The project directory

Change into your project directory and take a look around. What you will see will look very familiar. Once again, your project directory is capable of building three separate programs that you can run by issuing the commands *./*run app**, *./**run exp***, or *./**run gtest***. As in the previous project, there are no precompiled libraries, though there is a fair amount of pre-written code; after reading through the project write-up, take a look through the provided code and be sure you understand what problems it solves, so you can understand what parts of the problem you'll need to solve yourself. The usual *__app__*, *__core__*, *__exp__*, and *__gtest__* directories are available for writing your code, and they serve the same purpose as always.

Note that you'll be required to write code in the *__app__*, *__core__*, and *__gtest__* directories during this project. (More on this later.)

Our abstraction of a street map

Real-life street maps, such as those you see online like Google Maps or those displayed by navigation systems in cars, are a handy way for people to determine an appropriate route to take from one location to another. They present an abstraction of the world as a scaled-down drawing of the actual streets. In order to be useful to us, a street map needs to give us the names of streets and freeways, to accurately demonstrate distances and directions, and to show us where the various streets and freeways intersect.

For our program, we'll need to develop a different abstraction of a street map. Our abstraction must contain the information that is pertient to the problem we're trying to solve, presented in a way that will make it as easy as possible for our program to solve it. Not surprisingly, a picture made up of lines and words is not an abstract that is useful to our program; it would require a tremendous amount of effort to design and implement an algorithm to interpret the lines and words and build up some alternative representation that's more convenient. It's better that we first design the more convenient representation, then train our program to read and understand an input file that specifies it. To do so, we'll need to consider the problem a bit further.

Our program's main job is to discover the shortest distance or driving time between two */locations/*. There's no reason we couldn't think of locations as being any particular point on a street map (for example, any valid street address, or even any valid GPS coordinate). For simplicity, though, we'll think of them as points on the map in which decisions would need to be made, such as:

- The intersection of two or more streets
- A point on a freeway at which there is an entrance and/or an exit

Connecting pairs of locations on the map are stretches of road. In order to solve our problem, we'll need to know two things about each stretch of road:

- Its length, in miles
- The current speed of traffic traveling on it, in miles per hour

Our map will consist of any kind of road that a car can pass over (e.g., /*streets*/ or /*freeways*/), though it will not make a clear distinction between them. In general, all of these are simply stretches of road that travel in a single direction. For example, you could think of a simple two-lane street as a sequence of intersections, connected by stretches of road of equal length running in opposite directions; note, though, that the speed of traffic on either side of the street might be different.

In real life, many intersections control traffic using stop signs or traffic lights. Our program will ignore these controls; we'll instead assume that the traffic speeds on streets have been adjusted appropriately downward to account for the average time spent waiting at stop signs and lights.

Also, to keep the problem relatively simple, absolute directions (i.e., north, south, east, and west) will not be considered by our program or reported in its output. For that reason, they won't be included in our abstraction of a street map, except optionally in the names of locations.

The output of our program will be a /*trip*/. A trip is a sequence of visits to locations on the map. For example, when I used to live in Costa Mesa (and parked my car at UCI in a parking lot where Bren Hall now stands; this wasn't recent!), my typical trip home from UCI looked like this:

- Begin at Peltason & Los Trancos
- Continue to Bison & Peltason
- Continue to Bison & California
- Continue to Bison & 73N on-ramp
- Continue to 73N @ Birch
- Continue to 73N @ 73N-to-55N transition
- Continue to 55N @ Baker
- Continue to 55N Baker/Paularino off-ramp & Baker
- Continue to Baker & Bristol

In addition to the information above, your program will also output information about the distance in miles and (sometimes) driving time of each of the segments of the trip, as well as the overall distance and (sometimes) driving time for the whole trip.

Representing our abstraction of a street map

If you consider all of the data that we'll need to represent this abstraction, the task of organizing it can seem overwhelming. However, there is a well-known data structure that represents this system in a straightforward way: a */directed graph/*. Using a directed graph, locations on our map can be represented as */vertices/*, and the stretches of road connecting locations can be represented as */edges/*. (Since traffic travels in only one direction on a given stretch of road, it makes good sense that the graph should be directed.)

