Alessandro Allegranzi

BU MET CS 767

Assignment 5 Genetic Algorithms

6/12/2024

Table of Content

[1. Representation of the Data 2](#_Toc168508830)

[2. Representation of a Route 3](#_Toc168508834)

[3. Crossover 3](#_Toc168508836)

[4. Mutation 4](#_Toc168508843)

[5. Result on the Given Data 5](#_Toc168508845)

[6. Result on Your Data 6](#_Toc168508849)

[7. Source Code 7](#_Toc168508851)

[8. Comments on Performance 7](#_Toc168508853)

[References 7](#_Toc168508855)

[Evaluation 8](#_Toc168508856)

[Appendix 1 8](#_Toc168508857)

[Appendix 2 11](#_Toc168508858)

6/12/2023

MET CS 767 Assignment 5: GA’s

*Alessandro Allegranzi*

You are to implement the *traveling salesman problem* using a genetic algorithm in the manner outlined below—preferably in Python, otherwise Java. We’ll call this project T-Order because it has a unique way to represent routes. You can assume that there is a route connecting every pair of cities, and that no two distances are equal. The latter simplifies coding, as you will see.

Eric Braude created this technique to perform trouble-free crossover for Traveling Salesman. It may be published elsewhere (although we have not located any such reference), and if so, please do not such work for this assignment.

The instructions are otherwise the same as in the previous assignments.

It is recommended that you build this application by modifying an AI-generated genetic algorithm traveling salesman application rather than building from scratch, which is onerous.

### >>AI generation used (or check: I did not use AI generation here \_X\_). Please collapse this.

PARAGRAPH DESCRIBING YOUR VALUE ADDED TO AI-GENERATED MATERIAL

Your response replaces this.

YOUR PROMPT SEQUENCE

[1] Your first prompt replaces this.

[2]

# Representation of the Data

## Taking into account how crossover below will operate (see Section 3), explain how T-Order will *represent* the data (e.g., “a list consisting of …”) in general.

## As an example, include showing how the following is to be represented when you use your representation:

## B(oston) to L(ondon) 3(k miles), L to M(umbai) 4.5, M to S(hanghai) 3.1, S to L 5.7, B to M 7.6, and B to S 7.8.

The above is only an example: your assignment should apply to any traveling salesman problem with unique node distances. You can refer to the nodes as cities if you wish.

The T-Order algorithm version I wrote represents the data as a dict of city pair keys and distance values as below. It seemed the simplest way to implement the “node” data. Please see an example below for the above list in the instructions.

distances = {

("Boston", "London"): 3.0,

("London", "Mumbai"): 4.5,

("Mumbai", "Shanghai”): 3.1,

("Shanghai", "London"): 5.7,

("Boston", "Mumbai"): 7.6,

("Boston", "Shanghai”): 7.8

}

I unfortunately did not read the requirements closely at first for the draft submission, so I had to re-format the data implementation. I originally set up the data to use x and y coordinates on a 2-D plane, based on reference [1]. Please see [appendix 1](#appendix1) more for more details on the original T Order algo I worked on.

# Representation of a Route

## Explain how T-Order will *represent* a route in general. Include, as an example based on the above example, how the following route will be represented: *B to L to M to S to B*.

The route will be a list of city names, ordered in the order the route will take.

['North Reading', 'Quincy', 'Boston', 'Needham', 'Arlington']

It will read like “North Reading to Quincy, Quincy to Boston, Boston to Needham, and Needham to Arlington,” and so forth.

Given the data example from question 1, it would be something like:

['Boston, 'Mumbai', 'London', 'Shanghai']

# Crossover

## Define a crossover function consistent with the following.

## T-Order should create a child from two parents by simple cuts, as in the example below. (Bold and italics are added to clarify what part of the child comes from what part of the parents.)

## Parent 1: Boston 🡪 2nd closest unvisited city[[1]](#footnote-1) 🡪 2nd closest unvisited city[[2]](#footnote-2) 🡪 closest unvisited city 🡪 Boston

## *Parent 2: Boston 🡪 closest unvisited city 🡪 2nd closest unvisited city 🡪 closest unvisited city 🡪 Boston*

## Child route from these parents—with crossover point at 1:

## Boston 🡪 2nd closest unvisited city *🡪 2nd closest unvisited city 🡪 closest unvisited city 🡪 Boston*

def crossover(parent1, parent2):

"""

This function implements the crossover operation. as described in the

Assignment it uses a single crossover point, randomly selected.

Args:

parent1 (list): first parent.

parent2 (list): second parent.

