CHAPTER 14

Pathfinding

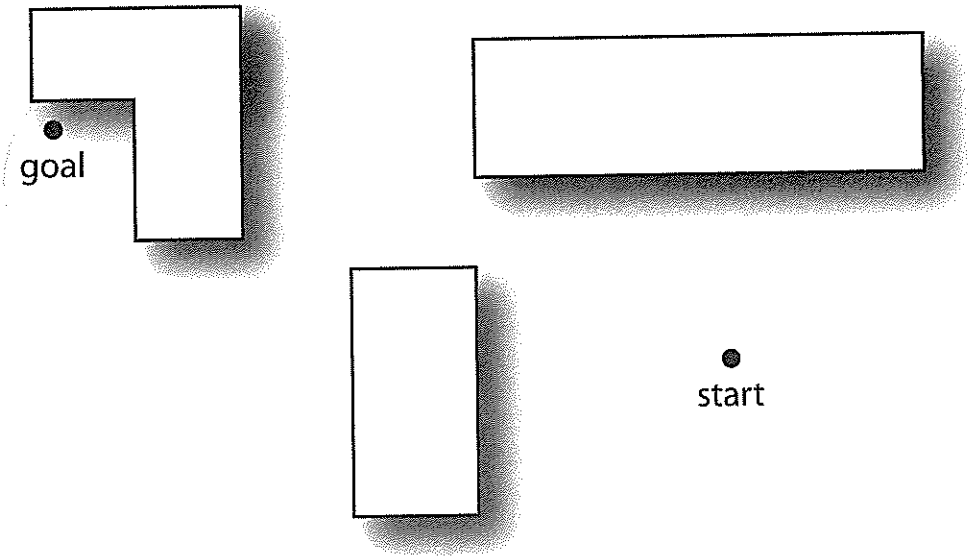
TOPICS IN THIS CHAPTER

- Nodes
- Search Algorithms
- Navigation Meshes
- Adding obstacles to your map

Pathfinding is an interesting problem that has been largely conquered in the field of computing. The goal is to find the shortest path from one point to another, regardless of the obstacles that stand between these two points (see Figure 14-1). There are discrete concepts of pathfinding, including search algorithms, nodes, and navigational meshes. Let's explore these concepts.

Pathfinding

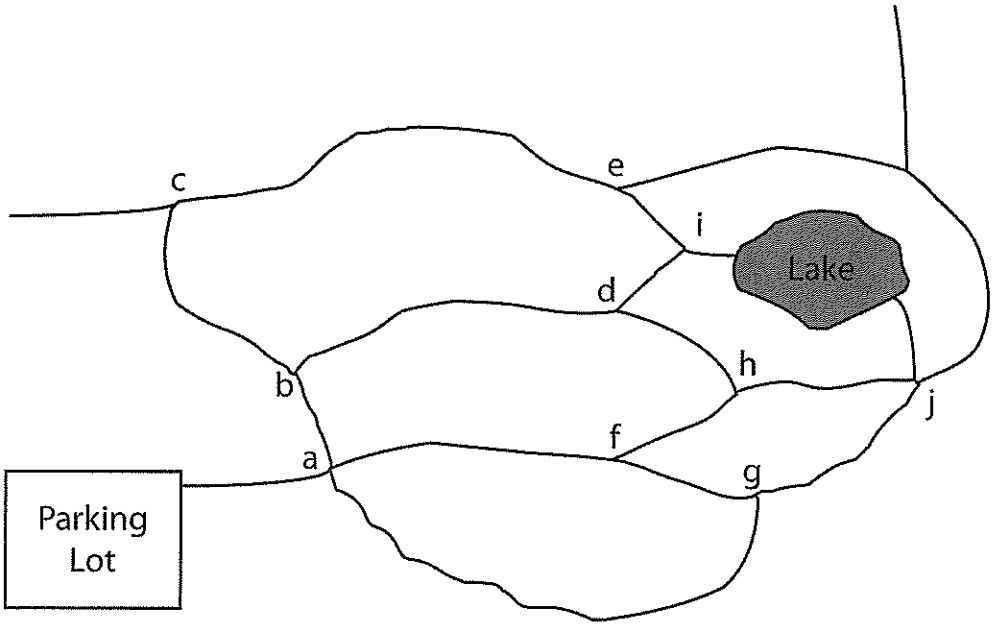
Humans have the built-in ability to quickly find the shortest path from one point to another. Take a look at Figure 14-1. Can you trace the shortest path to the goal? It probably took you no thought at all. Surprisingly, humans have a hard time describing *how* they did it—that is, the thought process behind how they arrived at the solution. It is especially difficult for a programmer to describe to a computer (in the form of a program) how to find the shortest path.