Each vertex in the graph will have a human-readable name for the location it represents. For example, a vertex might be named *__Culver & Harvard__*, or it might be named *__I-405N @ Jamboree__*. The name will be used only for display purposes; it won't have any significance in our algorithm. The vertices should be numbered uniquely and consecutively, starting at zero. If there are */n/* vertices, they should be numbered 0 .. */n/* − 1.

Each edge will contain the two necessary pieces of information about the stretch of road it represents: the distance between the two vertices (in miles, stored as a *__double__*) and the current speed of traffic (in miles per hour, also stored as a *__double__*).

Since a trip is a sequence of visits to adjacent locations on the map, locations are represented as vertices, and two locations are adjacent only when there is a stretch of road (i.e., an edge) connecting them, a trip can be represented as a path in the graph. So our main goal is to implement a kind of */shortest path/* algorithm.

The program

The goal of your program is to take as standard input (i.e., via *__std::cin__*) a description of all of the locations on a map and the stretches of road that connect them. It then performs two tasks:

1 Ensures that it is possible for every location to be reached from every other location. If we think of the locations and roads as a directed graph, that boils down to the problem of determining whether the graph is */strongly connected/*. If not, the message *__Disconnected Map__* should be printed and the program should end.

2 Determines, for a sequence of */trip requests/* listed in the input, shortest distances or shortest times between pairs of locations.

Check out this [sample input], a copy of which you'll find in the *__inputs__* directory in your project directory. A description of its format follows.

The input is separated into three sections: the locations, the road segments connecting them, and the trips to be analyzed. Blank lines (and, similarly, lines containing only spaces) should be ignored. Lines beginning with a *#* character indicate comments and should likewise be ignored. This allows the input to be formatted and commented, for readability.

The first section of the input defines the names of the map locations. First is a line that specifies the number of locations. If there are /*n*/ locations, the next /*n*/ lines of the input (not counting blank lines or comments) will contain the names of each location. The locations will be stored in a directed graph as vertices. Each vertex is to be given a number, with the numbers assigned consecutively in the order they appear in the input, starting at 0.

The next section of the input defines the road segments. Each road segment will be an edge in the directed graph. The first line of this section specifies the number of segments. Following that are the appropriate number of road segment definitions, with each segment defined on a line with four values on it:

1 The vertex number where the segment begins
2 The vertex number where the segment ends
3 The distance covered by the segment, in miles
4 The current speed of the traffic on the segment, in miles per hour

Finally, the trips are defined. Again, the section begins with a line specifying the number of trips. Following that are an appropriate number of trip requests, with each trip request appearing on a line with three values on it:

1 The starting location for the trip
2 The ending location for the trip
__3__ *__D__* if the program should determine the shortest distance, or *__T__* if the program should determine the shortest driving time

Your program should read the vertices and edges from the input, build the graph, then process the trip requests in the order that they appear. The output for each trip request is described later in this write-up.

You may assume that the input will be formatted according to the rules described above, but you /*may not*/ assume that the input we'll use to test the program will be identical to the sample. Different numbers of vertices, different configurations of edges, different names, difference distances and speeds, etc., are possible.

How the program works when you first start

The program compiles, but doesn't do anything when you run it. The member functions declared in the *__Digraph__* class template – see below – are not yet implemented, and the *__app__* contains a fair amount of code, but has an empty *__main()__* function. It's up to you to fill in the missing details. (More on this below.)

Implementing your Digraph

You'll be implementing a class template *__Digraph<VertexInfo, EdgeInfo>__*, which is a directed graph that carries an object of type *__VertexInfo__* associated with each vertex and an object of type *__EdgeInfo__* associated with each edge. While your program will use your *__Digraph__* class template with one particular kind of each – your *__VertexInfo__* might be a string containing the location's name, while your *__EdgeInfo__* might be a structure specifying distance and speed – your *__Digraph__* class template should be generic and reusable, so it should not make assumptions about the program as a whole.

You'll find a declaration of *__Digraph__* in the file *__core__/*Digraph.hpp** *in your project directory. One of the primary goals of this project is to implement that class template, including all of the member functions declared in *Digraph.hpp**. *Note that we will be running our own unit tests against your implementation, so /you cannot/* change the signatures of public member functions in any way, and you cannot change the fact that *__Digraph__* is a template with two type parameters (VertexInfo and EdgeInfo), as our unit tests will presume that these things have not changed. You can, however, add any additional member functions and member variables to the class you'd like; it's just important that existing members remain unaltered.

