**Capacitated Vehicle Routing Problem multiple type vehicles with varying cost and capacities using Simulated Annealing Method**

Atakan Özerkan, Koray Baysal and Sevde Zümrüt İspay

Yaşar University, University Avenue No:37-39 Bornova, İZMİR, TURKEY bilgi@yasar.edu.tr

**Abstract.** The Capacitated Vehicle Routing Problem (CVRP) is a complex problem in transportation logistics that involves finding an optimal route for a fleet of vehicles to service a set of customers while satisfying various constraints. The objective of the CVRP is to minimize the total distance traveled by the vehicles while meeting all the constraints. Simulated annealing is a stochastic optimization algorithm inspired by the process of annealing in metallurgy. It is used to find the global minimum cost function in a large search space. The algorithm starts with an initial solution and iteratively perturbs it by making random moves to neighboring solutions. During the process, the algorithm gradually reduces the probability of accepting moves that increase the cost function, with the aim of avoiding local optimum and moving towards the global optimum. This study examines the book distribution operations of a vendor in Ankara, Turkey, which involves delivering books to eight different cities across the country: Ankara, Istanbul, Izmir, Bursa, Antalya, Konya, Adana, Samsun, and Erzurum. To optimize the delivery process, the study calculates the distance matrix for these cities and considers the capacities of the vendor's vehicles, which include one large and two small vehicles with capacities, respectively. This project aims to provide inform the design of efficient and cost-effective book distribution strategies for vendors operating in similar contexts.

**Keywords:** simulated annealing, genetic algorithm, tabu search

1. INTRODUCTION

Problem Definition

### This project observed the Capacitated Vehicle Routing Problem (CVRP) multiple type vehicles with varying costs and capacities using the Simulated Annealing Method. The CVRP is a widely studied combinatorial optimization problem that concerns the effective allocation of vehicles to a set of customers while satisfying capacity constraints and minimizing the total distance traveled. CVRP has numerous practical applications in transportation, logistics, and waste management. Various approaches have been proposed to tackle the CVRP, including genetic algorithms, tabu search, and simulated annealing. This project utilized the Simulated Annealing method to solve a problem. Simulated Annealing is a stochastic optimization algorithm that searches for the global minimum of a cost function in a large search space by iteratively improving a randomly generated solution through random movements. The algorithm reduces the probability of accepting moves that increase the cost function, preventing a solution from being trapped in a local minimum. Our problem involved a book seller in Ankara who needed to sell books to 8 different cities with his vehicles (Ankara, Istanbul, Izmir, Bursa, Antalya, Konya, Adana, Samsun, Erzurum) while meeting the demands, and doing so with the lowest cost and shortest travel distance. In the Overview of Resolution Procedures and Literature Review section, a literature review was conducted to determine the appropriate methods to use. The Modeling Approach section details the use of Python and Excel to implement the chosen method.

1. LITERATURE REVIEW

Considering the contributions made in this study and the wide scope of the subject, the literature review has been examined under three headings: simulated annealing, genetic algorithm and tabu search.

1. Simulated Annealing

The researchers propose the Simulated Annealing (SA) algorithm for the Bentonite Distribution and Delivery Problem (BDDP). BDDP is the problem of assigning multiple resources (e.g., workers or machines) in the most appropriate way to fulfill a certain workload within a given time. The article suggests that the SA algorithm produces more effective results compared to other optimization methods and can be used to solve problems in BDDP (Lijun Wei & et al 2018). [1]

The authors evaluate the effectiveness of the Simulated Annealing (SA) algorithm in optimizing a logistics company to distribute goods from a warehouse to various customer points. They demonstrate that the use of SA algorithm in logistics produces better results than traditional methods, and it is possible to obtain a faster solution. As authors state that SA algorithm is used for solving the proposed model in large-scale problems and acceptable results are obtained (2006). [2]

1. Genetic Algorithm

"The results demonstrate that the proposed algorithm can generate good quality solutions in a reasonable amount of time, and is able to handle a range of constraints, including time windows, vehicle capacity, and others." [3]

