FOR14 – Exam, Autumn 2023

Candidates: 63, 75

The purpose of the algorithm is to maximize the amount of bookings that can be placed for a car by arranging the bookings in the most efficient way. Previously, an algorithm was provided to solve this. Nevertheless, a new one has been written to solve the new problems and enhance the code.

To enhance the code, the ask is as follows: Revise the system to enable more dynamic and adaptable categorization of cars into subsets, thereby improving scalability. Provide deeper analytical insights, particularly through detailed metrics like idle time and unusable time. This enhanced analysis will aid in optimizing fleet utilization and efficiency. Furthermore, there is a strong emphasis on transparency, especially regarding the reshuffling algorithm. The company requires clear tracking of each reservation's original and new car assignments post-reshuffling. Additionally, they desire a preview feature to visualize potential reshuffling outcomes before implementation, ensuring informed decision-making and stakeholder approval. Thus, the focus is on creating a system that is not only scalable and capable of detailed analysis, but also operates with complete transparency in its scheduling and reservation management processes.

In order to develop the solution, the following assumptions have been made:

1. Bookings must have a minimum gap of 30 minutes.

After a vehicle's return by a customer, the company needs to inspect it for any issues and accommodate potential delays in vehicle return. Consequently, a safety margin between bookings is essential. We have designated a 30-minute interval between bookings as this safety margin when doing the reshuffling, as the company considers this interval between bookings to be wasted time anyway.

2. The aim of the reshuffling is to have customer satisfaction unchanged while minimizing costs.

It's stated that vehicles with the same number of seats provide equivalent customer satisfaction. Therefore, we will give preference to assigning customers to less expensive cars within the same seat category. This is under the assumption that the category_id says something about how expensive a car is. We noticed in the CSV file that lower category_id values often are paired with less exclusive cars. This approach aims to keep the same customer constant and minimize costs.

However, this strategy assumes that all cars yield similar profit margins, a point which could be contested. Ideally, one would optimize based on profit by considering factors like profit ratios, price elasticity of demand, overall demand, and operational costs to determine the most profitable car allocation. But such an optimization is beyond this project's scope, as we lack the necessary data.

To exemplify this, for prioritization within the 5-seat car category, we will order them by ascending category_id: 1, 3, 5.

Candidates: 63, 75

3. Number of seats is a better measure for interchangeability than the car category.

As this is given in the text, an evaluation of interchangeability should be based on number of seats, not on car category.

Now that the assumptions have been outlined, we can move on to the description of classes, before the logic behind the algorithm is justified in the end.

Description of classes:

Schedule:

- Data description:
    - EarliestStart (start on first booking in the fleet), latestEnd (end on last booking in the fleet), trips
- Public interface:
    - Initialize reshuffled trips, calculating productive time of each car, calculating unusable time of each car, calculating idle time of each car, calculating wasted time of each car, calculating utilization of each car, adding reservation to schedule, checking if the car can accommodate booking, reshuffles the bookings, provision of report to evaluate effectiveness of reshuffling

Car:

- Data description:
    - Car_id, model_id, location_id, car_number, icon_url, model_row
- Public interface:
    - List all reservations of the car

Fleet:

- Data description:
    - List of cars you want to reorganize
- Public interface:
    - Add car to list of cars, find cars with same amount of seats

Reservation:

- Data description:
    - trip_id, driven_km, start_ts, ends_ts, car_id, duration_hours (duration of reservation in hours), seats, category_id, category_name
- Public interface:
    - None

