A genetic approach to solving the traveling salesman problem

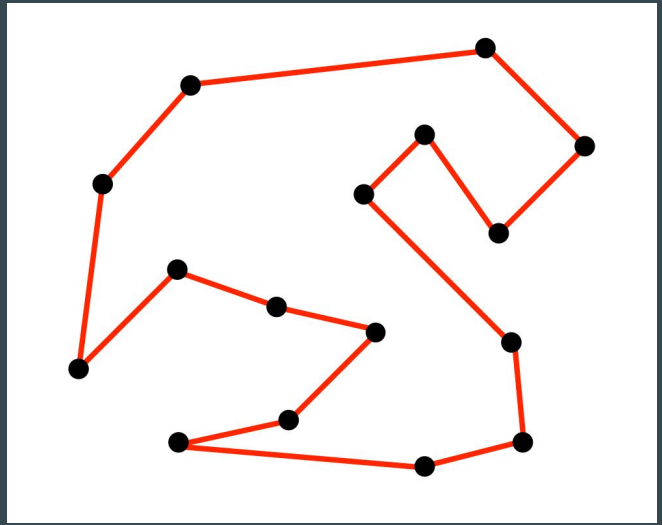
The traveling salesman problem (or TSP)

Lore behind the problem name:
A salesman needs to visit n *different cities, and wants to plan a tour in which he visits each city* ***exactly once*** *in the* ***most efficient*** *way possible.*

Abstract problem:
Find the (or one) shortest path going through each city/point exactly once and coming back to the starting point in the end

In graph terms:
Find a *Hamiltonian cycle* that minimizes total edge cost

-> the problem is **NP-hard**



A few traditional approaches for finding the optimal solution

- Greedy approach cannot guarantee global optimality (think about it for a few moments if you're not convinced)
- Naive brute force solution time complexity is O(n!): compute the cost of each possible path (n! possibilities) and keep only the best one
- There exists a DP solution that is $O(2^n * n^2)$

A basic genetic approach applied to the TSP

- *Individual* : a path going through each city exactly once then coming back to its starting point
  -> the path can be represented as an array of *n* integers, where the *i*-th integer corresponds to the ID (or index) of the (*i+1*)th city to visit in the path's order

- *Fitness* : the shorter the individual's path, the higher its fitness
  -> we can define fitness to be inversely proportional to the individual's path length

A basic genetic approach applied to the TSP

- *Crossover* : ‘fusion’ between two (or more) individuals, or paths in our case
  -> there are multiple way to do this, which one you choose is up to you
- *Mutation* : small modifications in the path of an individual
  -> same here, many possibilities !

... I’m being vague on purpose because making these choices and implementing them will be your job in a few moments ;)

A quick walkthrough of my implementation

...

... Because code written by someone else is always scary

Python project structure

- *parameters.py* : initializes problem parameters (e.g. number of cities to visit) and algorithm parameters (e.g. population size, mating pool proportion)
- *individual.py* : definition of the class *Individual*, which contains useful data & methods to handle individuals in our problem
- *functions.py* : where many core methods of our genetic algorithm are defined (other than individual-specific methods)
- *main.py* : the file you'll need to run, and where the structure of the algorithm is implemented. Take your time and read it slowly, it should not be too difficult to understand if you read the comments and understand the idea of a genetic algorithm

Cities

- Randomly generated on a discrete grid (their coordinates are integers on this grid) at the beginning of the program
- Stored in an array of tuples of ints
  -> ex : `[(5, 9), (12, 3)]` represents 2 cities whose coordinates are respectively (5, 9) and (12, 3) on the discrete grid
- Referred to with 0-based indexes when used in a path for example
  -> ex : `1` in the example above corresponds to the city at coordinates `(12, 3)`

Individuals

- The individual's path is an array of city indexes
  -> ex: `[2, 3, 1, 0]` means that the path starts from the 3rd city in the array of cities, then goes through the 4th, 2nd and 1st cities (indexes start at 0) in this order, and *finally comes back to the starting point*, that is to say the 3rd city
- Each one has a unique ID and a generation ID (mainly for printing purposes)
- An individual's fitness can be computed easily by using the distance matrix created at the beginning of the main (stores distances between each couple of cities so that you don't need to compute those Euclidean distances every time you evaluate an individual's fitness)

Genetic process implemented here: initialization

- A group of cities are randomly generated on a grid
- Distances between each couple of cities are pre-calculated to optimize fitness calculation
- A random population is initially generated by shuffling city indexes together
- Fitness of each individual is calculated

Genetic process implemented here: main loop

- Individuals of the current population are *ranked* according to their fitness
- Individuals are selected to be part of the *mating pool* according to their fitness
- Same goes for the *elite*; these individuals constitute the first individuals of our new generation
- Individuals in the mating pool are *bred* together to create newborn individuals
- Each of these newborn individuals have a fixed chance of being *mutated*
- Their *fitness* is computed and they are added to the new generation

Your job now: fill in the following parts of the code !

After you read through *main.py*, write the body of these functions (signaled with a TODO comment which should be colored if you use an IDE like JetBrains')

- `mutate` (*individual.py*): apply a random mutation to an individual (or more than one if you want)
- `elitistSelection` (*functions.py* like the rest of the following methods): pick the best individuals from the current generation
- `selectMatingPool`: choose mating pool members according to their ranking
- `crossover`: create a new individual by keeping characteristics of both parents

A few useful tools

- You can plot the cities on a 2D scatter plot using `plotCities` in *functions.py*
- You can also plot the path of any individual using the instance method `plot` in *individual.py*
- If you need a lighter way to get info about an individual (using plots is pretty slow and should not be done too often), simply call `print` on it

Slides: Emma Neiss for INSAlgo (March 2021)

Template: Recycled version of *Graph Theory*, Louis Sugy for INSAlgo

TSP graphical example :
https://algorist.com/problems/Traveling_Salesman_Problem.html

A comprehensive tutorial that helped me a lot : Evolution of a salesman: A complete genetic algorithm tutorial for Python