Homework 3 - Order Batching Problem

Stefano Tonini*, Lorenzo Sciotto, Jacopo Dallafior

*Group representative

At least 1000 words per homework (without code)

1. Problem description

The Order Batching Problem is a logistics optimization challenge that involves efficiently grouping a set of orders for warehouse picking. The goal is to create batches that minimize the total travel distance for an order picker. In this specific scenario, we are given a set of orders and a matrix of locations within the warehouse for each item. The task is to compute order picking batches based on different seed rules and evaluate their performance using a specific accompanying rule.

Efficient order batching is crucial for optimizing warehouse operations, reducing picking time, and minimizing the travel distance for order selectors. This problem is of significant applicability in the field of logistics and force chain operation, where functional effectiveness directly impacts costs and client satisfaction. By optimizing the order picking process, companies can ameliorate overall storehouse productivity and meet client prospects more effectively.

2. Mathematical model

The problem is modeled as a clustering and optimization challenge. The seed rules represent different strategies for initiating the batch creation process, while the accompanying rule considers the spatial relationships between orders. The adaptation involves incorporating real-world constraints, such as the maximum capacity of each batch, to make the model applicable in practical scenarios.

The implemented solution uses a heuristic approach to address the Order Batching Problem. Instead of solving the mathematical model directly, which could be computationally expensive for large instances, a heuristic method was applied to obtain feasible solutions quickly.
The heuristic involves:
Seed Rule: Starting with a seed order for each batch (random order, largest order, or order with the furthest COG).

Accompany Rule 1 (Euclidean Distance): Adding orders to batches based on the similarity matrix and the Euclidean distance between COGs.

Accompany Rule 2 (Sum of Euclidean Distances): Introducing a new rule where orders are added to batches based on minimizing the sum of Euclidean distances between each location of the order to be added and the closest order line location of the seed.

This heuristic approach provides a pragmatic way to tackle the problem, especially for real-world scenarios where finding an optimal solution might be computationally infeasible.

In conclusion, the applied model is a heuristic method that leverages seed rules and accompanying rules to efficiently generate batches for the Order Batching Problem. This approach strikes a balance between solution quality and computational efficiency, making it suitable for practical applications. The effectiveness of the heuristic can be further evaluated through performance metrics and visualizations, as demonstrated in the next sections.

3. Main code components

To make a well-structured code in the beginning every library used in the code with their respective abbreviation must be imported (‘numpy’, ‘seaborn’, ‘matplotlib.pyplot’, ’tabulate’, ‘math’ and ‘pandas’).
In order to solve this problem CPLEX solver was not used because the approach was heuristic.
Secondly the input csv file must be imported and correctly stored using the library ‘pandas’. The two csv files, named “wh_coords.csv” and “orders.csv”, contained all the necessary information to define the coordinates and the orders for the mathematical model. Considering the former, it was a csv file made of 360 rows, each of them for one storage location and two columns containing the “x” and “y” coordinates.
Regarding the latter file, it was about the 15 customer orders and the storage location for each of them.

```
wh_coords = pd.read_csv("./wh_coords.csv")

lines = open('./orders.csv', 'r').readlines()
orders = [list(map(float, line.strip().split(','))) for line in lines]

X = wh_coords['x'].values
Y = wh_coords['y'].values
```

In addition, after importing the data and the libraries some important functions used in the code were defined, more precisely:

1. COG Function:

The Center of Gravity function calculates the COG for a given order based on the locations of its items within the warehouse. This function is fundamental in determining the starting point for each batch.

```
def COG(order):
    x = 0
    y = 0
    for line in order:
        index = int(line) - 1
        x += X[index]
        y += Y[index]
    return [x / len(order), y / len(order)]
```

This function takes an order as input and calculates its Center of Gravity based on the locations of its items within the warehouse.

