Tutorial

Team Graph_Algos

Sean Garvey (sjg2174)
Minh Truong (mt3077)
Sammy Tbeile (st2918)

The primary idea behind our graphing library is to make it so that a user can very simply and easily perform commonly used graph algorithms on a set of data. The two main component of our library are our Algorithms and Graph classes. We have also developed a Graph_Helper class that enables a user to initialize a Graph from their data.

1) The Graph
The first step to any good Graph library should be a way to represent a graph.

Our Algorithms class requires a graph representation that is initialized from our Graph class. Our Graph class takes in the users' representation and stores its own representation internally in the form of an adjacency list and matrix. Our primary motivation is to validate the structure and data integrity of the user's graph representation as well as enable our algorithms to be able to pick which internal graph representation allows it to run as efficiently as possible.

The user can simply supply an adjacency matrix or list representation to our generic Graph class. The interface is of the type: template<typename Container> Graph(Container container);

For an Adjacency Matrix: Container<Container<double>>
For an Adjacency List: Container<Container<pair<unsigned long, double>>>

The generic 'Container' type can be of any ordered and iterable collection (vector, list, array etc..).

include “graph.hpp”

Graph graph = Graph(Adjacency_Matrix);
Graph graph = Graph(Adjacency_List);

2) Graph Helper

If the user does not have any type of graph representation, they would be able to construct a Graph from their data with the help of Graph_Helper.
To start with, we call the constructor 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>(MATRIX);
```

Upon constructing the graph helper, we can add 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 matrix when they constructed the graph helper then they can access this tuple by calling get_matrix_tuple. Similarly, if we had constructed our graph helper as a list representation we can call get_list_tuple().

The tuple can be used as follows:

```
  auto matrix          = get<0>(matrix_tuple);
  auto code_index_map  = get<1>(matrix_tuple);
  auto index_code_vector = get<2>(matrix_tuple);
```

The matrix, which is an adjacency matrix representation of a graph, that is returned can then be used to initialize 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
```

3) The Algorithms

After creating a Graph from their own representation, the user can now use the algorithms in our Algorithms class. 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.

include “algorithms.hpp”

The interface for each algorithm are:

BellFord:
vector<pair<unsigned long, double>> BellFord(Graph graph, unsigned long origin_index, unsigned long destin_index);

auto path = Algorithms::BellFord(graph, origin_index, destin_index);
BellFord returns vector of pairs that can be use to construct a path from origin_index to destin_index.

Dijkstras:
vector<pair<unsigned long, double>> Dijkstras(Graph graph, unsigned long origin_index, unsigned long destin_index);

auto path = Algorithms::Dijkstras(graph, origin_index, destin_index);
Dijkstras returns vector of pairs that can be use to construct a path from origin_index to distin_index.

Johnson's:
vector<vector<vector<pair<unsigned long, double>>>> Johnsons(Graph   graph);

auto paths = Algorithms::Johnsons(graph);
Johnsons returns a matrix of paths. Let 'result' be the return value of Johnsons, result[0][100] would return the path from vertex 0 to vertex 100. Accessing this returns a vector of pairs, which has the exact representation of what is returned in Dijkstra’s and Bellman-Ford.

Tarjan’s:
vector<unsigned long> Tarjans(Graph graph);

auto art_points = Algorithms::Tarjans(graph);
Tarjans returns a vector of the vertices in the graph that are articulation points.

Prim's:
vector<vector<pair<unsigned long, double>>> Prims(Graph graph);

auto mst = Algorithms::Prims(graph);
Prims returns a spanning tree that is represented as an adjacency list representation of a graph.

4) 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];
}
```

5) 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) {
		// Terminate
		cout << "SQL Exception: " << e.what() << '\n' <<
e.getErrorCode()
			<< '\n' << e.getSQLState() << endl;
		terminate();
	}
```

We then want to query our database to associate our airport codes with their GPS coordinates:

```
	auto code_gps_map = unordered_map<string, pair<double, double>>();
	try {
		// Query
		auto result_set = statement->executeQuery(
			"SELECT * FROM Airports");
		while(result_set->next()) {
			auto code  = result_set->getString("FAA");
			double lat = result_set->getDouble("Lat");
			double lng = result_set->getDouble("Lng");
			try {
```

```
                        code_gps_map.at(code);
                        throw logic_error("Airport code: " + code + "
            already
                              exists in code_gps_map");
                  } catch (out_of_range& e) {
                        code_gps_map[code] = make_pair(lat, lng);
                  }
            }
      delete result_set;
       } catch (SQLException& e) {
            // Terminate
            cout << "SQL Exception: " << e.what() << '\n' <<
e.getErrorCode()
                  << '\n' << e.getSQLState() << endl;
            terminate();
       }
```

