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 simplier 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 travel_cost* $\sum_{i\in ALL}\sum_{j\in ALL}\sum_{a\in A} x_{i,j,a} * distance_{i,j} \; + \; wait\_cost * \sum_{i\in C}\sum_{a\in A} w_{i,a} \; + \; \sum_{i\in RBLE}\sum_{a\in A} s_{i,a}$

1) $\sum_{i\in ALL}\sum_{a\in A} x_{i,j,a} = 1\,, \quad j\in C$
2) $\sum_{i\in ALL}\sum_{a\in A} x_{i,j,a} \le 1\,, \quad j\in FO$
3) $\sum_{j\in REAL} x_{start,j,a} = 1\,, \quad a\in A$
4) $x_{start,j,a} = 0\,, \quad j\in FO,\; a\in A$
5) $\sum_{i\in RBLE} x_{i,s,a} = 1\,, \quad a\in A$
6) $\sum_{i\in ALL} x_{i,h,a} - \sum_{j\in ALL} x_{h,j,a} = 0\,, \quad h\in ALL,\; a\in A$
7) $x_{i,i,a} = 0\,, \quad i\in ALL,\; a\in A$
8) $c_{j,a} = service_j * \sum_{i\in ALL} x_{i,j,a}\,, \quad i\in C,\; a\in A$
9) $\sum_{i\in ALL}\sum_{j\in o+FO} c_{j,a} * x_{i,j,a} >= service_o\,, \quad a\in A$
10) $c_{j,a} \ge x_{i,j,a}\,, \quad i\in ALL,\; j\in RBLE,\; a\in A$
11) $\sum_{i\in ALL}\sum_{i\in o+FO} x_{i,j,a} \ge 1\,, \quad a\in A$
12) $x_{i,j,a} = 0\,, \quad i\in o+FO,\; j\in o+FO,\; a\in A$
13) $x_{i,o,a} = 0\,, \quad i\in C,\; a\in A$
14) $x_{j,j+clients,a} \le \sum_{i\in ALL} x_{i,j,a}\,, \quad j\in C,\; a\in A$
15) $\{x_{i,j,a} = 0 \;\; if\; i \ne j - clients\}\,, \quad i\in REAL,\; j\in FO,\; a\in A$
16) $window\_start_i * \sum_{h\in ALL} x_{h,i,a} \le s_{i,a}\,, \quad i\in C,\; a\in A$
17) $window\_end_i * \sum_{h\in ALL} x_{h,i,a} \ge s_{i,a}\,, \quad i\in C,\; a\in A$
18) $x_{i,j,a} * (s_{i,a} + c_{i,a} + distance_{i,j} + w_{j,a}) \ge l_{i,j,a} * lunch\_start\,,$
$i\in RBLE,\; j\in RBLE,\; a\in A$
19) $l_{i,j,a} * (s_{i,a} + c_{i,a} + distance_{i,j} + w_{j,a}) \le x_{i,j,a} * lunch\_end\,,$
$i\in RBLE,\; j\in RBLE,\; a\in A$
20) $l_{i,j,a} \le x_{i,j,a} * lunch\_end\,, \quad i\in RBLE,\; j\in RBLE,\; a\in A$
21) $t_a = max(s_{i,a},\; i\in RBLE)\,, \quad a\in A$
22) $work\_end * \sum_{i\in RBLE}\sum_{j\in RBLE} l_{i,j,a} \ge t_a - lunch\_start\,, \quad a\in A$
23) $work\_end * \sum_{i\in RBLE}\sum_{j\in RBLE} (1 - l_{i,j,a}) \ge lunch\_start - t_a,\;\; a\in A$
24) $s_{j,a} = \sum_{i\in RBLE,\; i!=j} x_{i,j,a} * (s_{i,a} + c_{i,a} +$
$distance_{i,j} + l_{i,j,a} * lunch\_len) + w_{j,a}\,, \quad j\in RBLE,\; a\in A$
25) $\sum_{i\in RBLE}\sum_{j\in RBLE} x_{i,j,a} * distance_{i,j} \; +$
$\sum_{i\in ALL}\sum_{j\in RBLE} x_{i,j,a} * service_j \; +$
$\sum_{i\in RBLE} w_{i,a} \; +$
$\sum_{i\in RBLE}\sum_{j\in RBLE} l_{i,j,a} * lunch\_len \; +$
$work\_start \le work\_end\,, \quad a\in A$

$x_{i,j,a} \in \{0,1\}\,, i\in ALL,\; j\in ALL,\; a\in A$
$s_{i,a} \in \mathbb{N}\,, i\in ALL,\; a\in A$
$c_{i,a} \in \mathbb{N}\,, i\in ALL,\; a\in A$
$w_{i,a} \in \mathbb{N}\,, i\in ALL,\; a\in A$
$l_{i,j,a} \in \{0,1\}\,, i\in ALL,\; j\in ALL,\; a\in A$
$t_a \in \mathbb{N}\,, a\in A$

$start = clients + 1$
$o = 0$
$C = \{1, ..., clients\}$
$ALL = \{0, ...2clients + 1\}$
$REAL = \{0, ..., clients\}$
$RBLE = ALL - \{s\}$
$FO = \{P + 2, ..., 2clients + 1\}$

1.3 Model implementation

1.3.1 Importing libraries

[1]:
```python
from gurobipy import*

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

1.3.2 Define file with clients data

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

1.3.3 Define costant parameters

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

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

