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 Monday, February 27
Due on Tuesday, March 21

User Interface and In-lab demo = 4/13
Autotester = 8/13
Coding Style = 1/13
Total marks = 13/100

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.

```cpp
#pragma once
#include <vector>
#include <string>
```
// 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 turn_penalty (in seconds) per turn implied by the path.
// A turn occurs when two consecutive street segments have different street IDs.
double compute_path_travel_time(const std::vector<unsigned>& path,
                                const double 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 when the time penalty to turn (change
// street IDs) is given by turn_penalty (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 turn_penalty);

// Returns the shortest travel time path (vector of street segments) from
// the start intersection to a point of interest with the specified name.
// The path will begin at the specified intersection, and end on the
// intersection that is closest (in Euclidean distance) to the point of
// interest.
// If no such path exists, returns an empty (size == 0) vector.
std::vector<unsigned> find_path_to_point_of_interest(const unsigned intersect_id_start,
                                                    const std::string point_of_interest_name,
                                                    const double 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 we assume this takes turn_penalty seconds, where turn_penalty is a parameter passed into your path-finding function. This models 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 travel times correctly, m3.h requires you to implement a compute_route_travel_time function and ece297exercise provides unit tests for it. Make sure you pass these unit tests, as you will not be able to find shortest travel time routes if this basic function is 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 turn_penalty is 15 seconds in this example.

The final function in m3.h finds a route from an intersection to a point of interest. Since points of interest are not connected to the street graph, you should return a path (vector of street segments) from the source intersection to the intersection that is closest to (minimum geographic distance from) the point of interest.
The point of interest is specified by name, and this name may not be unique (e.g., *Tim Hortons* will match many points of interest). In cases where more than one point of interest matches the specified name, you should find and return the path to the point of interest that can be reached with minimum travel time. Note that the (name-matched) point of interest reachable in minimum travel time may not always be the one that is geographically closest to the starting intersection, as shown in Figure 2.

![Figure 2: Two “Starbucks” points of interest; the one that is closest by geographic distance to the intersection of College & McCaul is not the one reachable in minimum travel time. For path-finding purposes, each Starbucks is considered to be located at the geographically closest intersection.](image)

### 3 User Interface

Implementing your path finding algorithm / code 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 somehow.
- Users must be able to enter commands that exercise your code to find a path between two intersections (specified by entering the street names at each intersection, e.g. Yonge Street and Bloor Street) and a path from an intersection to a point of interest (which is specified by its name, e.g. Tim Horton’s).
• Users must also be able to find paths between intersections by mouse clicking on inter-
sections.
• You must give informative error messages if the user mistypes a command.
• You must have a help command or help GUI button so new users can learn the interface.
• 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.

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 output and graphics. For example, directly printing the sequence of street
segment ids that your path finding code returns would be a completely unusable user inter-
face! 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.

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.

4 Optional Bonus: Partial Name Matches in User Interface

For better usability you could support partial name matches in these commands, where
your program can fill in the names of streets and points of interest or show a list of possible
matches based on a partially specified name. For example, if a user types Yong your program
could fill in the rest of the name as Yonge Street. If done well, partial name matching is
worth a bonus mark on this lab.

Street names and point of interest names can be long, so finding the names
that match a partially entered name can make your design much more
usable. There are libraries that can help you with that task. The GNU
readline library makes it much easier to support tab-completion of names.
Alternatively you could leverage <regex> in the C++ standard library,
which lets you match text patterns in strings.

Read the GNU Readline Quick Start Guide on the course web
page to see how you can use this library to implement tab-completion
of names.
5 Grading

- 8/13 marks will come from the autotester. The auto tester will test basic functionality, speed and that your have no memory errors or leaks (via a valgrind test).

- 4/13 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. If your interface supports partial name matching, you can earn a bonus mark on this part of the assignment.

- 1/13 marks will be based on your TA’s evaluation of your code style.

- Your TA will review svn 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.