The authors propose a genetic algorithm to optimize the CVRP, which is an important problem in transportation logistics. The algorithm is designed to minimize the total distance travelled by the vehicles while ensuring that each vehicle's capacity is not exceeded. The results of the experiments demonstrate the effectiveness of the proposed genetic algorithm, which can generate near-optimal solutions for the CVRP in a reasonable amount of time. [4]

1. Tabu Search

The authors state the periodic vehicle routing problem (PVRP) is a variant of the capacitated vehicle routing problem (CVRP) in which customers must be visited periodically. The algorithm is designed to optimize the routing plan while considering the capacity constraints of the vehicles, accessibility restrictions, and multiple vehicle trips (2008). [5]

The experimental results show that optimize the route for each vehicle while considering the capacity constraints and time window restrictions by Mohammad Deni Akbar and Rio Aurachmana (2020). [6]

1. TERMINOLOGY

* **import requests** : This module imports which allows the user to send HTTP requests using Python.
* **import json** : This module imports the JSON library.
* **import random** : This module offers a range of functions for producing random numbers and making random choices.
* **import math** : This module offers a variety of mathematical functions and constants, including trigonometric functions, logarithmic functions, and the constant pi.
* **requests.get()** : This module sends a get() request to a specified URL and returns the response.
* **json.loads()** : This module offers to decode a JSON string into a Python object. It is part of the built-in json module and allows you to convert JSON data into Python objects like lists or dictionaries.
* **.append()** : This module is a built-in list method that adds an element to the end of a list. By calling this method on a list object and providing an item as an argument, you can extend the list by one element at a time.
* **len()** : This module supplies an intrinsic function that calculates the length (total elements) of a provided object like a list, tuple, or string.
* **random.sample(range())** : This module imports helpful when you need to select unique random elements within a given range of numbers.
* **random.shuffle()** : This module represents a function that randomly rearranges the elements of a mutable sequence, like a list.
* **list(range())** : This module handles producing number lists for tasks like looping, indexing, or other operations involving integer sequences.
* **sum()** : This module takes an iterable and returns the sum of all the values in that iterable.
* **math.exp()** : This module is a function from the built-in math module that returns the exponential value of a given number.
* **enumerate()** : This module offers a built-in function that returns an iterator that generates pairs of indexes and corresponding values from an iterable object such as a list.
* **.join()** : This module represents a string method that returns a string concatenated with a specified delimiter.
* **map(str, i)** : This module is a built-in function that applies a specified function to each item of an iterable object and returns a map object, which can be converted to a list or tuple. And map(str, i) applies the str() function to each element of the iterable i, converting each element to a string. It returns a map object that can be used to convert the elements of i to strings.

1. MODELING APPROACH

To solve our problem, we first needed a distance matrix. For this reason, we wanted to create a distance matrix using the Bing Maps API. First, we identified the 9 locations and the delivery point according to our problem. We needed the latitude, longitude, and coordinates of these locations because the server would calculate the distances through this data. Then, we integrated these inputs into the Python code along with the trial key obtained from the website. This allowed us to determine the distances between cities and create a distance matrix.

This code uses the Bing Maps API to calculate the driving distances between different cities in Turkey. The latitude and longitude coordinates for each city are stored in a list, and a function is created to calculate the distances between each pair of cities. The function sends a request to the Bing Maps API with the origin and destination coordinates and extracts the travel distance from the JSON response. The distances are stored in a matrix, which is then printed to the console.

