Project 4: GooberEats

Time due: 11 PM, Thursday, March 12

Introduction

The Goober Corporation, the 17th most popular provider of ridesharing in the greater Westwood area, has decided to branch out beyond offering basic ridesharing services into robotic meal delivery services, and launch a service called GooberEats. GooberEats uses hundreds of mobile robots to deliver food to customers. Each delivery robot must pick up a bunch of food at a central meal preparation depot and then deliver this food in the most efficient way possible to one or more customers, finally returning back to the depot after all deliveries have been made to get more food for the next round of deliveries.

All delivery services leverage delivery logistics systems, and Goober Corp. needs one built for their service. A delivery logistics system is one that takes one or more delivery requests (e.g., deliver pasta primavera to the corner of Veteran and Strathmore and a double-double to the corner of Westwood and Wilshire, etc.), optimizes the delivery order (to reduce driving time and ensure the food doesn't cool too much) and generates turn-by-turn navigation and delivery instructions for a delivery robot so it can deliver all of its food efficiently. The more deliveries per day that a delivery robot can make and the less energy it uses, the better – so route optimization is key.

Given that the Goober leadership team is composed entirely of UCLA alumni, they've decided to offer the job to build a prototype of the new delivery logistics system to the students of CS32. Lucky you!

So, for your last project for CS32, your goal is to build a simple delivery logistics system that loads and indexes a bunch of Open Street Map geospatial data (which contains latitude and longitude data for thousands of streets) and then use this data to produce delivery plans and navigation instructions for GooberEats delivery robots.

The input to your delivery logistics system consists of:

1. A bunch of Open Street Map data that contains details on thousands of streets in Westwood, and lists the latitude and longitude of each street segment block by block.
2. The latitude and longitude of the central meal prep depot.
3. The latitude and longitude of one or more delivery stops (always on street corners), along with what food item must be delivered at each such street corner.

The output of your delivery logistics system is a set of driving directions for a GooberEats delivery robot to guide it from the central depot, where it picks up the food, to each of its delivery stops, then back to the central depot.

Your completed Project 4 solution should be able to produce relatively optimized delivery instructions like the following:

```
Starting at the depot...
Proceed north on Broxton Avenue for 0.08 miles
Turn left on Le Conte Avenue
Proceed west on Le Conte Avenue for 0.12 miles
Turn right on Levering Avenue
Proceed northwest on Levering Avenue for 0.08 miles
Turn right on Roebling Avenue
Proceed northeast on Roebling Avenue for 0.14 miles
Turn left on Landfair Avenue
Proceed northwest on Landfair Avenue for 0.11 miles
Turn right on Strathmore Drive
Proceed northeast on Strathmore Drive for 0.10 miles
DELIVER Pabst Blue Ribbon beer
Proceed east on Strathmore Place for 0.03 miles
Turn left on Charles E Young Drive West
Proceed north on Charles E Young Drive West for 0.19 miles
Turn left on De Neve Drive
Proceed west on De Neve Drive for 0.06 miles
DELIVER Chicken tenders
Proceed east on De Neve Drive for 0.06 miles
Turn right on Charles E Young Drive West
Proceed south on Charles E Young Drive West for 0.01 miles
Turn left on Bruin Walk
Proceed southeast on Bruin Walk for 0.28 miles
Turn right on Westwood Plaza
Proceed south on Westwood Plaza for 0.16 miles
DELIVER B-Plate salmon
Proceed west on Strathmore Place for 0.23 miles
Turn left on Strathmore Drive
Proceed southwest on Strathmore Drive for 0.10 miles
Turn left on Landfair Avenue
Proceed southeast on Landfair Avenue for 0.11 miles
Turn right on Roebling Avenue
Proceed southwest on Roebling Avenue for 0.14 miles
Turn left on Levering Avenue
Proceed southeast on Levering Avenue for 0.08 miles
Turn left on Le Conte Avenue
Proceed east on Le Conte Avenue for 0.12 miles
Turn right on Broxton Avenue
Proceed south on Broxton Avenue for 0.08 miles
You are back at the depot and your deliveries are done!
2.28 miles travelled for all deliveries.
```

If you're able to prove to Goober Corp's reclusive and bizarre CEO, Carey Nachenberg, that you have the programming skills to build a simple logistics engine, he'll hire you to build the full system, and you'll be rich and famous (at least famous to hungry UCLA undergrads).

Don't worry – it's easier than it seems!

Anatomy of a Delivery Logistics Engine

To build a delivery logistics system, you need to have the following components:

A Searchable Map Index

All delivery logistics systems operate on geolocation data, like the data you can find at Open Street Maps project:

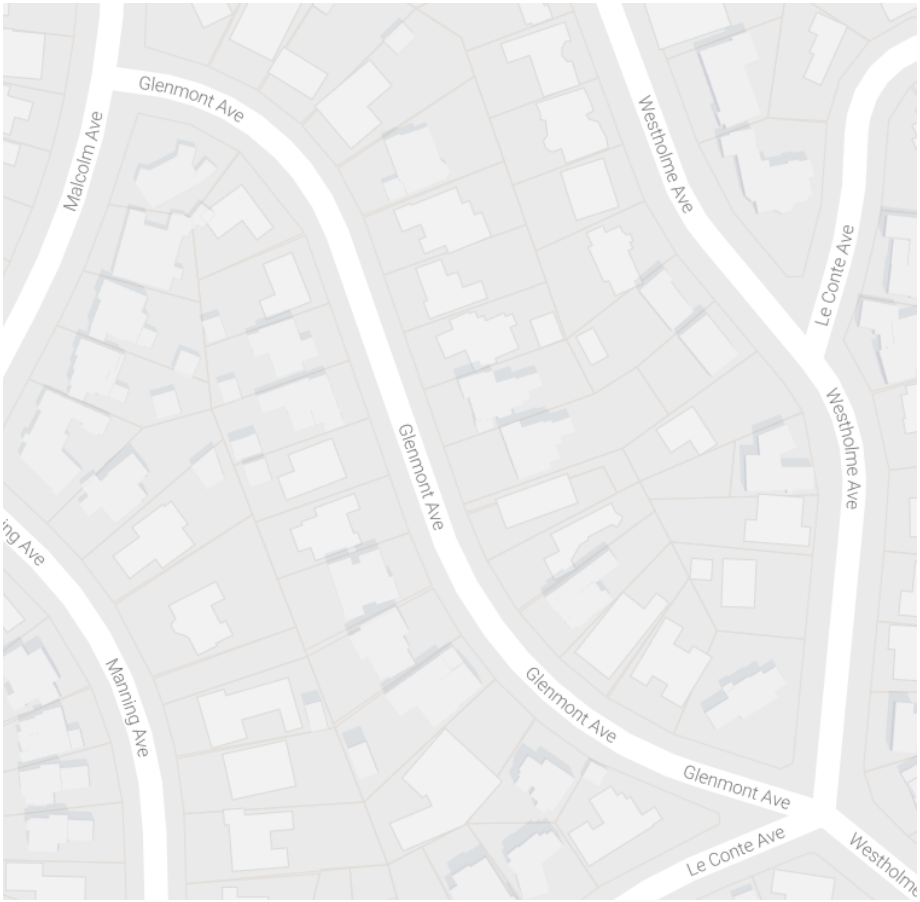
https://www.openstreetmap.org

Open Street Maps (OSM) is an open-source collaborative effort where volunteers can submit street map data to the project (e.g., the geolocations of various streets and businesses), and OSM incorporates this data into its ever-evolving street map database. Companies like Google and TomTom have their own proprietary street map data as well. In this project, we'll be using data from Open Street Maps, because it's freely available. (Just for fun: On a related note, you can get the latitude and longitude for an address at http://www.latlong.net)

The OSM data has geolocation (latitude, longitude) data for each street in its map. Each street is broken up into multiple line segments to capture the contours of the street. As you'll see, even a simple street like Glenmont Ave (which is a short street that is just one block long) may be broken up into many segments. This is done to capture the curvy contours of the street, since each individual segment can only represent a straight line. For example, here are the segments from OSM for Glenmont Avenue in Westwood - each row has the starting and ending latitude/longitude of a street segment that makes up a part of the overall street:

34.0671191,-118.4379955 34.0670930,-118.4377728
34.0670930,-118.4377728 34.0670621,-118.4376356
34.0670621,-118.4376356 34.0669753,-118.4374785
34.0669753,-118.4374785 34.0668906,-118.4373663
34.0668906,-118.4373663 34.0667584,-118.4372616
34.0667584,-118.4372616 34.0660314,-118.4369524
34.0660314,-118.4369524 34.0658228,-118.4368552
34.0658228,-118.4368552 34.0656493,-118.4367430
34.0656493,-118.4367430 34.0654861,-118.4365909
34.0654861,-118.4365909 34.0653477,-118.4363665
34.0653477,-118.4363665 34.0652111,-118.4359814
34.0652111,-118.4359814 34.0651391,-118.4356096

Notice that the ending latitude/longitude of each segment is the same as the starting lat/long of the following segment, resulting in a contiguous street. And here's what Glenmont Ave looks like visually:



Let's consider the first segment in our list, which is highlighted above in red and blue:

34.0671191,-118.4379955 34.0670930,-118.4377728

If you look up 34.0671191,-118.4379955 in Google Maps (just type in these coordinates into the Google Maps search bar), you'll see that this is the location of the intersection of Malcolm Ave and Glenmont Ave (in the upper-left corner of the map). And if you look up 34.0670930, -118.4377728, you'll see that this refers to a spot maybe 100 feet down and right from Malcolm Ave, at the point at which Glenmont Ave. begins to curve just bit. Notice that the second segment for Glenmont Ave:

34.0670930,-118.4377728 34.0670621,-118.4376356

is directly connected to the first segment - the ending latitude/longitude of the first segment (in blue) is exactly the same as the starting latitude/longitude of the second segment (in green). In this manner, OSM can represent a long curvy street by stitching together multiple connecting line segments. (By the way, if you're not familiar with the latitude/longitude system, don't worry about it - for the purposes of this project, just assume that these are x,y points on a 2D grid.)

