MET CS 767 Assignment 5: Genetic Algorithms

Brendan Torok

1. Representation of the Data

TravOrder represents each city label in a list. A dictionary of distances between each city is also maintained in TravOrder as an undirected graph meaning that travel can occur in both ways between city pairs and the distance is the same regardless of the direction. TravOrder also precomputes for each city a list of other reachable cities sorted by increasing distance. These three data structures are referenced as the algorithm progresses. The unvisited cities are then added to a separate list of visited cities and removed from the unvisited cities list. It is required that every city is reachable by every other city with this implementation.

Example of the dictionary: dist[(“B”, “L”)] = 3.0, dist[(“L”, “M”)] = 4.5

Example of sorted nearest neighbors for city B:

neighbors[“B”] = [(“L”, 3.0), (“M”, 7.6), (“S”, 7.8)]

Now the second closest city is represented as neighbors[“B”][1].

2. Representation of a Route

A route in TravOrder is not represented by a list of city names, but instead a sequence of ranked nearest neighbors. The complete route is represented by the start city, then a list of integers where each integer represents a ranking of the k-th closest unvisited city from the current city. The final city in the representation is the start city since the traveling salesman always ends the route in the starting city.

For the ranks, integers must be greater than 1, where 1 is the closest unvisited city, 2 is the 2nd closest unvisited city, and so on.

The route B to L to M to S to B is represented as: start, [1, 1, 1], start. In a different way of representing it: Boston -> 1st closest unvisited city -> 1st closest unvisited city -> 1st closest unvisited city -> Boston

3. Crossover

```python
import random
def crossover(parent1, parent2):
    # ensure the parent lengths are the same for crossover
    if len(parent1) != len(parent2):
        raise ValueError("Parents must have the same length for crossover")

    # get the length of the chromosome
    n = len(parent1)
```

```
    # randomly choose the cut point for the crossover between 1 and n - 1
    # to avoid 0 and n cases where there is no crossover and 100% of genes from one
parent are chosen
    cut_point = random.randint(1, n - 1)

    # complete the cut and crossover in one step with simple list splicing
    child = parent1[:cut_point] + parent2[cut_point:]
    return child
```

4. Mutation

In my implementation a mutation can occur to one rank per child. The probability of mutations is set as a parameter for each child. This means that not every child will have a mutation that occurs.

The function takes a gene and generates a random number, if the randomly generated number is greater than the mutation probability, then a mutation will not occur. If it is less than the probability, then the mutation function continues. If a mutation is to occur, the length of the gene is calculated and a random position in the gene is then chosen to mutate. The mutation occurs by selecting a random integer between 1 and the length of the gene, representing the new rank value. This bound is chosen because it is impossible for the rank 0 to be given as there is no 0^{th} closest neighbor. Additionally, it is not possible for the rank to be greater than the number of cities available in the cities list. When the new generated rank value is different from the current rank, the current rank is replaced with the new mutated value and the final mutated gene is returned. This function does not need to ensure the new rank is selectable because the way trav_order is implemented allows for ranks higher than what currently exists to wrap around by using a modulo function.

5. Result on the Given Data

My final result on the given data is the following:

```
Best gene:  [3, 1, 2]
Best route:  ['B', 'S', 'M', 'L', 'B']
Best distance:  18.4
Boston -> 3rd closest unvisited city -> 1st closest unvisited city -> 2nd closest
unvisited city -> Boston
```

This result looks to be accurate in both how the route is represented, constructed, and calculated. However this is not any different from the result when using a simple greedy algorithm which always chooses the closest unvisited city. This suggests that for such a small dataset, it is likely unnecessary to implement a genetic algorithm for the traveling salesman problem. However, if a dataset becomes much larger and there is a significant increase in the number of cities, it is possible that the traveling salesman algorithm with the genetic algorithm would improve the final result.

6. Result on Your Data

I scaled the dataset up to 20 unique cities. The dictionary for this is constructed by creating a set of used distances, then generating a city pair distance between 1 and 100 and checking if the

distance is already used. If it is not already used, then it is added to the dictionary and the used distances set is updated. The result of my code correctly traverses all cities and returns a best distance route. When comparing the results to a simple greedy algorithm it was able to get a better result, however when running the algorithm multiple times there were occasions where the result was significantly worse.

