

Algoritmi di Ottimizzazione

Agent Scheduling Problem

Francesco Ghinelli

Matr: 150117

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);

1.1 Approach to the problem

To solve this problem in an easier way, i've calculated a distance matrix, representing distance between all different positions.

To solve the problem of multiple admissible routes to the office from the same customer, I used variables that contain double the positions actually needed. In fact, the positions will start with that relating to the office, followed by those relating to the various customers, then there is a position that would be empty, which has been used as a fictitious starting and ending point, finally there are a quantity of positions equal to number of customers used to represent a journey from the customer to the office.

This approach is needed to keep a simpler model creating circular and continuous agent paths.

Data for this model is:

- $clients \rightarrow$ Clients count.
- $distance_{i,j}$, for i, j in $clients + 1$. \rightarrow Distance matrix.
- $work_start \rightarrow$ Agents working day start.
- $work_end \rightarrow$ Agents working day end.
- $lunch_start \rightarrow$ Agents lunch time window start.
- $lunch_end \rightarrow$ Agents lunch time window end.
- $window_start_i$, for i in $clients + 1$. \rightarrow Office/clients service window start.
- $window_end_i$, for i in $clients + 1$. \rightarrow Office/clients service window end.

Variables for this model are:

- $x_{i,j,a} \in \{0,1\}$, $i \in ALL$, $j \in ALL$, $a \in A \rightarrow$ Representing agent a doing path between i and j .
- $s_{i,a} \in \mathbb{N}$, $i \in ALL$, $a \in A \rightarrow$ Representing service time of agent a to client i .
- $c_{i,a} \in \mathbb{N}$, $i \in ALL$, $a \in A \rightarrow$ Representing agent a service lasting to client i .
- $w_{i,a} \in \mathbb{N}$, $i \in ALL$, $a \in A \rightarrow$ Representing agent a wait time before client i .
- $l_{i,j,a} \in \{0,1\}$, $i \in ALL$, $j \in ALL$, $a \in A \rightarrow$ Representing agent a doing lunch between i and j .
- $t_a \in \mathbb{N}$, $a \in A \rightarrow$ Representing max service time of agent a .

Objective function

Objective function for this model want to minimize agents traveling time, wait time and service time. First two are obvious, last one is needed to keep wait time near to the client that require them.

Objective function use two multiplicative coefficients applied to traveling times and to wait times to give more importance to one or the other.

Constraints for this model are:

1. Serve all clients.
2. All duplicated Offices must be visited from at most an Agent.
3. All Agents start their trip from fictitious location.
4. Agents can't start their trip from fictitious location and go to only destination Office.
5. All Agents end their trip in fictitious location.
6. Each served location has a served location before
7. Agent can't do loop between same Client.
8. Sum of time spent into a location in a specific trip must be equal to Service time of that location.
9. Sum of time spent into Office in a specific trip must be equal to Office service time.
10. Agent can go to a location only to spend time.
11. Agent must go office at least once or start from it [0].
12. Agent can't go office from office.
13. That is an only start Office position (Not fictitious).
14. Agent can go to (fictitious) Office only if he visited the associated Client.
15. Agent can go to (fictitious) Office just after he visit the associated Client.
16. Agent serve Client after his time window start.
17. Agent serve Client before his time window end.
18. If Agent has lunch it must be after lunch time start.
19. If Agent has lunch it must be before lunch time end.
20. If Agent have lunch between Clients I and J, agent have to make trip between I and J.
21. Getting maximum Agent service time.
22. If Agent working time is greater than lunch time than Agent must have lunch.
23. If Agent working time is lesser than lunch time than Agent must not have lunch.
24. Service time of Client J is equal to sum of: wait time of client J, service time of client I before J, trip between I and J, time of service of client I, lunch time if present.
25. Sum of minutes spent in: travels, servicing clients, waiting, eating at lunch.

1.2 Model

$$\min \text{travel_cost} * \sum_{i \in \text{ALL}} \sum_{j \in \text{ALL}} \sum_{a \in A} x_{i,j,a} * \text{distance}_{i,j} + \text{wait_cost} * \sum_{i \in C} \sum_{a \in A} w_{i,a} + \sum_{i \in \text{RBLE}} \sum_{a \in A} s_{i,a}$$