Returns:

child (list): child of the two parents.

"""

# Select a random crossover point.

crossover\_point = int(random.random() \* len(parent1))

child = parent1[:crossover\_point] # Copy the part of parent1 up to

the crossover point.

child += [city for city in parent2 if city not in child] # Append the

remaining cities from parent2.

return child

It would look like:

Parent 1: ['North Reading', 'Quincy', 'Boston', 'Needham', 'Arlington']

Parent 2: ['Quincy’, North Reading', 'Arlington', 'Boston', 'Needham']

Crossover point = index 1

Child: ['North Reading’, ‘Quincy’, 'Arlington', 'Boston', 'Needham']

# Mutation

## Explain (clearly) how your T-Order performs mutation.

Loosely based on, but simpler, than the example from Geeks for Geeks [2].

def mutate(individual, mutation\_rate):

"""

This function implements the mutation operation.

Args:

individual (list): individual to be mutated.

mutation\_rate (float): mutation rate.

Returns:

individual (list): mutated individual.

"""

for swapped in range(len(individual)):

if(random.random() < mutation\_rate):

swap\_with = int(random.random() \* len(individual))

city1 = individual[swapped]

city2 = individual[swap\_with]

individual[swapped] = city2

individual[swap\_with] = city1

return individual

1. The function iterates over each city in the individual (route).
2. For each city, it generates a random number between 0 and 1. If this number is less than the mutation rate, it proceeds with the mutation. This means that each city has a chance of being swapped with another city equal to the mutation rate.
3. If the mutation condition is met, it selects another random city in the route.
4. It then swaps the positions of the current city and the randomly selected city in the route.
5. This process is repeated for each city in the route, so multiple mutations can occur in a single call to mutate.
6. The function finally returns the (possibly mutated) individual.

# Result on the Given Data

## Describe the result from executing on the following example data …

## *B(oston) to L(ondon) 3(k miles), L to M(umbai) 4.5, M to S(hanghai) 3.1, S to L 5.7, B to M 7.6, and B to S 7.8*.

## What do you think of this result? Explain.

I ran the algorithm on the given example data with the given parameters:

# Define the size of the population

pop\_size = 100

# Define the number of elite individuals

elite\_size = 10

# Define the mutation rate

mutation\_rate = 0.01

# Define the number of generations

generations = 200

# Create city coordinates for a very small sample of 4 cities.

distances = {

("Boston", "London"): 3.0,

("London", "Mumbai"): 4.5,

("Mumbai", "Shanghai"): 3.1,

("Shanghai", "London"): 5.7,

("Boston", "Mumbai"): 7.6,

("Boston", "Shanghai"): 7.8

}

Given the small size of the city list, the algorithm found the optimal solution on the first generation and did not generate better routes on successive generations.

The best calculated route is:

1 ['London', 'Boston', 'Shanghai', 'Mumbai']

A graph with a line

Description automatically generated

You can see in the above graph the distance of the best route did not change over the successive generations because the best solution was already computed in the first run.

# Result on Your Data

## Describe the result from executing your application on illustrative data of your choice. What do you think of the result? Explain.

I set up a larger test to create 30 random cities with random distances (with nonsense, random names), and the algo did improve the results over successive generations.

# Create city coords for a larger sample to better test the algorithm.

# Function to generate a random city name

def generate\_city\_name():

return ''.join(random.choices(string.ascii\_uppercase, k=5))

# Generate 30 city names

city\_names = [generate\_city\_name() for \_ in range(30)]

distances = {}

for i in range(len(city\_names)):

for j in range(i+1, len(city\_names)):

city1 = city\_names[i]

city2 = city\_names[j]

distance = round(random.uniform(1, 20), 1) # generate a random distance between 1 and 20

distances[(city1, city2)] = distance

The best calculated route is:

1 ['MRSUR', 'JPJZS', 'MUOJZ', 'MUVAH', 'XPFQU', 'ZMNSR', 'ZNHXQ', 'ERWZA', 'CWNAC', 'NUGLM', 'JUILX', 'XIEWV', 'VDLNW', 'MTQXE', 'MLGPA', 'AQMXF', 'TXGPA', 'FSJBZ', 'GGDXZ', 'GEIPU', 'JBDSZ', 'XDSZZ', 'ZFHJS', 'KQYCB', 'KTWEW', 'BVVAN', 'PKCJL', 'KQKRP', 'LHHCQ', 'MEFPR']

A graph with blue lines

Description automatically generated

You can see in the above graph the algorithm produced the best solution around gen 85, before ticking up a bit, likely due to crossover or mutations. Genetic algorithms do not guarantee the best possible solutions [1]. The result was quite good though by the time the final 200th generation was completed.