import requests #to import the HTTP library

import json #to import the JSON library

API\_KEY = 'As09KQ-LgB54q-HZo-x4L6Bl9y0YdP7Aop8LhzYYFXJgPjz4EOXInX1d1q0nMLvx' #to make a request to the Bing Maps API

coordinates = [

    (39.9208, 32.8541),  # Ankara

    (41.0082, 28.9784),  # İstanbul

    (38.4192, 27.1287),  # İzmir

    (40.1960, 29.0503),  # Bursa

    (36.8621, 30.6387),  # Antalya

    (37.8722, 32.4975),  # Konya

    (37.0000, 35.3213),  # Adana

    (41.2785, 36.3361),  # Samsun

    (39.9022, 41.2675)   # Erzurum

] #to generate a list of bundles representing the latitude and longitude coordinates of different cities

def create\_distance\_matrix(coordinates): #to return a list of distances between cities

    distance\_matrix = [] #to create an empty list to store the #distance matrix

    for origin in coordinates: #to loop through each tuple in the list of coordinates

        row = [] #to create an empty list to store the distances between the origin city and each destination city

        for destination in coordinates: #to loop through each tuple in the coordinates list and assign it to the variable destination

            if origin == destination: #to check if the origin city and destination city are the same

                row.append(0) #for append a distance of 0 to the row list, if the origin city and destination city are the same

            else:

                response = requests.get(f'http://dev.virtualearth.net/REST/v1/Routes/DistanceMatrix?origins={origin[0]},{origin[1]}&destinations={destination[0]},{destination[1]}&travelMode=driving&key={API\_KEY}') #to send a GET request to the Bing Maps API with the origin and destination coordinates, the travel mode set to driving, and the API key

                data = json.loads(response.text) #for response and store it in the data variable

                distance = data['resourceSets'][0]['resources'][0]['results'][0]['travelDistance'] #to extract the travel distance from the JSON response and store it in the distance variable

                row.append(distance) #for append the distance value to the row list

        distance\_matrix.append(row) #for append the row list to the distance\_matrix list

    return distance\_matrix

distance\_matrix = create\_distance\_matrix(coordinates) #to call the create\_distance\_matrix function with the coordinates list and assign the resulting distance\_matrix list to a variable

for row in distance\_matrix: #to loop through each list in the distance\_matrix list and assign it to the variable row

    print(row)

**OUTPUT**

distance\_matrix = [

[0, 448611, 591924, 3863, 483555, 269697, 493869, 410305, 877571],

[450053, 0, 483966, 154226, 697236, 700143, 924315, 731766, 1228502],

[58949, 480755, 0, 34477, 460973, 562904, 900641, 1098323, 1595059],

[38726, 155622, 34934, 0, 549806, 494314, 815754, 77319, 1269923],

[484764, 698296, 460301, 550466, 0, 315262, 617247, 900634, 1247748],

[270057, 708694, 566826, 494365, 312669, 0, 343838, 589118, 93306],

[494349, 932986, 904386, 817784, 618029, 342106, 0, 724025, 836274],

[403507, 730996, 1102078, 772338, 901058, 59016, 724548, 0, 632475],

[871633, 1228601, 1599683, 1269943, 1248796, 937898, 836796, 632603, 0]

]

And after, this code defines a “simulated\_annealing“ function to solve the Capacitated Vehicle Routing Problem (CVRP) using the Simulated Annealing (SA) optimization method. The distances between cities are in the “distance\_matrix“, and each city's demand is in the “demands“ list. Each vehicle's capacity is in the “vehicles` list. The “simulated\_annealing“ function uses the distance matrix, vehicles, demands, and other optional parameters like the number of iterations, cooling rate (alpha), and initial temperature. It returns the best routes for each vehicle and the total distance traveled. The algorithm starts by creating an initial random solution with routes for each vehicle. Then, it tries to improve the solution by swapping two cities in the current routes and calculating the total distance for the new routes. If the new routes' total distance is smaller than the current routes, it accepts the new routes. Otherwise, it accepts the new routes with a certain probability based on the difference in distances and the current temperature using the cooling rate (alpha). After the given number of iterations, the function returns the best routes found and the total distance traveled. The results show the best routes for each vehicle and the total distance traveled.

