The Brockport Navigator

Michael Adams
The College at Brockport, mikeynick44@gmail.com

Follow this and additional works at: https://digitalcommons.brockport.edu/honors

Part of the Software Engineering Commons

Repository Citation
https://digitalcommons.brockport.edu/honors/80
The Brockport Navigator

A Senior Honors Thesis

Presented in Partial Fulfillment of the Requirements
for graduation in the College Honors Program

By
Michael Adams
Computer Science Major

The College at Brockport
August 2011

Thesis Director: Dr. Vishal Anand, Associate Professor, Computer Science

Educational use of this paper is permitted for the purpose of providing future students a model example of an Honors senior thesis project.
<table>
<thead>
<tr>
<th>Content</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.   Introduction</td>
<td>1</td>
</tr>
<tr>
<td>II.  The Android Operating System</td>
<td>2</td>
</tr>
<tr>
<td>III. Google Earth</td>
<td>3</td>
</tr>
<tr>
<td>IV.  SQL</td>
<td>6</td>
</tr>
<tr>
<td>V.   Parsers</td>
<td>7</td>
</tr>
<tr>
<td>VI.  The Custom View Class</td>
<td>9</td>
</tr>
<tr>
<td>VII. The Map Display</td>
<td>10</td>
</tr>
<tr>
<td>VIII. The Navigator</td>
<td>11</td>
</tr>
<tr>
<td>IX.  The Proposed Routing Algorithm</td>
<td>12</td>
</tr>
<tr>
<td>X.   Dijkstra’s Algorithm for Routing</td>
<td>15</td>
</tr>
<tr>
<td>XI.  Universality and Portability</td>
<td>16</td>
</tr>
<tr>
<td>XII. Enhancements and Future Work</td>
<td>17</td>
</tr>
<tr>
<td>XIII. Conclusion</td>
<td>18</td>
</tr>
</tbody>
</table>
Abstract

This paper discusses the creation processes used to design and develop the Brockport Navigator, an Android application developed to help acclimatize incoming students to the College at Brockport campus. The paper describes the basic concepts of the Android operating system and what is involved in developing Android applications, the process used to map out the campus via Google Earth, the SQLite database used to store all of the GPS (Global Positioning System) information, the CustomView class developed to display the application, and the Map Display and Navigator features of the application. Next, the paper discusses the routing algorithm that was written to handle route calculation, and the Dijkstra’s Algorithm, which later replaced the initial routing algorithm. Lastly, the paper explores the future enhancements and features that may be added.

I. Introduction

As a student of the Honors Program at The College at Brockport, I was tasked with a Senior Thesis and given the option to either create a product or perform a research study. Knowing I wanted to do something involving smartphone application development, I chose a combination of research and creation. Location-based applications in particular are very interesting, and the possibilities available given the user’s current latitude and longitude are almost endless. The College at Brockport admits over 1,500 new students a year [1], and the only tool available for navigating the campus was the online map, which was poorly scaled and not available when on the go. The idea for Brockport NAV spawned from personal experience in becoming acclimated to the campus.

This application is essentially for new students who are not yet familiarized with the campus. It has three main features: Map Display, Navigation, and Directory. The Map Display simply displays an image of the campus with a red dot for your current location. The Navigator, which is the core of the application, allows users to select one of the campus’s buildings, calculates a route to that building from the current position, and then displays that route on the screen. The Directory simply opens the browser to the online directory of Brockport’s faculty. The application is currently only available for Android powered devices, but will eventually be ported to other mobile operating systems as well (such as iOS and
Windows Phone 7). This will enable new students to learn the paths to their classes, and actually be navigated to the building in real time.

This paper focuses on the entire process of creating the Brockport Navigator application and is organized as follows: Section II explains the basics behind the Android operating system, programming applications for it, and the Directory feature. Section III explains the use of Google Earth in mapping out the campus for use in the application. Section IV describes the use of SQL to store and access geographical information for use in navigation. Section V discusses the parser I wrote to use the information from Google Earth and create a database from it. Section VI overviews development of a CustomView class to display an application with dynamic imagery. Section VII quickly explains the Map Display feature and how it works. Section VIII describes the Navigator feature and its workings. Section IX discusses the algorithm I wrote to calculate a path to the selected building. Section X overviews Dijkstra’s Single Source Shortest Path algorithm, which I replaced my own with. Section XI discusses the universality of my application. Section XII looks at planned and potential additions and enhancements. Section XIII concludes the paper.