We then want to make our graph helper object and add to it our vertexes and edges by querying our database and using the add_edge() method:

```
      auto graph_helper = Graph_Helper<string>(g);
       try {
            auto result_set = statement->executeQuery(
                  "SELECT * FROM WorldPaths");
            while(result_set->next()) {
                  auto origin = result_set->getString("Origin");
                  auto destin = result_set->getString("Destin");
                  double dist = result_set->getInt("Distance");
                  graph_helper.add_edge(origin, destin, dist);
            }
            delete result_set;
       } catch (SQLException& e) {
            // Terminate
            cout << "SQL Exception: " << e.what() << '\n' <<
e.getErrorCode()
                  << '\n' << e.getSQLState() << endl;
```

```
            terminate();
        }
```

Now that we've extracted all our relevant data, we can disconnect the database:

```
        try {
            delete statement;
            connection->close();
            delete connection;
        } catch (SQLException& e) {
            // Terminate
            cout << "SQL Exception: " << e.what() << '\n' <<
e.getErrorCode()
                 << '\n' << e.getSQLState() << endl;
            terminate();
        }
```

Now we're ready to begin the fun part! We can use our graph helper to construct our graph:

```
        auto matrix_tuple     = graph_helper.get_matrix_tuple();
        auto matrix           = get<0>(matrix_tuple);
        auto code_index_map   = get<1>(matrix_tuple);
        auto index_code_vector = get<2>(matrix_tuple);
        auto graph            = Graph(matrix);
```

Now we want to get our airprort indexes, this is as easy as using our code_index_map:

```
      unsigned long origin = code_index_map.at("HNL") //Honolulu =HNL
      unsigned long destin = code_index_map.at("DME") //Moscow =DME
```

Now we call Dijkstra's:

```
      auto path = Algorithms::Dijkstras(graph, origin_index, destin_index);
```

Now for the last step, we can output the shortest path to a csv and graph it using the included GraphPlotter.py!

GraphPlotter is run via the call python GraphPlotter.py. For Dijkstra's or Bellman-Ford's Algorithm the input file should be in the form of <latitude>, <longitude>, <weight>, <Airline Code>.

Following these instructions we can output to the csv in the format:

```
       cout << fixed;
 for(auto& e : res) {
      auto code   = index_code_vector[e.first];
      auto gps_pair = code_gps_map[code];
      auto lat    = gps_pair.first;
      auto lng    = gps_pair.second;
      auto weight   = e.second;
      cout << setprecision(8) << setw(12) << lat << ", " << setw(13)
<<
            lng << ", " << setprecision(0) << setw(6) << weight << ",
      "
            << code << '\n';
 }
```

Now all we need to do is run the GraphPlotter program with the command:

python GraphPlotter.py

After following the prompts, you should see your path displayed!

6) GraphPlotter
GraphPlotter.py

Sample application to plot the graph generated by the Algorithms library on a world map. GraphPlotter is run via the call python GraphPlotter.py <input_file>. It takes as an argument an input file in the form of a csv.
For Dijkstra's or Bellman-Ford's Algorithm the input file should be in the form of:
<latitude>, <longitude>, <weight>, <Airline Code>

For Prim's Algorithm, the input file should be in the form of:
<latitude>, <longitude>, <weight>, <Airline Code>, <neighbor1>_<neighbors2>_... Values inside neighbor (longitude, latitude, weight, and name) should be separated by semicolons.

For Tarjan's algorithm, the input file should be in the form of:
<latitude>, <longitude>, <name>

For Johnson's algorithm, the input file should be in the form of:
<latitude>, <longitude>, <weight>, <name>

Sets of paths should be separated by a line that only says the word done on it.

Upon starting the program, the user will be prompted to choose what type of algorithm they would like to use from a list of options:

```
Menu:
1. Shortest Path Algorithm
2. Prim's Algorithm
3. Tarjan's Algorithm
4. Johnson's Algorithm

Please input the number of your option:
```

The user will then be prompted to input where their input file is stored:

```
Where is your data stored? Please input the file name:
```

The application will then plot the inputted file.