

An Iterated Local Search for a Pharmaceutical Storage Location Assignment Problem with Product-cell Incompatibility and Isolation Constraints

Nilson F. M. Mendes^a, Beatrice Bolsi^b and Manuel Iori^c

Department of Sciences and Methods for Engineering, University of Modena and Reggio Emilia, Reggio Emilia, Italy

Keywords: Storage Allocation, Healthcare Supply Chain, Iterated Local Search, Warehouse Management.

Abstract: In healthcare supply chain, centralised warehouses are used to store large amounts of products close to hospitals and pharmacies in order to avoid shortages and reduce storage costs. To reach these objectives, the warehouses need to have efficient order retrieval and dispatch procedures, as well as a storage allocation policy able to guarantee the safe keeping of items. Considering this scenario, we present a Storage Location Assignment Problem with Product-Cell Incompatibility and Isolation Constraints, that models the targets and restrictions of a storage policy in a pharmaceutical product warehouse. In this problem, we aim to minimise the total distance travelled by the order pickers to recover all products required in a set of orders. We propose an Iterated Local Search algorithm to solve the problem, and present numerical experiments based on simulated data. The results show a relevant improvement with respect to a greedy full turnover procedure commonly adopted in real life operations.

1 INTRODUCTION

Healthcare services are strongly sensible to equipment or medicine shortages, as they could cause attendance delays and interruptions and consequently put patient lives at risk. Traditionally, the main approach to avoid this problem was the use of high inventory levels and constant item replenishment (Uthayakumar and Priyan, 2013) (Aldrighetti et al., 2019). However, this solution has been often considered expensive and hard to be managed, as it requires large dedicated spaces into facilities and workload of healthcare personnel (Volland et al., 2017).


Nowadays, more efficient approaches have been adopted, as acquisitions through *Group Purchasing Organisations* (GPO) and wholesalers. One of the most successful approaches is the sharing of centralised warehouses to store together products to be distributed to different customers located in the same geographical area. Centralised warehouses are particularly useful because they allow healthcare facilities to share a common structure to store a large volume of products and receive them quickly when they


are needed. This enables a constant material flow, a reduction in the personnel costs, a reduced storage space in the customer facility and a lower work burden over healthcare workers.


These advantages are highly dependent on the warehouse reliability and capability of delivering the ordered products in the short terms defined by the customers. This reliability is, in turn, a direct result of an efficient warehouse internal organisation, which requires a good *storage location policy*.

A storage location policy is a general strategy to assign *Stock Keeping Units* (SKU) to storage positions inside a warehouse. It aims at optimising a metric (e.g. total time or distance travelled to store and retrieve SKUs, congestion, space utilisation, pickers ergonomic), while considering issues like product re-allocations efforts, demand oscillation, picking precedence and storage restrictions.

The metric commonly adopted to evaluate the quality of these policies is the distance travelled by the pickers to retrieve all products in a list of orders. This metric is particularly relevant because picking operations accounts for around 35% of the total warehouse operational costs (Wang et al., 2020) and the time/energy spent to reach a product is a waste of resources that must be minimised. In other contexts, the

^a  <https://orcid.org/0000-0002-4924-7341>

^b  <https://orcid.org/0000-0002-6968-8689>

^c  <https://orcid.org/0000-0003-2097-6572>

evaluation may also consider issues like congestion, picker ergonomic, product storage conditions and total space utilisation, that can also lower the warehouse operation efficiency.

In this paper, we describe a problem originated from a real life operation of a pharmaceutical product distributor. It consists in the optimisation of a dedicated storage allocation policy in a picker-to-parts warehouse, i.e., a warehouse where each product has a fixed/dedicated position and pickers travel until product location to retrieve order items. Most specifically, we deal with a *Storage Location Assignment Problem with Product-Cell Incompatibility and Isolation Constraints* (SLAP-PCIIC). In this problem, some products cannot be assigned to some locations due to reasons like refrigeration or ventilation (*product-cell incompatibility*), and some other products need to be isolated from the unconstrained ones due to toxicity or contamination concerns (*isolation*). Furthermore, the problem considers a flexible warehouse layout configuration, with shelves not necessarily grouped on blocks or in a unique pavilion.

This study provides as main contribution an *Iterated Local Search* (ILS) algorithm, that takes a set of orders and a warehouse layout in input and returns a list of assignments of products to locations that minimises the total distance travelled to fulfill the orders.

The remainder of the paper is organised as follows: in Section 2, a concise literature review is presented; Section 3 provides a detailed problem description; Section 4 describes the ILS algorithm and the data processing done before starting the optimisation algorithm; Section 6 describes the numeric experiments carried out, results are presented and then the conclusions are drawn in Section 7.

2 LITERATURE REVIEW

The *Storage Location Assignment Problem* (SLAP) is a generalisation of the well known Assignment Problem in which the objective function is a complex function that usually depends on several factors, like warehouse layout, picking policy, picker routing policy and order batching (Dijkstra and Roodbergen, 2017). It aims at optimising some warehouse metrics like shipping time, equipment downtime, on time delivery, delivery accuracy, product damage, storage cost, labour costs, throughput, turnover and picking productivity (Staudt et al., 2015)(Reyes et al., 2019). The two most common metrics in the literature are picking travel time and travel distance (Reyes et al., 2019), which both require to solve special cases of either the *Travelling Salesman Problem* (TSP) or the *Vehicle*

Routing Problem (VRP), according to the presence of picker capacity constraints or to the simultaneous use of multiple pickers. Using the picker travel distance has as advantage an easier evaluation, as it does not require to deal with issues like congestion and items handling/sorting. Additionally, there are no concerns about picker average speed or searching and handling delays.

