50), imc.getCoordinates().getY() );
                sPanX.setValue((int)newCoords.getX());
            }
            else {
                newCoords.setLocation( imc.getCoordinates().getX(), imc.getCoordinates().getY()−(e.getWheelRotation()*50) );
                sPanY.setValue((int)newCoords.getY());
            }
            imc.mm.setOrigin(newCoords);
        }
    }

    /**
     * Listener methods for button actions
     */
    private class ButtonMethods implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if(e.getSource() == bSave) {
                imc.nm.saveNodes();
                setMessage("Saved");
            }
            if(e.getSource() == bLoad) {
                imc.nm.loadNodes();
                imc.repaint();
                setMessage("Loaded");
            }
        }
    }

    /**
     * Listener methods for slider actions
     */
    private class SliderMethods implements ChangeListener {
        public void stateChanged(ChangeEvent e) {
            JSlider s = (JSlider)e.getSource();
            Point newCoords = new Point();
            if(!s.getValueIsAdjusting()) {
                if(e.getSource()==sPanY)
                    newCoords.setLocation( imc.getCoordinates().getX(), s.getValue() );
                else if(e.getSource()==sPanX)
                    newCoords.setLocation( s.getValue(), imc.getCoordinates().getY() );
                imc.mm.setOrigin(newCoords);
            }
        }
    }


    public void setMessage(String m) {
        message.setText(m);
    }


    public String getAppletInfo() {
        return "(c) Ian Melnick, Union College 2005-2006";
    }
    public String[][] getParameterInfo() {
        String pinfo[][] = {
            {"m","String","Map file to load (specify location/floor only, i.e., map_scieneg_000.jpg is loaded with arg scieng_000)"}
            /*{"w","Integer","Width of map window (px)"},
            {"h","Integer","Height of map window (px)"}*/
        };
        return pinfo;
    }


    public void init() {
        MouseMethods mm = new MouseMethods();
        ButtonMethods bm = new ButtonMethods();
        SliderMethods sm = new SliderMethods();

        bLoad.addActionListener(bm);
        bSave.addActionListener(bm);
        sPanX.addChangeListener(sm);
        sPanY.addChangeListener(sm);
        imc.addMouseListener(mm);
        imc.addMouseMotionListener(mm);
        imc.addMouseWheelListener(mm);

        bottomL.add(bLoad);
        bottomL.add(bSave);
        bottomC.add(status);
        bottomR.add(message);
        bottom.add(bottomL);
        bottom.add(bottomC);
        bottom.add(bottomR);

        Container cp = getContentPane();
        cp.setLayout(new BorderLayout());
        cp.add(sPanX, BorderLayout.NORTH);
        cp.add(sPanY, BorderLayout.WEST);
        cp.add(bottom, BorderLayout.SOUTH);
        cp.add(imc);

        //System.out.println(this.getSize());
    }
}
```

## A.6 Errors.java

```
/*
 * project01: Description
 * Ian Melnick (dazed)
 * March 17, 2006, 2:04 PM
 * Course, Prof
 *
 * Errors.java
 *
 * Exception/Thread tracing info provided in:
 * http://www.rgagnon.com/javadetails/java-0420.html
 */

package project01;
```

```
/**
 *
 * author Ian Melnick
*/
public class Errors {

    private static String source() {
        StackTraceElement method = Thread.currentThread().getStackTrace()[4];
        StackTraceElement caller = Thread.currentThread().getStackTrace()[5];
        return method.getClassName() + "."
                + method.getMethodName() + "()"
                + " ["+caller.getMethodName() + ":"
                + caller.getLineNumber()+"]";
    }

    public static void debug(Object message) {
        System.out.println(source() + " " + message);
    }

}
```

## A.7 FloorLinker.java

```
/*
 * FloorLinker.java
 *
 * Created on February 7, 2006, 2:11 PM
 */

package project01;

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;

/**
 * A small map window that allows the user to select a floor-link node.
 * Intended to be launched from Editor mode, in NodeEditor in the Floors tab,
 * when user selects "Add New Link". This is not a primary mode window,
 * and therefore is not intended to be run on application launch. Cannot be
 * run as an applet.
 *
 * author Ian Melnick
*/
public class FloorLinker {
    private JFrame window;
    private MapComponent imc;
    private NodeItem source, destination;
    private String srcMap, destMap;

    public FloorLinker(String srcMap, String destMap, NodeItem source, Point location) {
        this.source = source;
        this.destination = null;
        this.srcMap = srcMap;
        this.destMap = destMap;

        window = new JFrame("Select corresponding node on " + destMap);
        window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        window.setSize(500, 350);

        imc = new MapComponent(destMap);
        imc.nm.loadNodes();
        imc.initMap();
        imc.mm.centerAroundPoint(location);

        MouseMethods mm = new MouseMethods();
        imc.addMouseListener(mm);
        imc.addMouseMotionListener(mm);
        imc.addMouseWheelListener(mm);

        Container cp = window.getContentPane();
        cp.setLayout(new BorderLayout());
        cp.add(imc);

        window.setVisible(true);
    }

    /**
     * Deletes nodeId in destMap from having a floor connection to srcMap.
     */
    public static void deleteConnection(String srcMap, String destMap, int nodeId) {
        MapComponent otherFloor = new MapComponent(destMap);
        otherFloor.nm.loadNodes();
        NodeItem next = otherFloor.nm.getNodeById(nodeId);
        next.delFloorLink(srcMap);
        otherFloor.nm.saveNodes();
    }

    /**
```

```
     * Listener methods for mouse actions
     */
    private class MouseMethods extends MouseInputAdapter implements MouseWheelListener {
        private Point mouseClickPoint, mouseReleasePoint, newNode;
        private boolean mouseDragged, scrollDirection; // scroll direction = false, use y, true use x
        public void mouseMoved(MouseEvent e) {
        }
        public void mouseClicked(MouseEvent e) {
            if((e.getButton()==1 || e.getButton()==3) && e.getClickCount()==2) {
                // Check if it was pressed over a visible node
                // Loop thru nodes and verify if mouse coords are within boundaries of any of the visible sections.
                //System.out.println("Pressed at: " + e.getPoint());

                // If double-click over a node, edit it
                destination = imc.nm.getNodeAtCoords(e.getPoint());