Candidates: 63, 75

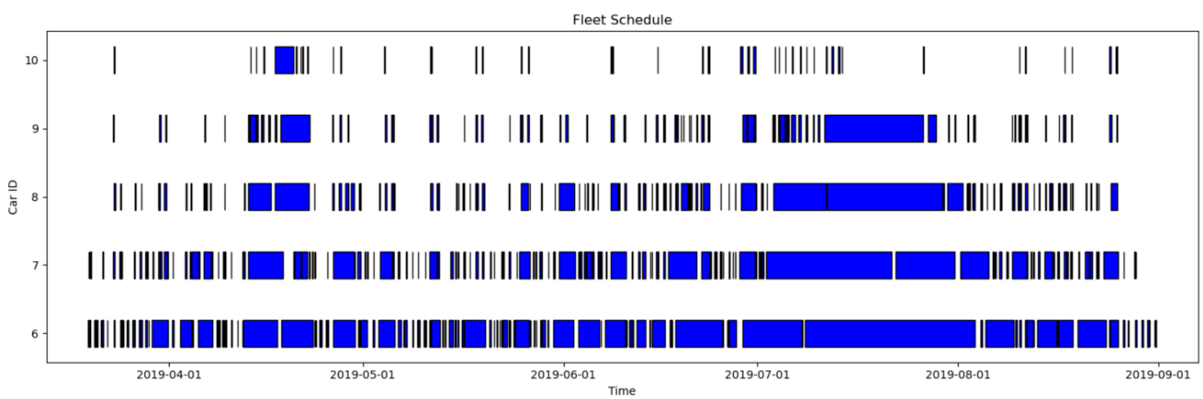
The following versions of python libraries have been used:

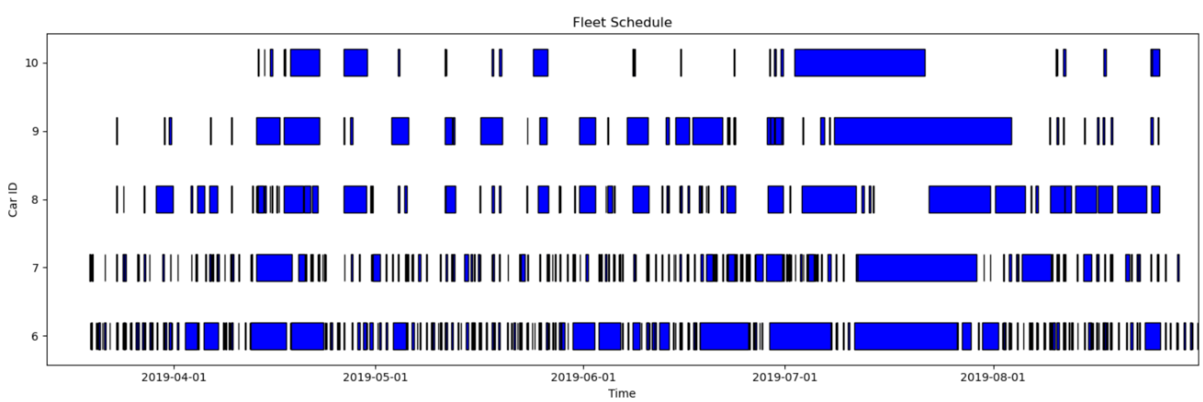
numpy==1.24.3
matplotlib==3.7.2
pandas==2.1.1

Justification of algorithm:

The algorithm we have developed sorts the bookings by duration, and then places the longest bookings first. This is because the longer bookings give more productive time and minimize idle time. Longer bookings are also harder to move between the cars schedules, and thy should therefore be reshuffled first. If one does not prioritize placing the longest bookings first, shorter bookings may crowd out longer bookings, which is sub-optimal in terms of car utilization.

To illustrate this, we have provided two figures based on example data earlier provided. The first one shows the booking schedule when longer bookings are prioritized, and the second figure shows the schedule when long bookings are not prioritized.





Through these figures, one sees that prioritization of longer reservations leads to better utilization, which in turn justifies the logic behind the algorithm. The provided algorithm also makes sure to only reshuffle booking between cars with the correct number of seats.

Candidates: 63, 75

Result after reshuffling:

After the implementation of the algorithm, the following metrics are reported:

- Number of leftover bookings: 0
- Reduction of utilized cars: 23 cars
- Reduction of utilized cars as percentage: 12.6%
- Increase in fleet utilization standard deviation: 2.0x

This was achieved using a dataset of 5000 bookings, which indicates that similar results can be expected if the algorithm is further implemented.

