COMP 3522 Assignment #2: Using a genetic algorithm to approximate a solution to the traveling salesman problem

Object Oriented Programming 2

Due Dec 1 11:59pm NO EXCEPTIONS

You may work in groups of two (2) for this assignment. Create a team or join an existing team on github classroom. Make sure to include both your names and student numbers in comments in main.cpp

Introduction

Assignment 2 is ready at last! For this take-home coding project, you will produce an object oriented implementation of the traveling salesman problem using a genetic algorithm. Fun for all shall be had!

We will examine the traveling salesman problem. Suppose we have an unordered container of cities to visit. We want to sort and visit the cities in a sequence that minimizes the distance traveled. With a small number of cities this is trivial, but with a large number of cities this can take a very, very long time (what is the big O of this type of problem)? A genetic algorithm is one approach that makes this easier.

A genetic algorithm is an algorithm that draws inspiration from theories of natural selection. That is, we start with a 'population' of sample candidates, evaluate their fitness, perform some sort of cross-over and mutation, and continue until we have a solution that most closely meets our needs or meets our termination criteria.

1 Setup

Please complete the following:

1. No late submissions will be accepted for any reason.
2. Clone your repo using github classroom: https://classroom.github.com/a/g5mY2R0T
3. Fill out your **name** and **student number** at the top of **main.cpp**
4. Ensure you commit and push your work frequently. You will not earn full marks if you don't
5. Include a plaintext readme file which must include your full name on the first line and your student number on the second line. Leave line three blank.
6. On line four of your readme.txt file, write either "100% complete" or describe any outstanding issues.
7. Your program must execute when pressing the play button in CLion. Any tweaks the markers must make to run your code will result in lost marks.
8. You may work in pairs and submit your work together

2 Requirements

Your second take-home programming assignment is about the Traveling Salesman problem. Given a list of cities and their coordinates, what is the shortest possible route that visits each city and returns to the original city

The Traveling Salesman Problem is one of the most intensely studied problems in the field of optimization. While there are exact algorithms for finding the shortest route (including brute-force search, of course), calculating the solution can take years! We will explore a heuristic that finds a good solution, possibly a very good solution, in a reasonable amount of time: a genetic algorithm.

A genetic algorithm is an algorithm that draws inspiration from theories of natural selection. That is, we start with a 'population' of sample candidates, evaluate their fitness, perform some sort of crossover and mutation, and continue until we have a solution that most closely meets our needs or meets our termination criteria.

Start by reading the accompanying paper entitled Genetic Algorithms, A Survey. It's not a heavy read, and it's very interesting.

Your program must implement an object oriented solution to the following scenario:

1. A city has a name and a set of coordinates which we will call x and y because x and y are easier to type than longitude and latitude.
2. We will limit coordinates for this simulation to the range [0, 1000]. That is, x and y for all cities will be between 0 and 1000 inclusive.
3. A tour is what we will call a list of cities. A tour contains a list of all the cities in the simulation and a fitness rating. The fitness rating evaluates the distance the traveling salesman would need to travel to visit the cities in the order they appear in the tour.
4. The program will start by creating a group of cities. Ensure each city is assigned a unique name or sequence number and a random set of coordinates.
5. Create a population of tours. Each tour must contain the entire list of cities sorted randomly. That is, we will have a data structure that manages a collection of tours, and each tour will manage a sequence of the cities on the map that begins in a randomly shuffled state.
6. Determine and record the fitness of each tour. The fitness must be a double that represents the quality of the tour. Shorter tours are better quality, and will have better fitness. A good idea is to use the inverse of the total distance traveled, possibly multiplied by some scalar.
7. Make a note of the shortest, i.e., fittest, tour. This is our starting distance or base_distance. This is where our genetic algorithm starts.
8. Implement the genetic algorithm iteratively until we observe a predetermined improvement, i.e., until we see that base_distance / best_distance > IMPROVEMENT_FACTOR:
   (a) Selection: keep the best tour by moving the fittest to the front of the population. We won't change it in this iteration, and we will call it an 'elite' individual
   (b) Crossover: mix the rest of the routes and create new routes. Replace every other tour in the population with a new tour generated by crossing some parents. Choose each parents by selecting a subset of tours from the population to represent potential parents, and selecting the fittest from the subset. That is, Each parent is the fittest of a subset of size PARENT_POOL_SIZE of the population, randomly selected. We cross the NUMBER_OF_PARENTS parents to create a child tour, and keep doing this to replace all of the non-elite tours in the population.

