Algoritmi di Ottimizzazione

Agent Scheduling Problem

Gabriele Felici
Matr: 150400

1 Agent Scheduling Problem

This isn't the classic Traveling Salesman Problem with Time Windows because there are more constraints to respect. Those constraints are:

30 minutes for lunch between 12:00 AM and 2:00 PM;

Limited working time (8 hours);

Limited waiting and traveling time;

Minimum time to spent in office (1 hour);

Importing libraries

[1]:
```python
import gurobipy as gb
from gurobipy import GRB

import math
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
import pandas as pd
```

Define file with clients data

[2]:
```python
FILE = "./TEST_SETS/test_3.txt"
```

Define costant parameters

[3]:
```python
# Default params
SUPPORTED_FORMAT = ['NUM', 'X', 'Y', 'DEMAND', 'READYTIME', 'DUEDATE', 'SERVICE']
# Macros for time values conversions
MINUTES = 60
HOURS = 3600
OFFSET_TIMES = 8*HOURS

COLUMNS_OPS = {'NUM': lambda x: float(x),
               'X': lambda x: float(x),
               'Y': lambda x: float(x),
               'DEMAND': lambda x: 1,
               'READYTIME': lambda x: float(x),
               'DUEDATE': lambda x: float(x),
               'SERVICE': lambda x: float(x)
              }
# Agents count
AGENTS = 7
AGENT_COST = 100000

# Multiplier for distance cost
TIME_PER_DISTANCE = 1
```

```
# Agent Working day start and end
WORKING_TIME_RANGE = (0, 8*HOURS)

# Agent Lunch break time range, lasting
LUNCH_BREAK_RANGE = (12*HOURS-OFFSET_TIMES, 13.5*HOURS-OFFSET_TIMES)
LUNCH_BREAK_TIME = 30*MINUTES

# Agents office parameters
OFFICE_NUM = 0
OFFICE_X = .0
OFFICE_Y = .0
OFFICE_READYTIME = WORKING_TIME_RANGE[0]
OFFICE_DUEDATE = WORKING_TIME_RANGE[1]
OFFICE_SERVICE = 1*HOURS
```

Read clients data

[4]:
```
def read_input_tsptw(filename):
    """ Function used to convert input file to usable data.
        :params filename: File to convert,
        :return: A dict with nodes parameters,
                 A distance matrix between nodes,
                 Nodes coordinates.
    """
    # Dict sed for locations parameters
    data_dict = dict()

    # List of node positions for plots
    nodes_x = list()
    nodes_y = list()

    # Add office to data for matrix distance calculation.
    data_dict.update({OFFICE_NUM: {'X': OFFICE_X,
                                   'Y': OFFICE_Y,
                                   'DEMAND': AGENTS,
                                   'READYTIME': OFFICE_READYTIME,
                                   'DUEDATE': OFFICE_DUEDATE,
                                   'SERVICE': OFFICE_SERVICE,}})

    # Add office to nodes
    nodes_x.append(OFFICE_X)
    nodes_y.append(OFFICE_Y)

    # Open file and read lines
    with open(filename, "r") as file:
        # Initialize columns in empty dict
        columns = file.readline().replace("#","").split()
```

```
        if columns != SUPPORTED_FORMAT:
            print("ERROR! Format not supported.")
            return

        # For each data line
        for line in file.readlines():
            node_dict = {k: COLUMNS_OPS[k](val) for k, val in zip(columns, line.
↪split())}

            # Get id
            node_id = node_dict.pop('NUM')
            # Insert new node in data dict
            data_dict.update({int(node_id): node_dict})
            # Get nodes positions
            nodes_x.append(float(line.split()[columns.index('X')]))
            nodes_y.append(float(line.split()[columns.index('Y')]))

    # Get distance matrix
    distance_matrix = compute_distance_matrix(nodes_x, nodes_y)
    return (data_dict, distance_matrix, dict(enumerate(zip(nodes_x, nodes_y))))


def compute_distance_matrix(nodes_x, nodes_y):
    """ Function used to compute the euclidean distance matrix.
        :param nodes_x: List of nodes x coordinates,
        :param nodes_y: List of nodes y coordinates,
        :return: Distance matrix between nodes."""
    # Get clients count and initialize distance matrix
    clients = len(nodes_x)
    distance_matrix = [[None for i in range(clients)] for j in range(clients)]
    for i in range(clients):
        # Set cost of trip between same agent and himself as null
        distance_matrix[i][i] = 0
        for j in range(clients):
            # Compute distance matrix calculating euclidean distance between␣
↪each node
            dist = compute_dist(nodes_x[i], nodes_x[j], nodes_y[i], nodes_y[j])
            distance_matrix[i][j] = dist
            distance_matrix[j][i] = dist
    return distance_matrix


def compute_dist(xi, xj, yi, yj):
    """ Function used to compute euclidean distance.
        :param xi: x coordinate of first node,
        :param xj: x coordinate of second node,
        :param yi: y coordinate of first node,
```

```
        :param yj: y coordinate of second node,
        :return: Euclidean distance between nodes. """
    exact_dist = math.sqrt(math.pow(xi - xj, 2) + math.pow(yi - yj, 2))
    return int(math.floor(exact_dist + 0.5)) * TIME_PER_DISTANCE
```

[5]:
```
# Getting locations parameters
data_dict, distance_matrix, positions = read_input_tsptw(FILE)
```

[6]:
```
# DEBUG RESTRICTIONS
CLIENTS = len(data_dict)
data_dict = {k: v for k,v in data_dict.items() if k < CLIENTS}
distance_matrix = [dm[:CLIENTS] for dm in distance_matrix[:CLIENTS]]

# ADD FITTICIOUS LOCATION
# This location is used to have a complete loop in Agent trips without␣
 ↪interfering
# with trips costs. Having a complete loop simplify the job of creating a trip.
# To not interfer with costs it's distance to all other locations is 0.
distance_matrix = [dm + [0,] for dm in distance_matrix]
distance_matrix = distance_matrix + [[0]*(CLIENTS+1)]
# Add location data
#remove office from data_dict
del data_dict[0]

# POSITIONS SETS FOR CLEANER MODEL
agent_list = list(range(AGENTS))
#--------------
clients_list = list(range(CLIENTS))
meets_duedates = [(i, data_dict[i]['READYTIME']) for i in data_dict.keys()]
```

1.1 Path cost

We define a function that, given a path defined as a list of actions, gets the cost of the path. Action are the following: 'office','meet','wait','travel', 'lunch' and at each one is assigned a start time (when the action starts during the working day) and a position (where).