Limitations to our implementation:

However, there are certain limitations to our algorithm that need to be acknowledged.

Firstly, within the schedule class, it's worth noting that the earliest start and latest end times are not calculated dynamically. Nevertheless, this is not a significant concern as we do not intend to add new bookings dynamically.

Secondly, there is also a limitation in the method used to calculate unusable time within the schedule class. The current approach does not account for any unusable time occurring before the first booking or after the last booking. Nonetheless, it's important to recognize that this limitation is expected to have a minimal impact on the overall calculations, making it a manageable issue.

<u>Pseudocode:</u>

To explain the logic behind the code, we have provided the following pseudocode:

Pseudocode for reshuffling method:

```
input: bookings of all cars in the fleet
output: optimized organization of bookings in the fleet
Sort bookings from longest to shortest duration
Group the sorted bookings based on seats of the car they are booked in
for reservation ∈ {grouped and sorted reservations} do
    applicable_cars = cars that can accommodate reservation (same number of seats)
    for car ∈ {applicable_cars} do
        if booking booking fits with a 30 minute gap from the bookings before and after then
            allocate booking to car
        end if
    end for
    if booking does not fit any schedule … then
        mark the booking as a leftover booking
    end if
end for
```

Pseudocode for checking if the car has available space for a new booking

```
input: cars' schedule, a new reservation and the intended pause between each booking
output: a check if the booking in question fits in the car's schedule
Assume that car has available space for booking
for booking ∈ {all bookings of the car} do
    If one of the following is true,
        1. adjusted_start is between the start and the end the booking
        2. adjusted_end is between the start and the end the booking
        3. adjusted_start is before the start of the booking AND adjusted_end is after
           the end of the booking
    then
        declare that booking does not fit
    end if
end for
continue declaring that booking fits
```

Pseudocode for adding reservation

input: a new reservation, specified schedule to add the reservation to (the actual schedule or the reshuffling schedule)
output: allocation of reservations
if the chosen schedule is the standard schedule **then**
    add the new reservation to the standard schedule
else
    if the chosen schedule is the reshuffling schedule **then**
        add the new reservation to the reshuffling schedule
    end if
end if

Pseudocode for calculating productive time:

input: a car's schedule and the chosen schedule
output: calculation of a car's productive time
productive time = 0
add the length of every booking in a car to *productive time*

Pseudocode for calculating unusable time:

input: a car's schedule and the chosen schedule
output: calculation of a car's unusable time
unusable time = 0
for booking ∈ {car} **do**
    time clearance = time between this booking and the next
    add the lowest of the *time clearance* and 30 minutes to *unusable time*
end for

Pseudocode for calculating idle time:

input: a car's schedule and the chosen schedule
output: calculation of a car's idle time
Find total time between end of first booking and start of last booking
Subtract productive time and unusable time from the total time

Pseudocode for calculating wasted time:

input: a car's schedule and the chosen schedule
output: calculation of a car's wasted time
wasted time = 0
for *booking* ∈ {all bookings of the car} **do**
    if *duration of booking* < 0.5 hours

Candidates: 63, 75

add *duration of booking* to *wasted time*
end if
end for

ChatGPT usage:

Generative AI has played a helpful role in assisting with the coding tasks throughout the examination. To ensure optimal transparency, we will first outline the structure of how the AI has been utilized, followed by providing an illustrative example, and concluding with a comprehensive list of all the sections where generative AI has been equivalently employed. This approach has been selected to avoid an exhaustive presentation of each individual query, which would be impractical. We believe that the outlined method offers the most comprehensive and transparent overview.

Outline of how the examination has been answered:

First, we begin by considering the specific requirements of our task and independently formulate an idea of what the code should achieve, including the overarching logic of each necessary part of the code.

Next, using our understanding of the code's intended purpose, and logic, we engage the generative AI to explore how it would implement only the logic that we initially did not know how to implement.