- 1) $\sum_{i \in \text{ALL}} \sum_{a \in A} x_{i,j,a} = 1, \quad j \in C$
- 2) $\sum_{i \in \text{ALL}} \sum_{a \in A} x_{i,j,a} \leq 1, \quad j \in \text{FO}$
- 3) $\sum_{j \in \text{REAL}} x_{\text{start},j,a} = 1, \quad a \in A$
- 4) $x_{\text{start},j,a} = 0, \quad j \in \text{FO}, a \in A$
- 5) $\sum_{i \in \text{RBLE}} x_{i,s,a} = 1, \quad a \in A$
- 6) $\sum_{i \in \text{ALL}} x_{i,h,a} - \sum_{j \in \text{ALL}} x_{h,j,a} = 0, \quad h \in \text{ALL}, a \in A$
- 7) $x_{i,i,a} = 0, \quad i \in \text{ALL}, a \in A$
- 8) $c_{j,a} = \text{service}_j * \sum_{i \in \text{ALL}} x_{i,j,a}, \quad i \in C, a \in A$
- 9) $\sum_{i \in \text{ALL}} \sum_{j \in o + \text{FO}} c_{j,a} * x_{i,j,a} \geq \text{service}_o, \quad a \in A$
- 10) $c_{j,a} \geq x_{i,j,a}, \quad i \in \text{ALL}, j \in \text{RBLE}, a \in A$
- 11) $\sum_{i \in \text{ALL}} \sum_{j \in o + \text{FO}} x_{i,j,a} \geq 1, \quad a \in A$
- 12) $x_{i,j,a} = 0, \quad i \in o + \text{FO}, j \in o + \text{FO}, a \in A$
- 13) $x_{i,o,a} = 0, \quad i \in C, a \in A$
- 14) $x_{j,j + \text{clients},a} \leq \sum_{i \in \text{ALL}} x_{i,j,a}, \quad j \in C, a \in A$
- 15) $\{x_{i,j,a} = 0 \text{ if } i \neq j - \text{clients}\}, \quad i \in \text{REAL}, j \in \text{FO}, a \in A$
- 16) $\text{window_start}_i * \sum_{h \in \text{ALL}} x_{h,i,a} \leq s_{i,a}, \quad i \in C, a \in A$
- 17) $\text{window_end}_i * \sum_{h \in \text{ALL}} x_{h,i,a} \geq s_{i,a}, \quad i \in C, a \in A$
- 18) $l_{i,j,a} * (s_{i,a} + c_{i,a} + \text{distance}_{i,j} + w_{j,a}) \geq l_{i,j,a} * \text{lunch_start},$
 $i \in \text{RBLE}, j \in \text{RBLE}, a \in A$
- 19) $l_{i,j,a} * (s_{i,a} + c_{i,a} + \text{distance}_{i,j} + w_{j,a}) \leq l_{i,j,a} * \text{lunch_end},$
 $i \in \text{RBLE}, j \in \text{RBLE}, a \in A$
- 20) $l_{i,j,a} \leq x_{i,j,a} * \text{lunch_end}, \quad i \in \text{RBLE}, j \in \text{RBLE}, a \in A$
- 21) $t_a = \max(s_{i,a}, i \in \text{RBLE}), \quad a \in A$
- 22) $\text{work_end} * \sum_{i \in \text{RBLE}} \sum_{j \in \text{RBLE}} l_{i,j,a} \geq t_a - \text{lunch_start}, \quad a \in A$
- 23) $\text{work_end} * \sum_{i \in \text{RBLE}} \sum_{j \in \text{RBLE}} (1 - l_{i,j,a}) \geq \text{lunch_start} - t_a, \quad a \in A$
- 24) $s_{j,a} = \sum_{i \in \text{RBLE}, i \neq j} x_{i,j,a} * (s_{i,a} + c_{i,a} +$
 $\text{distance}_{i,j} + l_{i,j,a} * \text{lunch_len}) + w_{j,a}, \quad j \in \text{RBLE}, a \in A$
- 25) $\sum_{i \in \text{RBLE}} \sum_{j \in \text{RBLE}} x_{i,j,a} * \text{distance}_{i,j} +$
 $\sum_{i \in \text{ALL}} \sum_{j \in \text{RBLE}} x_{i,j,a} * \text{service}_j +$
 $\sum_{i \in \text{RBLE}} w_{i,a} +$
 $\sum_{i \in \text{RBLE}} \sum_{j \in \text{RBLE}} l_{i,j,a} * \text{lunch_len} +$
 $\text{work_start} \leq \text{work_end}, \quad a \in A$