The cost is computed on waits and travel actions, that must be minimized.

$$c_s = \begin{cases} 0 & \text{if schedule } s \text{ does not contain waits or travels} \\ \sum_{w \in s} w + H & \text{if work } w \text{ is wait or travel} \end{cases} \tag{1}$$

As the formula shows, an extra high cost H is applied if there is at least one wait or travel: if there is a wait (or travel), the schedule contains also a meet, and the schedule must be assigned to an agent. The extra cost H represent the cost of the agent, which is useful to minimize agents in an ILP model.

[7]:
```python
def path_cost(path):
    """ Function used to compute the cost of an agent schedule.
        :param path: a schedule that contains tuples of actions that an agent␣
 ↪does
        :return: the computed cost
    """
    p = path.copy()
    p.append(('end', WORKING_TIME_RANGE[1], -1)) #useful for the last work␣
 ↪computation
    cost = 0
    for i in range(len(p)-1):
        time_range = p[i+1][1] - p[i][1]
        if time_range < 0:
            raise Exception("Error in the schedule " + str(path) + ": a time is␣
 ↪<0")
        if p[i][0] == 'wait' or p[i][0] == 'travel':
            cost = cost + time_range


    #we assign an extra cost to the schedule if the schedule has meets, which␣
 ↪means that
    #it must be choosen by an agent.
    nodes = [w[2] for w in path if w[0] == 'meet']
    if not nodes:
        return cost
    extra_cost = cost + AGENT_COST
    cost = cost + extra_cost
    return cost
```

1.2 ILP set cover

We define an ILP set cover solver: we provide a set of feasible schedules (each schedule has exactly one lunch and all the action times are in increasing order), and the model chooses some schedules respecting the following constraints: 1. every agents has only one schedule assigned. 2. every meet is in only one schedule among those selected. A binary variable x in the set {0,1} is assigned to each schedule.

$$
\begin{aligned}
\min \sum_{a \in A} \sum_{s_a \in S_A} c_{s_a} x_{s_a} & \\
\text{s.t.} \sum_{s_a \in S_A} x_{s_a} = 1 & \qquad \forall a \in A \\
\sum_{m \in s_a} x_{s_a} = 1 & \qquad \forall m \in M \\
x_{s_a} \in 0,1 & \qquad \forall a \in A, \quad \forall s_a \in S_a
\end{aligned}
$$

[8]:
```python
def set_cover_ILP(schedules):
    """ This function defines an integer programming model that uses a binary␣
 ↪variable
        for each possible schedule.
        :param schedules: the possible schedules
        :return: a gurobi model to optimize
    """
    #Constants
    scheds_list = list(range(len(schedules)))
    meets_list = list(range(len(meets_duedates)))
    positions = [m[0] for m in meets_duedates]

    # Create model
    mod = gb.Model("TSPTW")

    #Vars
    x = mod.addVars({(a,s): 0 for a in agent_list
                          for s in scheds_list },
               name="x",
               vtype=GRB.BINARY)

    #Constrs
    oneschedperagent = mod.addConstrs((gb.quicksum(x[a,s] for s in scheds_list)␣
 ↪== 1 for a in agent_list),
                                    name='one_sched_per_agent')

    meetsconstr = mod.addConstrs((gb.quicksum(x[a,s]
                                            for a in agent_list
                                            for s in scheds_list
                                            if pos in [work[2] for work in␣
 ↪schedules[s] if work[0] == 'meet'])
                                            == 1 for pos in positions),␣
 ↪name='meets_constr')

    #Obj
    mod.setObjective((gb.quicksum(x[a,s] * path_cost(schedules[s])
                                for a in agent_list
                                for s in scheds_list)), GRB.MINIMIZE)

    return mod
```

[9]:
```python
def schedules_from_mod(mod, scheds):
    """ Gets the agent schedules from an optimized model
        :param mod: the optimized gurobi model
        :return: dict that associate each agent to a schedule
    """
    l = []
```

```
    for v in mod.getVars():
        if v.x == 1:
            s = str(v.varName)
            s = s[1:len(s)]
            split = s.split(',')#
            cmd = 'l.append(scheds['+split[1]+')'
            exec(cmd)
    scheds = {i: l[i] for i in range(len(l))}
    return scheds
```

1.3 Useful functions for the model

Sort nodes Given a list of meets (as integers) sorts it using the start time of each meet.

Swap Given a combination of meets (as integers) for each agent, returns a list of possible new (and not necessarily feasible) combinations of meets for each agent, each obtained swapping meets in all the agent lists in the first combination.

Path to schedule Given a list of meets (as integers) returns a schedule associated to it, taking care of office work and lunch. The cost of this schedule is given by waits and travels, and can be computed calling the function "path_cost".

Is sched feasible Given a schedule returns true if the schedule is feasible, false otherwise. A schedule is feasible for the ILP model if it contains exactly one lunch action and each action starts after the previous.

All sched comb Calls the swap function to get new combination of nodes, and for each combination generates a schedule calling "path to schedule". Each generated schedule is inserted in a list if it's feasible. The list of schedules will be passed to the ILP set cover model.

Generate new schedules Generates more schedules. Starting from all the nodes of all agents in sequence, generates schedules splitting this sequence in two part for each index {i=1, i=2 .. i = n}

Optimize time windows Optimizes a schedule moving time windows. The goal is to do more work in office and reduce wait times. Starting from the wait meet, it is reduced moving the previous meet date next as long as his time window permits. The time moved is the minimum between the window range time and the wait time. This is done for the previous wait too until all meet are possibly moved.

