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]:
```
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]:
```
FILE = "./TEST_SETS/test_1.txt"
```

Define costant parameters

[3]:
```
# 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 = 10

# 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()]
```

2 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 and a position.

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

[7]:
```
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
    return cost
```

3 Primal master problem

We define the master problem: we provide a set of feasible schedules (each schedule has exactly one lunch and all the action times are in increasing order), and the master 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.

[8]:
```
def pmp(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 schedules.keys()
                      for s in range(len(schedules[a])) },
               name="x",
               vtype=GRB.BINARY)

    #Constrs
    oneschedperagent = mod.addConstrs((gb.quicksum(x[a,s] for s in␣
↪range(len(schedules[a]))) == 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 range(len(schedules[a]))
                                              if pos in [work[2] for work in␣
↪schedules[a][s] if work[0] == 'meet'])
                                              == 1 for pos in positions),␣
↪name='meets_constr')

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

    return mod
```

4 Useful functions for the model

[9]:
```
def swap(agents_paths):
    """ 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 agents paths
    """
    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 = []

    #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
    return all_paths

def schedules_from_mod(mod):
    """ 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[0]+']['+split[1]+')'
            exec(cmd)
    scheds = {i: l[i] for i in range(len(l))}
    return scheds

def optimize_time_windows(sched):
    """ Optimizes a schedule moving time windows. The goal is to do more work in␣
 ↪office.
        :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 = []
    for i, s in enumerate(sched):
        if s[0] == 'office':
            indexes.append(i)

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

    #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])

    if len(indexes) > 0 and 'wait' in [w[0] for w in sched[indexes[-1]:
 ↪len(sched)]]:
        slices.append(sched[indexes[-1]:len(sched)]) #if a wait is in the last␣
 ↪slice
    elif 'wait' in [w[0] for w in sched[0:len(sched)]]:
        slices.append(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:
                    #cerco l'indice del meet precedente a questo wait
```

```python
                    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]]:# se c'è il pranzo in mezzo
                            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))

    sched = newsched

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.
        :param path: an agent path expressed as a list of meets.
    """
    sched = []
    if not path:
            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
```

```python
        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.
        :return: True if the schedule is feasible, False otherwise.
    """
    time = 0.0
    for work in schedule:
        if time > work[1]:
            return False
        time = work[1]
    return True

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)
    for comb in all_combinations: #all_combination is a list of dicts
        for key in comb.keys():
            sched = path_to_schedule(comb[key])
            if is_sched_feasible(sched) and sched not in scheds[key]:
                scheds[key].append(sched)
    return scheds
```

5 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 the remaining meets.

[10]:
```
'''
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])
for i in range(len(agents_paths)):
    lunch_done = False
    if uncovered_meets:
        next_meet = uncovered_meets.pop(0)
        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 = []
        for m in uncovered_meets:
            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)
```

6 Model optimization

We iterate between master and slave problem until the cost convergence. The slave model swaps all paths nodes for all agents and generates feasible schedules. The master model chooses a subset of these schedules and starting from these reapplies the swap.

[11]:
```
#initial cost
last_cost = 0
scheds = paths_to_schedule()

for key in scheds.keys():
    for s in scheds[key]:
        optimize_time_windows(s)
for key in scheds.keys():
```

```
    last_cost = last_cost + path_cost(scheds[key][0])
print("Initial cost: " + str(last_cost))
while (True):
    scheds = all_schedule_comb(agents_paths)
    for key in scheds.keys():
        for s in scheds[key]:
            optimize_time_windows(s)
    mod = pmp(scheds)
    mod.optimize()

    scheds = schedules_from_mod(mod)
    agents_paths = {i: [w[2] for w in scheds[i] if w[0]=='meet'] for i in␣
 ↪range(len(scheds))}#update agents_paths
    cost = 0
    for key in scheds.keys():
        cost = cost + path_cost(scheds[key])
    print("\n-----------------\nPrev cost: " + str(last_cost) + ", cost: " +␣
 ↪str(cost)+"\n")
    if (last_cost <= cost):
        break
    else:
        last_cost = cost
```

