Step 7 Summary

Note: There was a systematic error affecting all files of step 7. In the class DeliveryCentre two variables self.parcels and self.PARCELS were mixed up. This resulted in trace messages using parcel numbers 2, 3, etc. instead of 0, 1, etc. and the histParcelDeliveryDelay() method either producing wrong results or crashing.

The correction affects the class DeliveryCentre:

```
class DeliveryCentre:

    def __init__(self, rec, M, Maps, W, C, D,
                 limit, timeLimit):
        self.rec = rec
        self.M = M
        self.Maps = Maps
        self.W = W
        self.C = C
        self.D = D

        self.limit = limit
        self.timeLimit = timeLimit

        self.overhang = []   # list of parcels to be delivered next day

        # generate and initialise all customers
        self.customers = [ Customer(rec, i, C[i]) for i in range(len(C)) ]

        # generate and initialise all drivers
        self.drivers = [ Driver(rec, i, self) for i in range(len(Maps)) ]

        self.PARCELS = []    # registry of all the parcels processed

        rec.env.process(self.process())
```

...

```
    def process(self):
        for day in range(len(self.D)):
            yield self.rec.env.timeout(nextHour(self.rec.env, 17.00))
            # make plan how to split workload for the day
            regions = splitCustomers(self.C, self.Maps)

            # initialise the workload for all drivers
            self.leftOver = [ [] for driver in self.drivers ]   # list of par
            self.parcels = [ [] for driver in self.drivers ]    # list of par
            self.dest = [ [] for driver in self.drivers ]       # list of uni
            self.tour = [ [self.W] for driver in self.drivers ] # tour planne

            # process overhang from previous day (if any)
            for p in self.overhang:
                driverId = regions[p.cust.id]
                self.__accept(driverId, p)
            self.overhang = []

            # generate the parcels newly arriving this day
            for c in self.D[day]:
                parcel = Parcel(self.rec, len(self.PARCELS),
                                day, self.customers[c])
                self.PARCELS.append(parcel)
                driverId = regions[c]
                parcel.arrivedAtDeliveryCentre(self.drivers[driverId])
                self.__accept(driverId, parcel)
```

And the class Recorder:

```
def histParcelDeliveryDelay(self):
    histPlot(self.parcel['delivery delay'].dropna(),
             discrete=True,
             xlabel='Days',
             title=f'Parcel Delivery Delay in Days ({self.parcels:3,d} Parcels)')
```

Please update your files manually or just download the files from Project 15/7/2024.

Step 7A Adapting Codebase for Multiple Drivers

Some of the data we collect in the Recorder class are specific for a driver, others are (as before) totals for all drivers. This requires in the constructor of the class Recorder an additional parameter with the number of drivers, and all daily tables become daily tables per driver. All the record...() routines affecting daily data, take as an additional parameter a driver object. Similarly the plotting routines take as additional (optional) parameter the number of a driver. When no driver is provided (default value None), the routine produce a cumulative value, which in general gives the same output as before.

To maintain the overall structure of the code of the plot and histogram routines in the recorder class, we introduce two private methods *__Title*() and *__Data*() that handle the summation or selection of data depending on the given parameter driver.

For the load distribution between multiple drivers we assume that we have a list of submaps (regions) of the original map M. Each driver is assumed to operate only within one region. The customers are split by region as well. Should a customer location be in multiple regions, a random choice is made. This allocation of customers is done at the beginning of every day, so that in case of imbalances in customer locations the workload is still reasonably fair allocated. The new section 6 handles the splitting of the customer locations.

In Step 7A we change the representation and the coding to work with a number of drivers but test it initially just with one driver (with fixed index 0).

Some minor improvements have been implemented in the plotting routines.

Step 7B Random Allocation

There is no difference in the code base between 7A and 7B. The only difference is in the test section. In 7A we are just testing with a single driver. While in 7B we are also testing with two drivers (were appropriate. In section 7B we assume identical regions hence operate with drivers with a random allocation of delivery tasks.

We should see a difference in 12.4.3 which is an absolutely instable system if we have only one driver available. Check the output from the various plot routines for individual drivers and the corresponding total plots.

