# Project: Order batching and robot routing in (half-)automated warehouses

Analysing Networks with OR-Methods

**Nele-Pauline Gaedke**, lg081947@stud.leuphana.de, 3041452

**Andrey Perov**, lg082043@stud.leuphana.de, 3041544

**Anas Suffo**, lg081936@stud.leuphana.de, 3041440

Major: Management & Data Science

Examinator: Prof. Dr. Lin Xie


Leuphana University Lüneburg

Lüneburg, Deutschland

30.08.2021

## Contents



## Acronyms

**AGV** automated guided vehicles

**SKU** Stock Keeping Unit

## 1. Introduction

The presented project deals with the important topic of order batching and robot routing in warehouses assisted by automated guided vehicles. Upcoming, the given problem is described and the goal defined in Sections 1.1 and 1.2, respectively. In Section 2, concept and implementation are discussed, including the explanation of the used heuristics. Afterwards, Section 3 presents the computational results, followed by Section 4 summarizing this documentation and giving an outlook on future research opportunities.

### *1.1. Problem description*

The given problem describes an automated guided vehicles (AGV)-assisted warehouse system. An AGV-supported picking system is characterized through human pickers and transportation robots working together to fulfill customer's orders. Each order consists of several items, a so called Stock Keeping Unit (SKU). The SKUs are stored in shelves which itself are organized in different storage areas. The distances between shelves and packing stations are known. In the problem considered in this project, two packing stations with two cobots and two different kind of layouts are given:

- Layout 24: In total 24 shelves are organized in two storage zones with 12 shelves each.
- Layout 360: In total 360 shelves are organized in six storage zones with 60 shelves each.

Each storage area has a human picker. Those human pickers only move within their storage area from shelf to shelf. When a cobot, the transport robot, enters the area, the human pickers goes to the shelf the desired item is stored in, and loads the item from the shelf into the bin carried by the cobot. The orders with details about the items and their quantities as well as the weights for all items are known in advance. In the same way as human pickers are assigned to a specific storage area, cobots are allocated to a certain packing station. As visible in Figure 1, two packing stations are considered in this problem. The cobot starts the tour from the packing station, goes to the necessary picking locations and returns back to the packing station to drop of the loaded items.

For solving the problem, several assumptions were made and are summarized in the following:

*Storage Policy.* Two different storage policies can be defined. The dedicated storage policy describes the situation that all items of a SKU are located in one shelf only. In contrast, the mixed storage policy allows items of one SKU to be stored in different shelves. The first will be the storage policy used for the main part of this project, while the latter will be taken into account only briefly in section 2.3.

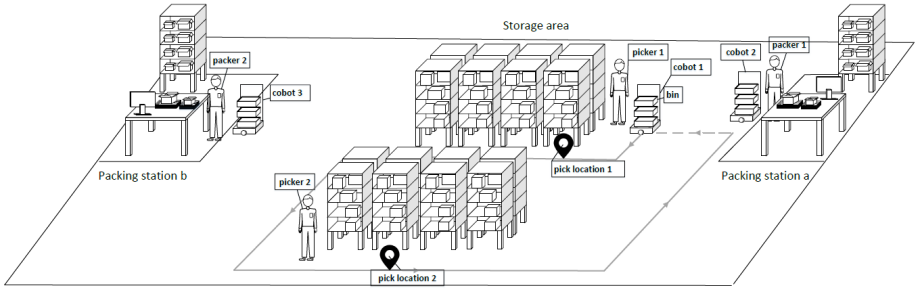


Figure 1: Example of a (half-)automated picking system with two packing stations

*Zoom.* Each storage area, called zoom or zone, contains one human picker. When a cobot arrives at a picking location in this area, it has to wait for the human picker to arrive. For calculating the waiting time, the walking speed of the human pickers is given as 1.3 m/s. The picking time is defined as 3 seconds per single item.

*Waiting Policy.* The human picker in each zoom follows the first come first serve policy, i.e. they pick all items for the cobot entering their zoom first and continue with the next cobot entering their packing area. As long as the human picker is still working on another cobots items, the other cobot has to wait in another zone.

*Cobot Assignment.* Each cobot is assigned to a packing station where it is charged. The tour starts and ends at this packing station. We assume that a cobots average moving speed is 2.0 m/s and that the battery will last long enough to finish a tour.