Before we can instruct a computer how to perform pathfinding, we need to develop an abstract representation of the environment that a computer program can understand. One component of this model consists of a map representation, as we explored in the previous chapter. The other component of pathfinding is a node—or rather, a set of nodes.

Nodes

Imagine a dense, impenetrable forest with lots of paths snaking throughout it. These paths occasionally link-up with other paths (see Figure 14-2). Wherever two or more paths link together, we have the option of going down one of two or more paths. We must make a decision which path to take. The places where these paths cross are called nodes, and they are a central concept of pathfinding.



Darkplace Forest

A node is a variation of a coordinate. In a two dimensional space, each node has an x and y coordinate. So how do nodes differ from coordinates? Each node is connected to at least one other node (see Figure 14-3). With the node model, there is no option to break away from one path onto another path. You can only change direction when you reach a new node.

In Figure 14-2, if you want to plot a route from the parking lot to the lake, there are a multitude of possible routes. The shortest path is A to B to D to I. Our goal is to have the computer calculate this type of path for our robot.

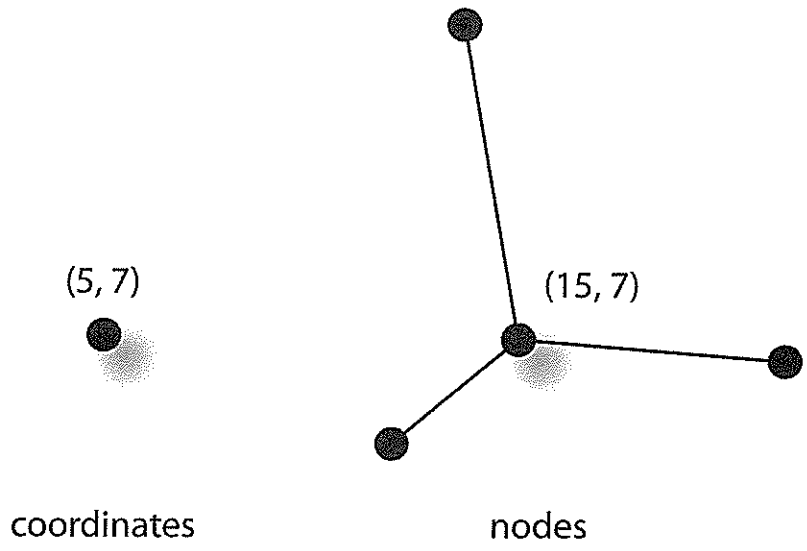


Figure 14-3: Comparing coordinates with nodes

Navigation Mesh

The nodes in the forest example above were all predetermined for you. You did not design the nodes because the forest paths were already long established before you arrived at the scene. The crisscrossing roads in a city are also predetermined for you, with every intersection representing a node

But what about a relatively open environment where there are no predetermined paths? Most indoor robotics problems consist of a relatively wide open area with some sporadic obstacles in the environment (see Figure 14-1). Where are the nodes and adjoining paths? As you can see, no predefined nodes exist in this type of environment as they do with roads or forest paths.

This problem leads us into the interesting problem of navigation meshes. To automate pathfinding, we ideally want to feed the robot a map and then let it figure out how to get from one point to another. So without any nodes, it needs to generate a set of nodes on its own. This complete set of nodes, which covers the entire area of a map, is called a *navigation mesh.*

Navigation meshes are used extensively in video games—normally first person shooters like Call of Duty and real-time strategy games like Starcraft—to help characters move from one location to another. Usually the navigation meshes are pre-calculated by the designers and included with the map data for a particular level.

Similarly, with a typical indoor environment, we must generate a set of nodes to overlay the map. There are many different ways you can generate a navigation mesh. A few common ones include:

- regular grid pattern
- randomly generated
- heuristically generated

There are downsides to some types of navigation meshes. For example, the grid mesh produces paths that are all perpendicular to one another without diagonal paths, which results in, well, very robotic movement by your robot. Figure 14-4a shows the shortest path a robot might take

you and I would take if we weren't locked onto a grid mesh, such as that shown in Figure 14-4b. By generating an optimal navigation mesh for a particular environment, we can find a shorter overall path.

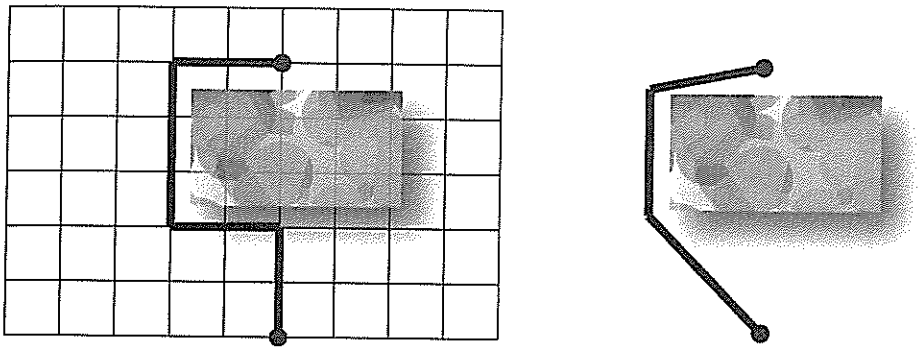


Figure 14-4: Plotting a path on a grid

As you can see above, there is a lot of empty white space within navigation meshes. Technically a more complete navigation mesh would include lots of interconnected nodes all across the map. For perfect coverage, you could include one node spaced at the atomic level. However, this is not practical as it would use too much memory and produce very long searches.

