Week 5 Exercises

Physics 91SI Spring 2011 Handout 16

Alex Ji and Zahan Malkani

Environment Variables

If you didn't do the exercise on environment variables in Week 3, you should go back and do that. (Handout 12, which you can find on the website). In particular, you should probably set a variable linking to our class afs space to save yourself a lot of typing/typos.

Breadth-First Search

We're going to implement simple breadth-first search to solve a maze. The below discussion is a simplification of the discussion in this wikipedia article: `http://en.wikipedia.org/wiki/Breadth-first_search`, so you can read through that on your own if you're interested. **For today's class, as long as you understand how to implement the BFS algorithm you don't need to understand everything.** Ask Alex or Zahan to help you out if you're having trouble getting to that point.

A "graph" is a model used in many computer science and scientific applications. Graphs are composed of "nodes", often represented as circles, and "edges" between nodes, represented as lines between those circles. There are many types of graphs: connected and unconnected, directed or undirected edges, with or without edge lengths, with or without cycles, with or without multiple edges between two nodes, and more. We're only looking at the simplest graphs today: connected, undirected graphs without edge lengths.

One thing you often want to do when you use graphs is, given a starting node, search through the graph for a particular ending node. There are two common ways to do this: breadth-first search and depth-first search. We're going to focus on a very simple breadth-first search today, which uses queues. (Depth-first search is very similar, except you use stacks; see the wikipeda article.)

First, some terminology: we say that two nodes are "neighbors" if they are connected by an edge.

The basic idea of breadth-first search is as follows: Start at a node. Look through all of its neighbors, and see if any of those are the end node. If not, look through all of the neighbors of the neighbors and see if any of those are the end node. Continue until you've found the end node or you've looked through the entire graph. I like to imagine a growing mold that begins growing at the start node and expands through all edges to other nodes. Breadth-first search is nice because it finds the shortest path between the start and end node (measured by number of edges).

The basic BFS algorithm is implemented as follows:

- Enqueue the start node. (`put`)
- While the queue is not empty:
  - Dequeue a node. (`get`)
  - If it is the end node, you're done; return a result.
  - Else, enqueue every neighboring node that you haven't yet seen. (Note you'll have to keep track of nodes you've already seen. A set would be a good data structure.)
- If you went through the whole queue, that means you went through the whole graph and didn't find the end node. Return something like "not found".

We're going to model a maze as a graph. The maze will be broken up into a grid of squares, and each square is a node, uniquely specified by its coordinates `row, col`. Two nodes are connected by an edge if you can walk from one square to the other. We are given a starting square (the start node) and we want to search through the maze to see how to get to the exit square (the end node). BFS is perfect for this.

Implement BFS

Your first task is to implement the breadth-first search (BFS) algorithm. Clone the `exercise5` repository into your working directory. We have given you a file `maze.py` that has a `Maze` class. This class reads in a maze from an input formatted as specified in `maze_input_format.rdoc`.

Invoke Python's help utility with `pydoc maze.py` and look at the methods we give you. Check out `parse_input` specifically to read in a maze and get going! At this stage, there should be no need to actually look at code in `maze.py`, it would just cause unecessary confusion.

Using these methods, write a function `find_end_node` in `shortestpath.py` that takes in a `Maze` object, starts at the start node, searches through the maze to find the end node, and prints out the coordinates of the final node.

You should implement this in two stages. First, write some tests for the function. We've given you a sample test file in `test_maze.py` that tests the Maze class; add two or three new mazes that you can test your function on (this shouldn't take more than 10-15 minutes). Then, use the algorithm we gave you to implement a BFS that finds the end node.

Find Shortest Path

Now instead of just the final coordinate, we're going to actually keep track of the path itself. Write a function in `shortestpath.py` called `find_shortest_path` that returns a list of nodes describing the path between the start and end nodes. If there is no path, return an empty list.

Make sure to write some tests beforehand (i.e. check the final length of the list and make sure the path is a legal path). HINT: you can use a tuple like `(node, list_of_nodes_in_path)` in your queue.

Nodes with more Information

We've provided you a file `mazes/sample_maze.dat` that not only has a maze, but associates each square in the test maze with a string. There's a `parse_layout` method that will read this file and return a maze as well as a dictionary. Using this dictionary and your shortest path code, write a function `find_sentence` that takes in the shortest path list and prints out the sentence associated with the shortest path.

New Maze implementation

If you look at the implementation of the `get_neighbors` method in the `Maze` class, you'll see that we create the set of neighbors each time a node is encountered. This is actually efficient for solving a maze because we visit each node at most once, but you can imagine other algorithms that wouldn't want to do this because you could visit the same square/node multiple times. If this is the case, you'd want to do something more efficient. One way to do this is to represent the maze as a dictionary mapping nodes to sets of nodes. In the constructor for the `Maze` class you would have to construct this dictionary, but then you could quickly access the neighbors

In the `maze.py` file, write some PSEUDOCODE at the top that implements this dictionary-based maze.