SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Capacitated Vehicle Routing Problem with Time Windows (CVRPTW) Solution Documentation

Rafael Josip Penić, Mario Pavečić

Zagreb, siječanj 2021.

Content

Description of the problem

The Vehicle Routing Problem (VRP) is a generic name given to a whole class of problems in which a set of routes for a fleet of vehicles based at one or several depots must be determined for a number of geographically dispersed cities or customers. The objective of the VRP is to deliver a set of customers with known demands on minimum-cost vehicle routes originating and terminating at a depot. In the two figures below we can see a picture of a typical input for a VRP problem and one of its possible outputs:

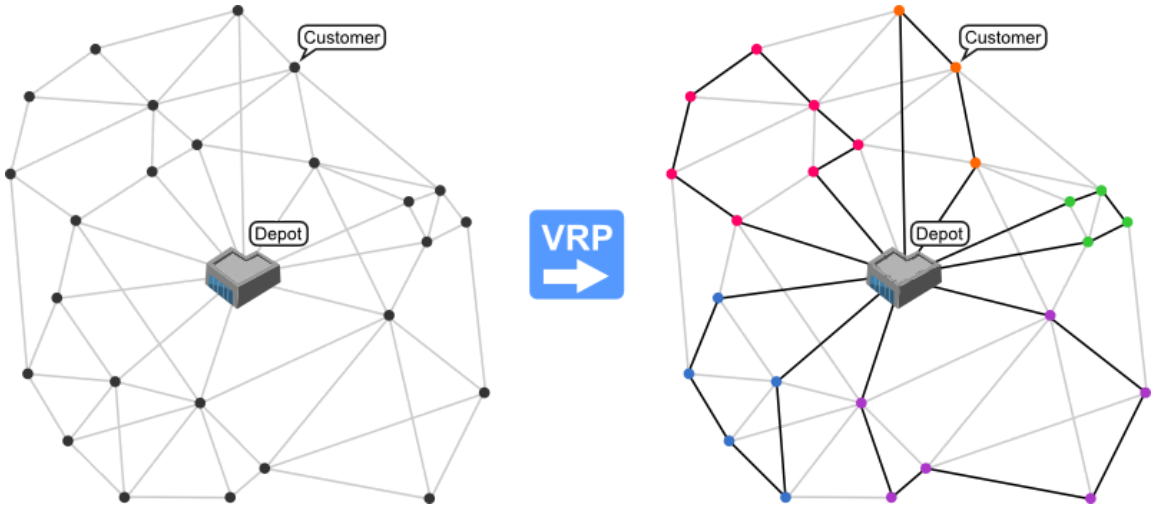

Figure 1.

CVRPTW belongs to the class of Vehicle Routing Problems (VRP), but, on top of the generic VRP formulation, defines additional constraints that occur in real-life scenarios. The capacity constraint in CVRP refers to the capacity of the vehicle: the total demand of all customers supplied on a single route must not exceed the vehicle capacity. On the other hand, the time window constraint in VRPTW refers to the interval in which a customer must be supplied (often called the “scheduling horizon”): all customers in a route need to be reached within their scheduling horizons, and the whole route needs to be started and finished within the working hours of the depot. CVRPTW includes both constraints.

---

*Figure and VRP description taken from http://neo.lcc.uma.es/vrp/vehicle-routing-problem

*CVRPTW description taken from https://www.fer.unizg.hr/_download/repository/HOM_project_2020.pd

Description of the implemented heuristic algorithm

To solve the CVRPTW problem, we used a simulated annealing algorithm.

First of all, we have to define a representation of a single solution. In our case, the solution is represented as a list of lists. For example, [[0, 4, 1, 2, 0], [0, 5, 3, 0], [0, 6, 0]] represents a solution with three routes. First vehicle visits customers 4, 1 and 2 (in that order), second visits 5 and 3 and third one only visits customer 6. Vehicles, of course, start and end their routes in customer 0 which represents a depot.

Secondly, we need to define which two solutions are considered neighbours and which are not. To generate a single neighbour of some solution, we “move” some chosen customer. We can move him “inside” of the current vehicle or we can put him in a certain position of another vehicle. As an illustration, let us consider the previously mentioned solution ([[0, 4, 1, 2, 0], [0, 5, 3, 0], [0, 6, 0]]). Some neighbours of this solution are [[0, 4, 1, 2, 0], [0, 5, 6, 3, 0]] (Customer 6 was moved from third to second route), [[0, 4, 2, 0], [0, 5, 3, 0], [0, 6, 1, 0]] (Customer 1 was moved from first to third route), etc. Notice how in the first example the third vehicle “vanished” because it no longer had any customers to serve. Naturally, when generating neighbours, we have to make sure neighbours satisfy given constraints.