In this context, several methods to optimise order picking routes have been proposed, both inside SLAP variants studies or in independent researches. As pointed out in (Dijkstra and Roodbergen, 2017), routing problems in warehouses can be seen as special case of the Steiner Travelling Salesman Problem, that in some layouts can be solved to optimality ((Ratliff and Rosenthal, 1983), (Lu et al., 2016) and (Scholz et al., 2016), (Cambazard and Catusse, 2018)) but in general layouts is mostly solved using heuristics ((De Santis et al., 2018), (Chen et al., 2019), (Roodbergen and Koster, 2001), (Theys et al., 2010)). The exact methods cited, however, are used to solve problems with already defined warehouse allocations (as can be seen also in (Gu et al., 2007) and (Reyes et al., 2019)) and they are mostly algorithms based on a graph theoretic algorithm for single-block warehouses (Lu et al., 2016). A noticeable exception for a joint storage assignment and routing exact optimisation strategy is presented in (Bolaños Zuñiga et al., 2020), but the proposed model reaches optimality only on small instances.

It is important to notice that the routing method efficiency is influenced by the warehouse layout. In both SLAP and picker routing problems, the warehouses can have a single block or multiple blocks. A block is defined as a set of parallel shelves with tight corridors among them (*aisles*), from where it is possible to pick products located in the two shelves at their borders. The connection between these corridors (and consequently between different pairs of shelves) is called *cross-aisle*.

In the same sense, SLAP literature cites frequently two types of picking policies: picker-to-parts and parts-to-picker. In a picker-to-parts warehouse, as those considered in SLAP-PCIIC, the picker receives an item list and visits each item position and transport the items to an accumulation/expedition point.

Once the evaluation method is defined and the parameters above are set, the main decision in SLAP is the storage policy. Among the most relevant policies, we can cite: *random storage*, *dedicated storage* and *group based storage* (Wang et al., 2020)(Žulj et al., 2018).

A random storage policy allocates products in empty positions inside the warehouse using a random

criteria (e.g. closest open location), without including any further complexity in the decision process. In opposite way, dedicated storage ranks the products according to some criteria - popularity, turnover, *Cube per Order Index* (COI) - putting the best ranked products close to the accumulation/expedition locations. Finally, class based storage separates products in groups and assigns the most interesting groups to places close to the accumulation/expedition positions, without fixing a specific position for each product in the set. The SLAP-PCIIC uses a dedicated storage policy where the metric is the total travelled distance.

Several studies report that random storage policies lead to a better space utilisation, due to frequent reuse of storage positions, but raise the travelled distances to pick the products (Muppani and Adil, 2008b) and require higher searching times or control over product locations (Quintanilla et al., 2015). The distance, on other hand, is successfully reduced in dedicated storage policy, but this policy raises re-allocations costs (because of demand fluctuations) and space utilisation, as empty spaces can be reserved to products not currently available. It is stated that class based policies (or zoning) is a balance between random and dedicated storage strategies, but it requires more strategic efforts to define the number of groups and their positions on the warehouse.

A dedicated storage policy is considered in (Dijkstra and Roodbergen, 2017), in which exact distance evaluations are used to define the product assignment. (Guerriero et al., 2013) proposes a non-linear model and an ILS to address a storage allocation problem in a multi-level warehouse considering the compatibility between product classes. (Wang et al., 2020) departs from an S-shape routing policy and multi-level storage to create a two-phase algorithm to assign the items to locations, and use a multi-criteria approximation to evaluate the solutions.

Class based policies are studied in (Rao and Adil, 2013), (Muppani and Adil, 2008b) and (Muppani and Adil, 2008a). In (Rao and Adil, 2013), class boundaries are defined based on the picking travel distance in a two-block and low-level warehouse where returning routing policy is used. The second uses a Simulated Annealing algorithm to define classes and assign locations to them inside a warehouse considering simultaneously space and picking costs.

3 PROBLEM DESCRIPTION

The SLAP can be shortly described as follows: given a set P of products to be stored, a tuple $\omega = (O_1, \dots, O_n)$ of (non necessarily distinct) orders, in

which an order $O_i \subseteq P$ is the subset of products to be picked up by a picker in a single route, and a set L of locations in a warehouse, define an assignment (i.e., an injective function) $g : P \rightarrow L$ in which an evaluation function $z = v(g, \omega)$, to be described next, is minimised. The warehouse is received in input in the form of a graph, and the travel distance d_{ij} between any pair of locations $i, j \in L$ is computed by invoking the Dijkstra algorithm.

In the SLAP-PCIIC, the SLAP variant described here, due to incompatibility and isolation constraints, g is, on the one hand, relaxed to a possible partial assignment, but on the other hand it is subjected to the constraints of the location-product eligibility, i.e., each product is assigned to at most a single location and each location receives at most a single product while respecting the incompatibility and (strong) isolation constraints. In this framework, $v(g, \omega)$ is defined as the sum of minimum travelled distance to pick all the products in each order (designated by $D(g, \omega)$), plus the non negative penalties for non desirable or missing allocations, (designated by $\Phi(g)$). Notice that if all products are allocated, then we have no contribution in the penalty Φ caused by missing allocations, meaning that the lack of product assignment to locations is highly deprecated. Following the company operational rules, it is assumed that the warehouse uses a picker-to-parts picking policy (the picker visits the products locations) and orders splitting/batching are not allowed, making each order an individual and independent route. These assumptions make it possible to decompose the distance $D(g, \omega)$ as the sum of the minimal travelling distances to pick the products in each order O in the ω tuple, designated by $d(g, O)$. With these considerations, and indicating with \mathcal{G} the set of relaxed admissible assignments $g : P \rightarrow L$, the SLAP-PCIIC objective function can be described as:

$$z = \min_{g \in \mathcal{G}} \sum_{i=1}^n d(g, O_i) + \Phi(g) \quad (1)$$

It can be noticed that to evaluate each one of the $d(g, O_i)$ it is necessary to solve another optimisation problem, more specifically a variant of the TSP that calls for the minimization of the travelled distance. Namely, if there is a single accumulation/expedition point, each product is assigned to (at most) a unique location and $O' = \{p_1, \dots, p_{|O'|}\} \subseteq O \subseteq P$ is the requested order deprived of those products lacking of location, then $d(g, O)$ is the minimum distance to depart from the accumulation/expedition point, visit all the locations $(g(p_1), \dots, g(p_{|O'|}))$ in the best possible sequence and then come back. Conversely, in the SLAP-PCIIC we allow the presence of more than one

accumulation/expedition point, so the picker can depart from any of these points and return to another if this operation reduces the total distance travelled $D(g, \omega)$. The algorithms can be easily adapted to deal with the case in which the expedition points are, instead, fixed.

