Caleb Way, Ethan Baldwin, Parker Cuddeback
MATH-318: Graph Theory

Dijkstra's Algorithm and Applications

I. Introduction

Throughout the course of the semester, we have been introduced to numerous theorems and concepts related to the field of Graph theory, each varying in complexity and applicability. Out of all the topics we discussed, however, we decided that the best concept for our project to focus on was the applications of Dijkstra's algorithm to real world scenarios. Given Caleb and Parker's extensive history in coding languages, as they are both Computer Science majors and Math minors, and Ethan's more math-oriented mind, majoring in Mathematics and minoring in Statistics, we reasoned we would be able to work efficiently together to develop an implementation that worked as desired. We initially had some difficulty in narrowing down a specific application, however we eventually settled on mapping systems and the application of the algorithm in determining the shortest path between two real points. After some consideration, we further narrowed down our implementation of the geographical area from the whole of Canton to the University and its buildings.

II. Concept

We began by examining Dijkstra's algorithm, as given in class. In a fully connected and weighted (di)graph, beginning with a starting vertex and labeling it with a current weight (0), we shade the current vertex and travel to the next unshaded vertex, following the path of lowest weight. After arriving at the next vertex, we add the weight of the edge to the current weight, label our new current vertex with this weight, and shade this new vertex and the edge we traveled along. We then continue this step until we have shaded all vertices in the graph. The result of this algorithm produces a Dijkstra tree, which outlines the shortest path from a given vertex to every other vertex in the tree.

The small example we focused on was a connected graph with 17 vertices and 25 edges, labeled A-Q, with the vertex K having the highest degree of 5 connections. We initially made this example so that we could implement our function to output the equivalent of performing these operations by hand. We checked numerous possible inputs, such as the shortest paths for A-M, L-H and O-C, calculating lengths of 21, 35, and 22, for each respective path.

III. Initial Implementation

We decided to begin our implementation in the Python programming language, as it is a language we are all familiar with, and rather than focusing on the whole of St. Lawrence University, to begin we focused on implementing the algorithm and ensuring it works for the aforementioned example. We created a class, called WeightedGraph, to both model the contents of a connected digraph as well as define functions to manipulate the data based on the algorithm. We first defined the contents of a WeightedGraph object as containing an adjacency and weight matrix, utilizing the __init__() function of class creation. This function allows us to specify the names and types of values associated with the variable. We then began the initial implementation of Dijkstra's algorithm. We begin with four variables to keep track of information regarding the algorithm;

- parent - A parent mapping of vertices, initially containing the current vertex with a child vertex of None (no child has been discovered yet).
- cost - The current cost of each vertex, as mapped through the algorithm's traversal
- new - A tuple value within a list, representing a stack, to keep track of the newest current vertex, along with its associated cost.
- old - A set of values, meant to represent the vertices we have already encountered (shaded).

As we have created the new list with the only tuple value of the parent vertex and its current weight of 0, we remove this value from the new list. We then add it to the old set to ensure we never travel back to this vertex. From here, we check for all the adjacent vertices based on the given graph, and calculate the sum of the current cost and the cost of the edge selected. We then examine the adjacent vertex; if it is not contained within list keeping track of costs, or the current cost is less than another path contained within costs ending at the same vertex, then we move to this adjacent vertex and add it to our parent mapping, with the previous vertex as the parent. We then update the cost associated with the current vertex in the costs mapping, and add the vertex/associated cost to the new list. The result of this function returns the shortest path from a given vertex to every other vertex in a given connected graph, a Dijkstra tree.

The final function we defined was to calculate the best path given a target vertex and the Dijkstra tree yielded from the previous function. If the vertex is in the Dijkstra tree, we begin the path as a list containing the target vertex. We then ascend back up the Dijkstra tree, setting the current vertex to the parent of the current vertex each time and adding them to the path list in order, until it comes across the entry containing the parent vertex. We have now reached the starting vertex that was input in the Dijkstra tree algorithm with a cost of 0 and no parent vertex, and so we reverse the path list to display the initial start and an in order traversal to the target vertex.

