PA2: Travelling Trojan (Due 9/20 @ 11:59PM)

The GitHub classroom link for this assignment is: https://classroom.github.com/a/DM_p-fY1.

In this lab, you will implement a genetic algorithm for determining the best tour of the landmarks in LA, starting and ending at LAX.

Functional Style
For this lab, you must (mostly) stick to a functional style. There will be some flexibility to the rules, but if the clear majority of your code is not following the functional style, there will be some points deducted.

Here are the main things we will be looking for:

- **No custom classes** – You should not declare your own classes for this lab. It's okay to use built-in ones, but don't make your own with member functions, etc. In fact, you don't even need to declare any structs. I've provided the two that you need at the top of TSP.h.
- **(Almost) no side effects** – Your functions should generally be side-effect free. Of course, as discussed in lecture there are two main exceptions. Any functions that read from the input file or write to the output file will need to have side effects. Also, any functions that use the random number generator will receive it by reference, and will modify the generator, which is also a side effect.
- **Effort to reduce iteration** – Try to avoid loops, and instead use things like `std::transform`, `std::accumulate`, `std::generate`, `std::adjacent_difference`. There are a few cases (like input/output) where using loops makes sense. But we want you to make an effort to use the above functions instead of loops.
- **Decomposition** – Try to avoid writing long functions

Command Line Parameters
As in the previous assignment, you need to parse command line arguments in ProcessCommandArgs in SrcMain.cpp. The arguments are as follows:

```
[program] inputfile popsize generations mutationchance seed
```

(Where argv[1] is input file, argv[2] is popsize, and so on). For example:

```
tsp-windows.exe input/locations.txt 8 5 10 1337
```

means that the input file is `input/locations.txt`, there is a population size of 8, there should be 5 generations, the mutation chance is 10%, and the starting seed value is 1337. *You may assume that your program will not receive invalid command line parameters*.

Output File
Your program should write its output to a single file called `log.txt`. ***Do not use any other file name***. The file comparison test cases will fail otherwise. We will ignore any console output, so feel free to write whatever you want to console. But your log file must follow the format in the instructions.

The sampleoutput directory contains the expected output for each test case. Because of platform differences, there are different output files for Mac, Linux, and PC.

We've also provided student test cases for each sample file. We've commented out all but the first one. Once you get the first one working, uncomment the others. If diff says there are discrepancies, that means your output does not match, and you need to fix the issue(s).

Source Files
In this lab, you should only edit TSP.h, TSP.cpp, SrcMain.cpp, and StudentTests.cpp. You should declare functions in TSP.h and implement them in TSP.cpp.

Your Own Test Cases
Part of your grade is the quality of your own test cases written in StudentTests.cpp. We want you to write one test case for every function you declare in TSP.h (a good number of functions for this assignment is around 10 in total).

Part 1: Setup and Generating the Initial Population

First, parse the command line parameters, and convert them to integers as needed. For the random number generator, construct a `std::mt19937` generator and pass in the seed integer as a parameter to the constructor.

Next, make a function that returns a `std::vector` of locations. This function should parse in the locations from the file specified in the command line parameter. For this function, there's no great way to avoid doing a loop, since you must check for EOF. Each line of the input file is in the following format:

```
Location,Latitude,Longitude
```

Now make function(s) to generate the initial random population. Since it uses the random number generator, you'll have to pass it by reference and there will be side effects.

The number of individuals in the population is based on the command line parameter. To ensure that your randomization mirrors the reference output, use the following procedure:

1. Create an initial vector that contains the sequential values from 0…n – 1, where n is the number of locations in the location file. Note that there is not an extra 0 at the end, because it's implied that you'll go back to the first location.
2. Use `std::shuffle` to shuffle from `begin() + 1` to `end()`. You use begin() + 1 because you don't want to displace the 0 at the beginning. ***Do not use*** `std::random_shuffle`.
3. Add this random shuffled vector to the population.
4. Repeat these steps for every member of the population. (You reset to the initial sequential vector to ensure that each shuffle is independent).

Once you've generated the initial population, output it to the `log.txt` file in the same format as in the reference files.

To test your code, run it with the parameters on the first page, since that is the smallest file and easiest to verify against the reference.

Part 2: Fitness

Create function(s) to compute the fitness of each member of the population. The output of the top-level function should be a vector of pairs. Each pair should have an integer corresponding to the individual in the population, and then a double that contains the fitness of that member. A pair is used so that when you later sort by fitness, you'll still know which member of the population it corresponds to.

For the fitness calculation, use the Haversine distance formula in the lecture notes. Make sure you use doubles for all the decimal numbers in your program, ***not floats***. Also, don't forget to convert from degrees to radians by using the `0.0174533` constant.