The best we can do is populate the space with a diverse selection of nodes, which is merely an approximation of the number of places the robot could navigate to on the map. As you can see, there are limitations with the node-search paradigm.

One other factor of consideration when generating a navigation mesh is node pruning. There are often illegal paths between nodes that are invalid because they would cause the robot to collide with walls or other objects (see Figure 14-5a). If you are plotting a path for a boat, you probably

ter channels if the topography below the surface is known. Therefore, you might want the node generation routine to create a navigation mesh that takes these factors into account, rather than relying on a grid or random node set.

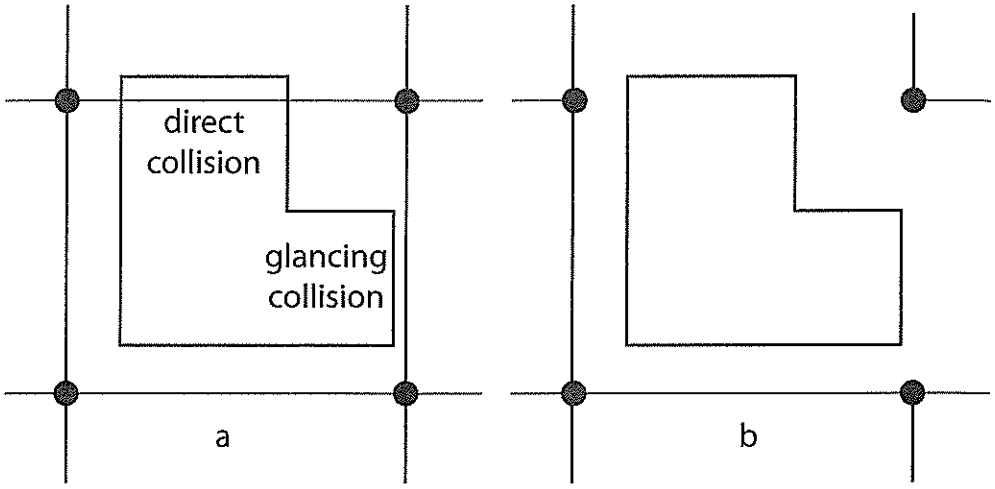


Figure 14-5: Pruning connections from a set of nodes

By checking the connections between nodes for collisions with the map geometry, we can tell which connections should be pruned. Then we merely remove these connections between the nodes (see Figure 14-5b). The leJOS node generation algorithm uses a pruning algorithm to keep a wide berth between your robot and other objects.

Search Algorithms

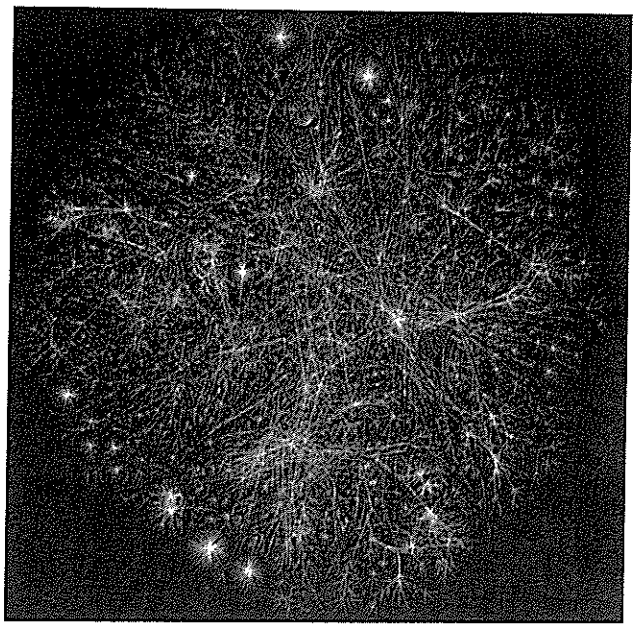
The final concept we will cover in the pathfinding topic is node searching. A node search uses an algorithm to search though a set of nodes to find the optimal path from a starting node to the goal node.

Modern search algorithms started with Dutch mathematician Edsger Dijkstra in 1956. He wanted an algorithm to

came up with—today called Djikstra's algorithm—is proven to find the shortest path between any two nodes.