[10]:
```
def sort_nodes(nodes):
    """ Sorts nodes in increasing order using start time of each meet.
    """
    tosort = [(n, data_dict[n]['READYTIME']) for n in nodes]
    _sorted = sorted(tosort, key=lambda node: node[1])
    sorted_nodes = [n[0] for n in _sorted]
    return sorted_nodes
```

[11]:
```
def swap(agents_paths, swap3=False):
    """ This function generates lists of paths for agents, swapping nodes in a␣
 ↪first assignment of paths to agents.
        An agent path is a list of integers that represents the meets sequence␣
 ↪that an agent must follow.
        For each generated path is also generated a variant of length +1 and -1.
        :param agents_paths: a dict that associate each agent to a path
        :return: more possible agent_paths (not feasible too, must be filtered)
    """
    lens = { i : len(agents_paths[i]) for i in range(len(agents_paths))}
    #init lens_reduced
    lens_reduced = {}
    counter = 0
    accumulate = True
    for i in agents_paths.keys():
        dim = len(agents_paths[i])
        if dim > 0:
            lens_reduced[i] = dim-1
            counter = counter+1
        elif accumulate:
            lens_reduced[i] = counter
            accumulate = False
        else:
            lens_reduced[i] = 0

    #init lens_increased
    lens_increased = {}
    budget = sum(len(agents_paths[i]) for i in agents_paths.keys())
    end = False
    for i in agents_paths.keys():
        dim = len(agents_paths[i])
        if budget >= dim+1:
            lens_increased[i] = dim+1
            budget = budget-dim-1
        elif not end:
            lens_increased[i] = budget
            budget = 0
            end = True
        else:
            lens_increased[i] = 0

    nodes = []
    all_lens = [lens, lens_reduced, lens_increased]

    for i in range(len(agents_paths)):
        nodes = nodes + agents_paths[i]
    all_paths = []
```

```python
    #swap
    for i in range(len(nodes)-1):
        for j in range(i+1, len(nodes)):
            new_nodes = nodes.copy()
            (new_nodes[i], new_nodes[j]) = (new_nodes[j], new_nodes[i])
            counter = 0
            for l in all_lens:
                for k in range(len(agents_paths)):
                    new_path = []
                    new_path = new_path + new_nodes[counter:counter+l[k]]
                    counter = counter + l[k]
                    agents_paths[k] = new_path
                all_paths.append(agents_paths.copy())
                counter = 0

    #swap 3
    if swap3:
        for i in range(len(nodes)-2):
            for j in range(i+1, len(nodes)-1):
                for h in range(j+1, len(nodes)):
                    new_nodes = nodes.copy()
                    (new_nodes[i], new_nodes[j], new_nodes[h]) = (new_nodes[h],␣
↪new_nodes[i], new_nodes[j])

                    counter = 0
                    for l in all_lens:
                        for k in range(len(agents_paths)):
                            new_path = []
                            new_path = new_path + new_nodes[counter:counter+l[k]]
                            counter = counter + l[k]
                            agents_paths[k] = new_path
                        all_paths.append(agents_paths.copy())
    return all_paths


def path_to_schedule(path):
    """ Gets a schedule starting from a path (a list of meet nodes associated to␣
↪an agent).
        Between each node is valued if there an agent has enough time to come␣
↪back in office,
        and possibly the lunch.
        :param path: an agent path expressed as a list of meets.
    """
    sched = []
    if not path: #no nodes => agent stays in office
            sched.append(('office', 0.0, 0))
```

```
            sched.append(('lunch', LUNCH_BREAK_RANGE[0], -1))
            sched.append(('office', LUNCH_BREAK_RANGE[0]+LUNCH_BREAK_TIME, 0))
    else:
        lunch_done = False
        lunch_shift = False
        meet_hour = data_dict[path[0]]['READYTIME']
        meet_end = meet_hour + data_dict[path[0]]['SERVICE']
        if OFFICE_SERVICE + distance_matrix[0][path[0]]*TIME_PER_DISTANCE <= 
meet_hour:
            start_travel =  meet_hour - 
distance_matrix[0][path[0]]*TIME_PER_DISTANCE
            sched.append(('office', 0.0, 0))
            if meet_hour >= LUNCH_BREAK_RANGE[0] + LUNCH_BREAK_TIME and 
meet_hour <= LUNCH_BREAK_RANGE[1] + LUNCH_BREAK_TIME and not lunch_done:
                sched.append(('travel', start_travel - LUNCH_BREAK_TIME, -1))
                sched.append(('lunch', meet_hour - LUNCH_BREAK_TIME, -1))
                lunch_done = True
            else:
                sched.append(('travel', start_travel, -1))
            sched.append(('meet', meet_hour, path[0]))
        else:
            sched.append(('wait', 0.0, -1))
            sched.append(('meet', meet_hour, path[0]))
        if meet_end >= LUNCH_BREAK_RANGE[0] and meet_end <= LUNCH_BREAK_RANGE[1] 
and not lunch_done:
            sched.append(('lunch', meet_end, -1)) # in this case next meet will 
start an half hour later
            lunch_done = True

        #iter 0 to n-1 taking i and i+1 node to add works between two meets.
        for j in range(len(path)-1):
            prev = path[j]
            _next = path[j+1]
            shift = 0.0
            if lunch_shift:
                shift = LUNCH_BREAK_TIME
                lunch_shift = False
            end_prev_meet = data_dict[prev]['READYTIME'] + 
data_dict[prev]['SERVICE'] + shift
            start_next_meet = data_dict[_next]['READYTIME']
            end_next_meet = start_next_meet + data_dict[_next]['SERVICE']
            diff = start_next_meet - end_prev_meet

            if ((start_next_meet >= LUNCH_BREAK_RANGE[0] + LUNCH_BREAK_TIME #è 
ora di pranzo
```

```
                and start_next_meet <= LUNCH_BREAK_RANGE[1] + LUNCH_BREAK_TIME)␣
↪or
                (end_next_meet >= LUNCH_BREAK_RANGE[0] and end_next_meet <=␣
↪LUNCH_BREAK_RANGE[1])) and not lunch_done:
                #Check if lunch must be done before the meet
                if start_next_meet >= LUNCH_BREAK_RANGE[0] + LUNCH_BREAK_TIME␣
↪and start_next_meet <= LUNCH_BREAK_RANGE[1] + LUNCH_BREAK_TIME and not␣
↪lunch_done: #check prima del meet
                    #if there is enough time to come back in office
                    if OFFICE_SERVICE +␣
↪distance_matrix[0][prev]*TIME_PER_DISTANCE +␣
↪distance_matrix[0][_next]*TIME_PER_DISTANCE + LUNCH_BREAK_TIME <= diff:
                        in_office = end_prev_meet +␣
↪distance_matrix[0][prev]*TIME_PER_DISTANCE
                        start_travel_to_meet = start_next_meet -␣
↪distance_matrix[0][_next]*TIME_PER_DISTANCE - LUNCH_BREAK_TIME #pranzo␣
↪all'arrivo
                        sched.append(('travel', end_prev_meet, -1))
                        sched.append(('office', in_office, 0))
                        sched.append(('travel', start_travel_to_meet, -1))
                        sched.append(('lunch', start_next_meet -␣
↪LUNCH_BREAK_TIME, -1))
                        sched.append(('meet', start_next_meet, _next))
                        lunch_done = True
                    else:
                        #per l'inizializzazione si è tenuto conto che wait sia␣
↪maggiore di LUNCH_BREAK_TIME (SOLO INIT)
                        travel_arrive = end_prev_meet +␣
↪distance_matrix[prev][_next]*TIME_PER_DISTANCE
                        lunch_diff = start_next_meet - travel_arrive
                        if lunch_diff >= LUNCH_BREAK_TIME:
                            sched.append(('travel', end_prev_meet, -1))
                            sched.append(('lunch', travel_arrive, -1))
                            sched.append(('wait', travel_arrive +␣