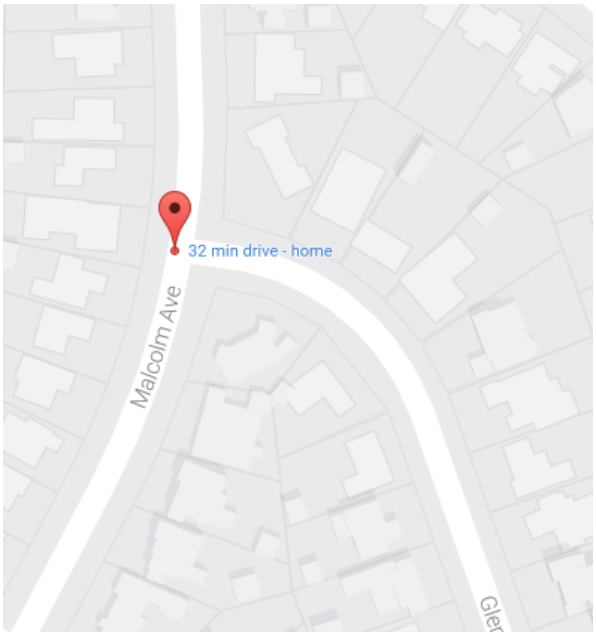
Now let's look at OSM's data for Malcolm Ave, which as we see in the map above intersects with Glenmont Ave. Here are a limited subset of the line segments that make up this much longer street:

...
34.0679593,-118.4379825 34.0676614,-118.4379719
34.0676614,-118.4379719 34.0673693,-118.4379684
34.0673693,-118.4379684 34.0671191,-118.4379955
34.0671191,-118.4379955 34.0668172,-118.4380882
34.0668172,-118.4380882 34.0665572,-118.4382046
34.0665572,-118.4382046 34.0660665,-118.4385079
34.0660665,-118.4385079 34.0654874,-118.4388836
...

You'll notice the highlighted line segment in the middle of the list. This line segment begins at coordinate

34.0671191,-118.4379955

which just happens to be the point of intersection of Glenmont and Malcolm, and was the first lat/long amongst the segments we showed you above for Glenmont Ave. We can thus see that these two streets intersect at this point!



So, you can imagine that given a starting geolocation (e.g., 34.0712323,-118.4505969 – the location of Sproul Landing) and an ending geolocation (e.g., 34.0687443,-118.4449195 – the location of Engineering VI), and given all of the street coordinates for all of the street segments in the OSM database, you should be able to find a contiguous chain of segments from the

starting location to the ending location, and then present this route to the user. Each segment that is part of this route will have its starting latitude/longitude matching the ending latitude/longitude of the previous segment.

To create a Searchable Map Index, you must load this street data into a data structure which permits efficient searching of street segments by latitude and longitude. For example, for this project, you'll need to build an Abstract Data Type/class called *StreetMap* that must:

1. Load all of the OSM map data from a text data file into a data structure of your choosing.
2. Implement a search of that data structure by latitude, longitude which returns all street segments that start at that searched-for location (a street segment includes the street's name, the street segment's starting latitude, longitude, and its ending latitude, longitude).

Point-by-point Routing

A point-by-point routing system leverages the Street Map Index ADT from the section above and a routing algorithm (like A*, or a simple queue-based algorithm) in order to compute a series of street segments which describe a route from a starting location (as described by a latitude, longitude on the map) to an ending location (as described by another latitude, longitude on the map).

For example, to navigate from 34.0625329,-118.4470263 (the intersection of Broxton and Weyburn) to 34.0685657,-118.4489289 (Theta Beta Pi frat at the corner of Gayley and Strathmore), a point-by-point routing system might produce the following segments:

```
34.0625329,-118.4470263  34.0632405,-118.4470467 Broxton Avenue
34.0632405,-118.4470467  34.0636533,-118.4470480 Broxton Avenue
34.0636533,-118.4470480  34.0636457,-118.4475203 Le Conte Avenue
34.0636457,-118.4475203  34.0636344,-118.4482275 Le Conte Avenue
34.0636344,-118.4482275  34.0636214,-118.4491386 Le Conte Avenue
34.0636214,-118.4491386  34.0640279,-118.4495842 Levering Avenue
34.0640279,-118.4495842  34.0644757,-118.4499587 Levering Avenue
34.0644757,-118.4499587  34.0661337,-118.4484725 Roebling Avenue
34.0661337,-118.4484725  34.0664085,-118.4487051 Landfair Avenue
34.0664085,-118.4487051  34.0665813,-118.4488486 Landfair Avenue
34.0665813,-118.4488486  34.0668106,-118.4490930 Landfair Avenue
34.0668106,-118.4490930  34.0670166,-118.4493580 Landfair Avenue
34.0670166,-118.4493580  34.0672971,-118.4497256 Landfair Avenue
34.0672971,-118.4497256  34.0683569,-118.4491847 Strathmore Drive
34.0683569,-118.4491847  34.0684706,-118.4490809 Strathmore Drive
34.0684706,-118.4490809  34.0685657,-118.4489289 Strathmore Drive
```

This could be converted to the following navigation directions through a simple transformation (e.g., measuring the length and compass direction of each segment above, consolidating multiple segments like the two Broxton segments above into one navigation order, measuring the angle between segments to determine if a turn onto a new street is a left or right, etc.):

```
Proceed north on Broxton Avenue for 0.08 miles
Turn left on Le Conte Avenue
Proceed west on Le Conte Avenue for 0.12 miles
Turn right on Levering Avenue
Proceed northwest on Levering Avenue for 0.08 miles
Turn right on Roebling Avenue
Proceed northeast on Roebling Avenue for 0.14 miles
Turn left on Landfair Avenue
Proceed northwest on Landfair Avenue for 0.11 miles
Turn right on Strathmore Drive
Proceed northeast on Strathmore Drive for 0.10 miles
```

For this project, you'll need to build an Abstract Data Type/class called *PointToPointRouter* that will leverage your Street Map ADT (described in the section above) to compute a route of street segments from a starting latitude, longitude to an ending latitude, longitude. The user of this class will request directions from a start point to an end point (e.g., from the latitude, longitude of the food depot to the latitude, longitude of Engineering VI).

Delivery Optimization

When delivering a bunch of items to various locations, to minimize energy usage and travel time you'll want to optimize the ordering of those deliveries. For example, imagine if you were given the following delivery list:

```
Sproul Hall: Deliver spam
Boelter Hall: Deliver chicken tenders
Hedrick Hall: Deliver coffee
Engineering VI: Deliver tacos
Rieber Hall: Deliver broccoli
```

If you simply took these delivery orders and followed them in their current order, you'd end up zig-zagging from Sproul all the way to Boelter Hall, then all the way back to Hedrick Hall, then all the way back to Engineering VI (next to Boelter), then all the way back to Rieber Hall. That would be a good workout, but an inefficient way to deliver food.

A better re-ordering of these deliveries would be:

```
Hedrick Hall: Deliver coffee
Rieber Hall: Deliver broccoli
Sproul Hall: Deliver spam
Engineering VI: Deliver tacos
Boelter Hall: Deliver chicken tenders
```

This delivery ordering would take much less time and energy. It would start at the top of the hill at Hedrick, proceed to Rieber and then to Sproul, finally travelling to South Campus where you'd make the two deliveries at the engineering buildings.

For this project, you'll need to build an Abstract Data Type/class called *RouteOptimizer* that will optimize the order of your deliveries to reduce overall travel distance/time. The user of this class will pass in the location of the food depot and an unordered list of deliveries, and your class must reorder those deliveries in a manner that reduces the total travel distance/time (if possible).

Street Map Data File

We will provide you with a simple data file (called *mapdata.txt)* that contains limited street map data for the Westwood, West Los Angeles, West Hollywood, Brentwood, and Santa Monica areas. This data file has a simplified format and was derived from OSM's more complicated XML-format data files (which are about twice as complicated as a 55 page CS 32 spec). Our *mapdata.txt* file basically has data on thousands of individual street segments, which together make up the entire map. Here's an entry for a particular street called Glenbarr Avenue from the *mapdata.txt* file:

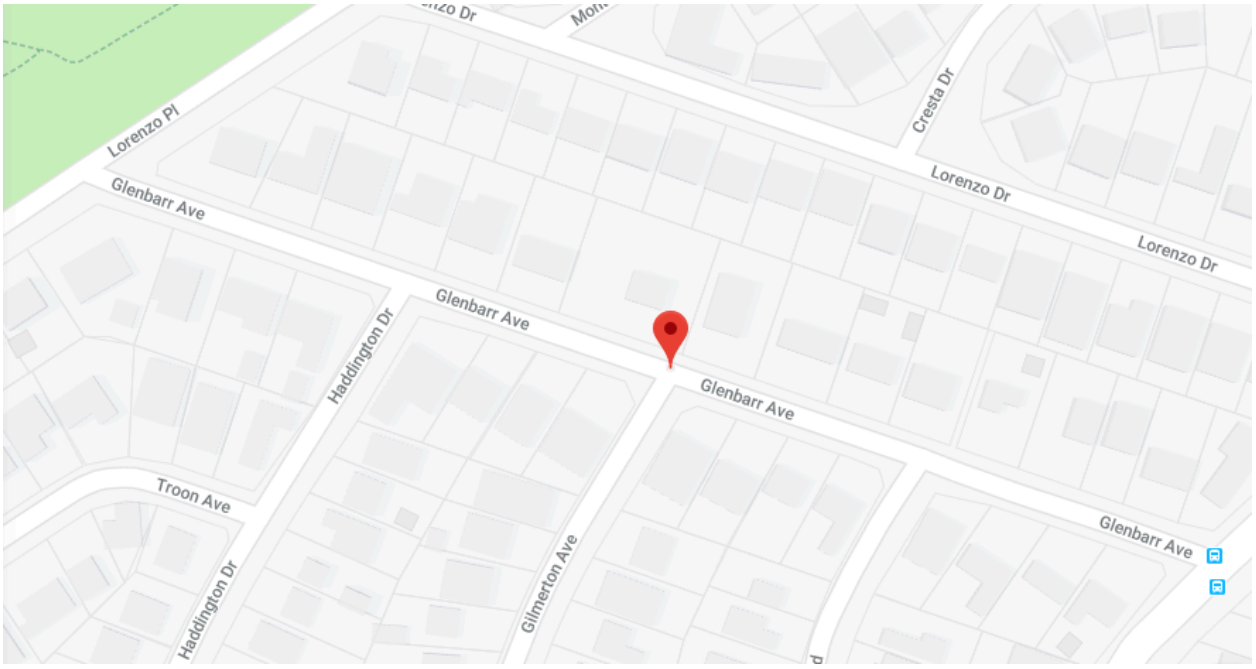
```
Glenbarr Avenue
4
34.0393431 -118.4071546 34.0396737 -118.4082925
34.0396737 -118.4082925 34.0399340 -118.4092305
34.0399340 -118.4092305 34.0402100 -118.4103110
34.0402100 -118.4103110 34.0405666 -118.4114703
```

The first line holds the name of the street we're about to provide geolocation data for. In the example above, we're providing data for Glenbarr Avenue.

