Novel Implementation of Dijkstra's Algorithm

Adam Mansour[1], Noah Shehata[2], Liam Osman[3], and Reece Williams[4]

College of Engineering, University of South Florida
Tampa, Florida, USA

[1]adammansour@usf.edu U48542843

[2]nshehata@usf.edu U77436492

[3]liamo@usf.edu U27815480

[4]raw21@usf.edu U77112040

Abstract— **Dijkstra's Algorithm is a famous and commonly used method of searching a weighted graph for the shortest path between any 2 points. This paper details the Authors' assignment to implement Dijkstra's Algorithm, and the details therein of how it is implemented. An adjacency list, as well as Edge, Vertex, and the Graph classes and the methods created in the project are explained in their function and particular implementation. This paper is intended to accompany the included source code files for the project to provide a comprehensive assessment.**

I. Introduction

This paper was created as part of the submission for the Final Project of the COP 4530 Data Structures course, and the four authors worked together to create all aspects of the code within, as well as to present it according to the requirements of the assignment. The assignment was to implement an undirected weighted graph ADT and methods to perform Dijkstra's Algorithm to find the shortest path between any two vertices within the graph.

II. Design Structure

There are many possible ways of implementing a graph data structure for this purpose, and the authors chose to implement the graph ADT as an adjacency list, rather than an adjacency matrix or an incidence matrix. The adjacency list was believed to be the easiest and most efficient choice possible, due to the sparseness of the graph meaning that there will be wasted memory in a large matrix, and the access time of the list will not be significantly more than an adjacency matrix. The Edge and Vertex classes, as well as the primary Graph file's methods will be detailed in the following sections of this paper.

III. Edge and Vertex Classes

A. Edge

The *Edge* class exists to create a template for our *Edge* objects which will connect the various vertices of our graph together. The *Edge* class contains 3 public data members and a constructor method, as follows: a *Vertex* pointer *vertex1*, a *Vertex* pointer *vertex2*, and the unsigned long *weight*, while the constructor simply assigns the parameters to those values.

The choice of data type for the variable *weight* was determined to give us the largest possible range of distance values, within reason. It is *signed* because distance cannot be a negative number and this choice doubles our total range. Being a *long* instead of just an *int* gives further range of values. Although it would be possible to extend the range even further with a *long long*, this level of range was deemed superfluous and too costly to memory. Our data is also all stored as integer values, so no *double*s were required. If the data were to be modified to decimal weights, this value would have to be changed to accommodate the new values.

B. Vertex

The *Vertex* class contains 2 public data members and a constructor method, as follows: a string pointer *label*, a list of *Edge* pointers *adjList*, and the constructor which assigns those values as parameters.

The list held within each *Vertex* comprises the main active component of the adjacency list which forms our graph. Each *Edge* pointed to within this list represents a weighted connection between two vertices in the graph.

IV. Edge and Vertex Manipulation Methods

A. addVertex

The *addVertex* method is used to create and add a new vertex (or node) to the graph. This process runs as follows:

1. A string named label is passed into the function as a parameter, to be the name of the vertex.
2. Each vertex is checked for already existing with the specified label. Because each vertex in the graph must have a unique label or identifier, this check ensures that none of the vertices are confused for each other.
3. If the vertex does not exist already, then a new *Vertex* is created with the previously passed label parameter.

B. removeVertex

The process in the *removeVertex* method is more complicated than the process to add a new vertex. It has the same passed parameter, the string *label* which identifies the vertex to be removed. The process follows in these steps:

1. The vertex of the following label is searched for and identified. If the vertex is not found and therefore does not exist to be deleted, return prematurely.
2. The edges connecting to the chosen Vertex are traced to their opposite vertices, and those vertices' connections are iterated through to identify this connection.
3. The edges connecting the vertices with the soon-to-be-deleted vertex of choice are removed, and their memory is freed up for the system.
4. Finally, after all connections to it are destroyed, the chosen vertex is deleted safely.

C. addEdge

The *addEdge* method is the longest and most complicated method of graph manipulation. Its purpose is to create an *Edge* between 2 *Vertex*'s. It is called with 3 parameters: a string *label1* representing the first *Vertex*, a string *label2* representing the second *Vertex*, and an *unsigned long* value for *weight*. The end result is that there will be an *Edge* connecting the two vertices passed in the argument, with the weight value also matching the weight passed in the argument. It accomplishes this goal with this procedure:

1. The program identifies and records the two vertices of the labels passed as arguments.
2. If one or both of the vertices are not found, then return from the function prematurely.
3. Check if the two vertices are the same, and return prematurely if so.
4. Check the two vertices for an already pre existing edge between them. If so, return prematurely.
5. A new *Edge* is created with the passed weight value
6. The *Edge* is pushed onto the lists of both vertices, thus connecting them to each other in the graph

D. *removeEdge*

The *removeEdge* method serves to remove an edge between two vertices, whose labels are passed into it as function parameters. The *removeEdge* function follows this procedure:

1. Check if any of the vertices are not within the graph by searching through it for both of them. If either is not found in the graph, return prematurely.
2. Iterate through the list of edges, searching for whenever the corresponding labels match the two given vertices.
3. If the edge is found, it gets deleted.

After this process is done, the edge is removed and the graph is clear to proceed. This only has to happen once because of the undirected nature of the graph type that was chosen to be constructed. There can only be one edge between any two vertices, with one weight value. If this graph had been made with directed edges, then this process would be much harder and would require having to terminate both directed edges between the vertices.