*Order Fulfillment.* If an order is assigned to a tour, all items of this order have to be picked in this tour. The order of picking the items can be independent of the item order insight the order and can be mixed with other items from additional orders in the same tour. We assume that the number of items available per picking location is always enough to satisfied the needed amounts.

*Cobot Loading Capacity.* A cobots loading capacity for carrying the items is limited to 18kg per tour, also called batch. The items are stored in bins, while we assume that there is no limit in the number of bin carried by the cobot.

*Packing Station.* When the cobot finished collecting all items, it returns to the assigned packing station were the human picker unloads all items. The assumed unload time is 20 seconds. The following step of packing the orders is time-wise defined as 60 seconds. Before the cobot leaves the packing zone for the next tour, new empty bins are put in the cobot. This preparation time is defined with 30 seconds. The preparation time is only necessary for the first leave of each cobot.

*Packing Station Workload.* Additionally, we assume similarly to [1], that the time needed to finish the last order will be minimized when the orders are evenly distributed among the stations. Therefore, we will try to balance the order assignment to the packing stations.

### *1.2. Goal*

The goal of the described problem is to minimize the makespan. The makespan is defined as the time needed until the last order is packed by the human picker. It includes the successful picking and packing of all orders defined in this problem. As mentioned in [1], minimizing the picking time can often be equated with minimizing the distances. For this reasons minimizing the distance is the focus for getting an initial solution (section 2.1), while the makespan is taken into account for the neighborhood design in section 2.2. In order to solve the presented problem, decisions have to be made about:

(1) Assigning orders to the stations (depots).
(2) Assigning orders to cobot tour (batches).
(3) Defining the picking sequence for each cobot in each tour.

In the following the defined sets, parameters and variables are listed, including the most important additional variables used in the pseudo code.

**Sets**

- $O$ - set of orders
- $V$ - set of nodes
- $V^D$ - set of depots
- $V^S$ - set of shelves
- $C$ - set of cobots for item picking
- $B$ - set of batches for cobot $c \in C$
- $P$ - Set of items for picking
- $Z$ - Set of picking zones

**Parameters**

- $c$ - maximum cobot capacity
- $d_{i,j}$ - distance from item i to item j, $\forall i, j \in V$
- $w_p$ - weight of item $p \in P$
- $w_o$ - weight of order $o \in O$
- $w_{v^D}$ - weight of orders assigend to depot
- $s_c$ - cobot moving speed
- $s_p$ - picker moving speed
- $t_{picking}$ - time needed to load one item from shelf into the cobot
- $t_{preparation}$ - time to prepare a cobot for the next batch
- $t_{unload}$ - time to unload cobot after finishing a batch
- $t_{packing}$ - time to pack an order

**Decision Variables**

- $a_{o,v^D} = \begin{cases} 1, & \text{if order } o \in O \text{ is assigned to depot } v^D \in V^D \\ 0, & \text{otherwise} \end{cases}$

**Variables used in Pseudocode**

- $\hat{D}^o$ - set of distances from order to all depots $\forall o \in O$
- $D_{i,j}$ - distances from order i to order j, $\forall i, j \in O$
- $B^*$ - overall minimum number of batches required to carry out all orders
- $b_{nr}$ - batch number

- $w_{b_i}$ - total weight for batch $b_i \in B$
- $d_{v^D,b}$ - total batch distance
- $path_{c_i}$ - 'flattened' path for each cobot $c \in C$ containing the routes for all batches in one
- $z_{p,c_1}$ - zones for items picked by cobot $c \in C$
- $z_{c_i}$ - cobot zone sequence
- $n_{p,v_i}$ - item count for number of items picked in shelf $v^S \in V^S$

## 2. Concept and Implementation

In the following chapter, the different steps for implementing a solution for the given problem are explained. First, the greedy heuristic designed to create an initial solution is described in 2.1. Afterwards the simulated annealing algorithm (2.2.1) and the implemented iterated local search (2.2.2) are explained in Subsection 2.2. Section 2.3 gives an overview over the approach used for the mixed shelf policy.

### *2.1. Heuristic Design*

In order to get the initial solution for the given problem, different algorithms were defined to work on the three decision, mentioned in 1.2.

