ECE297 Milestone 4
Traveling Courier

"A person who never made a mistake never tried anything new."

–Albert Einstein

Assigned on Wednesday, March 13 Autotester = 12/12

Due on Saturday, April 6 **Total marks = 12/100**

1 Objectives

In this milestone you will find a path through your map that has multiple stops, so that you can find a good route for a courier company driver who has multiple deliveries to make. This is a variation on a classic optimization problem called the traveling salesman problem.

After completing this milestone you should be able to:

1	Find a valid path through a graph that passes through a set of vertices.
2	Develop heuristics to solve a computationally hard problem.

2 Overview and Background

In this milestone you will extend your project with `m4.h` and one or more .cpp files and use these files to implement a variation of the traveling salesman algorithm.

The traveling salesman problem is *computationally hard* which means that there is no known algorithm that (1) gives a guaranteed optimal (lowest travel time) result and (2) has a computational complexity that is a polynomial of N, where N is the number of vertices the salesman must visit. In practice this means that guaranteed optimal algorithms have computational complexity at least exponential, $O(2^N)$, in the problem size and become impractically slow for large enough N. Therefore we must resort to heuristic (i.e. "seems like a good idea") methods that do not guarantee a perfect answer, but which can run much faster. Most optimization problems that are actively researched are computationally hard; coming up with better heuristics for such problems is important in many fields including the design of computer chips, transportation systems, and new pharamaceutical drugs.

3 Detailed Specification

Figure 1 shows the input you are given.