```
Initial cost: 35606.0
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 20 rows, 41 columns and 122 nonzeros
Model fingerprint: 0x12c159e0
Variable types: 0 continuous, 41 integer (41 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [6e+02, 2e+04]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Found heuristic solution: objective 34886.000000
Presolve removed 20 rows and 41 columns
Presolve time: 0.01s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.03 seconds
Thread count was 1 (of 8 available processors)

Solution count 2: 33155

Optimal solution found (tolerance 1.00e-04)
Best objective 3.315500000000e+04, best bound 3.315500000000e+04, gap 0.0000%
```

```
-----------------
Prev cost: 35606.0, cost: 33155.0

Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 20 rows, 41 columns and 122 nonzeros
Model fingerprint: 0x4a17b2c2
Variable types: 0 continuous, 41 integer (41 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [6e+02, 2e+04]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Found heuristic solution: objective 32435.000000
Presolve removed 20 rows and 41 columns
Presolve time: 0.00s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.01 seconds
Thread count was 1 (of 8 available processors)

Solution count 1: 32435

Optimal solution found (tolerance 1.00e-04)
Best objective 3.243500000000e+04, best bound 3.243500000000e+04, gap 0.0000%

-----------------
Prev cost: 33155.0, cost: 32435.0

Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 20 rows, 41 columns and 122 nonzeros
Model fingerprint: 0x4415c595
Variable types: 0 continuous, 41 integer (41 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [6e+02, 2e+04]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Found heuristic solution: objective 33155.000000
Presolve removed 20 rows and 41 columns
Presolve time: 0.00s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.01 seconds
Thread count was 1 (of 8 available processors)

Solution count 2: 32435
```

```
Optimal solution found (tolerance 1.00e-04)
Best objective 3.243500000000e+04, best bound 3.243500000000e+04, gap 0.0000%

-----------------
Prev cost: 32435.0, cost: 32435.0
```

[12]:

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

# 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 last_meet_or_office 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": lunch,
                        "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:
                    wait = sched[i+1][1] - s[1]

    # Print agent stats
```

```
    print(f"\nAgent {a+1}")
    print(pd.DataFrame(agent_trip_desc))
```

```
Agent 1
  POSITION  COST TO REACH  REACHED AT  SERVICE TIME  LEAVING TIME  LUNCH_BEFORE
WAIT_BEFORE
0        8            0.0      3600.0        2400.0        6000.0         False
3600.0
1        2          208.0      8400.0        2400.0       10800.0         False
2192.0
2        9          630.0     13000.0        2400.0       15400.0         False
1570.0
3   OFFICE          845.0     18045.0        3600.0        3600.0          True
0.0
4        1            0.0     24800.0        2400.0       27200.0         False
0.0

Agent 2
   POSITION  COST TO REACH  REACHED AT  SERVICE TIME  LEAVING TIME  LUNCH_BEFORE
WAIT_BEFORE
0         7            0.0      4700.0        2400.0        7100.0         False
4700.0
1         5          381.0     11900.0        2400.0       14300.0         False
4419.0
2        10          367.0     16800.0        2400.0       19200.0         False
2133.0
3         4          614.0     23400.0        2400.0       25800.0          True
1786.0

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         False
0.0
1        6            0.0      7100.0        2400.0        9500.0         False
0.0
2        3          573.0     10600.0        2400.0       13000.0         False
527.0
3   OFFICE          918.0     13918.0        3600.0        3600.0         False
0.0
4   OFFICE            0.0     16200.0        3600.0        3600.0          True
0.0

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         False
0.0
1   OFFICE            0.0     16200.0        3600.0        3600.0          True
0.0

Agent 5
  POSITION  COST TO REACH  REACHED AT  SERVICE TIME  LEAVING TIME  LUNCH_BEFORE
WAIT_BEFORE
0   OFFICE            0.0         0.0        3600.0        3600.0         False
0.0
1   OFFICE            0.0     16200.0        3600.0        3600.0          True
0.0

Agent 6
  POSITION  COST TO REACH  REACHED AT  SERVICE TIME  LEAVING TIME  LUNCH_BEFORE
WAIT_BEFORE
0   OFFICE            0.0         0.0        3600.0        3600.0         False
0.0
1   OFFICE            0.0     16200.0        3600.0        3600.0          True
0.0

Agent 7
  POSITION  COST TO REACH  REACHED AT  SERVICE TIME  LEAVING TIME  LUNCH_BEFORE
WAIT_BEFORE
0   OFFICE            0.0         0.0        3600.0        3600.0         False
0.0
1   OFFICE            0.0     16200.0        3600.0        3600.0          True
0.0

Agent 8
  POSITION  COST TO REACH  REACHED AT  SERVICE TIME  LEAVING TIME  LUNCH_BEFORE
WAIT_BEFORE
0   OFFICE            0.0         0.0        3600.0        3600.0         False
0.0
1   OFFICE            0.0     16200.0        3600.0        3600.0          True
0.0

Agent 9
  POSITION  COST TO REACH  REACHED AT  SERVICE TIME  LEAVING TIME  LUNCH_BEFORE
WAIT_BEFORE
0   OFFICE            0.0         0.0        3600.0        3600.0         False
0.0
1   OFFICE            0.0     16200.0        3600.0        3600.0          True
0.0

Agent 10
  POSITION  COST TO REACH  REACHED AT  SERVICE TIME  LEAVING TIME  LUNCH_BEFORE
WAIT_BEFORE
```

```
0   OFFICE            0.0         0.0        3600.0        3600.0         False
0.0
1   OFFICE            0.0     16200.0        3600.0        3600.0          True
0.0
```