V. Dijkstra's Algorithm

Dijkstra's Algorithm is a popular algorithm conceived by Dutch computer scientist Edsger W. Dijkstra in 1956 that is used for solving multiple single-source shortest path problems having weighted edges in the graphs. It is commonly used to find the shortest path between any two specifically called upon vertices in a graph.

The algorithm generates and records a set of visited vertices and a set of unvisited vertices. starting at the source vertex, it iteratively selects the unvisited vertex with the smallest tentative distance from the source. Then, it visits the neighbors of said vertex and updates their tentative distances if a shorter path is found. The process continues until the destination vertex is reached, or in the case that all reachable vertices have been visited.

The need for Dijsktra's algorithm came from the rise in popularity of applications where assessing the shortest path between two points was crucial. As an example, Dijkstra's algorithm could be used in the routing protocols for computer networks and also used by map systems to find the shortest path between starting points and their destinations.

Basic requirements for the implementation of Dijkstra's algorithm include: a graph, source vertex, destination vertex, and non-negative edges. For the graph component, Dijkstra's Algorithm can be implemented on any graph, but in this example we use an undirected weighted graph. The graph is represented as a set of edges and vertices. For the Source Vertex, this is the starting point of the search for the algorithm. The Destination Vertex marks the specific destination in which the algorithm will be terminated once it is reached as an end location. Lastly, non-negative edges are imperative to the functionality of Dijkstra's Algorithm. The algorithm will only work with positively weighted graphs because during the process the weights of the edges are added to find the shortest path. hence a negative weight will break the algorithm. In our example, there are no negative weights, but only positive integer values.

The *shortestPath* function in this project is what implements Dijkstra's Algorithm to calculate the shortest path between any two vertices. It follows this protocol:

1. Search the graph for the label provided as a start vertex in the function parameters. If it does not find a matching vertex, then prematurely return.
2. Search the graph for the label provided as an end vertex in the function parameters. Likewise, if it does not exist, prematurely return.
3. Initialize all edge distances as "infinity" and the previous vertices as empty
4. Set the distance to the start vertex as 0
5. Make a vertex queue which holds the vertices to be processed
6. Start looping to process the graph. For each cycle, the program compares the weighted values on each of the edges between the neighboring vertices of the vertex being examined. It updates the vertex queue if it finds a shorter distance than previously found, and by doing so will record the shortest path by the end of the loop. This is repeated until the graph is fully processed.
7. Now, the program has to reconstruct the shortest path found to the final destination. If it fails in creating this path, it throws an error that there is no possible path between the given vertices. Otherwise, it creates a vector path of the labels in the path.
8. The path is then processed backwards in reverse order to get the proper order of start to finish, as it had previously been added in reverse order.
9. The method returns the final shortest path.

VI. Difficulties encountered

In the process of this project, a number of difficulties were encountered. The miscoordination of a team of 4 contributors in one project, conflicting ideas of design choices, and scheduling concerns are all examples of some difficulties which arose out of doing a team project together. The team also encountered difficulties in passing all of the test cases used in the project.

Oftentimes during development, team members would unwittingly interfere with each others' plans and current efforts to contribute to the project. Since a large program features so many coordinated parts, any small adjustments such as changing the parameters of a function by one argument can completely break the function of another. Developers must ensure that they agree upon standards and plans, such as in a header file, which are then not changed, but followed and implemented independently. If changes are required, proper communication and teamwork should ensue to ensure that all affected parts of the program are adjusted properly and maintain functionality.

Even so, design choices must be agreed upon beforehand, and this can be problematic when teammates disagree. Design choices conflicted when one team member wished to implement the program using an adjacency matrix rather than the adjacency list which was eventually utilized. The team had to hold a meeting and discuss this decision, and convince each other on the benefits and negatives of each side. Although there is a faster query access time within an adjacency matrix, the team member in question was eventually convinced of how other design goals, such as memory usage and usability, should be taken into account along with the perspectives of other team members.

Just getting all the team members together and working at the same time proved challenging. In fact, there was only one time in which all the team members were all simultaneously working together on the project. For the rest of the time, the team had to coordinate asynchronously due to a myriad of reasons. From differing schedules, distractions and emergencies in life, and introverted social aptitudes, coordinating a team to work together can prove to be a serious challenge and hindrance to team productivity.

Although initial progress on the program went smoothly and without significant errors, the final testing and confirmation of provided test cases proved tedious. Many minor corrections and major design changes had to be implemented, such as separating the *Edge* and *Vector* classes into their own separate classes, instead of being within the *Graph* class as we had originally designed it. Each time, we could solve some test cases, but more errors would arise and could not be quelled.

VII. Conclusion

This assignment to create a novel implementation of Dijkstra's Algorithm provided the team with a great experience in the trials and tribulations of teamwork, and the issues which can arise. The team, after much debate, agreed upon results and managed to design a working implementation of Dijkstra's Algorithm, using an Adjacency List Graph and a number of methods to manipulate and use it.

Acknowledgment

The authors would like to thank Daniel Jordan and Jonathan Koch for their helpful work as Peer Leaders to educate us and guide us in our journey exploring Data Structures and Computer Science. Additional thanks to the Teaching Assistants and their hard work supporting the class.