The second line specifies a count, C, of how many segments there are for this particular street (in this case, four segments make up the entire street).

The next C lines hold the starting and ending geo-coordinates of the street segments in latitude, longitude format as four numbers separated by whitespace: *start_latitude start_longitude end_latitude end_longitude*.

And here is a Google map showing the four street segments (the first segment is at the lower-right and the last segment ends on the upper-left; each segment in this example starts/ends at an intersection like Gilmerton Ave, Haddington Dr, Lorenzo Pl, etc.):



Here's a slightly longer example from our map data file:

```
...
Glendon Avenue
3
34.0591340  -118.4426546 34.0589680 -118.4424895
34.0589680  -118.4424895 34.0582358 -118.4421816
34.0582358  -118.4421816 34.0572000 -118.4417620
Wellworth Avenue
1
34.0575140, -118.4405712 34.0572000,-118.4417620
…
```

Notice how the first Glendon Avenue segment has an ending geo-coordinate that matches the second Glendon Avenue segment's starting geo-coordinate (34.0589680,-118.4424895). Further, notice that the second Glendon Avenue segment has an ending geo-coordinate that matches the starting geo-coordinate for the third Glendon Avenue segment (34.0582358, -118.4421816). So these three segments are effectively chained together by their start/end coordinates. Now consider the Wellworth Avenue segment. Its ending geo-coordinate (34.0572000,-118.4417620) is the same as the ending geo-coordinate of the third Glendon Avenue segment. So this means that this location defines an intersection between Glendon Avenue and Wellworth Avenue. And in fact, if you look this up in Google maps, this is what you'll see:

So now you know how our mapping data is encoded. And hopefully you're beginning to see that if you have some clever data structures, given any geo-coordinate, you can determine all segments that start or end at that point. You could then follow each such segment to its other end, and figure out what segments it's connected to, and so on.

What Do You Need to Do?

For this project you will create five classes (each will be described below in more detail):

1. You will create a class template *ExpandableHashMap* that works much like the C++ STL *unordered_map* and that implements an open hash table. To ensure that it doesn't exceed a maximum load factor (that is specified as a constructor parameter), an *ExpandableHashMap* must occasionally grow its number of buckets and rehash its items automatically as you add more data to it. This object will hold associations between an arbitrary type of key (e.g., a *GeoCoordinate* containing a latitude, longitude) and an arbitrary type of value (e.g., vector of street segments that start or end on that *GeoCoordinate*).
2. You will create a class *StreetMap* that can be used to look up a geo-coordinate (e.g., a latitude/longitude) and will return all segment(s) that are associated with that coordinate. That is, it will return all segments that either start at the specified geo-coordinate or end at the specified geo-coordinate.
3. You will create a class *PointToPointRouter* that allows the user to specify a starting (latitude, longitude) location, an ending (latitude, longitude) location, and will then return a vector of street segments that describe a path from the starting point to the ending point.

4. You will create a class *DeliveryOptimizer* that optimizes your delivery order (e.g., rearranges the order of deliveries) in an attempt to minimize total travel distance/time for your robots.
5. You will create a class *DeliveryPlanner* which uses all of the above classes to compute optimized, turn-by-turn directions for a delivery robot that takes it from the depot through all of its deliveries and back to the depot.

What Will We Provide?

We Provide a Bunch of Support Source Files

We'll provide you with a header file named *provided.h* which you **must not** modify. It defines:

- A *GeoCoord* struct that you can use to hold a particular latitude/longitude.
- A *StreetSegment* struct that represents a street segment and consists of starting and ending *GeoCoord*s and a street name.
- A *DeliveryRequest* structure that holds a delivery request (e.g., what food to deliver at what latitude, longitude location)
- A *DeliveryResult* enumeration that names the various possible delivery results that the *DeliveryPlanner::generateDeliveryPlan()* method may return (e.g., successfully computed a route, no route found, geolocation was not found, etc.)
- A *DeliveryCommand* class that represents delivery commands that tell a robot what to do (e.g., proceed down a street, turn left or right, drop off a delivery, etc.). The *DeliveryPlanner::generateDeliveryPlan()* method produces a sequence of these *DeliveryCommand*s.
- *StreetMap*, *PointToPointRouter*, *DeliveryOptimizer*, and *DeliveryPlanner* classes. The code you write in other files will implement these classes (details later on).
- A function *distanceEarthMiles* to compute the distance between two *GeoCoords*, a function *angleOfLine* to compute the angle of a *StreetSegment*, and a function *angleBetween2Lines* to compute the angle between two *StreetSegments*. You can use these functions to determine things like what direction the robot should be travelling in (e.g., south, northwest, etc.), the distance to travel on that street (e.g., 0.34 miles), and whether the robot should turn left or right when it reaches a turn.

We'll provide a simple *main.cpp* file that brings your entire program together and lets you test it. You will not turn it in with your solution. Your program **MUST** work correctly with this file, unmodified. (Of course, during development you can modify it or use a completely different main routine, but the program you turn in must work correctly with our main.cpp as provided.)

We also provide you with *mapdata.txt*, a large data file that contains mapping data (the names of each street and the latitudes and longitudes for each street segment) for about 20,000 street segments. During development, you can use a small subset of this file or make up your own small data files for initial testing, and then make sure your program works with this large file.

We Provide a Test Main Program

If you compile your code with our *main.cpp* file, you can use it to test your completed classes. Our *main.cpp* file implements a command-line interface, meaning that if you open a Windows/macOS command shell (e.g., by typing "cmd.exe" in the Windows start box in the bottom-left corner of the screen, or by running the Terminal app in macOS), and switch to the directory that holds your compiled executable file, you can run our test harness code.

From the Windows command line, for example, you can run the test harness as follows:

```
C:\PATH\TO\YOUR\CODE> GooberEats.exe c:\path\to\the\mapdata.txt c:\path\to\your\deliveries.txt
```

You must specify a path to the map data file and a path to delivery list data file that you create which specifies the location of the depot (the first line of the file) and a bunch of lines containing a list of all of the deliveries to make. The delivery list data file must have the following format:

```
Depot_Latitude Depot_Longitude
Delivery1_Latitude Delivery1_Longitude:food item to be delivered
Delivery2_Latitude Delivery2_Longitude:food item to be delivered
…
DeliveryN_Latitude DeliveryN_Longitude:food item to be delivered
```

For example, here's an example delivery data file which specifies that the food depot is at Broxton and Weyburn (34.0625329,-118.4470263) and that Chicken tenders should be delivered to Sproul Landing (34.0712323 -118.4505969), B-Plate salmon should be delivered to Engineering VI (34.0687443 -118.4449195), and Pabst Blue Ribbon beer should be delivered to Beta Theta Pi at the corner of Gayley and Strathmore (34.0685657 -118.4489289):

```
34.0625329 -118.4470263
34.0712323 -118.4505969:Chicken tenders
34.0687443 -118.4449195:B-Plate salmon
34.0685657 -118.4489289:Pabst Blue Ribbon beer
```

Our test program will then run, take the files you passed on the command line and pass them to your classes so they can load the appropriate map data, optimize the route (possibly shuffling the order of your deliveries to save time/energy), compute navigation directions from the depot to the delivery locations and back, and generate a set of delivery commands. The test program will then take the results from your classes and print them to the screen like this:

```
Starting at the depot...
Proceed north on Broxton Avenue for 0.08 miles
Turn left on Le Conte Avenue
Proceed west on Le Conte Avenue for 0.12 miles
Turn right on Levering Avenue
Proceed northwest on Levering Avenue for 0.08 miles
Turn right on Roebling Avenue
Proceed northeast on Roebling Avenue for 0.14 miles
Turn left on Landfair Avenue
Proceed northwest on Landfair Avenue for 0.11 miles
Turn right on Strathmore Drive
Proceed northeast on Strathmore Drive for 0.10 miles
DELIVER Pabst Blue Ribbon beer
Proceed east on Strathmore Place for 0.03 miles
Turn left on Charles E Young Drive West
Proceed north on Charles E Young Drive West for 0.19 miles
Turn left on De Neve Drive
Proceed west on De Neve Drive for 0.06 miles
DELIVER Chicken tenders
Proceed east on De Neve Drive for 0.06 miles
Turn right on Charles E Young Drive West
Proceed south on Charles E Young Drive West for 0.01 miles
Turn left on Bruin Walk
Proceed southeast on Bruin Walk for 0.28 miles
Turn right on Westwood Plaza
Proceed south on Westwood Plaza for 0.16 miles
DELIVER B-Plate salmon
Proceed west on Strathmore Place for 0.23 miles
Turn left on Strathmore Drive
Proceed southwest on Strathmore Drive for 0.10 miles
Turn left on Landfair Avenue
Proceed southeast on Landfair Avenue for 0.11 miles
Turn right on Roebling Avenue
Proceed southwest on Roebling Avenue for 0.14 miles
Turn left on Levering Avenue
Proceed southeast on Levering Avenue for 0.08 miles
Turn left on Le Conte Avenue
Proceed east on Le Conte Avenue for 0.12 miles
Turn right on Broxton Avenue
Proceed south on Broxton Avenue for 0.08 miles
You are back at the depot and your deliveries are done!
2.28 miles travelled for all deliveries.
```

Details: The Classes You Must Write

You must write correct versions of the following classes to obtain full credit on this project. Your classes must work correctly with our provided classes (**you MUST make no modifications to our provided classes, structs, and functions OR YOU WILL GET A ZERO on this project**).