Defining an optimal picker routing in the scenario above is relatively simple if the warehouse is organised in blocks of identical and parallel shelves. However, in the SLAP-PCIIC, the shelves can have different sizes, cell quantities, orientations and positioning and also be located in different pavilions. To deal with this setting, a regular distance matrix containing the distances between each pair of locations is considered as the input of the distance minimisation method, disregarding any further information about the warehouse organisation.

The second part of the objective function, the penalty value $\Phi(g)$, is the sum of two terms: the number of products not assigned to any location $\Phi_1(g)$ and the number of undesired allocations $\Phi_2(g)$. In this sense, the possible configurations of the function g are limited by a set F of assignment incompatibilities and a set I of isolation constraints. Each assignment incompatibility $f \in F$ is a hard constraint (i.e., it must be strictly respected) composed by a tuple of three values (p, n, c) representing a product p , the nature n of the incompatible location (cell, shelf or pavilion) and the code c of the incompatible location, respectively. For instance, the incompatibility $(p_1, \text{"shelf"}, k_1)$ defines that product p_1 cannot be allocated on shelf k_1 .

An isolation constraint is based on the product classification. Given a set T of types, representing the most relevant product characteristic to the storage (toxic, radioactive, humid, etc...), an isolation constraint $\iota \in I$ is a tuple of three values (t, n, s) specifying that products of type $t \in T$ should be allocated in an isolated $n \in \{\text{"cell"}, \text{"shelf"}, \text{"pavilion"}\}$ with an enforcement $s \in \{\text{"weak"}, \text{"strong"}\}$. The enforcement s defines if the isolation is a hard constraint or can be relaxed with a penalty.

4 ITERATED LOCAL SEARCH

In this section, we describe the ILS algorithm that we developed to solve the SLAP-PCIIC. The ILS first invokes a constructive algorithm to generate an initial solution, and then attempts improving the solution by means of three different neighbourhood structures.

The constructive algorithm is divided into two phases (see Algorithm 1). In the first phase (lines from 6 to 23), it assigns locations to products with-

out isolation constraints (i.e., products that are not present in any tuple $\iota \in I$) or with isolation constraints containing an enforcement $s = \text{"weak"}$. In the second phase (lines from 25 to 40), it assigns locations to isolated products, according to their types. This procedure favors the allocation of a greater number of products (i.e., it helps controlling $\Phi(g)$ value), since the constrained products are fewer and could give rise to a large number of unusable locations.

Initially, all the products are sorted by descending order of popularity, i.e., the products more frequently required are put first, and those less frequently after, where popularity is calculated from the tuple of orders ω . Similarly, the storage locations are sorted by distance from the closest accumulation/expedition point. Then, the most popular product is assigned to the closest empty location, if this assignment is allowed. When, instead, an assignment is forbidden, the algorithm tries iteratively the next location, until an allowed position is found or the loop reaches the last position. If a product belongs to a strongly isolated type, it is not assigned during this phase.

In the second phase of the constructive algorithm, all the products belonging to a strongly isolated type are divided by type (line 25 in Algorithm 1) and, similarly, the warehouse locations that are isolated are grouped by level (block, shelf or cell), in line 26. The allocation is done according to the structure size, starting from the isolated blocks and finishing with the isolated cells. At each step, the product types that have the corresponding isolation level are selected and sorted by decreasing order of frequency. Then, each type is assigned to the isolated area where it is possible to maximise the total frequency, without worrying with the internal assignment optimisation. After each step, the list of available positions and products is updated. When all the available isolated spaces are occupied or all the products are assigned, the algorithm ends.

It is important to notice that it may be impossible to assign all the products to the locations in the warehouse due to incompatibility and isolation constraints. This can happen even when the number of available locations is higher than the number of products. Furthermore, the constructive algorithm, as a heuristic method, may be not able to find an initial valid assignment even when it exists. In both the cases mentioned, the solution evaluation procedure penalises the objective function according to the number of products not assigned (i.e., the value of $\Phi_1(g)$).

A valid solution, however, can still present one of the side effects of allocating groups of isolated products together in the warehouse. The first is assigning relatively good positions to several products with a