2. Euclidean Distance Function:

The Euclidean distance function computes the distance between two points in the warehouse. It is used to measure the spatial separation between the COGs of different orders, influencing the accompanying rule.

```
def euclid_dist(x1, y1, x2, y2):
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
```

The Euclidean distance function computes the straight-line distance between two points (x1, y1) and (x2, y2) in the warehouse.

3. Batch Creation Algorithm:

The main code includes an algorithm that creates batches according to the specified seed rules and accompanying rules. It iteratively forms batches, considering the maximum capacity constraint and the spatial relationships between orders.

To implement the first accompany rule, euclidean distance between orders, the following function was coded:

```
def most_similar(i, indexces):    # i is the last oder added to the batch
    min_sim = np.inf    # Start assigning a huge value for the mean similarity
    J_sim = -1
    for j in range(len(orders)):
        simil = simil_matr[i,j]
        if simil < min_sim and i!=j and j in indexces:
            min_sim = simil
            J_sim = j
    return J_sim
```

This function used the similar matrix that was coded previously with these lines of code:

```
simil_matr = np.zeros([len(orders),len(orders)])

for i in range(len(orders)):
   xi,yi = COG(orders[i])
   for j in range(len(orders)):
       if i != j:
           xj,yj = COG(orders[j])
           simil_matr[i,j] = euclid_dist(xi,yi,xj,yj)
```

Furthermore in order to implement the second accompany rule two functions were coded

```
def total_euclidean_distance(seed, order2):
   total_dist = 0
   for loc1 in order2:
       min_dist = float('inf')
       for loc2 in seed:
           loc1 = int(loc1)
           loc2 = int(loc2)
           dist = euclid_dist(X[loc1-1], Y[loc1-1], X[loc2-1], Y[loc2-1])
           if dist < min_dist:
               min_dist = dist
       total_dist += min_dist
   return total_dist
```

```
def most_close(i, indexces):   # i is the last oder added to the batch
   min_sim = np.inf   # Start assigning a huge value for the mean similarity
   J_sim = -1
   for j in range(len(orders)):
       simil = total_euclidean_distance(orders[i],orders[j])
       if simil < min_sim and i!=j and j in indexces:
           min_sim = simil
           J_sim = j
   return J_sim
```

The algorithm iterates through different seed rules, calculates the accompanying rule, and forms batches accordingly. It ensures that the batches adhere to the specified constraints and optimization criteria.

For both the accompany rules three different seeds were used and afterwards were evaluated using the distances between order lines in each batch, assuming that the order

picker starts and ends each batch at the depot and visits order lines following their location number in ascending order.

The first seed was Random order, basically the first order is chosen randomly

```
# Create the batches
np.random.seed(0)

rndm = np.random.randint(0,len(orders))

batches = [[orders[rndm]]]
capacity = len(orders[rndm])
```

The second seed was the largest order

```
def biggest_order():
   l = []
   dist = []
   for i in range(0,len(orders)):
       l.append(len(orders[i]))
       xi,yi = COG(orders[i])
       dist.append(euclid_dist(0,0,xi,yi))
   return l,dist

l,dist = biggest_order()

max_O = np.max(l)

min = np.inf
j = -1
for i in range(0,len(orders)):
   if l[i] == max_O:
       if(dist[i]<min):
           min = dist[i]
           j = i

# Create the batches
largest = j

batches = [[orders[largest]]]
capacity = len(orders[largest])
```

The last one was the Order with furthest COG from depot (located in (0, 0)) that was coded

```
def furthest_COG():
   l = []
```

```
    dist = []
    for i in range(0,len(orders)):
        l.append(len(orders[i]))
        xi,yi = COG(orders[i])
        dist.append(euclid_dist(0,0,xi,yi))
    return l,dist

l,dist = furthest_COG()

max_O = np.max(l)

min = 0
j = -1
for i in range(0,len(orders)):
    if(dist[i]>min):
            min = dist[i]
            j = i
# Create the batches
furthest = j

batches = [[orders[furthest]]]
capacity = len(orders[furthest])
```