ExpandableHashMap

You must implement a class tempate named *ExpandableHashMap* that, like an STL *unordered_map*, lets a client associate items of a key type with items of a (usually different) value type, with the ability to look up items by key. For example, an *ExpandableHashMap* object associating students' names with their GPAs would have string as the key type and double as the value type. Your implementation **must** use an open hash table and you may use STL containers EXCEPT for *map*, *unordered_map*, *set*, *unordered_set, multimap, unordered_multimap, multiset,* and *unordered_multiset* to implement your class if you like. An empty hash map must start with 8 buckets and have a default maximum load factor of 0.5 if no argument is passed to its constructor.

Here's an example of how you might use *ExpandableHashMap*:

```
void foo()
{
      // Define a hashmap that maps strings to doubles and has a maximum
      // load factor of 0.3. It will initially have 8 buckets when empty.
      ExpandableHashMap<string,double> nameToGPA(0.3);

      // Add new items to the hashmap.  Inserting the third item will cause
      // the hashmap to increase the number of buckets (since the maximum
      // load factor is 0.3), forcing a rehash of all items.
      nameToGPA.associate("Carey", 3.5);  // Carey has a 3.5 GPA
      nameToGPA.associate("David", 3.99); // David beat Carey
      nameToGPA.associate("Abe", 3.2);    // Abe has a 3.2 GPA

      double* davidsGPA = nameToGPA.find("David");
      if (davidsGPA != nullptr)
          *davidsGPA = 1.5;   // after a re-grade of David's exam
            nameToGPA.associate("Carey", 4.0);  // Carey deserves a 4.0
                                                // replaces old 3.5 GPA
      double* lindasGPA = nameToGPA.find("Linda");
      if (lindasGPA == nullptr)
          cout << "Linda is not in the roster!" << endl;
      else
          cout << "Linda's GPA is: " << *lindasGPA << endl;
}
```

Your implementation **must** have the following public interface:

```
template <typename KeyType, typename ValueType>
class ExpandableHashMap
{
public:
       ExpandableHashMap(double maximumLoadFactor = 0.5);     // constructor
       ~ExpandableHashMap();// destructor; deletes all of the items in the hashmap
       void reset();        // resets the hashmap back to 8 buckets, deletes all items
       int size() const;    // return the number of associations in the hashmap

       // The associate method associates one item (key) with another (value).
       // If no association currently exists with that key, this method inserts
       // a new association into the hashmap with that key/value pair.  If there is
       // already an association with that key in the hashmap, then the item
       // associated with that key is replaced by the second parameter (value).
       // Thus, the hashmap must contain no duplicate keys.
       void associate(const KeyType& key, const ValueType& value);

       // If no association exists with the given key, return nullptr; otherwise,
       // return a pointer to the value associated with that key.  This pointer can be
       // used to examine that value, and if the hashmap is allowed to be modified, to
       // modify that value directly within the map (the second overload enables
       // this).  Using a little C++ magic, we have implemented it in terms of the
       // first overload, which you must implement.
       const ValueType* find(const KeyType& key) const;
       ValueType* find(const KeyType& key);
};
```

Requirements for ExpandableHashMap

Here are the requirements for your ExpandableHashMap class:

1. You **must** implement your own expandable hash map in your *ExpandableHashMap* class (i.e., maintain a collection of linked lists of associations, etc.).
2. A newly constructed *ExpandableHashMap* must have 8 buckets and no associations.
3. The *ExpandableHashMap* constructor parameter is the maximum load factor for the hashmap. If the caller does not pass an argument to the constructor or passes a value that is not positive, then maximum load factor must be 0.5.
4. For every type of key that your program will use for an *ExpandableHashMap* you **must** define an overloaded hash function named ***hash()*** (not something else like *myHash* or *hashFunction*). Each such hash functions must accept the key data passed by constant reference as its only parameter and return an *unsigned int*. For example, if you have one

ExpandableHashMap using a *GeoCoord* as a key and another using a *vector<int*>* as a key (we'd be very surprised if you used the latter!):

```
unsigned int hash(const GeoCoord& key)  // yields value in range 0 to ~4 billion
{ … }
unsigned int hash(const vector<int*>& key)  // yields value in range 0 to ~4 billion
{ … }
```

See the next section for details about where to put hash function implementations.

5. Your *ExpandableHashMap* class **must** be a class template enabling a client to map any type of item to any other type of item, e.g., a name (string) to a GPA (double), or a GeoCoord (struct) to a collection of street segments (a vector of *StreetSegment* objects).
6. If, when adding an item to your expandable hash map, its load factor would increase above the maximum load specified during construction, then your *associate()* method must:
   - Create a new internal hash map with double the current number of buckets.
   - Rehash all items from the current hash map into the new, larger hash map.
   - Replace the current hash map with the new hash map.
   - Free the memory associated with the original, smaller hash map.

   Note: This means that after an *associate()* call, all pointers into your hash map returned by the *find()* method prior to the call are potentially invalidated.[1]
7. Your *ExpandableHashMap* class **must** use the public interface shown above. You may add private members to this class, but you must **not** add or change any public members.
8. If a user of your class associates the same key twice (e.g., “David” to 3.99, then “David” to 1.5), the second association must overwrite the first one (i.e., "David" will no longer be associated with 3.99, but will henceforth be associated with 1.5). There **must** be at most one mapping for any unique key.
9. *ExpandableHashMap* objects do not need to be copied or assigned. To prevent incorrect copying and assignment of *ExpandableHashMap* objects, these methods can be declared to be deleted (C++11) or declared private and left unimplemented (pre-C++11).
10. Your member functions **MUST NOT** write anything to *cout*. They may write to *cerr* if you like (to help you with debugging).
11. The Big-O of inserting an item into your hash map must be O(1) in the average case, with the understanding that some insertions may result in an O(N) insertion time should the hash table need to be resized, where N is the number of associations in the table.
12. The Big-O of searching for an item in your hash map must be O(1) in the average case, with the understanding that some searches may result in O(N) steps in a pathological case where there are many collisions.

[1] We're not requiring you to do so, but using std::list's *splice* method (or doing something equivalent if you're implementing a list yourself), you can guarantee that the pointers will never be invalidated by a rehash because the nodes are simply relinked instead of their data being copied elsewhere.

Hash Functions

We are providing you with a hash function for the *GeoCoord* type:

```
unsigned int hash(const GeoCoord& g)
{
      return std::hash<std::string>()(g.latitudeText + g.longitudeText);
}
```

This hash function leverages the STL's *hash* class template, which is declared in the header `<functional>`. You'll need to transform the value returned by the hash function to produce a bucket number that is valid for the number of buckets your map currently has. You may declare hash functions for other types if your implementations of the classes for this project use *ExpandableHashMap*s with those types as keys, but it's unlikely that your design would require such maps.

Now for some requirements imposed by the C++ language rules:

1. You **must** put any hash function implementation in exactly one of your .cpp files, **not** in *ExpandableHashMap.h*. (We've put the one for hashing a GeoCoord in *StreetMap.cpp*; any other .cpp file would have been OK, but it simplifies our grading scripts if for that one, every student's is in the same file.)
2. To avoid getting certain strange compilation or linker errors, inside the implementation of any member function of *ExpandableHashMap* that calls *hash()*, you **must** have a prototype declaration for the hash function. Here's a little hint for this:

   ```
   template<typename KeyType, typename ValueType>
   class ExpandableHashMap
   {
      ...
      unsigned int getBucketNumber(const KeyType& key) const
      {
          ...
          unsigned int hash(const KeyType& k);  // prototype
          unsigned int h = hash(key);
          ...
      }
      ...
   };
   ```

   While some older compilers are forgiving if you don't declare the prototype inside the implementation of a class template member function, Standard C++ has arcane "two-phase lookup" rules for names used in templates that often surprise even experienced C++ programmers.

StreetMap Class

The *StreetMap* class is used to load data from the *mapdata.txt* file we provide and make the data searchable. The *StreetMap* class **must** have the following public interface:

```
class StreetMap
{
public:
      StreetMap();
      ~StreetMap();
      bool load(std::string mapFile);
      bool getSegmentsThatStartWith(const GeoCoord& gc,
                                    std::vector<StreetSegment>& segs) const;
};
```

StreetMap() and possibly ~StreetMap()

The constructor must initialize the *StreetMap* object. You probably won't need to write much code to do this. This spec imposes no requirements on its time complexity.