Algorithm 1: Greedy algorithm.

```

1:  $g \leftarrow \emptyset$  ▷ Start with an empty assignment
2:  $L^* \leftarrow \text{distanceSort}(L)$  ▷ Locations are ordered by distance
3:  $A \leftarrow L^*$  ▷ Available locations
4:  $P \leftarrow \text{sortByFrequency}(P)$ 
5: ▷ First part of greedy algorithm
6: for each  $p \in P$  do ▷ For each product in  $P$ 
7:   if  $\text{isStronglyIsolated}(p)$  then
8:     continue
9:   end if
10:   $l \leftarrow \text{firstAvailable}(A)$ 
11:  while true do
12:    if  $l == \text{null}$  then
13:      break
14:    end if
15:    if  $\text{isForbidden}(l, p)$  then
16:       $l \leftarrow \text{nextAvailable}(A)$ 
17:      continue
18:    end if
19:     $g \leftarrow g \cup (p, l)$ 
20:     $A \leftarrow A \setminus \{l\}$ 
21:    break
22:  end while
23: end for
24: ▷ Second part of greedy algorithm
25:  $\Lambda \leftarrow \text{stronglyIsolatedProductsByType}(P)$ 
26:  $\Psi \leftarrow \text{availableIsolatedStructures}(A)$ 
27:  $\mu \leftarrow \text{allocateOnIsolatedBlock}(Y, \Psi)$ 
28: ▷ First allocation. Assigns products isolated by block
29:  $g \leftarrow g \cup \mu$ 
30:  $\Psi \leftarrow \text{updateAvailableIsolateStructures}(\Psi, \mu)$ 
31:  $\Lambda \leftarrow \text{updatestronglyIsolatedProductsByType}(\mu, P)$ 
32: ▷ Second allocation. Assigns products isolated by shelf
33:  $\mu \leftarrow \text{allocateOnIsolatedShelf}(Y, \Psi)$ 
34:  $g \leftarrow g \cup \mu$ 
35:  $\Psi \leftarrow \text{updateAvailableIsolateStructures}(\Psi, \mu)$ 
36:  $\Lambda \leftarrow \text{updatestronglyIsolatedProductsByType}(\mu, P)$ 
37: ▷ Third allocation. Assigns products isolated by cells
38:  $\mu \leftarrow \text{allocateOnIsolatedCell}(Y, \Psi)$ 
39:  $g \leftarrow g \cup \mu$ 
40:  $\Psi \leftarrow \text{updateAvailableIsolateStructures}(\Psi, \mu)$ 

```

low number of requests due to the existence of some very popular products in the set, that are responsible of a skewed popularity of that group. The second effect is approximately the opposite, i.e., very popular products can be allocated in bad positions due to the fact that average popularity is low for their set. Both these problems are similar to those that are reported in class or zone based storage location problems (Rao and Adil, 2013) (Muppani and Adil, 2008b).

After the greedy algorithm creates an initial solution, its objective function value is calculated according to the procedure described in Section 4.1. The ILS heuristic enters then in a loop that explores the solution space (lines 4 to 25 of Algorithm 2). This loop is composed by three neighbourhood structures that are combined as a single local search, and a perturbation method.

In a loop iteration, each neighbourhood structure evaluates a small set of neighbours of the current solution (denoted by g) and picks the one with lowest objective function value (thus using a best improvement criteria). The best solution in the loop, denoted by g^w , is then compared with the best global solution, g^* , and every time g^w is better, it is assigned to g^* and the number of iterations without improvement (line 22) is reset. Finally, the iteration closes with a perturbation of g^* . The loop stops if the number of iterations without improvement reaches the value of the input parameter *iterations without improvements* (IWI).

To avoid non significant improvements, we assume that an assignment g_1 is better than an assignment g_2 if and only if the objective function value of g_1 is at least 0.1% lower than that of g_2 (i.e., the ratio of between difference of solution values, $v(g_1, O) - v(g_2, O)$, and the old solution value $v(g_2, O)$ must be smaller than $\delta = -0.001$).

Algorithm 2: ILS algorithm.

```

1:  $g^* \leftarrow \text{initialSolution}(L, P)$  ▷ Get greedy assignment
2:  $g \leftarrow g^*$ 
3:  $\text{nonImprovingIter} \leftarrow 0$ 
4: while  $\text{nonImprovingIterations} < \text{IWI}$  do
5:    $g^w \leftarrow g$  ▷ Initialize the best solution on loop
6:    $g' \leftarrow \text{mostFrequentLocalNeighbourhood}(g)$ 
7:   if  $\frac{v(g^w, O) - v(g', O)}{v(g^w, O)} \geq \delta$  then
8:      $g^w \leftarrow g'$ 
9:   end if
10:   $g' \leftarrow \text{insideShelfLocalNeighbourhood}(g)$ 
11:  if  $\frac{v(g^w, O) - v(g', O)}{v(g^w, O)} \geq \delta$  then
12:     $g^w \leftarrow g'$ 
13:  end if
14:   $g' \leftarrow \text{insidePavilionNeighbourhood}(g)$ 
15:  if  $\frac{v(g^w, O) - v(g', O)}{v(g^w, O)} \geq \delta$  then
16:     $g^w \leftarrow g'$ 
17:  end if
18: ▷ Update best global solution
19:   $\text{nonImprovingIter} \leftarrow \text{nonImprovingIter} + 1$ 
20:  if  $\frac{v(g^*, O) - v(g^w, O)}{v(g^*, O)} \geq \delta$  then
21:     $g^* \leftarrow g^w$ 
22:     $\text{nonImprovingIter} \leftarrow 0$ 
23:  end if
24:   $g \leftarrow \text{perturbation}(g^*)$ 
25: end while

```

All the neighbourhood structures used in the local search are simple location swaps between two products. The first neighbourhood (*mostFrequentLocalNeighbourhood*) consists in swapping the assignments of two products belonging to the subset of 20% most required products. The second (*insideShelfNeighbourhood*) consists of swapping the assignments of two products assigned to locations in the same shelf, working as an intensification of the search. The third neighbourhood (*insidePavilion-*

Neighbourhood) is a wider search, in which pairs of products assigned to the same pavilion have their locations swapped.

The neighbourhood structures work following the same steps: randomly choose two products, check if the swap between these products is valid, evaluate the change in objective function and store the best solution found.

To check the swap validity, three rules are used: (1) all swaps that assign a product to a forbidden position are not allowed; (2) no swaps are allowed between a product belonging to a strongly isolated type and a product not belonging to a strongly isolated type; (3) no swaps between products of different types are allowed. It is important to notice that validity does not mean feasibility. In fact, while the two first rules were created to avoid infeasible solutions in the search, the third was created to filter the moves in order to avoid recalculating the penalties related to weak isolated types. It can be noticed that the second and third rules are complementary (the third makes the second redundant). In the numeric tests we tested two algorithm versions, one with rules (1) and (2) (*type-free* swap policy), and the other with rules (1) and (3) (*same-type* swap policy).