For every single seed and for both accompany rules the performance was computed using this two functions

```
def calculate_distance_between_points(point1, point2):
    # Assuming points are in the format [x, y]
    return np.sqrt((point2[0] - point1[0])**2 + (point2[1] - point1[1])**2)

def calculate_batch_distance(batch, wh_coords):
    total_distance = 0.0
    depot = [0.0, 0.0]  # Depot coordinates

    l = []

    for order in batch:
        for i in range(len(order)):
            l.append(order[i])
        # Sort order in ascending order
        sorted_order = sorted(l, key=lambda x: int(x))

        current_location = depot  # Start from the depot
    for line in sorted_order:
        index = int(line) - 1  # Convert float to integer and adjust for zero-based
indexing
```

```
        order_location = [wh_coords.iloc[index]['x'], wh_coords.iloc[index]['y']]
        total_distance += calculate_distance_between_points(current_location,
        order_location)
        current_location = order_location

    # Return to the depot after completing the batch
    total_distance += calculate_distance_between_points(current_location, depot)
    return total_distance
```

In addition for visualize the orders assigned to every batches this command was used

```
tot_performance = 0
# Calculate performance for each batch
for i, batch in enumerate(batches):
    performance = calculate_batch_distance(batch, wh_coords)
    print(f"Batch {i + 1} Performance: {performance}")
    tot_performance += performance

print(f"Total Performance: {tot_performance}")

for n, batch in enumerate(batches):
    print(f'Batch{n+1}:{batch}')
```

In conclusion for sake of visualization, using the 'matplotlib.pyplot' library, the order picker path for the best performance Accompany rule and seed rule configuration chosen from the six different configurations was plotted using these lines of code

```
def plot_orders_and_route(batches, wh_coords, depot=[0.0, 0.0]):
    for idx, batch in enumerate(batches):
        plt.figure(figsize=(10, 6))

        # Plot warehouse locations
        plt.scatter(wh_coords['x'], wh_coords['y'], marker='o', color='blue',
label='Locations')
        plt.scatter(depot[0], depot[1], marker='s', color='green', label='Depot')

        all_points = [point for order in batch for point in order]
        all_points = sorted(all_points)

        # Connect the depot to the first point of the first order
        first_order_location = np.array([[wh_coords.iloc[int(line) - 1]['x'],
wh_coords.iloc[int(line) - 1]['y']] for line in [all_points[0]]])
        plt.plot([depot[0], first_order_location[0, 0]], [depot[1],
first_order_location[0, 1]], linestyle='-', color='black', label='Route from
Depot')
```

```
        # Connect all points in ascending order
        order_location = np.array([[wh_coords.iloc[int(line) - 1]['x'],
wh_coords.iloc[int(line) - 1]['y']] for line in all_points])
        plt.plot(order_location[:, 0], order_location[:, 1], marker='o',
linestyle='-', label=f'Batch {idx + 1}')

        # Annotate order locations
        for i, txt in enumerate(all_points):
            plt.annotate(txt, (order_location[i, 0], order_location[i, 1]),
textcoords="offset points", xytext=(0, 5), ha='center')

        # Connect the last point of the last order to the depot
        last_order_location = np.array([[wh_coords.iloc[int(line) - 1]['x'],
wh_coords.iloc[int(line) - 1]['y']] for line in [all_points[-1]]])
        plt.plot(last_order_location[:, 0], last_order_location[:, 1],
linestyle='-', color='black', label='Route to Depot')
        plt.plot([last_order_location[-1, 0], depot[0]], [last_order_location[-1,
1], depot[1]], linestyle='-', color='black')

        # Annotate depot
        plt.annotate('Depot', (depot[0], depot[1]), textcoords="offset points",
xytext=(0, 5), ha='center')

        plt.xlabel('X-coordinate')
        plt.ylabel('Y-coordinate')
        plt.title(f'Order Routes - Batch {idx + 1}')
        plt.legend()
        plt.grid(True)

        plt.tight_layout()
        plt.show()

# Visualize each batch
plot_orders_and_route(batches, wh_coords)
```

4. Results and insights

The Order Batching Problem addresses a critical aspect of warehouse logistics, aiming to optimize order picking processes. By employing mathematical models and adapting them to real-world constraints, the presented solution provides valuable insights and practical applications. The code components, functions, and results collectively contribute to a

comprehensive understanding of the problem and its potential impact on operational efficiency.