Because Djikstra's algorithm finds the most optimal path between nodes, you might think that there is no point in doing any further research into node searches. After all, nothing can beat the path it generates. However, in the 1960s, his algorithms saw practical use with early computer networks, specifically in routers. Variations of his algorithm are used by network routing protocols to make the Internet faster by finding the shortest route for data packets between nodes (see Figure 14-6). Obviously engineers want fast data transmission, so they set to work on improving the speed of these search algorithms.

Since then, a whole family of search algorithms have been produced, the most famous being A* (pronounced A star). A* can improve search speeds by a factor of 20 times or more over Djikstra's original algorithm.



Node search algorithms are mainly used in video games for pathfinding, consumer GPS devices to plot routes, web-based mapping applications such as Google maps and of course, mobile robots. The Mars Pathfinder mission, launched in 1996, used D* pathfinding. Later, a variation on this, called D* lite, superseded D* to become the most popular search algorithm for use in robotics.

The bottom line is that all of these search algorithms produce the same results. It is a matter of how fast they produce results, and how many resources they use in terms of memory and computing horsepower. This can be a factor for devices with relatively slow processors and limited memory, such as the NXT brick.

Let's examine a pathfinding example on your NXT brick. First we will need to alter the environment and map data from chapter 13.

Adding Obstacles to a Map

Pathfinding is not very interesting unless there are obstacles between the starting node and the goal node. Add one or more obstacles to the test area from chapter 13. To keep things simple, I added a single box in the middle of my test area (see Figure 14-7). Measure the dimensions of the box and the relative location of the box compared to two perpendicular walls. Now we will need to add this obstacle to our existing map data.

Let's return to SVG Edit from the previous chapter and modify the map we created.

1. Launch the SVG-Edit application at:

   http://code.google.com/p/svg-edit/

2. Click on the menu icon and select Open Image (see Figure 14-8). Do not select Import SVG as this will add

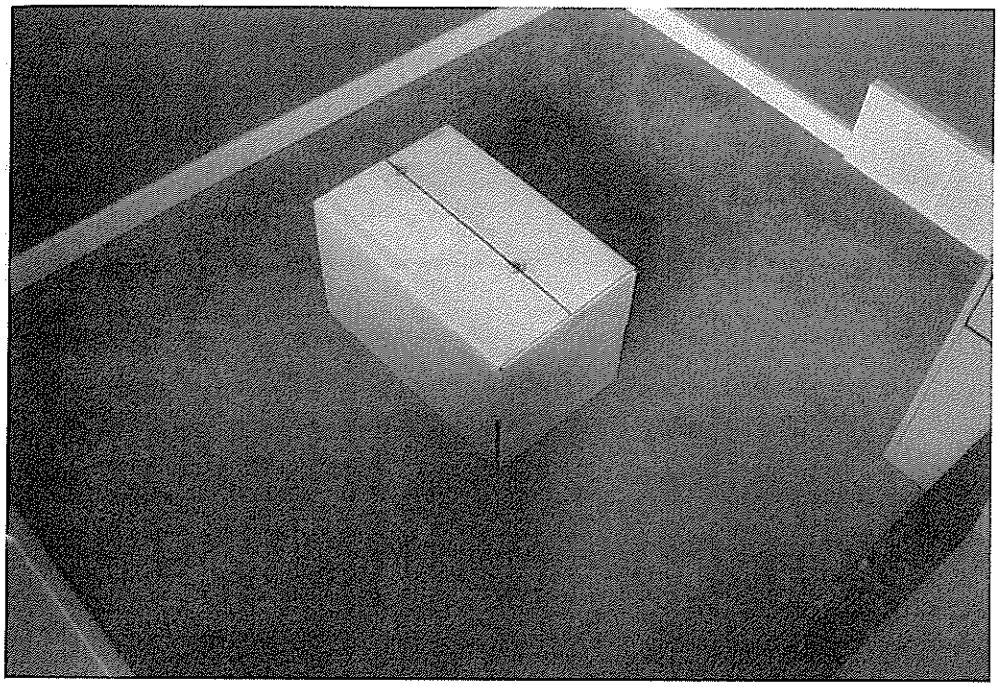
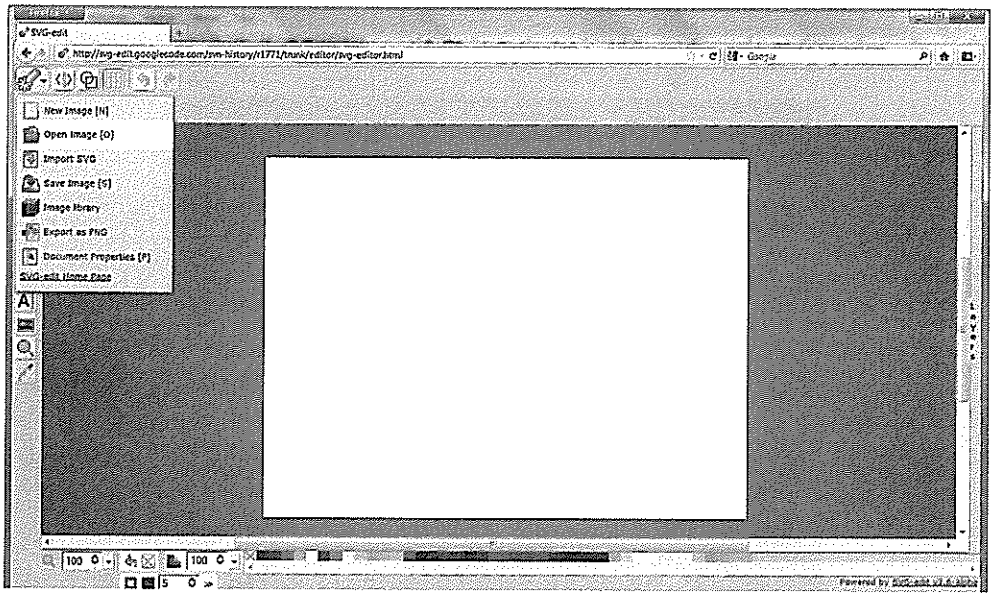


