Miljenko Šuflaj
0036501442

Heuristic Optimization Methods

Project report: Capacitated Vehicle Routing Problem with Time Windows

Zagreb, January 2021

Contents

1 Introduction

This task was solved using **Python 3.8.5**.

2 Problem description

We are 2 types of information: system information and customer information

2.1 System information

In a system, we are given a number of vehicles and vehicle capacity. Each vehicle has the same capacity.

2.2 Customer information

Customer informations consists of a septuple, all integers:

- customer number
- x coordinate
- y coordinate
- demand
- ready time
- due time
- service time

Additionally, a customer with the customer number 0 is a *depot*.

2.3 Problem formulation

Given some system information and a list of customer information, we are to construct a list of routes starting from the *depot* and ending in *depot* visiting all customers exactly once. A route is valid if:

- every time point a customer is visited is between its ready time and due time (both inclusive)
- the sum of customer demands is less or equal to vehicle capacity (we'll refer to this as *cost* from now on)

A solution is valid if the number of routes in it is less or equal to the number of vehicles given to us as part of the system information. Every time we visit a root, we have to spend time equal to service time on it. After that, we can move elsewhere. Also, there is travel time equal to the $d(x, y)$, where $d(\cdot, \cdot)$ represents

a distance function between 2 customers. In our case, the distance function is the *euclidean norm*.
We rank solutions by sorting them by $|solution|$ ascending, $\sum d(solution)$ ascending. In other words, we value smaller solutions primarily, and then solutions with a smaller total distance.

2.4 Additional details

There are some additional details that are a consequence of our problem formulation.

- every route starts at the depot which is serviced at time=0
- service at a customer will start no sooner than its ready time (even if we arive at the customer sooner)
- service at a customer can end after due time

3 Algorithm

To solve this problem 2 algorithms were used:

- a greedy, deterministic algorithm (from now on called Greedy)
- a greedy, non-deterministic algorithm (from now on called Merger)

3.1 Greedy algorithm

The idea of this algorithm is fairly simple - first we preprocess the customer list in order to get rid of infeasible customers. These are customers whose service cannot be finished before the depot is closed. Then we define a pick-order, a degenerated priority queue so to say - we sort all of our customers (aside from *depot* in the order: **due time** ascending, ready time - service time ascending, **demand** descending. This will be an ordered list of candidates, ordered in a way we greedily access those whose due time is sooner, who are ready sooner and which have a greater demand.
Then we enter the solution construction phase - we construct a number of routes, all in the same way.

3.1.1 Route construction

We look at *two* candidates:

- **first** (referring to the first available customer in our priority list)
- **nearest** (referring to the feasible customer nearest to our last customer)

First we determine which of these candidates are feasible - if only one of them is, we automatically select them to be the next customer. If none are feasible, then we end the current route creation, as it means that no other customer can be added in a way that the route remains valid. If both are feasible, then we *negotiate*. We want to take the one which has the higher demand. If that's **first**, we take it no questions asked. However, if **nearest** has a higher demand, then we take it *only if we can take **first** instantly after*.

Initially, the reasoning for this condition is that we want to get rid of customers with early due times as soon as possible to give the algorithm more space in choosing better routes. Later we noted that it does not necessarily improve the algorithms performance, but empirically we saw significantly better results with than without it.

Finally, after a route is built, we check if the last customer was the *depot*. If this happens, then our route construction failed, we throw out a warning that we couldn't find a solution and we return whatever we had until now. If this doesn't happen, we make the last customer the *depot* and we add the route to our list of routes.

3.1.2 Evaluating customer candidate feasibility

For the customer candidate to be feasible, it has to obey certain rules:

- service must start at a time between ready time and due time, inclusive
- its demand must be lower than *max cost*
- it must be possible to go from the customer to the *depot* before it closes

Variable *max cost* is calculated via formula:

$$C_{max} = c_{vehicle} - C_{route} \tag{1}$$

where C stands for *capacity*. Therefore, the pseudocode for a function checking a customer candidate's feasibility would be:

```
def is_feasible(vehicle_capacity, depot, route, customer):
    service_start = (
        route.last.serviced_at
        + route.last.service_time
        + ceil(d(route.last, customer))
    )

    if customer.ready_at <= service_start <= customer.due_time:
        max_cost = vehicle_capacity - route.cost

        if customer.demand <= max_cost:
            service_end = (
                service_start
                + customer.service_time
                + ceil(d(customer, depot)
            )

            if service_end <= depot.due_time:
                return True

    return False
```

3.1.3 Pseudocode

Now that we've defined that function, we can roughly sketch the pseudocode of the Greedy algorithm. Again, we'll start on the next page so it fits in 1 page.

```
def greedy(system, customers):
    depot = customers.depot

    eligible_customers := customers that can end before depot closing
    customer_pick_order = sorted(
        eligible_customers,
        key=lambda x: (x.due_time, x.ready_time - x.service_time, -x.demand)
    )

    routes = []

    while there are non-serviced customers in customer_pick_order:
        current_route = Route()
        current_route.add(depot)

        while there are non-serviced customers in customer_pick_order:
            first = first non-serviced customer in customer_pick_order
            nearest = nearest feasible customer

            if any(is_feasible(x) for x in (first, nearest)):
                chosen = optimal candidate
            else:
                break

            current_route.add(chosen)

        if current_route.last == depot:
            warn no solution
            break
        else:
            current_route.add(depot)
            routes.append(current_route)

    return routes
```

Again, we note, that the optimal candidate is chosen as described in the section 3.1.1.

