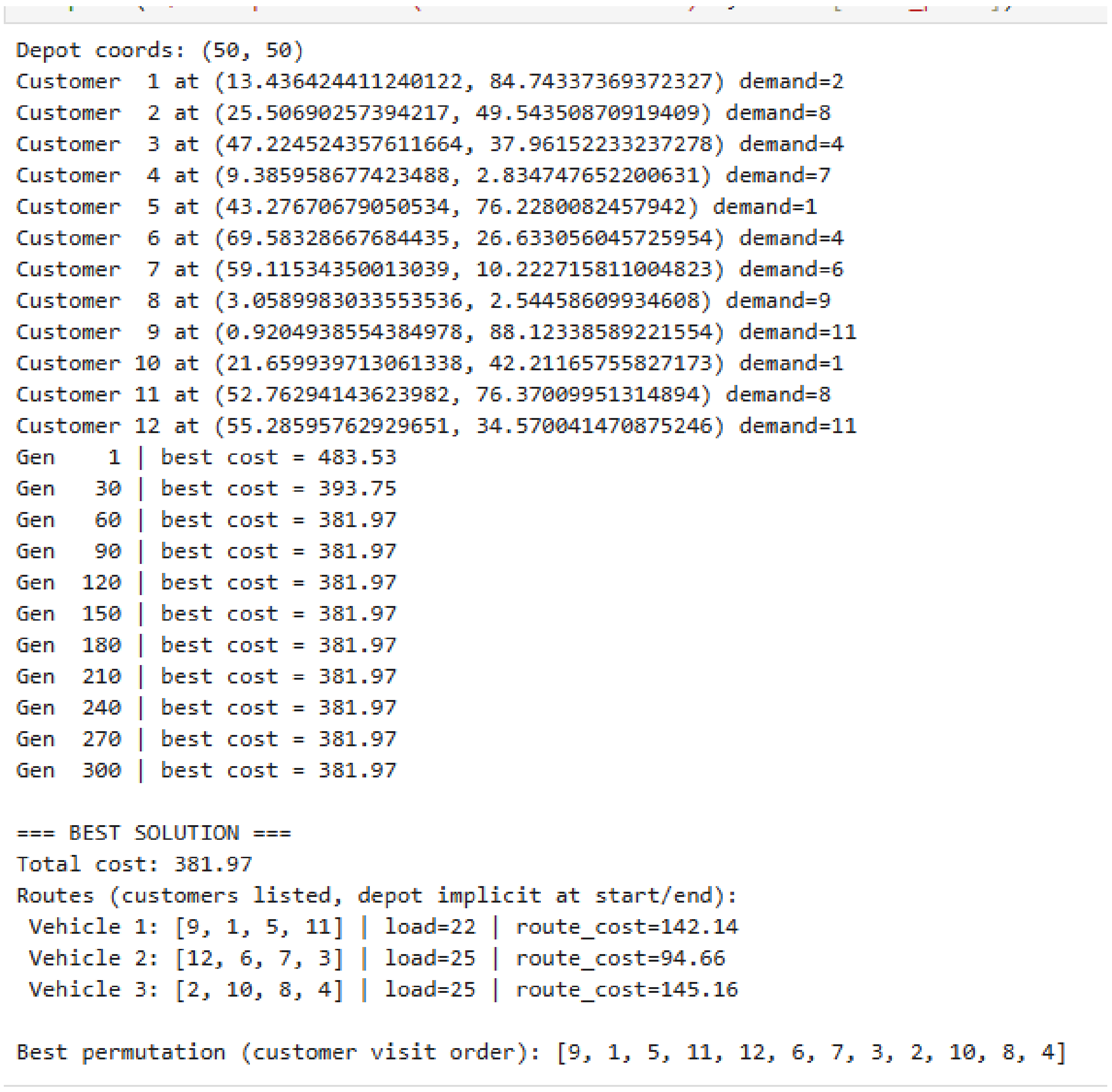
Lab 1

Genetic Algorithm

```python
import math
import random
import copy
from typing import List, Tuple

random.seed(1)

# --------------------------
# Utility functions
# --------------------------

def euclidean(a: Tuple[float, float], b: Tuple[float, float]) -> float:
    return math.hypot(a[0] - b[0], a[1] - b[1])

def compute_distance_matrix(coords: List[Tuple[float, float]]) -> List[List[float]]:
    n = len(coords)
    dist = [[0.0]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            dist[i][j] = euclidean(coords[i], coords[j])
    return dist

# --------------------------
# Problem instance generator
# --------------------------

def generate_instance(num_customers=15, grid_size=100, max_demand=10, depot=(50,50)):
    """
    Generates random customers with demands and coordinates.
    Index 0 is the depot, customers are 1..num_customers
    """
    coords = [depot]
    demands = [0]
    for _ in range(num_customers):
        coords.append((random.uniform(0, grid_size), random.uniform(0, grid_size)))
        demands.append(random.randint(1, max_demand))
    return coords, demands

# --------------------------
# Chromosome -> Routes (split by capacity)
# --------------------------
# Chromosome -> Routes (split by capacity)
# --------------------------

def split_routes_from_perm(perm: List[int], demands: List[int], capacity: int) -> List[List[int]]:
    """
    Greedy split: take customer sequence in perm, keep adding to current route
    while capacity is not exceeded. Start new route when needed.
