END478E

Applied Optimization Models

Sabancı Optimization Challenge’24

The Vehicle Routing Problem with Flexible Time Windows

Prof. Dr. Burhaneddin Sandıkçı

Course Assistant: Kübra Çetin Yıldız

CRN: 20933

Date: 23.05.2024

İbrahim Berk Özkan 70200021

Oğuzhan Kahraman 70200065

Samed Kocaman 70200017

1. Used Software and Algorithms:

The problem was solved using the Python programming language. The Cheapest Insertion and Simulated Annealing algorithms were employed to reach the solution. All these applications were executed on Spyder (Python 3.11). For each instance, vehicle capacity and customer information were converted from txt files to Python files using the Format_Converter.jpynb.

2. Solution Approach:

To produce faster and more stable results with the Simulated Annealing algorithm, a feasible initial solution that meets the problem's constraints is generated using the Cheapest Insertion algorithm. This initial solution is then inputted into the Simulated Annealing algorithm, which aims to improve the routes and approach optimality. Our code runs the Simulated Annealing algorithm a total of 1000 times. Each run may use the solution from the previous run as its starting point if it is better than all preceding solutions. For example, our initial solution from the Cheapest Insertion algorithm is used as the starting solution for the first run of the Simulated Annealing process. If the second run of the Simulated Annealing generates a better solution, it will then serve as the starting solution for the third run, and so on, until the total number of runs is reached. Initially, a "cheapest insertion" algorithm is used to create a valid solution, which is then refined by the "simulated annealing" algorithm. Upon completion, the code provides the "routes" and "cost" outputs.

3. Used Algorithms:

3.1 Cheapest Insertion Algorithm:

This algorithm, as the name suggests, aims to find the lowest-cost insertion. The algorithm follows these steps:

Initial State: Each customer is marked as unvisited, except for the starting point (usually a depot), and no services are provided to any customer yet.

Finding the Cheapest Insertion: At each step, the customer with the lowest cost is added to the current routes. The cost is calculated based on the distance increase between the existing customers and the added customer, and the constraints of the time windows.

Unvisited Customers: Once a customer is added to a route, they are marked as visited and are not considered in the next selection.

Route Completion: This process continues until all customers are visited or the vehicle capacity is reached.

3.2 Simulated Annealing Algorithm:

The Simulated Annealing is used to optimize the initially obtained solution. Essentially, it explores the solution space randomly, applying various changes to seek better solutions. This algorithm involves:

Starting Temperature and Cooling: The process starts at a certain temperature, which is reduced after each iteration. The temperature affects the probability of accepting changes.

Mutation: Random changes are made to the existing routes (e.g., swapping the positions of two customers, reversing a section of the route).

Acceptance Criteria: If the new solution is better than the current one, or is acceptable with a certain probability (dependent on the temperature and the quality of the solution), it is updated as the current solution.

Improvement: The best-found solution is continuously recorded.

Total Operation Period: The code repeats the simulated annealing process 1000 times, using the best solution from each iteration as the starting solution for the next.

Parameters Used in the Simulated Annealing Algorithm:

max_iterations=50000: This parameter determines the duration the algorithm will run. A higher iteration number allows for a broader search in the solution space, potentially improving the chance of finding a better global optimum.

start_temp=5000: The starting temperature dictates the initial randomness level and the likelihood of accepting worse solutions, allowing for a broader exploration of the solution space initially.

cooling_rate=0.95: This rate determines how much the temperature decreases after each iteration, controlling how the search behavior changes over time. A 0.95 rate means a 5% reduction per iteration, providing a relatively slow cooling process that allows for a more detailed exploration of the solution space and increases the chances of finding better solutions.

4. Detailed Examination of Python Code and Used Functions:

-calculate_distance(customer1, customer2):

This function calculates the Euclidean distance between two customers. It uses the coordinates (x and y) provided in the parameters for customer1 and customer2, computes the distance, and returns the rounded result.

-calculate_insertion_cost(route, customer_to_insert, insert_position, customers):

This function calculates the cost of inserting a specific customer into a specific position in a route. The cost is derived from the increased distances due to the new customer's addition compared to the previous distances.

-calculate_route_cost(route, customers):

This function calculates the total cost for a given route. The total cost of a route comprises the total distance traveled by the vehicle while visiting customers and a fixed cost. The total distance is calculated as the sum of the distances the vehicle travels to visit each customer. The fixed cost is computed as 2nc_max, where n is the number of customers in the instance, and c_max is the maximum distance between two consecutive customers on the route.

-is_insertion_feasible(route, customer_id, position, customers, current_time, vehicle_capacity, depot_due_date):

This function checks whether it is feasible to insert a specific customer at a specific position. It considers constraints such as vehicle capacity, customer readiness time, and service duration. It also checks whether the vehicle's return time to the depot meets the depot due date. If the customer is added, the new route's timing is checked for compatibility with the customers' demands.

-cheapest_insertion(customers, vehicle_capacity):

This function starts with an empty list of routes and uses the least costly insertion method to add customers to routes. At each step, the least costly customer is selected and inserted into the appropriate position. This process continues until all customers have been visited.