To evaluate the solution after a swap, we reevaluate the distances of the routes affected by that swap and the total of products with weak isolation constraints allocated in altered areas. As the number of swaps performed during the algorithm execution is huge and the number of orders can easily reach some thousands, the procedures to evaluate them must have a strong performance.

After all neighbourhoods have been explored and the global best solution has possibly been updated, a perturbation is performed over the best global solution. It consists in $|P|/20$ unconstrained and valid swaps, chosen randomly and applied in a loop. The resulting solution is then used as the initial solution in the next ILS iteration.

4.1 Solution Evaluation

To evaluate the routing distance, we propose a combination of two ideas: the former is using different TSP algorithms according to the size of the instance and the latter is to keep a mapping of routes that passes from each position to recalculate only picking distances of routes containing products involved in the swap. In the latter case, initially the algorithm retrieves all the routes affected and controls if a route passes from both locations where the products are currently allocated. If the route has both locations on it, the reevaluation is useless (because the set of loca-

tions to be visited does not change), otherwise it is evaluated with the new location replacing the old one.

The evaluation is controlled by the parameter $TSP(\alpha, \beta)$. In this parameter, the constants α and β are two integers representing the order size thresholds used to choose each algorithm method for estimating the minimum distance to visit the product locations in a route. Let $|O|$ be the number of items in an order O in the tuple ω , if $|O| \leq \alpha$ an exhaustive search is run (i.e., all the possibilities are tested and then the distance found is optimal). Otherwise, if $\alpha < |O| \leq \beta$, a closest neighbour algorithm is used to initialise the route and a subsequent quick local search is performed. In this local search, $(|O| - 1)$ swaps among two consecutive locations are tested in $(|O| - 1)$ iterations (always departing from the first to the last product) and the current solution is updated when the swap reduces the smaller distance. Finally, if $|O| > \beta$, just the closest neighbour heuristic is used.

The route evaluation method described above presents a quadratic complexity, as the exponential time approach is limited according to the number of visited points. Although it does not guarantee an optimal route, it is better suited to our purpose of using non block-based warehouse layouts than algorithms based on the method proposed in (Ratliff and Rosenthal, 1983).

In order to explain how the evaluation of penalties is done by breaking weak isolation constraints, we use Algorithm 3 below. This pseudo code shows the process for shelves, but it is practically identical for cells and pavilions.

Algorithm 3: Isolation penalty evaluation after a swap.

```

1: totalPenalty ← 0
2: S ← allShelves(L)
3: Tw ← WeakIsolationTypes()
4: Ps ← productsAllocated(g, s)  ∀s ∈ S
5: for each s ∈ S do
6:   Pi ← {p | p ∈ Ps, type(p) ∈ Tw}
7:   Pj ← {p | p ∈ Ps, type(p) ∉ Tw}
8:   Ts ← {type(p) | p ∈ Ps}
9:   ▷ Group products with isolation constraints by type
10:  Ht ← {p ∈ Pi | type(p) = t}  ∀t ∈ Tw
11:  if |Pi| = 0 or |distinct(Ts)| = 1 then
12:    continue
13:  end if
14:  penalty ← 0
15:  x ← maxt ∈ Tw (|Ht|)  ▷ Type with max cardinality
16:  r ← x
17:  if |Pj| ≥ |Pi| then
18:    penalty ← Wpen * |Pi|2 / |Ps|
19:  else
20:    penalty ← Wpen * (|Pj|2 + r) / |Ps|
21:  end if
22:  totalPenalty ← totalPenalty + penalty
23: end for

```

First of all, the algorithm gets all products allocated and groups them by shelves (line 4). For each shelf, products are grouped by type (line 9) and then the algorithm counts the number of different product types assigned to the shelf. If there is only one type assigned to the shelf or if no assigned product has an isolation constraint $\iota \in I|n = \text{“shelf”}$ (see isolation constraint definition on Section 3), no penalty is applied (lines 11 to 13). Otherwise, the actual penalty evaluation is performed (lines 14 to 22).

The penalty strategy is based on the division of the products assigned to a structure (which can be a shelf, a cell or a pavilion) into two groups, one with isolation constraints and another without. One of these groups is defined as the minority (resp. majority) if it is the group with less (resp. more) assignments in a structure.

Departing from the *minority (majority)* definition, the method tries to push the search through the predominant configuration by penalising minority groups. For example, if the shelf is mostly occupied by products belonging to types without isolation constraints, it penalises the products with isolation constraints (lines 17 and 18) that are the minority. Similarly, it penalises products belonging to types without isolation constraints if they are the minority in the shelf (lines 19 and 20). Furthermore, to differentiate among similar assignments, we consider the proportion of products belonging to the minority/majority over the total number of products, instead of simply counting the number of products. This decision was taken due to the fact that only counting the products was causing no changes in the total penalty after a swap.

5 INSTANCE SETS

In this section, we describe the instance sets built up to test the algorithm performances. We created three warehouse layouts W_1, W_2 and W_3 with several differences between them, not only regarding the number of positions, but also regarding how these positions are distributed in the area. A short description of the warehouses is provided in Table 1.

Table 1: Warehouse layout overview.

ID	pavilions	shelves	cells	accum/exp points
W_1	1	10	200	3
W_2	1	10	240	3
W_3	2	12	260	3

The experiments were divided into two parts. The first part was aimed at analysing the algorithm per-

formance considering only the travelled distance and thus its suitability at dealing with directly calculated distances. The second part tested the performance when considering the incompatibility and isolation constraints, in order to check if these constraints were well handled in the algorithm.

By using the instance set I_1 , we tested both the insertion of products and the algorithm parameters. We experimented the insertion of two sets of products in the warehouses, one with 100 and the other with 200 products. For each product set, we tested scenarios with $|\omega| = 500, 1000$ and 5000 orders. For each combination of warehouse, number of products and number of orders, five realistic instances were created, for a total of $|I_1| = 3 \cdot 2 \cdot 3 \cdot 5 = 90$ instances. In all these instances, we used cells with only one position/level, as showed in Table 1.