Finally, once the AI provides its output, we optimize it, taking into account the specific context in which the code will be used. This optimization process involves making manual adjustments ourselves.

Thus, generative AI has been used in the second step, and the code is written by us. Here are examples of how it has been done in practice:

Example 1:

You

python for key and value in dict

ChatGPT

In Python, to iterate over key-value pairs in a dictionary, you can use a loop with the `.items()` method of the dictionary. Here's an example with a sample dictionary:

python Copy code

```python
sample_dict = {"key1": "value1", "key2": "value2", "key3": "value3"}

for key, value in sample_dict.items():
    print(f"Key: {key}, Value: {value}")
```

This code will output:

mathematica Copy code

```
Key: key1, Value: value1
Key: key2, Value: value2
Key: key3, Value: value3
```

Each iteration of the loop retrieves a key-value pair from the dictionary.

Candidates: 63, 75

This is then how it looked in the code:

```python
# returns a subset of cars with the same nubmer of seats. Input: x=numbe
def get_cars_by_seats(self, x, sort_by=None):
    # List to hold cars with x number of seats
    cars_with_x_seats = []

    # Iterate through all cars in the fleet
    for car_id, car in self.cars.items():
        # Check if the car has x seats
        if hasattr(car, 'seats') and car.seats == x:
            cars_with_x_seats.append(car)

    # Sort the list if a sort_by property is provided
    if sort_by is not None:
        cars_with_x_seats.sort(key=lambda car: getattr(car, sort_by))

    return cars_with_x_seats
```

Example 2:

You

python if is nan

ChatGPT

In Python, you can check if a value is `NaN` (Not a Number) using the `math.isnan()` function or by comparing it with `numpy.nan` if you're using NumPy. Here's how you can do it in both ways:

1. Using `math` module:

python Copy code
```python
import math

value = float('nan')  # Example NaN value
is_nan = math.isnan(value)
```

1. Using `numpy`:

python Copy code
```python
import numpy as np

value = np.nan  # Example NaN value
is_nan = np.isnan(value)
```

This is then how it looked in the code:

```python
# Loop through each row in the dataframe and create a Car object
for _, row in car_raw.iterrows():
    if not np.isnan(row['car_id']):
        car = Car(row['car_id'], row['model_id'], row['location_id'], row['car_number'], row['icon_url'])
        fleet.add_car(row['car_id'], car)
```

Candidates: 63, 75

Example 3:

You

i have a large dataframe, only keep the first 1000 rows

ChatGPT

Error analyzing

It seems I don't have access to your dataframe. To keep only the first 1000 rows of your dataframe, you can use the following code snippet in your Python environment:

python Copy code

```python
import pandas as pd

# Assuming your dataframe is named 'df'
# Replace 'df' with the actual name of your dataframe

# df = pd.read_csv('path_to_your_csv_file.csv') # Uncomment this line i

# Keep only the first 1000 rows
df_trimmed = df.head(1000)

# 'df_trimmed' now contains only the first 1000 rows of the original da
```

Make sure to replace `'df'` with the actual name of your dataframe. If you're loading the data from a file, uncomment and modify the `read_csv` line accordingly. [>_]

This is then how it looked in the code:

```python
# keep only the first 5 000 bookings for testing purposes
trips_raw = trips_raw.sort_values(by='start_ts')
trips_raw = trips_raw.head(5000)
```

Candidates: 63, 75

Example 4:

You

bookings = [<__main__.Reservation at 0x2a3fdb0fa10>,
<__main__.Reservation at 0x2a3fdb0cb10>,
<__main__.Reservation at 0x2a3fd708650>,
<__main__.Reservation at 0x2a3fdb0db10>,
<__main__.Reservation at 0x2a3fa733050>,
<__main__.Reservation at 0x2a3fdae1d90>,
<__main__.Reservation at 0x2a3fdbb8290>,
<__main__.Reservation at 0x2a3fdae3290>,
<__main__.Reservation at 0x2a3fdbb8d10>,
<__main__.Reservation at 0x2a3fdbb9450>,
<__main__.Reservation at 0x2a3fdb0d110>, ...]