-simulated_annealing(customers, initial_routes, initial_cost, vehicle_capacity, max_iterations=50000, start_temp=5000, cooling_rate=0.95):

This function improves the given initial routes using the simulated annealing algorithm. It applies various mutations to the routes and assesses the results of these mutations. As the temperature value decreases, the probability of accepting new solutions diminishes.

-mutate_routes(routes, customers, vehicle_capacity):

This function applies random mutations to the existing routes to produce new routes. These mutations might include swapping customers' positions, reversing routes, or adding customers randomly. The resulting routes are checked for compliance with vehicle capacity, customer timing constraints, and the vehicle's return limit to the depot through the validate_route function.

-validate_route(route, customers, capacity):

This function checks the constraints for a given route. If the route is valid, it returns True; otherwise, False. The checks include whether customer demands are met on time and whether the vehicle's capacity is exceeded.

-calculate_total_cost(routes, customers):

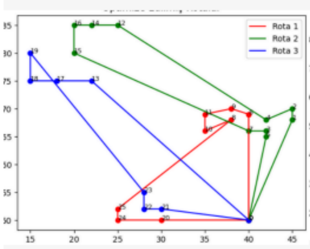
This function calculates the total cost for all given routes. Each route's cost is computed using the calculate_route_cost function.

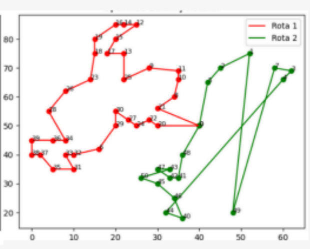
-run_multiple_simulations(num_simulations):

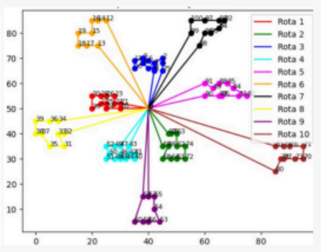
This function repeats the simulated annealing process a specified number of times. Each iteration uses the best solution from the previous iteration as the starting point. The best solution is continually updated.

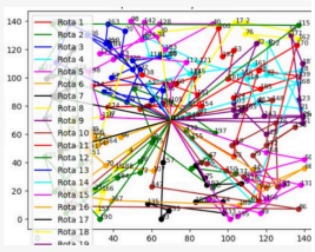
Computational Analysis

Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 16.0 GB RAM PYTHON









instance 1

Route #1: 20 24 25 23 22 21
Route #2: 5 3 7 8 10 11 9 6 4 2 1
Route #3: 13 17 18 19 15 16 14 12
Cost 6239.7

25 Customer
20000 Run
~1000 Seconds

instance 7

Route #1: 20 22 24 27 30 29 6 32 33 31 35 37 38 39 36 34 28 26 23 18 19 16 14 12 15 17 13 25 9 11 10 8 21
Route #2: 5 2 1 49 7 3 4 44 40 46 45 50 43 47 42 41 48
Cost 14650.52

50 Customer
20000 Run
~1500 Seconds

instance 12

Route #1: 81 61 68
Route #2: 95 67 94 90
Route #3: 69 98 88 53 78 60
Route #4: 82 52 99 57 86 77 25 48
Route #5: 72 71 38 37 35 43
Route #6: 2 45 5 8 7 46 3 1 4 100 70
Route #7: 14 47 15 16 11 9 10 13 17
Route #8: 42 39 36 40 44 41 54 96
Route #9: 64 22 19 49 20 24 91 80
Route #10: 31 29 27 30 50
Route #11: 59 75 87 97 58 74
Route #12: 12 73 79 6 55
Route #13: 65 83 23 21 18 89
Route #14: 92 62 33 28 26 32 34 93
Route #15: 63 76 51 85 84 56 66
Cost 305265.86

100 Customer
20000 Run
~4000 Seconds

instance 18

●
●
●

200 Customer
20000 Run
~9000 Seconds

The image illustrates the results of solving the "Vehicle Routing Problem with Flexible Time Windows" using a heuristic optimization model in Python. Each instance represents a different problem size, with varying numbers of customers:

> **Instance 1**: 25 customers, solved in ~1000 seconds, with a total cost of 6239.7.
>
> **Instance 7**: 50 customers, solved in ~1500 seconds, with a total cost of 14650.52.
>
> **Instance 12**: 100 customers, solved in ~4000 seconds, with a total cost of 30235.86.
>
> **Instance 18**: 200 customers, solved in ~9000 seconds.

Each graph shows the routes for vehicles serving the customers, indicating the complexity and computational effort required to solve larger instances. The different routes and associated costs highlight the performance of the heuristic optimization in finding feasible solutions efficiently.

The computational time increases approximately linearly with the number of customers due to the following reasons:

> **Complexity**: Each additional customer adds complexity to the routing problem, requiring more calculations to determine optimal routes.
>
> **Heuristic Search Space**: As the number of customers grows, the heuristic algorithm needs to explore a larger search space to find near-optimal solutions, increasing the time needed for each run.
>
> **Iterations**: The heuristic algorithm likely performs a set number of iterations (20,000 runs) for each problem size, and more customers mean more data to process in each iteration.

These factors contribute to the near-linear increase in computational time as the number of customers rises. Because when adding another customer does not add another dimension like we do in simplex, it does not increase exponentially.