We then implemented the adjacency matrix g to represent the previously described small example. The adjacency side of the matrix was a simple key-value pairing, with the key being each vertex, and the values associated with each key are sets of the adjacent vertices. The weight side is similarly defined, with the keys being tuple values representing the forward and backward direction of each edge, and the respective values in this case being the defined edge's weight.

Applying the algorithms to this graph for each of the aforementioned starting and ending vertices, we confirmed that we received the same values as calculated by hand; yielding an A-M path of length 21, L-H path of length = 35, and O-C path of length 22.

IV. Utilizing ArcGIS

In our initial meeting about our project with Dr. Lock, she referred us to Carol Cady, who is the head of the GIS, or geographic information system, department through the SLU Library. Upon meeting with her, Carol showed us the ArcGIS system that the university runs. Within the system, there were already pre-measured distances and maps from prior work that Carol and members of the department had created and saved. After setting us up with accounts for the system, Carol showed us a method of mapping out the university on a map. Carol then gave us the option of using one of the pre-burnt, hardcoded maps that already existed for our project, but we opted instead to create our own, giving us full and total power over what we included or excluded in our map. And so, we decided to include everything campus-side of Main Street, and Miner Street.

Our method consisted of creating a layer of edges and vertices on the existing St. Lawrence Campus map, with vertices being the entrances of buildings, and intersections between roads. Luckily, ArcGIS had built in features that would allow us to save all of our data and drop points anywhere we wanted on the map. Our edges that we plotted consisted of all the roads and walkways on the campus, as well as edges from buildings to the nearest road/vertex to assure that we can get to each and every building when running our program. In all, we included 73 points on campus to be our vertices. After meticulously laying out what would be our final map, we set to work giving each vertex and edge a unique name that would be easily recognizable, and

one that we could translate and store in our program to make for easy utilization when running our code.

After storing all of our vertices and edges in our code as both their abbreviated names as well as their full names, we took to measuring the length of each edge on our map. This was made easy by a measuring feature built into ArcGIS, which would allow us to click on a vertex, drag the cursor to another vertex, and we would be given a distance in a unit of our choosing. We decided that using meters as our unit was most appropriate, as it is internationally known and accepted as the standard for measuring. After measuring each edge, we would add the length, in meters, into the description of each edge. Once we had all 93 edges measured, we then began to translate all of our data into our code.

V. Visual Implementation

After mapping out all the information we found relevant or manageable for the project in ArcGIS, we began to implement the final component of our project. We wanted to not only be able to display the distance between any two of the implemented points for the user, rather we wanted to show them a visualization of the shortest path between their starting vertex, or current location, to every other location contained within our representation of campus. To accomplish this, we began to utilize a Python package called NetworkX, a visualization software that includes a number of different graph representations. The re-implementation of the Dijkstra tree algorithm is essentially the same; only modified to facilitate the graph representation provided by NetworkX. We ensured the function still worked as intended utilizing a smaller example, and after receiving equivalent Dijkstra trees for all inputs, we moved on to arguably one of the most difficult aspects of the project; data definition. With 73 vertices and 93 edges going in both edges, we had to perform a total of 259 data entries to accurately depict the map of campus.

We then tested the algorithm on our map of campus, testing to depict the shortest path to every other location from Owen D. Young Library as the source vertex. We initially faced some difficulty in the visualization aspect of the graph, as it was too small and the data was illegible, and required us to search through the package documentation for a method to resize the vertices, edges, text and resolution. After resolving this issue, the software was able to accurately visualize all the information regarding both the general campus map we developed, as well as the resulting Dijkstra tree with the library as the source.

VI. Applications

The applications we aimed to include in this project were manipulations of the graph in response to given starting and target vertices. We were able to accurately calculate and represent the shortest path from any of our defined buildings to any other, display this path within a tree representing the shortest path to every vertex from the start, and we could calculate the average distance of every path from the start vertex. If we were to continue this project's development any further, we would first aim to include a more accurate and in depth depiction of all the roads and buildings around campus, as well as develop a more user friendly interface to make this available to all students, possibly even replacing the current campus map. We could also potentially expand this implementation to include less obvious paths, such as cutting across grass or directly through the park.

Bibliography

Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, “Exploring network structure, dynamics, and function using NetworkX”, in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008.
https://networkx.org/documentation/networkx-1.10/reference/citing.html