Implementation tradeoffs

There are two well-known approaches that can be used to implement a graph: an */adjacency matrix/* and */adjacency lists/*. As we've discussed in class, sparse graphs (that is, graphs with few edges outgoing from each vertex) are better implemented using adjacency lists, since an adjacency matrix would waste substantial amounts of memory and time storing and processing the vast number of blank cells in the matrix. Our street map is clearly a sparse graph, since each vertex will have edges to and from only a few relatively "nearby" vertices. So, adjacency lists are a clearly superior approach in our case. You are required to use this approach to represent your graph.

One approach is to primarily store a *__std::map__* with keys being vertex numbers (which are not necessarily consecutive or zero-based, making something like *__std::vector__* or an array a tougher choice) and values being a structure containing information about each vertex (among other things, a *__std::list__* containing information about each edge outgoing from that vertex).

Erring on the side of using well-behaved classes wherever possible in your *__Digraph__* implementation will reduce the complexity of memory management significantly; consider that when you're making your choices.

Constraints on the template arguments

You have somewhat more freedom here to choose reasonable constraints for your template arguments. You might be surprised how few constraints you need to impose on them, because there is relatively little that your *__Digraph__* needs to do with *__VertexInfo__* and *__EdgeInfo__* objects besides store them, find them, and return them. In a comment near the top of *__Digraph.hpp__*, list any constraints on the *__VertexInfo__* and *__EdgeInfo__* type parameters introduced by your template.

Sanity-checking your Digraph implementation

To ensure that your Digraph is compatible with our unit tests, a set of "sanity-checking" unit tests are included in the *__gtest__* directory in your project directory. They make no attempt to validate any of the Digraph's functionality, but they do at least ensure that your Digraph contains all of the necessary member functions and will be compatible with our unit tests. Initially, the sanity-checking unit tests will not compile — this is why they are all commented out — but as you work, you'll be able to gradually uncomment them and see them compile. If you haven't successfully uncommented and compiled all of the sanity-checking unit tests, your Digraph will not compile against our unit tests.

As you work, you may discover that the sanity-checking tests that once compiled and linked successfully suddenly don't anymore; this is actually a clue that something important may have changed, so you'll want to be cognizant of it. Many of the errors you'll get from the sanity-checking tests are actually linker errors, which can be a bit difficult to unravel when you haven't had a lot of practice with them, but if you compile relatively often, there won't be many candidates whenever you have a problem; focus on changes you made most recently and you'll find your likely culprit.

Finding the shortest paths

The problem we need to solve, that of finding the fastest or shortest trip along a network of roads, is not an uncommon one in computing. In fact, it's so common that it's already been solved abstractly. Our problem is an instance of the /*single-source, positive-weighted, shortest-path problem*/. In other words, from one particular vertex (a "single source"), we'll be finding the shortest path to another vertex, where all of the edges have a "positive weight" (in our case, distance or speed, neither of which will ever be negative or zero) associated with them. We'll use a well-known algorithm called Dijkstra's Shortest-Path Algorithm to solve this problem.

Dijkstra's Algorithm actually finds the shortest path from some start vertex to /*all*/ the other vertices in a graph — this doesn't slow the algorithm down, since it needs to calculate them all in order to find the desired answer — though we're only interested in one of the paths that it will find. There's a benefit to Dijkstra's calculation of all the shortest paths from some vertex. Suppose the file has multiple trips starting from the same vertex. With Dijkstra's Algorithm, we can compute shortest paths from any particular start vertex to all other vertices once for distance and once for time, storing the results in memory. Then, to learn the shortest path from that start vertex to any other vertex, we can just look up the answer. Use this approach in your program; it is likely the file will contain multiple trips that start from a particular place, and only a poorly-designed solution would require the program to re-compute data it has already computed, unless memory was at such a premium that there wasn't enough space to store previous results (and, since this program is running on PCs and laptops, there will be plenty of memory avaliable).

For each vertex /*v*/, Dijkstra's Algorithm keeps track of three pieces of information: /*k*//*v*/, /*d*//*v*/, /*p*//*v*/.