Should you need to write one to properly dispose of all memory a *StreetMap* object allocates, you must declare and implement a destructor that does so. This spec imposes no requirements on its time complexity.

load()

The job of the *load()* method is to load all of the data from an Open Maps data file and map each *GeoCoord* to all of the street segment(s) that start with that *GeoCoord* so that all segments associated with a geo-coordinate can be quickly located. For each *StreetSegment* S, you **must** add two mappings to your data structure:

1. One that maps the segment's starting GeoCoord (S.start) to the segment S:
   S.start maps to (S.start, S.end, S.name)
2. One that maps the segment's ending GeoCoord (S.end) to the reverse of S:
   S.end maps to (S.end, S.start, S.name)

This ensures that if a GeoCoord G maps to a StreetSegment S, the StreetSegment's starting GeoCoord (S.start) always matches G. For example, you might use an algorithm like this to build the data structure from the data in the Open Maps data file:

```
For each street segment S being processed in the Open Map data file:

   1. Extract the starting GeoCoord (i.e., latitude, longitude) from S and call it B
   2. some_mapping_data_structure[B].add_item(S)
   3. Extract the ending GeoCoord and call it E
   4. R = Reverse(S) // Reverse() swaps the starting/ending GeoCoords in segment S
   5. some_mapping_data_structure[E].add_item(R)
```

For example, given the following street segments from the Open Maps data file, e.g.:

```
Glenmere Way
3
34.0724746 -118.4923463 34.0731003 -118.4931016 // S1
34.0731003 -118.4931016 34.0732851 -118.4931016 // S2
34.0732851 -118.4931016 34.0736122 -118.4927669 // S3
```

Your *StreetMap* object must hold the following mappings:

```
(34.0724746, -118.4923463) ->
      (34.0724746, -118.4923463), (34.0731003, -118.4931016), Glenmere Way // S1

(34.0731003, -118.4931016) ->
      (34.0731003, -118.4931016), (34.0724746, -118.4923463), Glenmere Way // Rev(S1)
      (34.0731003, -118.4931016), (34.0732851, -118.4931016), Glenmere Way // S2

(34.0732851, -118.4931016) ->
      (34.0732851, -118.4931016), (34.0731003, -118.4931016), Glenmere Way // Rev(S2)
      (34.0732851, -118.4931016), (34.0736122, -118.4927669), Glenmere Way // S3

(34.0736122, -118.4927669) ->
      (34.0736122, -118.4927669), (34.0732851, -118.4931016), Glenmere Way // Rev(S3)
```

Your *load()* method must load all of the data from the specified map data file into a container mapping *GeoCoords* to bunches of *StreetSegments* using your *ExpandableHashMap* in some way. You'll need to ensure that you've loaded every street segment from the file, and indexed both its forward and reverse versions as shown in the pseudocode above. The format of the map data file is specified above in the Street Map Data File section. Note: You may use a more efficient data structure than the one we describe if you like, so long as your class behaves as described and leverages your *ExpandableHashMap*. For instance, you could map each *GeoCoord* to a vector of *pointers to StreetSegments*, and then hold a bunch of unique *StreetSegments* in a second data structure.

Your class will open our data file and read the data line by line from this file into your *StreetMap* object (see the File I/O and File input writeups on the class web site). The *load()* method must return true if the data was loaded successfully, and false otherwise (e.g., the file could not be found). You may assume that the data in the map data file is formatted correctly as detailed in this specification, so you don't have to check for errors in its format.

getSegmentsThatStartWith()

A call to *getSegmentsThatStartWith()* must retrieve all of the *StreetSegments* and reversed *StreetSegments* whose *start* geo-coordinate matches the *GeoCoord* parameter *gc*, and place them in the *segs* reference parameter. If there are no such *StreetSegment*s, this function leaves *segs* unchanged and returns false. Otherwise, if the *segs* parameter contained values prior to the call to the *getSegmentsThatStartWith()* method, these values must be discarded and replaced with just the new results. So, for example, assuming the data in the description of *load()* above, a call to this method with a *GeoCoord* parameter of:

```
(34.0732851, -118.4931016)
```

should return the following segments in *segs*:

```
(34.0732851, -118.4931016), (34.0731003, -118.4931016), Glenmere Way // Rev(S2)
(34.0732851, -118.4931016), (34.0736122, -118.4927669), Glenmere Way // S3
```

and a call with a parameter of:

```
(34.0724746, -118.4923463)
```

should return the following segment in *segs*:

```
(34.0724746, -118.4923463), (34.0731003, -118.4931016), Glenmere Way // S1
```

The StreetMap Class vs. the StreetMapImpl Class

We need to be able to test your code to make sure it's correct. To automate testing, we need to require that your classes use the exact interfaces we specify (the same function names, same parameters, same return types, same const requirements). Students often make inadvertent changes to these class intefaces making grading difficult, so we're using a trick to force you to make your classes meet our requirements.

To ensure that you do not change the interface to *StreetMap* in any way, we will implement that class for you! But don't get your hopes up that we're doing any significant work for you here: Our implementation is to simply give *StreetMap* just one private data member variable, a pointer to a *StreetMapImpl* object (which you're actually going to implement in *StreetMap.cpp* and which is going to do all of the real work for the *StreetMap* class - you must write all of the real code for the *StreetMapImpl* class in *StreetMap.cpp*). Our member functions of *StreetMap* simply delegate their work to the functions that you write in *StreetMapImpl*.[2] You are forbidden from changing our *StreetMap* class or source file in any way (you won't even get to submit the .h file for it), so we can control how it works, and you have to make sure your *StreetMapImpl*

[2] This is an example of what is called the pimpl idiom (from "pointer-to-implementation").

code can be called by it and compiles with it. You still have to do all of the work of implementing the *StreetMapImpl* functions.

Other than *StreetMap.cpp*, no source file that you turn in may contain the name *StreetMapImpl*. Thus, your other classes must not directly instantiate or even mention *StreetMapImpl* in their code. They may use the *StreetMap* class that we provide (which indirectly uses your *StreetMapImpl* class). This forces those other files to use the required interface for *StreetMap* and makes it easier for us to test your code.

Requirements for StreetMapImpl

Here are the requirements for your *StreetMapImpl* class that implements the *StreetMap* functionality:

1. It **must** adhere to the specification above.
2. It must **not** access any other *Impl* classes that you write.
3. It must **not** use any STL associative containers (i.e., *map*, *multimap*, *set*, *multiset*, or the unordered_ versions of those classes). It **may** use the STL *vector*, *list*, *stack*, *queue*, and *priority_queue* classes if you wish.
4. It must **not** write anything to *cout*. It may write to *cerr* if you wish (to help you with debugging).
5. If there are N lines in the input mapping data file, then *load()* must run in O(N) time, and getSegmentsThatStartWith() must run in O(1) time.

PointToPointRouter Class

The *PointToPointRouter* class is responsible for computing an efficient route from a source geo coordinate to a destination geo coordinate, if one exists. It must use the *StreetMap* class to obtain street information for this routing.

```
class PointToPointRouter
{
public:
      PointToPointRouter(const StreetMap* sm);
      ~PointToPointRouter();
      DeliveryResult generatePointToPointRoute(
            const GeoCoord& start,
            const GeoCoord& end,
            std::list<StreetSegment>& route,
            double& totalDistanceTravelled) const;
};
```

PointToPointRouter() and possibly ~PointToPointRouter()

Your *PointToPointRouter* constructor must accept a pointer to a fully-constructed *StreetMap* object containing loaded street map data. This spec imposes no requirements on its time complexity. Notice that this constructor doesn't take a pointer to a *StreetMapImpl* object as a parameter, but to a *StreetMap* object. This way, if you have bugs in your *StreetMapImpl* class, we can substitute our correct version of the class for yours within *StreetMap* and then test your *PointToPointRouter* class with our correct version. This will help you get points for one class even if another class it relies upon has bugs.

Should you need to write one to properly dispose of all memory a *PointToPointRouter* object allocates, you must declare and implement a destructor that does so. This spec imposes no requirements on its time complexity.

generatePointToPointRoute()

After constructing a *PointToPointRouter* object, its user may call *generatePointToPointRoute()* one or more times to compute a list of street segments that lead from a *start GeoCoord* to an `end` *GeoCoord.* The returned *route* may contain both *StreetSegments* directly found within the Open Maps data file, as well as reversed *StreetSegments* synthetically generated by your *StreetMap* class. Assuming a valid set of connecting *StreetSegments* can be found between the two coordinates, the first *StreetSegment* returned via the output *route* parameter must have a starting *GeoCoord* that matches the passed-in `start` parameter, and the last *StreetSegment* returned must have an ending *GeoCoord* that matches your `end` parameter. If the `start` and `end` parameters are the same (e.g., you're navigating to where you already are standing), then the *route* parameter must be cleared/empty and the *totalDistanceTravelled* result must be set to 0, since a route from a position to itself requires no travel. Otherwise, your must set *totalDistanceTravelled* to the total length, in **miles**, of all street segments traversed to travel from the source to the destination locations.

Your *generatePointToPointRoute()* method must return one of the following codes, depending on the result:

- **DELIVERY_SUCCESS**: A path was found from the source to the destination.
- **NO_ROUTE**: No route was found linking the source to the destination address.
- **BAD_COORD**: The source or destination geo coordinate was not found within the mapping data.

If a route cannot be found or an invalid start/end coordinate is passed in, then your method may leave the *route* and *totalDistanceTravelled* parameters in any state you like (e.g., modify them, not modify them, etc.).

As with the other classes you must write, the real work will be implementing the auxiliary class *PointToPointRouterImpl* in *PointToPointRouter.cpp*. **Other than *PointToPointRouter.cpp*, no source file that you turn in may contain the name *PointToPointRouterImpl*.** Thus, your other classes must not directly instantiate or even mention *PointToPointRouterImpl* in their code. They may use the *PointToPointRouter* class that we provide (which indirectly uses your *PointToPointRouterImpl* class).

Requirements for PointToPointRouterImpl

Here are the requirements for your *PointToPointRouterImpl* class:

1. It **must** adhere to the specification above.
2. It must not directly access any other *Impl* classes that you write.
3. You **may** use any and all STL classes you like.
4. It must **not** write anything to *cout*. It may write to *cerr* if you wish (to help you with debugging).
5. Assuming there are N segments in our mapping data, your *generatePointToPointRoute*() method must run in O(N) time. However, if you implement your route-finding algorithm efficiently, it should generally run in far less time.

How to Implement a Route-finding Algorithm

Right now you're probably thinking: "It's got to be rocket science to compute an optimal route between two coordinates on a map… only a company as awesome as Google could do that :)". But in fact, nothing could be further from the truth! It's possible to implement an optimal routing algorithm in just a few hundred lines of code (or less!) using a simple STL *queue* and STL *set* or even better, an algorithm known as A*. You're welcome to use A* if you like, but if you're a little intimidated by this, why not use an algorithm like the queue-based maze searching algorithm you implemented in your homework?