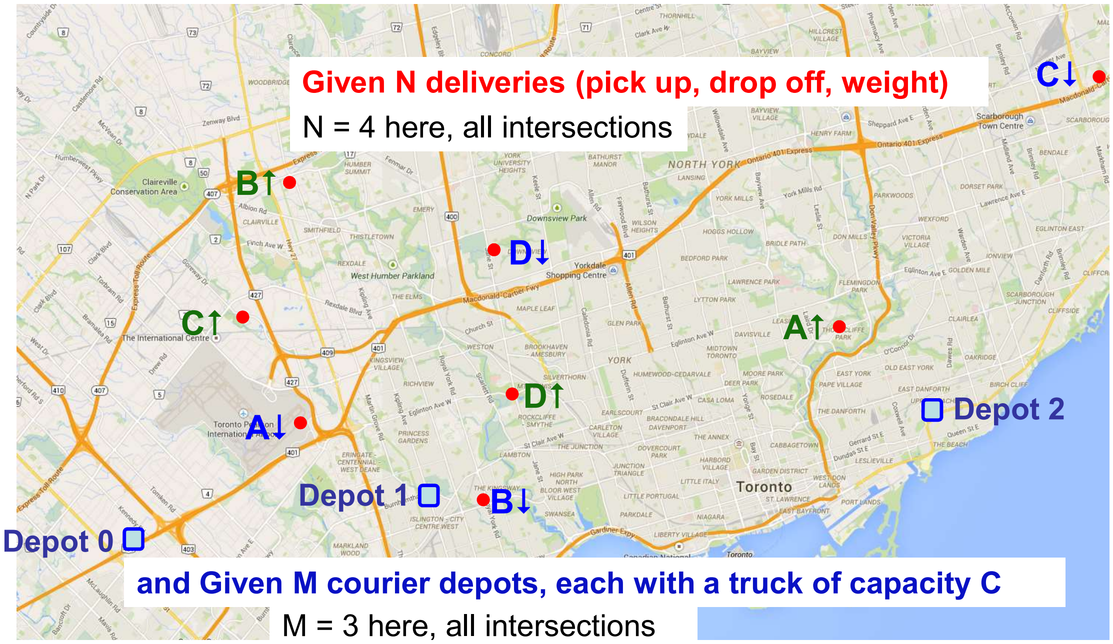


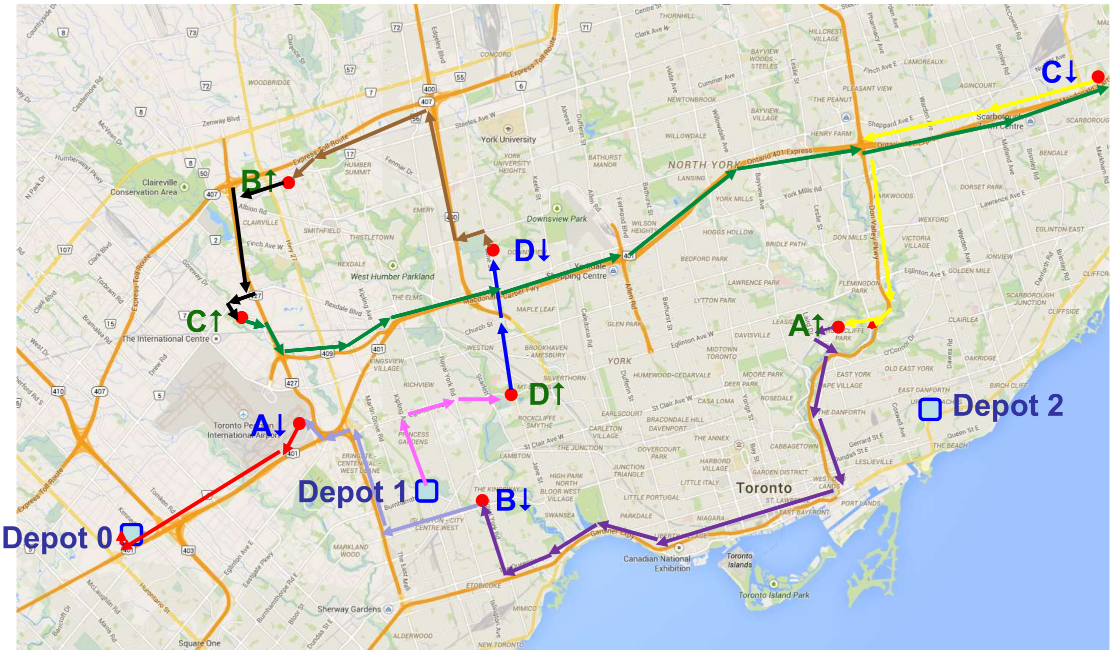
Figure 1: Input to your courier delivery algorithm. In this example you are given 3 depot intersections, and 4 deliveries to make. Each delivery has a pick up intersection (shown here as a letter followed by an up arrow) and a corresponding drop off location (shown here as the same letter with a down arrow).

The courier company has a set of M depots for their delivery trucks, and you can start your day's deliveries from any one of those M depots. At the end of the day you need to return your truck to one of the depots, but it doesn't have to be the same one at which you started.

After picking up your delivery truck, you need to make N deliveries, where each delivery has an intersection at which you pick up the package, and another intersection where you drop off the corresponding package. You can visit these intersections in any order you like, but you must visit the intersections such that all the deliveries are made. A delivery is made when you visit the pick up intersection to pick the package, and then some time later visit the corresponding drop off intersection. It is possible that a single intersection may appear more than once as a pick up location. In that case you can choose to pick up all the packages at that location at once if you have room in your truck; since your truck has a finite weight capacity, sometimes you may not be able to pick up all the packages at once. Similarly, an intersection could appear more than once as a drop off location; in that case, when you visit the intersection you will drop off all the packages you have already picked up that need to go to that intersection.

Figure 2 shows the output your algorithm must generate – adelivery route that that begins at a depot, follows connected street_segments to reach intersections such that all N deliveries are made, and ends at a (possibly the same or possibly a different) depot.

We will unit test your algorithm by calling your `traveling_courier` function which must



Output: vector of CourierSubpath
CourierSubpath: {start_intersection, end_intersection,
deliveries picked up at start, vector of street_segment_ids to follow}
Each colour is a different CourierSubPath

Figure 2: A possible solution generated by your courier delivery algorithm. This example solution contains 9 CourierSubpaths. It starts at depot 1, picks up D, drops off D, picks up B, picks up C, drops off C, picks up A, drops off B, drops off A, and finally ends at depot 0.