II. The Android Operating System

Android is an open-source mobile operating system developed by Google. The operating system itself is a form of Linux, written in C/C++ [2], but application development is done in the Java programming language. Screen layouts and a few other components are coded in XML [3]. The standard for developing Android applications is with the Eclipse IDE using the Android plug-in. Although I did have an Android device to test the application with, the Android plug-in allows you to emulate an android device and run your application on it. With the emulator, however, multi-touch is not supported, and hence not able to be tested. Location services also have to be simulated through a telnet service instead of truly being read from the device. Actually having an Android device is a strong recommendation for anyone considering development. Alongside the standard Java library, you are provided Android’s programming library as well. The main foundation of an Android application consists of activities and intents. Activities are the actual functions of the application (such as displaying the map, or navigating to a building), and intents are used as indications to begin an activity. For example, in this application, the main starting screen is an activity, and clicking the “Navigator” button launches an intent to start the Navigator activity. The developer can either specify the activity directly (by giving the intent the name of an activity class they created) or indirectly, by specifying the type of action and the parameter to carry it out. Android is then able to use the application
tag (found in the parameter) and the type of action to carry out the activity. This latter method is how the Directory feature of the application is launched. The intent can be told what type of action to carry out, such as a viewing action (Intent.ACTION_VIEW), and then told what exactly it is supposed to view (“http://www.brockport.edu/publications-phone/fsdir.html”). Android will look at the application tag (“http” in this case), and knowing that this needs to be viewed, it will look through its list of applications that are capable of viewing something with that tag. If there is only one, it will use that application. If there are many capable applications, it will ask the user which one to use [4]. Developers may specify which actions and tags their application can use so that their application can be presented as an option. In this manner, Android does not require the developer to know which applications are installed on the end-user’s phone. Thus, if someone opening the Directory feature of this application has multiple web browser applications, Android will ask them which browser they would like to open that page with. To quickly mention the Directory feature, it simply opens the web browser to Brockport’s online directory, where the user may look up a faculty member and from there start composing an email to them, dial their office phone, or find their office number and building. This was included to allow users quick and mobile access to contacting their professors, since they may not be at a computer when they need the information.

III. Google Earth

Considering the campus map supplied by the college’s website was very poorly scaled and not entirely accurate to positioning (see Fig 3.1, 3.2 and 3.3) [5], it was not a valid option for that map to be displayed in the application. The college supplied image does show the correct general layout, but it is not absolutely true to the geography. Considering direct latitude and longitudes were going to be used, an accurate representation of the campus was needed. Since the geographic information was going to be pulled from Google Earth, it was best that the image of the map come from the same source. Thus, screenshots of the satellite imagery from Google Earth were used. There was a minor issue here in that the clearest image of the campus was from 2005, before the campus townhomes had been constructed. To remedy this, the 2009 image was taken (which had the townhomes), the area comprising the townhomes was selected, and Photoshop was used to insert that over the 2005 image. With some edge blurring and color matching, this gave a clear image of the campus that included the townhomes. With an appropriate image to display, the next step was mapping out the campus. In order to convert latitude and longitude into pixel coordinates to be displayed on the screen, the geographical boundaries of the campus needed to be known. Google Earth was used to find the latitude and longitude of the four corners of the
area used for the image, which consequently gave the width and height of the campus in degrees. The actual conversion will be discussed in Section VI.
In order to calculate a path, a large, connected graph of the campus’s walkways would be needed. Google Earth allows users to put place marks and paths down on the map, which can later be exported to a file [6]. This is what was used to accomplish the mapping. Place marks were put at each turn, crosswalk, and along long sidewalks. Paths were then used to connect all the place marks according to the walkways on campus (see Fig 3.4). Google allows you to name the place marks and paths, so each place mark was given a number, and each path two numbers – the two place marks it connected. The final result was a fully connected graph consisting of about 900 nodes (place marks). Each place mark held all the geographical information regarding that spot (latitude, longitude, altitude, among many others), as well as the name that had been given to it. Google Earth allows users to export all of their place marks and paths into a .kml file, which can be opened as a text file to display all of the information. An example of the data found in a .kml file can be seen in Fig 5.1 in Section V. The file contains a significant amount of information about Google Earth itself before getting into your place marks and paths, so that was simply discarded. Then, the place marks were separated into one .txt file, and the paths into a separate .txt file. These .txt files were the input to the parsers (discussed in Section V). The next step was to lay out a database to hold all of this information for use in the application.
IV. SQL