Regarding the algorithm parameters, we investigated two values: the maximum number of IWI, used as stopping criteria, and the TSP thresholds described in Section 4. Three values of maximum number of IWI were tested, and three different combination of the two TSP thresholds. Additionally, we tested the local search with and without incompatibility of swaps between products of different types. In this way, $3 \cdot 3 \cdot 2 = 18$ algorithm parameter combinations were experimented.

To test the algorithm capabilities in managing incompatibility and isolation constraints, we used a subset of the initial instances, using 3 variations to each combination of warehouse, number of products and number of orders, for a total of $3 \cdot 2 \cdot 3 \cdot 3 = 54$ instances. For each of these 54 instances, we tested 5 incompatibility and isolation constraints, leading to a new instance set I_2 , with $|I_2| = 5 \cdot 54 = 270$.

In all the instances mentioned above, the number of products by order was determined following a Poisson distribution with average number of events equal to 6. The products inside each order were defined following a uniform distribution, but preventing the same product from being requested twice in the same order.

6 COMPUTATIONAL EVALUATION

In this section, we describe the numeric experiments performed to test the efficiency of the proposed ILS algorithm. In these experiments, we observed the algorithm performance on different instances, the average run time, the local search capacity of improving the initial solution, the influence of the parameters on the final solution, and the solution quality.

The algorithm was implemented in C++17 language, with source code being compiled with `-O3` flag. The tests were performed on a Dell Precision Tower 3620 computer with a 3.50GHz Intel Xeon E3-1245 processor and 32GB of RAM memory, running the Ubuntu 16.04 LTS operational system. All the executions were performed on a single thread, without any significant concurrent process.

The results obtained for the I_1 instance set are presented in Table 2, grouped by the TSP evaluation parameters. Each line in this table represents the average over five instances of the computational results with a specific algorithm parameter setting. Column z_0 gives the average initial solution value produced by the greedy algorithm, column z_{best} gives the average solution value produced by the ILS, column time(s) reports the average run time of the ILS in seconds, and column %G shows the average percentage gain of z_{best} with respect to z_0 .

We start our analysis from the effect of the TSP evaluation parameters on the solution value. As the constructive algorithm is not affected by these parameters, it provides always the same initial solution to a given instance, but with a different objective function value, due to the difference in the solution evaluation.

As expected, using exact evaluations (and better heuristics) in more routes leads to a decrease in the objective function values, but considering the initial solution value, this variation is inferior to 5% in total from the most precise to the less precise evaluation, which suggests that simple heuristics are sufficiently efficient to check the quality of a storage assignment. This evidence is in accordance with what is stated in the literature and practised in real scenarios, mainly by the pickers, that tend to follow the most intuitive and sub-optimal routes (De Santis et al., 2018) (Elbert et al., 2017). On the other hand, the improvement on evaluation precision is unequivocally counterbalanced by a significant run time increase if the whole algorithm is considered. The total run time using the $TSP(7, 11)$ configuration is over the double of the total run time using the $TSP(5, 9)$ configuration.

An interesting effect of the TSP evaluation parameter in the solution is the negative correlation between the evaluation precision and the improvement obtained by the local search. In other words, if we increase the number of routes that are evaluated by an exact method (or better heuristics), we get less improvement over the greedy solution. A possible cause of this result is the higher probability of a travel distance over-estimation when evaluating good-quality solutions if the evaluation precision is low. This could guide the search to a region with low-quality solutions, increasing the convergence time without im-

proving the solution quality.

In the same sense, it is noticeable that an increase in the number of products and orders in the instances also reduces the gains over the initial solution. This is an expected result, as it is harder to balance all picker route distances if there are more routes to balance and more points to visit among the different routes.

When comparing the results of scenarios with the same evaluation parameters but different IWI values, it is possible to notice that a higher IWI only slightly improves the average gains over the initial solution (on average less than 0.7 percentage points increase for each two more IWI), even with significantly higher run times. Among the hypotheses to explain this behaviour, we can cite a bad performance of the perturbation method to escape from local optima, a too quick convergence of the method to values close to an average local optimum, the existence of too many local optima, or the existence of several regions of the solution space with very similar characteristics causing repetitive searches.

It is interesting to notice that when comparing “type-free swap” with “same-type swap” results, the latter on average gets better solutions with a higher run time. We expect that a more restricted swap could lead to a quick convergence and thus a worse solution. As the differences between solutions are relatively small, while between run times are relevant, we can suppose that the method, when using a more restricted local search, performs more but smaller improvements. This explanation is compatible with concerns about meta-heuristic parameter calibration, in which a developer tries to balance the size of intensification and diversification steps in order to find a better algorithm performance.

In Table 3, we show the algorithm results for instances with isolation and allocation incompatibility constraints. In the table, Φ_0 gives the average value of the initial penalties, Φ_{best} the average value of the penalties in the best solutions produced by the ILS, $\%G_z$ the average percentage gain over the initial objective function values, and $\%G_\Phi$ the average percentage gain over the initial penalties. In Table 3, the block *Isolation 1* refers to instances containing one type of weak isolation constraints, the block *Isolation 2* refers to instances containing one type of weak isolation and one type of strong isolation constraints. Besides that, both the blocks *Isolation 1* and *Isolation 2* in the table are subjected to allocation incompatibilities.

We can first observe a satisfactory improvement over the initial solution obtained by the local search, although smaller than that observed in the previous results. Nevertheless, in instances without hard iso-

Table 2: Computational results on instance set I_1 . Each line is an average on 90 instances.