3.2 Merger algorithm

This is an improvement algorithm. The idea behind this algorithm is a bit more convoluted than the Greedy algorithm. The idea is to select at least 2 routes and then try and construct less routes than initially given, or at least routes with better customer distribution than before. The way this is done is by creating a customer pool, similarly to the Greedy algorithm. The construction phase of the algorithm is identical to the construction phase of the Greedy algorithm.

The difference is the non-deterministic nature of the algorithm. Before, customers were queue according to a deterministic rule. Now, we access customers in a random order. Firstly, we record the number of iterations no improvement was seen, let's refer to this as n_{eq}. Then, we select a random permutation of routes with the distribution of $U \sim [2, n_{eq}]$. We forward these routes to a construction algorithm and evaluate the original, as well as the new routes with an objective function. If the new routes are better, we return them, and `None` instead (signifying there was no change).

3.2.1 Objective function

As said in the problem description, we value compact solutions first and foremost, and then short solutions. One of possibilities for a function is the following:

```
def loss(routes):
    length_portion = routes.length * 1000000
    cost_portion = sum(
        abs(route.cost - routes[i - 1])
        for i, route in enumerate(routes[1:])
    )

    return length_portion - cost_portion
```

Named loss, one can assume that lower values of this function describe better routes. Firstly, the smaller the length, the smaller the `length_portion`. Secondly, because we subtract `cost_portion` from `length_portion`, we want it to be bigger. In other words, we rank routes that have bigger absolute differences between neighbouring routes as better than those with more uniformly distributed costs. The motivation for this is that we want greater differences between routes in hopes that routes with smaller costs will be able to be merged with other moderately sized routes.

3.2.2 Pseudocode

As before, the pseudocode will be shown on the next page due to fitting purposes. Although the merging algorithm won't be shown (it's almost identical to the Greedy algorithm if you were to give it a restricted set of customers), we'll call it `merge_routes`.

```
def merger(system, routes, max_selected_routs):
    # Nothing to merge
    if len(routes) < 2:
        return None

    max_selected_routes = max(
        2,
        min(
            max_selected_routes,
            len(routes)
        )
    )

    # Exclusive upper bound requires the + 1
    n_routes = random.uniform(2, max_selected_routes + 1)
    selected_routes = random.permutation(routes, n=n_routes)
    other_routes = routes except the selected routes

    new_routes = merge_routes(selected_routes)

    # If this happens, no improvement happened
    if new_routes is None:
        return None
    else:
        return new_routes + other_routes
```

4 Results

Using the described heuristic algorithm, we calculated solutions for 6 instances:

- instance 1 - max vehicles: 25, capacity: 100
- instance 2 - max vehicles: 50, capacity: 200
- instance 3 - max vehicles: 100, capacity: 700
- instance 4 - max vehicles: 150, capacity: 1000
- instance 5 - max vehicles: 200, capacity: 700
- instance 6 - max vehicles: 250, capacity: 200

Earlier instances took the Greedy algorithm a few seconds to solved, while instances 6 took the greedy algorithm 2.5 minutes *on average* to solve. The algorithm was run for 1 minute, 5 minutes and 25 minutes (unlimited). The reason we limited the unlimited execution to 25 minutes is because there is no explicit end to the Merger algorithm. It runs infinitely. We also record the number of iterations it takes to reach our solution. Keep in mind that we consider

our Greedy algorithm to run for 1 iteration.

4.1 Results - 1 minute

We didn't manage to find a solution in under a minute for instances 5 and 6 - this is due to their greater size and difficulty. Other results are as follows:

- instance 1 - length: 25, distance: 3548.20, iterations: 237
- instance 2 - length: 28, distance: 8298.84, iterations: 136
- instance 3 - length: 26, distance: 14825.41, iterations: 27
- instance 4 - length: 26, distance: 33203.75, iterations: 16

We can see that as the iterations increase, the number of iterations done decreases because of the more time needed for the Greedy algorithm to conclude, as well as potentially larger routes the algorithm has to merge.

4.2 Results - 5 minutes

This time the algorithm found a solution for all instances. They are as follows:

- instance 1 - length: 24, distance: 3557.75, iterations: 882
- instance 2 - length: 28, distance: 8205.79, iterations: 283
- instance 3 - length: 24, distance: 16853.47, iterations: 79
- instance 4 - length: 26, distance: 33016.95, iterations: 37
- instance 5 - length: 53, distance: 35269.02, iterations: 40
- instance 6 - length: 128, distance: 104878.55, iterations: 54

We can see that all of the solutions are valid for the given constraints.

4.3 Results - indefinitely

The results don't vary wildly. They are:

- instance 1 - length: 23, distance: 3435.34, iterations: 4260
- instance 2 - length: 26, distance: 7909.55, iterations: 1375
- instance 3 - length: 23, distance: 16276.11, iterations: 324
- instance 4 - length: 25, distance: 32481.75, iterations: 158
- instance 5 - length: 53, distance: 35269.02, iterations: 88
- instance 6 - length: 128, distance: 104878.55, iterations: 119

We can see that all instance results managed to improve slightly aside from instances 5 and 6. Given more time perhaps they could improve a bit as well.

5 Conclusion

Although the algorithm runs slowly, it's not because it does a lot of things. It could likely be sped up severely by writing it in a faster language. Also, it wasn't extremely optimized. Sure, there was consideration put into it but it was likely not the best version that can be written in Python. Finally, the Greedy algorithm doesn't give us a lot to work with because of how naive and greedy it is. Oh well.

Considering this problem has a time constraint in its definition, perhaps a better initial solution could be obtained by making the search a bit more random. Secondly, because the problem is related to graph theory, perhaps some techniques from that area could speed the algorithm up, if not solve it better.

Finally, the biggest reason for a somewhat disappointing performance of the algorithm is the lack of time. Even now, these last words are being written with as the author slowly loses to sleep. But deadlines, no matter how nasty, must be met.