



Funded by
the European Union
NextGenerationEU



CZECH
RECOVERY PLAN

MSMT
MINISTRY OF EDUCATION,
YOUTH AND SPORTS

Demonstration of codes for SNF-A

Ivan Eryganov



- The code in Jupyter Notebook plans working day of bikesharing rebalancing operator for Brno.
- It works with a dynamic network flow model and data about rentals.
- The optimization is solved using pyomo language for modelling mathematical programming problems.
- Different solvers can be used.

```

#The main code for article and Green Deal presentations
importing the libraries
import pymo.environ as pyo
import datetime as dt
import pickle
import dateutil as df1
import Article_Aggregated_Solution as aas
import Article_Disaggregated_Solution as ads
import numpy as np

#Main parameters of the algorithm
preferred_demand_interval=4 #hours
optimality_horizon=24 #hours
time_relocations=15 #min
#If the balance nodes should be created
balance_nodes=True
#If the relaxation should be performed
relaxation=True
#Window for relocations
window=5 #min
#In article and exercise we consider only one type of bike, but in the future we can add more types
Bike_type=['standart']
solver='gurobi'
#If the demand information should be extracted and reconstructed
new_timestamp=1
standart_repair_times={'standart':(['Unlocked':10.5,'Check':10.5,'Serr':['10,15'],'Berr':['30,45'])}
solver_time_limit=300
wip_gap=0.0000001

#Basic artificial optimization scenario
warehouse_location=['Skial Brno':[49.2134492,16.6862456]]
vehicles and its capacity
cars=[0:25, 1:25]
vehicles_indices=list(cars.keys())
start and end position of the vehicles
start_position={0:'Skial Brno',1:'Skial Brno'}
end_position={0:len(cars):'Skial Brno',1:len(cars):'Skial Brno'}
#State of the vehicles at the beginning of the optimization
vehicles_state={0:('standart':[0,0,0],[0,0]),1:('standart':[0,0,0],[0,0])}
#List of stations that should not be visited
list_not=[0:['Cormovova - Albert'],1:['CEITEC VUT']]
#List of stations that should be visited by a particular vehicle
list_assigned=[0:['CEITEC VUT'],1:['Cormovova - Albert']]
#List of stations that are not assigned to any vehicle but should be visited
list_unassigned=['Jostova']
currently_rented={'standart':[0,0]}
#Timestamps for the vehicles