import random #to generate random numbers and make random selections

import math #for mathematical functions and constants

distance\_matrix = [

     [0, 448611, 591924, 3863, 483555, 269697, 493869, 410305, 877571],

     [450053, 0, 483966, 154226, 697236, 700143, 924315, 731766, 1228502],

     [58949, 480755, 0, 34477, 460973, 562904, 900641, 1098323, 1595059],

     [38726, 155622, 34934, 0, 549806, 494314, 815754, 77319, 1269923],

     [484764, 698296, 460301, 550466, 0, 315262, 617247, 900634, 1247748],

     [270057, 708694, 566826, 494365, 312669, 0, 343838, 589118, 93306],

     [494349, 932986, 904386, 817784, 618029, 342106, 0, 724025, 836274],

     [403507, 730996, 1102078, 772338, 901058, 59016, 724548, 0, 632475],

     [871633, 1228601, 1599683, 1269943, 1248796, 937898, 836796, 632603, 0]

] #distance between cities

vehicles = [50, 25, 25] #vehicle capacities

demands = [0, 15, 13, 8, 12, 7, 11, 9, 10] #demands of cities

def simulated\_annealing(distance\_matrix, vehicles, demands, num\_iterations=10000, alpha=0.99, initial\_temperature=100): #to define the SA function

    def total\_distance(route, distance\_matrix): #takes the parameters of a route and distance\_matrix and to calculate the total distance of the given route

        distance = 0 # to calculate the total distance of the given route

        for i in range(len(route) - 1): # to loop every element in the route list except the last element

            distance += distance\_matrix[route[i]][route[i + 1]] #to calculate distance from current city to next city in loop

        return distance #to return the total distance

    def swap\_two\_nodes(route): #to create a new route by taking a route parameter and randomly swapping two cities on the route

        new\_route = route[:] #to create a new route without changing the original route

        i, j = random.sample(range(1, len(route) - 1), 2) #to randomly assign and randomly select two different indexes in the route

        new\_route[i], new\_route[j] = new\_route[j], new\_route[i] #to relocate

        return new\_route #to generate new solutions

    def generate\_initial\_solution(distance\_matrix, vehicles, demands): #to generate a random solution initially by assigning cities to random vehicles

        cities = list(range(1, len(distance\_matrix))) #to create a city list

        random.shuffle(cities) #to mix the cities so that the initial solution is random

        routes = [] #to include the route of each vehicle in the solution

        vehicle\_idx = 0 #to represent the index of the current vehicle in the vehicles list

        remaining\_capacity = vehicles[vehicle\_idx] #to keep track of how much load a vehicle can carry and when it will be full

        current\_route = [0] #to represent the current vehicle's route and 0 depots i.e. starting city

        for city in cities: #to start a loop to check the requests on each city in the cities list

            demand = demands[city] #to be used to compare the city's demand with the remaining capacity of the current vehicle

            if remaining\_capacity - demand >= 0: #to check if the remaining capacity of the existing vehicle can meet the city's demand

                current\_route.append(city) #if the vehicle can meet the city's demand, to add the city to the existing route

                remaining\_capacity -= demand #to subtract the city's demand from the remaining capacity of the current vehicle

            else:

                current\_route.append(0) #ends the current route and to return the route to the repository

                routes.append(current\_route) #to add the current route to the list of routes

                vehicle\_idx += 1 #to switch to the next vehicle

                if vehicle\_idx < len(vehicles): #if there are more available tools to run the code in this block

                    remaining\_capacity = vehicles[vehicle\_idx] #to update the current vehicle's capacity

                else:

                    break

                current\_route = [0] #to start a new route and set the warehouse as the starting city

                current\_route.append(city) #to add the current city to the new route

                remaining\_capacity -= demand #to subtract the city's demand from the remaining capacity of the current vehicle

        if current\_route: #to run the codes in this block if the last created route is not empty

            current\_route.append(0) #to finish the last route and return the route to the repository

            routes.append(current\_route) #to add the last route to the list of routes

        return routes #to return the starting routes

    def calculate\_total\_distance(routes, distance\_matrix): #to calculate the total distance of all routes

        return sum(total\_distance(route, distance\_matrix) for route in routes)

    routes = generate\_initial\_solution(distance\_matrix, vehicles, demands) #to create the starting routes using the given