                // Set up the links between nodes
                try {
                    destination.addFloorLink(srcMap, source);
                    source.addFloorLink(destMap, destination);
                    imc.nm.saveNodes();
                } catch (NullPointerException err) {
                    // No node double-clicked...
                }
                window.dispose();
            }
        }
        public void mousePressed(MouseEvent e) {
            this.mouseClickPoint = e.getPoint();
            if(e.getButton()==2)
                this.scrollDirection = !this.scrollDirection;
        }
        public void mouseDragged(MouseEvent e) {
            this.mouseDragged = true;
        }
        public void mouseReleased(MouseEvent e) {
            this.mouseReleasePoint = e.getPoint();
            if(this.mouseDragged && !this.mouseReleasePoint.equals(this.mouseClickPoint)) {
                // If dragged over node, create edge
                // If dragged over nothing, move node
                if(e.getButton()==1 || e.getButton()==3) {
                    NodeItem selNode = imc.nm.getNodeAtCoords(this.mouseClickPoint);
                    NodeItem relNode = imc.nm.getNodeAtCoords(this.mouseReleasePoint);
                    try {
                        if(relNode!=null && !imc.nm.connected(selNode, relNode)) {
                            EdgeItem edge = new EdgeItem(imc.map(), ++imc.edgeId);
                            selNode.addEdge(edge);
                            relNode.addEdge(edge);
                        }
                        else // Move node
                            selNode.setLocation(imc.mm.mouseCoordsToMapCoords(this.mouseReleasePoint));
                        imc.repaint();
                    } catch (java.lang.NullPointerException error) {
                        // move map when no node available
                        imc.mm.offsetByMouse(this.mouseClickPoint, this.mouseReleasePoint);
                        //sPanY.setValue((int)imc.getCoordinates().getY());
                        //sPanX.setValue((int)imc.getCoordinates().getX());
                    }
                }

                this.mouseDragged=false;
            }
        }
        public void mouseWheelMoved(MouseWheelEvent e) {
            Point newCoords = new Point();
            if(this.scrollDirection) {
                newCoords.setLocation( imc.getCoordinates().getX()−(e.getWheelRotation()*50), imc.getCoordinates().getY() );
                //sPanX.setValue((int)newCoords.getX());
            }
            else {
                newCoords.setLocation( imc.getCoordinates().getX(), imc.getCoordinates().getY()−(e.getWheelRotation()*50) );
                //sPanY.setValue((int)newCoords.getY());
            }
            imc.mm.setOrigin(newCoords);
        }
    }

}
```

## A.8 Main.java

```
/*
 * Main.java
 *
 * Created on January 5, 2006, 10:49 AM
 */

package project01;

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import com.bruceeckel.swing.*;
import java.lang.*;


/**
 *
 * author Ian Melnick
*/
public class Main extends JApplet {
    enum AppModes { Editor, Router }
    public Args a;


    /** Creates a new instance of Main */
    public Main() {
        a = Utility.getArgs(new String[0], this);
    }
    public Main(String[] args) {
        a = Utility.getArgs(args, this);
    }

    public void launch() {
        switch(a.mode) {
            case Router:
                Console.run(new Router(a), (int)a.resolution.getWidth(), (int)a.resolution.getHeight());
                break;
            case Editor:
                Console.run(new Editor(a), (int)a.resolution.getWidth(), (int)a.resolution.getHeight());
                break;
        }
    }


    /**
     * param args the command line arguments
*/
    public static void main(String[] args) {

        Main prog = new Main(args);
        prog.launch();

        /*} catch (java.lang.OutOfMemoryError err) {
            JOptionPane.showMessageDialog(null, "Ran out of memory; try increasing heap size. If using browser, increase through Java control panel—look for Runti
            System.out.println("Ran out of memory, start with increased heap size: -Xmx96m");
            System.exit(1);
        }*/


    }

}
```

## A.9 MapComponent.java

```
/*
 * MapComponent.java
 *
 * Created on January 8, 2006, 1:50 PM
 */
// <applet code='Main' width='200' height='50'></applet>

package project01;

import javax.swing.*;
import java.awt.*;
import java.util.*;
import com.bruceeckel.swing.*;
import java.lang.*; // Runtime
import java.io.*;   // IO Streams
import java.net.URL;
//import java.beans.*; // XMLEncoder/Decoder
//import org.devlib.schmidt.imageinfo.*; // for getting image dpi
import com.sun.image.codec.jpeg.*;   // for getting image dpi

/**
 * Manages and draws nodes and edges.
 * <p>
 * Note: Original name was "InteractiveMapComponent" so "imc" is found
 * throughout src when using it
 *
 * author Ian Melnick
*/
public class MapComponent extends JComponent {

    // class vars—current coords for drawing, etc etc.
    public NodeManager nm;
    public MapManager mm;
    String              map;          // map name to use/load
    URL                 mapPath;      // location of image
    ImageIcon           mapImage;     // stores the image data
    //JPEGImageDecoder mapImageInfo; // contains image resolution in dpi
    Point               mapCoords; // current image coordinates relative to top-left corner of screen
    Dimension           mapSize;      // loaded map image dimensions
    LinkedList<NodeItem> nodes; // array of nodes
    int                 nodeSize;     // visible node diameter
    int                 nodeId;       // node counter for setting ID numbers
    public int          edgeId;       // edge counter for setting ID numbers
    Runtime             r;            // gets info such as memory use
    String              dataDir;      // directory to save/load data to/from
    boolean             drawInvis;    // draw invisible nodes?


    public MapComponent(String map) {
        nm        = new NodeManager();
        mm        = new MapManager();
        r         = Runtime.getRuntime();
        this.map = map;
        nodeSize = 20;
        drawInvis = false;

        mapPath = MapComponent.class.getResource("images/map_"+map+".jpg"); // http://java.sun.com/docs/books/tutorial/uiswing/misc/icon.html
        if(mapPath==null) {
            JOptionPane.showMessageDialog(null, "Requested map not found", "Error", JOptionPane.ERROR_MESSAGE);
            System.exit(1);
        }

        // System.getenv requires Java 1.5
        if(System.getenv("HOMEDRIVE")==null) {
            dataDir = System.getenv("HOME");
        }
        else {
            dataDir = System.getenv("HOMEDRIVE") + System.getenv("HOMEPATH");
        }

        // initMap()

        nodes     = new LinkedList<NodeItem>();
        nodeId    = 0;
        edgeId    = 0;
    }
    public void initMap(boolean drawInvisibles) {
        this.drawInvis = drawInvisibles;
        this.initMap();
    }
    public void initMap() {
        // Set up mapImageInfo
        /*try {
            mapImageInfo = JPEGCodec.createJPEGDecoder(mapPath.openStream());
        } catch (IOException err) {
            System.err.println("MapComponent: mapImageInfo: " + err);
        }*/

        // Set up mapImage
        mapImage = new ImageIcon(mapPath);