Figure 14-7: Adding an obstacle in the test area



3. Using your measurements, add the new obstacle(s) to your map (see Figure 14-9).

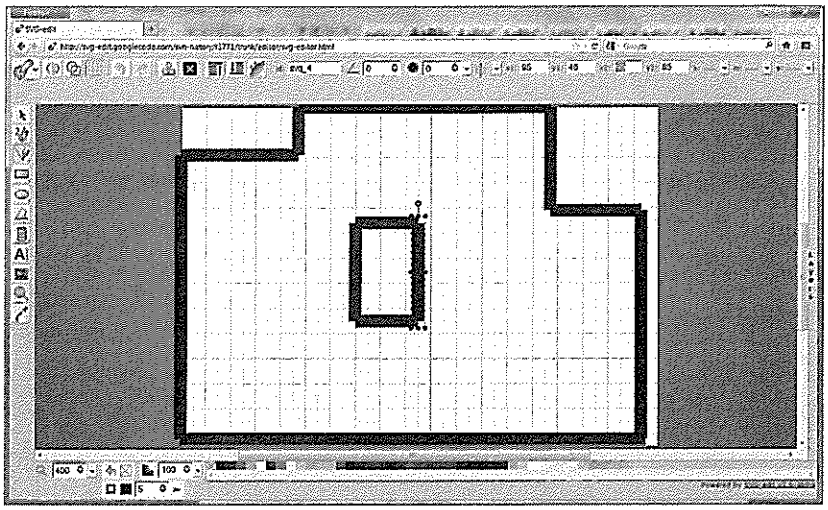
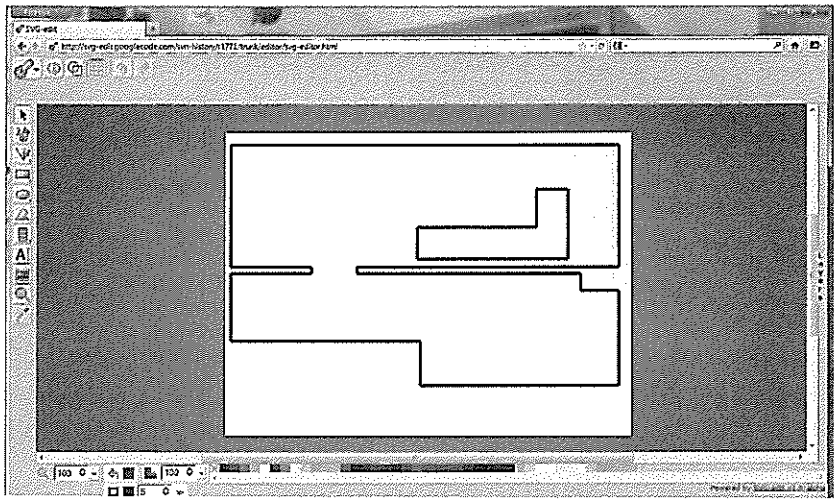


