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_3.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 = 6

# 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 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.

[8]:
```
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 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
```

```
                    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]
```

```python
    #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
```

```
        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]:
```
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])
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 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 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 = set_cover_ILP(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

#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(exec_time)
```

```
Initial cost: 65180.0
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 33 rows, 227 columns and 910 nonzeros
Model fingerprint: 0x7e922dbe
Variable types: 0 continuous, 227 integer (227 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [4e+02, 2e+04]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 3 rows and 73 columns
Presolve time: 0.03s
```

```
Presolved: 30 rows, 154 columns, 614 nonzeros
Variable types: 0 continuous, 154 integer (154 binary)

Root relaxation: objective 5.654833e+04, 74 iterations, 0.01 seconds

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

     0     0 56548.3333    0   14          - 56548.3333      -     -    0s
H    0     0                    60873.000000 56548.3333  7.10%     -    0s
H    0     0                    57396.000000 56548.3333  1.48%     -    0s
*    0     0               0    57370.000000 57370.0000  0.00%     -    0s

Cutting planes:
  Gomory: 1
  Clique: 1
  Zero half: 5

Explored 1 nodes (84 simplex iterations) in 0.12 seconds
Thread count was 8 (of 8 available processors)

Solution count 3: 57370 57396 60873

Optimal solution found (tolerance 1.00e-04)
Best objective 5.737000000000e+04, best bound 5.737000000000e+04, gap 0.0000%

-----------------
Prev cost: 65180.0, cost: 57370.0

Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 33 rows, 226 columns and 906 nonzeros
Model fingerprint: 0x5e7bdbfb
Variable types: 0 continuous, 226 integer (226 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [4e+02, 1e+04]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 4 rows and 79 columns
Presolve time: 0.02s
Presolved: 29 rows, 147 columns, 580 nonzeros
Variable types: 0 continuous, 147 integer (147 binary)

Root relaxation: objective 5.246525e+04, 48 iterations, 0.00 seconds

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

```
    0     0 52465.2500    0   16          - 52465.2500      -      -    0s
H    0     0                    57590.000000 52465.2500  8.90%     -    0s
H    0     0                    55003.000000 52465.2500  4.61%     -    0s
     0     0 55003.0000    0   19 55003.0000 55003.0000  0.00%     -    0s

Cutting planes:
  Gomory: 2
  Clique: 1
  Zero half: 1

Explored 1 nodes (56 simplex iterations) in 0.07 seconds
Thread count was 8 (of 8 available processors)

Solution count 2: 55003 57590

Optimal solution found (tolerance 1.00e-04)
Best objective 5.500300000000e+04, best bound 5.500300000000e+04, gap 0.0000%

-----------------
Prev cost: 57370.0, cost: 55003.0

Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 33 rows, 225 columns and 895 nonzeros
Model fingerprint: 0x7c6596f2
Variable types: 0 continuous, 225 integer (225 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [4e+02, 1e+04]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 7 rows and 100 columns
Presolve time: 0.02s
Presolved: 26 rows, 125 columns, 441 nonzeros
Variable types: 0 continuous, 125 integer (125 binary)

Root relaxation: objective 4.925700e+04, 42 iterations, 0.00 seconds

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

     0     0 49257.0000    0    8          - 49257.0000      -      -    0s
H    0     0                    54409.000000 49257.0000  9.47%     -    0s
H    0     0                    52458.000000 49257.0000  6.10%     -    0s
     0     0 50304.0000    0    8 52458.0000 50304.0000  4.11%     -    0s
H    0     0                    51730.000000 50304.0000  2.76%     -    0s

Cutting planes:
  Gomory: 2
```

```
  Zero half: 1

Explored 1 nodes (49 simplex iterations) in 0.09 seconds
Thread count was 8 (of 8 available processors)

Solution count 3: 51730 52458 54409

Optimal solution found (tolerance 1.00e-04)
Best objective 5.173000000000e+04, best bound 5.173000000000e+04, gap 0.0000%

-----------------
Prev cost: 55003.0, cost: 51730.0

Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 33 rows, 155 columns and 676 nonzeros
Model fingerprint: 0x64d656a2
Variable types: 0 continuous, 155 integer (155 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [4e+02, 1e+04]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 11 rows and 93 columns
Presolve time: 0.01s
Presolved: 22 rows, 62 columns, 228 nonzeros
Variable types: 0 continuous, 62 integer (62 binary)

Root relaxation: objective 4.857400e+04, 20 iterations, 0.00 seconds

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

*    0     0               0    48574.000000 48574.0000  0.00%     -    0s

Explored 0 nodes (20 simplex iterations) in 0.03 seconds
Thread count was 8 (of 8 available processors)

Solution count 1: 48574

Optimal solution found (tolerance 1.00e-04)
Best objective 4.857400000000e+04, best bound 4.857400000000e+04, gap 0.0000%

-----------------
Prev cost: 51730.0, cost: 48574.0

Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 33 rows, 155 columns and 676 nonzeros
Model fingerprint: 0xfcab9dc3
```

```
Variable types: 0 continuous, 155 integer (155 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [4e+02, 1e+04]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 11 rows and 93 columns
Presolve time: 0.01s
Presolved: 22 rows, 62 columns, 228 nonzeros
Variable types: 0 continuous, 62 integer (62 binary)

Root relaxation: objective 4.857400e+04, 19 iterations, 0.00 seconds

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

*    0     0               0    48574.000000 48574.0000  0.00%     -    0s

Explored 0 nodes (19 simplex iterations) in 0.04 seconds
Thread count was 8 (of 8 available processors)

Solution count 1: 48574

Optimal solution found (tolerance 1.00e-04)
Best objective 4.857400000000e+04, best bound 4.857400000000e+04, gap 0.0000%

-----------------
Prev cost: 48574.0, cost: 48574.0

1.441546
```