↪LUNCH_BREAK_TIME, -1))
                            sched.append(('meet', start_next_meet, _next))
                            lunch_done = True
                        else: #it's lunch time but there's no time to come back␣
↪in office
                            sched.append(('travel', end_prev_meet, -1))
                            sched.append(('wait', travel_arrive, -1))
                            sched.append(('meet', start_next_meet, _next))
                #Check if lunch can be done after the meet
                if end_next_meet >= LUNCH_BREAK_RANGE[0] and end_next_meet <=␣
↪LUNCH_BREAK_RANGE[1] and not lunch_done: #end meet check
```

```
                    if OFFICE_SERVICE +␣
↪distance_matrix[0][prev]*TIME_PER_DISTANCE +␣
↪distance_matrix[0][_next]*TIME_PER_DISTANCE <= diff:
                        in_office = end_prev_meet +␣
↪distance_matrix[0][prev]*TIME_PER_DISTANCE
                        start_travel_to_meet = start_next_meet -␣
↪distance_matrix[0][_next]*TIME_PER_DISTANCE
                        sched.append(('travel', end_prev_meet, -1))
                        sched.append(('office', in_office, 0))
                        sched.append(('travel', start_travel_to_meet, -1))
                        sched.append(('meet', start_next_meet, _next))
                        sched.append(('lunch', end_next_meet, -1)) #prossima␣
↪iterazione end_prev_meet swifta
                        lunch_done = True
                        lunch_shift = True
                    else:#no time to come back in office
                        travel_arrive = end_prev_meet +␣
↪distance_matrix[prev][_next]*TIME_PER_DISTANCE
                        if _next == path[-1]: #if _next+1 == len(path) corner␣
↪case
                            sched.append(('travel', end_prev_meet, -1))
                            sched.append(('wait', travel_arrive, -1))
                            sched.append(('meet', start_next_meet, _next))
                            sched.append(('lunch', end_next_meet, -1))
                            lunch_shift = True
                            lunch_done = True
                        else:
                            lunch_diff = data_dict[path[j+2]]['READYTIME'] -␣
↪end_next_meet
                            if lunch_diff >= LUNCH_BREAK_TIME:#check diff
                                sched.append(('travel', end_prev_meet, -1))
                                sched.append(('wait', travel_arrive, -1))
                                sched.append(('meet', start_next_meet, _next))
                                sched.append(('lunch', end_next_meet, -1))
                                lunch_shift = True
                                lunch_done = True
                            else: #it's lunch time there's no time
                                sched.append(('travel', end_prev_meet, -1))
                                sched.append(('wait', travel_arrive, -1))
                                sched.append(('meet', start_next_meet, _next))

            else:#it's not lunch time
                if OFFICE_SERVICE + distance_matrix[0][prev]*TIME_PER_DISTANCE +␣
↪distance_matrix[0][_next]*TIME_PER_DISTANCE <= diff: #enough time to come back␣
↪in office
```

```
                    in_office = end_prev_meet + distance_matrix[0][prev]*TIME_PER_DISTANCE
                    start_travel_to_meet = start_next_meet - distance_matrix[0][_next]*TIME_PER_DISTANCE
                    sched.append(('travel', end_prev_meet, -1))
                    sched.append(('office', in_office, 0))
                    sched.append(('travel', start_travel_to_meet, -1))
                    sched.append(('meet', start_next_meet, _next))
                else:
                    #travel -> wait -> meet
                    waiting = end_prev_meet + distance_matrix[prev][_next]*TIME_PER_DISTANCE
                    sched.append(('travel', end_prev_meet, -1))
                    sched.append(('wait', waiting, -1))
                    sched.append(('meet', start_next_meet, _next))
        #end
        last = path[len(path)-1]
        shift = 0.0
        if lunch_shift:
            shift = LUNCH_BREAK_TIME
            lunch_shift = False
        last_meet_end = data_dict[last]['READYTIME'] + data_dict[last]['SERVICE'] + shift
        office_arriving = last_meet_end + distance_matrix[0][last]*TIME_PER_DISTANCE
        if WORKING_TIME_RANGE[1] - office_arriving >= OFFICE_SERVICE:
            sched.append(('travel', last_meet_end, -1))
            sched.append(('office', office_arriving, 0))
        else:
            sched.append(('wait', last_meet_end, -1)) #end of the day

    #check lunch num
    lunch_num = len([m for m in sched if m[0] == 'lunch'])
    if lunch_num > 1:
        raise Exception('Error, more than one lunch in the schedule: ' + str(path) + ' --- ' + str(sched))
    elif lunch_num == 0:#if lunch is not in the schedule, must try to get it during office time
        sched.append(('end', WORKING_TIME_RANGE[1], -1))
        for i in range(len(sched)-1):
            r1=sched[i][1]
            r2=sched[i+1][1]
            if sched[i][0] == 'office':
                if (r1 <= LUNCH_BREAK_RANGE[0] and r2 >= LUNCH_BREAK_RANGE[0] + LUNCH_BREAK_TIME):
                    sched.insert(i+1, ('lunch', LUNCH_BREAK_RANGE[0], -1))
```

```
                    sched.insert(i+2, ('office', LUNCH_BREAK_RANGE[0] +␣
↪LUNCH_BREAK_TIME, 0))
                    break
                elif r1 >= LUNCH_BREAK_RANGE[0] and r2 <= LUNCH_BREAK_RANGE[1]␣
↪and r2-r1 >= LUNCH_BREAK_TIME:
                    sched.insert(i+1, ('lunch', r1, -1))
                    sched.insert(i+2, ('office', r1 + LUNCH_BREAK_TIME, -1))
                    del(a[i])
                    break
                elif r1 <= LUNCH_BREAK_RANGE[1] and r2 >= LUNCH_BREAK_RANGE[1] +␣
↪LUNCH_BREAK_TIME:
                    sched.insert(i+1, ('lunch', LUNCH_BREAK_RANGE[1], -1))
                    sched.insert(i+2, ('office', LUNCH_BREAK_RANGE[1] +␣
↪LUNCH_BREAK_TIME, 0))
                    break
        del(sched[-1])#remove end
    return sched

def paths_to_schedule():
    """ Transforms agent path into schedules.
        :return: a dict agent-schdule
    """
    scheds = {a: [] for a in agent_list}
    for i in agent_list:
        scheds[i].append(path_to_schedule(agents_paths[i]))
    return scheds

def is_sched_feasible(schedule):
    """ Checks if a scheudule is feasible, looking if each value is more than␣
↪the previous.
        The schedule must also contain a lunch activity.
        :return: True if the schedule is feasible, False otherwise.
    """
    lunch_done = False
    time = 0.0
    for work in schedule:
        if work[0] == 'lunch':
            lunch_done = True
        if time > work[1]:
            return False
        time = work[1]

    return lunch_done

def all_schedule_comb(agents_paths):
    """ Makes the nodes swap and gets new schedules from the combinations.
        :param agents_paths: agent-path dict
```

```
    """
    scheds = paths_to_schedule()
    all_combinations = swap(agents_paths, True)
    for comb in all_combinations: #all_combination is a list of dicts
        for key in comb.keys():
            nodes = sort_nodes(comb[key])
            sched = path_to_schedule(nodes)
            if is_sched_feasible(sched) and sched not in scheds[key]:
                scheds[key].append(sched)

    #convert list of dict in a list of schedules
    _scheds = []
    for key in scheds.keys():
        for s in scheds[key]:
            if is_sched_feasible(s) and s not in _scheds:
                _scheds.append(s)
    return _scheds
```