```

Figure: Reading the libraries, setting the parameters of calculation

```
File Edit Selection View Go Run Terminal Help
roadngdata (Workspace)
Version, Balance, Released.py:rb
#Basic artificial optimization scenario
roadngdata.py
+ Code + Markdown | Interrupt | Restart | Clear All Outputs | Go To | View data | Jupyter Variables | Outline
venv (3.9.13) (Python 3.9.13)
missing.pkl
missing.pkl
myma_binocular
Modification with the e-bikes.py:rb
new_places.pkl
nextbike_solution.py
nextbike-All rentals_2023_1_3_archiv.csv
nextbike-All rentals_2023_1_3.csv
nextbike-All rentals_2023_4_6_archiv.csv
nextbike-All rentals_2023_4_6.csv
nextbike-All rentals_2023_7_9_archiv.csv
nextbike-All rentals_2023_7_9.csv
nextbike-All rentals_2023_10_12.csv
od_PwLk.py
odjeady_0.pkl
odjeady_1.pkl
odjeady_3.pkl
odjeady_4.pkl
odjeady_5.pkl
odjeady_6.pkl
odjeady_7.pkl
odjeady_8.pkl
odjeady_9.pkl
output.png
pokus.rpz
prjeady_0.pkl
prjeady_1.pkl
prjeady_3.pkl
prjeady_4.pkl
prjeady_5.pkl
prjeady_6.pkl
prjeady_7.pkl
prjeady_8.pkl
prjeady_9.pkl
roadngdata code-workspace
README.md
requirements.txt
station_info.pkl
stations.pkl
static_new_scenario.pkl
Version, Balance, Released.py:rb
VRP_Second Version.py:rb
vyjadky_info.pkl
vyhledakator

#Reading data for the vehicles
work_start=(dtdefi.round_time(dt.datetime(2024, 4, 1, 8,0),time_relocations),1:dtdefi.round_time(dt.datetime(2024, 4, 1, 8,0),time_relocations))
work_end=(dtdefi.round_time(dt.datetime(2024, 4, 1, 16,0),time_relocations),1:dtdefi.round_time(dt.datetime(2024, 4, 1, 16,0),time_relocations))
#Reading data about the stations
with open('stations.pkl', mode='rb') as file:
    stations = pickle.load(file)
with open('station_info.pkl', mode='rb') as file:
    first = pickle.load(file)
with open('stations.pkl', mode='rb') as file:
    stations_locations = pickle.load(file)

#Simple function for scenario generation with only one type of bikes considered
def generate_scenario(number_of_bikes,probability_of_state,probability_of_spawn):
    state_stations={station:[type[0,0,0],[0,0],[0,0]] for type in number_of_bikes.keys()} for station in probability_of_spawn.keys()}
    for type in number_of_bikes.keys():
        elements=list(probability_of_state.keys())
        probabilities=list(probability_of_state.values())
        state=sp.random.choice(elements,number_of_bikes[type], p=probabilities)
        elements=list(probability_of_spawn.keys())
        probabilities=list(probability_of_spawn.values())
        place=sp.random.choice(elements,number_of_bikes[type], p=probabilities)
        for k in range(number_of_bikes[type]):
            station_info=state_stations[place[k]]
            list_bikes=station_info[type]
            #there are only three types of bikes in warehouse: ok, sarr and berr
            if place[k]== 'Skidil Bmo':
                if state[k] in [0,1,2]:
                    list_bikes[0]+=1
                elif state[k] in [3,4]:
                    list_bikes[1][state[k]-3]+=1
                else:
                    list_bikes[4][state[k]-5]+=1
            else:
                if state[k] in [0,1,2]:
                    list_bikes[state[k]]=1
                elif state[k] in [3,4]:
                    list_bikes[3][state[k]-3]+=1
                else:
                    list_bikes[4][state[k]-5]+=1
            station_info[place[k]]=list_bikes
            state_stations[place[k]]=station_info
        return state_stations

#A little bit stochastic state of the network
places=list(stations.keys())
places.append('Skidil Bmo')
probability_of_spawn={place:[w(places) for place in places]
probability_of_state={0:0.7, 1:0.22,2:0.005,3:0.01,4:0.01,5:0.005,6:0.005}
```

