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 or list 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 anddecide whether we would like to use an adjacency matrix or list representation (we use a matrix in this tutorial):

auto graph\_helper = make\_graph\_helper(MATRIX);

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);

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);