        Errors.debug("Image: " + mapImage);// + " (at: " + mapImageInfo.getJPEGDecodeParam() + ")");
        Errors.debug("Mem used: " + r.totalMemory() + " bytes");
        Errors.debug("Mem max : " + r.maxMemory() + " bytes");
        Errors.debug("Mem free: " + r.freeMemory() + " bytes");

        mapCoords = new Point(0, 0); // -800, -500
        mapSize  = new Dimension(mapImage.getIconHeight(), mapImage.getIconWidth());
    }
    /*public void paintComponent(Graphics g) {
        //super.paintComponent(g);
    }*/
    //note: paintComponent used to be in separate files for each
    // primary mode window. Since I never changed it for any of
    // these modes, I've merged it back into MapComponent...
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        mapImage.paintIcon(this, g, (int)this.mapCoords.getX(), (int)this.mapCoords.getY());

        // loop thru nodes and draw on map relative to current image position.
        ListIterator<NodeItem> nli, nlj;
        NodeItem tn, un;
        EdgeItem ec;
        Point tp, up;
        Point minCorner = mm.mapMinCorner();
        Point maxCorner = mm.mapMaxCorner();

        nli = this.nodes.listIterator(); // For drawing each node
        while(nli.hasNext()) {
            tn = nli.next();

            // Draw/print node if it's within current shown coordinates
            // 1. Get node map position
            // 2. Determine if it's currently visible:
            //    a. Get component size // this.getWidth(), this.getHeight();
            //    b. Based on origin coordinates and window size, are these coords currently visible?
            // 3. Draw oval around point

            // Max corner is current map coordinate
            // at origin (this.getCoordinates()) + window size (this.getWidth(), this.getHeight()).
            tp = mm.positiveMapCoords(tn.location());

            // If the current node is in between the 1st corner and the max corner then it should be shown,
            // along with its edges (next block)
            if( (tp.getX() <= maxCorner.getX() && tp.getX() > minCorner.getX()) &&
                (tp.getY() <= maxCorner.getY() && tp.getY() > minCorner.getY())) {
                // Draw oval relative to current display coords
                //g.fillOval( ((int)(tpos.getX()-minCorner.getX()-(nodeSize/2))), ((int)(tpos.getY()-minCorner.getY()-(nodeSize/2))), nodeSize, nodeSize );
                if(tn.selected())
                    drawNode(g, Color.red, tn);
                else if(!tn.selected()) {
                    if(tn.visibility())
                        drawNode(g, Color.green, tn);
                    else if(!tn.visibility() && drawInvis)
                        drawNode(g, Color.yellow, tn);
                }

                // Find which node(s) are related to this one and draw connecting edges.
                // IAN DRAW DISTANCE FOR LINE HERE . . . use ec = sharedEdge(ns, nd)
                g.setColor(Color.black);
                tp = mm.mapCoordsToMouseCoords(tn.location());
                nlj = nm.connectedNodes(tn).listIterator();
                while(nlj.hasNext()) {
                    un = nlj.next();
                    up = mm.mapCoordsToMouseCoords(un.location());
                    ec = nm.sharedEdge(tn, un);
                    g.drawLine((int)tp.getX(), (int)tp.getY(), (int)up.getX(), (int)up.getY());
                    g.drawString(String.valueOf(ec.getDistance()), (int)( ((up.getX()−tp.getX())/2)+tp.getX() ), (int)(((up.getY()−tp.getY())/2)+tp.getY()) );
                }

            }
        }
    }
    void drawNode(Graphics g, Color c, NodeItem n) {
        g.setColor(c);
        Point minCorner = mm.mapMinCorner();
        Point p = new Point(mm.positiveMapCoords(n.location()));
        g.fillOval( ((int)(p.getX()−minCorner.getX()−(nodeSize/2))), ((int)(p.getY()−minCorner.getY()−(nodeSize/2))), nodeSize, nodeSize );
        g.setColor(Color.black);
        g.drawString(String.valueOf(n.id()), ((int)(p.getX()−minCorner.getX()−(nodeSize/2))), ((int)(p.getY()−minCorner.getY()−(nodeSize/2))));
    }

    public Point getCoordinates() { return mapCoords; }
    public Dimension getMapSize() { return mapSize; }
    public String map()     { return map; }

    /** Return distance in Feet between two nodes */
    public int calcDistance(NodeItem a, NodeItem b) {
        // TODO: Both sun and schmidt libraries don't extract dpi, but windows does!
        /*
         * Pixel distance:
         *   x = selNode.location().getX() - relNode.location().getX()
         *   y = selNode.location().getY() - relNode.location().getY()
         * x.toPositive()
         * y.toPositive()
         *
         * Convert pixels to inches, to feet:
         *   feetwidth = (imc.getMapSize().getWidth() / getMapHorizDPI) * 40
         *   feetheight= (imc.getMapSize().getHeight() / getMapVertDPI) * 40
         */
        Point distancePx = mm.positiveMapCoords(mm.difference(a.location(), b.location()));
        int mapHorizDPI = 96; // TODO: Needs to be dynamic!
        int mapVertDPI = 96;
        int mapScale = 40; // constant; 1" = 40'
        int feetWidth = (int)((distancePx.getX() / mapHorizDPI) * mapScale);
        int feetHeight = (int)((distancePx.getY() / mapVertDPI) * mapScale);
        int distanceFt = (int)Math.sqrt(Math.pow(feetWidth,2)+Math.pow(feetHeight,2));
        return distanceFt;
    }

    class MapManager {
        public void setOrigin(Point p) {
            // Check that we're still on the map; if yes, redraw
            if( (p.getY() > −mapSize.getHeight() && p.getY() <= 0) &&
                (p.getX() > −mapSize.getWidth() && p.getX() <= 0) ) {
                mapCoords.setLocation(p);
                repaint();
            }
        }
```