- /*k*//*v*/ is a boolean flag that indicates whether the shortest path to vertex /*v*/ is known. Initially, /*k*//*v*/ is ***false*** for all vertices.
- /*d*//*v*/ is the length of the shortest path found thusfar from the start vertex to /*v*/. When the algorithm begins, no paths have been considered, so /*d*//*v*/ is initially set to ∞ for all vertices, except the start vertex, for which /*d*//*v*/ = 0.
- /*p*//*v*/ is the predecessor of the vertex /*v*/ on the shortest path found thusfar from the start vertex to /*v*/. Initially, /*p*//*v*/ is ***unknown*** for all vertices, except for the start vertex, for which /*p*//*v*/ is ***none***.

As the algorithm proceeds, it will need to calculate the /*cost*/ for individual edges. The cost of the edge from /*v*/ to /*w*/ will be called /*C*/(/*v*/, /*w*/). How you calculate the cost depends on whether you're minimizing driving distance or driving time:

- If you're minimizing driving distance, /C/(/v/, /w/) is the number of miles on the edge from /v/ to /w/.
- If you're minimizing driving time, /C/(/v/, /w/) is the amount of time (in seconds, let's say) required to drive along the edge from /v/ to /w/, given its length and traffic speed.

  Dijkstra's Algorithm proceeds in phases. The following steps are performed in each pass:

  1 From the set of vertices for which /k//v/ is ***false***, select the vertex /v/ having the smallest /d//v/. In other words, of the shortest paths to each vertex that we've found that we're not yet sure about, pick the one that is the shortest.

  2 Set /k//v/ to ***true*** for the vertex you picked in step 1. The shortest of the "unknown" paths is now considered to be known.

  3 For each vertex /w/ adjacent to /v/ (i.e., there is an edge from /v/ to /w/) for which /k//w/ is ***false***, test whether /d//w/ is greater than /d//v/ + /C/(/v/, /w/). If it is, set /d//w/ to /d//v/ + /C/(/v/, /w/) and set /p//w/ to /v/. In other words, if the path through /v/ to /w/ is better than the shortest path we'd found to /w/ so far, the shortest path to /w/ (so far) is the path we've just found through /v/ to /w/.

  For each pass, exactly one vertex has its /k//v/ set to ***true*** (in other words, we discover one known shortest path per pass).

  Here is psuedocode for the algorithm. Notice the use of a priority queue, which allows you to efficiently find the vertex with the smallest /d//v/ in step 1.

```
for each vertex v
{
    set kv to false
    set pv to unknown (or none, if v is the start vertex)
    set dv to ∞ (or 0, if v is the start vertex)
}

let pq be an empty priority queue
enqueue the start vertex into pq with priority 0

while (pq is not empty)
{
    vertex v = the vertex in pq with the smallest priority
    dequeue the smallest-priority vertex from pq

    if (kv is false)
    {
        kv = true

        for each vertex w such that edge v → w exists
        {
            if (dw > dv + C(v, w))
            {
                dw = dv + C(v, w)
                pw = v
                enqueue w into pq with priority dw
            }
        }
    }
}
```

At the conclusion of the main loop, the /*d*//*v*/ value corresponding to the end vertex will be the amount of the shortest path. You can find the actual path of vertices by working your way backward from the end vertex to the start vertex, following the /*p*//*v*/ values as you go along. (This implies, of course, that you need to store all the /*p*//*v*/ values.)

Remember that, after the algorithm has finished, you should store the results in memory so that you can look them up later. I suggest storing the /*p*//*v*/ values long-term. There's no reason to store the /*k*//*v*/ values, because they will all be *__true__* after the algorithm is completed. And there's no need to store the /*d*//*v*/ values, because you will need to lookup the times or distances between each vertex in the path anyway, since we always output all of the segments of a trip, and you can easily sum these up to calculate a total while you're generating your answer.

As you can see from the pseudocode, you will need to use a priority queue in order to implement Dijkstra's Algorithm efficiently. You can implement your own priority queue, if you'd like – though it should perform its enqueues and dequeues in O (log n) time if you do – but you can also use ***std::priority_queue*** from the C++ standard library if you prefer. Note that ***std::priority_queue*** is a little bit trickier than it looks. In particular, it stores objects without separate priority values; it then uses a *compare function* to determine which object is considered to be more important.

