The primary idea behind our graphing library is to make it so that a user can very simply and easily perform commonly used algorithms on a set of data.

1. **The Graph**

The first step to any good Graph library should be a way to represent a graph. We have two methods available to the user:

1. The user can simply supply an adjacency matrix (Container<Container<double>>) or list (Container<Container<pair<unsigned long, double>>>) representation to our container.

auto graph= Graph(matrix);

1. The user can also use our graph\_helper class. We will use this option for the tutorial

To start with, we call make\_graph\_helper and decide whether we would like to use an adjacency matrix or list representation (we use a matrix in this tutorial):

auto graph\_helper = Graph\_Helper<string>(g);

Graph\_helper provides the user with a tuple containing: the matrix, a list of indexes, and an associative vector to associate the indexes with their data:

auto matrix = get<0>(matrix\_tuple);

auto code\_index\_map = get<1>(matrix\_tuple);

auto index\_code\_vector = get<2>(matrix\_tuple);

The matrix that is returned can then be passed into our Graph container.

auto graph= Graph(matrix);

At this point, if we wanted to, we could use a few handy methods to get some more information:

graph.get\_num\_edges(); //returns the number of edges in the graph

graph.get\_num\_vertices(); //returns the # of vertices

graph.get\_weight(0,5); //returns the weight of vertex[0][5]

graph.has\_negative\_weights(); //returns true if there are

//negative weights

1. **Graph Helper**

At this time, it is probably helpful to discuss a bit more about the graph helper. Upon constructing the graph helper using make\_graph\_helper(<MATRIX or LIST>), we can an edge to the graph by simply calling graph\_helper.add\_edge(Vertex origin, Vertex destination, double distance). The most useful part of the graph\_helper object though is the tuple that it returns. The 0th element of the tuple is the graph in the user specified representation (either adjacency list or adjacency matrix). The 1st element is a map that associates vertices with their indexes. The 2nd element of the tuple is vector which associates indexes with the vertex map. If the user specified a matrixd when they constructed the graph helper than they can access this tuple by called get\_matrix\_tuple. Similarily, if we had constructed our graph helper as a list representation we can call get\_list\_tuple().

1. **The Algorithms**

The next step is to run through our algorithms. Our graph structure is immutable allowing a user to perform multiple algorithms on the same set of data without needing to worry about it being overwritten. To run an algorithm all we need to do is call Algorithms::<algorithm name> and pass in the relevant arguments.

The arguments for each algorithm are:

Bellman-Ford: graph, start\_index, and stop\_index

Dijkstra’s: graph, start\_index, and stop\_index

Prim’s: graph

Tarjan’s: graph

Johnson’s: graph

For example, to run Dijkstra’s we would call:

auto path = Algorithms::Dijkstras(graph, origin\_index, destin\_index);

Bellman-Ford and Dijkstra’s return a vector<pair<unsigned long, double>>. Each pair is a vertex in the path. The unsigned long is the index of the vertex and the double is the weight to get to that vertex.

Prim’s returns a vector<vector<pair<unsigned long, double>>>. Each vector is a vertex and its predecessor. Each pair in the vector represents a vertex. The unsigned long is the index of the vertex and the double is the weight to get to that vertex.

Tarjan’s returns a vector<unsigned long>. The vector contains the indexes of each articulation point.

Johnson’s returns a vector<vector<vector<pair<unsigned long, double>>>>. The left index denotes the starting vertex and the middle index denotes the ending vertex. Accessing this returns a vector of pairs, which has the exact representation of what is returned in Dijkstra’s and Bellman-Ford.

1. **Output**

Because we used the graph helper, we can now use our associative vector to make sense of the indexes that are returned.

For example we can iterate through the path that Dijkstra returned to get a String that each index represents:

for(auto& e : path) {

auto code = index\_code\_vector[e.first];

}

1. **A Worked Example: The Shortest Path From Honolulu to Moscow**

For this example, we’re going to make use of World Airline flight data. The data can be downloaded from: <http://openflights.org/data.html>. If you’d like to follow along, please download the data and put it in a mysql database (other databases will work just make sure to adjust accordingly). For this tutorial, we’re going to use Dijkstra’s algorithm:

We start in our main by verifying that the user specified the correct number of arguments and store the origin vertex and destination vertex :

if(argc != 3) {

cerr << "usage: " << argv[0] << " <origin> <destin>" << endl;

exit(1);

}

string origin = argv[1];

string destin = argv[2];

We then connect our database and check for errors:

// Get Environment Variables

char \*url = getenv(url\_env\_var);

char \*username = getenv(username\_env\_var);

char \*password = getenv(password\_env\_var);

char \*database = getenv(database\_env\_var);

// Check Missing Variables

auto errs = vector<string>();

if(!url) errs.push\_back(url\_env\_var);

if(!username) errs.push\_back(username\_env\_var);

if(!password) errs.push\_back(password\_env\_var);

if(!database) errs.push\_back(database\_env\_var);

// If Missing, Throw Exception

if(errs.size() != 0) {

string msg;

for(string s : errs) {

msg += s + '\n';

}

throw logic\_error("\nMissing Environment Variables:\n" + msg);

}

// Setup Database Connection

try {

driver = get\_driver\_instance();

connection = driver->connect(url, username, password);

connection->setAutoCommit(0);

connection->setSchema(database);

statement = connection->createStatement();

} catch (SQLException& e) {

// Termiante

cout << "SQL Exception: " << e.what() << '\n' << e.getErrorCode() <<

'\n' << e.getSQLState() << endl;

terminate();

}