$$x_{i,j,a} \in \{0,1\}, i \in \text{ALL}, j \in \text{ALL}, a \in A$$

$$s_{i,a} \in \mathbb{N}, i \in \text{ALL}, a \in A$$

$$c_{i,a} \in \mathbb{N}, i \in \text{ALL}, a \in A$$

$$w_{i,a} \in \mathbb{N}, i \in \text{ALL}, a \in A$$

$$l_{i,j,a} \in \{0,1\}, i \in \text{ALL}, j \in \text{ALL}, a \in A$$

$$t_a \in \mathbb{N}, a \in A$$

$$\text{start} = \text{clients} + 1$$

$$o = 0$$

$$C = \{1, \dots, \text{clients}\}$$

$$\text{ALL} = \{0, \dots, 2\text{clients} + 1\}$$

$$\text{REAL} = \{0, \dots, \text{clients}\}$$

$$\text{RBLE} = \text{ALL} - \{s\}$$

$$\text{FO} = \{P + 2, \dots, 2\text{clients} + 1\}$$

1.3 Model implementation

1.3.1 Importing libraries

```
[1]: from gurobipy import*

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

1.3.2 Define file with clients data

```
[2]: FILE = "./TEST_SETS/test_2.txt"
```

1.3.3 Define constant 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 = 3

# 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
```

1.3.4 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
    data_dict.update({OFFICE_NUM: {'X': OFFICE_X,
                                    'Y': OFFICE_Y,
                                    'DEMAND': AGENTS,
                                    'READYTIME': OFFICE_READYTIME,
                                    'DUE DATE': OFFICE_DUE DATE,
                                    '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))

```

```

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

```

1.4 Gurobi Model

1.4.1 Setup model parameters

```
[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 FICTITIOUS 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 interfere 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
data_dict.update({CLIENTS: {'X': 0,
                              'Y': 0,
                              'DEMAND': AGENTS,
                              'READYTIME': WORKING_TIME_RANGE[0],
                              'DUE DATE': WORKING_TIME_RANGE[1],
                              'SERVICE': 0,}})

# POSITIONS SETS FOR CLEANER MODEL
agent_list = list(range(AGENTS))
all_pos = list(range(CLIENTS*2))
start_pos = CLIENTS
client_pos = list(range(1,CLIENTS))
only_start_office_pos = 0
no_duplicates_pos = list(range(0,CLIENTS))
destination_office_pos = list(range(CLIENTS+1, CLIENTS*2))
office_pos = [only_start_office_pos]+destination_office_pos
reachable_pos = [p for p in range(CLIENTS*2) if p not in [start_pos,]]
```

1.4.2 Add Multipliers to minimize wait time over travel time

```
[7]: # Multiplier for distance and wait costs
TRAVEL_COST_MULTIPLIER = 1
WAIT_COST_MULTIPLIER = 2
```

1.4.3 Create the model

```
[8]: # Create model
mod = Model("TSPTW")
```

Academic license - for non-commercial use only - expires 2021-08-11
Using license file /opt/gurobi/gurobi.lic

1.4.4 Variables

IMPORTANT! Office IS ONE AND ONLY ONE but because i wasn't unable to find a better solution for multiple Agent visiting it i've repeated Office (originally in position 0) in all position greater than number_of_clients + 1.

WARNING! Position from number_of_clients + 2 forward are office positions reachable only from client_pos = office_pos - number_of_clients.

WARNING! Position number_of_clients + 1 is used as fictitious start and end trip loop location, position 0 is office too, Agents can only start from office 0 they can't go there.

```
[9]: # Agent trip
x = mod.addVars({(i,j,a): 0 for i in all_pos
                    for j in all_pos
                    for a in agent_list},
                name="x",
                vtype=GRB.BINARY)

# Serve time
s = mod.addVars({(i,a): 0 for i in all_pos
                    for a in agent_list},
                name="s",
                vtype=GRB.INTEGER)

# Serve Client/Office lasting
c = mod.addVars({(i,a): 0 for i in all_pos
                    for a in agent_list},
                name="c",
                vtype=GRB.INTEGER)

# Wait time
w = mod.addVars({(i,a): 0 for i in all_pos
                    for a in agent_list},
                name="w",
                vtype=GRB.INTEGER,
                lb=0)

# Lunch done between customers
l = mod.addVars({(i,j,a): 0 for i in all_pos
                    for j in all_pos
                    for a in agent_list},
                name="l",
                vtype=GRB.BINARY)

# Max working time
t = mod.addVars({(a): 0 for a in agent_list},
                name="t",
                vtype=GRB.INTEGER)
```

1.4.5 Constraints

Trip constraints

```
[10]: # 1 - All client must be visited from an Agent
      _= mod.addConstrs((quicksum(x[i,j,a]
                                for i in all_pos
                                for a in agent_list) == 1
                                for j in client_pos),
                        name="ServeAll")