Swap policy	IWI	TSP(5,9)				TSP(6,10)				TSP(7,11)			
		z_0	z_{best}	time(s)	%G	z_0	z_{best}	time(s)	%G	z_0	z_{best}	time(s)	%G
Same-type	6	1675434.0	1025435.0	683.0	41.88	1644702.6	1059339.3	788.0	38.27	1604009.9	1149117.5	1519.7	30.85
	8	1675434.0	1016371.8	830.8	42.70	1644702.6	1047753.8	991.4	39.19	1604009.9	1137799.8	1709.8	31.47
	10	1675434.0	1007464.1	1038.8	43.34	1644702.6	1042574.2	1126.9	39.70	1604009.9	1130933.5	2078.8	32.10
Type-free	6	1675434.0	1035816.2	589.0	41.39	1644702.6	1065611.6	722.7	37.59	1604009.9	1153347.8	1224.9	30.18
	8	1675434.0	1030057.3	709.2	41.85	1644702.6	1059577.0	893.1	38.21	1604009.9	1144094.8	1525.9	30.75
	10	1675434.0	1024599.0	866.0	42.26	1644702.6	1055123.4	1090.6	38.56	1604009.9	1143325.2	1858.5	31.20

Table 3: Computational results for instances with isolation and incompatibility constraints on instance set I_2 . Each line is an average on 3 instances. Parameters used: 8 IWI, TSP(7, 11). Swap policy: *type-free*.

Id	P	W	ω	Isolation 1							Isolation 2						
				z_0	Φ_0	z_{best}	Φ_{best}	%G _z	%G _{Φ}	time(s)	z_0	Φ_0	z_{best}	Φ_{best}	%G _z	%G _{Φ}	time(s)
1	500	W ₁	405287.9	51048.9	225120.3	37610.6	44.32	25.38	276.5	632041.1	344081.8	503372.6	317109.3	20.36	7.84	288.2	
2	1000		749806.0	49873.4	430050.9	31858.6	42.63	35.43	713.7	1187816.0	621282.7	977614.8	590412.1	17.64	4.96	499.6	
3	5000		3558644.9	49217.2	2193491.7	52373.3	38.36	-6.53	2839.9	5659723.9	2795037.9	4846198.5	2786653.8	14.37	0.30	2119.7	
4	500	W ₂	383964.6	54282.0	240575.6	42760.2	37.33	20.93	261.0	613230.2	353249.2	534527.6	341856.0	12.82	3.15	203.0	
5	1000		704103.1	51699.1	469837.4	42271.7	33.28	17.90	680.6	1162098.5	644019.8	1025637.5	619162.5	11.77	3.90	489.7	
6	5000		3332955.4	54780.4	2276342.0	52617.7	31.70	2.89	3005.5	5555117.6	2933844.9	4960398.8	2911717.1	10.71	0.76	2388.9	
7	500	W ₃	371185.3	58468.7	251748.4	36607.4	32.19	36.99	334.2	575261.8	346403.8	518039.3	334616.3	9.94	3.39	402.6	
8	1000		673638.7	49932.3	504716.8	38462.8	25.07	21.97	611.3	1096946.6	632364.3	1001514.8	621692.1	8.70	1.73	519.2	
9	5000		3221278.3	58714.3	2444756.2	46692.6	24.10	19.00	3519.7	5296059.8	2928326.8	4907968.1	2915739.4	7.32	0.43	2897.5	
10	500	W ₁	438229.7	27666.7	326534.0	26333.3	25.48	4.87	281.8	728354.9	363274.5	576015.8	297922.2	20.98	18.17	278.5	
11	1000		854100.0	32666.7	633261.7	24000.0	25.83	26.33	766.9	1223601.2	503051.5	1022193.0	412985.7	16.44	17.91	622.5	
12	5000		4208520.8	51535.1	3145456.1	59535.1	25.26	-25.94	2824.4	5413919.5	1718497.8	4499844.1	1673331.1	16.88	2.62	3092.1	
13	500	W ₂	425402.5	30307.5	306401.6	25061.0	27.97	14.48	254.7	711924.2	362643.5	546403.6	277711.3	23.24	23.43	282.3	
14	1000		814109.8	28141.2	622948.2	23876.2	23.47	15.01	760.5	1210574.2	515456.9	993881.1	447135.4	17.88	13.18	678.9	
15	5000		4046685.1	32227.3	2971407.6	30121.8	26.57	6.28	3593.7	5265289.3	1724671.7	4412597.4	1637695.8	16.19	5.02	3621.2	
16	500	W ₃	431234.1	32694.1	356368.3	23407.3	17.36	28.40	441.0	711317.5	360032.5	611663.2	297630.6	13.98	17.25	229.6	
17	1000		833826.9	31042.3	689633.4	24334.8	17.29	21.15	775.5	1208464.6	504536.9	1055921.7	426959.7	12.61	15.21	532.4	
18	5000		4165773.8	30477.8	3403163.1	31035.4	18.31	-2.27	3409.6	5349024.8	1733807.5	4761622.8	1645191.5	10.98	5.11	3864.5	
Totals			1731243.0	42656.8	1257468.8	42760.2	37.33	20.93	1486.4	2523801.4	1100055.3	2180204.3	341856.0	12.82	3.15	1362.0	

Isolation constraints, this metric raises to 19.4% and 15.1% in blocks *Isolation 1* and *Isolation 2*, respectively. These results may suggest that if the constructive algorithm does not handle well isolation constraints, then local search has troubles in improving the solution.

We notice that the local search reduces proportionally less the penalty value than the overall objective function value, except in instances with 200 products in warehouse W_3 (lines 16 to 18 of Table 3), suggesting that the method could be giving more relevance to travel distance.

The average run time is smaller than one hour for almost all instance settings, except on instances on lines 15 and 18 and *Isolation 2*. It suggests that the method converges in an acceptable time even in more realistic and complex mid-size instances.

7 CONCLUSION

In this paper, we study the Storage Location Assignment Problem with Product-Cell Incompatibility and Isolation Constraints, a generalisation of the classic Storage Location Problem in which some products may need to be allocated in reserved positions and some other products can not be assigned to specific

locations. This problem lies in the context of a pharmaceutical logistic operator aiming at more flexibility in defining warehouse layouts, while improving its performance.

