COMP 3512 Assignment #2: Using a genetic algorithm to approximate a solution to the travelling salesman problem

Christopher Thompson
cthompson98@bcit.ca

BCIT CST — due on Friday 23 November 2018 before 23:59:59 PM

Introduction

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

We will examine the travelling salesman problem. Suppose we have an unordered list of cities to visit. We want to sort and visit the cities in a sequence that minimizes the distance travelled. 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. Create a new project in CLion. Let's call it GeneticAlgorithm. Choose C++ Executable for the project type and ensure you select C++14 as the Language Standard.
2. Examine the project files that are created. Note there is a file called main.cpp that contains a main method stub. Remember that CLion uses cmake, which is a build tool similar to ant. Open the CMakeLists.txt file and ensure you set the correct compiler flags: -Wall -Wextra -pedantic -std=c++14 and so on...
3. Add this project to version control. From the VCS menu in CLion select Import into Version Control | Create Git Repository... to add the project to a repo.
4. Add the project to GitHub by returning to the VCS menu and selecting Import into Version Control | Share Project on GitHub. Call it GeneticAlgorithm, make sure it's private, and add a first commit comment. It's fine to add all the files for your first commit.
5. Visit GitHub and ensure your repository appears. Add me as a collaborator. In GitHub, I am known as chris-thompson. You'll recognize my avatar.
6. Ensure you commit and push your work frequently. You will not earn full marks if you don't.

2 Requirements

Your second take-home programming assignment is about the Travelling 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 Travelling 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 cross-over 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 travelling 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 travelled 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 best_distance / base_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 parent by selecting a subset of tours from the population to represent potential parents, and and then select 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. Do this NUMBER_OF_PARENTS times (maybe tours have more than two parents). We cross the NUMBER_OF_PARENTS parents to create a child tour, and keep doing this to replace all of the 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.

      To cross three parents, select two random indices, and use the cities from one parent to populate the mixed tour up to and including the first index, and then the cities from a second parent

to populate the mixed tour up to and including the second index, and then the cities from the third parents to finish the 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 mutate up some tours (excluding the elite tour!). Calculate a random mutation value for each city in a specified specified tour. If this value < MUTATION_RATE, then the city is swapped with a randomly chosen 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. Some constants and behaviours 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, maybe 64.

(d) ITERATIONS the minimum 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 5 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) shuffle_cities to shuffle the cities in a tour.

(k) get_distance_between_cities to calculate the as-the-crow-flies distance between two cities.

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

(m) determine_fitness determines the fitness of a tour.

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

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

(p) mutate to mutate a tour.

(q) contains_city checks if a tour contains a specified city.

3 Grading

This take-home assignment will be marked out of 10. For full marks this week, you must:

1. (2 points) Commit and push to GitHub after each non-trivial change to your code
2. (6 points) Successfully write and test a program that implements the requirements using an object oriented solution
3. (2 points) Write code that is commented and formatted correctly using good variable names, efficient design choices, atomic functions, thorough tests including a full suite of unit tests.

Good luck, and have fun.