Homework #7

Due Wednesday, March 28, 11:59pm
(Late if submitted after 12:04am)

Collaboration: You may not look up solutions to these questions from any resources (e.g. previous years' solutions, the Internet, other textbooks, etc.). You may work on the problems with other students in this class according to the "empty hands" policy. The "empty hands" policy states that you must each tear up all notes or discard recorded material (e.g. code) created while discussing the problem with that student PRIOR to beginning to write up your solution. You may not discuss the problem again once either one of you has started writing up the final solution you will turn in. Before the due date and time, you may only discuss solutions/approaches with other students in the class, course staff, and peer tutors.

Submission: Submit source code file to CSHW.

Guidelines: For this assignment, you will be expected to follow the style guidelines discussed in class as well as documented in the "Coding Standards" document posted on Moodle. That means your variable and function names should be descriptive and properly formatted with respect to capitalization, all function implementations should be preceded with a comment explaining what the function does, your code should be consistently indented to enhance readability, constants should be declared to be immutable, you should not have any global variables, `main` should return a value, etc. When your code is compiled, it should not generate any warnings. Your code should use C++ functionality, not C functionality. Additionally, you should use the approach specified in the question even if you already know features of C++ that could be used in place of what is specified in the question.

Genetic Algorithms and the Travelling Salesperson Problem

Part 1: Randomized Search

The travelling-salesperson problem (TSP) is a problem that's very simple to formulate, and yet impossible to solve efficiently. As described in this Wikipedia page (https://en.wikipedia.org/wiki/Travelling_salesman_problem), TSP asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?" TSP has been proven to be NP-hard, meaning that no algorithm exists to always compute the shortest route faster than exponential time. But since TSP is an important problem with many applications (think for example of the path a soldering robot has to take across a printed circuit board), there exist many efficient algorithms that approximate a "good-enough" solution.

In this assignment, we'll write a program that searches for the best solution to the TSP problem among a set of randomized solutions. For simple inputs, even randomized search can do well enough. In the next assignment, we'll improve our search much further.

City locations

Look at the attached example TSV files. Each has a number of lines with two integers on each line (row). The numbers represent the X and Y coordinates (respectively) of each city the salesperson has to visit. So, for example, the file "linear.tsv", with the following coordinates:

```
1 1
5 1
15 1
```

Has three cities arranged on a horizontal line (in which case, the optimal TSP solution is trivial: start with one end of the line, go through <5,1>, next go through <15, 1>, and then finish by going back to the other end of the line <1,1>, with a total distance of 28 units).

The file "five.tsv" has five cities, which makes the optimal solution (shortest path) a little harder to guess.

You can visualize these cities on the plane using any graphing program, and I recommend you try `gnuplot` on the virtual machine[1]. Simply type "gnuplot" at the shell, and then the following command at the prompt:

```
gnuplot>  plot [0:20][0:25] "five.tsv" with points
  ^        ^          ^           ^          ^
prompt   command   range for axes  file    type of plot
```

You can also use "with linepoints" instead of "with points" to draw a path between all cities, in the order they appear in the file.

This path is not necessarily the shortest path. If you like, you can reorder lines in the file and experiment with plotting to try and find the shortest path (hint: the shortest total distance is: 31.5247 units for "five.tsv")

Finally, take a look at "challenge.tsv" and try to plot it yourself. This file has 50 cities, which makes it infeasible for any human or algorithm to find a provably optimal solution. We'll search for one randomly instead and see how close we get.

Program interface

The file "tsp.h" presents the interface (API) for almost all the functions you need to write for this assignment. You should read it carefully, but **never** modify it. Instead, you will write all your functions in a separate "tsp.cc" file that should include "tsp.h".

This file defines three new types for our use, with the keyword "`using`" that introduces an alias:

- `coord_t` represents a <x,y> pair of a single city's coordinates. You can access the elements of a `std::pair` using the `.first` and `.second` attributes. For example, look at this valid code:

```
coord_t loc = { 4,5 };
loc.first = 7;
loc.second = 3 + loc.first;
```

- `cities_t` represents the full set of cities, as a vector of locations (coordinate pairs).
- `permutation_t` represents a permutation on the original ordering of the cities. It is a vector of unique indices, with 0 standing for the first city. For example, the vector { 4, 3, 2, 1, 0 } represents all the cities in "five.tsv" in reverse order. We'll try out different random permutations of the city ordering (really, different paths across all cities) and see which one is shortest.