*Orders to Depots.* The first algorithm Greedy Assignment of Orders to Depots takes all orders and the distances for each order item to both depots as an input. In order to minimize the makespan (=time), we want to ensure, that the number of orders assigned to both depots is balanced. For this approach, the dictionary including the distances is sorted so that the depot with the shortest distance is in first position (line 1). In the next step, the algorithm checks whether the order can be assigned to more than one depot. If there is only one option, the order will be assigned to this depot (line 10). If more than one depot is available, the balancing condition checks for orders with higher priority that should be assigned to certain depots first. The prioritization is based on $\mu$, which is calculated as the difference between the distance to the closest and second closest depot (line 2). In the next step, the orders will be reordered based on $\mu$, starting with the order corresponding to the highest value of $\mu$. Before assigned the orders to the preferred depot, the balancing constrain is

checked (line 3). If the assignment is possible, the order gets assigned to the depot, if not, the depot will be removed from the list of possible depots and the algorithms runs again until all orders are assigned (line 6).

**Algorithm 1** Greedy Assignment of Orders to Depots

**Data:** $O, \hat{D}^o, w_{v^D}, w_o, c, B^*$

**Result:** Assignment of all orders $o \in O$ to the depots $V^D \in V$

1 **Initialization:** $a_{o,v^D} = 0$, sort $\hat{D}^o$ in ascending order `//closest depot first`

**if** *length* $\hat{D}^o > 1$ `//more than one depot available` **then**

2 $\mu = \hat{D}^o[1]$ - $\hat{D}^o[0]$ $\forall o \in O$`//balancing`

sort O according to $\mu$ in descending order

**for** $o \in O$ **do**

3 **if** $(\frac{w_{v^D}+w_o}{c}) \leq (\frac{|B^*|}{S})$ **then**

4 $a_{o,v^D} = 1$ `//assign order to depot`

$w_{v^D} = w_{v^D} + w_o$, $o \leftarrow O \setminus o$ `//add order weight to depot`

5 **else**

6 $\hat{D}^o \leftarrow \hat{D}^o \setminus \hat{d_{o,i}}$

OrderToDepots$(O, \hat{D}^o, w_{v^D}, w_o, c, B^*)$

7 **end if**

8 **end for**

9 **else**

10 $a_{o,v^D} = 1; w_{v^D} \leftarrow w_{v^D} + w_o; O \leftarrow O \setminus o$ `//assign order to depot`

11 **end if**

*Orders to Batches.* After assigning orders to the two depots in a balanced way, the next step is to assign orders to batches, cobot tours, for both depots. For assigning orders to batches, the algorithm Order Assignment to Batches is applied for each depot, as seen in algorithm Greedy Assignment of Orders to Batches.

Depot-wise the algorithm starts with assigning orders to the first batch by looping over the list of orders assigned to this depot. As long as the total weight of the new order added to the batch weight is not higher than the cobot capacity, the order is added to the batch. When adding the first order to a batch, the order is chosen randomly. For the second order, the function NewOrderOfBatch is executed, which is comparing the minimal distances from all items in the

**Algorithm 2** Greedy Assignment of Orders to Batches

**Data:** $V^D, O$

**Result:** Assignment of all orders $o \in O$ in a depot to a batch $b_i \in B$

1 **for** $v^D \in V^D$ **do**

2 $\quad b_i \leftarrow AssignOrdersToBatches(v^D, O)$

3 **end for**

orders already assigned to the order with all items of the unassigned orders (line 3).

**Algorithm 3** Order Assignment to Batches

**Data:** $O, D, w_o, D^o_{i,j}, c$

**Result:** Assignment of all orders $o \in O$ to a batch $b_i \in B$

1 **Function: AssignOrdersToBatches**

**Initialization:** $b_{nr} = 1$, $b_i = [\,]$

**while** $O_i \neq \emptyset$ `//iterate till all orders are assigned` **do**

2 $\quad w_{b_i} = 0, O_{temp} = [\,]$

**while** $any((w_{b_i} + w_o) \leq c)$ **do**

3 $\quad\quad o_{new} = NewOrderOfBatch(O_i, O_{temp}, D^o_{i,j}[i])$ `// closest order`

**if** $w_{b_i} + w_{o_{new}} \leq cs$ **then**

4 $\quad\quad\quad w_{b_i} \leftarrow w_{b_i} + w_{o_{new}}, O_{temp} \leftarrow O_{temp} \cup o_{new}, O_i \leftarrow O_i \setminus o_{new}$