```java
        public void centerAroundPoint(Point p) {
            p = positiveMapCoords(p);
            Point newloc = new Point( −(int)(p.getX()−(getWidth()/2)), −(int)(p.getY()−(getHeight()/2)) );
            setOrigin(newloc);
        }
        public Point mouseCoordsToMapCoords(Point mouseCoords) {
            Point minCorner = mapMinCorner();
            //Point maxCorner = new Point((int)(-this.getCoordinates().getX()+this.getWidth()), (int)(-this.getCoordinates().getY()+this.getHeight()));
            return new Point( (int)(−mouseCoords.getX()−minCorner.getX()), (int)(−mouseCoords.getY()−minCorner.getY()) );
        }
        public Point mapCoordsToMouseCoords(Point mapCoords) {
            Point minCorner = mapMinCorner();
            Point p = new Point(positiveMapCoords(mapCoords));
            return new Point( (int)(p.getX()−minCorner.getX()), (int)(p.getY()−minCorner.getY()) );
        }
        public Point positiveMapCoords(Point negativeMapCoords) {
            return new Point( (int)(Math.abs(negativeMapCoords.getX())), (int)(Math.abs(negativeMapCoords.getY())) );
        }
        public Point mapMinCorner() {
            return new Point(positiveMapCoords(getCoordinates()));
        }
        public Point mapMaxCorner() {
            return new Point((int)(−getCoordinates().getX()+getWidth()), (int)(−getCoordinates().getY()+getHeight()));
        }
        public void offsetByMouse(Point mcp, Point mrp) {
            Point oc = new Point((int)(mrp.getX()−mcp.getX()), (int)(mrp.getY()−mcp.getY()));
            Point mc = new Point((int)(getCoordinates().getX()+oc.getX()), (int)(getCoordinates().getY()+oc.getY()));
            setOrigin(mc);
        }
        public Point difference(Point a, Point b) {
            return new Point((int)(a.getX()−b.getX()),(int)(a.getY()−b.getY()));
        }
    }

    class NodeManager {
        public void addNode(Point p) {
            nodes.add(new NodeItem(map, p, ++nodeId));
            repaint();
        }
        public void delNode(NodeItem n) {
            NodeItem c;
            EdgeItem ei;
            ListIterator<NodeItem> l = connectedNodes(n).listIterator();
            Iterator<EdgeItem> e;

            while(l.hasNext()) {
                c = l.next();
                e = n.edges().iterator();
                while(e.hasNext()) {
                    try {
                        ei = e.next();
                        if(c.hasEdge(ei))
                            c.delEdge(ei);
                    } catch (java.util.ConcurrentModificationException er) {
                        //continue;
                        break;
                    }
                }
            }
            nodes.remove(n);
            repaint();
        }
        public NodeItem getNodeAtCoords(Point p) {
            p = mm.positiveMapCoords(mm.mouseCoordsToMapCoords(p));
            ListIterator<NodeItem> nli = nodes.listIterator();
            Point m;
            Object tobj;
            NodeItem tnod, rnod;
            rnod = null;
            while(nli.hasNext()) {
                tnod = nli.next();
                m = mm.positiveMapCoords(tnod.location());

                // If p is within tnod.location() and tnod.location()+nodeSize
                // Make sure that coords being compared are relative to screen, like passed mouse coords...
                if( (p.getX() >= m.getX()−(nodeSize) && p.getX() < m.getX()+(nodeSize)) &&
                    (p.getY() >= m.getY()−(nodeSize) && p.getY() < m.getY()+(nodeSize))) {
                    rnod = tnod;
                    break;
                }
                else
                    rnod = null;
            }
            return rnod;
        }
        public NodeItem getNodeById(int nid) {
            NodeItem n;
            ListIterator<NodeItem> nli = nodes.listIterator();
            while(nli.hasNext()) {
                n = nli.next();
                if(n.id()==nid)
                    return n;
            }
            return null;
        }
        public boolean connected(NodeItem ns, NodeItem nd) {
            if(connectedNodes(ns).contains(nd))
                return true;
            else
                return false;
        }
        public LinkedList<NodeItem> connectedNodes(NodeItem n) {
            /* Return a LinkedList of nodes connected to n by common edges */
            // NOTE: Using Generics, no longer need to cast objects when retrieving from List
            ListIterator<NodeItem> l;
            Iterator<EdgeItem> e;
            NodeItem c;
            LinkedList<NodeItem> r = new LinkedList<NodeItem>();
            LinkedList<NodeItem> tnodes = new LinkedList<NodeItem>(nodes);
            tnodes.remove(n);

            l = tnodes.listIterator(); // For finding which node contains edge relationship
            while(l.hasNext()) {
                c = l.next();
                e = n.edges().iterator();
                while(e.hasNext())
                    if(c.hasEdge( e.next() ))
                        r.add(c);
            }

            return r;
        }
        public EdgeItem sharedEdge(NodeItem ns, NodeItem nd) {
            EdgeItem ei;
            Iterator<EdgeItem> el = ns.edges().iterator();
            while(el.hasNext()) {
                ei = el.next();
                try {
                    if(nd.hasEdge(ei))
                        return ei;
                } catch(NullPointerException err) {
                    continue;
                }
            }
            return null;
        }
        /** Generates a list of visible nodes for use in a JComboBox */
        public Vector<MapItem> nodeVector() {
            Vector<MapItem> nodeVector = new Vector<MapItem>();
            NodeItem n;
            ListIterator<NodeItem> nli = nodes.listIterator();
            //nodeVector.add(new MapItem(-1, map()));
            while(nli.hasNext()) {
                n = nli.next();
                if(n.visibility()) {
                    nodeVector.add(n);
                }
            }
            return nodeVector;
        }
        public void saveNodes() {
            /* Save the node list to corresponding map data file */

            /*try {
                XMLEncoder e = new XMLEncoder(new BufferedOutputStream(
                    new FileOutputStream("/tmp/dat_"+this.map+".xml")));
                e.writeObject(this.nodes); e.close();
            } catch (Exception er) {
                System.out.println(er);
            }*/
            try {
                FileOutputStream out = new FileOutputStream(dataDir + "/map_"+map+".dat");
                ObjectOutputStream s = new ObjectOutputStream(out);
                s.writeObject(String.valueOf(nodeId));
                s.writeObject(String.valueOf(edgeId));
                s.writeObject(nodes);
                s.flush();
                out.close();
            } catch (Exception e) {
                Errors.debug(e);
            }
        }
        @SuppressWarnings("unchecked") // need this to avoid readObject type warning which can't be fixed
        public void loadNodes() {
            /* Load the node list from corresponding map data file */
            try {
                FileInputStream in = new FileInputStream(dataDir + "/map_"+map+".dat");
                ObjectInputStream s = new ObjectInputStream(in);
                nodeId= Integer.parseInt((String)s.readObject());
                edgeId= Integer.parseInt((String)s.readObject());
                nodes = (LinkedList<NodeItem>)s.readObject();
            } catch (Exception e) {
                Errors.debug(e);
            }
        }
    }

}
```

## A.10 MapItem.java

```java
/*
 * project01: Description
 * Ian Melnick (dazed)
 * March 14, 2006, 4:04 PM
 * Course, Prof
 *
 * MapItem.java
 */

package project01;
import java.io.Serializable;

/**
 * Describes an item to be placed on a map.
 *
 * author Ian Melnick
*/
public class MapItem implements Serializable {
    /** This node's unique identification number. */
    protected int id;