[12]:
```
def generate_node_paths(agent_paths, swap3=False):
    """
    Generates more schedules. Starting from all the nodes of all agents in␣
 ↪sequence,
    generates schedules splitting this sequence in two part for each index {i=1,␣
 ↪i=2 .. i = n}
    """
    nodes = []
    for i in range(len(agents_paths)):
        nodes = nodes + agents_paths[i]
    node_paths = []

    #swap
    for i in range(len(nodes)-1):
        for j in range(i+1, len(nodes)):
            new_nodes = nodes.copy()
            (new_nodes[i], new_nodes[j]) = (new_nodes[j], new_nodes[i])
            for k in range(1,len(nodes)-1):
                new_path = new_nodes[0:k].copy()
                if new_path not in node_paths:
                    node_paths.append(new_path)

    #swap 3
    if swap3:
        for i in range(len(nodes)-2):
            for j in range(i+1, len(nodes)-1):
                for h in range(j+1, len(nodes)):
                    new_nodes = nodes.copy()
```

```python
                    (new_nodes[i], new_nodes[j], new_nodes[h]) = (new_nodes[h],␣
↪new_nodes[i], new_nodes[j])
                    for k in range(1,len(nodes)-1):
                        new_path = new_nodes[0:k].copy()
                        if new_path not in node_paths:
                            node_paths.append(new_path)
    return node_paths

def generate_new_schedules(node_paths):
    scheds = []
    for node_path in node_paths:
        scheds.append(path_to_schedule(sort_nodes(node_path)))
    return scheds
```

[13]:

```python
def optimize_time_windows(sched):
    """ Optimizes a schedule moving time windows.
        The goal is to do more work in office and reduce wait times .
        :param sched: the schedule to optimize.
    """
    sched.append(('end', WORKING_TIME_RANGE[1], -1)) #useful for compute last␣
↪wait range
    waits = [w for w in sched if w[0] == 'wait']
    waits.reverse()

    #save office indexes in the schedule
    indexes = [0]
    for i, s in enumerate(sched):
        if s[0] == 'office':
            indexes.append(i)
            prev_work = s[0]

    if len(indexes) in [0,1] and not sched[-2][0] == 'wait' and not sched[0][0]␣
↪== 'wait': #no slices to optimize
        return sched

    #must optimize a series of slices. A slice contains one or more wait actions.
    slices = []
    for i in range(len(indexes)-1):
        slices.append(sched[indexes[i]:indexes[i+1]+1])
    slices.append(sched[indexes[-1]:len(sched)])

    #optimize office-office slice or first/last slice if it contains 'wait'
    for s in slices:
        waits = [w for w in s if w[0] == 'wait']
        waits.reverse()
        for w in waits: #waits are in reverse order(reverse()). The optimization␣
↪is done staring from last 'wait'
```

```
            for i, work in enumerate(s):
                if w == work:
                    #searching the index before this wait
                    j = i
                    while not s[j][0] == 'meet':
                        if j == 0:
                            break
                        j = j-1

                    _min = 0
                    if s[j][0] == 'meet':
                        wait_time = s[i+1][1] - s[i][1]
                        meet_time = data_dict[s[j][2]]['DUEDATE'] -␣
↪data_dict[s[j][2]]['READYTIME']
                        if 'lunch' in [w[0] for w in s[j:i+1]]:#lunch is between␣
↪meet and wait
                            lunch_index = -1 #get lunch index
                            for l, w in enumerate(s):
                                if w[0] == 'lunch':
                                    lunch_index = l
                            lunch_time = LUNCH_BREAK_RANGE[1] - s[lunch_index][1]
                            _min = min(wait_time, meet_time, lunch_time)
                        else:
                            _min = min(wait_time, meet_time)
                    elif s[j][0] == 'office':#no meet, office time is extended␣
↪with wait_time
                        if 'lunch' in [w[0] for w in s[j:i+1]]:# if lunch is␣
↪between j and i
                            lunch_index = -1 #get lunch index
                            for l, w in enumerate(s):
                                if w[0] == 'lunch':
                                    lunch_index = l
                            lunch_time = LUNCH_BREAK_RANGE[1] - s[lunch_index][1]
                            _min = min(wait_time, lunch_time)
                        else:
                            wait_time = s[i+1][1] - s[i][1]
                            _min = wait_time
                        j = j+1 #the first is office, there's no meet moving but␣
↪the next work will start later
                    for k in range(j,i+1):
                        newval = list(s[k])
                        newval[1] = newval[1] + _min
                        s[k] = tuple(newval)
                    break

    #rebuild schedule from slices
    newsched = []
```

```
    for s in slices:
        newsched = newsched + s[0:len(s)-1]

    #if the optimized schedule starts with 'wait' must check that there is␣
↪enough time for office work
    if newsched[0][0] == 'wait':
        diff = newsched[1][1] - newsched[0][1]
        if diff >= OFFICE_SERVICE +␣
↪distance_matrix[0][sched[1][2]]*TIME_PER_DISTANCE:
            start_travel = newsched[1][1] -␣
↪distance_matrix[0][newsched[1][2]]*TIME_PER_DISTANCE
            newsched.pop(0)
            newsched.insert(0, ('travel', start_travel, -1))
            newsched.insert(0, ('office', 0.0, 0))

    if not is_sched_feasible(newsched):
        return None

    return newsched
```

