Programming Assignment #5

Due: 2015/12/10 at 11:59pm
Points: 20

Overview

For our final programming assignment, we'll finish off the theme of maps by writing a program to generate driving directions. A quick preview (user input in blue):

```
$ ./route
Starting address: 620 W Washington St
Ending address:   4477 Progress Dr
0.250207 miles on W Washington St
0.837454 miles on Laporte Ave
0.0397415 miles on Wilber St
1.76114 miles on Lincoln Way W
0.0729842 miles on Maplewood Ave
0.400587 miles on Commerce Dr
0.0442461 miles on Progress Dr
```

Warning: We think this is the hardest programming assignment in the class (but hopefully also the most fun). You have three weeks to complete it, so plan ahead.

Distribution

Update your local Git repository (as described at `https://bitbucket.org/CSE-30331-FA15/cse-30331-fa15`) and change to the directory PG5. It contains the following files:

`pg5.pdf`	this file
`street_map.{cpp,hpp}`	skeleton solution
`southbend.v2.map`	a map of South Bend and Notre Dame
`Makefile`	simple Makefile
`route.cpp`	command-line navigation
`test{1,3,4,5a,5b}.cpp`	test programs
`random.txt`	used by `test5b`
`measure.c`	measure time and memory usage
`README.md`	README

The format of `southbend.v2.map` is an extension of that used in PG4. The first few lines look like this:

```
N: Bercliff Dr
E: 11397633 11397632 0.0482335773516 https://maps.googleapis.com/...
E: 11397632 11397627 0.0391453933473 https://maps.googleapis.com/...
R: 0 300 314 11397628 11397633 0.0361305195783 https://maps.googleapis.com/...
R: 0 316 332 11397633 11397632 0.0482335773516 https://maps.googleapis.com/...
```

Each line begins with `N:`, `E:`, or `R:`. A line that starts with `N:` gives the name of a street. Then, the lines that start with `R:` are exactly as in PG4:

1. A parity: 0 means even, 1 means odd.
2. The starting address of the range (inclusive).
3. The ending address of the range (inclusive).
4. The ID of the starting node.
5. The ID of the ending node.
6. The length in miles of this street segment.
7. The URL of an image showing this street segment on a map.

The lines that start with `E:` describe street segments that have no addresses on them. These lines have only 4 fields:

1. The ID of the starting node.
2. The ID of the ending node.
3. The length in miles of this street segment.
4. The URL of an image showing this street segment on a map.

(There is some redundancy in this file format. If we were designing it again from scratch, we'd probably do it a little differently.)

Task

For this assignment, you are free to use any classes and functions from the standard library. You are also expected to reuse code from PG4, and can use other students' PG4 solutions as long as you cite them.

1. First, we need the geocoder to return information that is usable for navigation. Modify your map-reading code from PG4 to store the starting and ending node (as `int`s) and the length (as a `float`) of each street segment. You can just skip the `E:` lines for now. Then modify your geocoding function to have signature

   ```
   bool street_map::geocode(const std::string &address,
                            int &u, int &v, float &pos) const;
   ```

   where the output arguments `u` and `v` are the start and end nodes of the street segment found, and the output argument `pos` is how far from `u` the address lies. This is not an exact science; let's say that the position is

   $$\mathtt{pos} = \mathtt{length} \times \frac{\mathtt{hn} - \mathtt{fromhn}}{\mathtt{tohn} - \mathtt{fromhn} + 2},$$

   where `length` is the length of the segment, `fromhn` and `tohn` are its starting and ending house numbers, and `hn` is the house number that was searched for.

   At this point, `test1` should pass all tests.

2. Extend your map class to maintain an adjacency list for an *undirected* graph representing the map. Every `E:` line or `R:` line specifies an edge of this graph. Store a street name and a length (as a `float`) for each edge. Note that it's possible for an edge to occur more than once in the map file. In order for the test programs to work, let's say arbitrarily that **the canonical name of a street is the one that occurs last in the file**. Below, when we refer to the canonical name of a street, this is what we mean.

3. Now we'll implement the navigation algorithm step by step. The first step is to implement Dijkstra's algorithm. Write a function

   ```
   bool street_map::route3(int source, int target, float &distance) const;
   ```

   that takes two node IDs, `source` and `target`, and finds the shortest distance between them. If successful, it puts the shortest distance into `distance` and returns `true`. Otherwise, it returns `false`.

   Note that the standard Dijkstra's algorithm requires the priority queue to have a `increase_key` operation, which `std::priority_queue` doesn't have. The version of Dijkstra's algorithm we used in class allows the frontier to have multiple copies of the same node, and therefore it doesn't have this problem. You are free to pursue an alternative solution to this problem (e.g., implementing `increase_key`) if you want to.

   At this point, `test3` should pass all tests.

4. Next, write a function

   ```
   bool street_map::route4(int su, int sv, float spos,
                           int tu, int tv, float tpos,
                           float &distance) const;
   ```

   that takes the result of geocoding source and target addresses (in `su`/`sv`/`spos` and `tu`/`tv`/`tpos`, respectively), and computes the shortest distance between them. Now the source and target are no longer nodes of the graph; they lie along edges of the graph.

   If the source and target addresses are on the same street segment, return the distance between them.

   Otherwise, use a modified Dijkstra's algorithm. One way to do this is to hallucinate nodes for them. For example, if the source point is 304 Bercliff Dr, then you can hallucinate a source node that is 0.09 miles from node 11397628 and 0.27 miles from node 11397633. To do this, you can create fake IDs for the source and target, and add special cases to the algorithm to handle them:

   SOURCE ← −2
   TARGET ← −1
   sl ← length of edge (su, sv)
   tl ← length of edge (tu, tv)
   add SOURCE to marked
   add su to frontier with distance spos
   add sv to frontier with distance sl - spos
   while frontier is not empty **do**
       take node u with distance d from frontier
       add u to marked
       if u = tu **then**
           add TARGET to frontier with distance d + tpos
       if v = tv **then**

add TARGET to frontier with distance `d + tl - tpos`
if u = TARGET **then**
break
⋮ (Dijkstra's algorithm as usual)

At this point, `test4` should pass all tests.

5. Finally, write a function

```
bool street_map::route(int su, int sv, float spos,
                       int tu, int tv, float tpos,
                       std::vector<std::pair<std::string, float>> &steps) const;
```

that actually generates driving directions for the shortest path. The function should clear out the old contents of `steps` (if any) and put the the shortest path into `steps`. Each element of `steps` is a pair consisting of a **canonical** street name and a distance in miles. If two consecutive elements have the same name, they should be merged (and their distances added).

At this point, `test5a` should pass all tests, and the command-line tool `route` should also work as shown at the beginning of the assignment. The test program `test5b` should run in 30 seconds or less.

6. In `README.md`, write a short description of the data structures and algorithms you used, and any other feedback you'd like to offer about what you learned or didn't learn from this assignment.

Rubric

Part 1 correct and `test1` passes all tests	3
Part 2 correct	3
Part 3 correct and `test3` passes all tests	3
Part 4 correct and `test4` passes all tests	3
Part 5 correct and `test5a` passes all tests	3
`test5b` runs in 30 seconds or less	1
No errors according to Valgrind	1
Good style	1
Good comments	1
Good README	1
Total:	20

Submission

1. To submit your work, upload it to your repository on Bitbucket:
   (a) `git add` *`filename`* for every file that you created or modified
   (b) `git commit -m` *`message`*
   (c) `git push`

   The last commit made at or before 11:59pm will be the one graded.

2. Double-check your submission by cloning your repository to a new directory, running `make` and running all tests.