Lab 2 - Travel Salesman Problem (TSP)

Repository overview

The solution for this problem is implemented in the section "Solution".

The folder cities contains the cities for each instance of the problem (not here).

Official solution

The solution is given in two different versions:

- (pure deterministic) greedy, where the salesman goes each time to the nearest city
- evolutionary, starting from random greedy: here, a GA approach is used, starting from a population produced by a greedy implementation, with some randomness

Greedy

The greedy algorithm is based on an iteration over all the cities, where each one is visited at each iteration: starting from a given city, the closest one is selected to be visited at the next step.

However, this approach is very good at optimizing the distance between pair of cities in the forward trip, but not at minimizing the distance between the last city and the starting one. For this reason, the base algorithm is executed iteratively on all cities in the problem space: in this way, the backward edge changes among executions and the penalty it introduced is in some way minimized by picking the best starting city.

Evolutionary

The evolutionary algorithm is implemented using a GA approach:

- representation: integer, one-time-items, where each gene is a city, represented by its index
- mutation: inversion mutation
- recombination: inver-over crossover, to preserve traits in the edges, since this TSP is modeled by a DAG, where only adjacent cities matter
- parent selection: fitness-proportional, by randomly selecting the best individual among 10 candidates
- survivor selection: deterministic, fitness-based
- population model: steady state, with offspring size equal to $\frac{2}{3}$ of population summarized

Initialization

The initialization plays a relevant role here, since it is observed that the final result has a strong dependency with the technique used.

Here a randomized greedy is used: it follows the same approach of the algorithm used in the greedy solution, but here the k-th nearest city is chosen with a probability of $9 * {10}^{-(1+k)}$, where k starts from 0. This operation is repeated starting from all cities, so the cardinality of the population is equal to the number of cities.

Therefore, the individual of the first generation represent something slightly worse than possible local optima, allowing the evolutionary algorithm to explore the state space without missing better solutions.

It is possible to have a number of initial greedy solutions different by the desired population size:

- if larger, they are randomly sampled without replacement and using a uniform distribution
- if smaller, the population is extended with random individuals

Evaluation

The evaluation is performed as follows:

- cost of the solution, in terms of distance covered by the Hamiltonian cycle, expressed in kilometers (km)
- number of calls to the cost function to find out the best solution: in the greedy approach it is equal to the number of cities, whereas it is smaller than the number of generations in the evolutionary approach

For each instance (except for Chinese one), a reference value is provided: it comes from Wolfram tool and it is expected to be nearly optimal (even if not absolute).

Collaborations

The following parts:

- choice of initial population
- basic idea behind greedy strategy

have been done in collaboration with Vincenzo Avantaggiato s323112.

Results

The results are summarized in the following table:

Instance name	Greedy cost	Greedy calls	EA cost	EA calls	Best result
Vanuatu	1475.528	8	1345.545	3	1345.54
Italy	4436.032	46	4263.110	898	4172.76
Russia	40051.587	167	34216.317	1317	32722.5
US	46244.333	326	40047.079	1997	39016.62
China	62116.045	726	55240.404	4991	-

Notes::

- the columns *Greedy/EA calls*: displays the number of generations necessary to the algorithm to find the (local) optimal solution;
- the columns *Greedy/EA cost*: displays the cost of the solution found, as absolute value (opposite of fitness, conceptually);
- the results are collected among several tries, only the best for each instance is reported.

Observations

The evolutionary approach used is sub-optimal (wrt the Wolfram result), but it can be executed in few minutes at the maximum. The greedy solution is worse than the evolutionary one, but its execution time is in the order of the seconds.

The instances **USA** and **China** could be significantly improved by increasing the number of generations, but it has been avoided due to limitations of computational resources.

Particular patterns

In the instance **Vanuatu**, the optimal solution is reached after few steps, due to the reduced dimension of the state (and problem) space. In the instance **Italy**, the techniques adopted are not able to improve the fitness after few hundreds generations, despite having reached a good solution. This happened quite often, but for some reason there are random configurations that allow to find better solutions.