Of course, there are a few differences between maze-searching and geo-navigation:

1. In your original queue-based maze searching algorithm, you enqueued integer-valued x,y coordinates, whereas in this project you're enqueuing real-valued latitude, longitude coordinates.
2. In your original maze searching algorithm, you "dropped breadcrumbs" in your maze array to prevent your algorithm from visiting the same square more than once, whereas in this project, there's no 2D array that you can use to track whether you've visited a square, so you'll have to figure out some other way to prevent your algorithm from re-visiting the same coordinates over and over.
3. In the original maze-searching algorithm, you could determine adjacent squares of the maze to explore through simple arithmetic - if you were at position (x,y) of the maze, you knew that the adjacent maze locations were (x-1,y), (x+1,y), (x,y-1), and (x,y+1). In this

project, you're going to have to leverage the *StreetMap* class to locate adjacent coordinates.

4. In the original maze-searching algorithm, you just had to determine if the maze was solvable and return a Boolean result (true or false). But for this project, you'll have to actually return back a full list of segments to provide segment-by-segment directions.

But, with just a few changes, you should be able to adapt your queue-based maze searching algorithm to one that does street navigation! Of course, you're still going to have to figure out how to return the whole route (segment-by-segment directions) back to the user. In your maze searching homework, you simply returned *true* or *false* indicating whether a path through the maze existed.., you didn't return the actual steps to get through the maze! To get credit for the *PointToPointRouterImpl* class, you have to return each segment that makes up that route!

How might you track the complete segment-by-segment route so you can return it back to the user? Well, one way to do so would be to maintain a map (using your templated *ExpandableHashMap* class) that associates a given geo-coordinate G to the previous geo-coordinate P in the route (e.g., we travelled **to** G directly **from** P). Let's call this map: *locationOfPreviousWayPoint*

Let's illustrate the approach with an example containing 3 segments: Segment A-B, Segment B-C, Segment C-D

Let's assume that your queue-based search algorithm is searching for a path from A to D. The algorithm could look up point A using the *StreetMap* class, and find the segment (A, B). At this point, the algorithm would queue geo coordinate B to be processed later, and add the following an association to your map:

locationOfPreviousWayPoint[B] -> A

The first association indicates that we reached location B directly from location A.

Next, the algorithm might dequeue point B, and then use the *StreetMap* class to determine that it can reach geo-coordinate C from B via segment (B,C). It would add C to its queue for later exploration. And, again, it would add the fact that it reached C from B to its waypoint map:

locationOfPreviousWayPoint[C] -> B

A bit later the algorithm would dequeue geo-coordinate C from the queue. Again, using the *StreetMap* class, it would determine that point C leads to point D, on *GeoSegment* (C,D). Again, it could add this fact to its map:

locationOfPreviousWayPoint[D] -> C

So every time we reach a new waypoint (e.g., B or C or D), the algorithm could add an entry to the *locationOfPreviousWayPoint* map that maps that waypoint to the geo-coordinate that we traveled *from* to get to that waypoint.

When we finally reach point D, our map might contain:

```
locationOfPreviousWayPoint[B] -> A
locationOfPreviousWayPoint[C] -> B
locationOfPreviousWayPoint[D] -> C
...
```

So how can we use this map to reconstruct our route from A to D? Well, starting from our destination point - location D - we can look up D in the map to determine how we got there (from C). This tells us that our last segment in our navigation was (C,D). We can then lookup point C in our map and determine that we got there from point B. This tells us that the next to last segment was (B,C). And so on. Eventually we'll reach point A, our starting point, allowing us to complete the first segment (A,B), and we'll have re-created the complete route. Each of these discovered segments can be added to an STL *list* and then returned to the user.

Since, like the maze searching algorithm, your routing algorithm must only visit each geo-coordinate once (otherwise it would potentially go in circles), you're guaranteed to have a single entry for each point in your *locationOfPreviousWayPoint* map for each geo-coordinate, enabling you to easily reconstruct the route. There should never be a case where the map has to associate a given waypoint with more than one previous coordinate. Cool, huh?

So, intuitively, what does your *overall* navigation algorithm do? Well, it basically moves out from the starting coordinate in concentric growing rings. It starts by locating all *GeoSegments* that start (or end) with our starting coordinate and adds their ending coordinates to our queue. It then finds all segments associated with these *GeoCoords* and adds the other end coordinates of their *GeoSegments* to our queue. And so on, and so on. Eventually, either the algorithm stumbles upon the ending coordinate, or it will work our way through the entire street map, completely empty out our queue, and realize that the destination can't be reached. If and when the algorithm finds the ending coordinate, it can then use the locationOfPreviousWayPoint map to trace a path back from the ending location to the starting location, following each geolocation it traversed back to the one just before it, and to the one before it, all the way to the start coordinate.

Now if you think hard, you'll realize that this algorithm won't necessarily find the shortest path (in miles) from your source coordinate to the destination coordinate. It will, however, find the path with the fewest number of segments between your source to your destination and return it to you. But the path with the smallest number of segments won't necessarily result in the shortest/fastest path. Consider a case where there are three points: A, B and C all on a straight line. A is at position 0, B is at position 10 miles, and C is at position 100 miles. There are two sets of roads that can take us from point A to point B:

- A curvy road that has 20 segments that proceed directly from A to B, with a total distance of 10 miles.
- A straight road that has 1 segment that proceeds from A to C (100 miles), and then a second straight segment that proceeds from C back to B (90 miles).

Our naive queue-based algorithm would end up selecting the second option, even though it's far less efficient, because it requires fewer hops/segments to reach the endpoint. This is suboptimal.

There are various ways to make your algorithm find a better/faster path, for instance using the A* algorithm. Or, you could do something really simple… For example, imagine that your algorithm is currently searching for a route and is at geo-coordinate X. Further, let's assume that X is connected to three outgoing segments (X,Y), (X,Z), and (X,Q), and that locations Y, Z and Q have not yet been visited.

Rather than just enqueuing Y, Z and Q into our queue in some arbitrary order, we could rank-order those three coordinates by their distance to our ultimate destination, and then insert each item into our queue in order of its increasing distance from the destination coordinate. So if location Z is .1 miles away from our destination, location Y is 2.5 miles, and location Q is .6 miles, we might enqueue location Z first, Q second and Y third. This "heuristic" will increase (though not guarantee) the likelihood that we'll find the shortest path first. Use your creativity to improve upon the basic queue-based navigation algorithm, or look up A* - it's actually pretty simple to implement and will give an optimal result.

DeliveryOptimizer Class

The *DeliveryOptimizer* class is responsible for taking an input set of delivery coordinates and ordering them in the most efficient order to speed up the overall delivery time and reduce energy usage by Goober's robots. It may use the *StreetMap* class to do so, but this is not required.

IF YOU ARE HAVING TROUBLE FINISHING THIS PROJECT, THEN DO THIS CLASS LAST. YOU CAN GET MOST OF THE CREDIT FOR THIS PROJECT BY HAVING THIS CLASS DO VIRTUALLY NOTHING AT ALL.

As mentioned in the introduction, it often makes sense to re-order your delivery coordinates to reduce the overall travel distance of your robots. For example, imagine if you were a delivery person and you were given 6 packages to deliver – 3 to locations in LA and 3 to locations in NYC. It would be extremely expensive for you to deliver a package to a location in LA, then fly to NYC to deliver the next package, then fly back to LA to deliver the next package, then back to NYC for the next package, and so forth. Instead, you'd want to reorder your deliveries so all of

your LA deliveries occur first, followed by a single flight to NYC, followed by all of your NYC deliveries, followed by one final flight back to LA.

In a similar fashion, you'll want to optimize the order of your deliveries for your GooberEats project, since the initial list of deliveries provided to you may not be in the most efficient order.

Your *DeliveryOptimizer* class has the following interface:

```
class DeliveryOptimizer
{
public:
      DeliveryOptimizer(const StreetMap* sm);
      ~DeliveryOptimizer();
      void optimizeDeliveryOrder(
            const GeoCoord& depot,
            std::vector<DeliveryRequest>& deliveries,
            double& oldCrowDistance,
            double& newCrowDistance) const;
};
```

`DeliveryOptimizer()` and possibly `~DeliveryOptimizer()`

Your *DeliveryOptimizer* constructor takes a pointer to a fully-constructed and loaded *StreetMap* object, which it **may optionally use** to perform route optimization. This spec imposes no requirements on its time complexity. Hint: It is possible to perform optimization without this *StreetMap* data!

Should you need to write one to properly dispose of all memory a *DeliveryOptimizer* object allocates, you must declare and implement a destructor that does so. This spec imposes no requirements on its time complexity.

optimizeDeliveryOrder()

Your *optimizeDeliveryOrder()* method accepts the coordinates of the GooberEats food depot, and a vector of delivery requests (each consists of a *GeoCoord* indicating an intersection where to drop off the food and the name of the food item to deliver) as input:

```
      struct DeliveryRequest
      {
            std::string item;
            GeoCoord location;
      };
```

Your method must:

1. Determine the old "crow's distance", in **miles**, from the depot, to each of the successive delivery coordinates in their initial order, and then back to the depot. This is the distance that a crow would fly if it had to go directly from each point to the next along the shortest flight path (ignoring streets below) starting and ending at the depot. You can use the function
   ```
   double distanceEarthMiles(const GeoCoord& g1, const GeoCoord& g2)
   ```
   in *provided.h* to compute the distance between each pair of geo-coordinates, and thus the overall travel distance.
2. Attempt to optimize the delivery order in some way (how is up to you), placing the (possibly) re-ordered deliveries back into the *deliveries* vector, to reduce the overall travel distance. For instance, if you had a bunch of deliveries that were geographically close to each other, you could place those close to each other in your vector[3].
3. After (optionally) re-ordering the delivery locations to optimize for travel distance, compute the new crow's distance, in **miles**, for your newly-proposed delivery ordering.

If you are slammed for time, then this method may do nothing at all to re-order the deliveries, and just compute the starting and ending crow distances (which would be the same).

