ECE297 Milestone 3
Pathfinding and Giving Directions

"If you don't know where you are going, you'll end up someplace else."

– Yogi Berra

"If you don't know where you're going, any road will take you there."

– Lewis Carroll

Assigned on Tuesday, February 26

Due on Tuesday, March 19

User Interface and In-lab demo = 5/15

Autotester = 8/15

Coding Style and Project Management = 2/15

Total marks = 15/100

1 Objective

In this milestone you will extend your code to find good travel routes between two points embedded in a graph. Algorithms to find paths in graphs are important in a very wide range of areas, including GIS applications like yours, integrated circuit and printed circuit board design, networking / internet packet routing, and even marketing through social media.

By the end of this assignment, you should be able to:

1	Find the shortest path between two nodes in a graph.

2	Develop a user interface for finding and reporting travel directions.

2 Path Finding

Your code should implement the three functions shown in `m3.h`; we will automatically test these functions with unit tests. Your `load_map` function will always be called by the unit tests before we call any of the functions in Listing 1. Some tests of each function will be public and available to you to run with `ece297exercise 3` while others will be private and run only by the automarker after you submit your code.

```
1 /*
2  * Copyright 2019 University of Toronto
3  *
```

```
 * Permission is hereby granted, to use this software and associated
 * documentation files (the "Software") in course work at the University
 * of Toronto, or for personal use. Other uses are prohibited, in
 * particular the distribution of the Software either publicly or to third
 * parties.
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */
#pragma once
#include <vector>
#include <string>


// Turn type: specifies if the next ture is right, left or going straight
enum class TurnType
{
    STRAIGHT, // going straight
    RIGHT, // turning right
    LEFT, // turning left
    NONE // no turn detected
};


// Returns the turn type between two given segments.
// street_segment1 is the incoming segment and street_segment2 is the outgoing
// one.
// If the two street segments do not intersect, turn type is NONE.
// Otherwise if the two segments have the same street ID, turn type is
// STRAIGHT.
// If the two segments have different street ids, turn type is LEFT if
// going from street_segment1 to street_segment2 involves a LEFT turn
// and RIGHT otherwise.  Note that this means that even a 0-degree turn
// (same direction) is considered a RIGHT turn when the two street segments
// have different street IDs.
TurnType find_turn_type(unsigned street_segment1, unsigned street_segment2);


// Returns the time required to travel along the path specified, in seconds.
// The path is given as a vector of street segment ids, and this function can
// assume the vector either forms a legal path or has size == 0.  The travel
// time is the sum of the length/speed-limit of each street segment, plus the
// given right_turn_penalty and left_turn_penalty (in seconds) per turn implied
// by the path.  If the turn type is STRAIGHT, then there is no penalty
double compute_path_travel_time(const std::vector<unsigned>& path,
                                const double right_turn_penalty,
```

```
                                const double left_turn_penalty);


// Returns a path (route) between the start intersection and the end
// intersection, if one exists. This routine should return the shortest path
// between the given intersections, where the time penalties to turn right and
// left are given by right_turn_penalty and left_turn_penalty, respectively (in
// seconds).  If no path exists, this routine returns an empty (size == 0)
// vector.  If more than one path exists, the path with the shortest travel
// time is returned. The path is returned as a vector of street segment ids;
// traversing these street segments, in the returned order, would take one from
// the start to the end intersection.
std::vector<unsigned> find_path_between_intersections(
      const unsigned intersect_id_start,
                  const unsigned intersect_id_end,
                  const double right_turn_penalty,
                  const double left_turn_penalty);
```

Listing 1: m3.h

The most important function in `m3.h` is `find_path_between_intersections`. This function must pass 3 types of tests; you can obtain part marks if you pass some of the checks but fail others, but you will have to pass them all to obtain full marks.

- The route must be legal – that is, it must be a sequence of street segments which form a connected path from the start intersection to the end intersection. You must also respect one way streets, and not try to travel down them in the wrong direction. Note that it is also possible that no legal route exists between two intersections, in which case you should return an empty (size = 0) vector of street segments.
- Your route should have the minimum possible travel time between the two intersections, where travel time is defined below.
- You should find the route quickly; you will fail performance tests if your path-finding code is not fast enough.

The travel time required to traverse a sequence of street segments is the sum of two components.

- The time to drive along each street segment, which is simply the length of the street segment divided by its speed limit.
- The time to make turns between street segments. We assume that no turn is required when you travel from one street segment to another *if they are both part of the same street* – that is, we are assuming you hit only green lights when you are going straight down a street. When you travel from a street segment that is part of one street (e.g. *Yonge Street*) to a street segment that is part of another street (e.g. *Bloor Street*) however, you will have to make a turn and this will cost extra time. A left turn takes **left_turn_penalty** seconds and a right turn takes **right_turn_penalty** seconds; where left_turn_penalty and right_turn_penalty are parameters passed into your path-finding function. These parameters model the average time it takes for you to wait for a break in traffic and/or a traffic light to turn green so you can turn.

To make it easier to verify that you compute turns and travel times correctly, `m3.h` requires you to implement `turn_type` and `compute_route_travel_time` functions and `ece297exercise` provides unit tests for them. Make sure you pass these unit tests, as you will not be able to find shortest travel time routes if these basic functions are incorrect. See Figure 1 for an example of paths and travel times.