Figure 14-9: Adding an obstacle to the map

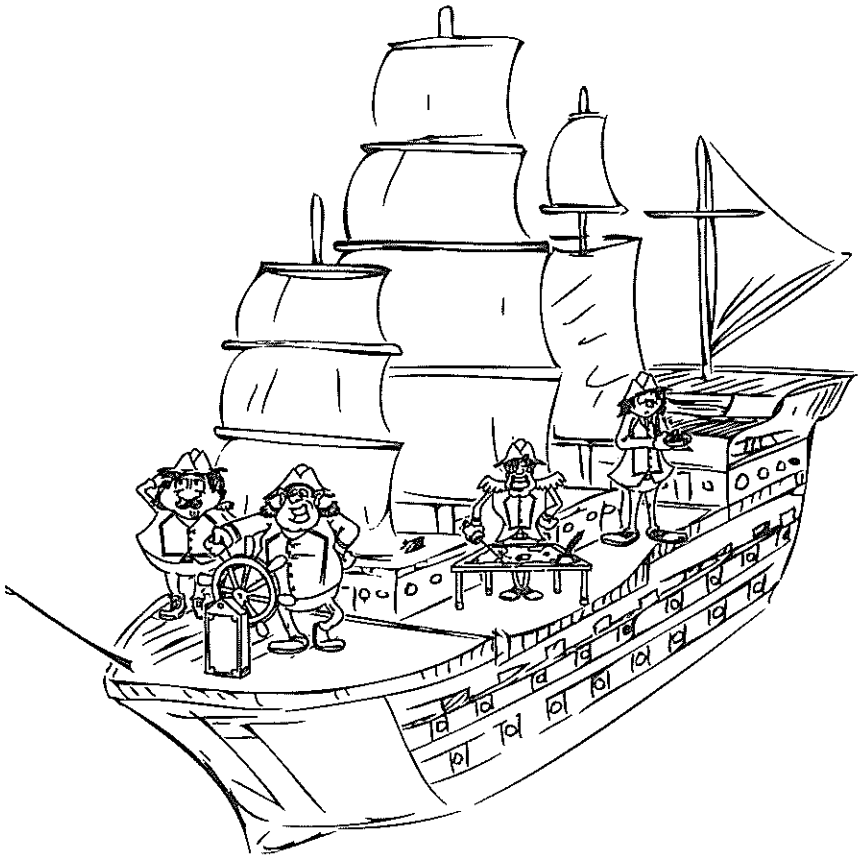
4. Alternately, you could create a larger floor plan to give the pathfinding algorithms a real challenge (see Figure 14-10).



When completed, save your map file to a folder or your desktop. We will use this map later in this chapter.

Programming Pathfinding

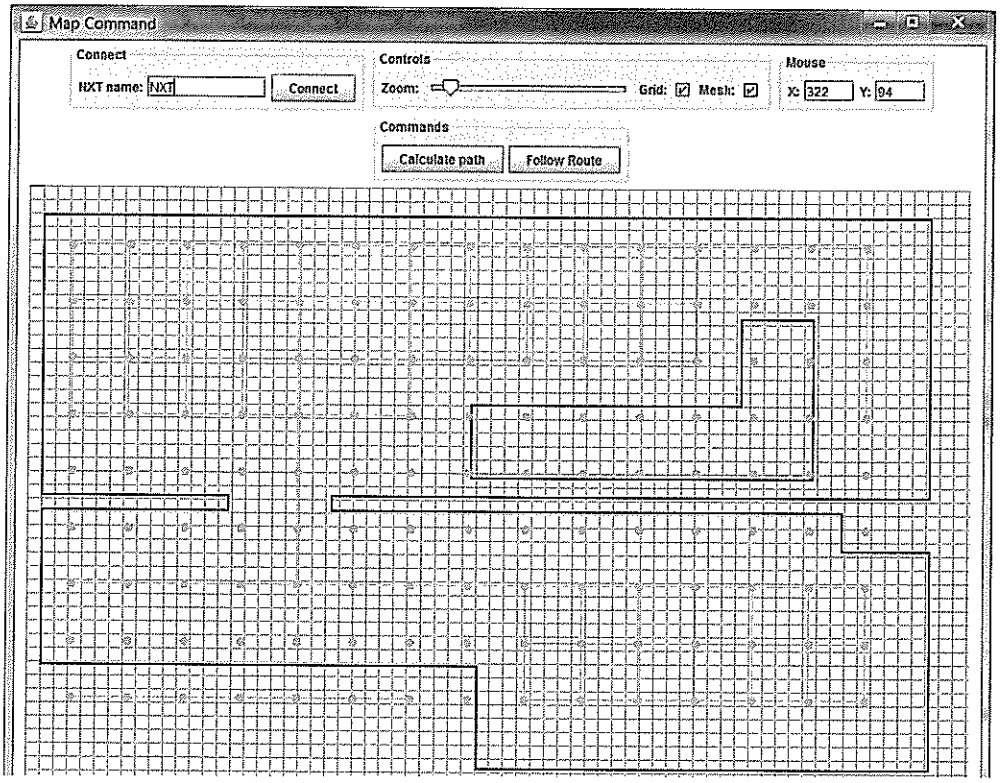
The application in this section is a culmination of all the ideas we explored in the navigation chapters. It uses not only a Pilot, Navigator, PoseProvider (hidden) and map data, but also a PathPlanner. In effect, all the key players are onboard the same boat (see Figure 14-11).