[12]:
```python
import pickle
# Enlarge printable size
pd.set_option('display.max_columns', 10)
pd.set_option('display.width', 1000)

stats = dict()

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

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

    # 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        14            0.0      1000.0        2400.0        3400.0           0.0
1000.0
1         7          700.0      4100.0        2400.0        6500.0           0.0
0.0
2        17          446.0      9000.0        2400.0       11400.0           0.0
2054.0
3         9          280.0     14000.0        2400.0       16400.0           0.0
2320.0
4        16          334.0     20600.0        2400.0       23000.0           1.0
2066.0
5        23          440.0     23600.0        2400.0       26000.0           0.0
```

```
160.0
6        27           71.0     26300.0        2400.0       28700.0           0.0
229.0

Agent 2
   POSITION  COST TO REACH  REACHED AT  SERVICE TIME  LEAVING TIME  LUNCH_BEFORE
WAIT_BEFORE
0        21            0.0      3000.0        2400.0        5400.0           0.0
3000.0
1        11          114.0      7000.0        2400.0        9400.0           0.0
1486.0
2         3          474.0     11600.0        2400.0       14000.0           0.0
1726.0
3        18           70.0     16000.0        2400.0       18400.0           0.0
1930.0
4        15          495.0     20800.0        2400.0       23200.0           1.0
105.0
5        26          136.0     23500.0        2400.0       25900.0           0.0
164.0

Agent 3
   POSITION  COST TO REACH  REACHED AT  SERVICE TIME  LEAVING TIME  LUNCH_BEFORE
WAIT_BEFORE
0        13            0.0      3670.0        2400.0        6070.0           0.0
3670.0
1         6          430.0      6500.0        2400.0        8900.0           0.0
0.0
2        20          400.0     13300.0        2400.0       15700.0           0.0
4000.0
3        10          228.0     16000.0        2400.0       18400.0           0.0
72.0
4         4          614.0     23000.0        2400.0       25400.0           1.0
2186.0
5         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        8            0.0      4600.0        2400.0        7000.0           0.0
4600.0
1       19          798.0      8220.0        2400.0       10620.0           0.0
422.0
2        5          361.0     10981.0        2400.0       13381.0           0.0
0.0
3       24          819.0     14200.0        2400.0       16600.0           0.0
0.0
4   OFFICE          604.0     19004.0        3600.0        3600.0           1.0
```

```
0.0

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           0.0
0.0
1       12            0.0      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            0.0     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       25            0.0     24800.0        2400.0       27200.0           0.0
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           0.0
0.0
1        2            0.0      8400.0        2400.0       10800.0           0.0
0.0
2   OFFICE          284.0     11084.0        3600.0        3600.0           0.0
0.0
3   OFFICE            0.0     16200.0        3600.0        3600.0           1.0
0.0
```

[13]:
```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([])
```

[14]:
```python
# 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)
```