    /** Describes this node, i.e., room number. */
    protected String description;

    /** The name of the map that this item is a member of. */
    protected String map;

    /** Creates a new instance of MapItem */
    public MapItem()              { this.id = −1;              }
    public MapItem(int id)        { this.id = id;              }
    public MapItem(int id, String map) { this(id); this.map = map; }
    public MapItem(int id, String map, String d) { this(id,map); this.description = d; }

    public int id()               { return this.id;            }
    public void setId(int id)     { this.id = id;              }

    /** Set: This node's string representation. */
    public void setDescription(String d){ this.description = d; }

    /** Get: This node's description. */
    public String description()  { return this.description; }

    public void setMap(String m) { this.map = m;       }
    public String map()          { return this.map;    }

    /** Get: This node's string representation. */
    public String toString()     { return /*map+": "+*/description; }
}
```

## A.11 NodeEditor.java

```java
/*
 * NodeEditor.java
 *
 * Created on December 11, 2005, 8:22 PM
 */

// http://java.sun.com/docs/books/tutorial/uiswing/events/windowlistener.html
// http://java.sun.com/j2se/1.4.2/docs/api/java/awt/event/WindowAdapter.html

package project01;

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;


/**
 * Dialog for changing node (and edge) properties.
 *
 * author Ian Melnick
*/
public class NodeEditor {
    JFrame window;
    JTabbedPane tabs;
    JPanel bottom, npf, ecf, ecp, fpf, fcp;

    JTextField description, distance;
    JCheckBox visible, delete, deledge, linksFloors;
```

```java
JComboBox connections, floorLinks;
int dnodeId;
JButton save, addFloorLink, delFloorLink;

NodeItem node;
MapComponent imc;
LinkedList<NodeItem> cl;

WindowMethods wm;
ButtonMethods bm;
ItemMethods im;

/**
 * Creates a new instance of NodeEditor
 */
public NodeEditor(NodeItem node, MapComponent imc) {
    this.node = node;
    this.imc = imc;
    this.dnodeId = 0;
    this.cl = imc.nm.connectedNodes(node);

    wm = new WindowMethods();
    bm = new ButtonMethods();
    im = new ItemMethods();

    window = new JFrame("Modify Node " + this.node.id());
    window.setLayout(new BorderLayout());
    window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);


    tabs = new JTabbedPane();
    bottom = new JPanel();
    npf = new JPanel();
    ecf = new JPanel();
    ecp = new JPanel();
    fpf = new JPanel();
    fcp = new JPanel();


    bottom.setLayout(new FlowLayout());
        save = new JButton("Update");
        bottom.add(save);

    npf.setLayout(new GridLayout(6,2));
        npf.add(new JLabel("ID"));
        npf.add(new JLabel(String.valueOf(node.id())));

        npf.add(new JLabel("Location"));
        npf.add(new JLabel(node.location().toString()));

        description = new JTextField(node.description());
        npf.add(new JLabel("Description"));
        npf.add(description);

        visible = new JCheckBox("Visible",node.visibility());
        delete = new JCheckBox("Delete", false);
        npf.add(visible);
        npf.add(delete);

    ecp.setLayout(new GridLayout(2,3)); // Edge properties panel, shown only when edge selected via ecf
        ecp.setVisible(false);
        //ecp.add(new JLabel("Orientation N/A"));

        distance = new JTextField(); // set in ButtonMethods
        ecp.add(new JLabel("Distance (Paces)"));
        ecp.add(distance);

        deledge = new JCheckBox("Delete", false);
        ecp.add(deledge);

    ecf.setLayout(new BorderLayout());
        if(cl.size() > 0) {
            Vector<String> nids = new Vector<String>();
            nids.add("Select connected node to modify edge");
            ListIterator<NodeItem> li = cl.listIterator();
            NodeItem t;
            while(li.hasNext()) {
                t = li.next();
                nids.add(String.valueOf(t.id()));
            }
            connections = new JComboBox(nids);
            ecf.add(connections, BorderLayout.NORTH);
        }

        ecf.add(ecp/*, BorderLayout.SOUTH*/);

    // container for floorLink properties
    fpf.setLayout(new BorderLayout());
    fcp.setLayout(new FlowLayout());
        addFloorLink = new JButton("Add New Link");
        delFloorLink = new JButton("Remove Selection");
        initFloorLinkPane();

    tabs.addTab("Node", null, npf, "Change Node Properties");
    tabs.addTab("Edges", null, ecf, "Change Edge Properties");
```

```
        tabs.addTab("Floors", null, fpf, "Change Floor-Link Properties");
        window.add(tabs);
        window.add(bottom, BorderLayout.SOUTH);

        window.addWindowListener(wm);
        save.addActionListener(bm);
        addFloorLink.addActionListener(bm);
        delFloorLink.addActionListener(bm);
        if(node.isFloorLink()) floorLinks.addActionListener(bm);
        if(cl.size() > 0)     connections.addActionListener(bm);

        window.pack();
        window.setVisible(true);
    }


    private void initFloorLinkPane() {
        fpf.removeAll();
        //fpf.add(new JLabel("Get down on the floors, whores!"));
        /*
         * Checkbox: This node connects to other floors
         * If checkbox is yes, display:
         *   List of connected floors
         *   Remove button
         *   Add button
         */
        linksFloors = new JCheckBox("Links floors",node.isFloorLink());
        linksFloors.addItemListener(im);
        fpf.add(linksFloors, BorderLayout.NORTH);
        fpf.add(fcp);

        // floorLink editor pane (shown only when selected via fpf)
        fcp.removeAll();
        floorLinks = new JComboBox(node.connectedFloors().toArray());
        fcp.add(addFloorLink);
        if(node.isFloorLink()) {
            fcp.add(floorLinks);
            fcp.add(delFloorLink);
        }
        fcp.setVisible(node.isFloorLink());
    }



    private class WindowMethods extends WindowAdapter {
        public void windowClosed(WindowEvent e) {
            node.select();
            imc.repaint();
        }
    }

    private class ItemMethods implements ItemListener {
        public void itemStateChanged(ItemEvent e) {

            if(e.getSource() == linksFloors) {
                switch(e.getStateChange()) {
                    case ItemEvent.SELECTED:
                        fcp.setVisible(true);
                        break;
                    case ItemEvent.DESELECTED:
                        fcp.setVisible(false);
                        //TODO: turn this into a method
                        Set<String> floors = node.connectedFloors();
                        Iterator<String> j = floors.iterator();
                        String floor;
                        NodeItem connection;
                        while(j.hasNext()) {
                            floor = j.next();
                            connection = node.floorLink(floor);
                            //connection.delFloorLink(floor);
                            FloorLinker.deleteConnection(imc.map(), floor, connection.id());
                            node.delFloorLink(floor);
                        }
                        imc.nm.saveNodes();
                        initFloorLinkPane();
                        break;
                }
            }