2 Initialization

Must be provided an initial solution to give for the first iteration. We use a greedy algorithm that for each agent he tries to attempt as many meets as possible choosing to all still available meets.

[14]:
```
import datetime
exec_start_time = datetime.datetime.now()

'''
Greedy init: each agent takes all the meets he can do.
'''
agents_paths = {a:[] for a in agent_list}
uncovered_meets = meets_duedates.copy()
uncovered_meets.sort(key=lambda x: x[1]) #meets are sorted in increasing order␣
↪on the start time
for i in range(len(agents_paths)):
    lunch_done = False
    if uncovered_meets:
        next_meet = uncovered_meets.pop(0) #the agent has not already served any␣
↪client, so takes the first meet
        freeat = next_meet[1] + data_dict[next_meet[0]]['SERVICE']
        if next_meet[1] <= LUNCH_BREAK_RANGE[0] and freeat >=␣
↪LUNCH_BREAK_RANGE[1]:
            raise Exception("Error, a meet takes all the lunch time range.")
        agents_paths[i].append(next_meet[0])
```

```
        if next_meet[1] >= LUNCH_BREAK_RANGE[0] + LUNCH_BREAK_TIME and␣
↪next_meet[1] <= LUNCH_BREAK_RANGE[1] + LUNCH_BREAK_TIME:#lunch already done
            lunch_done = True
        elif freeat >= LUNCH_BREAK_RANGE[0] and freeat <= LUNCH_BREAK_RANGE[1]:␣
↪#it's lunch time, freeat is moved an half hour later
            freeat = freeat + LUNCH_BREAK_TIME
            lunch_done = True

        to_remove = [] #traces meets to remove from uncovered_meets
        for m in uncovered_meets: #agent takes the next client he can serve
            if m[1] <= LUNCH_BREAK_RANGE[0] and m[1] +␣
↪data_dict[m[0]]['SERVICE'] >= LUNCH_BREAK_RANGE[1]:
                raise Exception("Error, a meet takes all the lunch time range.")

            if freeat +␣
↪distance_matrix[agents_paths[i][-1]][m[0]]*TIME_PER_DISTANCE <= m[1]:
                freeat = m[1] + data_dict[m[0]]['SERVICE']
                agents_paths[i].append(m[0])
                to_remove.append(m)
                if m[1] >= LUNCH_BREAK_RANGE[0] + LUNCH_BREAK_TIME and m[1] <=␣
↪LUNCH_BREAK_RANGE[1] + LUNCH_BREAK_TIME and not lunch_done:
                    lunch_done = True
                if freeat >= LUNCH_BREAK_RANGE[0] and freeat <=␣
↪LUNCH_BREAK_RANGE[1] and not lunch_done:
                    freeat = freeat + LUNCH_BREAK_TIME
                    lunch_done = True

        for m in to_remove:
            uncovered_meets.remove(m)
```

3 Model optimization

We iterate between ILP set cover solver and schedules generation until the cost convergence. Given a solution all paths nodes for all agents are swapped and feasible schedules are generated. The ILP model chooses a subset of these schedules and starting from this provided solution generates new schedules for the next step.