Result:

```
Best gene:  [8, 5, 18, 1, 5, 2, 3, 15, 2, 14, 10, 17, 15, 7, 16, 18, 1, 19, 12]
Best route:  ['L', 'C', 'Q', 'B', 'T', 'J', 'A', 'R', 'E', 'G', 'K', 'D', 'F', 'M',
'H', 'S', 'N', 'O', 'P', 'I', 'L']
Best distance:  345.84700000000004
Greedy route:  ['L', 'M', 'H', 'S', 'O', 'N', 'K', 'D', 'F', 'E', 'G', 'A', 'R', 'Q',
'B', 'T', 'I', 'P', 'C', 'J', 'L']
Greedy distance:  371.891
```

Increasing the cities to 30 cities and changing the population size and mutation probability resulted in the genetic algorithm generating significantly worse results every time the algorithm was run in comparison to a simple greedy algorithmic approach. Because of this some changes clearly are necessary in my code. This was due to my hyperparameters of population size and the number of generations used. After increasing the population size, number of generations, and mutation probability the genetic algorithm outperformed the simple greedy approach in every test.

```
Best gene:  [11, 2, 3, 2, 5, 25, 2, 3, 1, 2, 1, 20, 20, 1, 16, 29, 14, 13, 12, 1, 1,
1, 29, 1, 16, 1, 19, 17, 20]
Best route:  ['T', '4', 'Z', 'E', 'N', 'H', 'V', 'M', 'I', 'C', 'Y', 'K', 'Q', 'A',
'2', '3', 'P', 'L', 'J', 'U', 'D', 'O', 'X', 'B', 'G', 'F', 'W', 'S', 'R', '1', 'T']
Best distance:  365.969
Greedy route:  ['T', '1', 'M', 'F', 'G', 'C', 'Y', 'K', 'P', '3', '4', 'Z', 'Q', '2',
'A', 'B', 'X', 'O', 'D', 'U', 'J', 'L', 'N', 'R', 'S', 'H', 'V', 'I', 'W', 'E', 'T']
Greedy distance:  409.32900000000006
```

7. Key Code

The crossover function can be seen above. The mutate, trav_order, random_gene, evaluation, and select_best_gene functions can be found in appendix 1.

```python
def run_ga(
        start_city,
        cities_list,
        neighbors_map,
        population_size=20,
        num_generations=50,
        mutation_prob=0.2,
        best_k=3
):
    num_steps = len(cities_list) - 1
    # generate the initial population
    population = [random_gene(num_steps) for _ in range(population_size)]

    # instantiate variables we are looking for
    best_gene = None
```

```
    best_route = None
    best_distance = float("inf")

    for gen in range(num_generations):
        # evaluate current population
        fitnesses = []
        for gene in population:
            route, total_dist, fitness = evaluate_gene(gene, start_city, cities_list,
neighbors_map)
            fitnesses.append(fitness)
            # if the current total distance is better than best, update best values
            if total_dist < best_distance:
                best_distance = total_dist
                best_gene = gene[:]
                best_route = route[:]
        # create next generation values using crossover and mutation
        new_population = []
        while len(new_population) < population_size:
            # select parents
            parent1 = select_best_gene(population, fitnesses, k=best_k)
            parent2 = select_best_gene(population, fitnesses, k=best_k)

            # complete crossover
            child_gene = crossover(parent1, parent2)

            # mutate child gene
            child_gene = mutate(child_gene, mutation_prob=mutation_prob)

            new_population.append(child_gene)

        population = new_population

    return best_gene, best_route, best_distance
```

8. Source Code

Source code is available in this github link: https://github.com/btorok-bu/METCS767_hw5

9. Comments on Performance

The alternative approach to solving the traveling salesman problem with a genetic algorithm is using permutations to represent the possible routes, evolving them through generations to find a shorter distance. In my implementation, the permutation is stored as a list of integers, each representing a city. The list must contain every city exactly once. When running the algorithm, an initial population of random permutations is created, then their fitness (total distance) is evaluated, and fitter parents are selected for crossover to create new children. The crossover and mutation operators maintain the permutation properly, preventing illegal routes. In

crossover this is done by using an ordered crossover in which a random subsection of parent 1 is copied directly to the offspring. The remaining empty spots in the offspring are filled by copying cities from parent 2. When copying from parent 2, any city that is already in the offspring are ignored. The permutation is maintained in mutation through swap and inversion by selecting two indices at random and swapping the cities to disrupt the order without causing duplicated cities. The complete implementation of this can be seen in appendix 2.