      # 2 - All duplicated Offices must be visited from at most an Agent
      _= mod.addConstrs((quicksum(x[i,j,a]
                                for i in all_pos
                                for a in agent_list) <= 1
                                for j in destination_office_pos),
                        name="ServeDuplicatesOffice")

      # 3 - All Agents start their trip from fictitious location
      _= mod.addConstrs((quicksum(x[start_pos,j,a]
                                for j in no_duplicates_pos) == 1
                                for a in agent_list),
                        name="StartFromFitticious")

      # 4 - Agents can't start their trip from fictitious location and go to only destination_
      ↪Office
      _= mod.addConstrs((x[start_pos,j,a] == 0
                                for j in destination_office_pos
                                for a in agent_list),
                        name="NotStartFromFitticiousToOffice")

      # 5 - All Agents end their trip in fictitious location
      _= mod.addConstrs((quicksum(x[i,start_pos,a]
                                for i in reachable_pos) == 1
                                for a in agent_list),
                        name="EndInFitticious")

      # 6 - Each served location has a served location before
      _= mod.addConstrs((quicksum(x[i,h,a] for i in all_pos) -
                        quicksum(x[h,j,a] for j in all_pos) == 0
                        for h in all_pos
                        for a in agent_list),
                        name="ContinuousLoops")

      # 7 - Agent can't do loop between same Client
      _= mod.addConstrs((x[i,i,a] == 0
                        for a in agent_list
                        for i in all_pos),
                        name="NoSelfLoops")
```

Service time constraints

```
[11]: # 8 - Sum of time spent into a location in a specific trip must be equal to Service
#      time of that location
_= mod.addConstrs((c[j,a] == quicksum(x[i,j,a] for i in all_pos) *
    ↪data_dict[j]['SERVICE']
            for j in client_pos
            for a in agent_list),
            name="ServingTime")

# 9 - Sum of time spent into Office in a specific trip must be equal to Office service
    ↪time
_= mod.addConstrs((quicksum(c[j,a]*x[i,j,a]
            for i in all_pos
            for j in office_pos) >= data_dict[0]['SERVICE']
            for a in agent_list),
            name="ServingTimeOffice")

# 10 -Agent can go to a location only to spend time
_= mod.addConstrs((c[j,a] >= x[i,j,a]
            for i in all_pos
            for j in reachable_pos
            for a in agent_list),
            name="GoOnlyIfNeeded")
```

Office constraints

```
[12]: # 11 - Agent must go office at least once or start from it [0]
_= mod.addConstrs((quicksum(x[i,j,a]
            for i in all_pos
            for j in office_pos) >= 1
            for a in agent_list),
            name="ServeOffice")

# 12 - Agent can't go office from office
_= mod.addConstrs((x[i,j,a] == 0
            for i in office_pos
            for j in office_pos
            for a in agent_list),
            name="NoOfficeFromOfficeB")

# 13 - That is an only start Office position (Not Fitticious)
_= mod.addConstrs((x[i,0,a] == 0
            for i in client_pos
            for a in agent_list),
            name="NoOfficeFromOfficeC")

# 14 - Agent can go to (Fitticious) Office only if he visited the associated Client
_= mod.addConstrs((x[j,j+CLIENTS,a] <= quicksum(x[i,j,a] for i in all_pos)
            for j in client_pos
            for a in agent_list),
```

```
name="GoOfficeOnlyIfVisitedItsClient")
```

```
# 15 - Agent can go to (Fitticious) Office just after he visit the associated Client
```

```
_ = mod.addConstrs((x[i,j,a] == 0
    for i in no_duplicates_pos
    for j in destination_office_pos
    for a in agent_list
    if i != j-CLIENTS),
    name="GoOfficeAfterItsClient")
```

Time windows constraints

[13]: *# 16 - Agent serve Client after his time window start*

```
_ = mod.addConstrs((data_dict[i]['READYTIME'] *
    quicksum(x[h,i,a]
        for h in all_pos) <= s[i,a]
    for i in client_pos
    for a in agent_list),
    name="ServeAfterTWStart")
```

```
# 17 - Agent serve Client before his time window end
```

```
_ = mod.addConstrs((data_dict[i]['DUEDATE'] *
    quicksum(x[h,i,a]
        for h in all_pos) >= s[i,a]
    for i in client_pos
    for a in agent_list),
    name="ServeBeforeTWEnd")
```

Lunch break constraints

[14]: *# 18 - If Agent has lunch it must be after lunch time start*