        }
    }

    /* Listener methods for button and combobox actions */
    private class ButtonMethods implements ActionListener {
        public void actionPerformed(ActionEvent e) {

            if(e.getSource() == save) {
                // set node properties as modified in dialog
                if(delete.isSelected())
                    imc.nm.delNode(node);
                else {
                    if(visible.isSelected()!=node.visibility())
                        node.visible();
                    node.setDescription(description.getText());
                }

                // set edge properties
                saveEdgeProperties();

                // close window
                window.dispose();
            }

            if(e.getSource() == connections) {
                try {
                    if(dnodeId!=0)
                        saveEdgeProperties();
                    dnodeId = Integer.parseInt((String)connections.getSelectedItem());
                    distance.setText(((imc.nm.sharedEdge(node, imc.nm.getNodeById(dnodeId))).getDistance())+"");
                    ecp.setVisible(true);
                } catch (NumberFormatException ex) {
                    dnodeId = 0;
                    //System.err.println(ex);
                    // Remove properties window
                    ecp.setVisible(false);
                }
            }

            // Floor-link actions

            if(e.getSource() == addFloorLink) {
                String[] images = Utility.map_images(imc.map().split("_")[0]);

                // Create new FloorLinkers for floors above and below this one.
                int thisFloor = Character.valueOf((imc.map().split("_")[1]).charAt(0));
                int otherFloor;
                String otherMap;
                FloorLinker linker;
                for(int i=0; i<images.length; i++) {
                    otherFloor = Character.valueOf((images[i].split("_")[2]).charAt(0));
                    otherMap = images[i].split("map_")[1];
                    otherMap = otherMap.substring(0, otherMap.length()-4);
                    if((otherFloor==thisFloor+1 || otherFloor==thisFloor-1) && !node.hasFloorLink(otherMap)) {
                        // One FloorLinker at a time—otherwise Heap barfs
                        // Linker takes care of updating node data once user selects dest node
                        linker = new FloorLinker(imc.map(), otherMap, node, imc.getCoordinates());
                        break;
                    }
                }
                //imc.nm.saveNodes();
            }

            if(e.getSource() == delFloorLink) {
                //System.out.println("delFloorLink");
                // Delete link from node on each floor
                // NodeItem.delFloorLink( (String)linksFloors.getSelectedItem() );
                String floor = (String)floorLinks.getSelectedItem();
                Set<String> updFloors = node.connectedFloors();
                Iterator<String> i = updFloors.iterator();
                NodeItem next;
                while(i.hasNext()) {
                    next = node.floorLink(i.next());
                    FloorLinker.deleteConnection(imc.map(), floor, next.id());
                }
                node.delFloorLink(floor);
                imc.nm.saveNodes();
                initFloorLinkPane();
            }

        }

        //

        private void saveEdgeProperties() {
            NodeItem dnode = imc.nm.getNodeById(dnodeId);
            EdgeItem edge = imc.nm.sharedEdge(node, dnode);
            if(dnodeId > 0 && deledge.isSelected()) {
                node.delEdge(edge);
                dnode.delEdge(edge);
            }
            else if(edge!=null) {
                try {
                    edge.setDistance(Integer.parseInt(distance.getText()));
                } catch(NumberFormatException err) {
                    JOptionPane.showMessageDialog(null, "Invalid Distance", "Error", JOptionPane.ERROR_MESSAGE);
                }
            }
        }
    }

    }
```

## A.12 NodeItem.java

```
/*
 * NodeItem.java
 *
 * Created on October 29, 2005, 3:30 PM
 */

package project01;

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.Serializable;


/**
 * Describes a node on a map.
 *
 * author Ian Melnick
*/
public class NodeItem extends MapItem implements Serializable {
    /** Coordinates that this NodeItem is located at. */
    private Point location;

    /** Whether or not this node is currently selected (i.e., by the user). */
    private boolean selected;

    /** Whether or not this node is visible to the user (invisible if corner).*/
    private boolean visible;

    /** Dijkstra value: this node's distance from the starting point. */
    private int distanceToMe;

    /** Dijkstra value: node ID of node directly before this one on path. */
    private int sourceNode;

    /** Holds list of EdgeItems this node is connected to. */
    private Set<EdgeItem> edges;

    /**
     * Dijkstra value: edge that provides overall shortest route that this is
     * connected to.
     */
    private EdgeItem sourceEdge;

    /**
     * Stores nodes that this node is connected to by elevators/stairs.
     * String map name, NodeItem the node. Zero length if this node isn't
     * a floor link.
     */
    private Map<String,NodeItem> floorLink;


    /**
     * Create a new NodeItem.
     */
    public NodeItem(String map, Point p, int id) {
        this.map = map;
        this.location = p;
        this.visible = true;
        this.selected = false;
        this.id = id;
        this.description = "n"+this.id+" ("+this.location.getX()+", "+this.location.getY()+")";
        this.edges = new HashSet<EdgeItem>();
        this.floorLink = new HashMap<String,NodeItem>();
        this.resetDijkstraValues();
    }

    /** Get: This node's coordinates. */
    public Point location()        { return this.location;        }

    /** Set: This node's coordinates. */
    public void setLocation(Point p) { this.location = p;     }

    /** Get: Whether or not this node is selected (i.e., by user). */
    public boolean selected()    { return this.selected;        }

    /** Set: Toggle whether node is selected. */
    public void select()           { this.selected = !this.selected; }

    /** Get: Whether or not this node is visible to user. */
    public boolean visibility() { return this.visible;          }

    /** Set: Toggle visibility. */
    public void visible()          { this.visible = !this.visible; }

    /** Get: List of EdgeItems this node is connected to. */
    public Set<EdgeItem> edges() { return this.edges;      }

    /** Get: Dijkstra value: return distance from start node. */
    public int getDistToMe()    { return this.distanceToMe; }

    /** Set: Dijkstra value: set distance from start node to this. */
    public void setDistToMe(int dist) { this.distanceToMe = dist; }
    /**
     * Set: Dijkstra value: edge that we follow to get to this node along
     * the shortest path.
     */
    public void setSrcEdge(EdgeItem e) { this.sourceEdge = e; }

    /**
     * Get: Dijkstra value: edge that we follow to get to this node along
     * the shortest path.
     */
    public EdgeItem getSrcEdge() { return this.sourceEdge; }