[13]:
```python
print(scheds)
```

```
{0: [('wait', 0.0, -1), ('meet', 3600.0, 8), ('travel', 6000.0, -1), ('wait',
6208.0, -1), ('meet', 8400.0, 2), ('travel', 10800.0, -1), ('wait', 11430.0,
-1), ('meet', 13000.0, 9), ('lunch', 15400.0, -1), ('travel', 17200.0, -1),
('office', 18045.0, 0), ('travel', 24305.0, -1), ('meet', 24800.0, 1), ('wait',
27200.0, -1), ('end', 28800, -1)], 1: [('wait', 0.0, -1), ('meet', 4700.0, 7),
('travel', 7100.0, -1), ('wait', 7481.0, -1), ('meet', 11900.0, 5), ('travel',
14300.0, -1), ('wait', 14667.0, -1), ('meet', 16800.0, 10), ('lunch', 19200.0,
-1), ('travel', 21000.0, -1), ('wait', 21614.0, -1), ('meet', 23400.0, 4),
('wait', 25800.0, -1), ('end', 28800, -1)], 2: [('office', 0.0, 0), ('travel',
5223.0, -1), ('meet', 7100.0, 6), ('travel', 9500.0, -1), ('wait', 10073.0, -1),
('meet', 10600.0, 3), ('travel', 13000.0, -1), ('office', 13918.0, 0), ('lunch',
14400, -1), ('office', 16200, 0), ('end', 28800, -1)], 3: [('office', 0.0, 0),
('lunch', 14400, -1), ('office', 16200, 0), ('end', 28800, -1)], 4: [('office',
0.0, 0), ('lunch', 14400, -1), ('office', 16200, 0), ('end', 28800, -1)], 5:
[('office', 0.0, 0), ('lunch', 14400, -1), ('office', 16200, 0), ('end', 28800,
-1)], 6: [('office', 0.0, 0), ('lunch', 14400, -1), ('office', 16200, 0),
('end', 28800, -1)], 7: [('office', 0.0, 0), ('lunch', 14400, -1), ('office',
16200, 0), ('end', 28800, -1)], 8: [('office', 0.0, 0), ('lunch', 14400, -1),
('office', 16200, 0), ('end', 28800, -1)], 9: [('office', 0.0, 0), ('lunch',
14400, -1), ('office', 16200, 0), ('end', 28800, -1)]}
```

[ ]:
```python

```