Path planning: how Google maps and game bots actually work

Out: Friday, May 20
Due: Friday, May 27, noon

Overview

In this assignment, you will implement a priority queue and then use that queue to implement Dijkstra's shortest path algorithm. For the priority queue, we have given you the shell of the class for a binary heap, with a number of the methods left blank, as well as a set of test cases to let you verify that it works. For path planning, we have provided you classes to represent graphs, nodes, and edges. We have not given you test cases for path planning, but we have given you a GUI for testing out your path finder.

Implementing a binary heap

You should begin by unzipping the directory and open the PathPlanner solution in Visual Studio. Then go to the BinaryHeap.cs file. This contains the fields, constructor, and Add method for a binary heap, but the methods for ExtractMin, DecreasePriority (called DecreaseKey in the lecture slides), MoveUp (the while loop from HeapInsert in the lectures), and Move Down (called Heapify in the lectures) are unimplemented. You should begin by filling in these methods.

Things to be aware of:

- The Node class (for nodes of the graph) has a field in it called **QueuePosition**. Since DecreasePriority takes an actual node as an argument (rather than its position), but other heap procedures like MoveUp and MoveDown need positions as arguments, your DecreasePriority method will need to get that information from the QueuePosition field.
- That means you need to make sure when writing your methods that you set the QueuePosition field any time you add an object to the queue or move it around. The Swap method that we've included takes care of that for you, so you won't need to worry about it when just swapping elements.

Fields we've already added for you in the BinaryHeap class:

- count
  The number of elements in the heap. Be sure to keep this up to date, i.e. decrement it when you remove something.

- data
  An array of nodes. This is where you store the nodes in the heap.
- priorities
  An array of doubles (i.e. numbers). This is where you store their priorities. So that a node stored at position 10 in data, has its priority at position 10 in priorities.

We've also given you some useful methods:

- Parent(i), LeftChild(i), RightChild(i)
  Compute the positions of the neighbors of the node at position i.
- Swap(i, j)
  Swaps the elements at positions i and j. It will rearrange the elements of both the data and priorities arrays, and will also update the QueuePriority fields of the nodes at those positions.
- TestHeapValidity()
  Checks the data and priorities arrays to see if the heap actually satisfies the heap property. Useful for debugging.
- Add(node, priority)
  Adds the node to the heap with the specified priority. You won't use it for this part of the assignment, but you will need it for the path finding part. You may also want to look at its code to see an example of how to manipulate the heap.

What you should implement

Your job is to implement the following methods:

- MoveDown(int position)
  Repeatedly moves the node at the specified position downward in the heap if it is larger than either of its children, until the Heap Property has been satisfied. This is simply the Heapify procedure from the lectures and the book with a more descriptive name. MoveDown and MoveUp both work by swapping elements, so they should use the Swap procedure we've already written for you.
- MoveUp(int position)
  The reverse of MoveDown; repeatedly moves the node upward when it's smaller than its parent, until the Heap Property is reestablished. This is the while loop of HeapInsert procedure from the lectures. We've broken it out as a separate procedure, because you'll need to call it from DecreasePriority.
- DecreasePriority(node, newPriority)
  Changes the node's priority and moves it appropriately in the heap. This was called DecreaseKey in the lectures and book. Again, it is renamed here because DecreasePriority is a more accurate name.
- ExtractMin()
  Removes the node with the lowest priority from the heap and returns it.

Again, a set of tests have been provided to help you debug you heap implementation.

Implementing Dijkstra's shortest path algorithm

Next you should implement the path finder. Open the UndirectedGraph.cs file and find the FindPath method. Fill it in with Dijkstra's algorithm.

We have **not** provided automated tests for your path finder. For this one, you're on your own. Write some unit tests. As usual, we won't grade them, but we recommend you write them.

In addition, we have provided you with a full graphical user interface for testing your path finder, as well as a sample graph to test it on. Simply run the application (i.e. press the play button or F5), and it will pop up a window with a picture of the graph and text boxes where you can enter then names of nodes, then press Find Path to run your path finder. It will show the resulting path in green. If your code throws an exception, it will display the exception in a dialog box. You will then need to set a breakpoint in your FindPath algorithm to debug it. Good luck.

Notes:

- We have already included **NodeCost** and **Predecessor** fields in the Node class, since they are needed for the algorithm.
- The FindPath procedure should return the actual path as a list of nodes (i.e. an object of type List<Node>). You can get the path by starting at the end and following the predecessor links. Note, however, that you should return the path with the start as the firsts element of the path, and the end as the last element. You may find the Reverse() method of the list class to be useful for this.

Turning it in

As usual, choose Clean Solution from the Build menu of Visual Studio, then exit Visual Studio, make a zip file of your project directory, and upload the zip file to Canvas.