Figure: Generation of scenarios to test robustness

```

#A little bit stochastic state of the network
places=list(stations.keys())
places.append('Sklad Brno')
probability_of_spawn={place:1/len(places) for place in places}
probability_of_state={0:0.71, 1:0.25, 2:0.005, 3:0.01, 4:0.01, 5:0.005, 6:0.005}
bikes_total={'standart':500}
state_initial_generate_scenario(bikes_total,probability_of_state,probability_of_spawn)

#Establishing timestamps
import math

from data_flow import Union

#function to calculate timestamps for the vehicles, work, and horizon
def times(stations,new_timestamp,vehicles_indices,work_end,work_start,preferred_demand_interval,optmality_horizont):
    timestamp_list_work=[]
    list_by_vehicle=[]
    timestamp_list=[]
    for i in vehicles_indices:
        hours_not_int=(work_end[i]-work_start[i]).total_seconds()/(3600)
        time_interval=int(hours_not_int,preferred_demand_interval)
        number_of_time_interval_work=math.Floor(hours_not_int/time_interval)+1
        #timestamp for work
        initial_list=[work_start[i] + dt.timedelta(hours=time_interval*x) for x in range(number_of_time_interval_work)]
        initial_list=Union(initial_list,[work_end[i]])
        initial_list.sort()
        list_by_vehicle.update({i:initial_list})
        timestamp_list_work=Union(timestamp_list_work,initial_list)
    timestamp_list_work.sort()
    changed=1
    while changed==1:
        changed=0
        for k in range(len(timestamp_list_work)-1):
            if timestamp_list_work[k+1]-timestamp_list_work[k] > dt.timedelta(hours=preferred_demand_interval):
                additional=(timestamp_list_work[k] + dt.timedelta(hours=preferred_demand_interval*x) for x in range(1,math.Floor((timestamp_list_work[k+1]-timestamp_list_work[k]).total_seconds()/(3600*preferred_demand_interval))))
                timestamp_list_work=Union(additional,timestamp_list_work)
                changed=1
                timestamp_list_work.sort()
                break
    timestamp_list=timestamp_list_work.copy()
    end=timestamp_list[-1]
    additional=[timestamp_list[-1] + dt.timedelta(hours=preferred_demand_interval*x) for x in range(1,math.Floor(optmality_horizont/preferred_demand_interval)+1)]
    timestamp_list+=additional
    timestamp_list=Union(timestamp_list,[end + dt.timedelta(hours=optmality_horizont)])
    timestamp_list.sort()
    if new_timestamp==1:
        def1_data_preparation_new(stations,timestamp_list)
    return list_by_vehicle,timestamp_list,timestamp_list_work
list_by_vehicle,timestamp_list,timestamp_list_work=times(stations,new_timestamp,vehicles_indices,work_end,work_start,preferred_demand_interval,optmality_horizont)

```

Figure: Creation of timestamps and pulling the required data about demand

```

def times_relocations(time_relocations, timestamp_list_work):
    timestamp_relocations=[]
    for k in range(len(timestamp_list_work)-1):
        zacatek=timestamp_list_work[k]
        konec=timestamp_list_work[k+1]
        minutes_not_int=(konec-zacatek).total_seconds//60
        time_interval=min(minutes_not_int,time_relocations)
        number_of_time_interval=work.math.floor(minutes_not_int/time_interval)+1
        initial_list=[zacatek + dt.timedelta(minutes=time_interval*x) for x in range(number_of_time_interval)]
        initial_list=Union(initial_list,[konec])
        initial_list.sort()
        timestamp_relocations=Union(timestamp_relocations,initial_list)
        timestamp_relocations.sort()
        changed=1
        while changed==1:
            changed=0
            for k in range(len(timestamp_relocations)-1):
                if timestamp_relocations[k+1]-timestamp_relocations[k] > dt.timedelta(minutes=time_interval):
                    additional=[timestamp_relocations[k] + dt.timedelta(minutes=time_interval*x) for x in range(1,math.floor((timestamp_relocations[k+1]-timestamp_relocations[k]).total_seconds//60*time_interval)+1)]
                    timestamp_relocations=Union(additional,timestamp_relocations)
                    changed=1
            timestamp_relocations.sort()
            break
    return timestamp_relocations

```

```

if relaxation==True:
    model_p=as_optimize(list_by_vehicle,timestamp_list, timestamp_list_work,solver,solver_time_limit,skip_gap,list_not,state_initial,bikes_total,vehicles_state,Bike_Type,vehicles_indices,warehouse_location,cars,currently_rented,
                        as_optimize_without_interruption(list_by_vehicle,timestamp_list, timestamp_list_work,solver,solver_time_limit,skip_gap,list_not,state_initial,bikes_total,vehicles_state,Bike_Type,vehicles_indices,warehouse_location,cars,cur