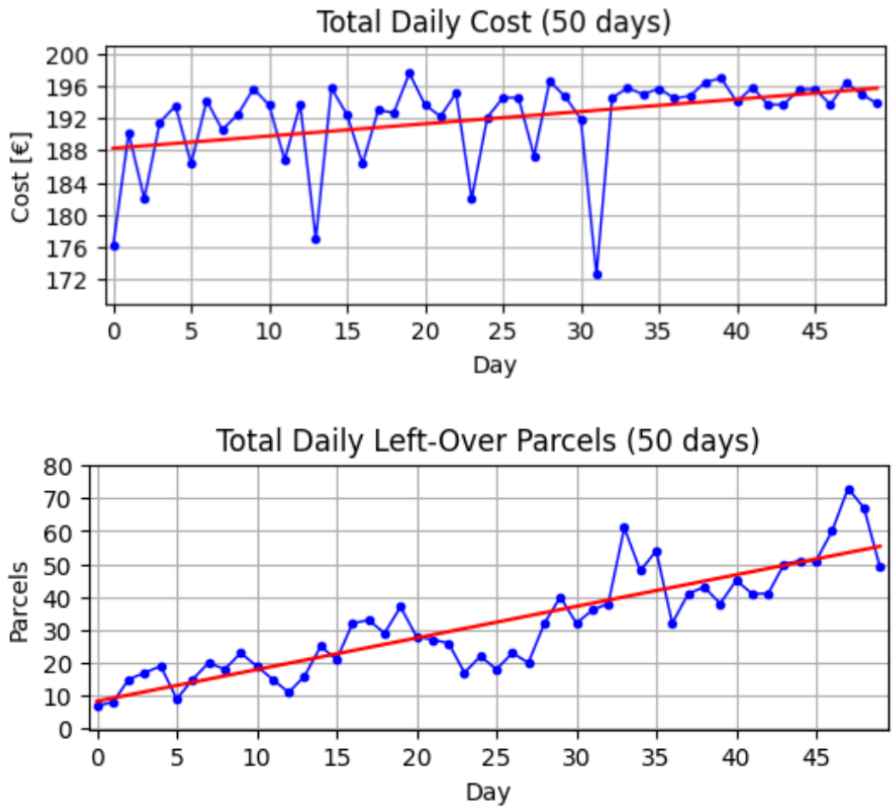


Figure 1 System Performance with Random Driver Allocation

You can use 7A/7B as the base for your own further refinement. Using two drivers with random allocation makes the simulation of the High Demand System 12.4.3 nearly stable.

Step 7C Manual Static Allocation

To improve the performance we can try a manual split of the overall map into regions like shown below.

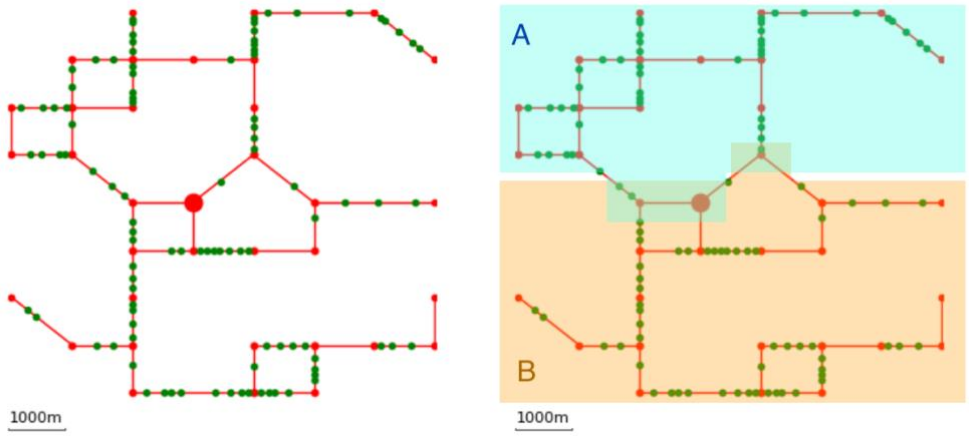


Figure 2 Plan for Static Driver Allocation

I got some questions to Step 7C indicating that the idea of constructing overlapping submaps A and B wasn't clear enough. You need to construct your own A and B matching your map M. Step 7 Static Allocation v2 contains additional comments and explanations and tests if your choice of A and B is ok. It doesn't change anything on the algorithm.

We use this in 12.4.3.2 to improve on the performance of the system with random driver allocation. With static allocation the system is absolutely stable, and the daily average cost reduces from €195 to about €152.