Lab 2 - Travelling Salesman Problem (TSP) solution

Imports

```
import functools
import pandas as pd
import numpy as np
import geopy.distance
import matplotlib.pyplot as plt
from tqdm.auto import tqdm
from dataclasses import dataclass
```

General constants

```
PATH = "cities/"
INSTANCES = [
    "vanuatu.csv",
    "italy.csv",
    "russia.csv",
    "us.csv",
    "china.csv"
]
BEST_RESULTS = [
    -1_345.54,
    -4_172.76,
    -32_722.5,
    -39_016.62,
    None
]
```

Classes

```
class City:

    @staticmethod
    def distance(start, end):
        return geopy.distance.geodesic(
            (start.lat, start.lon), (end.lat, end.lon)
        ).km

    def __init__(self, name, lat, lon):
        self.name: str = name
        self.lat: float | np.float64 = lat
        self.lon: float | np.float64 = lon

    def __repr__(self):
        return f"{self.name}"

    def __str__(self):
        return f"{self.name} ({self.lat}°, {self.lon}°)"
```

```
@dataclass
class Individual:
    genome: np.ndarray
    fitness: np.float64 | float = None
```

Helper functions

```
def distance_matrix(coordinates: list) -> np.ndarray:
    num_cities = len(coordinates)
    dist_matrix = np.zeros((num_cities, num_cities))

    for i in range(num_cities):
        for j in range(i+1):
            dist_matrix[i, j] = dist_matrix[j, i] = City.distance(coordinates[i],
coordinates[j]) if i != j else 0

    return dist_matrix


def counter(fn):
    """Simple decorator for counting number of calls"""

    @functools.wraps(fn)
    def helper(*args, **kargs):
        helper.calls += 1
        return fn(*args, **kargs)

    helper.calls = 0
    return helper
```

```
@counter
def cost(solution: np.ndarray, dist_matrix: np.ndarray) -> np.float64 | float:
    """Cost of a cycle"""
    return np.sum(
        np.array([
            dist_matrix[start, end] for (start, end) in zip(solution[:-1],
solution[1:])
        ])
    )


def fitness(individual, dist_matrix) -> np.float64 | float:
    solution = np.array(individual.genome.tolist() + [individual.genome[0]])
    return -cost(solution, dist_matrix)
```

Parent selection

```
def parent_selection(population) -> Individual:
    BUCKET_SIZE = 10
    candidates = sorted(np.random.choice(population, BUCKET_SIZE), key=lambda e:
e.fitness, reverse=True)
    return candidates[0]
```

Crossover

```
def cycle_xover(p1: Individual, p2: Individual) -> Individual:

    num_cities = p1.genome.size
    genome = p1.genome.copy()

    a, b = np.random.randint(num_cities-1), np.random.randint(num_cities-1)
    l1, l2 = min(a, b), max(a, b)
    segment = p1.genome[l1:l2+1]

    genome[l1:l2+1] = p1.genome[l1:l2+1]
    others = p2.genome[~np.isin(p2.genome, segment)]

    assert (len(others) - l1) == (len(genome) - (l2+1))

    genome[:l1] = others[:l1]
    genome[l2+1:] = others[l1:]

    return Individual(genome)


def inver_over_xover(p1: Individual, p2: Individual) -> Individual:
```

```
    """INVER-OVER crossover"""
    genome1 = p1.genome.copy()
    num_cities = p1.genome.size
    genome = np.zeros(num_cities, dtype=np.int16)

    p1_start = np.random.randint(num_cities-1)
    genome1 = np.roll(genome1, -p1_start)

    p2_start = p2.genome.tolist().index(genome1[0])
    p1_end = genome1.tolist().index(p2.genome[(p2_start+1) % num_cities])

    genome[0], genome[1] = genome1[0], genome1[p1_end]

    genome[2:p1_end+1], genome[p1_end+1:] = genome1[1:p1_end][::-1],
genome1[p1_end+1:]
    genome = np.roll(genome, p1_start)

    return Individual(genome)
```

Mutation

```
def scramble_mutation(p: Individual) -> Individual:
    SIGMA = 0.5
    genome = p.genome.copy()
    mask = np.random.random(len(genome)) < SIGMA
    genome[mask] = np.random.permutation(genome[mask])
    return Individual(genome)


def inversion_mutation(p: Individual) -> Individual:
    a, b = np.random.randint(0, p.genome.size-1), np.random.randint(0,
p.genome.size-1)
    l1, l2 = min(a, b), max(a, b)

    genome = p.genome.copy()
    genome = np.roll(genome, -l1)
    genome[:l2-l1+1] = genome[:l2-l1+1][::-1]

    return Individual(genome)


def swap_mutation(p: Individual) -> Individual:
    l1, l2 = map(np.random.randint, [0,0], [p.genome.size, p.genome.size])
    genome = p.genome.copy()
    genome[l1], genome[l2] = genome[l2], genome[l1]
    return Individual(genome)
```

