Project01

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.

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.

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.

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. (?, ?)

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. (?, ?)

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