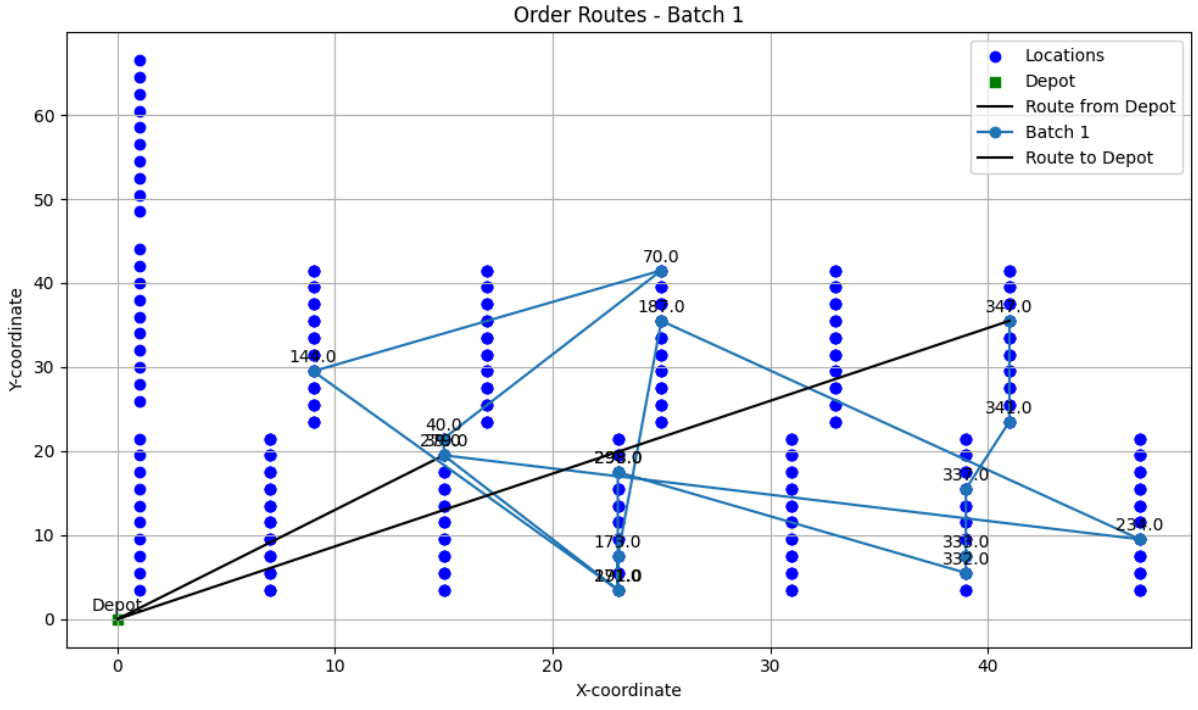
Importing python library ‘tabulate’ the performance results can be displayed for every combination of accompany rule and seed in a table format:

Accompany Rule	Seed rule	Total Performance
Euclidean distance	Random order	757.7764024608532
Euclidean distance	Largest order	818.9199400444885
Euclidean distance	Order with furthest COG from depot (0,0)	785.7718404333513
Sum of travel distances	Random order	896.7786913970706
Sum of travel distances	Largest order	865.8808140971784
Sum of travel distances	Order with furthest COG from depot (0,0)	896.7786913970706

