CS-312 Project 2 - Dijkstra's Algorithm

Cameron Jewett, Marissa Kulz, and Chris Sousa

Dijkstra's Algorithm

The basics

Dijkstra's Algorithm is a simple pattern used to find the distance from a starting node to all other nodes in a weighted graph. While it was named for Edsger Dijkstra, who published his design for pathfinding in 1959, it (or a near analogue of it) had been described prior by other mathematicians.

The basic pattern is simple:

1. Mark all nodes unvisited.
2. Assign starting values of ∞ except the starting node, which is set to 0.
3. From the starting node, set all adjacent node's distances to that of the edge between.
4. Mark the starting node visited.
5. Take a node that now has a distance.
6. Compare the total distance from that node to each adjacent node to the existing distance, and assign the adjacent nodes the smaller of the two.
7. Continue until all nodes have been visited, or the smallest unvisited node has a distance of infinity(indicating disconnected nodes.)

Implementation

We implemented this algorithm to run on a **Weighted Digraph**. While the only strict requirement is that the algorithm is intended for weighted graphs with no negative-weight edges, allowing it to run on a digraph will by nature include the ability to run on an undirected graph, at least using an adjacency matrix. While this inflates runtimes, it is negligible on a basic demo application like this.

Our `Weighted_Digraph` Class contains a collection of `std::vector` objects in order to store all the necessary data. The graph itself is stored as a `std::vector<std::vector<int>>`, which gives us a dynamically allocated two-dimensional array to hold the adjacency matrix. The shortest paths calculated by Dijkstra's algorithm are recorded in a `std::vector<int>`, while the paths are stored in a `std::vector<std::string>`. Additionally there is a `std::queue<int>`, which contains the indices of any nodes which have been reached, but not yet searched *from*.

In order to generate our internal graph, it must be read from a file. The file contains whitespace separated data in the form of a matrix. Each line is one row of the matrix, each entry an integer. The integers represent the weight of an edge going from the node on that row, to the node on that column. That is, position *(i, j)* in the matrix represents the edge weight **From** node **i to** node **j**. If there is no edge, it should be set to -1.

With the adjacency matrix examined, we can discuss the actual algorithm. What our implementation does once the file is loaded is start from vertex `0`. It generates the starting values of the starting node as 0, and the remaining nodes as -1 to represent infinity. Since Dijkstra's algorithm will not handle negative weights, this is fine. We put node `0` into the queue, and enter the while loop. At each iteration, the queue is checked for contents, and the first item

popped out for assessment. The matrix is scanned across the appropriate row for adjacencies. When an adjacency is found, we compare the existing shortest path to the distance from the current node across the discovered edge, and take the shorter of the two. Anytime a node's current distance is updated, that node is added to the queue so that we can check its adjacencies and verify that there is not now a shorter path. Once the queue is empty, we have traversed all possible edges and found the shortest paths.

This is not a strict adherence to the pattern, which generally require marking the current smallest distance as permanent at each iteration. However, it still follows the correct algorithm, and will come to the correct answers.

Additionally, we are recording the actual paths, not just the shortest distances. This is primarily for demonstration purposes.