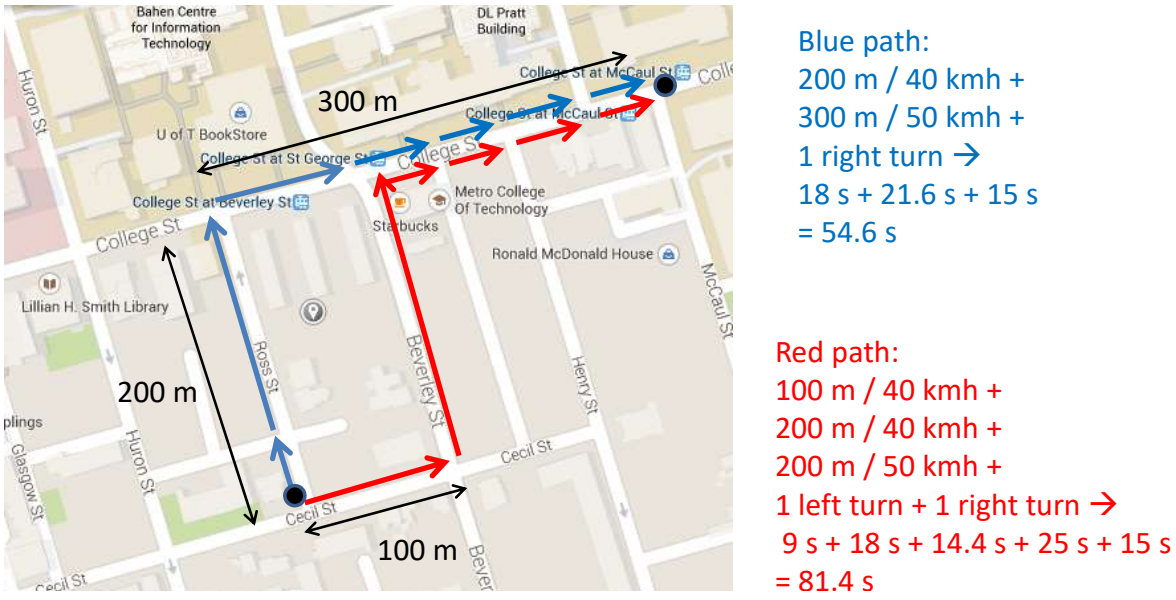

Figure 1: Two paths between Cecil & Ross and College & McCaul; one has a lower travel time than the other. The right_turn_penalty is 15 seconds and the left_turn_penalty is 25 seconds in this example.

3 User Interface

Implementing your path finding algorithm is only part of this milestone; you also need to make this functionality usable by end users. You should decide on commands that will be typed and/or entered with mouse clicks by the user to specify what path he/she is interested in finding. The required features of your user interface are:

- Your program must work with any map without recompilation, so the user should be able to specify the map of interest.
- Users must be able to enter commands that exercise your code to find a path between two intersections (specified by entering street names at each intersection, e.g. Yonge Street and Bloor Street).
- Your interface must allow users to specify partial street names (e.g. Yong) instead of full street names when the partial name is sufficient to identify the street.

- Users must also be able to find paths between intersections by mouse clicking on intersections.
- You must display the found path in the user interface; for full marks this path should be easy to understand and give a good sense of how to follow it.
- You must print out detailed travel directions to the console or in the graphics. For full marks, these travel directions must be sufficiently detailed and clear that a typical driver could follow them.
- You must give informative error messages if the user mistypes something (e.g. an invalid street or map name).
- You must have some method (e.g. a help GUI button or notes in dialog boxes) so new users can learn the interface.

Beyond meeting the basic requirements above, your user interface will be evaluated on how user-friendly and intuitive it is. You should produce *user-friendly* driving directions, using both text and graphical drawing. For example, directly printing the sequence of street segment ids that your path finding code returns would be a completely unusable user interface! Print out clear directions giving all the information you would want if you were being given driving directions. Draw an informative picture of the route to follow. Integrating both your intersection input and driving directions output into the graphics will generally lead to a more user-friendly design than having the user type and read text at the command prompt.

When creating a user interface like this it is a good idea to test it on people who are not part of the development group. Feedback from a person not involved in the design (friends, family, etc.) is very useful in identifying what is intuitive and what is obscure in your interface.

One part of making your interface more user-friendly is to support good matching of (partial) input text to street names. You can use your milestone1 functions to achieve a basic partial name matching, but you can also explore other options to do more than match string prefixes. One option is using the regular expression `< regex>` feature in the C++ standard library, which lets you match general patterns in strings.

4 Grading

- 8/15 marks will come from the autotester.

The auto tester will test basic functionality, speed and that your code has no memory errors or leaks (via a valgrind test).

- 5/15 marks will be assigned by your TA based on an in-lab demo of your interface.

Your interface should meet all the requirements listed above, be easy to use, and feel fast and interactive. The sophistication and ease of use of your interface will be compared to that of the average design in the class to help assess this mark.

- 2/15 marks will be based on your TA's evaluation of your code style and project management, which includes planning and tracking tasks on your wiki and using git effectively.

  Your TA will review git logs and your wiki page, and ask team members questions about the design and each member's contribution to it. If a team member has made a small contribution or shows poor knowledge of the design, his or her mark may be reduced and if merited some marks may be transferred to other team members.