[15]:
```
#initial cost
last_cost = 0
scheds = paths_to_schedule()#dict that associate each agent to a schedule
for key in scheds.keys():#optimize time windows in each agent schedule
    s1 = optimize_time_windows(scheds[key][0])
    scheds[key][0] = s1
for key in scheds.keys():
    last_cost = last_cost + path_cost(scheds[key][0])
```

```
print("Initial cost: " + str(last_cost))
print("\n-----------------------------------\n")

while (True):
    _scheds = [scheds[key][0] for key in scheds.keys()]
    _scheds = all_schedule_comb(agents_paths)#returns a list of schedules
    node_paths = generate_node_paths(agents_paths, True)
    _scheds = _scheds + generate_new_schedules(node_paths)

    for s in range(len(_scheds)):
        _scheds[s] = optimize_time_windows(_scheds[s])

    _scheds = [s for s in _scheds if not s == None]#remove all None (a None means a schedule is unfeasible)
    #a lot of equals schedules without meet can be choosen, so these must be in the schedules set before the optimization
    for c in clients_list:
        _scheds.append([('office', 0.0, 0), ('lunch', 14400, -1), ('office', 16200, 0)])

    mod = set_cover_ILP(_scheds)
    mod.optimize()

    scheds = schedules_from_mod(mod, _scheds)#now scheds is a dict of schedules (a sched for each agent)
    agents_paths = {i: [w[2] for w in scheds[i] if w[0]=='meet'] for i in range(len(scheds))}#update agents_paths
    #Compute current cost
    cost = 0
    for key in scheds.keys():
        cost = cost + path_cost(scheds[key])
    print("\nPrevious cost: " + str(last_cost) + ", current cost: " + str(cost)+"\n")
    print("---------------------------------------------\n")
    if (last_cost <= cost):
        break
    else:
        last_cost = cost

#get execution time
exec_end_time = datetime.datetime.now()
delta_exec_time = exec_end_time - exec_start_time
exec_time = delta_exec_time.total_seconds()
print("Execution time: " + str(exec_time))
```

```
Initial cost: 598040.0
```

```
-----------------------------------

Using license file C:\Users\gabriele\gurobi.lic
Academic license - for non-commercial use only
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 34 rows, 2156 columns and 10934 nonzeros
Model fingerprint: 0xbae0e833
Variable types: 0 continuous, 2156 integer (2156 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+05, 1e+05]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 0 rows and 203 columns
Presolve time: 0.06s
Presolved: 34 rows, 1953 columns, 10731 nonzeros
Variable types: 0 continuous, 1953 integer (1953 binary)

Root relaxation: objective 5.874240e+05, 135 iterations, 0.02 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 587424.000    0    8          - 587424.000      -     -    0s
H    0     0                    587424.00000 587424.000  0.00%     -    0s
     0     0 587424.000    0    8 587424.000 587424.000  0.00%     -    0s

Explored 1 nodes (135 simplex iterations) in 0.18 seconds
Thread count was 8 (of 8 available processors)

Solution count 1: 587424

Optimal solution found (tolerance 1.00e-04)
Best objective 5.874240000000e+05, best bound 5.874240000000e+05, gap 0.0000%

Previous cost: 598040.0, current cost: 587424.0

---------------------------------------------

Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 34 rows, 2723 columns and 13006 nonzeros
Model fingerprint: 0xd36acb4d
Variable types: 0 continuous, 2723 integer (2723 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+05, 1e+05]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
```

```
Presolve removed 0 rows and 203 columns
Presolve time: 0.04s
Presolved: 34 rows, 2520 columns, 12803 nonzeros
Variable types: 0 continuous, 2520 integer (2520 binary)

Root relaxation: objective 5.813920e+05, 111 iterations, 0.01 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

*    0     0               0    581392.00000 581392.000  0.00%     -    0s

Explored 0 nodes (111 simplex iterations) in 0.11 seconds
Thread count was 8 (of 8 available processors)

Solution count 1: 581392

Optimal solution found (tolerance 1.00e-04)
Best objective 5.813920000000e+05, best bound 5.813920000000e+05, gap 0.0000%

Previous cost: 587424.0, current cost: 581392.0

---------------------------------------------

Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 34 rows, 3430 columns and 16471 nonzeros
Model fingerprint: 0x32cd31d1
Variable types: 0 continuous, 3430 integer (3430 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+05, 1e+05]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 0 rows and 203 columns
Presolve time: 0.04s
Presolved: 34 rows, 3227 columns, 16268 nonzeros
Variable types: 0 continuous, 3227 integer (3227 binary)

Root relaxation: objective 5.790320e+05, 85 iterations, 0.01 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

*    0     0               0    579032.00000 579032.000  0.00%     -    0s

Explored 0 nodes (85 simplex iterations) in 0.15 seconds
Thread count was 8 (of 8 available processors)
```

```
Solution count 1: 579032

Optimal solution found (tolerance 1.00e-04)
Best objective 5.790320000000e+05, best bound 5.790320000000e+05, gap 0.0000%

Previous cost: 581392.0, current cost: 579032.0

---------------------------------------------

Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 34 rows, 3045 columns and 15078 nonzeros
Model fingerprint: 0x9fb298be
Variable types: 0 continuous, 3045 integer (3045 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+05, 1e+05]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 0 rows and 203 columns
Presolve time: 0.06s
Presolved: 34 rows, 2842 columns, 14875 nonzeros
Variable types: 0 continuous, 2842 integer (2842 binary)

Root relaxation: objective 5.785320e+05, 84 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

*    0     0               0    578532.00000 578532.000  0.00%     -    0s

Explored 0 nodes (84 simplex iterations) in 0.13 seconds
Thread count was 8 (of 8 available processors)

Solution count 1: 578532

Optimal solution found (tolerance 1.00e-04)
Best objective 5.785320000000e+05, best bound 5.785320000000e+05, gap 0.0000%

Previous cost: 579032.0, current cost: 578532.0

---------------------------------------------

Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 34 rows, 2401 columns and 12460 nonzeros
Model fingerprint: 0x61cc4ce6
Variable types: 0 continuous, 2401 integer (2401 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
```

```
  Objective range  [1e+05, 1e+05]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 0 rows and 203 columns
Presolve time: 0.04s
Presolved: 34 rows, 2198 columns, 12257 nonzeros
Variable types: 0 continuous, 2198 integer (2198 binary)

Root relaxation: objective 5.785320e+05, 91 iterations, 0.01 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 578532.000    0    6          - 578532.000      -     -    0s
H    0     0                    578532.00000 578532.000  0.00%     -    0s
     0     0 578532.000    0    6 578532.000 578532.000  0.00%     -    0s

Explored 1 nodes (91 simplex iterations) in 0.14 seconds
Thread count was 8 (of 8 available processors)

Solution count 1: 578532

Optimal solution found (tolerance 1.00e-04)
Best objective 5.785320000000e+05, best bound 5.785320000000e+05, gap 0.0000%

Previous cost: 578532.0, current cost: 578532.0

---------------------------------------------

Execution time: 8.323599
```