Operators selection

```
xover = inver_over_xover
mutation = inversion_mutation
```

Greedy

```
def greedy_solve(coordinates, dist_matrix, city, rnd):
    """Greedy algorithm with random initialization: sub-optimal"""
    temp = dist_matrix.copy()

    num_cities = len(coordinates)
    visited = np.full(num_cities, False)

    solution = -np.ones(num_cities+1, dtype=np.int16)
    solution[0], visited[0] = city, True
    for step in range(num_cities-1):
        temp[:, city] = np.inf

        selected = 0
        sorted_indexes = np.argsort(temp[city])
        while rnd and np.random.rand() < 0.2 and selected < num_cities - step - 2:
            selected += 1

        city = sorted_indexes[selected]

        solution[step+1] = city
        visited[city] = True

    solution[-1] = solution[0]
    #print(solution[:-1])
    assert set(solution[:-1]) == set(range(num_cities))

    return solution, -cost(solution, dist_matrix)
```

```
def greedy_init(coordinates, dist_matrix, rnd=False):

    best_fitness, best_sol = -np.inf, None
    solutions =  []
    for start in tqdm(range(coordinates.size)):
        sol, fitness_sol = greedy_solve(coordinates, dist_matrix, start, rnd)
        solutions.append(sol[:-1])

        if fitness_sol > best_fitness:
            best_fitness, best_sol = fitness_sol, sol

    assert best_sol is not None

    return solutions, best_fitness, best_sol
```

Evolutionary

```
def single_mutation(p: Individual):
    from_pos = np.random.randint(p.genome.size)
    to_pos = np.random.randint(p.genome.size)
    genome = p.genome.copy()
    genome[to_pos], genome[from_pos] = p.genome[from_pos], p.genome[to_pos]
    return Individual(genome)
```

```
def evolutionary_solve(coordinates, dist_matrix: np.ndarray, start_population,
pop_size, max_generations):

    OFFSPRING_SIZE = int(2*pop_size / 3)

    start_pop_len = len(start_population)
    num_cities = len(coordinates)
    population = [Individual(start_individual) for start_individual in
start_population]

    for i in population:
        i.fitness = fitness(i, dist_matrix)

    assert len(population) == start_pop_len and start_pop_len >= 0

    # Discard some individuals if too many wrt start ones
    if start_pop_len > pop_size:
        population = np.random.choice(population, size=pop_size,
replace=False).tolist()
        np.random.shuffle(population)

    # Extend population if needed more individuals than start population
    elif start_pop_len < pop_size:
        population.extend(
            [Individual(np.random.permutation(num_cities)) for _ in range(pop_size
- start_pop_len)]
        )

        # Compute fitness for new individuals
        for i in population[start_pop_len:]:
            i.fitness = fitness(i, dist_matrix)

    champions = [max(population, key=lambda i: i.fitness).fitness]

    for _ in tqdm(range(max_generations)):
        offspring = []
        for _ in range(OFFSPRING_SIZE):
            if np.random.random() < 0.1:
                p = parent_selection(population)
                o = mutation(p)
            else:
```

```
                p1 = parent_selection(population)
                p2 = parent_selection(population)
                o = xover(p1, p2)

            offspring.append(o)

        for i in offspring:
            i.fitness = fitness(i, dist_matrix)

        population.extend(offspring)

        # Elitism + generational model
        # population.sort(key=lambda i: i.fitness, reverse=True)
        # population = population[:RETAIN_SIZE_ELITIST]

        # Survivor selection
        population.sort(key=lambda i: i.fitness, reverse=True)
        population = population[:pop_size]

        champions.append(population[0].fitness)

    return population[0].genome, population[0].fitness, champions
```