```
_ = mod.addConstrs((l[i,j,a]*(s[i,a] +
    c[i,a] +
    distance_matrix[i if i in no_duplicates_pos else 0][j if j
    in no_duplicates_pos else 0] +
    w[j,a]) >= l[i,j,a]*LUNCH_BREAK_RANGE[0]
    for i in reachable_pos
    for j in reachable_pos
    for a in agent_list),
    name="LunchTimeStart")
```

```
# 19 - If Agent has lunch it must be before lunch time end
```

```
_ = mod.addConstrs((l[i,j,a]*(s[i,a] +
    c[i,a] +
    distance_matrix[i if i in no_duplicates_pos else 0][j if j
    in no_duplicates_pos else 0] +
    w[j,a]) <= l[i,j,a]*LUNCH_BREAK_RANGE[1]
    for i in reachable_pos
    for j in reachable_pos
    for a in agent_list),
    name="LunchTimeEnd")
```

```

# 20 - If Agent have lunch between Clients I and J, agent have to make trip between I
↳ and J
_= mod.addConstrs((l[i,j,a] <= x[i,j,a]
                    for i in reachable_pos
                    for j in reachable_pos
                    for a in agent_list),
                    name="LunchTime")

# 21 - Getting maximum Agent service time
_= mod.addConstrs((t[a] == max_(s[i,a] for i in reachable_pos)
                    for a in agent_list),
                    name="MaxServiceTime")

# 22 - If Agent working time is greater than lunch time than Agent must have lunch
_= mod.addConstrs((WORKING_TIME_RANGE[1] * (quicksum(l[i,j,a]
                                                        for i in reachable_pos
                                                        for j in reachable_pos)) >= t[a] -
↳ LUNCH_BREAK_RANGE[0]
                    for a in agent_list),
                    name="AgentNeedLunchI")

# 23 - If Agent working time is lesser than lunch time than Agent must not have lunch
_= mod.addConstrs((WORKING_TIME_RANGE[1] * (1 - quicksum(l[i,j,a]
                                                        for i in reachable_pos
                                                        for j in reachable_pos)) >=
↳ LUNCH_BREAK_RANGE[0] - t[a]
                    for a in agent_list),
                    name="AgentNeedLunchII")

```

Service time constraint

```

[15]: # 24 - Service time of Client J is equal to sum of:
#       wait_time_of_client_J, service_time_of_client_I_before_of_J,
↳ trip_between_I_and_J,
#       time_of_service_of_client_I, lunch_time_if_present
_= mod.addConstrs((s[j,a] == quicksum(x[i,j,a]*
                                        (s[i,a] +
                                        c[i,a] +
                                        distance_matrix[i if i in no_duplicates_pos else
↳ 0][j if j in no_duplicates_pos else 0] +
                                        l[i,j,a]*LUNCH_BREAK_TIME)
                                        for i in reachable_pos if i!=j) +
                    w[j,a]
                    for a in agent_list
                    for j in reachable_pos),
                    name="RouteInTime")

```

Working day constraint

```
[16]: # 25 - Sum of minutes spent in: travels, servicing clients, waiting, eating at lunch
      _ = mod.addConstrs((quicksum(x[i,j,a] *
                                distance_matrix[i if i in no_duplicates_pos else 0][j if j in no_duplicates_pos else 0]
                                for i in reachable_pos
                                for j in reachable_pos) +
                        quicksum(x[i,j,a] *
                                data_dict[j if j in no_duplicates_pos else 0]['SERVICE']
                                for i in all_pos
                                for j in reachable_pos) +
                        quicksum(w[i,a]
                                for i in reachable_pos) +
                        quicksum(l[i,j,a]
                                for i in reachable_pos
                                for j in reachable_pos) *
                        LUNCH_BREAK_TIME +
                        WORKING_TIME_RANGE[0] <= WORKING_TIME_RANGE[1]
                        for a in agent_list),
      name="MaxHours")
```

1.4.6 Objective Function

This objective function wants to minimize time spent traveling between clients and awaiting for clients time windows.

```
[17]: mod.setObjective(quicksum(x[i,j,a] *  
                                distance_matrix[i if i in no_duplicates_pos else 0][j if j in_  
                                no_duplicates_pos else 0]  
                                for i in all_pos  
                                for j in all_pos  
                                for a in agent_list) * TRAVEL_COST_MULTIPLIER +  
                                quicksum(w[i,a]  
                                for i in reachable_pos  
                                for a in agent_list) * WAIT_COST_MULTIPLIER +  
                                quicksum(s[i,a]  
                                for i in reachable_pos  
                                for a in agent_list),  
                                GRB.MINIMIZE)
```

1.4.7 Model optimization

Optimize problem using simplex method.