The runtime of these is significantly different. A simple parameter search with the permutation implementation took a total of 170s, whereas the same parameter search with the rank implementation took a total of 454s. The permutation implementation has a time complexity of $O(Pn^2G)$, and space complexity of $O(Pn)$, where P is the number of genes, n is number of cities, and G is the generational loop. The rank implementation has the same time complexity of $O(Pn^2G)$ and space complexity of $O(Pn + n^2)$. In this case, the big-O does not reveal the reason for why the rank implementation performs significantly worse because in the rank implementation the cost of n^2 occurs in the trav_order function where every single fitness evaluation scans the list of neighbors, builds a new python list, then performs a lookup for every gene and every generation. However, this n^2 loop only occurs in crossover for the permutation implementation which only happens when creating a new generation, greatly improving the performance of the algorithm. Big O analysis for both implementations is in appendix 3.

The best parameters were different between the two implementations, but the rank implementation described through this assignment outperformed the permutation implementation for the best distance. The best distance for the permutation implementation is 312.87, whereas the best distance for the rank implementation is 290.09. The rank implementation likely results in better results by baking a greedy heuristic into the decoder by sorting the neighbors by distance. When the rank is small, it is biased toward nearest few neighbors and if a rank is too large, it defaults to the nearest neighbor. This means that the genetic algorithm does not need to learn that nearest neighbors for this implementation, whereas for the permutation implementation this is not included since there's no neighbor sorting logic and the random permutation generated is just a random tour with no preference for short edges. The mutation logic also may result in more dramatic changes to the routes for the permutation implementation as swapping two city positions in the graph results in four edges changing, whereas only changing one only affects two edges. This means that evolution occurs faster in the permutation implementation, but the algorithm already has a larger task as it does not prefer short edges in the decoder.

To further improve the performance of the ranked genetic algorithm, I implemented an improved version that sets a max rank parameter of 5 to further increase the preference for shorter edges. This ensures the starting generation has all paths under rank 5. The mutation function also will only mutate genes to be within this same range. To improve the performance of trav_order, only up to max rank nearest unvisited neighbors are collected into a list, meaning the candidates list now no longer loops through all available cities, and will only create a list of the max rank for each. These changes decreased the runtime of the genetic algorithm from 454s to 292s while also improving best route performance with a best distance of 236.12. While this algorithm still has a longer runtime than the permutation implementation, the result is substantially better which more easily justifies the time cost. The additional code and complete results are in appendix 4.

References

[1] https://towardsdatascience.com/genetic-algorithm-6aefd897f1ac/

[2] https://www.woodruff.dev/day-16-solving-the-traveling-salesperson-problem-with-genetic-algorithms-permutation-chromosomes/

Evaluation

Note: We focus the **quality** (not quantity) of your submission. Material that doesn't respond to the criteria reduces grades.

Criterion *(based on the value of your significant prompts and your text)*	***D***	***C***	***B***	***A***	***Letter Grade***	***%***
Extent to which your ***added technical content is functional and accurate, and shows demonstrably that you understand the AI output, and have learned technically.***	***Low*** *extent.*	***Satisfatory*** *extent*	***Good*** *extent*	*Went significantly* ***beyond*** *what's required.*		*0.0*
Extent to which ***every significant claim you made is supported with clear, relevant logical reasoning that explains its validity.***	***Low*** *extent.*	***Satisfatory*** *extent*	***Good*** *extent*	*Went significantly* ***beyond*** *what's required.*		*0.0*
Extent to which ***your added material probes the key mechanisms in depth.***	***Low*** *extent.*	***Satisfatory*** *extent*	***Good*** *extent*	*Went significantly* ***beyond*** *what's required.*		*0.0*
				Assignment Grade:		*0.0*

The resulting grade is the average of these, using A+=100 (outstanding--rare), A=95 (excellent in all ways), A-=90 (excellent), B+=87 (excellent / very good), B=85 (very good), B-=80 (good) etc.