```

```

termination_condition: TerminationCondition.optimal
best_feasible_objective: 562.0
best_objective_bound: 562.0
562.0
termination_condition: TerminationCondition.optimal
best_feasible_objective: 122.0
best_objective_bound: 122.0
122.0
termination_condition: TerminationCondition.optimal
best_feasible_objective: 515.0
best_objective_bound: 515.0
515.0
termination_condition: TerminationCondition.optimal
best_feasible_objective: 0.0
best_objective_bound: 0.0

```

```

def times_relocations(time_relocations,timestamp_list_work):
    timestamp_relocations=[]
    for k in range(len(timestamp_list_work)-1):
        zacatek=timestamp_list_work[k]
        konec=timestamp_list_work[k+1]
        minutes_not_int=(konec-zacatek).total_seconds//60
        time_interval=min(minutes_not_int,time_relocations)
        number_of_time_interval=work.math.floor(minutes_not_int/time_interval)+1
        initial_list=[zacatek + dt.timedelta(minutes=time_interval*x) for x in range(number_of_time_interval)]
        initial_list=Union(initial_list,[konec])
        initial_list.sort()
        timestamp_relocations=Union(timestamp_relocations,initial_list)
        timestamp_relocations.sort()
        changed=1
        while changed==1:
            changed=0
            for k in range(len(timestamp_relocations)-1):
                if timestamp_relocations[k+1]-timestamp_relocations[k] > dt.timedelta(minutes=time_interval):
                    additional=[timestamp_relocations[k] + dt.timedelta(minutes=time_interval*x) for x in range(1,math.floor((timestamp_relocations[k+1]-timestamp_relocations[k]).total_seconds//60*time_interval)+1)]
                    timestamp_relocations=Union(additional,timestamp_relocations)
                    changed=1
            timestamp_relocations.sort()
            break
    return timestamp_relocations

```

Figure: Pre-optimization to establish the promising stations to visit, creation of timestamps for relocations

```

#Preparing the data for the main optimization
#Establishing timestamps for the relocations
timestamp_relocations=times_relocations(timestamp_list_work)
#Creating indicators that will remember if the vehicle start or end position is a true station or not
def indicators(start_position,end_position,state_initial):
    indicator_start={}
    indicator_end={}
    for k in start_position.keys():
        if start_position[k] in state_initial.keys():
            indicator_start.update({k:0})
        else:
            indicator_start.update({k:1})
    for k in end_position.keys():
        if end_position[k] in state_initial.keys():
            indicator_end.update({k:0})
        else:
            indicator_end.update({k:1})
    return indicator_start, indicator_end
indicator_start, indicator_end= indicators(start_position,end_position,state_initial)
#Establishing the station that can be visited
stations_with_relocations, stations_only_demand, dict_relocations_from, dict_relocations_to, new_locations=ads.locations_and_matrix(indicator_start,indicator_end, start_position, end_position,state_initial,model_p>window, list_assi

Python

#The main part optimization
New_model=pyo.ConcreteModel()
#Creating sets
ads.create_basic_sets(indicator_start, indicator_end,New_model, Bike_Type, timestamp_relocations, vehicles_indices,warehouse_location, stations_with_relocations, timestamp_list,timestamp_list_work, stations_only_demand)
#Creating parameters
ads.create_parameters(New_model,cars,vehicles_state,currently_rented)
#Creating binary variables
ads.create_binary(New_model)
#Complete description of the demand variables and corresponding balance nodes
ads.create_demand(New_model,model_p)
ads.demand_balance(New_model,timestamp_list)
#Hold variables
ads.hold(New_model,time_relocations)
#Constraints for different type of bikes
ads.Berr(New_model,timestamp_list,timestamp_relocations)
ads.Serr(New_model,timestamp_list_work,timestamp_relocations)
ads.Unlocker(New_model,timestamp_list_work,timestamp_relocations)
ads.Check_and_@((New_model,timestamp_list_work,timestamp_relocations)
#Description of relocation process and changes in number of bikes in vehicle
ads.relocations(New_model,list_assigned,list_unassigned,list_not,model_p,dict_relocations,to)
ads.state_vehicle(New_model,timestamp_relocations)
#Model of line required for relocations and repairs
ads.time_relocations_repair(New_model,timestamp_relocations>window, standart_repair_times)
#travelling constraints
ads.travel(New_model,timestamp_relocations)
#Binary constraints that make the whole model realistic
ads.binary_constraints(list_assigned,list_not,start_position,end_position,indicator_start,indicator_end,New_model,work_start,work_end,cars,timestamp_relocations,list_unassigned)