The following application connects to the NXT and instructs the robot to follow a calculated path. It works best with a large map, such as a whole floor with doorways rather than a single room. The initial pose and the target are set using popup menus.

There is no new code for this project. We simply use Map Command on the PC side and the NXTSlave code from the previous chapter. However, we will select different Map Command options this time.

1. Run nxjmapcommand.bat and load in your SVG file from the previous section. Resize the zoom bar until the map fits your screen (see Figure 14-12). Make sure both grid and mesh are selected.



2. Set down your NXT robot at a location within your test area and note the coordinates. Now run the NXTSlave program.
3. On Map Command, enter the name of your NXT brick and click Connect.
4. Now move the pointer to the starting location of your robot on the map. When you see the correct coordinates in the upper right corner, click the grid and select Set Pose. A new node will appear at the location connected to the nearest nodes (see Figure 14-13).

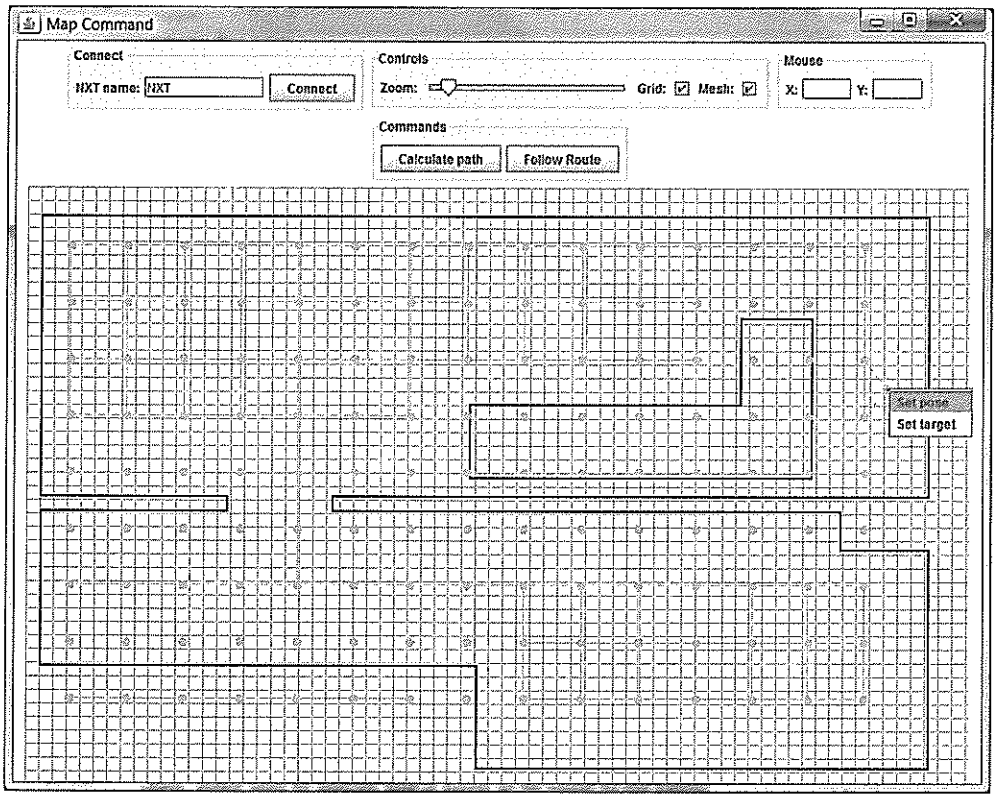


Figure 14-13: Selecting the starting node

5. Now move the pointer to a destination on the map,

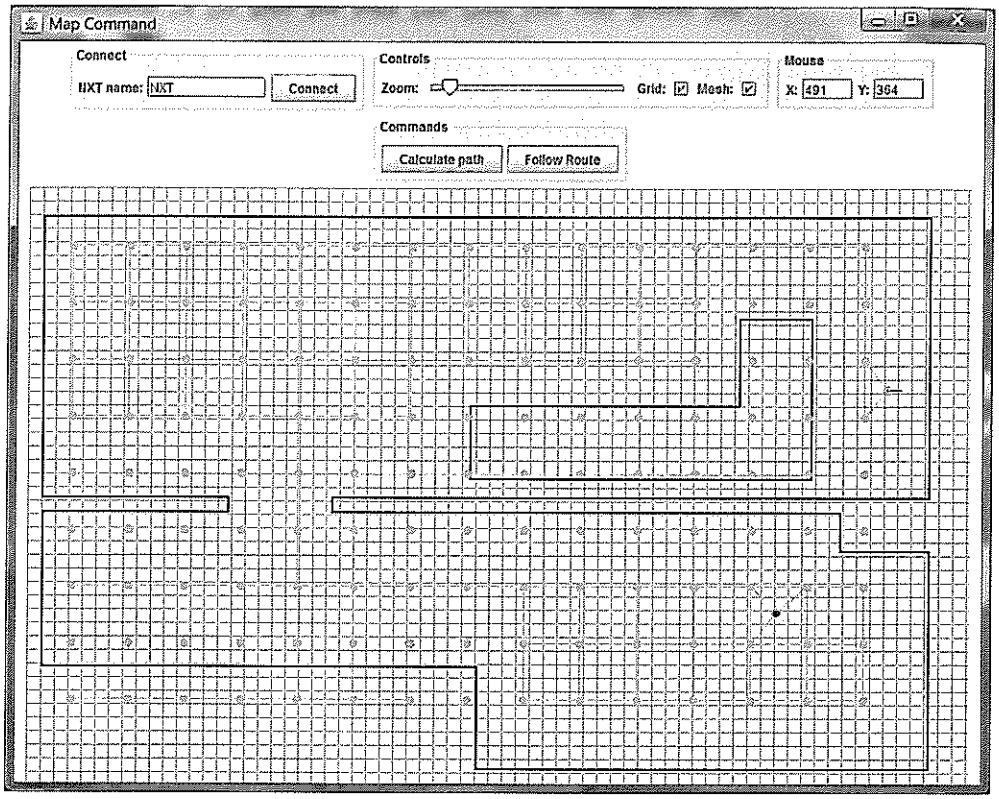


Figure 14-14: Selecting the destination

6. Now click Calculate Path. The algorithm will quickly find the shortest path between the robot and destination nodes. A line will show the path the robot will take (see Figure 14-15).
7. Finally, click Follow Route and your robot will begin moving to the target destination.

That's a simple demonstration of the pathfinding algorithm. You can also access the pathfinding classes directly and program your own custom applications. You can examine these classes in the lejos.robotics.pathfinding package.

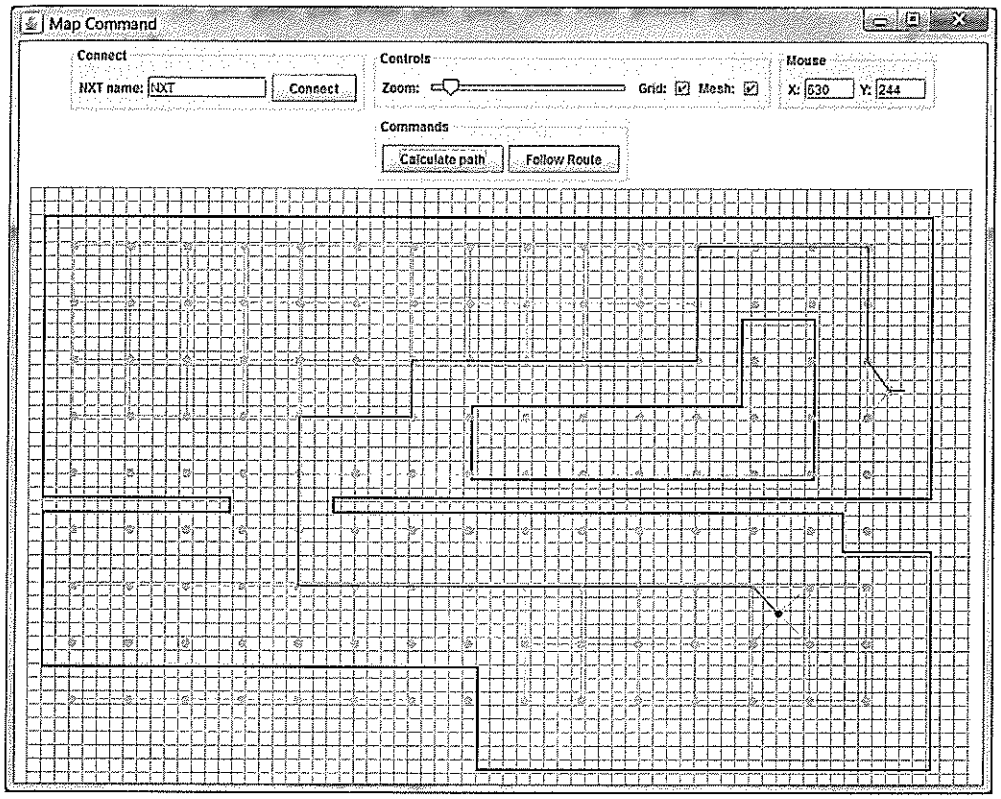


Figure 14-15: Generating a path