have the function prototype shown in Listing 1.

```
/*
 * Copyright 2019 University of Toronto
 *
 * 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>

struct DeliveryInfo {
    //Specifies a delivery order (input to your algorithm).
    //
    //To satisfy the order the item-to-be-delivered must have been picked-up
    //from the pickUp intersection before visiting the dropOff intersection.

    DeliveryInfo(unsigned pick_up, unsigned drop_off, float weight)
        : pickUp(pick_up), dropOff(drop_off), itemWeight(weight) {}

    //The intersection id where the item-to-be-delivered is picked-up.
    unsigned pickUp;

    //The intersection id where the item-to-be-delivered is dropped-off.
    unsigned dropOff;

    // Weight of the item in pounds (lb)
    float itemWeight;
};


struct CourierSubpath {
    // Specifies one subpath of the courier truck route

    // The intersection id where a start depot, pick-up intersection or drop-off intersection
    // is located
    unsigned start_intersection;

    // The intersection id where this subpath ends. This must be the
    // start_intersection of the next subpath or the intersection of an end depot
    unsigned end_intersection;

    // Street segment ids of the path between start_intersection and end_intersection
    // They form a connected path (see m3.h)
    std::vector<unsigned> subpath;

    // Specifies the indices from the deliveries vector of the picked up
    // delivery items at the start_intersection (if a pick up is to be made)
    // Will be length zero if no delivery item is picked up at the start intersection
    std::vector<unsigned> pickUp_indices;
};


// This routine takes in a vector of N deliveries (pickUp, dropOff
// intersection pairs), another vector of M intersections that
// are legal start and end points for the path (depots), right and left turn
// penalties in seconds (see m3.h for details on turn penalties),
// and the truck_capacity in pounds.
//
// The first vector 'deliveries' gives the delivery information.  Each delivery
```

```
// in this vector has pickUp and dropOff intersection ids and the weight (also
// in pounds) of the delivery item. A delivery can only be dropped-off after
// the associated item has been picked-up.
//
// The second vector 'depots' gives the intersection ids of courier company
// depots containing trucks; you start at any one of these depots and end at
// any one of the depots.
//
// This routine returns a vector of CourierSubpath objects that form a delivery route.
// The CourierSubpath is as defined above. The first street segment id in the
// first subpath is connected to a depot intersection, and the last street
// segment id of the last subpath also connects to a depot intersection.  The
// route must traverse all the delivery intersections in an order that allows
// all deliveries to be made with the given truck capacity. Addionally, a package
// should not be dropped off if you haven't picked it up yet.
//
// The start_intersection of each subpath in the returned vector should be
// at least one of the following (a pick-up and/or drop-off can only happen at
// the start_intersection of a CourierSubpath object):
//      1- A start depot.
//      2- A pick-up location (and you must specify the indices of the picked
//                              up orders in pickup_indices)
//      3- A drop-off location.
//
// You can assume that N is always at least one, M is always at least one
// (i.e. both input vectors are non-empty), and truck_capacity is always greater
// or equal to zero.
//
// It is legal for the same intersection to appear multiple times in the pickUp
// or dropOff list (e.g. you might have two deliveries with a pickUp
// intersection id of #50). The same intersection can also appear as both a
// pickUp location and a dropOff location.
//
// If you have two pickUps to make at an intersection, traversing the
// intersection once is sufficient to pick up both packages, as long as the
// truck_capcity fits both of them and you properly set your pickup_indices in
// your courierSubpath.  One traversal of an intersection is sufficient to
// drop off all the (already picked up) packages that need to be dropped off at
// that intersection.
//
// Depots will never appear as pickUp or dropOff locations for deliveries.
//
// If no valid route to make *all* the deliveries exists, this routine must
// return an empty (size == 0) vector.
std::vector<CourierSubpath> traveling_courier(
    const std::vector<DeliveryInfo>& deliveries,
          const std::vector<unsigned>& depots,
    const float right_turn_penalty,
    const float left_turn_penalty,
    const float truck_capacity);
```

Listing 1: m4.h

We will always call your `load_map` function before calling `traveling_courier`. Your algorithm must return a valid path within **45 seconds** of *wall clock* time; if your code takes more than **45 seconds** we will consider it a failure for that test case.

You will always be given at least one delivery location and at least one depot, and you can assume that intersections in the depot vector will not appear as pick up or drop off locations in the deliveries vector. Note that it is also possible that no path to make all the deliveries exists; in this case you should return an empty (size == 0) vector.

Note that we are using *wall clock* time to measure when your program exceeds the time limit. The UG machines on which we measure your program have 4 cores (CPUs) and each core can execute 2 threads at a time using hyperthreading, so you can use multi-threading to reduce your wall clock time if you wish.

4 Grading

Your entire mark in this milestone will depend on the performance of your code. The unit tests for this milestone are arranged in increasing difficulty (problem size), with the smallest problems being 5 or less delivery locations, and the largest having 200 or more deliveries. For each size of problem, we will:

- Test that you can find and return a legal solution within the 45 second wall clock time limit.
- If you find a legal solution within the time limit, we will also evaluate its quality (travel time) and compare it to the quality of our reference solutions. You can see how your quality compares to our reference solutions and to the solutions of your classmates using the leaderboard web page described below.

Your grade will be determined from both of these components – some marks will be given for finding legal solutions, and other marks will depend on your solution quality. Some unit tests will be public, and others that are similar but use different intersections and/or different city maps will be run during final grading.

Note also that we have made some changes to the usual traveling salesman formulation, so while the ideas you find in the traveling salesman literature will be helpful, the exact code and algorithms required will not be the same.

5 Contest

In addition to the grades above, bonus marks and prizes are available for this lab. The top team will receive souvenir globes as trophies, and the teams with the 4 best solutions will receive movie pass gift cards plus **bonus marks** as shown below:

- 1st place: 4% bonus on total course mark
- 2nd place: 3% bonus on total course mark
- 3rd place: 2% bonus on total course mark

- 4th place: 1% bonus on total course mark

The quality of each team's solution will be assessed by computing the geometric average of the travel time for a set of test inputs, all of which will be fairly large (on the order of 60 or more deliveries). To qualify, a team must compute a legal solution for each test within the 45 second wall clock time limit (i.e. any failed test means the team cannot compete). If two teams tie on the average travel time metric, the tie will be broken by lowest geometric average wall clock time required.

An anonymized team id will be posted on your team wiki page. Each time you submit milestone 4 with `ece297submit 4` your average travel time score and anonymized id will be posted to the leaderboard web site at http://ug251.eecg.utoronto.ca/ece297s/contest_2019/ . This leaderboard will use data from the public test cases run by exercise and submit. However, the final contest results will also include private test cases that are similar to the public ones but use different intersections and/or cities to prevent any hard-coding or overtuning.