5 $\quad\quad$ **else**

6 $\quad\quad\quad o_{new} = NewOrderOfBatch(O_i, O_{temp}, D^o_{i,j})[i+1]$

7 $\quad\quad$ **end if**

8 $\quad\quad b_{nr} \leftarrow b_{nr} + 1, i \leftarrow i + 1$

9 $\quad$ **end while**

10 **end while**

The next order that will be checked for batch assignment, is the one with the total minimum distance to the orders already assigned to the current batch. If adding the new order is feasible without violating the capacity constraint, the order is added to the current batch, the batch weight and the batch list are updated while the assigned order is removed from the list of unassigned orders (line 4). If the order cannot be added to the batch due to the capacity constraint,

another one is picked while keeping the constraint. If no further order can be added to a batch due to the capacity constraint, the batch is closed and new orders are assigned to the new batch. The algorithm stops when all orders are assigned to batches.

*Greedy Route.* The final part for the initial solution includes finding the best route for each defined batch (algorithm 4). For the algorithm initialization all shelves not yet visited are stored in list remaining. As the orders include item ID, color and letter for each item in this order, the corresponding shelf for this item is determined and stored. The depot is set as the start location, the distance is set to 0. While looping through all shelves in a batch, the next shelf visited is found in a greedy way with the algorithm Find Greedy Route. The distance and the tour will be updated and the algorithm continues checking for the next shelf closest to the currently visited one. The procedure stops when all shelves /items in the batch are picked and the cobot returned to the packing station.

**Algorithm 4** Create Greedy Route for Batches

**Data:** $B$, $V^D$, $O_i$

**Result:** Determine greedy route for each batch $b_i \in B$ for depots $v^D \in V^D$

1 **Initialization**: $path_{v^D,b} = [\,], d_{v^D,b} = [\,]$

**for** $v^D$ $inV^D$ `//one shelf corresponds to one item` **do**

2 | **for** $b_i \in B$ **do**

3 | | $path_{v^D,b}, d_{v_D,b_i} = FindBestRoute(b_i, v^D)$

4 | **end for**

5 | $d_{v_D,b} = \sum d_{v_D,b_i}$

6 **end for**

Looking into the algorithm FindBestRoute 5 in more detail, the next shelf to be visited is determined in a greedy way. For this, the distances from the current shelf to all other possible next shelves is calculated (line 3). The result is sorted and the shelf closest to the current one, is selected. The tour is updated and the now visited shelf is removed from the list of not visited shelves (line 5). After generating the route and the distance for the shelf sequence, the return to the packing station is added (line 7)

**Algorithm 5** Find Greedy Route

**Data:** $B, V^D, O_i$

**Result:** Determine greedy route

**Function: FindBestRoute**

**Initialization:** $p_{v^S}$ = shelf IDs corresponding to batch items, head = depot, path = [head], total batch distance $d_{v_D,b_i} = 0$

**while** *length* $(p_{v^S}) > 0$ **do**

  **for** $s^D \in p_{v^S}$ **do**

    $d = \text{distance}(head, v^S)$ `// distance from last to current shelf`

  **end for**

  $p_{v^S,best}, d_{best} = sorted(\text{d}[0])$`// select shortest path`

  $p_{v^S} \leftarrow \text{p}_{v^S} \setminus p_{v^S,best}$

  head $= p_{v^S,best}$, path $\leftarrow path \cup d_{best}, d_{v_D,b_i} \leftarrow d_{v_D,b_i} + d_{best}$

**end while**

$d_{v_D,b_i} \leftarrow d_{v_D,b_i} \cup distance(head, depot)$ `// return distance to depot`

$path \leftarrow path \cup depot$

*2.2. Neighborhood Design*

*Makespan Calculation.* While the focus in section 2.1 was on finding an initial solution for the given problem, the second part discusses algorithms for solution optimization. For this purpose, a simulated annealing algorithm and an iterated local search were implemented. As already mentioned, the goal of this problem is to minimize the makespan. For the greedy initial solution in 2.1 only the distance was taken into consideration when creating batches and tours. For optimizing this solution, the makespan has to be calculated and then used as an input for the simulated annealing procedure. For calculating the makespan, the following information are given:

- cobot speed $s_c = 2m/s$
- picker speed $s_p = 1.3m/s$
- picking time/item $t_{picking} = 3s$
- preparation time/batch $t_{prep} = 30s$
- unload time/batch $t_{unload} = 20s$
- packing time/order $t_{packing} = 60s$

The logic behind calculating the makespan is that only one human picker is available per packing zone. That means, when a cobot wants to enter a picking zone which already is occupied by another cobot, the cobot has to wait until the human picker is available again. In order to check the picker availability, we determine the zone of the shelf for the next item in the path is in. As the two cobots work in parallel, another preparation step is needed. Working in parallel means that one cobot already can start picking items of the second or third batch, while the first cobot still is working on the first order. The item picking sequence therefore is required to be one long tour sequence so that the cobots can continue working on the next batch immediately after finishing the previous batch.