```

Figure: The main optimization routine

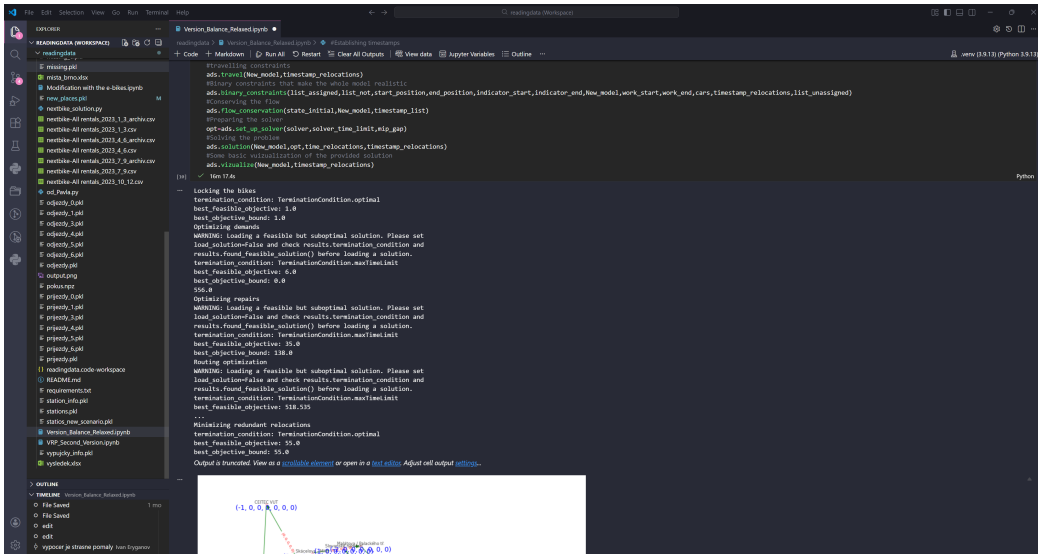


Figure: Log of the optimization

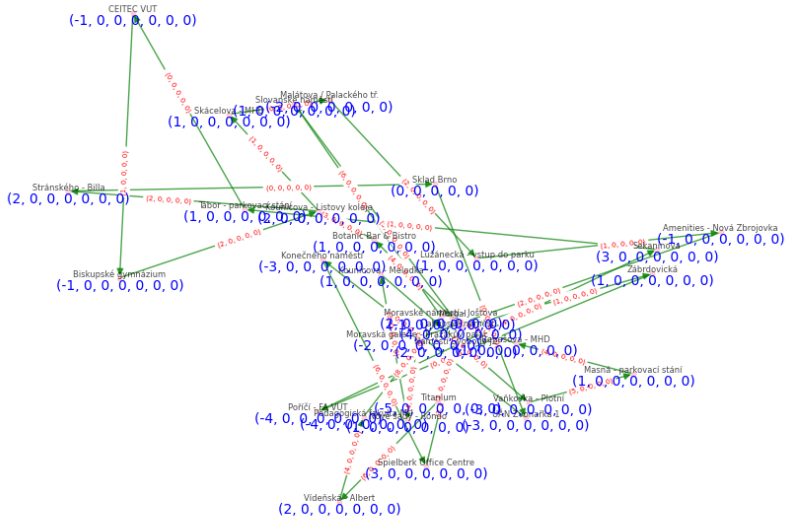


Figure: Basic visualization of operations and route