As with the other classes you must write, the real work will be implementing the auxiliary class *DeliveryOptimizerImpl* in *DeliveryOptimizer.cpp*. **Other than *DeliveryOptimizer.cpp*, no source file that you turn in may contain the name *DeliveryOptimizerImpl*.** Thus, your other classes must not directly instantiate or even mention *DeliveryOptimizerImpl* in their code. They may use the *DeliveryOptimizer* class that we provide (which indirectly uses your *DeliveryOptimizerImpl* class).

Requirements for *DeliveryOptimizerImpl*

Here are the requirements for your *DeliveryOptimizerImpl* class:

1. It **must** adhere to the specification above.

[3] If it's not obvious, this is a VERY difficult problem called the "Traveling Salesman Problem." To compute the optimal solution to this problem, which results in an optimal ordering of deliveries resulting in the shortest route possible, would require O(N!) computations which for N=100 deliveries would take 9.332622e+157 iterations according to today's best-known algorithm!!!! However, there are "heuristic" algorithms you can use to compute pretty good, if not optimal, solutions. Try to be creative and come up with your own approach – it doesn't have to be perfect to get full credit, obviously! If you're interested in a really cool algorithm that uses randomness to obtain good solutions, look into an approach called Simulated Annealing (Carey and David can provide more details if you're interested).

2. It must not directly access any other *Impl* classes that you write.
3. You **may** use any and all STL classes you like.
4. It must **not** write anything to *cout*. It may write to *cerr* if you wish (to help you with debugging).
5. Assuming there are N delivery locations passed to your *optimizeDeliveryOrder()* method (N <= 100), your *optimizeDeliveryOrder()* method must run in $O(N^4)$ time. However, if you implement your algorithm efficiently, it should generally run in far less time.

DeliveryPlanner Class

The *DeliveryPlanner* class is responsible for taking as input the location of the depot and a set of delivery requests (each request specifying an *GeoCoord* guaranteed to be in the *mapdata.txt* file, and item to deliver, like “Sardines”) and producing an efficient set of turn-by-turn directions for the robot to navigate from the depot to the various drop-off points to drop off the food. This class brings all of your other classes together to solve the overall problem of delivery logistics.

```
class DeliveryPlanner
{
public:

      DeliveryPlanner(const StreetMap* sm);
      ~DeliveryPlanner();
      DeliveryResult generateDeliveryPlan(
            const GeoCoord& depot,
            const std::vector<DeliveryRequest>& deliveries,
            std::vector<DeliveryCommand>& commands,
            double& totalDistanceTravelled) const;
};
```

DeliveryPlanner() and possibly ~DeliveryPlanner()

Your constructor takes a pointer to a fully-constructed and loaded *StreetMap* object, which it **may optionally use** to perform route optimization. This spec imposes no requirements on its time complexity.

Should you need to write one to properly dispose of all memory a *DeliveryPlanner* object allocates, you must declare and implement a destructor that does so. This spec imposes no requirements on its time complexity.

generateDeliveryPlan()

This method must take in the depot location (a *GeoCoord*) and a vector of delivery requests (shown in the description of DeliveryOptimizer::optimizeDeliveryOrder above) and produce a vector of *DeliveryCommands* (see *provided.h*). If no route is possible or an invalid depot or delivery request location is passed in, then your method may leave the *commands* and *totalDistanceTravelled* parameters in any state you like. A *DeliveryCommand* looks like this:

```
class DeliveryCommand
{
public:
        // Make this DeliveryCommand a Proceed command
      void initAsProceedCommand(std::string direction,
                                std::string streetName, double distance);
        // Make this DeliveryCommand a Turn command
      void initAsTurnCommand(std::string turnDirection,
                             std::string streetName);
        // Make this DeliveryCommand a Deliver command
      void initAsDeliveryCommand(std::string item);
      ... // other important functions to look up when you start the project
};
```

As you can see, we can have several different types of delivery commands that can be created:

- **Proceed on a street**: Proceed forward on a street of a given name for a given distance (in miles)
- **Turn on a street**: Turn left or right on a street of a given name
- **Deliver an item**: Deliver a given item at the current spot

To generate these *DeliveryCommand*s, you may use something like the following pseudocode:

- First, reorder the order of the delivery requests (using the *DeliveryOptimizer* class) to optimize/reduce the total travel distance.
- Then generate point-to-point routes between the depot to each of the successive (optimized/reordered) delivery points, then back-to the depot after all the deliveries have been made (using the *PointToPointRouter* class).
- For each sequence of point-to-point *StreetSegment*s generated by *PointToPointRouter* in the previous step*, g*enerate a sequence of *DeliveryCommands* representing instructions to the delivery robot. This involves:
    - Converting the sequence of *StreetSegments* produced by the *PointToPointRouter* class (e.g., from the depot to the first delivery coordinate, or from the N^{th} to the $N+1^{st}$ delivery coordinate, or from the last delivery coordinate back to the depot) into one or more ***proceed*** or ***turn*** *DeliveryCommands.*

    - After generating the ***proceed*** and ***turn*** *DeliveryCommands* to get to the robot to the next delivery location, generate a ***deliver*** *DeliveryCommand* indicating that a food item should be delivered at that location.
- Compute the total travel distance over all of the street segments.
- Return the delivery commands and total travel distance to the user.

Upon completion, a GooberEats robot should be able to use the delivery commands produced to deliver its food items.

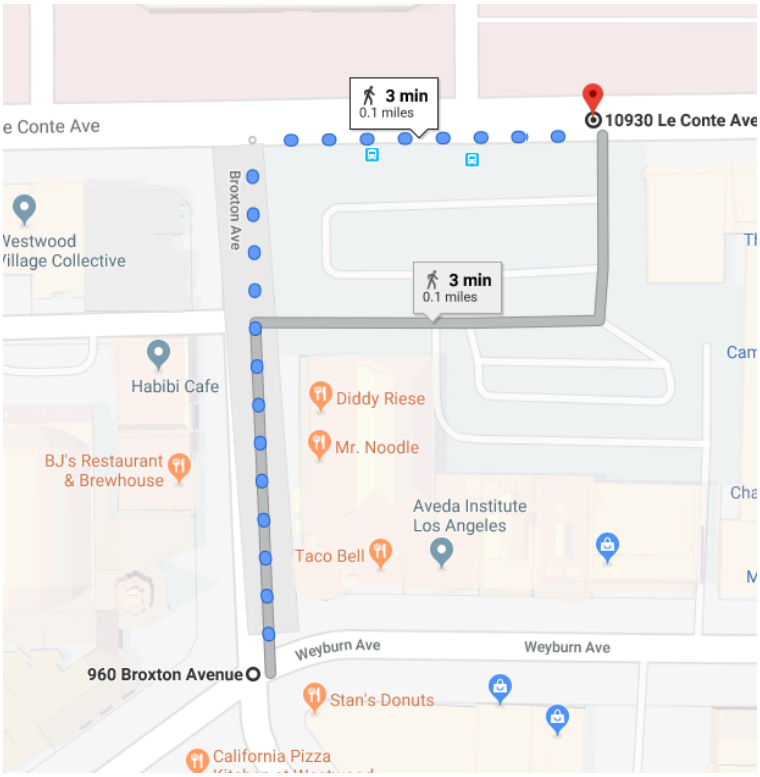
The sequence of delivery commands produced must meet the following requirements:

- No two successive ***proceed*** *DeliveryCommand*s may contain the same street name. If two or more adjacent proceed commands would have the same street name, they must be merged into a single ***proceed*** *DeliveryCommand* with the same street name and the combined travel distance.
- When generating a ***proceed*** command, set the direction based on the angle of direction of the *first street segment* that makes up the ***proceed*** command:
    - 0 <= angle < 22.5: east
    - 22.5 <= angle < 67.5: northeast
    - 67.5 <= angle < 112.5: north
    - 112.5 <= angle < 157.5: northwest
    - 157.5 <= angle < 202.5: west
    - 202.5 <= angle < 247.5: southwest
    - 247.5 <= angle < 292.5: south
    - 292.5 <= angle < 337.5: southeast
    - angle >= 337.5: east
- When generating a ***turn*** *DeliveryCommand,* if the angle between the two differently named streets is:
    - < 1 degrees or > 359 degrees: Do not generate a ***turn*** command. Instead, just generate a ***proceed*** command for the second street since there is no real turn occurring from the first street.
    - >= 1 degrees and < 180 degrees: Generate a ***left*** turn command.
    - >= 180 degrees and <= 359 degrees: Generate a ***right*** turn command.
- Never generate a ***turn*** delivery command as your first command when leaving the depot – assume that when you leave the depot you are always facing in the direction you need to go on the first street you intend to travel on, and therefore don't need turn left or right to start travelling on it. Therefore, always generate a ***proceed*** command as the first *DeliveryCommand* when leaving the depot (unless the delivery of the food item is *at the depot,* in which case you can generate a ***delivery*** command as your first command).
- After emitting a ***delivery*** command, never emit a ***turn*** command right afterward. Assume that when you finish dropping off an item, you are always facing in the direction you need to go on the next street you intend to travel on, and therefore don't need turn left or right to start travelling on it.