As already mentioned, checking the zone availability for the next shelf the item is stored in is a required step for the *first come, first serve* policy. For this reason, a new route of zone ID's instead of the shelf ID's has to be generated so that each cobot has one route sequence. This sequence contains all zone ID's visited in order to pick all order items assigned to the specific packing station. Next to the waiting and moving time, the makespan also consists of the time, the human picker needs to pick the items of the shelf. Therefore, a dictionary containing the amount of all items needed to be picked from the different shelves are generated as one of the input data for calculating the makespan.

**Algorithm 6** Calculate Cobot Time in a Zone

**Data:** $t_{start}, Z, C, path_{c_i}, z_{p,c_i}, z_{c_i}, s_{picker}, s_{cobot}, t_{picking}, d_{i,j} \forall i, j \in V, n_{p,v_i}$

**Result:** Calculate time of a cobot in a zone

1 delete $(path_{c_i}[0], z_{p,c_i}[0])$ `// delete previous waypoints`

$v^S_{last}, t_{serving} = \text{PickerServing}(Z, C, path_{c_i}, z_{p,c_i}, s_{picker}, t_{picking}, d_{i,j}, n_{p,v_i})$

$t_{finish} = t_{start} + t_{serving}$

$t_{occupation} = (t_{start}, t_{finish})$

delete $z_{c_i}$

For calculating the time, a cobot spends in a zone, the described and prepared data are used as the input data for algorithm 6. The idea is that the cobot enters the packing zone at a specific time $t_{start}$. The cobot then moves to the specific shelf and the picker loads in the items. As the human picker is slower as the cobot, we assume that the time needed to move from one shelf

to another is depended on the human walking speed only, as the cobot has to wait for the human to arrive. This picking time is calculated in the algorithm Calculate Human Picker Serving Time. When the items are picked, the cobot leaves the zone and this time is saved as time, the specific zone is blocked for other cobots to enter the same zone.

The function *PickerServing* (7) calculates the time needed for the human picker to arrive at the shelf and to load in the number of items (line 2). The total serving time is then generated as the adding those two values to the already calculated time in this zone. The corresponding waypoints and the item counts are deleted and the algorithm repeated until all shelves in one zone are visited.

**Algorithm 7** Calculate Human Picker Serving Time

**Data:** $Z, C, path_{c_i}, z_{p,c_i}, s_{picker}, s_{cobot}, t_{picking}, d_{i,j} \forall i, j \in V, n_{p,v_i}$

**Result:** Calculate serving time of a cobot in a zone

1 **Function: PickerServing**

$t_{picking} = t_{picking} * n_{p,v_{i0}}$, $t_{serving} = t_{picking}$, delete $n_{p,v_{i0}}$

**while** $z_{p,c_1} = zone$ `// when zone is blocked` **do**

2 $t_{picker,travel} = distance(path_{c_i}[0], path_{c_i}[1])/s_{picker}$ `// travel time`

$t_{picking,total} = t_{picking} * n_{p,v_{i0}}$

$t_{serving} = t_{serving} + t_{picker,travel} + t_{picking,total}$

delete $z_{p,c_0}, n_{p,v_{i0}}, path_{c_i}[0]$

3 **end while**

4 $id_{last} = path_{c_i}[0]$

As seen in 2.1, the initial solution output is a total distance and a picking sequence for each cobot. This initial solution is used as the input for calculating the makespan. As calculating the makespan involves different scenarios due to the cobot interaction, the algorithm is nested, the most important parts are visible in 8. For the initialization step, the start and finish times for the cobot tour are set to the time of the end of the preparation for the tour, the occupation time for each zone set to 0 to be available. Because a cobot can already finish a new batch before the packer at a packing station finished packing an order, the occupation time for the depots also have to be tracked with $t_{occupation,v^D}$. In the implemented algorithm cobots and packing stations (depot) are used

