Dijkstra's Algorithm for Shortest Path Optimization

Alan Su and Peter Holmes

MATH260: Summer II 2020

1 Grid Class (grid.py)

To begin to approach graphical path-finding problems, as is the application of Dijkstra's algorithm, we had to first construct a class that our algorithm could operate on. To do this, we create a node-based class (Grid), that used inputs of 'node_list' (a list of the names of all desired nodes), num (the number of desired nodes), and an optional input of 'gridsize', for use with true grids. The class would also store a nested dictionary object that contained the adjacency list for each node, and a list of distances to each of those adjacent nodes. While distance is specified as the numerical parameter of each edge, it really can be any measure of weight that the user needs. Possibilities include time, congestion level, risk, and much more. When a Grid object is printed, it prints the adjacency and distance lists for each node iteratively. Additional functions were included to allow for easier implementation of Dijkstra's algorithm.

Initially, the functionality of the Grid class was just to create a graph of nodes connected by edges, which can be defined by the edge() function. These graphs could be of any shape and have any number of connections. In order to better mirror real-life pathfinding in cities, we created the truegrid() function that creates Grid object that has a defined grid shape. Nodes are labeled with tuples that denote their x and y positions within the graph. Two additional functions, file_reader() and gridprinter(), were also written to complement these truegrid objects. file_reader() simplifies the process by which true grids can be created. It takes in a text file where horizontal distances between nodes are specified in odd-number lines and vertical distances between nodes are specified in even-number lines. The function reads the file, splitting the data into two separate lists, one that keeps track of the horizontal distances and another that keeps track of the vertical distances. This function is called in truegrid() to generate a true grid object from the file. gridprinter() can be used with a true grid object as the input to print a visual representation of that grid, starting with node (0, 0) in the top left corner, increasing in the second coordinate when moving to the right and increasing in the first coordinate when moving down. This function has spacing that provides for distance values up to three digits.

2 Dijkstra's Algorithm (dijkstra.py)

The main function is in the file is dijkstra(), which takes in a grid object, a start node and an optional input of nodes to avoid. dijkstra() serves as a main function that calls dijkstra_alg() to actually do the shortest path optimization. dijkstra_alg() does this recursively by maintaining a 'visited' dictionary, recursively visiting more and more nodes until the visited dictionary returns True for all nodes. 'pathlist' is initialized as a dictionary with all nodes as keys and values of None for all, it is updated with the last node visited in the shortest path to the key node before reaching it. 'distlist' is initialized as a dictionary with all nodes as keys and values of infinity (using inf function from math module) except for the start node, which has a value of zero; this dictionary then is updated with the distance value of the shortest path to each node from the start node. dijkstra_alg() begins

by visiting all nodes adjacent to the current node, finds the distance to those nodes and then uses its own distance to the start point to calculate the total distance of the adjacent node to the start point using this path. If it is shorter than the distance for the adjacent node in 'distlist', it will update the value in 'distlist' with the new shortest path distance and also update 'pathlist' with the current node, the node preceding the target node in the shortest path to it. The code is also written so that if there are two or more paths with the same length, they are all recorded in 'pathlist' using lists. After this, adjacent nodes are sorted by their shortest distance to the start node in 'distlist' and dijkstra_alg() is called again on all the adjacent nodes in order by increasing distance. Essentially, dijkstra_alg() takes initialized 'pathlist' and 'distlist' dictionaries, searches through the entire grid, visiting every node and checking every path, updating the dictionaries accordingly.

After 'distlist' and 'pathlist' are generated by dijkstra_alg(), we have enough information to deduce the shortest path to each node in the graph from the start node. A basic in-console user interface is also written, prompting the user to input what node they want to go to from the start node. After the end node is determined, path_printer() is called by dijkstra() to print the path between the two. Like dijkstra(), path_printer() is also a main function that mostly just prints after paths are calculated using rpath(). The printing function was perhaps one of the most difficult functions to write. The way it works is by starting at the end node, and scanning backwards according to 'pathlist', a record of the adjacent node right before it in the shortest path, then going to the adjacent node and doing the same until the start node is reached. Since we had to scan backward one node at the time, we needed to not only account for the possibility of there being tied shortest paths for a certain node in the path, but also the permutations of multiple nodes with tied shortest paths. Thus, to create a function that would print not just one, but all shortest paths, we had to write the rpath() function that recursively finds all the shortest paths between two nodes. It carries a list, 'travelpath', through each recursion, updating it with the next node if it is a single node, and copies the current path into new entries in the list when the path diverges due to there being ties in the next shortest path node. Ultimately, this returns 'travelpath' which now contains a full path in each entry of the list if there are multiple. From here, path_printer() relatively easily iteratively prints each path.

The user interface includes some basic error handling and allows the user to successfully try multiple points, as well as printing all paths to all nodes from the start node with input 'all'. Overall, some unique functionalities of dijkstra() include: using tuples, integers or strings as node names, printing all permutations of the shortest path, and accounting for nodes specified to be avoided.

3 Avoidance Functionality

In addition to the aforementioned description of our implementation of Dijkstra's algorithm as well as a Grid Class, for our project we decided to add an additional parameter of "avoidance".We recognized that merely having Dijkstra's to dictate a path of shortest distance was highly idealized and would only be realistic in a vacuum. In actuality, there are many other factors that might impact one's decision of route or the time taken to traverse a path rather than merely distance. For instance, while navigating a city, there might be certain areas known to have heavy traffic, or other streets that are closed for construction. Therefore, you might have prior knowledge of particular locations that you wish to avoid, even if "by the crow fly's" purely from a distance standpoint - it might be the believed shortest path. As such, within our dijkstra() function, by specifying nodes that you wish to avoid (this parameter is vectorized as well - so you can input multiple nodes at once), it retroactively alters all adjacent distances to the nodes in question to an infinite distance, so that upon running Dijkstra standard algorithm, those nodes are for certain left off the resultant ideal path. Ultimately, this project aided in demonstrating how an algorithm can begin to be made more adaptive in order to be more correctly implemented for a real-world model or scenario.

4 Built-in Test and Examples

Our project uses 4 test functions to illustrate the scope of our Dijkstra functions and Grid Class. test() runs Dijkstra's algorithm on a grid partially manually created with specified edge connections and distances using Grid.edge, and the remainder of potential connections between vertices created via the Grid.fill function which assigned a default distance value of 12. The letter named vertex grid also showed the ability of the dijkstra() function to modify the ideal route when specifying nodes to avoid. test2() illustrates similar functionality although now on a numerically named vertex grid that was entirely manually created. Additionally, test2() shows the ability to input a vectorized list of nodes to avoid and the algorithm adjusts the path accordingly. test3() has the grid creation done in a different method than manual input. Rather, it parses in a .txt file that contains adjacent horizontal and vertical distances between nodes and creates a corresponding grid with (m,n) coordinates notation similar to a matrix. Even with an input start location as a tuple due to the coordinate nature of the grid within test3(), the Dijkstra's algorithm we constructed is robust enough to still run and output the shortest route, as well as taking into consideration, once again nodes to avoid, which still are able to be vectorized even in tuple notation. Lastly, test4() displays the ability of our dijkstra() function to not crash upon determining that there are multiple paths that are equal and qualify for the shortest route - and rather displays all such different paths that one could take that are equal in shortest distance.