# Source Code

## Paste your source code below—or refer to an appendix. It should accompany this doc as well.

Colab Notebook:

<https://colab.research.google.com/drive/11c4z2AAY5ral1F6z7J6W6Gw2hQcaDAuV?usp=sharing>

# Comments on Performance

## Compare the performance of T-Order with at least one known Traveling Salesman GA implementation.

Generally, it is difficult to estimate the time complexity of a genetic algorithm [1]. The speed of the algorithm depends on population size, number of generations, etc. Mutation rate also has the potential to increase run time if we alter enough routes in each generation.

The T-Order algorithm implemented for this assignment uses the python sorted() library, which is used in the fitness function (rank\_routes) and runs in O (n log n) time, n being the population size. Other library methods, like random.sampe(), which has a time complexity of O(k), where k is the number of elements, run even faster. Very generally, the T-Order algorithm implemented here has a time complexity of O (n log n).

As a comparison, the Geeks for Geeks implementation has a time complexity of O(n^2) as it uses a nested loop in the fitness function [2].

# References

Show that you used a wide variety of resources by listing them below and clearly indicating in the body above where you used. Make sure to use proper referencing in your paper. We suggest using APA format, but other formats are fine as long as they clearly distinguish your work from work of others in your response. In general, observe the stated plagiarism rules.

[1] Shendy, Ramez. “Traveling Salesman Problem (TSP) using Genetic Algorithm (Python)”. *Medium*. <https://medium.com/aimonks/traveling-salesman-problem-tsp-using-genetic-algorithm-fea640713758>. Aug 5, 2023.

[2] “Traveling Salesman Problem Using Genetic Algorithm.” *Geeks For Geeks*. <https://www.geeksforgeeks.org/traveling-salesman-problem-using-genetic-algorithm/>. April 30, 2024

# Evaluation



# Appendix 1

Results for algorithm version using coordinates:

Colab Notebook: <https://colab.research.google.com/drive/19RuwMJqgCGTB-pZRZxuNOhgvOAgTId0h?usp=sharing>

My original implementation had the data represented by a python dictionary consisting of a city name as key, and then x, y coordinates as value. Based on reference [1]. An example below:

x = [12, 18, 20, 22, 25]

y = [15, 18, 20, 22, 25]

cities\_names = ["Boston", "Quincy", "North Reading", "Arlington", "Needham"]

city\_coords = dict(zip(cities\_names, zip(x, y)))

**Data**

The actual dict would look like:

city\_coords = {

"Boston": (12, 15),

"Quincy": (18, 18),

"North Reading": (20, 20),

"Arlington": (22, 22),

"Needham": (25, 25)

}

**Results**

I initially ran the algo on a very small list of 5 cities. The input looked like:

city\_coords = {

"Boston": (12, 15),

"Quincy": (18, 18),

"North Reading": (20, 20),

"Arlington": (22, 22),

"Needham": (25, 25)

}

The algo immediately found the best route in the first gen and didn’t really improve upon results after that.

The best calculated route is:

1 ['Boston', 'Quincy', 'North Reading', 'Arlington', 'Needham']

A graph with a line

Description automatically generated

Visualization of route:

A graph with blue lines and red text

Description automatically generated

I then set up a larger test to create 20 random cities with random distances, and the algo did improve the results over successive generations, although it did not need the full 200 gens to find the optimal solution.

cities\_names = [f"city\_{i}" for i in range(1, 21)]

x = [random.randint(1, 100) for \_ in range(20)]

y = [random.randint(1, 100) for \_ in range(20)]

city\_coords = dict(zip(cities\_names, zip(x, y)))

The best calculated route is:

1 ['city\_19', 'city\_17', 'city\_1', 'city\_9', 'city\_18', 'city\_7', 'city\_16', 'city\_6', 'city\_12', 'city\_20', 'city\_5', 'city\_15', 'city\_14', 'city\_4', 'city\_3', 'city\_8', 'city\_13', 'city\_10', 'city\_11', 'city\_2']

A graph with blue lines

Description automatically generated

Visualization of the best route:

A graph with lines and numbers

Description automatically generated

# Appendix 2

1. i.e., compared to all other direct hops from the city just visited (Boston in this case) [↑](#footnote-ref-1)
2. i.e., compared to all other direct hops from the city just visited [↑](#footnote-ref-2)