To illustrate the above with an example, let's say that the following delivery instructions are provided to your *DeliveryPlanner* class via a *deliveries.txt* file:

```
34.0625329 -118.4470263
34.0636671 -118.4461842:Sardines
```

This indicates that the food depot is roughly at 960 Broxton Ave (34.0625329, -118.4470263), and the delivery location for a bunch of Sardines is roughly at 10930 Le Conte Avenue (34.0636671, -118.4461842):



Furthermore, here are excerpts from your mapdata.txt data file:

```
Broxton Avenue
8
34.0636533 -118.4470480 34.0632405 -118.4470467
34.0632405 -118.4470467 34.0625329 -118.4470263
… // 6 more street segments that make up Broxton Ave
…
Le Conte Avenue
54
… // a few dozen more street segments
34.0636671 -118.4461842 34.0636625 -118.4464708
34.0636625 -118.4464708 34.0636533 -118.4470480
… // a few dozen more street segments
```

In the above example, your *PointToPointRouter* class would likely output the following four *StreetSegments* which proceed from the depot (34.0625329, -118.4470263) north along the two segments that make up Broxton Ave[4] and then east two street segments along Le Conte Ave – this gets the robot from the depot to its first drop-off point:

```
34.0625329 -118.4470263 34.0632405 -118.4470467 Broxton Ave
34.0632405 -118.4470467 34.0636533 -118.4470480 Broxton Ave
34.0636533 -118.4470480 34.0636625 -118.4464708 Le Conte Ave
34.0636625 -118.4464708 34.0636671 -118.4461842 Le Conte Ave
```

These would be converted into the following *DeliveryCommands:*

- One ***proceed*** command: Proceed ***north*** on Broxton Avenue for 0.08 miles (this combines the first two *StreetSegments* above)
- One ***turn*** command: Turn ***right*** on Le Conte Avenue
- One ***proceed*** command: Proceed ***east*** on Le Conte Avenue for 0.05 miles (this combines the second two *StreetSegments* above)
- One ***delivery*** command: Deliver Sardines

After the delivery, your planner would then need to route to the next stop, which in this case would be back to the depot. The *PointToPointRouter* would likely output the following four *StreetSegments* which return it back to the depot:

```
34.0636671 -118.4461842 34.0636625 -118.4464708 Le Conte Ave
34.0636625 -118.4464708 34.0636533 -118.4470480 Le Conte Ave
34.0636533 -118.4470480 34.0632405 -118.4470467 Broxton Ave
34.0632405 -118.4470467 34.0625329 -118.4470263 Broxton Ave
```

These would be converted into the following *DeliveryCommands:*

- One ***proceed*** command: Proceed ***west*** on Le Conte Avenue for 0.05 miles (this combines the first two *StreetSegments* above)
- One ***turn*** command: Turn ***left*** on Broxton Avenue
- One ***proceed*** command: Proceed ***south*** on Broxton Avenue for 0.08 miles (this combines the second two *StreetSegments* above)

So, as you can see, while your *PointToPointRouter* class outputs lists of street segments, your *DeliveryPlanner* class is responsible for converting/consolidating those results into *DeliveryCommands* that instruct the robot to navigate and drop things off.

The complete list of commands would therefore look like this:

---

[4] Note that the first two segments travelling along Broxton have their starting and ending coordinates reversed from the ones shown in the mapdata.txt data file, since we're traveling in the opposite direction of the segments from the data file.

- One ***proceed*** command: Proceed ***north*** on Broxton Avenue for 0.08 miles
- One ***turn*** command: Turn ***right*** on Le Conte Avenue
- One ***proceed*** command: Proceed ***east*** on Le Conte Avenue for 0.05 miles
- One ***delivery*** command: Deliver Sardines
  One ***proceed*** command: Proceed ***west*** on Le Conte Avenue for 0.05 miles
- One ***turn*** command: Turn ***left*** on Broxton Avenue
- One ***proceed*** command: Proceed ***south*** on Broxton Avenue for 0.08 miles

Requirements for *DeliveryPlannerImpl*

Here are the requirements for your *DeliveryPlannerImpl* class:

1. It **must** adhere to the specification above.
2. It **must not** directly access any other *Impl* classes that you write.
3. You **may** use any STL classes you like.
4. It must **not** write anything to *cout*. It may write to *cerr* if you wish (to help you with debugging).
5. Since it uses your other classes, it must have a Big-O equal to the sum of the Big-Os for those classes.

Project Requirements and Other Thoughts

Make sure to read this entire section before beginning your project!

1. In Visual C++, make sure to change your project from UNICODE to Multi Byte Character set, by going to Project / Properties / Configuration Properties / General / Character Set
2. The entire project can be completed in under 600 lines of C++ code beyond what we've already written for you, so if your program is getting much larger than this, talk to a TA – you're probably doing something wrong.
3. Before you write a line of code for a class, think through what data structures and algorithms you'll need to solve the problem. How will you use these data structures? Plan before you program!
4. Don't make your program overly complex – use the simplest data structures possible that meet the requirements.
5. You must not modify any of the code in the files we provide you that you will not turn in; since you're not turning them in, we will not see those changes. We will incorporate the required files that you turn in into a project with special test versions of the other files.
6. Your Impl classes (e.g., *PointToPointRouterImpl*, *DeliveryRouterImpl*, etc.) must **never *directly*** use your other Impl classes. They MUST use our provided wrapper classes instead:

```
INCORRECT:
      class DeliveryPlannerImpl
      {  ...
         PointToPointRouterImpl m_p2pRouter;  // BAD!
         ...
      };

CORRECT:
      class DeliveryPlannerImpl
      {  ...
         PointToPointRouter m_p2pRouter;  // GOOD!
         ...
      };
```

7. Make sure to implement and test each class independently of the others that depend on it. Once you get the simplest class coded, get it to compile and test it with a number of different unit tests. Only once you have your first class working should you advance to the next class.
8. You may use only those STL containers (e.g., vector, list) that are explicitly permitted by this spec. Use the *ExpandableHashMap* class if you need a map, for example; do **not** use the STL *map* or *unordered_map* class.
9. Let *Whatever* represent *StreetMap*, *PointToPointRouter*, and *DeliveryOptimizer*. Subject to the constraints we imposed (e.g., no changes to the public interface of the *Whatever* class, no mention of *WhateverImpl* in any file other than *Whatever*.cpp, no use of certain STL containers in your implementation), you're otherwise pretty much free to do whatever you want in *Whatever*.cpp as long as it's related to the support of only the *Whatever* implementation; for example, you may add members (even public ones) to the *Whatever*Impl class (but not the *Whatever* class, of course) and you may add non-member support functions (e.g., a custom comparison function for *sort()*).

If you don't think you'll be able to finish this project, then take some shortcuts. For example, if you can't get your *ExpandableHashMap* class working, use the substitute *ExpandableHashMap* class we provide so that you can proceed with implementing other classes, and go back to fixing your *ExpandableHashMap* class later.

You can still get a good amount of partial credit if you implement most of the project. Why? Because if you fail to complete a class (e.g., *StreetMapImpl*), we will provide a correct version of that class and test it with the rest of your program (by changing our *StreetMap* class to use our correct, finished version of the class instead of your version). If you implemented the rest of the program properly, it should work perfectly with our version of the *StreetMapImpl* class and we can give you credit for those parts of the project you completed (This is why we're using *Impl* classes and non-*Impl* classes).

But whatever you do, make sure that ALL CODE THAT YOU TURN IN BUILDS without errors under both g32 and either Visual Studio or clang++!

What to Turn In

You must turn in **six to eight** files. These six are required:

ExpandableHashMap.h	Contains your hash map class template implementation
StreetMap. cpp	Contains your street map index
PointToPointRouter.cpp	Contains your point to point router
DeliveryOptimizer.cpp	Contains your delivery optimizer
DeliveryPlanner.cpp	Contains your overall delivery planner
report.docx, report.doc, or report.txt	Contains your report

These two are optional:

support.h	You may define support constants/classes/functions in these support files
support.cpp	and use them in your other source files

Use support.h and support.cpp if there are constants, class declarations, functions, and the like that you want to use in *more than one* of the other files. (If you wanted to use something in only one file, then just put it in that file.) Use support.cpp only if you declare things in support.h that you want to implement in support.cpp.

You are to define your class declarations and all member function implementations directly within the specified .h and .cpp files. You may add any #includes or constants you like to these files. You may also add support functions for these classes if you like (e.g., *operator*<). Make sure to properly comment your code.

You must submit a brief (You're welcome!) report that presents the big-O for the average case of the following methods. Be sure to make clear the meaning of the variables in your big-O expressions, e.g., "If the *StreetMap* holds N geo-coordinates, and each geo-coordinate is associated with S street segments on average, *getSegmentsThatStartWith()* is $O(S^2 \log N)$."

- StreetMap: *load*(), *getSegmentsThatStartWith*()
- PointToPointRouter: *generatePointToPointRoute*()
- DeliveryOptimizer: *optimizeDeliveryOrder()*

Grading

- 90% of your grade will be assigned based on the correctness of your solution
- 5% of your grade will be based on your report

- 5% of your grade will be based on the optimality of your returned routes

Optimality Grading (5%)

Your route-finding algorithm does not need to find an optimal solution to get most of the credit for Project 4; it need only return a valid solution. That said, 5% of the points for Project 4 will be awarded based on the optimality of the routes your algorithm finds.

Given a test case, you will get 1 point for your route for that test case if it is a valid route that is within 10% of our optimal least-total-distance solution for that test case. We will then total up the number of points you received, divide it by the total number of test cases, and multiply this by 5% to compute your optimality grade (out of a total of 5%). So if your algorithm were to return an almost-optimal solution (within 10% of our solution) in 35 of our 50 test cases, then you'd get a 70% optimality rating (35/50), and we'd give you 70%*5% points for optimality, or 3.5% out of the possible 5% of your grade for optimality.

Useful Points For Testing

We've located these street intersections on the map for you, and they may be useful for you to test your code:s

Broxton and Weyburn: 34.0625329 -118.4470263
Sproul Landing Intersection 34.0712323 -118.4505969
De Neve Suites Intersection 34.0718238 -118.4525699
Levering and Strathmore 34.0656797 -118.4505131
Kelton and Ophir 34.0683189 -118.4536522
Strathmore and Westwood Plaza (Eng IV) 34.0687443 -118.4449195
Gayley and Strathmore (Beta Theta Pi frat) 34.0685657 -118.4489289
De Neve Plaza @ Charles E Young Drive 34.0706349 -118.4492679
Le Conte and Westwood Blvd 34.0636860 -118.4453568
Broxton and Weyburn 34.0625329 -118.4470263
Hilgard and Weyburn 34.0616291 -118.4416199
Charles E Young and De Neve 34.0711774 -118.4495120
Hilgard and Manning 34.0660665 -118.4385079

Good luck!