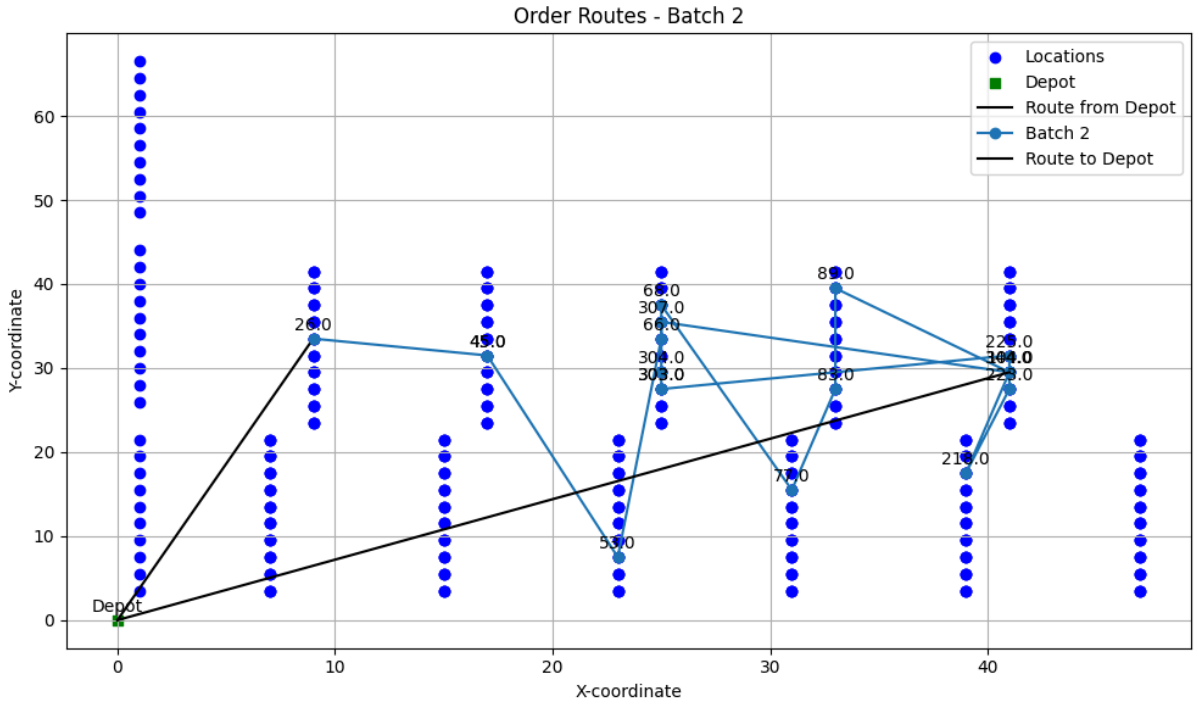
It is also interesting to point out that, as reported in the previous table, the Euclidean distance accompany rule outperforms the sum of travel distances accompany rule in all seed rules.
The above statement cannot be generalized to all cases but is only specific to the case under study and the data related to the imported CSV files.
In conclusion, for visualizing the solution, it was possible to plot the best performance solution path of the order picker that corresponds to Euclidean distance as Accompany Rule and Random order for the seed rule giving a Total Performance equal to 757.7764024608532.

With three batches composed as stated after:

Batch 1: [[332.0], [291.0, 234.0, 40.0], [171.0, 298.0], [298.0], [341.0, 39.0], [144.0, 333.0, 70.0, 337.0], [173.0, 187.0, 347.0, 279.0]]
With the following order picker path



Batch 2: [[77.0, 53.0, 223.0, 68.0], [307.0, 304.0, 45.0, 26.0, 218.0, 344.0], [83.0, 45.0], [66.0, 303.0], [303.0, 89.0], [225.0, 104.0]]



Batch 3:[[264.0, 19.0, 314.0, 290.0, 34.0], [22.0, 4.0]]