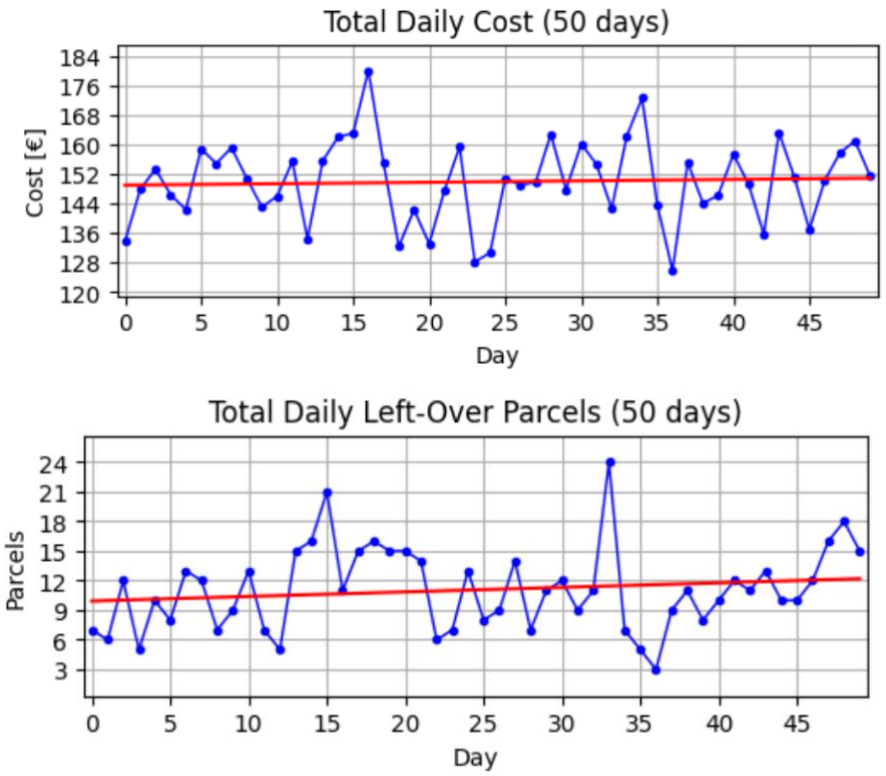
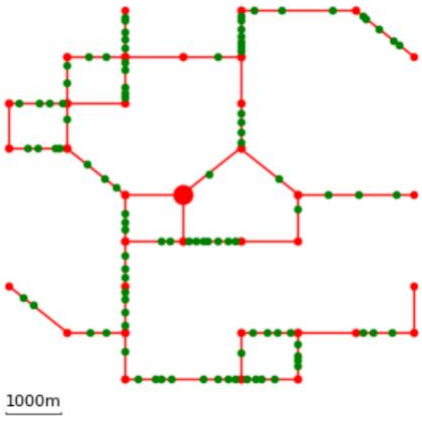


Figure 3 System Performance with Static Driver Allocation

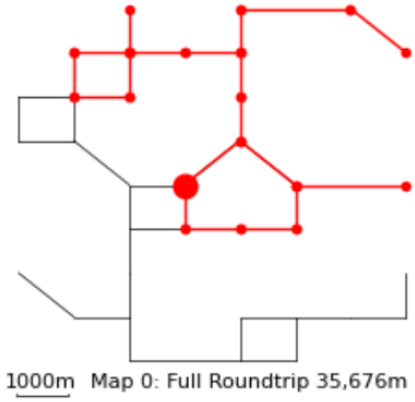
Step 7D Automatic Static Allocation

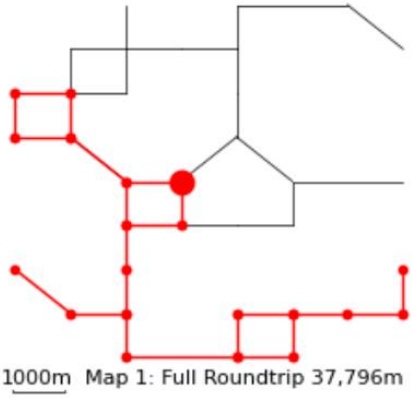
The manual split of a map as applied in section 7C can be quite tricky. We are looking at a way to automatically construct a static allocation. For this purpose we use an IP Solution to the Chinese Postman Problem (Provided in Section 6 “Static Route Assignment”). For a given parameter N this algorithm construct a coverage of all edges of the graph by the routes of N postmen. The algorithm seeks to minimise the total length of the routes of the postmen. Based on this routes we construct regions (minimal subgraphs covering the route). The post office will be in all such regions. Then the customers will be split between the regions. In general there will be a certain overlap between the regions. Customers with location in such overlapping areas are randomly assigned one of the regions.