    /**
     * Set: Dijkstra value: node ID that is right before us along the
     * shortest path.
     */
    public void setSrcNodeId(int id) { this.sourceNode = id; }

    /**
     * Get: Dijkstra value: node ID that is right before us along the
     * shortest path.
     */
    public int getSrcNodeId()   { return this.sourceNode;   }

    /** Resets the Dijkstra values to ready for a new shortest-path run. */
    public void resetDijkstraValues() {
        setDistToMe(−1);
        setSrcNodeId(−1);
        setSrcEdge(null);
    }

    /** Whether or not this node is connected to a specific EdgeItem. */
    public boolean hasEdge(EdgeItem ei) {
        if(edges.contains(ei)) return true;
        else                   return false;
    }

    /** Connect an EdgeItem to this node if node doesn't already have it. */
    public void addEdge(EdgeItem ei) {
        edges.add(ei);
    }

    /** Disconnect an EdgeItem from this node. */
    public void delEdge(EdgeItem ei) {
        edges.remove(ei);
    }

    /** Whether or not this node links to other floors */
    public boolean isFloorLink() {
        return !floorLink.isEmpty();
    }

    /** Whether or not this node has a link to specified floor */
    public boolean hasFloorLink(String mapName) {
        return floorLink.containsKey(mapName);
    }

    /** Add a floor/node to list of linked floors */
    public void addFloorLink(String mapName, NodeItem linkedNode) {
        if(!floorLink.containsKey(mapName))
            floorLink.put(mapName, linkedNode);
    }

    /** Remove a floor/node from list of linked floors */
    public void delFloorLink(String mapName) {
        floorLink.remove(mapName);
    }

    /** Get list of floors this node connects to */
    public Set<String> connectedFloors() {
        return floorLink.keySet();
    }

    /** Based on a floor, return the connected node */
    public NodeItem floorLink(String mapName) {
        return floorLink.get(mapName);
    }
}
```

## A.13 Router.java

```
/*
 * Router.java
 *
 * Created on January 8, 2006, 1:45 PM
 */
// <applet code='Router' width='200' height='50'></applet>

package project01;

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
//import com.bruceeckel.swing.*;
//import java.io.*;
import java.util.*;
import project01.DirectionsFrame;


/**
 * Sets up the environment and runs the Map Shortest Path Route program;
 * contains listener methods that work with the {link MapComponent}.
 *
 * author Ian Melnick
 */
public class Router extends Main {

    private JComboBox cStart, cEnd;
    //private JSlider sPanX, sPanY;
    private JPanel bottom, bottomL, bottomC, bottomR;
    private JLabel status;
    private MapComponent imc;
    private Vector<MapItem> nodes;
    private Container cp;
    private ButtonMethods bm;
    private MouseMethods mm;


    public Router(Args a) {
        this.a = a;
        setMap(a.map);
        setNodeList();
    }

    public void setMap(String mapName) {
        imc = new MapComponent(mapName);
        imc.initMap();
        imc.nm.loadNodes();
    }
    public String getMap() {
        return imc.map();
    }
    public MapComponent.MapManager getMapManager() {
        return this.imc.mm;
    }

    private void setNodeList() {
        nodes = new Vector<MapItem>(1);
        nodes.add(new MapItem(−1, imc.map(), "---- "+imc.map()+" ----"));
        nodes.addAll(imc.nm.nodeVector());
        // Open the imc's of all maps and append their nodes to this vector
        MapComponent j;
        for(String mapName : Utility.map_images(null)) {
            mapName = mapName.split("map_")[1];
            mapName = mapName.substring(0, mapName.length()−4);
            if(mapName.compareTo(imc.map())!=0) {
                nodes.add(new MapItem(−1, mapName, "---- "+mapName+" ----"));
                j = new MapComponent(mapName);
                j.nm.loadNodes();
                nodes.addAll(j.nm.nodeVector());
            }
        }
        j = null;
    }
    private void showDirections(Dijkstra d) {
        new DirectionsFrame(this, d);
    }


    /**
     * Listener methods for mouse actions
     */
    private class MouseMethods extends MouseInputAdapter implements MouseWheelListener {
        private Point mouseClickPoint, mouseReleasePoint, newNode;
        private boolean mouseDragged, scrollDirection; // scroll direction = false, use y, true use x
        public void mouseMoved(MouseEvent e) {
        }
        public void mouseClicked(MouseEvent e) {
            if((e.getButton()==1 || e.getButton()==3) && e.getClickCount()==2) {
                NodeItem selNode = imc.nm.getNodeAtCoords(e.getPoint());
                try {
                    if(selNode==null ||
                            (cStart.getSelectedItem().getClass()==NodeItem.class &&
                            cEnd.getSelectedItem().getClass()==NodeItem.class)) {
                        cStart.setSelectedItem(nodes.get(0));
                        cEnd.setSelectedItem(nodes.get(0));
                    }
                    else if(cStart.getSelectedItem().getClass()!=NodeItem.class)
                        cStart.setSelectedItem(selNode);
                    else if(cEnd.getSelectedItem().getClass()!=NodeItem.class)
                        cEnd.setSelectedItem(selNode);
                } catch (java.lang.NullPointerException error) {
                    Errors.debug("Not a valid point");
                }
            }
        }
        public void mousePressed(MouseEvent e) {
            this.mouseClickPoint = e.getPoint();
            if(e.getButton()==2)
                this.scrollDirection = !this.scrollDirection;
        }
        public void mouseDragged(MouseEvent e) {
            this.mouseDragged = true;
        }
        public void mouseReleased(MouseEvent e) {
            this.mouseReleasePoint = e.getPoint();
            if(this.mouseDragged && !this.mouseReleasePoint.equals(this.mouseClickPoint)) {
                if(e.getButton()==1 || e.getButton()==3) {
                    imc.mm.offsetByMouse(this.mouseClickPoint, this.mouseReleasePoint);
                }

                this.mouseDragged=false;
            }
        }
        public void mouseWheelMoved(MouseWheelEvent e) {
            Point newCoords = new Point();
            if(this.scrollDirection) {
                newCoords.setLocation( imc.getCoordinates().getX()-(e.getWheelRotation()*50), imc.getCoordinates().getY() );
                //sPanX.setValue((int)newCoords.getX());
            }
            else {
                newCoords.setLocation( imc.getCoordinates().getX(), imc.getCoordinates().getY()-(e.getWheelRotation()*50) );
                //sPanY.setValue((int)newCoords.getY());
            }
            imc.mm.setOrigin(newCoords);
        }
    }