For computing the distance of the path, you could first use `std::adjacent_difference` to compute the Haversine distance between each stop, and then use `std::accumulate` to add them up. Don't forget to also add in the distance from the last location in the vector back to location 0.

Now output the fitness of the members to `log.txt` in the same format as the reference files.

Part 3: Selection

For selection (and also crossover), we will be using a lot of random values. So it's particularly important you implement this to specification, or your output will diverge from the reference files.

Write function(s) for the selection process. Your top-level function should return a vector of pairs, where each pair contains to integers. These integers are the parent A and parent B individuals selected. It is possible that some pairs contain the same two parents.

First, you need to generate a probability vector based on the fitness results:

1. Use `std::sort` to sort the fitness vector. You'll have to pass in a custom binary predicate function as the third parameter to sort, otherwise it won't sort by the fitness. Remember that you want the sorted vector to be in *ascending order*, because the "most fit" individual has the lowest score.
2. Create a vector of probabilities that each member of the population is selected. Initially, this should just be `1.0 / popsize` for each individual. This is a good opportunity to use `std::generate`. *Note that the indices in this vector should correspond to the individual's population index number* (so this most will likely not be sorted by fitness).
3. The two individuals with the highest fitness should have their probability multiplied by 6.0.

4. The remainder of the top half of the fit individuals (eg. from rank 2 to rank size / 2 – 1) should have their probability multiplied by 3.0.
5. Renormalize the probability vector to sum to 1.0 (use `std::accumulate` to compute the sum, then divide using `std::transform`).

Once you have a probability vector, you need to pick two random parents to form a pair:

1. Construct a `std::uniform_real_distribution<double>`
2. Generate a random double for the first parent, and use it to select an individual, from the probability vector (explained below).
3. Generate a random double for the second parent, and use it to select an individual from the probability vector
4. Repeat steps 1-3 for every other pair (make a new distribution each time)

The way we'll select from the probability vector is as follows. Suppose the probability vector for a population of 4 is as follows:

Individual	0	1	2	3
p	0.10	0.40	0.10	0.40

This means that individual 0 has a 10% chance of being selected, individual 1 a 40% chance, etc.

The algorithm for picking an index is simple. Starting at index 0, you perform a running sum of the probability. If at any index, the running sum becomes >= the random double, you select that individual.

For example, suppose the random double is 0.55. So:

- At index 0, running sum = 0.10, continue
- At index 1, running sum = 0.50, continue
- At index 2, running sum = 0.60, >= 0.55, so select index 2

So in the above example, the following ranges of random values would select each individual:

- Individual 0 = [0, 0.10]
- Individual 1 = (0.10, 0.50]
- Individual 2 = (0.50, 0.60]
- Individual 3 = (0.60, 1.00]

In any event, once you've generated the pairs, output them to the `log.txt` file in the format as in the reference files. Verify your implementation works correctly before moving on.

Part 4: Crossover, Generations, and Final Output

Now create function(s) to handle crossover. We will be implementing the crossover approach discussed in the lecture notes. Given a pair to crossover, follow these steps:

1. Use a `std::uniform_int_distribution<int>` from `1` to `size - 2`
2. Generate an integer from the distribution in (1). This is the crossover index
3. Use a `std::uniform_int_distribution<int>` from `0` to `1`.
4. Generate an integer from the distribution in (3). If it's a 1, parent A goes first. If it's a 0, parent B goes first.
5. Selected first parent copies all elements from beginning up to and including crossover index into child (hint: you can use `std::copy_n`)
6. Second parent will start at the beginning of its vector, and copy over all elements that don't already appear in the child (hints: you can use `std::copy_if`)
7. Create a `std::uniform_real_distribution<double>`
8. Generate a double from the distribution in (7). If this is <= the mutation chance, you should mutate:
    a. Create a `std::uniform_int_distribution<int>` from 1 to size - 1
    b. Use the distribution in (a) to select a random first index
    c. Use the distribution in (a) to select a random second index
    d. Use `std::swap` to swap the indices from (b) and (c).

Once you've crossover all of the pairs, this should return a new population. You should then output this new population into the `log.txt`. The generation number starts at 0 for the initial population.

If your code seems to work for the first population, now write a loop so it does the number of crossover generations as specified by the command line parameter (so if it's 5, you should end at generation 5). Hopefully, everything matches other than the final path output.

For the final path output, compute/display the fitness of the last generation and select the member with the highest fitness as the solution. Then just output the path (including the return to location 0 at the end), with the distance of the path, as in the reference file.

Grading Rubric

Part	**Points**
Graded Unit Tests	55
Your Test Cases Quality/Variety	25
Functional Programming Points	10
Code Quality (incl. following style guide)	10
Total	**100**