1. **设计思路与程序结构：**

本project使用C++语言以一个比较OOP的方式实现，先设计了一个基类ShortestPathAlgorithm，提供了两个最短路算法公用的通过所有点的前驱得到所有最短路的函数(递归实现)，也提供了供两个最短路算法去实现的寻找最短路的虚函数。该部分代码放在shortest\_path\_algorithm.cc/.h中。此外，这两个文件内还实现了记录一条最短路的必要信息的类ResultPath。

然后分别在Bellman-Ford.cc/.h和Dijkstra.cc/.h中实现了Bellman-Ford算法和Dijkstra算法，二者都继承了类ShortestPathAlgorithm。其中Dijkstra算法是用优先队列实现的版本。这里简单说明一下如何修改两个算法，使之能够找到所有最短路。两个算法的修改方式很相似，都是在进行边(u,v)松弛操作的时候，把大于的条件改为大于等于，但是如果是大于，则要删除v点已经记录的所有前驱。

而图的存储则是用邻接边来实现。在文件traffic\_map.cc/.h中实现了图的存储及其文件读取。

最后是顶层的navigation.cc/.h，类Navigation通过调用其他文件中的类实现了程序的种种功能，并对外提供了运行函数。类中有一个成员是ShortestPathAlgorithm类型的指针，而类的构造函数中会让用户选择最短路算法来给该指针赋值。

main函数在test.cc中，其通过构造Navigation对象并调用其运行函数来运行/测试程序的功能。

此外，本project的代码风格遵循Google C++ Style Guide。

1. **功能与使用说明**

程序对于两个单独的站点间的最短路提供了两种输出模式：输出一条换成次数最小的最短路，输出所有最短路。对于多个站点间的最短路提供了三种输出方式：输出一条换乘次数最小的最短路，输出所有最短路，分段输出(即分别输出每相邻的两个目的站间的最短路，而这每段的输出方式则与两个单独的站点间的最短路的输出方式相同)。

编译全部.cc文件得到可执行文件(a.exe)后，直接运行，按提示操作即可。

P.S.程序具有较强的鲁棒性，一般的错误或无效输入将不会影响程序的正常运行，用户只需重新输入指令即可。但文件操作的对象文件应该具有规范的格式，否则程序运行的结果可能会与预期不同。

1. **一些设计细节和我的思考**

像两种算法的逻辑这种大同小异的东西就不再赘述。主要讲讲一些特别的或者说可以变通的地方。

首先是图的存储，由于站点数总数较少(324个)，故使用一个以用邻接边作为模板参数的vector为元素的数组去存图(graph\_)，每个点对应一个数组下标，这样做的好处是可以迅速访问每一个点的每一条出边。但在实际情况中，站点是以字符串形式呈现的，所以需要建立string与int下标之间的联系。于是在TrafficMap类中，我维护了两个成员，一个是string类型的数组(station\_int\_to\_str\_)，其下标与站点名一一对应，另一个是以string为键int为值的map(station\_str\_to\_int)，每个站点名对应的int值就是其在下标。这样，我在内部实现中就只把点当作是int类型的数字，很大的提高了算法和程序的效率。而string只作为对外展示的对象，只需在读入、输出或是不同模块连接时进行string和int的转化即可，而这个转化的开销是相当低的(string到int为logN，int到string为1)。

此外，本程序的可扩展性很强，例如如果想要添加步行、公交等其他出行方式，只需在邻接边类中添加相应参数，然后做一些必要的添加即可。

另外，本程序在对两条最短路的连接时效率较低，这将会在站点输入特别多，需要连接次数特别多的时候体现的非常明显。思考后发现主要是两个原因，一个是函数传参问题，应该传引用，参数是两条最短路，其结构比较复杂，包含的数据量也很多，传值复制进去开销很大。二是内部实现算法问题，不应当返回一个新的路径，然后把两条要连接的最短路copy进去。这还是因为数据量的问题。应当直接修改前面的条路，直接将后面的路并入其中即可。(实际上这样并不好实现，修改后面的路是最快的，但是由于要传引用，加之后面的路还要再作为下一轮循环的前面的路复用，故不能修改后面的路的值，所以只能这么做)。综上，在这一点上可优化的空间是较大的，但由于期末时间实在太紧张，加之这一点并不影响程序正常、正确运行，故没有实现优化。