The output

For each of the trip requests in the input file, your program should output a neatly-formatted report to the console that includes each leg of the trip with its distance and *or time (as appropriate), and the total distance and* or time for the trip.

If the trip request asks for the shortest distance, the output might look something like the following. (These are phony trips, to show you the output format; they are not related to the sample data file provided above.)

```
Shortest distance from Alton & Jamboree to MacArthur & Main
  Begin at Alton & Jamboree
  Continue to Main & Jamboree (1.1 miles)
  Continue to Jamboree & I-405N on ramp (0.3 miles)
  Continue to I-405N @ MacArthur (1.3 miles)
  Continue to MacArthur & I-405N off ramp (0.1 miles)
  Continue to MacArthur & Main (0.2 miles)
Total distance: 3.0 miles
```

On the other hand, if the trip request asks for the shortest time, the output might look like this:

```
Shortest driving time from Alton & Jamboree to MacArthur & Main
  Begin at Alton & Jamboree
  Continue to Alton & MacArthur (2.7 miles @ 33.7mph = 4 mins 48.8 secs)
  Continue to Main & MacArthur (1.1 miles @ 40.1mph = 1 min 38.7 secs)
Total time: 6 mins 27.5 secs
```

When outputting a time, you should separate it into its components – hours, minutes, and seconds – as appropriate. Here are some examples:

```
32.5 secs
2 mins 27.8 secs
13 mins 0.0 secs
3 hrs 13 mins 12.3 secs
6 hrs 0 mins 0.0 secs
```

Don’t show hours if there are zero of them. Don’t show hours or minutes if there are zero of both of them.

The provided code

A fair amount of code has been provided already in the *__app__* directory in your project directory, so some of what is described above is actually already implemented. In particular, you’ll find these things useful:

All of the input parsing is already implemented. In particular, check out the *__InputReader__* class, which handles the detail of reading lines of input and skipping the non-meaningful ones (the comments and blank lines); the *__RoadMapReader__* class, which reads the map information from the input and returns a *__RoadMap__* (which is a particular instantiation of the *__Digraph__* class template); and the *__TripReader__* class, which reads the trip requests from the input and returns a vector of *__Trip__* structures (each describing one trip request).

A *__RoadMapWriter__* class will allow you to see a human-readable representation of your graph, which you might find useful in debugging.

Some data types that you’ll probably need are defined: *__RoadMap__*, *__RoadSegment__*, *__Trip__*, and *__TripMetric__*.

Any of the provided code in the *__app__* directory can be replaced or removed; it’s up to you. This code is provided primarily to let you focus on the interesting parts of the project, without spending too much time on grunt work.

Note, too, that the provided code will compile but will not link initially. That’s because it depends on implementations of the member functions of the *__Digraph__* class template, which are missing (and which you’ll need to implement).

Unit testing

It would be a good idea to separately unit test your *__Digraph__* implementation. While this is not an explicit requirement, we will be running a set of unit tests as part of how we grade the project, so you’ll want to be sure you handled the various scenarios correctly, including those that didn’t come up directly in the application you’re building; unit tests are a great, simple way to do that.

Before building a program atop a data structure implementation, it's vital to be sure that the data structure actually works as intended, so you'd be well-advised to implement your unit tests alongside your *__Digraph__* implementation work, proceeding with the rest of the program only once you've got a *__Digraph__* that is complete and correct. (This is the right strategy if you're thinking about partial credit for incomplete work, as a large part of the challenge here — and, therefore, a large portion of the credit — is allocated to your *__Digraph__*.)

As usual, write your unit tests in the *__gtest__* directory in your project directory. Write them in a source file */other than/* the provided one that contains sanity-checking unit tests; your unit tests have a separate goal.

Writing your own unit tests is not a requirement, but I would strongly recommend it. Among other reasons, this gives you a way to work on your *__Digraph__* implementation and verify that it works piecemeal, even though the provided code in the *__app__* directory won't link until many of the parts of *__Digraph__* are implemented. Simply work on whichever part of *__Digraph__* you want to implement, write unit tests for it, and run the *__./__*_build gtest_* _command to build /only/_ your unit tests, and it won't matter that the *__app__* is not ready to be built yet.