Sort bookings by reservation.start_ts

Foreleseren

To sort the `bookings` list by the `start_ts` attribute of each `Reservation` object in Python, you can use the `sorted()` function or the `.sort()` method of the list. The key difference between these two approaches is that `sorted()` returns a new list, while `.sort()` modifies the list in place.

Here's how you can do it using `sorted()`:

```python
sorted_bookings = sorted(bookings, key=lambda reservation: reservation.
```

And here's how to do it using `.sort()`:

```python
bookings.sort(key=lambda reservation: reservation.start_ts)
```

In both cases, `lambda reservation: reservation.start_ts` is a function that takes a `Reservation` object and returns its `start_ts` attribute, which is used by `sorted()` or `.sort()` to determine the order of the objects in the list.

This is then how it looked in the code:

```python
#Returns the idle time both before and after bookings have been reshuffled
def calculate_idle_time(self, car, specified_schedule="trips"):
    car_id = car.car_id

    # bookings = the specified schedule
    if specified_schedule=="trips":
        bookings = self.trips[car_id]
    elif specified_schedule=="reshuffled_trips":
        bookings = self.reshuffled_trips[car_id]

    # sort bookings by start_ts to calculate the time between bookings
    bookings.sort(key=lambda reservation: reservation.start_ts)
```

Example 5:

Candidates: 63, 75

You

flattened_list.sort(key=lambda x: x.duration_hours)

sort this with the largest first

ChatGPT

To sort the `flattened_list` by the `duration_hours` attribute in descending order (i.e., with the largest values first), you can modify the `sort` method by setting the `reverse` parameter to `True`. Here's how you do it:

```python
flattened_list.sort(key=lambda x: x.duration_hours, reverse=True)
```

This line of code will sort `flattened_list` based on the `duration_hours` attribute of its elements, with the elements having the highest `duration_hours` values appearing first in the list.

This is then how it looked in the code:

```python
#Reshuffles the bookings into a more optimal schedule
def reshuffle(self, minutes_pause=0):
    #################################################
    # sort by duration and group by number of seats #
    #################################################
    # Flatten the dictionary into one long list, and add any leftover reservations
    flattened_list = [reservation for sublist in self.trips.values() for reservation in sublist]

    # initialize the reshuffled schedule
    self.initialize_reshuffled_trips()

    # Sort the flattened list by duration_hours
    flattened_list.sort(key=lambda x: x.duration_hours, reverse=True)
```

<u>Usage of ChatGPT:</u>

ChatGPT has been used equivalently in other parts of the implementation. This includes:

- Grouping reservations buy number of seats
- Query:

Reservations = {3: [<__main__.Reservation at 0x257b461b4d0>, <__main__.Reservation at 0x257b45cf190>, <__main__.Reservation at 0x257b45c9310>, <__main__.Reservation at 0x257b45bdc50>, <__main__.Reservation at 0x257b45be6d0>], 4: [<__main__.Reservation at 0x257b4179290>, ...]

I have a dict of reservations of different cars. Group the list by reservation.seats.

Output:

```
grouped_reservations = {}
for reservation in flattened_list:
    if reservation.seats in grouped_reservations:
        grouped_reservations[reservation.seats].append(reser
    else:
        grouped_reservations[reservation.seats] = [reservati
```

Our code:

```
# Group the sorted list by reservation.seats
grouped_reservations = {} # {seats: [reservation]}
for reservation in flattened_list:
    if reservation.seats in grouped_reservations:
        grouped_reservations[reservation.seats].append(reservation)
    else:
        grouped_reservations[reservation.seats] = [reservation]
```

- Finally, it is important to note that much of our code is a variation of the code turned in for assignment 2 and 3. In these assignments we used ChatGPT the same way as shown in these examples, and we have manually improved the code in preparation for the exam.