An object needed to be defined to represent these nodes, and there needed to be an efficient way to access them in the application. There were five pieces of information for each node – ID (Integer; primary key; the number given to each node in Google Earth), Name (Text; name of the building that node corresponds to, or the ID if it is not a building), Expansions (Text; the list of adjacent nodes as a string of IDs), Latitude (Numeric) and Longitude (Numeric). In the application, a *LocationNode* object was defined, consisting of the same five attributes, as well as methods for retrieving adjacent nodes, computing distance to another node, and checking equality. Android has library support for SQLite databases, so this was chosen as the way to hold the information of all the nodes. At the time I had very little knowledge in the field of databases and SQL, but alongside SQL’s online reference material [7] Google provides many thorough tutorials and examples on their Android Developer website [8]. I did their full practice tutorial on the Notepad applications to learn how to use SQLite in Android. Once that had been completed, I had a pretty good understanding of how to create a database, how to
access information from it, and how to modify information in it (though this was not something that needed to be done in the application). When creating an Android application, you may include any files you need (such as an SQLite database) in the “Assets” folder, which you can then pull files from inside the application. Once the database is in the phone, it stays there until the application is deleted. Determining variables during runtime and using them to construct and run raw queries was the main method of accessing data from the SQLite database.

For example, if the user chose to be routed to the building “Harrison”, an SQL query would be created that ends in “WHERE name='Harrison'”. The result is a Cursor object, which you may traverse row by row, each of which you can call column values from. Data from these columns would be returned back to the calling functions. Creation of the database and verification of query returns was done using a program called “SQLite Database Browser” [9], which is open-source software. It provided a GUI for verifying the contents of the database (See Fig 4.1), which was useful for finding and correcting mistakes made during the mapping process.

V. Parsers

Now that a database had been laid out, the information gathered from Google Earth somehow needed to be inserted into the SQLite database. There was now a list of all the edges in the graph, and a list of all the nodes in the graph, with their name, latitude and longitude. My parser read in two .txt files, the first being the
list of edges, and the latter the list of nodes. I created a String array to hold Strings for adjacent nodes, then parsed the list of edges .txt file by skipping any line that did not start with “Name” (the line in each edge that named the two nodes it connected). When it did start with name, a String variable would be made for each name, then added to the other’s String of adjacent nodes. When it finished with edges, it read through each node in the node list, pulling only the name (number that had been given to it), latitude and longitude. See Fig 5.1 for example input. For each node it created a statement reading “INSERT INTO locations VALUES ([name], [name], [adjacent nodes], [latitude], [longitude]);”. SQLite Database Browser was used to run all these insert statements and create a database file. Since every node’s id and name were the same value, the buildings had to be marked out by changing the name value in their row to the building name. This was done manually in the same program by changing the value of the appropriate node’s name to its building name. The end result was an SQLite database file contain the id, name, adjacent list, latitude and longitude of all ~900 of the nodes on campus. This could then be loaded onto the phone to be used in the application.