    """
    routes = []
    cur_route = []
    cur_load = 0
    for cust in perm:
        d = demands[cust]
        if cur_load + d <= capacity:
            cur_route.append(cust)
            cur_load += d
        else:
            # close route and start new
            if cur_route:
                routes.append(cur_route)
            cur_route = [cust]
            cur_load = d
    if cur_route:
        routes.append(cur_route)
    return routes

def route_cost(route: List[int], dist_matrix: List[List[float]]) -> float:
    if not route:
        return 0.0
    cost = 0.0
    prev = 0  # depot
    for cust in route:
        cost += dist_matrix[prev][cust]
        prev = cust
    cost += dist_matrix[prev][0]  # return to depot
    return cost

def total_cost(routes: List[List[int]], dist_matrix: List[List[float]]) -> float:
    return sum(route_cost(r, dist_matrix) for r in routes)

# --------------------------
# Genetic Algorithm components
# --------------------------

def init_population(pop_size: int, customer_ids: List[int]) -> List[List[int]]:
    pop = []
    return sum(route_cost(r, dist_matrix) for r in routes)

# --------------------------
# Genetic Algorithm components
# --------------------------

def init_population(pop_size: int, customer_ids: List[int]) -> List[List[int]]:
    pop = []
    for _ in range(pop_size):
        perm = customer_ids[:]
        random.shuffle(perm)
        pop.append(perm)
    return pop

def tournament_selection(pop: List[List[int]], fitnesses: List[float], k=3) -> List[int]:
    selected = random.sample(list(range(len(pop))), k)
    best = min(selected, key=lambda i: fitnesses[i])
    return copy.deepcopy(pop[best])

def order_crossover(parent1: List[int], parent2: List[int]) -> Tuple[List[int], List[int]]:
    """Order Crossover (OX) for permutations."""
    n = len(parent1)
    a, b = sorted(random.sample(range(n), 2))
    def ox(p1, p2):
        child = [None]*n
        # copy slice
        child[a:b+1] = p1[a:b+1]
        # fill remaining from p2 order
        pos = (b+1) % n
        p2_i = (b+1) % n
        while None in child:
            val = p2[p2_i]
            if val not in child:
                child[pos] = val
                pos = (pos+1) % n
            p2_i = (p2_i+1) % n
        return child
    return ox(parent1, parent2), ox(parent2, parent1)

def swap_mutation(perm: List[int], mut_rate=0.2) -> None:
    """In-place swap mutation (may perform multiple swaps based on mut_rate)."""
    n = len(perm)
    for i in range(n):
        if random.random() < mut_rate:
            j = random.randrange(n)
            perm[i], perm[j] = perm[j], perm[i]
    best_routes = split_routes_from_perm(best_perm, demands, capacity)
    best_cost = fitnesses[best_idx]
    return {
        'best_perm': best_perm,
        'best_routes': best_routes,
        'best_cost': best_cost,
        'dist_matrix': dist_matrix,
        'history': best_history
    }

# --------------------------
# Example / Run
# --------------------------

if __name__ == "__main__":
    # Example instance
    NUM_CUSTOMERS = 12
    VEHICLE_CAPACITY = 30
    coords, demands = generate_instance(num_customers=NUM_CUSTOMERS, grid_size=100, max_demand=12, depot=(50,50))

    print("Depot coords:", coords[0])
    for i in range(1, len(coords)):
        print(f"Customer {i:2d} at {coords[i]} demand={demands[i]}")

    result = ga_vrp(
        coords=coords,
        demands=demands,
        capacity=VEHICLE_CAPACITY,
        pop_size=120,
        gens=300,
        crossover_rate=0.9,
        mutation_rate=0.12,
        elitism=4
    )

    print("\n=== BEST SOLUTION ===")
    print(f"Total cost: {result['best_cost']:.2f}")
    print("Routes (customers listed, depot implicit at start/end):")
    for idx, r in enumerate(result['best_routes'], start=1):
        load = sum(demands[c] for c in r)
        rcost = route_cost(r, result['dist_matrix'])
        print(f" Vehicle {idx}: {r} | load={load} | route_cost={rcost:.2f}")
    print("\nBest permutation (customer visit order):", result['best_perm'])
```



```
Depot coords: (50, 50)
Customer  1 at (13.436424411240122, 84.74337369372327) demand=2
Customer  2 at (25.50690257394217, 49.54350870919409) demand=8
Customer  3 at (47.224524357611664, 37.96152233237278) demand=4
Customer  4 at (9.385958677423488, 2.834747652200631) demand=7
Customer  5 at (43.27670679050534, 76.2280082457942) demand=1
Customer  6 at (69.58328667684435, 26.633056045725954) demand=4
Customer  7 at (59.11534350013039, 10.222715811004823) demand=6
Customer  8 at (3.0589983033553536, 2.54458609934608) demand=9
Customer  9 at (0.9204938554384978, 88.12338589221554) demand=11
Customer 10 at (21.659939713061338, 42.21165755827173) demand=1
Customer 11 at (52.76294143623982, 76.37009951314894) demand=8
Customer 12 at (55.28595762929651, 34.570041470875246) demand=11
Gen    1 | best cost = 483.53
Gen   30 | best cost = 393.75
Gen   60 | best cost = 381.97
Gen   90 | best cost = 381.97
Gen  120 | best cost = 381.97
Gen  150 | best cost = 381.97
Gen  180 | best cost = 381.97
Gen  210 | best cost = 381.97
Gen  240 | best cost = 381.97
Gen  270 | best cost = 381.97
Gen  300 | best cost = 381.97

=== BEST SOLUTION ===
Total cost: 381.97
Routes (customers listed, depot implicit at start/end):
 Vehicle 1: [9, 1, 5, 11] | load=22 | route_cost=142.14
 Vehicle 2: [12, 6, 7, 3] | load=25 | route_cost=94.66
 Vehicle 3: [2, 10, 8, 4] | load=25 | route_cost=145.16

Best permutation (customer visit order): [9, 1, 5, 11, 12, 6, 7, 3, 2, 10, 8, 4]
```