Appendix 1

```python
def mutate(gene, mutation_prob=0.2):
    rng = random
    # decide if this child will have a mutation
    if rng.random() > mutation_prob:
        return gene[:] # no mutation occurs, return copy of original gene

    n = len(gene)

    max_rank = n # bound the max rank a mutation can have
    new_gene = gene[:] # make a copy of the existing gene

    # choose the position to mutate
    position = rng.randrange(n)
    # get the old rank in the original position that will be mutated
```

```
    old_rank = new_gene[position]
    new_rank = rng.randint(1, max_rank)
    # if the new rank is the same, then generate another rank
    while new_rank == old_rank:
        new_rank = rng.randint(1, max_rank)
    new_gene[position] = new_rank # set the new rank
    return new_gene
```

```
def trav_order(start, cities_list, neighbors_map, gene):
    # instantiate the unvisited cities list and remove the start city
    unvisited = set(cities_list)
    unvisited.remove(start)

    route = [start]
    total_distance = 0.0
    current_city = start
    # need to visit each city exactly once
    steps_needed = len(cities_list) - 1
    # loop through every city needed to visit
    for step in range(steps_needed):
        if step < len(gene):  # if the step is less than the length, then add the rank
            rank = gene[step]
        else:
            rank = 1 # default to closest unvisited city
        candidate_neighbors = [
            (nbr, d) for nbr, d in neighbors_map[current_city] if nbr in unvisited
        ]

        # wrap the rank if it exceeds the number of available neighbors
        idx = (rank - 1) % len(candidate_neighbors)
        next_city, step_distance = candidate_neighbors[idx]

        # update route and tracking variables
        route.append(next_city)
        total_distance += step_distance
        unvisited.remove(next_city)
        current_city = next_city
    # update distance variables for the complete route
    return_distance = distance(current_city, start)
    total_distance += return_distance
    # close route by returning to start city
    route.append(start)

    return route, total_distance

def random_gene(num_steps, max_rank=None):
    # Create a random rank-encoded gene.
```

```
    if max_rank is None:
        max_rank = num_steps  # simple choice for small problems
    return [random.randint(1, max_rank) for _ in range(num_steps)]


# evaluate whether a gene is good and return the route, total distance, and fitness
def evaluate_gene(gene, start_city, cities_list, neighbors_map):
    # complete route and fitness for gene
    route, total_distance = trav_order(start_city, cities_list, neighbors_map, gene)
    fitness = 1.0/total_distance
    return route, total_distance, fitness


def select_best_gene(population, fitness, k=3):
    # select k random genes and return the best
    indices = random.sample(range(len(population)), k=k)
    best_idx = max(indices, key=lambda i: fitness[i])
    return population[best_idx]
```