In our algorithm, in each step we generate a random neighbour of the current solution. First, we randomly choose one of the vehicles and then we randomly choose a customer from the chosen vehicle. After that, we move this customer to a random position as previously explained. It is worth noting that this way of customer choice won’t give uniform distribution. In other words, not all customers will have an equal chance of being chosen. This is caused by the fact that we first choose a vehicle. That way, customers in vehicles with less customers will have a greater chance of being chosen and that is exactly what we want because that way, we will “move” towards solutions with less routes which is exactly what we want.

As with most heuristic algorithms, we needed an initial solution from which we would start our search. We generated such a solution with a greedy algorithm. Basically, all this algorithm does is force vehicles to first visit customers which have a minimum sum of their ready time and distance to the vehicle’s current position. At first, such “choice” might seem a bit odd but it gave better results than when we only took the customer's distance in consideration.

Objective function of our algorithm is the number of routes needed to service all the customers. In case two solutions use the same number of routes, a better solution is the one with a lesser total distance.

To reduce the temperature in each iteration, we used geometric and very slow temperature decrease.

initial solution	Initial solution is generated with a greedy algorithm.
neighbourhood definition	Example: [[0, 2, 1, 0], [0, 3, 0]] neighbours: - [[0, 3, 2, 1, 0]] - [[0, 2, 3, 1, 0]] - [[0, 2, 1, 3, 0]] - [[0, 1, 2, 0], [0, 3, 0]] - [[0, 1, 0], [0, 2, 3, 0]] - [[0, 1, 0], [0, 3, 2, 0]] - [[0, 2, 0], [0, 1, 3, 0]] - [[0, 2, 0], [0, 3, 1, 0]]
temperature decrement function	Geometric decrement : $T = \alpha * T$ Very slow decrease : $T = T / (1 + \beta * T)$
stopping criteria	Algorithm stops once the temperature is low enough. (e.g. 0.01)

Pseudocode

```
main() {
  instance_file_name = readFromInput()

   instance = create_instance_object_from_file(instance_file_name)
   instance.find_initial_solution()

   solution = simulated_annealing_algorithm(instance)
   print(solution)
}
```

```
class Customer:
   init(cust_no, x, y, demand, ready_time, due_date, service_time):
      cust_no = cust_no
      is_served = False
      vehicle_num = null

   served(vehicle_num):
      self.is_served = True
      self.vehicle_num = vehicle_num

   unserve(self):
      is_served = False
      vehicle_num = null
```

```
class Vehicle:
   init(self, id, depo, max_capacity):
      x = depo.x
      y = depo.y
      capacity = max_capacity
      last_service_time = 0
      service_route = [(depo, 0)]
      total_distance = 0

   serve_customer(self, customer):
      if last service time + distance to the customer is not in customer service interval or demand is greater then
vehicle capacity or there will be no time to return to depo:
         return False
      update vehicle position (x, y), last_service_time, capacity, service_route and total distance
      customer.served(id)
      return True

   serve_customer_force(self, customer):
```

```
    if customer ready time is greater then distance to vehicle + last_service_time:
        temp = last_service_time
        last_service_time = customer.ready_time - ceil(distance(customer, self))
        if serve_customer(customer):
            return True
        else:
            last_service_time = temp
    return False

remove_customer(self, customer):
    remove customer from route and update values for other points in the route



try_to_serve_customer(self, new_customer):
    if service_route is empty:
        return serve_customer(new_customer) or serve_customer_force(new_customer)
    for i in shuffled service_route indexes:
       try to serve customer by placing it in the index i in service route
        if serving is successful:
          return True
    return False
```

```
class Instance:
  init(num_vehicles, capacity, customer_list):
    num_vehicles = num_vehicles
    capacity = capacity
    vehicles = initialize list of Vehicles
    this.customer_list = customer_list

  find_initial_solution(self): // Use greedy search to find initial solution
    for i, vehicle in enumerated vehicles:
      while True:
        customer_list.sortBy((c) -> distance(c, v) + c.ready_time)
        found = False
        for customer in customer_list:
          if customer is served or customer is depo:
            continue
          if vehicle.serve_customer(customer):
            found = True
            break
        if not found:
          for customer in customer_list:
            if customer is served or customer  is depo:
              continue
            if vehicle.serve_customer_force(customer):
              found = True
              break
        if not found:
          break
```

```
            customer_list.sort((c) -> c.cust_no)
            if all_served(self.customer_list[1:]):
                break
        self.customer_list.sort(key = lambda c: c.cust_no)

        for vehicle in self.vehicles:
          if vehicle not in depo:
              vehicle.return_home()



    generate_random_neighbour(self):
        rand_cust = choose random customer from customer_list, excluding depo
        current_serving_vehicle = vehicles[rand_cust.vehicle_num]
        current_serving_vehicle.remove_customer(rand_cust)
        v = None
        while rand_cust is not served:
            if self.get_neighbour(rand_cust):
                return
            self.get_neighbour(rand_cust, True)

    get_neighbour(self, customer, useEmptyTimeSlot=False):
        shuffled_vehicles = randomly shuffle vehicles
        for vehicle in shuffled_vehicles:
            if useEmptyTimeSlot or vehicle.last_service_time != 0:
                if vehicle.try_to_serve_customer(customer):
                    return True
```

```
objective_function(number of vehicles, total distance):
  return number of vehicles * total distance
```

```
simulated_annealing_algorithm(instance, temp_start = 100, update_temp = lambda t : 0.999 * t, stop_criterion =
lambda t : t <= 0.001):
    curr_solution = incumb_solution = copy instance

    curr_dist, curr_vhcls = incumb_dist, incumb_vhcls = curr_solution.get_total_distance_and_vehicles()

    temp = temp_star

    while not stop_criterion(temp):
        neighbour = copy curr_solution
        neighbour.generate_random_neighbour()

        neighbour_dist, neighbour_vhcls = neighbour.get_total_distance_and_vehicles()

        neighbour_obj = objective_function(neighbour_vhcls, neighbour_dist)
        curr_obj = objective_function(curr_vhcls, curr_dist)
```

```
    if (neighbour_obj < curr_obj) \
       or random.random() < e ^ (- (abs(curr_obj - neighbour_obj)) / temp):
         curr_solution = neighbour
         curr_dist, curr_vhcls = neighbour_dist, neighbour_vhcls

         if (curr_obj < objective_function(incumb_vhcls, incumb_dist)):
                  incumb_solution = curr_solution
                  incumb_dist, incumb_vhcls = curr_dist, curr_vhcls


    temp = update_temp(temp)

return incumb_solution
```