The default map we have stored in data.pickled:

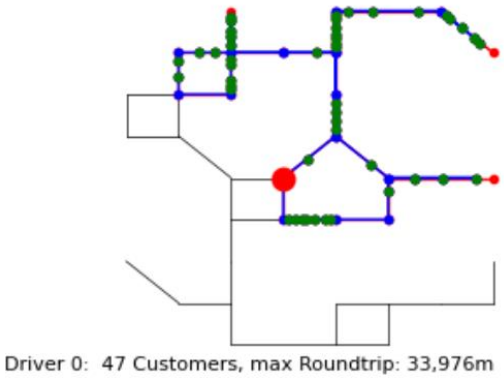


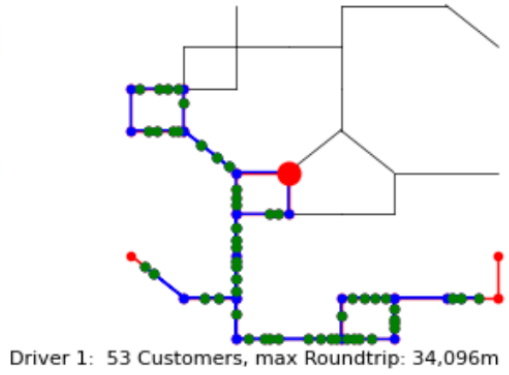
...will be split into two regions, which have only one edge in common:





The customers are split between the regions and the maximal delivery routelength within the regions is calculated:





Step 7D utilises the automatic splitting of the map into similarly sized submaps, but uses otherwise the same greedy algorithm for the determining the delivery route.

It is possible to improve upon the greedy algorithm using the method *shortenLoop*() method in secion 6.1. This will be done in Step 7E, but you can safely ignore this.

Step 7E Route Planning based on CPP

Here we concentrate all the Route Planning in a single step that will be executed before the simulation. The method *planCPP*() is defined in section 6.3:

```
def planCPP(M, W, C, n=1,
            maxLength=None, balance=None,
            timing=False, timeLimit=10, plot=False):
    L = solveCPP(M, W, n=n,
                 maxLength=maxLength, balance=balance,
                 timing=timing, timeLimit=timeLimit)
    if len(L)==0: # unfeasible solution
        return [], [], []

    MM, LL = splitMap(M, L)
    CC = groupCustomersByRegions(C, MM)

    if plot:
        for i in range(len(MM)):
            plotMap(MM[i], w=W, frame=M, scale=True,
                    text=f"Map {i}: Full Roundtrip {int(L[i][2]):,d}m")
            MMT = addTargets(MM[i], CC[i])
            path = shortenLoop(LL[i], MMT, [W] + CC[i])
            plotMap(MMT, T=CC[i], P=path, w=W, frame=M, lwP=1, msPT=3,
                    text=f"Driver {i:d}:  {len(CC[i]):d} Customers, "
                         f"max Roundtrip: {pathLength(path):,d}m")
    return MM, LL, CC
```

The simulation routine is modified in that the Plan generated by the above method is passed into the simulation() method and from there to the Recorder and to the class DeliveryCentre.

```
def simulation(M, W, C, Plan, p=0.2, days=10, seed=0,
               log=False, plot=False, timing=False):

    random.seed(seed)
    D = generateDeliveries(p, len(C), days, seed)

    env = simpy.Environment()
    rec = Recorder(env, M, W, C, D, Plan, days,
                   log=log, plot=plot, timing=timing)

    print(f"Simulating delivery of {sum([len(d) for d in D]):d} parcels "
          f"over {len(D):d} days to {len(C):d} customers "
          f"using {rec.drivers:d} drivers")

    # initialise all customer processes
    # initialises delivery center and creates all parcels
    DC = DeliveryCentre(rec, M, W, C, D, Plan,
                        BIKE_RANGE, DELIVERY_TIME_LIMIT)

    env.run()
    rec.finish()

    if DC.getInventory()>0:
        print(f"Delivery Centre Inventory at the end of last day: "
              f"{DC.getInventory():d} parcels")

    return rec
```

The Simulation run itself is simplified:

First initial planning step may be repeated a few times until successful.

```
Plan = planCPP(M, W, C, n=2,
                        maxLength=40000, balance=0.1, timeLimit=40,
                        timing=True, plot=True)
```

Followed by the simulation step:

```
rec5 = simulation(M, W, C, Plan, p=0.25, days=50)
```

The control parameters for the planning step are:

- n, the number of regions into which the map should be split,
- the maximal Length of a roundtrip in a region (optional)
- the balance, i.e. the maximal percentage deviation of a roundtip from the mean roundtrip length, and
- a timeLimit for the IP Solver.

The default timeLimit is 10sec. When there is no solution, the system responds with “Infeasible” and returns a plan consisting of empty lists. When the solvers runs out of time, but has already a candidate solution, the system responds with “Solution possibly not optimal”. It is then up to the user to decide to run with the results so far or to increase the timeLimit.