interchangeably as one cobot is uniquely assigned to one packing station. As described before, each cobot has a zone sequence containing all shelves that have to be visited to fulfill all orders. The algorithm iterates over all the zones and selects the next pod that needs to be visited. If both cobots have the same zone as the next zone, the cobot who can arrive earlier goes there, given time when cobots finished their previous steps and travel time to the zone (line 6). The time, the cobot needs to get to the zone is calculated as the travel time (distance over speed), adding waiting time in case the cobot has to wait for the human picker. The second cobot goes to the blocked zone after first one. If the first cobot can not finish while second cobot travels to the zone, the waiting time is calculated (line 7) so that the cobots exactly arrives in the zone, when the picker is free and moved to the first location of the second cobot for the particular zone. When the cobots work in different zones, the time for traveling (line 11), picking as well as packing (line 12) and waiting for the human picker (line 15) are calculate. If the cobot finishes his tour before packing is finished it needs to wait before going to the packing station too. In addition to the makespan, the algorithm also gives the waiting time as a separate output for the input path sequence.

**Algorithm 8** Calculate Makespan

**Data:** $s_{init}$

**Result:** Calculate makespan as the time until last order is packed

**Initialization**: $t_{start} = 0, t_{finish} = 0, t_{occupation,z} = (t_{start}, t_{finish})$,
$t_{startpack} = 0, t_{finishpack} = 0, t_{occupation,v^D} = (t_{startpack}, t_{finishpack})$
**while** *any*$(z_{ci} \neq \emptyset)$// visit all zones & depots **do**
  $z_{next} = [\ ]$
  **for** *cobot* $c \in C$ **do**
    $z_{next} \leftarrow z_{next} \cup (z_{ci}[0])$
  **end for**
  **if** $z_{next,c_i} = z_{next,c_j}$ //cobots want to visit same zone **then**
    $c_{first}, c_{second} = CompareDistances(c_i, c_j)$ //decide cobot order
    **for** *cobot* $c \in C$ **do**
      $t_{start} = t_{current,c_i} + t_{travel}(+t_{waiting})$
      $t_{waiting,c_{second}} = t_{finish} + t_{picker,travel} - t_{travel} - t_{current,c_{first}}$
    **end for**
  **else**
    **for** *cobot* $c \in C$ **do**
      $t_{travel} = distances(path_{ci}[0], path_{ci}[1])$
      **if** $z_{next} = cobot$ **then**
        $calculate(t_{current}, t_{packstart}, t_{packfinish}, t_{waiting})$
      **else**
        **if** $pos_{picker} \neq 0$ **then**
          $z_{picker,travel} = distances(pos_{picker}, path_{ci}[1])$
        **else**
          $t_{start} = t_{current,c_i} + t_{travel}(+t_{waiting})$//if cobot waits for picker
        **end if**
      **end if**
    **end for**
  **end if**
**end while**

*2.2.1. Simulated Annealing*

The implemented simulated annealing algorithm is inspired by [2]. For the Simulated Annealing Algorithm several values are initialized. To control the algorithms performance, the initial temperature, the minimal temperature and the maximum number of iterations are set (line 1). The cooling schedule is specified as a linear cooling schedule. Compared to a exponential or logarithmic schedule, the linear decrease in temperature will allow the algorithm to explore a broader neighborhood, move further away from the initial solution and to reduce the risk of getting stuck in a local minima. For each step, the algorithm compares the new with the current energy. If the current energy is better, the neighbor gets accepted, if its worse, it get accepted with a probability based on a critical value (line 3). If the critical value is smaller, the neighbor gets accepted and will be checked for whether it provides a better solution than the current best one. If the energy is smaller, the current best solution will be updated (line 7), otherwise the algorithms checks the next neighbor. This process continues until either the maximum number of iterations or the minimum temperature is reached.