```
[18]: mod.params.Method = 0
      mod.params.TimeLimit=3000
```

```
Changed value of parameter Method to 0
  Prev: -1  Min: -1  Max: 5  Default: -1
Changed value of parameter TimeLimit to 3000.0
  Prev: inf  Min: 0.0  Max: inf  Default: inf
```

```
[19]: mod.optimize()
```

```
Gurobi Optimizer version 9.1.2 build v9.1.2rc0 (linux64)
Thread count: 6 physical cores, 12 logical processors, using up to 12 threads
Optimize a model with 13402 rows, 10965 columns and 72009 nonzeros
Model fingerprint: 0x962c0781
Model has 10212 quadratic constraints
Model has 3 general constraints
Variable types: 0 continuous, 10965 integer (10584 binary)
Coefficient statistics:
  Matrix range      [1e+00, 3e+04]
  QMatrix range     [1e+00, 2e+03]
  QLMatrix range    [1e+00, 2e+04]
  Objective range   [1e+00, 1e+03]
  Bounds range      [1e+00, 1e+00]
  RHS range         [1e+00, 3e+04]
  QRHS range        [4e+03, 4e+03]
Presolve removed 10494 rows and 5370 columns
Presolve time: 0.20s
Presolved: 47269 rows, 25086 columns, 141996 nonzeros
Presolved model has 5289 SOS constraint(s)
Variable types: 0 continuous, 25086 integer (10023 binary)

Root relaxation: objective 2.239344e+05, 3274 iterations, 0.12 seconds
```

Nodes		Current Node			Objective Bounds			Work	
Expl	Unexpl	Obj	Depth	IntInf	Incumbent	BestBd	Gap	It/Node	Time
0	0	223934.410	0	730	-	223934.410	-	-	1s
0	0	224233.083	0	773	-	224233.083	-	-	2s
0	0	224242.979	0	686	-	224242.979	-	-	2s
0	0	224242.979	0	687	-	224242.979	-	-	2s
0	0	225072.775	0	622	-	225072.775	-	-	2s
0	0	225235.479	0	553	-	225235.479	-	-	2s
0	0	225235.479	0	556	-	225235.479	-	-	2s
0	0	225751.187	0	628	-	225751.187	-	-	2s
0	0	225760.743	0	600	-	225760.743	-	-	2s
0	0	226000.365	0	595	-	226000.365	-	-	3s
0	0	226167.398	0	623	-	226167.398	-	-	3s
0	0	226169.145	0	629	-	226169.145	-	-	3s

0	0	226307.094	0	629	-	226307.094	-	-	3s
0	0	226379.387	0	642	-	226379.387	-	-	3s
0	0	226382.989	0	605	-	226382.989	-	-	3s
0	0	226471.220	0	591	-	226471.220	-	-	3s
0	0	226490.407	0	654	-	226490.407	-	-	3s
0	0	226531.336	0	629	-	226531.336	-	-	3s
0	0	226547.694	0	702	-	226547.694	-	-	3s
0	0	226603.372	0	710	-	226603.372	-	-	3s
0	0	226720.070	0	645	-	226720.070	-	-	4s
4	1	226721.656	0	856	-	226721.656	-	0.0	5s
H	4	0			284160.00000	227846.729	19.8%	0.0	5s
H	4	0			282360.00000	227846.729	19.3%	0.0	5s
H	162	140			275017.00000	228563.374	16.9%	101	6s
H	266	181			267087.00000	228579.152	14.4%	76.3	7s
H	280	199			267081.00000	228579.152	14.4%	73.8	7s
H	318	209			267042.00000	228579.152	14.4%	68.5	7s
4792	922	237241.505	25	234	267042.000	232581.000	12.9%	32.9	10s
5084	1003	237442.862	32	285	267042.000	232712.579	12.9%	34.3	15s
5351	1025	infeasible	41		267042.000	232712.579	12.9%	35.1	20s
8087	506	244751.276	41	93	267042.000	233120.940	12.7%	34.4	25s
9221	613	236628.310	42	247	267042.000	235328.322	11.9%	33.7	30s
13555	310	infeasible	40		267042.000	249364.000	6.62%	30.6	35s

Cutting planes:

Learned: 4
 Gomory: 2
 Cover: 13
 Implied bound: 327
 Projected implied bound: 10
 Clique: 6
 MIR: 33
 StrongCG: 1
 Flow cover: 53
 RLT: 37
 Relax-and-lift: 58
 BQP: 2

Explored 15469 nodes (453114 simplex iterations) in 35.71 seconds

Thread count was 12 (of 12 available processors)

Solution count 6: 267042 267081 267087 ... 284160

Optimal solution found (tolerance 1.00e-04)

Best objective 2.670420000000e+05, best bound 2.670420000000e+05, gap 0.0000%

1.5 Printing results

First i'll extract agents trips with relative data and order those trips in a domino like form.