4 Print results

[16]:

```
import pickle
# Enlarge printable size
pd.set_option('display.max_columns', 10)
pd.set_option('display.width', 1000)

stats = dict()
dataframes_not_zero_cost = []

# For each agent visit list
for a in scheds.keys():
    agent_trip_desc = list()
    sched = scheds[a]
    last_meet_or_office = None
    lunch = False
```

```
    wait = 0.0
    for s in sched:
        if s[0] == 'meet' or s[0] == 'office':
            pos_desc = {"POSITION": s[2] if s[2] else 'OFFICE',
                        "COST TO REACH":␣
 ↪distance_matrix[last_meet_or_office][s[2]]*TIME_PER_DISTANCE if not␣
 ↪last_meet_or_office == None else 0.,
                        "REACHED AT": s[1],
                        "SERVICE TIME": data_dict[s[2]]['SERVICE'] if s[2] else␣
 ↪3600.0,
                        #"REAL S. TIME": c[prev, v_mod, a].X,
                        "LEAVING TIME": s[1] + data_dict[s[2]]['SERVICE'] if␣
 ↪s[2] else 3600.0,
                        "LUNCH_BEFORE": 1.0 if lunch else 0.0,
                        "WAIT_BEFORE": wait
                       }
            last_meet_or_office = s[2]
            lunch = False
            wait = 0.0
            agent_trip_desc.append(pos_desc)

        if s[0] == 'lunch':
            lunch = True

        if s[0] == 'wait':
            for i, s2 in enumerate(sched):
                if s == s2 and not i == len(sched)-1:
                    wait = sched[i+1][1] - s[1]

    stats.update({a+1: pd.DataFrame(agent_trip_desc)})

    travel_costs = pd.DataFrame(agent_trip_desc).sum(axis=0)["COST TO REACH"]
    wait_costs = pd.DataFrame(agent_trip_desc).sum(axis=0)["WAIT_BEFORE"]
    total_costs = travel_costs + wait_costs
    if (travel_costs):
        dataframes_not_zero_cost.append(pd.DataFrame(agent_trip_desc))

#remove idle agents
stats2 = dict()
for a in range(len(dataframes_not_zero_cost)):
    stats2.update({a+1: pd.DataFrame(dataframes_not_zero_cost[a])})
stats = stats2

# Print agent stats
for key in stats.keys():
    print(f"\nAgent {key}")
    print(pd.DataFrame(stats[key]))
```

Agent 1

	POSITION	COST TO REACH	REACHED AT	SERVICE TIME	LEAVING TIME	LUNCH_BEFORE	WAIT_BEFORE
0	OFFICE	0.0	0.0	3600.0	3600.0	0.0	0.0
1	11	446.0	7000.0	2400.0	9400.0	0.0	0.0
2	20	830.0	11142.0	2400.0	13542.0	0.0	912.0
3	24	658.0	14200.0	2400.0	16600.0	0.0	0.0
4	OFFICE	604.0	19004.0	3600.0	3600.0	1.0	0.0
5	25	918.0	25800.0	2400.0	28200.0	0.0	0.0

Agent 2

	POSITION	COST TO REACH	REACHED AT	SERVICE TIME	LEAVING TIME	LUNCH_BEFORE	WAIT_BEFORE
0	14	0.0	600.0	2400.0	3000.0	0.0	600.0
1	7	490.0	4300.0	2400.0	6700.0	0.0	810.0
2	17	332.0	9000.0	2400.0	11400.0	0.0	1968.0
3	3	186.0	12600.0	2400.0	15000.0	0.0	1014.0
4	10	374.0	15668.0	2400.0	18068.0	0.0	294.0
5	16	732.0	20600.0	2400.0	23000.0	1.0	0.0
6	6	714.0	24400.0	2400.0	26800.0	0.0	686.0

Agent 3

	POSITION	COST TO REACH	REACHED AT	SERVICE TIME	LEAVING TIME	LUNCH_BEFORE	WAIT_BEFORE
0	OFFICE	0.0	0.0	3600.0	3600.0	0.0	0.0
1	12	648.0	5700.0	2400.0	8100.0	0.0	0.0
2	OFFICE	648.0	8748.0	3600.0	3600.0	0.0	0.0
3	22	594.0	13000.0	2400.0	15400.0	0.0	0.0
4	OFFICE	594.0	17794.0	3600.0	3600.0	1.0	0.0

```
5        4           546.0     23000.0         2400.0        25400.0            0.0
0.0
6        1           144.0     25800.0         2400.0        28200.0            0.0
256.0

Agent 4
  POSITION  COST TO REACH  REACHED AT  SERVICE TIME  LEAVING TIME  LUNCH_BEFORE
WAIT_BEFORE
0   OFFICE           0.0         0.0         3600.0         3600.0            0.0
0.0
1       21         514.0      5000.0         2400.0         7400.0            0.0
0.0
2       19         648.0      9000.0         2400.0        11400.0            0.0
952.0
3        5         361.0     11900.0         2400.0        14300.0            0.0
139.0
4       18         671.0     16000.0         2400.0        18400.0            0.0
1029.0
5       15         495.0     20800.0         2400.0        23200.0            1.0
105.0
6       26         136.0     23500.0         2400.0        25900.0            0.0
164.0

Agent 5
  POSITION  COST TO REACH  REACHED AT  SERVICE TIME  LEAVING TIME  LUNCH_BEFORE
WAIT_BEFORE
0       13           0.0      1200.0         2400.0         3600.0            0.0
1200.0
1        8         453.0      4600.0         2400.0         7000.0            0.0
547.0
2        2         208.0      9400.0         2400.0        11800.0            0.0
2192.0
3        9         630.0     13000.0         2400.0        15400.0            0.0
570.0
4   OFFICE         845.0     18045.0         3600.0         3600.0            1.0
0.0
5       23         909.0     23600.0         2400.0        26000.0            0.0
0.0
6       27          71.0     26300.0         2400.0        28700.0            0.0
229.0
```

[17]:
```python
domino_list = list()
for k in scheds.keys():
    agent_visit_list = [w[2] for w in scheds[k] if w[0] == 'meet' or w[0] ==␣
 ↪'office']
    if len(agent_visit_list)>1:
        domino_agent = list()
```

```
        for i in range(len(agent_visit_list)-1):
            domino_agent.append((agent_visit_list[i], agent_visit_list[i+1]))
        domino_list.append(domino_agent)
    else:
        domino_list.append([])
```

[18]:
```
# Dump result dict on file
with open(f"./TEST_SETS/{FILE.split('/')[-1].split('.')[0]}_solution_heuristic",␣
 ↪"wb") as file:
    pickle.dump({"stats": stats, "domino": domino_list, "positions": positions,␣
 ↪"time": exec_time}, file)
```

[ ]:
```
```