**Algorithm 9** Simulated Annealing Algorithm

**Data:** initial solution $s_{init} = makespan, step_{max}, t_{min}, t_{max}, \alpha = 0.9$
cooling schedule = $linear$,

**Result:** Find improved cobot item picking sequence

1 **Initialize**: $s_{current} = s_{init}, s_{best} = s_{current}, E_{current} = makespan,$
$E_{best} = E_{current}$, step $\leftarrow 1, accept \leftarrow 0, T = t_{max}$
**repeat**
2 $s_{prop}$, $o_{prop}$ = FindNeighborSolution($s_{current}$)
**if** $s_{prop} < s_{current}$ *or U(0,1)* $< \exp(\frac{-dE}{T})$ **then**
3 $E_{current} = E_{new}, s_{current} = s_{prop}, o_{current} = o_{prop}, step \leftarrow step + 1$
4 **end if**
5 **if** $s_{current} < s_{best}$ **then**
6 $s_{best} = s_{current}, E_{best} = E_{current}, accept \leftarrow accept + 1$
7 **end if**
8 $T \leftarrow \alpha T$
9 **until** $step \geq step_{max}$ ***or*** $T \leq t_{min}$ ***or*** $T \leq 0$;

The algorithm *FindNeighborSolution* generates a new neighbor for each iteration. This is done in two steps. First, the batches in one depot are broken up and the orders in randomly reassigned to new badges. Second, the tour within one batch again is created in a greedy way. After getting the greedy cobot route for each batch, two nodes within a tour are picked by random and exchanged to generate a new solution.

Having two randomizes parts in generating the neighborhoods increases the possibility of the neighborhood leading to an optimal solution when starting from any solution in the solution space. The major restriction for the neighborhood created for this problem is, that all neighbors have the identical order to depot assignment. The Greedy Assignment of Orders to Depots algorithm starts picking the first order that needs to be assigned in a random way and builds up the following orders to be assigned based on the first one. Generating neighbors also providing different solutions for this step, would lead to a more diverse neighborhood and would ensure the simulated annealing algorithm to lead to an optimal solution.

### *2.2.2. Iterated Local Search*

For the Local Search Algorithm, the initial solution is used as the input. The algorithm runs until the maximum number of iterations is reached. Again, a new solution is generated from the defined neighborhood (line 2). If the solution is better as the best one, the solution gets updated to be the best one.

**Algorithm 10** Local Search Algorithm

**Data:** initial solution $s_{init}$ = makespan, $iter_{max}$

**Result:** Find improved cobot item picking sequence

1 **Function: LocalSearch**

**Initialization**: $s_{current} = s_{init}, s_{best} = s_{current}, iter \leftarrow 0$

**repeat**

2 $s_{current} = FindLocalNeighbor(s_{current})$

**if** $s_{current} < s_{best}$ **then**

3 $s_{best} = s_{current}$ `//update best solution`

4 **end if**

5 **until** $iter \geq iter_{max}$;

In order to implement an iterated local search algorithm, this solution is then used as the input for the Iterated Local Search Algorithm. Here again a new solution is generated and compared to the current best one. This time, the new solution is picked using the function *PerturbationBatches* (line 2).

**Algorithm 11** Iterated Local Search Algorithm

**Data:** $s_{init}$ = solution from function *LocalSearch*

**Result:** Find improved cobot item picking sequence

1 **Initialization**: $s_{current} = s_{init}, s_{best} = s_{current}, iter \leftarrow 0$

**repeat**

2 $s_{current} = PerturbationBatches(s_{current})$

**if** $s_{current} < s_{best}$ **then**

3 $s_{best} = s_{current}$ `//update best solution`

4 **end if**

5 $iter \leftarrow iter + 1$

6 **until** $iter \geq iter_{max}$;

As visible in 12, the algorithms first reassigns orders to new batches. The original algorithm Order Assignment to Batches picks the first order randomly and the following one in a greedy way. For this reason, the order sequence is changed and new batches are generated while keeping the weight constraint.

In the second step, the algorithm randomly selects two nodes in the route sequence for each batch in each depot. Those two nodes will be swapped so that a new solution gets created.

Referring to task 2.2.3, the designed perturbation strategy for an iterated local search will work as intended. As the two nodes that will be swapped, are selected randomly, this strategy can lead to a big variety of different alternative solutions. Additionally also the assignment of orders to batches is part of the perturbation step. Limitations can be identified in scenarios, when changing two nodes or reassigning batches may lead to an infeasible solution. Repairing an infeasible solution again involves a new algorithm and is not covered in the scope of this project. In order to increase the degree of perturbation, two or more pairs could be identified for the swapping step instead of one pair.