Figure 5.1 KML file text for 1 node followed by one edge
VI. The Custom View Class

Android has two main options for displaying two dimensional images: ImageView or creating a CustomView class [10]. I had started with ImageView only to discover that ImageView was for static images that did not need to update dynamically. Since the user’s location needed to be continually updated, it was necessary to define a CustomView class. There were many things that needed to be taken into account for this, such as displaying the image of the campus, handling touch events for panning and zooming, converting the latitude and longitude of each node to a pixel coordinate, drawing the user’s location, and drawing the route to the desired building. There are three main objects that are used when drawing to the screen: Canvas, Bitmap, and Drawable [11]. The Drawable is the actual picture file from the resources folder (for example, I had a ‘campus.jpg’), the Bitmap is created from the Drawable, and the Canvas is what the Bitmap is drawn onto. There were two types of touch events that needed to be supported: single touch swipes for panning, and multi touch pinches for zooming. I learned how to do this through a tutorial in “Hello Android! 3e” [12], which uses a matrix to affect how the canvas is drawn using parameters such as scale and offsets. For single touch swipes, you track how far on the screen the user has moved their finger, and then use this to translate the matrix and drag the image in that direction. For multi touch pinching, you see how far apart the user’s fingers were when they put their fingers down, and compare it to how far apart they are when the user lifts them up. You use this to scale the matrix and hence scale the image. Now that the image of the campus can be panned and zoomed, a way to draw the nodes in their respective place on the image is needed. The latitude and longitude fields in the LocationNode were of the following form: North latitude and East longitude are positive, South latitude and West longitude were negative, and the values were expressed as doubles. These numbers needed to be converted into integer pixel coordinates so that they could be drawn to the screen. The pixel dimensions of the image of the campus were known, and the dimensions of a scaled image (i.e. after the user zooms) was obtained by multiplying the width/height by the scale value in the matrix. The point of origin of the campus was known (Northwest corner, since Android’s (0, 0) point is in the top left corner of the screen) and the width and height of the campus in degrees were known (from the four corners of the campus found in Google Earth). The user’s current latitude and longitude (found via the LocationManager object included in the Android library [13]) could then be translated to pixel coordinates via the following formula (example using longitude):

$$xPixel = scaledImageWidth * \left( \frac{|currentLong - longOrigin|}{longWidth} \right) + xOffset$$
where scaledImageWidth is the pixel width of the current image size (i.e. image zoomed), currentLong is the user’s current longitude value, longOrigin is the longitude of the Northwest corner, longWidth is the width of the campus in degrees, and xOffset is the current image offset value (i.e. image panned). To update the user’s location, it is set to change every two seconds or 1.5 meters, whichever comes first. When that happens, the LocationManager updates the current latitude and longitude, and forces the canvas to redraw with this new information. The user is simply represented as a red dot on the map. For drawing the path, each node in the current path has a line drawn from itself to the next node in the path. There is also a dot drawn at the start and end nodes. All parts of the path are drawn in blue. When a new building is selected, the path is cleared and then recalculated. Originally, the path was going to start shortening itself as the user walked along it, but this would have caused issues if the user did not follow the path directly. Thus, the path is simply calculated once and then displayed, showing the user’s progress along it, but not removing any lines or dots from the path.

VII. The Map Display

The purpose of the Map Display is to simply allow the user to view the campus at their leisure, and to indicate where on the campus they are. This way, the user does not have to select a building; they may simply walk around and see where they are going on the campus (see Fig 7.1, 7.2). This is also where the the idea for the Center Screen function came about. When the user pans the image around, their dot moves as well. I wanted to allow the user to be able to center the image on their dot, so I wrote a method that changed the offset values in the canvas’s matrix to have the user’s current location at the center of the screen. This is done automatically when Map Display and Navigator start, and there is also a button for it in the context menu.
VIII. The Navigator

The Navigator is the core of the application, allowing the user to select one of the buildings on campus and be shown a path from where they are to that building. Only the parts leading up to and after the actual route calculation will be discussed here, as that will be covered in the following two sections. For a path to be calculated, three things need to be known: where to start, where to stop, and how the nodes are connected. When the user clicks the “Navigator” button, they are brought to a screen with a list of all the buildings. When they click one of the buildings, the LocationSelect activity launches an intent to start the Navigator activity, attaching the name of the selected building to the intent. When the Navigator activity starts, it retrieves the name from the intent, and then queries the database for the row with a name value matching that name. A LocationNode object is then made for that row in the database. To find the starting node, the latitude and longitude of the user’s current location is taken and incremented/decremented a very small amount to create a small square area. All
the nodes that are in this square area are found and then the distance between the user’s current location and each of these nodes is computed (see Fig 8.1).