```
[20]: def domino(trip):
    """ Function to calculate a domino sorted trip from a random one.
        :param trip: List of trips between nodes,
        :return: Domino list of visitated nodes,
                Ordered list of visitated nodes.
    """
    sorted_trip = list()
    sorted_visit = list()
    # Set dest as fitticious start position
    dest = start_pos
    # If trip not empty
    while len(trip):
        # For each position couple ([start, end]) in trip
        for t in trip:
            # If start position is dest
            if t[0] == dest:
                # If positions in couple aren't fitticious start position
                # append couple replacing fitticious offices with real one
                if t[0] != start_pos and t[1] != start_pos:
                    sorted_trip.append((t[0] if t[0] < CLIENTS else 0,
                                       t[1] if t[1] < CLIENTS else 0))
                    # Update destination
                    dest = t[1]
                    break
            # If fitticious end is reached stop loop oterwhise append latest
            # position reached
            if dest == start_pos:
                break
        else:
            sorted_visit.append(dest if dest < CLIENTS else 0)
    return sorted_trip, sorted_visit

# Initialize main lists
domino_list = list()
agent_visit_list = list()

# For each agent
for a in agent_list:
    agent_trip = list()
    # For each position
    for i in all_pos:
        # For each other position
        for j in all_pos:
            # If agent do trip between those store trip
            if x[i,j,a].X:
                agent_trip.append((i,j))
    # Sort trip locations and make a domino list
```

```
domino_agent, agent_visit = domino(agent_trip)

# update main lists
domino_list.append(domino_agent)
agent_visit_list.append(agent_visit)
```

1.5.1 Print sorted agents trips

```
[21]: # For each agent trip
for a, avl in enumerate(agent_visit_list):
    print(f"Agent {a+1} trip: {avl}")
```

```
Agent 1 trip: [0, 8, 11, 2, 20, 10, 15, 4, 1]
Agent 2 trip: [14, 7, 6, 17, 3, 9, 18, 0]
Agent 3 trip: [0, 13, 12, 19, 5, 0, 16]
```

1.5.2 Print trips stats

Print all stats regarding agents trips in an ordered way.

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

stats = list()
# For each agent visit list
for a, avl in enumerate(agent_visit_list):
    agent_trip_desc = list()
    # For each visited position
    for i, v in enumerate(avl):
        prev = avl[i-1] if i else start_pos
        # If previous is a fictitious office
        if prev == 0 and i-2 > 0:
            prev = avl[i-2]+CLIENTS

        # If fictitious office
        v_mod = prev+CLIENTS if not v and i-1 >= 0 else v

    # Get stats
    pos_desc = {"POSITION": v if v else 'OFFICE',
                "COST TO REACH": distance_matrix[avl[i-1]][v] if i else 0.,
                "REACHED AT": s[v_mod,a].X,
                "SERVICE TIME": data_dict[v]['SERVICE'],
                "REAL S. TIME": c[v_mod, a].X,
                "LEAVING TIME": s[v_mod, a].X+c[v_mod, a].X,
                "LUNCH_BEFORE": l[prev, v_mod, a].X,
                "WAIT_BEFORE": w[v_mod, a].X,
                "TW_START": data_dict[v]['READYTIME'],
                "TW_END": data_dict[v]['DUE DATE']}
    stats.append(pos_desc)
```

```
agent_trip_desc.append(pos_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	REAL S. TIME	LEAVING TIME
	LUNCH_BEFORE	WAIT_BEFORE	TW_START	TW_END		
0	OFFICE	0.0	-0.0	3600.0	3600.0	3600.0
0.0		-0.0	0.0	28800.0		
1	8	364.0	3964.0	2400.0	2400.0	6364.0
0.0		-0.0	3600.0	4600.0		
2	11	82.0	6446.0	2400.0	2400.0	8846.0
0.0		-0.0	6000.0	7000.0		
3	2	269.0	9115.0	2400.0	2400.0	11515.0
0.0		-0.0	8400.0	9400.0		
4	20	914.0	12429.0	2400.0	2400.0	14829.0
0.0		0.0	12300.0	13300.0		
5	10	228.0	15057.0	2400.0	2400.0	17457.0
0.0		0.0	15000.0	16000.0		
6	15	652.0	19909.0	2400.0	2400.0	22309.0
1.0		0.0	19800.0	20800.0		
7	4	41.0	22350.0	2400.0	2400.0	24750.0
0.0		-0.0	22000.0	23000.0		
8	1	144.0	24894.0	2400.0	2400.0	27294.0
0.0		0.0	24800.0	25800.0		

Agent 2

	POSITION	COST TO REACH	REACHED AT	SERVICE TIME	REAL S. TIME	LEAVING TIME
	LUNCH_BEFORE	WAIT_BEFORE	TW_START	TW_END		
0	14	0.0	-0.0	2400.0	2400.0	2400.0
0.0		-0.0	0.0	1000.0		
1	7	700.0	3100.0	2400.0	2400.0	5500.0
0.0		-0.0	3100.0	4100.0		
2	6	171.0	5671.0	2400.0	2400.0	8071.0
0.0		0.0	5500.0	6500.0		
3	17	324.0	8395.0	2400.0	2400.0	10795.0
0.0		-0.0	8000.0	9000.0		
4	3	258.0	11053.0	2400.0	2400.0	13453.0
0.0		-0.0	10600.0	11600.0		
5	9	94.0	13547.0	2400.0	2400.0	15947.0
0.0		-0.0	13000.0	14000.0		
6	18	144.0	16091.0	2400.0	2400.0	18491.0
0.0		0.0	15300.0	16300.0		
7	OFFICE	984.0	21275.0	3600.0	3600.0	24875.0
1.0		-0.0	0.0	28800.0		

Agent 3

	POSITION	COST TO REACH	REACHED AT	SERVICE TIME	REAL S. TIME	LEAVING TIME
--	----------	---------------	------------	--------------	--------------	--------------

LUNCH_BEFORE	WAIT_BEFORE	TW_START	TW_END			
0 OFFICE	0.0	-0.0	3600.0	2383.0	2383.0	
0.0	-0.0	0.0	28800.0			
1 13	731.0	3114.0	2400.0	2400.0	5514.0	
0.0	-0.0	2400.0	4300.0			
2 12	186.0	5700.0	2400.0	2400.0	8100.0	
0.0	-0.0	5700.0	6700.0			
3 19	447.0	8547.0	2400.0	2400.0	10947.0	
0.0	-0.0	8500.0	9500.0			
4 5	361.0	11308.0	2400.0	2400.0	13708.0	
0.0	-0.0	10900.0	11900.0			
5 OFFICE	1338.0	15046.0	3600.0	2203.0	17249.0	
0.0	-0.0	0.0	28800.0			
6 16	551.0	19600.0	2400.0	2400.0	22000.0	
1.0	0.0	19600.0	20600.0			

1.5.3 Plot the trips graph