Appendix 2

Permutation based genetic algorithm implementation:

```
# add functions for permutation GA traveling salesman implementation

def decode_route_perm(start_city, perm):
    # get the route given a start city and permutation
    route = [start_city] + perm + [start_city]
    total_distance = 0.0
    # iterate through route and add up total distance of route
    for i in range(len(route) - 1):
        total_distance += distance(route[i], route[i+1])
    return route, total_distance


def random_perm_gene(cities_list, start_city):
    # return a random permutation containing all cities except start city
    perm = [c for c in cities_list if c != start_city]
    random.shuffle(perm)
    return perm


def evaluate_perm_gene(gene, start_city):
    # get the route, distance, and fitness for a gene thats perm-encoded
    route, total_distance = decode_route_perm(start_city, gene)
```

```
    fitness = 1.0 / total_distance
    return route, total_distance, fitness

# reuse the best gene code - no changes necessary
def select_best_gene(population, fitness, k=3):
    # select k random genes and return the best
    indices = random.sample(range(len(population)), k=k)
    best_idx = max(indices, key=lambda i: fitness[i])
    return population[best_idx]
def order_crossover(parent1, parent2):
    # get the length of parent 1
    n = len(parent1)

    # choose a random slice [i, j) - this will be copied to child
    i = random.randint(0, n - 2)
    j = random.randint(i + 1, n - 1)

    # create a child chromosome, populate with None values same length as parent 1
    child = [None] * n

    # copy slice from parent1
    child[i:j] = parent1[i:j]

    # fill remaining positions from parent2 in order,
    p2_idx = 0
    for k in range(n):
        if child[k] is None:
            # find next city in parent2 that is not yet in child
            while parent2[p2_idx] in child:
                # move to next index if city already in child
                p2_idx += 1
            # if value is not in child, then populate and move to next index
            child[k] = parent2[p2_idx]
            p2_idx += 1

    return child
```

```
def mutate_swap(gene, mutation_prob=0.2):
    # decide if mutation will occur
    if random.random() > mutation_prob:
        return gene[:]

    # get the length of the gene
    n = len(gene)
    # choose two random indicies in range of length of gene
    i, j = random.sample(range(n), 2)
```

```
    new_gene = gene[:]
    # swap the values of the positions and return the new gene
    new_gene[i], new_gene[j] = new_gene[j], new_gene[i]
    return new_gene
```

```
def run_ga_perm(
    start_city,
    cities_list,
    population_size=100,
    num_generations=200,
    mutation_prob=0.6,
    best_k=3,
):
    # initialize population
    population = [
        random_perm_gene(cities_list, start_city)
        for _ in range(population_size)
    ]

    best_gene = None
    best_route = None
    best_distance = float("inf")

    for gen in range(num_generations):
        fitnesses = []
        for gene in population:
            route, total_dist, fitness = evaluate_perm_gene(gene, start_city)
            fitnesses.append(fitness)

            if total_dist < best_distance:
                best_distance = total_dist
                best_gene = gene[:]
                best_route = route[:]

        # build next generation
        new_population = []
        while len(new_population) < population_size:
            parent1 = select_best_gene(population, fitnesses, k=best_k)
            parent2 = select_best_gene(population, fitnesses, k=best_k)

            child = order_crossover(parent1, parent2)
            child = mutate_swap(child, mutation_prob=mutation_prob)
            new_population.append(child)

        population = new_population
```

```
    return best_gene, best_route, best_distance
```

Parameter search function:

```
def parameter_search(start_city, cities_list, neighbors_map, population_sizes,
                     num_generations_list, mutation_probs, best_k_list,
                     trials_per_setting=5, perm=False):
    all_results = []
    best_overall = None
    # Cartesian product of all hyperparameter choices
    for pop_size, n_gen, mut_prob, k in itertools.product(population_sizes,
                                                          num_generations_list,
                                                          mutation_probs,
                                                          best_k_list):
        trial_distances = []
        trial_genes = []
        trial_routes = []

        for t in range(trials_per_setting):
            random.seed(89)
            # if perm flag is set, then run perm GA implementation
            if perm is True:
                best_gene, best_route, best_distance = run_ga_perm(
                    start_city=start_city,
                    cities_list=cities_list,
                    population_size=pop_size,
                    num_generations=n_gen,
                    mutation_prob=mut_prob,
                    best_k=k
                )
            # else run with rank GA implementation
            else:
                best_gene, best_route, best_distance = run_ga(
                    start_city=start_city,
                    cities_list=cities_list,
                    neighbors_map=neighbors_map,
                    population_size=pop_size,
                    num_generations=n_gen,
                    mutation_prob=mut_prob,
                    best_k=k
                )

            trial_distances.append(best_distance)
            trial_genes.append(best_gene)
            trial_routes.append(best_route)

        avg_distance=statistics.mean(trial_distances)
```

```
        best_idx=min(range(len(trial_distances)), key=lambda i: trial_distances[i])
        result = {
            "population_size": pop_size,
            "num_generations": n_gen,
            "mutation_prob": mut_prob,
            "best_k": k,
            "avg_distance": avg_distance,
            "best_distance": trial_distances[best_idx],
            "best_gene": trial_genes[best_idx],
            "best_route": trial_routes[best_idx],
        }
        all_results.append(result)

        if best_overall is None or avg_distance < best_overall['avg_distance']:
            best_overall = result

    return best_overall, all_results
```

Appendix 3:

Permutation implementation Big-O analysis:

1. decode_route_perm()
    a. concatenates list of length n -> O(n)
    b. loops over len(route) -1 for ~ n+1 edges -> each distance is O(1), loop costs O(n)
    c. Time = O(n), space = O(n)
2. random_perm_gene()
    a. list comprehension over cities list of size n -> O(n)
    b. random shuffle on list length n-1 -> O(n)
    c. Time = O(n), space = O(n)
3. evaluate_perm_gene()
    a. calls decode_route_perm() -> O(n)
    b. Time = O(n), space = O(n)
4. select_best_gene()
    a. random sample for k indices -> O(k)
    b. get max over k indices – each lambda lookup is O(1)
    c. Time = O(k), space = O(k)
5. order_crossover()
    a. allocates child O(n)
    b. copies splice of parent -> worst case is copying the entire array -> O(n)
    c. While loop over range k if the values are the same
        i. Nested for loop with worst case O(n) for each -> $O(n^2)$
    d. Time = $O(n^2)$, space = O(n)
6. mutate_swap()
    a. No mutation case, copy of gene -> O(n)
    b. Mutation case, same copy + constant swap time -> O(n)

    c. Time = O(n), space = O(n)
7. Overall GA complexity for run_ga_perm()
    a. Initialization of population calls random_perm_gene each time (each O(n))
    b. Time = O(P * n), space = O(P*n), where P is population
8. Main loop
    a. Per generation loops over size of P and evaluates gene (O(n))
        i. Time = O(P*n) per generation
    b. Offspring creation per generation, building P children in while loop
        i. Call select_best_gene twice -> 2* O(k) -> O(k)
        ii. Call order_crossover -> $O(n^2)$
        iii. Call mutate_swap -> O(n)
        iv. Total per child: $O(k + n^2 + n) = O(n^2 + k)$
        v. Per generation: $O(P * (n^2 + k)) = O(Pn^2 + Pk)$
    c. Combine and evaluate offspring per generation
        i. Fitness evaluation: O(Pn)
        ii. Offspring creation: $O(Pn^2 + Pk)$
        iii. Total per generation dominant term is $O(Pn^2)$
    d. Overall time = $O(Pn^2G)$ where G is number of generations
    e. Overall space = O(Pn)

Rank order Big-O analysis:

1. Preprocessing to build neighbors dictionary
    a. 1st loop through dist.items -> $O(n^2)$
    b. Each append is O(1)
    c. 2nd loop through each city, neighbor list length is n – 1
    d. Sort each list: O((n-1) log(n-1)) = O(n*logn)
    e. There are n city lists -> $O(n^2 * logn)$
    f. Time = $O(n^2 * logn)$, space = $O(n^2)$ (neighbors stores all adjacency info)
2. trav_order()
    a. list comprehension scans neighbors once -> O(n)
    b. build candidate neighbors n steps * O(n) each time -> $O(n^2)$
    c. Time = $O(n^2)$, space = O(n)
3. crossover()
    a. both cutpoints -> O(n)
    b. concatenation -> O(n)
    c. Time = O(n), space = O(n) (for new child)
4. mutate()
    a. no mutation case -> O(n)
    b. Mutation case O(n)
    c. Time = O(n), space = O(n) (for new gene)
5. random_gene()
    a. list comprehension over num_steps -> O(n)
    b. Time = O(n), space = O(n)

6. evaluate_gene()
    a. this is dominated by trav_order, therefore time and space is the same
    b. Time = $O(n^2)$, space = $O(n)$
7. select_best_gene()
    a. sampling k indices and getting max over the indices -> O(k)
    b. Time = O(k), space = O(k)
8. run_ga() for overall genetic algorithm
    a. initialize population makes P calls to random_gene(), each O(n) time and space
    b. Time = O(P*n), space = O(P*n)
9. Main generational loop
    a. For each gene, evaluate_gene is called which is $O(n^2)$ -> $O(P*n^2)$
    b. New population is created per generation
        i. For each child select best gene called twice -> O(k)
        ii. crossover() called once -> O(n)
        iii. mutate() called once -> O(n)
    c. Total per child -> O(n + k)
    d. Per generation -> $O(P*n^2)$
    e. Overall time = $O(Pn^2*G)$, space = $O(Pn + n^2)$, where G is number of generations

Appendix 4:

Improved implementation:

```
MAX_RANK = 5

def random_gene(num_steps, max_rank=MAX_RANK):
    # Create a random rank-encoded gene with ranks in [1, max_rank]
    return [random.randint(1, max_rank) for _ in range(num_steps)]


def mutate(gene, mutation_prob=0.2, max_rank=MAX_RANK):
    rng = random
    if rng.random() > mutation_prob:
        return gene[:]  # no mutation, just copy

    n = len(gene)
    new_gene = gene[:]

    position = rng.randrange(n)
    old_rank = new_gene[position]
    new_rank = rng.randint(1, max_rank)
    while new_rank == old_rank and max_rank > 1:
        new_rank = rng.randint(1, max_rank)
    new_gene[position] = new_rank

    return new_gene
```

```python
def trav_order(start, cities_list, neighbors_map, gene):
    unvisited = set(cities_list)
    unvisited.remove(start)

    route = [start]
    total_distance = 0.0
    current_city = start
    steps_needed = len(cities_list) - 1

    for step in range(steps_needed):
        # get rank for this step, default to 1 if gene shorter
        if step < len(gene):
            rank = gene[step]
        else:
            rank = 1

        # collect up to MAX_RANK nearest unvisited neighbors
        candidates = []
        for nbr, d in neighbors_map[current_city]:
            if nbr in unvisited:
                candidates.append((nbr, d))
                if len(candidates) == MAX_RANK:
                    break

        if not candidates:
            raise RuntimeError(
                f"No unvisited neighbors left from {current_city}"
            )

        # wrap rank into the available candidates
        idx = (rank - 1) % len(candidates)
        next_city, step_distance = candidates[idx]

        route.append(next_city)
        total_distance += step_distance
        unvisited.remove(next_city)
        current_city = next_city

    # close the route
    return_distance = distance(current_city, start)
    total_distance += return_distance
    route.append(start)

    return route, total_distance
```

Performance of permutation:

```
=== Best overall setting for permutation implementation ===
population_size: 200
num_generations: 200
mutation_prob : 0.8
best_k        : 5
avg_distance  : 312.8769999999999
best_distance : 312.8769999999999
best_gene     : ['Y', 'Z', 'W', '3', 'Q', 'H', 'C', 'M', 'D', 'S', 'K', 'N', 'V', '1',
'G', 'J', 'A', '2', 'L', 'O', 'P', 'U', 'B', 'X', 'I', 'F', 'R', 'E', '4']
best_route    : ['T', 'Y', 'Z', 'W', '3', 'Q', 'H', 'C', 'M', 'D', 'S', 'K', 'N', 'V',
'1', 'G', 'J', 'A', '2', 'L', 'O', 'P', 'U', 'B', 'X', 'I', 'F', 'R', 'E', '4', 'T']
```

Performance of original rank implementation:

```
=== Best overall setting for original rank implementation ===
population_size: 100
num_generations: 200
mutation_prob : 0.8
best_k        : 2
avg_distance  : 290.09899999999993
best_distance : 290.09899999999993
best_gene     : [8, 6, 29, 2, 27, 26, 4, 23, 22, 1, 20, 19, 19, 18, 17, 1, 3, 26, 1,
2, 10, 25, 8, 19, 16, 25, 7, 15, 12]
best_route    : ['T', 'Y', 'M', 'C', 'H', 'Q', 'S', 'K', 'Z', '4', 'E', 'R', 'F', 'I',
'V', 'N', 'A', 'P', 'U', 'W', 'G', 'J', 'B', 'X', 'D', '1', '3', 'O', 'L', '2', 'T']
```

Performance of improved rank implementation:

```
=== Best overall setting for improved rank implementation ===
population_size: 200
num_generations: 300
mutation_prob : 0.6
best_k        : 2
best_distance : 236.12899999999996
best_gene     : [1, 2, 1, 1, 2, 2, 2, 1, 1, 1, 3, 1, 1, 1, 1, 1, 4, 1, 1, 1, 1, 2, 1,
1, 1, 1, 4, 1, 4]
best_route    : ['T', '4', 'E', 'R', 'F', 'I', 'V', 'N', 'A', 'W', '3', 'G', 'J', 'S',
'K', 'Z', 'B', 'U', 'P', 'O', 'L', '2', '1', 'D', 'M', 'C', 'Q', 'H', 'X', 'Y', 'T']
```

Link to LLM Chat: https://chatgpt.com/share/692fa26a-31f4-8006-aea8-bdc2dad81482

Example asking ChatGPT for code review and code efficiency help:

```
def random_perm_gene(cities_list, start_city):
    # create a random permutation gene that contains all cities apart from start
    new_perm = cities_list.remove(start_city)
    # randomly shuffle the city
    random.shuffle(new_perm)
    return new_perm
```

Is this a better implementation or does it take longer to execute this code?