    temperature = initial\_temperature #to set the starting temperature

    for \_ in range(num\_iterations): #to perform the specified number of iterations

        new\_routes = [swap\_two\_nodes(route) for route in routes] #to create new routes by randomly swapping two nodes on existing routes

        current\_distance = calculate\_total\_distance(routes, distance\_matrix) #to calculate the total distance of available routes

        new\_distance = calculate\_total\_distance(new\_routes, distance\_matrix) #to calculate the total distance of new routes created

        if new\_distance < current\_distance: #to check if the total distance of new routes is less than the total distance of existing routes

            routes = new\_routes #to consider the new routes as existing routes if the total distance of the new routes is lower

        else:

            probability = math.exp(-(new\_distance - current\_distance) / temperature) #to calculate the probability that new routes will be accepted

            if random.random() < probability: #to decide whether to accept new routes by comparing a random number with the probability of being accepted

                routes = new\_routes #to update existing routes if new routes are accepted

        temperature \*= alpha #to control the rate of temperature drop

    return routes, calculate\_total\_distance(routes, distance\_matrix) #to return the best found routes and the total distance of those routes

routes, total\_distance = simulated\_annealing(distance\_matrix, vehicles, demands) #to calculate the best routes and the total distance using the function

print("Optimal routes:")

for i, route in enumerate(routes, 1): #to get the index of each route and the route using the list of routes in a loop by starting the indexes at 1

    print(f"Vehicle {i}: {' -> '.join(map(str, route))}") #to convert all elements to string and print the route of each vehicle by separating all cities with ->

print(f"\nTotal distance: {total\_distance}")

**OUTPUT**

Optimal routes:

Vehicle 1: 0 -> 7 -> 5 -> 8 -> 4 -> 0

Vehicle 2: 0 -> 3 -> 1 -> 0

Vehicle 3: 0 -> 6 -> 2 -> 0

Total distance: 4362929

The total distance traveled by all vehicles is 4,362,929 units. This solution minimizes the overall distance while satisfying the demands of each city and considering the vehicle capacities. The simulated annealing algorithm proves to be an effective approach for solving such complex optimization problems.

References

[1] Lijun Wei, Zhenzhen Zhang, Defu Zhang, Stephen C.H. Leung, A simulated annealing algorithm for the capacitated vehicle routing problem with two-dimensional loading constraints, European Journal of Operational Research, Volume 265, Issue 3, 2018, Pages 843-859, ISSN 0377-2217, <https://doi.org/10.1016/j.ejor.2017.08.035>.

[2] R. Tavakkoli-Moghaddam, N. Safaei, Y. Gholipour, A hybrid simulated annealing for capacitated vehicle routing problems with the independent route length, Applied Mathematics and Computation, Volume 176, Issue 2, 2006, Pages 445-454, ISSN 0096-3003, <https://doi.org/10.1016/j.amc.2005.09.040>.

[3] Mohammed, M.A., Ghani, M.K.A., Hamed, R.I., Mostafa, S.A., Ahmad, M.S., & Ibrahim, D.A. (2017). Solving vehicle routing problem by using improved genetic algorithm for optimal solution. Journal of Computational Science, 21, 255-262, <https://doi.org/10.1016/j.jocs.2017.04.012>.

[4] M. A. Mohammed, M. S. Ahmad and S. A. Mostafa, "Using Genetic Algorithm in implementing Capacitated Vehicle Routing Problem," 2012 International Conference on Computer & Information Science (ICCIS), Kuala Lumpur, Malaysia, 2012, pp. 257-262, doi: 10.1109/ICCISci.2012.6297250.

[5] Alonso, F., Alvarez, M. & Beasley, J. A tabu search algorithm for the periodic vehicle routing problem with multiple vehicle trips and accessibility restrictions. J Oper Res Soc 59, 963–976 (2008), https://doi.org/10.1057/palgrave.jors.2602405.

[6] Akbar, M.D., & Aurachmana, R. (2020). Hybrid genetic-tabu search algorithm to optimize the route for capacitated vehicle routing problem with time window. International Journal of Industrial Optimization, 1(1), 15-28. https://doi.org/10.22190/IJIO.200115.