**Algorithm 12** Perturbation Strategy

**Data:** initial solution $s_{init}$

**Result:** Find improved cobot item picking sequence

**Function: PerturbationBatches**

**for** $v^D \in V^D$ **do**

  **for** $o \in O$ **do**

    reassign $a_{o,v^D}$ `//create new batches`

  **end for**

  **for** $b_i \in B$ **do**

    $pos_1 = \text{random}(b_i)$

    $pos_2 = \text{random}(b_i)$

    $b_i = \text{SwapPositions}(b_i, pos_1, pos_2)$ `// swap two items in route`

  **end for**

**end for**

*2.3. Mixed Shelf Policy*

The difference between dedicated and mixed shelf policy is that one item can be stored in more than one shelf. For generating an initial solution, the algorithms 1, 2, 3, 4, 5 were used in a slightly adapted way. In the following, the most important changes will be described. The first adaptation was necessary when preparing the data for the algorithms. As we used dictionaries containing the corresponding shelf id for each item, the dictionary this time simply includes a list of corresponding shelf id's instead of one unique shelf only. When assigning orders to depots (algorithm 1), the same balancing constraint is used. For calculating the distances from orders to depots, the shelves minimizing the total distance to the depots are used. For the algorithm 3, 4 and 2 the same changes had to be made. Instead of picking the next item minimizing the distance, this time all possible shelf/item combinations are compared and the next item in a specific shelf minimizing the distance chosen.

## 3. Computational Results

As the goal of the problem is find a route sequence for each cobot that minimizes the makespan, the makespan solution for the initial solution and the

neighborhood approach are to be compared. As the task involves different order layouts and two storing policies, each sub task has a makespan output which can be found in the .xml files. The makespan for the initial greedy solution are given in the corresponding .xml files. The calculation is based on the greedy route algorithm implemented in 2.1. As this is the initial solution found in a greedy way, it is expected for the neighborhood heuristic to lead to better results. For the iterated local search algorithm, the used perturbation strategy in combination with the local search algorithm, the makespan is smaller than in the initial solution, resulting in a better solution for the problem. Compared to the already improved solution in the iterated local search algorithm, the simulated annealing approach even yields to a slightly better solution for most running scenarios. Depending on the number of iterations, the best result was achieved with $t_{max} = 1000$, $step_{max} = 1000$ and $\alpha = 0.8$ for most orders. Comparing the three results, it becomes clear, that by using the defined neighborhood, the initial solution could be improved.

## 4. Summary and Outlook

The goal for this project was to minimize the makespan which is defined as the time needed until the last order packing is finished. The initial solution was found in a greedy way, first assigning orders to depots, then orders in each depot to batches and afterwards generating greedy cobot tours for each batch. In the next task, a neighborhood was defined and used in simulated annealing and iterated local search. Both solutions yield to better makespan solutions than the initial one. Nevertheless rerunning the algorithms using a different neighborhood might lead to better results and should be focused on in further projects, as mentioned in 2.2.1. Regarding the perturbation strategy in the iterated local search algorithm, including a higher degree of randomness within one perturbation step might also enable the algorithm to explore a bigger space of possible solutions, as described in 2.2.2. Another topic worth focusing on in following work is the continuation of implementing the mixed shelf policy.

*Remarks.* All solutions for the different order layouts can be found in the .xml files. The result part in this documentation only provides an intuition behind the comparison of initial solution and neighborhood approach.

Additionally, some order scenarios included orders with a total order weight of more than 18. As the cobot maximum capacity for one batch is 18 and all items of one order have to be picked in one batch, this situation lead to an infeasible solution. We decided to remove those orders from the solution execution and only take into consideration the orders with a total weight smaller or equal 18.

## References

[1] L. Xie, H. Li, L. Luttmann, Formulating and solving integrated order batching and routing in multi-depot agv-assisted mixed-shelves warehouses (2021).

[2] N. Rooy, Effective simulated annealing with python (2020).
URL https://nathanrooy.github.io/posts/2020-05-14/simulated-annealing-with-python/

**Declaration of Authorship**

We hereby declare that

1. we have authored this group project independently and that we have not used other than the declared sources/resources.
2. we have explicitly marked all material which has been quoted either literally or by content from the used sources.
3. according to our knowledge, the content or parts of this group project have not been presented to any other examination authority and have not been published.

| | |
|---|---|
| Anas Suffo | Date |
| Andrey Perov | Date |
| Nele-Pauline Gaedke | Date |