    /**
     * Listener methods for button/combobox actions
     */
    private class ButtonMethods implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if(e.getSource() == cEnd &&
              (cStart.getSelectedItem().getClass()==NodeItem.class &&
               cEnd.getSelectedItem().getClass()==NodeItem.class) ) {

                NodeItem start = (NodeItem)cStart.getSelectedItem();
                NodeItem end = (NodeItem)cEnd.getSelectedItem();

                if(start.map().compareTo(imc.map())!=0) {
                    setMap(start.map());
                    update();
                }

                showDirections(new Dijkstra(imc.nm, imc.nm.getNodeById(start.id()), end));
            }
        }
    }

    /**
     * Listener methods for slider actions
     */
    /*
     * private class SliderMethods implements ChangeListener {
     *   public void stateChanged(ChangeEvent e) {
     *       JSlider s = (JSlider)e.getSource();
     *       Point newCoords = new Point();
     *       if(!s.getValueIsAdjusting()) {
     *           if(e.getSource()==sPanY)
     *               newCoords.setLocation( imc.getCoordinates().getX(), s.getValue() );
     *           else if(e.getSource()==sPanX)
     *               newCoords.setLocation( s.getValue(), imc.getCoordinates().getY() );
     *           imc.mm.setOrigin(newCoords);
     *       }
     *   }
     *}*/

    public String getAppletInfo() {
        return "(c) Ian Melnick, Union College 2005-2006";
    }
    public String[][] getParameterInfo() {
        String pinfo[][] = {
            {"m","String","Map file to load (specify location/floor only, i.e., map_scieneg_000.jpg is loaded with arg scieng_000)"}
            /*{ "w", "Integer", "Width of map window (px)"},
            { "h", "Integer", "Height of map window (px)"} */
        };
        return pinfo;
    }


    public void init() {
        cp      = getContentPane();
        mm      = new MouseMethods();
        bm      = new ButtonMethods();

        status = new JLabel();
        cStart = new JComboBox(nodes);
        cEnd  = new JComboBox(nodes);

        cStart.addActionListener(bm);
        cEnd.addActionListener(bm);

        bottom = new JPanel(new GridLayout(1,3));
        bottomL = new JPanel(new FlowLayout());
        bottomC = new JPanel(new FlowLayout());
        bottomR = new JPanel(new FlowLayout());

        bottomL.add(new JLabel("From"));
        bottomL.add(cStart);
        bottomL.add(new JLabel("to"));
        bottomL.add(cEnd);
        bottomR.add(status);
        bottom.add(bottomL);
        bottom.add(bottomR);

        cp.setLayout(new BorderLayout());
        cp.add(bottom, BorderLayout.SOUTH);

        update();
    }

    public void update() {
        status.setText("map: " + imc.map());

        imc.addMouseListener(mm);
        imc.addMouseMotionListener(mm);
        imc.addMouseWheelListener(mm);

        if(cp.getComponentCount()>=2)
            cp.remove(1);

        cp.add(imc);

        //contentPane.paintComponents(contentPane.getGraphics());
        cp.validate();
    }

}
```

## A.14 Utility.java

```
/*
 * Utility.java
 *
 * Created on January 5, 2006, 10:55 AM
 */

package project01;

import java.awt.Dimension;
import java.io.*;
import java.net.*;
import java.util.LinkedList;
import java.util.jar.*;


/**
 *
 * author Ian Melnick
*/
public class Utility {
    public static Args getArgs(String[] args, Main prog) {
        Args a = new Args();

        if(args.length>0 && (args[0].equals("-h") || args[0].equals("--help"))) {
            System.out.println("Usage: java -jar Project01.jar <Editor|Router> <map> <width> <height>");
            System.exit(0);
        }

        if(args.length>0 && args[0]!=null) {
            try {
                a.mode = a.mode.valueOf(args[0]);
            } catch (IllegalArgumentException err) {
                System.out.print("Can't do " + args[0] + ", but I do know these: ");
                for(Main.AppModes op : Main.AppModes.values()) {
```

```
                System.out.print(op + " ");
            }
            System.exit(1);
        }
    }
    else {
        a.mode = a.mode.valueOf("Router");
    }

    if(args.length>1 && args[1]!=null) {
        a.map = args[1];
    }
    else {
        try {
            a.map = prog.getParameter("m");
        } catch (java.lang.NullPointerException e) {
            a.map = "scieng_100";
        }
    }

    // When called as an applet, it will get width/height properties
    // the normal way; only init() will be called, so we don't have to
    // worry about main().
    if(args.length>3 && (args[2]!=null && args[3]!=null)) {
        a.resolution = new Dimension(Integer.parseInt(args[2]),
                                Integer.parseInt(args[3]));
    }
    else {
        a.resolution = new Dimension(900,700);
    }

    return a;
}

public static String[] map_images(String matching) {
    if(matching==null) matching = "";
    final String mapDescrip = matching;
    final String lookForDir = "images/";
    final String mapPrefix = "map_";

    // Open directory containing map images
    // If running from JAR, must extract names from JAR.
    // Otherwise, just scan directory containing images.
    String imgLoc = Utility.class.getResource(lookForDir).toString();
    if(imgLoc.startsWith("jar:")) {
        LinkedList<String> imgList = new LinkedList<String>();

        URI filePath = URI.create(imgLoc.split("!")[0].substring(4));
        JarInputStream jin = null;
        try {
            jin=new JarInputStream(new FileInputStream(new File(filePath)));
        } catch (Exception er) {
            Errors.debug(er);
        }

        JarEntry jar = null;
        String pathParts[] = null;
        try {
            while( (jar = jin.getNextJarEntry()) != null )
                if(jar.getName().contains(lookForDir + mapPrefix) && (pathParts = jar.getName().split("/"))!=null)
                    imgList.add(pathParts[pathParts.length-1]);
        } catch (Exception er) {
            Errors.debug(er);
        }

        return imgList.toArray(new String[0]);
    }
    else {
        File imageDir = new File(URI.create(imgLoc));
        if(!imageDir.isDirectory())
            System.err.println(imageDir + " is not a directory! Continuing...");

        // Only allow map images from same building as current map
        // FilenameFilter: http://javaalmanac.com/egs/java.io/GetFiles.html
        // imc.map().split("_")[0] == _building
        FilenameFilter filter = new FilenameFilter() {
            public boolean accept(File dir, String name) {
                return name.startsWith(mapPrefix + mapDescrip);
            }
        };
        return imageDir.list(filter);
    }

    //return null;
  }
}

class Args {
    public Main.AppModes mode;
    public Dimension resolution;
    public String map;
}
```