![Figure 8.1 Square area from which to choose start node. Red starred node is chosen because it is the closest to the user’s location (origin point)](image)

Whichever node returns the smallest distance value is set as the user’s start node. Having the start node and end node, and each node knowing its neighbors, the route calculation is now possible. The Navigator needs to construct a vector of nodes to represent the path from start to end. It does this differently for each of the algorithms, so that will be discussed in their respective sections. Once the route has been calculated, the Navigator sets the path vector in the CustomView object to the one it just calculated. Just as in the Map Display, every two seconds or 1.5 meters the current latitude and longitude are updated and the user’s location redrawn. The path does not change once it is calculated; it simply tracks the user’s movements and displays their red dot moving along the path.

IX. The Proposed Routing Algorithm

The algorithm I developed was a recursive, best-first algorithm. The concept was for each node to look at all its adjacent nodes and move to the one that was closest to the goal. The initial algorithm worked as follows: the function would be called with the start node, the start node would look at all of its adjacent nodes and calculate their distance from the goal node; whichever node had the shortest distance to the goal node was selected as the next node to visit. Once the goal node was visited, the algorithm started to unwind, adding each node to the path as it did. Once it was back to the start node, it would return the path which would then be set in the CustomView. The first problem with this method is that,
occasionally, of the node’s neighboring nodes, the one closest to the goal was the one it had just come from (see Fig 9.1).

Figure 9.1 With G as the goal node, node 1 expands to 2 (closest to G) and node 2 expands to 1. This will repeat until the stack overflows and the application crashes.

Thus, it would go back and forth between the two until there was a stack overflow. To solve this, another parameter was added to the method – the node that was just visited. Now it could look at the potential expansion, and if it was the node that was just visited, it would skip that and get the next one. The problem with this is that certain buildings only connected to one other node, so if the algorithm happened to expand to a building, it could not go to any other nodes since it skips the one it came from (see Fig 9.2).
Figure 9.2 Algorithm expands to Holmes building and cannot backtrack because the only node to expand to is the one it just came from.

Figure 9.3 Greedy approach finds questionable path
This was fixed by only allowing it to skip the node it came from if it had other nodes to go to. I now had a working algorithm that computed the path fairly quickly, but the greedy design led to some interesting paths. Since it chooses the neighbor that is closest to the goal node at that particular point, it does not always choose the one that may lead to a better path. It was computing paths that worked, but not paths that most students take to quickly get where they need to go (see Fig 9.3). The worst case complexity of this algorithm is $O(n \log n)$. This is because each time the function is called, the node builds a TreeMap of the adjacent nodes, using their distance to the goal node as a key. Reheaping a TreeMap has a complexity of $O(\log n)$. This is only done a small number of times (ranging from about 1 to 6 or 7 – it’s the number of adjacent nodes), so this is considered $O(\log n)$. In the worst case, this would have to be called for every node in the graph, or $n$ times, giving a complexity of $O(n \log n)$.

X. Dijkstra’s Algorithm for Routing

Considering the difficulties brought up by my initial routing algorithm, it seemed a better choice to implement a more well-known routing algorithm. Dijkstra’s Single Source Shortest Path algorithm was chosen partly because it was suggested by a professor, and partly because I had recent experience with it from a course. There were many key differences between my algorithm and Dijkstra’s. Firstly, my algorithm is recursive, whereas Dijkstra’s is iterative. To find the number of nodes in order to create the arrays and priority queue, the database of nodes is queried for the maximum id number. One small modification made is that when the goal node is found, the algorithm is stopped to reduce the wait time for the user. Typical execution does not finish until the stack of nodes is empty and all of the true shortest paths are found. There is a small part of recursion used to build the path after the algorithm is run. Since the id of the goal node is known, and a path to it has been computed, a method called ‘pathBuilder’ can be called on the goal node, which adds itself to the path and then recursively calls the function with its parent, returning when the parent is null. Implementing Dijkstra’s algorithm gave much more intelligent paths with no crashing, giving the user optimal routes to their destination with reliability. On the downside, it was also much slower than my algorithm when computing longer paths. For short paths, there was not much computation needed, so response times were comparable to my algorithm. With long paths, however, where my algorithm took a greedy, straight-for-the-goal approach, Dijkstra’s takes a more breadth-first approach. This caused long path computations to take up to 3-5 seconds, making the app feel somewhat slow. Considering the benefit of having a proper and efficient path without the app crashing, Dijkstra’s algorithm was chosen over my own.
The source code still contains my algorithm, it is simply commented out. The complexity of Dijkstra’s algorithm using the priority queue implementation is $O(e + n\log n)$, where $e$ is the number of edges and $n$ is the number of vertices. This explains why it executes more slowly than my algorithm.