```python
    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]:
```python
# 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 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
data_dict.update({CLIENTS: {'X': 0,
                            'Y': 0,
                            'DEMAND': AGENTS,
                            'READYTIME': WORKING_TIME_RANGE[0],
                            'DUEDATE': 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")
```

```
--------------------------------------------
Warning: your license will expire in 11 days
--------------------------------------------
```

```
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 fitticious start and end trip loop location, position 0 is office too, Agents can only start from office 0 they can't go there.

[9]:
```python
#  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 contraints

[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 fitticious 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 fitticious 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 fitticious 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((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] +
                              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]) <= x[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 funtion want 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: 0xf9d39e64
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 10518 rows and 5370 columns
Presolve time: 0.32s
Presolved: 41245 rows, 27423 columns, 135609 nonzeros
Presolved model has 5289 SOS constraint(s)
Variable types: 0 continuous, 27423 integer (10023 binary)

Root relaxation: objective 2.237344e+05, 3051 iterations, 0.17 seconds

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

     0     0 223734.410    0  591          - 223734.410      -     -    2s
     0     0 223966.911    0  668          - 223966.911      -     -    3s
     0     0 224634.710    0  435          - 224634.710      -     -    3s
     0     0 224664.326    0  625          - 224664.326      -     -    3s
     0     0 224664.326    0  628          - 224664.326      -     -    4s
     0     0 225601.972    0  723          - 225601.972      -     -    4s
     0     0 225682.500    0  647          - 225682.500      -     -    4s
     0     0 225686.526    0  635          - 225686.526      -     -    4s
     0     0 225890.724    0  753          - 225890.724      -     -    5s
     0     0 225992.140    0  712          - 225992.140      -     -    5s
     0     0 226010.909    0  727          - 226010.909      -     -    5s
     0     0 226114.390    0  717          - 226114.390      -     -    5s
```

```
     0     0 226114.390    0  696          - 226114.390      -     -    5s
     0     0 226259.373    0  736          - 226259.373      -     -    5s
     0     0 226279.348    0  674          - 226279.348      -     -    5s
     0     0 226354.044    0  763          - 226354.044      -     -    5s
     0     0 226407.768    0  677          - 226407.768      -     -    6s
H    4     0                    276992.00000 226817.496  18.1%   0.0    9s
H    4     0                    273973.00000 227674.285  16.9%   0.0    9s
     4     0 228157.905    0  436 273973.000 228157.905  16.7%   0.0   10s
H    4     0                    273283.00000 228188.201  16.5%   0.0   10s
H   42    41                    273220.00000 228256.325  16.5%   247   10s
H   70    71                    273160.00000 228256.325  16.4%   166   11s
H   82    82                    272980.00000 228256.325  16.4%   154   11s
H   85    82                    272674.00000 228256.325  16.3%   152   11s
H  287   170                    270220.00000 228296.374  15.5%  76.6   12s
   967   433 235537.392   23  324 270220.000 228923.919  15.3%  59.1   15s
* 1094   441              37    268621.00000 229144.784  14.7%  56.8   15s
* 1931   665              37    267208.00000 229847.641  14.0%  51.3   16s
  3910  1103 252960.078   27  902 267208.000 230821.713  13.6%  43.9   20s
H 3919  1053                    266842.00000 230821.713  13.5%  43.8   21s
  3941  1073 230821.713   26  395 266842.000 230821.713  13.5%  45.2   25s
  4176  1105 infeasible   37       266842.000 230821.713  13.5%  46.0   30s
  4397  1072 245802.823   38  209 266842.000 230821.713  13.5%  45.6   35s
  5099   982     cutoff   33       266842.000 230821.713  13.5%  45.1   40s
  5819   820 237684.588   44   69 266842.000 230934.000  13.5%  44.0   45s
  6501   659 231909.938   36   95 266842.000 231909.938  13.1%  42.8   51s
  7849   540 236500.000   36    3 266842.000 233868.832  12.4%  40.8   56s
  9239   582 237035.829   39  389 266842.000 236473.150  11.4%  39.7   61s
 10018   628 257314.763   35  104 266842.000 237430.183  11.0%  39.4   65s
 12918   488     cutoff   48       266842.000 248800.867  6.76%  35.6   71s

Cutting planes:
  Learned: 3
  Cover: 12
  Implied bound: 252
  Projected implied bound: 15
  Clique: 11
  MIR: 35
  StrongCG: 1
  Flow cover: 33
  Inf proof: 1
  RLT: 22
  Relax-and-lift: 54

Explored 17271 nodes (539415 simplex iterations) in 74.42 seconds
Thread count was 12 (of 12 available processors)

Solution count 10: 266842 267208 268621 ... 273973

Optimal solution found (tolerance 1.00e-04)
Best objective 2.668420000000e+05, best bound 2.668420000000e+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]:
```python
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: [0, 13, 12, 19, 5, 0, 16]
Agent 3 trip: [14, 7, 6, 17, 3, 9, 18, 0]
```

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 = dict()
# 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 fitticious office
        if prev == 0 and i-2 > 0:
            prev = avl[i-2]+CLIENTS

        # If fitticious 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]['DUEDATE']
                   }
```

```python
        agent_trip_desc.append(pos_desc)

    # Store path stats
    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  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   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        2003.0       17049.0
0.0          -0.0        0.0  28800.0
6       16          551.0     19400.0        2400.0        2400.0       21800.0
1.0          -0.0    19400.0  20400.0

Agent 3
```

```
  POSITION  COST TO REACH  REACHED AT  SERVICE TIME  REAL S. TIME  LEAVING TIME
LUNCH_BEFORE  WAIT_BEFORE  TW_START   TW_END
0        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
```

[23]:

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