Analysis of results and discussion

Initial solution is generated using a greedy algorithm which uses the sum of ready time and distance between each customer and current vehicle position. This gave us a pretty satisfying initial solution, but of course it could be improved. To improve this solution we used a simulated annealing algorithm which chooses neighbours, with some random factor, from search space.

As simulated annealing uses a random factor, to get the best results from it, input parameters of this algorithm need to be adjusted. Initial temperature and its decreasing is the most important thing to be adjusted. To get the best results we needed to test out different parameters for this. Parameters can't be generalized, so for each problem instance there are different parameters that get the best results.

In this project, six different problem instances were given for the solution algorithm to be tested on. Most obvious difference in problem instances is the number of customers to be served and the number of vehicles available for serving customers. For example, in the first instance there are only 100 customers and 25 vehicles. In this case the initial temperature can be a lot lower than in other, more complicated, problem solutions since search space is a lot smaller. However, a bigger temperature will also work on this solution, but the algorithm will find a better solution while the temperature is still high, and then "jump" around the local optimum it found. Downside of setting temperature high for smaller problem instances is that execution time will be the same as with some lower temperature, but the solution will mostly be the same value as if we started with lower temperature.

When it comes to temperature decrease procedures, there are multiple options. We decided to try out geometric ($T = \alpha * T$) and very slow $(T / (1 + \beta * T))$ decrease. Both procedures gave somewhat similar results but geometric decrease proved to be faster (which was to be expected).

At first, objective function might sound a bit problematic to define since we are minimizing two values at the same time (total distance and number of vehicles) and we also have to consider that one of those values (number of vehicles) is more important than the other. Question is, how are we going to define a function which will take all these things into consideration. First thing we tried was the sum of total distance and number of vehicles. This did not work well because it often prefered solutions with greater numbers of vehicles. In other words, it focused too much on total distance. After that, we tried to multiply the values. This objective function gave good results. In theory, this objective function might (in some cases) prefer solutions with greater number of vehicles if such solutions had relatively low total distance but in reality neighbouring solutions usually have relatively close total distances so it is not a problem.

Conclusion

Simulated annealing worked as expected, decreasing the number of vehicles used to serve all customers as well as distance traveled gotten from initial solution, which was much better than expected.

One further improvement of the current algorithm could be reheating, which would mean to remember the temperature on which we got the best found solution and when the temperature hits the stopping criteria, reset it to that temperature. And repeat this for some predefined number of times.

Another improvement could be using some libraries optimised for working with lists, like numpy, to make operations quicker and execution time shorter.