Solver

```
MAX_GENERATIONS = [100, 1000, 2000, 2000, 5000]
POPULATION_SIZES = [100, 100, 200, 200, 200]
```

```
def solve(PATH, INSTANCES):
    for (INSTANCE, BEST_RESULT, MAX_GEN, POP_SIZE) in list(zip(INSTANCES,
BEST_RESULTS, MAX_GENERATIONS, POPULATION_SIZES)):

        print(f"Instance {INSTANCE}")

        cities = pd.read_csv(f"{PATH}{INSTANCE}", header=None, names=["name",
"lat", "lon"])

        coordinates = np.array([City(city.name, city.lat, city.lon) for city in
cities.itertuples()])
        dist_matrix = distance_matrix(coordinates)

        _, fitness_greedy, _ = greedy_init(coordinates, dist_matrix, rnd=False)
        greedy_solutions, _, _ = greedy_init(coordinates, dist_matrix, rnd=True)
        calls_greedy = len(coordinates)

        _, fitness_ea, champions = evolutionary_solve(coordinates, dist_matrix,
greedy_solutions, POP_SIZE, MAX_GEN)
        best_ea = max(champions)
```

```
        plt.figure(figsize=(14,8))
        plt.plot(champions, color="blue")
        plt.scatter(range(len(champions)), champions, marker=".", color="blue")
        plt.hlines(fitness_greedy, xmin=0, xmax=len(champions), linestyles="-",
color="red")
        if BEST_RESULT is not None:
            plt.hlines(BEST_RESULT, xmin=0, xmax=len(champions), linestyles="-",
color="darkgreen")
        plt.show()

        print(f"Greedy solution: {fitness_greedy:.3f}\nCost calls:
{calls_greedy}")
        print(f"EA solution: {fitness_ea:.3f}")
        print(f"Best solution: {F'{BEST_RESULT:.3f}' if BEST_RESULT is not None
else '-'}")
        print(f"Number of steps: {champions.index(best_ea)}")
```

```
solve(PATH, INSTANCES)
```

Lab 2 - Travelling Salesman Problem (TSP) - Issues done

To: Martina Plumari

Greedy

The simple nearest-neighbor policy is here enhanced with 2-opt strategy, that is specifically referred to solve the TSP in literature. In fact, the results obtained are significantly better than a simple greedy algorithm: it is evident how the possibility of transforming crossing routes into more simple ones leads to shorter paths.

The "version 2" removes the constraint of the maximum number of iterations, basing its stop condition only on reaching the steady state: in this way, it is possible to stop the algorithm only when no improvements are made, for a given number of tries. Indeed, this strategy finds slightly better results than the previous one.

Evolutionary algorithm

This algorithm starts from greedy solution, by:

- accepting clones if "version 1" is used
- accepting (probably) similar individuals, as if they start from the same city, if "version 2" is used Note : here it could be beneficial (to increase diversity) to extend greedy solutions with random ones, instead of accepting copies or similarities (case of Vanuatu and Italy), or discard some of them, either with random or deterministic criteria (other instances).

Using inver over crossover and inversion mutation is highly beneficial, whereas insert mutation allows achieving good results as well. It may be possible that using "version 2" solutions as initial population could

have lead to better general results, due to higher randomness, but some other parameters have changed, so it is not possible to give a clear interpretation of it.

Moreover, adding STEADY_STATE allows saving a lot of time, but setting the value at 300 could lead to miss some improvements; anyway, they would be small for the resources we have, so I think this strategy is efficient enough; the mutation probability does not seem to influence so much the final result (basing on what I have seen in my exercise, too), indeed adaptive probability does not help to improve the result.

Finally, adapt the number of generations to the size of the problem (that is the number of cities) is highly beneficial, to:

- avoid wasting time in the smaller instances
- find good results in the larger instances, by running the algorithm for more generations.

Final note : thank you for adding comments and explanations, hoping always for this clarity!

To: Adriano De Cesare

First issue:

- there is no documentation
- code is not commented
- sometimes variables does not convey their meaning, which could be useful in this case
- there are no results shown, except for Vanuatu instance with greedy approach

Evolutionary

The evolutionary strategy follows a modern GA flow, with mutation performed on the offspring with a given probability, and crossover always executed: this may be lead to worse results than following a hyper-modern flow. Genetic operators:

- inversion mutation has shown to work very good in solving this problem
- partially mapped crossover preserves a good quantity of information about the edges, but not as much as other methods (such as, inver-over crossover); anyway, I think it would crash if parent1 and parent2 are NumPy arrays (as in this case) due to attempt to broadcast the sum with different-shaped arrays.

Greedy

The standard solution is realized with a nearest-neighbor policy: despite not seeing any result here (except for Vanuatu instance), it works quite well considering its simplicity. Unfortunately, I am not able to interpret the other greedy approach (startgreedy2) and its auxiliary functions, since the code is too obscure and without any explaining comment or documentation.