      To cross two parents, select a random index and use the cities from one parent to populate the mixed tour up to and including that index, and then the cities from the second parent to top up the tour, making sure we don't add cities that are already in the mixed tour.

If a city in a following parent has already been added to the tour, simply move to the next city in the tour.

(c) Mutation: Randomly select 20 to 30 percent of your tours to mutate (excluding the elite tour!). Feel free to change the number of tours to mutate. Calculate a random mutation value for each city in a specified specified tour. If this value < MUTATION_RATE, then the city is swapped with the adjacent city from the same tour.

(d) Evaluation: assign each new tour a fitness level.

(e) Report: provide the user with information about the algorithm's progress.

9. Implement the Singleton and Facade design patterns by creating a **SingletonFacade** class to hide the complexity of running the genetic algorithm above. The main function in main.cpp should get an instance of the singleton and call a `run()` function to start the steps listed above.

10. Some constants and behaviors you may wish to consider include (but are not limited to):

(a) CITIES_IN_TOUR the number of cities we are using in each simulation, start with 32 but it would be nice if the user can choose

(b) POPULATION_SIZE the number of candidate tours in our population maybe 32 but it would be nice if the user can choose

(c) SHUFFLES the number of times a swap must be effected for a shuffle to finish if writing your own custom swap function. Maybe 64 is a good number. If using the standard library shuffle function, then 1 shuffle per tour is fine.

(d) ITERATIONS the maximum number of times the algorithm should iterate maybe 1000

(e) MAP_BOUNDARY the largest legal coordinate should be 1000

(f) PARENT_POOL_SIZE the number of members randomly selected from the population when choosing a parent, from which the fittest is made a 'parent' maybe 5 is a good number

(g) MUTATION_RATE is probably low like 15 percent but it would be nice if the user can choose

(h) NUMBER_OF_PARENTS the actual number of 'parent' tours crossed to generate each 'offspring' tour

(i) NUMBER_OF_ELITES should start at 1, but I am curious how changing this would modify the algorithm's effectiveness

(j) IMPROVEMENT_FACTOR is a percentage that indicates what percent the new elite fitness needs to improve over the base distance before exiting the algorithm loop

(k) shuffle_cities to shuffle the cities in a tour

(l) get_distance_between_cities to calculate the distance between two cities

(m) get_tour_distance reports the distance between the cities as they are listed in a tour

(n) determine_fitness determines the fitness of a tour

(o) select_parents will select the parents for a new tour from a population

(p) crossover creates a new tour from a given set of parent tours

(q) mutate may mutate a tour

(r) contains_city checks if a tour contains a specific city.

3 Expected Output

Your output should clearly indicate how the algorithm is improving every iteration. This includes but isn't limited to:

1. Iteration number

2. If new elite found
- Display "NEW ELITE FOUND: " and only the Elite's distance

   If no new Elite found

- Display the Elite's distance
- Display the Best Non Elite distance

3. Improvement over base so far

At the end, there should be a final report of the results that include:

1. Number of iterations
2. Report of the base and best distance
3. Whether your improvement factor was achieved
4. Output of the base route.
5. Output of the route taken to achieve the best distance.