XI. Universality and Portability

The way the code was written, this app can be used for any campus, park, or other area desired. The only things that would be needed to use this for another area would be: an image of the area (satellite or otherwise), the geographical boundaries of the area (four corners), and a Google Earth mapping of the area. The Google Earth mapping would be the most intensive of the three, and the amount of work would depend on the size of the area. The Brockport campus is 435 acres [14], and that took roughly 10-12 hours to map out. Porting this application to other mobile devices (iPhone, Windows Phone 7, etc.) should not be incredibly challenging, since most of it revolves around the location sensor and
SQLite databases. The most challenging aspect would be learning how to do custom views for those two devices. Since iOS is much more widely used than Windows Phone 7, that would take first priority in porting the application.

XII. Enhancements and Future Work

While the basic application is finished, there are still some enhancements I would like to make to it before putting it up on the market. The first and easiest is a recalculate button. Users should have the ability to calculate a new path from their current location, in case they stray or simply want a new path. This is a standard feature on most navigation units, so it should definitely be included. This will be simple as there is a “firstTime” boolean that can just be set back to true, and the path will be cleared and recalculated. After that, the navigator will be made to route to the closest entrance of a building, rather than the main entrance. As it currently stands, the navigator will route to the entrance of the building that is most commonly used by students, but not always the closest entrance to where you currently are. The potential issue with adding this is that some of the alternative entrances lead to ground floors, whereas students should typically be on the first floor. This may confuse new students since they are not familiar with which floor classes are normally held on. A message could be displayed, saying to go up a floor if a ground floor entrance is chosen, or it could be kept as going to the main entrance. I will have to implement it and test it to see how well it works out. Another feature that would be useful, but more complicated to implement, would be a class scheduler. The user could enter their class schedule into the app, and then a reminder could alarm them 10-15 minutes before their class starts and route them to the building that the class is in. Many new students end up late to class because they cannot find the building that their class is located in, so this would help them leave at an early enough time and show them how to get there. A couple other computer science students had heard about my project, and had additional thoughts for features. Their focus was on tying in web content and social networking features. The easier idea they had was for users to be able to find open computer labs and dining halls. A student would, for example, choose ‘Computer Lab’, and it would check the campus website for which computer labs are currently open (by using current day of the week, current time stamp, etc.), choose the nearest one (by comparing to current latitude and longitude), and navigate the user to that lab. This would not be too challenging, as it simply involves parsing a web page and comparing it to values retrieved from the device. More complicated would be the social networking ideas they brought up. Ideas included displaying the location of friends on the map image in addition to your own, having emoticons for the user’s current mood, being able to text or call someone that you can see near you on the map, reminders for special events,
being able to mark favorite locations around campus, among a few others. This would be a significant amount of coding and testing, so if this does end up being implemented, it likely would not be for a while yet.

XIII. Conclusion

Brockport NAV was a pleasure to work on, and will hopefully be a useful and time-saving tool for incoming freshmen. It solves a real world problem in an interesting and mobile way. While Dijkstra’s algorithm was not quite as quick as the initially implemented algorithm, it provides efficient paths in a reasonable amount of time. With the addition of an indication that the route is being calculated, it should not seem slow or sluggish to the end-user. Brockport NAV will be up on the Android App Market by the beginning of the fall semester (August 29th, 2011). Developing this application provided a good overall introduction into programming for mobile devices, both in dealing with their limitations, and tapping into the potential of their location-based services. I am excited to see what may be in store for this application, both in terms of using it on other campuses, and expanding and combining it with existing and new social networking applications.

References