```
[23]: def plot_model(list_of_trips, positions, limit=math.inf):
    """ Function used to plot trips graph.
        :param list_of_trips: List of edges,
        :param positions: Nodes coordinates,
        :param limit: Limit number of nodes,
    """

    # Create figure
    plt.figure(figsize=(16,9))
    G = nx.Graph()
    # Add nodes to graph
    G.add_nodes_from({pk for pk in positions.keys() if pk < limit})

    # Set labels dict
    node_labels = dict()
    # Set nodes positions
    for n, p in positions.items():
        if n < limit:
            G.nodes[n]['pos'] = p
            node_labels[n] = n

    # Define edges colors
    edge_colors = [plt.cm.tab20.colors[i] for i in range(len(list_of_trips))]

    # Define edges labels list
    edge_labels = dict()

    # Add edges
    for i,trip in enumerate(list_of_trips):
        for j,t in enumerate(trip):
            G.add_edge(t[0], t[1],
                       color=edge_colors[i],
                       alpha=0.5,
                       weight=4)
            edge_labels.update({t: f"{j} (LUNCH)" if l[t[0], t[1],i].X else j})

    # Set edges parameters
    edges = G.edges()
    colors = [G[u][v]['color'] for u,v in edges]
    weights = [G[u][v]['weight'] for u,v in edges]
    alphas = [G[u][v]['alpha'] for u,v in edges]

    # Draw Nodes
    nx.draw_networkx_nodes(G, positions,
                           node_size=200,
                           node_color='c')

    # Draw Labels
    nx.draw_networkx_labels(G, positions,
                            node_labels,
```

```
        font_size=10,  
        font_color='k')  
  
# Draw Edges  
nx.draw_networkx_edges(G, positions,  
                        alpha=0.5,  
                        width=weights,  
                        edge_color=colors)  
  
# Draw Edges Labels  
nx.draw_networkx_edge_labels(G, positions,  
                             edge_labels=edge_labels)  
  
# Show graph  
plt.show()
```

```
plot_model(domino_list, positions, limit=CLIENTS)
```