4 Grading

This assignment will be marked out of 20. For full marks, you must:

1. (2 points) Commit and push to GitHub after each non-trivial change to your code
2. (8 points) Successfully write and test a program that implements the requirements using an object oriented solution
3. (6 points) Clear output indicating how the algorithms is improving as well as a final report of the results
4. (4 points) Write code that is commented and formatted correctly using good variable names, efficient design choices, atomic functions, thorough tests

You may work in pairs and submit your work together. Good luck, and have fun.

Sample output:

Original elite: Distance: 9017.88
(sausu->tsuyani->nawohu->hireyo->okin->sasona->kanre->wonose->eyuko->mukiwo->kot

suyu->mauhe->ikisa->muhaa->noyuko->ronori->tauri->erayu->shimeki->kowoki->sausu)

--- STARTING ALGORITHM ---

Iteration: 0

NEW ELITE FOUND:

Distance: 8598.85

Improvement over base: 1.04873

Iteration: 1

NEW ELITE FOUND:

Distance: 8323.25

Improvement over base: 1.08346

Iteration: 2

NEW ELITE FOUND:

Distance: 7372.14

Improvement over base: 1.22324

Iteration: 3

Elite distance: 7372.14
Best non-elite distance: 9088.85
Improvement over base: 1.22324

Iteration: 4
Elite distance: 7372.14
Best non-elite distance: 8406.17
Improvement over base: 1.22324

Iteration: 5
NEW ELITE FOUND:
Distance: 7029.23
Improvement over base: 1.28291

Iteration: 6
NEW ELITE FOUND:
Distance: 6512.82
Improvement over base: 1.38463

Iteration: 7
NEW ELITE FOUND:
Distance: 5815.84
Improvement over base: 1.55057

Iteration: 8
Elite distance: 5815.84
Best non-elite distance: 6864.05
Improvement over base: 1.55057

Iteration: 9
Elite distance: 5815.84
Best non-elite distance: 6436.06
Improvement over base: 1.55057

Iteration: 10
NEW ELITE FOUND:
Distance: 5815.07
Improvement over base: 1.55078

…

Iteration: 320
Elite distance: 4190.01
Best non-elite distance: 6150.27
Improvement over base: 2.15223

Iteration: 321
Elite distance: 4190.01
Best non-elite distance: 5772.07
Improvement over base: 2.15223

Iteration: 322
Elite distance: 4190.01
Best non-elite distance: 5339.8
Improvement over base: 2.15223

Iteration: 323
Elite distance: 4190.01
Best non-elite distance: 4772.55
Improvement over base: 2.15223

Iteration: 324
Elite distance: 4190.01
Best non-elite distance: 4524.76
Improvement over base: 2.15223

Iteration: 325
Elite distance: 4190.01
Best non-elite distance: 7039.32
Improvement over base: 2.15223

Iteration: 326
Elite distance: 4190.01
Best non-elite distance: 6084.58
Improvement over base: 2.15223

Iteration: 327

Elite distance: 4190.01

Best non-elite distance: 4999.41

Improvement over base: 2.15223

Iteration: 328

Elite distance: 4190.01

Best non-elite distance: 6839.36

Improvement over base: 2.15223

Iteration: 329

NEW ELITE FOUND:

Distance: 4118.16

Improvement over base: 2.18978

Iteration: 330

NEW ELITE FOUND:

Distance: 3743.19

Improvement over base: 2.40914

--- FINISHED ALGORITHM ---

Total iterations: 331

Original elite:

Distance: 9017.88
(sausu->tsuyani->nawohu->hireyo->okin->sasona->kanre->wonose->eyuko->mukiwo->kotsuyu->mauhe->iki
sa->muhaa->noyuko->ronori->tauri->erayu->shimeki->kowoki->sausu)

Best elite:

Distance: 3743.19
(ikisa->erayu->shimeki->noyuko->sausu->kotsuyu->ronori->hireyo->mauhe->okin->nawohu->eyuko->wono
se->mukiwo->muhaa->sasona->kanre->kowoki->tauri->tsuyani->ikisa)

Improvement factor reached!

Improvement factor: 2.40914