Final Report

Constructing the graph:

When constructing the graph we read the route data into a vector of vectors and the airport data into two maps. One is mapping the airport id with the position of it, the other is mapping the code for the airport with the id. We used the data file to construct the airport graph and the outcome is shown in the test cases.

```
TEST_CASE("Correctly reading the file") {
  Graph g_("routes_testcase 1.txt", "airports_testcase 1.txt");

  REQUIRE(g_.routes.size() == 16);
  REQUIRE(g_.position.size() == 8);
  REQUIRE(g_.ID["TIA"] == "1190");
  REQUIRE(g_.ID["LIME"] == "1525");

 //BV,1463,BGY,1525,TIA,1190,,0,737
  REQUIRE(g_.routes[0][2] == "BGY");
  REQUIRE(g_.routes[0][3] == "1525");
  REQUIRE(g_.routes[0][4] == "TIA");
  REQUIRE(g_.routes[0][5] == "1190");
  //BV,1463,FCO,1555,PMO,1512,,0,737
  REQUIRE(g_.routes[8][2] == "FCO");
  REQUIRE(g_.routes[8][3] == "1555");
  REQUIRE(g_.routes[8][4] == "PMO");
  REQUIRE(g_.routes[8][5] == "1512");

  std::pair<std::string, std::string> TIA("41.4146995544","19.7206001282");
  std::pair<std::string, std::string> FCO("41.8002778","12.2388889");
  REQUIRE(g_.position["1190"] == TIA);
  REQUIRE(g_.position["1555"] == FCO);

TEST_CASE("Generating the airport graph correctly") {
  AirportGraph a_("routes_testcase 1.txt", "airports_testcase 1.txt");
  REQUIRE(int(a_.getEdgeWeight("TIA", "FCO"))==624);
  REQUIRE(int(a_.getEdgeWeight("PMO", "BRI"))==455);
  REQUIRE(a_.getEdgeLabel("BLQ", "TIA")=="BLQ---to---TIA");
}
```

Breadth-First Search:

Performing a breadth-first search on the airport graph we constructed.
The output is a vector which contains all the airports which are visited in the BFS Traversal.
Example: Input (FCO), Output (FCO, TIA, PMO, DME, CTA, BLQ, BGY, BRI)
In the test cases we use small datasets for faster running time, but we also tested that the traversal works for the original large dataset which contains all the airports and routes.
The BFS traversal is effective in searching all the adjacent airports that can be reached from the start airport, and it provides foundation for the Dijkstra's shortest path algorithm.

Dijkstra's shortest path algorithm:

The inputs are a start vertex and an end vertex. The output is a vector which contains all the airports of the shortest path between the start vertex and the end vertex.
Example: Input ("BGY", "DME") Output ("BGY", "TIA", "FCO", "DME")
BFS assumes that all edges have the same weight. Different from BFS, Dijkstra is based on the weighted graph. It could find the shortest path between two locations. In our test case, output is exactly the same as the expected result.

Map projection:
project the shortest path on the world map.
the Input is the dataset and the start and end airport. example:("routes.txt", "airports.txt", "CGO", "ORD")
the output is the projected map.