Assignment

1) **Read in the city description file**

Implement the function `read_cities`. It should be passed a file name in a `string` argument, read in the file line by line, and for each line, create a new city coordinate pair in a vector of types `cities_t`, which is eventually returned. You may assume the file is of valid format and all coordinates are positive.

2) **Measure the distance of a path**

Implement the function `total_path_distance` by iterating over the given cities **in the order of the given permutation** and accruing the total distance travelled. Then add the distance from the last city to the first to complete the cycle. The distance between any two cities is the Euclidean distance between their coordinates.

You can test your function manually, for example, like this (from a `main` function):

```
std::cout << total_path_distance(read_cities("five.tsv"), { 0, 1, 2, 3, 4 }) << "\n";
```

Or

```
std::cout << total_path_distance(read_cities("five.tsv"), { 3, 2, 4, 0, 1 }) << "\n";
```

[1] To ensure it's installed, go to the cs-virtualmachine folder and type "git pull" and "vagrant up --provision".

3) **Generate a random permutation**
Implement the function `random_permutation`. You'll have to figure out yourself a good algorithm to return a valid permutation (no skipped values, no gaps) and generate the vector of indices you'll have to return. Note that the indices must be all unique, and the ordering needs to be "very random" for the search to work well later. You can implement this any way you like, but please document your algorithm in the code.

You'll need random numbers to generate the permutation, which you can get from C++'s standard library. An example can be found here: http://www.cplusplus.com/reference/random/uniform_int_distribution/.

4) **Randomized search**
Write a new file called `randomized.cc` with a `main()` function that will perform the actual search. The `main()` function will read in the cities file "challenge.csv" (or any of the smaller ones while you're debugging it); it will iterate for a given number of iterations (say, 1,000,000); and each time would generate a new randomized ordering, compute the distance along the path, compare the permutation's distance to the previous shortest distance, and if shorter, update the shortest distance and continue. On every update to the shortest distance, `main()` should print the iteration number, followed by a space (or tab), and the total distance of the path so far. The output of randomized on "challenge.tsv" might look something like this:

```
./randomized
1    27245.5
3    26025.5
6    25397.5
7    24444.8
10    23643.1
15    22753.3
104    21638.6
253    21341.9
1513    20644.6
3014    19764.5
15824    19581.6
59094    19345.2
109372    19233.7
547158    18914.6
632615    18437.6
874257    17786.4
```

Note: To compile both files into one binary, include them both in the compilation, e.g.,
```
g++-7 -g -std=c++11 -Wall randomized.cc tsp.cc -o randomized
```

5) **Visualization**
It would be nice to visualize your results, both the best path you found, and how long it took you to find it. For the latter, redirect the output of your program to a file like so:
```
./randomized > speed.tsv
```

You could also just copy the output of `main()` and paste it into the "speed.tsv" file (with two columns, one for iteration number, one for shortest distance so far). Then plot it using:

```
gnuplot> plot "speed.tsv" with linepoints
```

You can save this plot to a file as follows:
```
gnuplot> set term gif
Terminal type set to 'gif'
Options are 'nocrop font "/usr/share/fonts/truetype/liberation/LiberationSans-
Regular.ttf,12" fontscale 1.0 size 640,480 '
gnuplot> set out "speed.gif"
gnuplot> plot "speed.tsv" w lp
```

Include "speed.gif" with your submission.

Next, implement the function `save_permuted_cities()` that writes a file with all the city coordinates **in the order defined in the given permutation**. Then, add a call to this function inside your `main()` function's loop, so that whenever a best (shortest) solution is found, it is saved to the file "shortest.tsv". Finally, plot the resulting file in `gnuplot` as follows, and include "shortest.gif" with your submission. (You can optionally connect the first and last cities with a line by copying the first city (line) in the file as the last one as well.)

```
gnuplot> set term gif
Terminal type set to 'gif'
Options are 'nocrop font "/usr/share/fonts/truetype/liberation/LiberationSans-
Regular.ttf,12" fontscale 1.0 size 640,480 '
gnuplot> set out "shortest.gif"
gnuplot> plot "shortest.tsv" w lp
```