We propose an ILS method to solve the problem and we test it on a large set of instances to demonstrate its suitability in providing good solutions within an acceptable run time. We could notice that variations on the stopping criteria cause relevant changes in the run time, but just slight changes in the solution quality. On the other hand, the variations in the parameters related to the TSP solutions (to determine the pickers' routes) prove to be very influential in both run time and solution quality. Large instances are well handled by the ILS algorithm, though with higher run times, with this raise strongly related to the number of products to be allocated and less to the number of orders considered. Instances with isolation and incompatibility constraints present lower improvements in the local search phase, but still relevant for commercial purposes.

For future researches, we suggest to investigate more deeply the influence of initial allocation of strongly isolated types on the overall performance of the optimization method. General weighting of the different terms in the objective function 1 should be addressed, after an extensive analysis of several fac-

tors, e.g. incidence of the ratio between warehouse dimensions and the number of orders, or the distribution of the various types of penalties among products. This could also lead to the development of interesting multi-objective optimization algorithms. Investigating the suitability of the method to more dynamic situations is also relevant, mainly when different sources of information are available for the orders.

REFERENCES

- Aldrighetti, R., Zennaro, I., Finco, S., and Battini, D. (2019). Healthcare supply chain simulation with disruption considerations: a case study from northern Italy. *Global Journal of Flexible Systems Management*, 20(1):81–102.
- Bolaños Zuñiga, J., Saucedo Martínez, J. A., Salais Fierro, T. E., and Marmolejo Saucedo, J. A. (2020). Optimization of the storage location assignment and the picker-routing problem by using mathematical programming. *Applied Sciences*, 10(2):534.
- Cambazard, H. and Catusse, N. (2018). Fixed-parameter algorithms for rectilinear steiner tree and rectilinear traveling salesman problem in the plane. *European Journal of Operational Research*, 270(2):419–429.
- Chen, F., Xu, G., and Wei, Y. (2019). Heuristic routing methods in multiple-block warehouses with ultra-narrow aisles and access restriction. *International Journal of Production Research*, 57(1):228–249.
- De Santis, R., Montanari, R., Vignali, G., and Bottani, E. (2018). An adapted ant colony optimization algorithm for the minimization of the travel distance of pickers in manual warehouses. *European Journal of Operational Research*, 267(1):120–137.
- Dijkstra, A. S. and Roodbergen, K. J. (2017). Exact route-length formulas and a storage location assignment heuristic for picker-to-parts warehouses. *Transportation Research Part E: Logistics and Transportation Review*, 102:38–59.
- Elbert, R. M., Franzke, T., Glock, C. H., and Grosse, E. H. (2017). The effects of human behavior on the efficiency of routing policies in order picking: The case of route deviations. *Computers & Industrial Engineering*, 111:537–551.
- Gu, J., Goetschalckx, M., and McGinnis, L. F. (2007). Research on warehouse operation: A comprehensive review. *European Journal of Operational Research*, 177(1):1–21.
- Guerrero, F., Musmanno, R., Pisacane, O., and Rende, F. (2013). A mathematical model for the multi-levels product allocation problem in a warehouse with compatibility constraints. *Applied Mathematical Modelling*, 37(6):4385–4398.
- Lu, W., McFarlane, D., Giannikas, V., and Zhang, Q. (2016). An algorithm for dynamic order-picking in warehouse operations. *European Journal of Operational Research*, 248(1):107–122.
- Muppani, V. R. and Adil, G. K. (2008a). A branch and bound algorithm for class based storage location assignment. *European Journal of Operational Research*, 189(2):492–507.
- Muppani, V. R. and Adil, G. K. (2008b). Efficient formation of storage classes for warehouse storage location assignment: a simulated annealing approach. *Omega*, 36(4):609–618.
- Quintanilla, S., Pérez, Á., Ballestín, F., and Lino, P. (2015). Heuristic algorithms for a storage location assignment problem in a chaotic warehouse. *Engineering Optimization*, 47(10):1405–1422.
- Rao, S. S. and Adil, G. K. (2013). Optimal class boundaries, number of aisles, and pick list size for low-level order picking systems. *IIE Transactions*, 45(12):1309–1321.
- Ratliff, H. D. and Rosenthal, A. S. (1983). Order-picking in a rectangular warehouse: a solvable case of the traveling salesman problem. *Operations Research*, 31(3):507–521.
- Reyes, J., Solano-Charris, E., and Montoya-Torres, J. (2019). The storage location assignment problem: A literature review. *International Journal of Industrial Engineering Computations*, 10(2):199–224.
- Roodbergen, K. J. and Koster, R. (2001). Routing methods for warehouses with multiple cross aisles. *International Journal of Production Research*, 39(9):1865–1883.
- Scholz, A., Henn, S., Stuhlmann, M., and Wäscher, G. (2016). A new mathematical programming formulation for the single-picker routing problem. *European Journal of Operational Research*, 253(1):68–84.
- Staudt, F. H., Alpan, G., Di Mascolo, M., and Rodriguez, C. M. T. (2015). Warehouse performance measurement: a literature review. *International Journal of Production Research*, 53(18):5524–5544.
- Theys, C., Bräysy, O., Dullaert, W., and Raaijmakers, B. (2010). Using a tsp heuristic for routing order pickers in warehouses. *European Journal of Operational Research*, 200(3):755–763.
- Uthayakumar, R. and Priyan, S. (2013). Pharmaceutical supply chain and inventory management strategies: Optimization for a pharmaceutical company and a hospital. *Operations Research for Health Care*, 2(3):52–64.
- Volland, J., Fügener, A., Schoenfelder, J., and Brunner, J. O. (2017). Material logistics in hospitals: A literature review. *Omega*, 69:82–101.
- Wang, M., Zhang, R.-Q., and Fan, K. (2020). Improving order-picking operation through efficient storage location assignment: A new approach. *Computers & Industrial Engineering*, 139:106186.
- Žulj, I., Glock, C. H., Grosse, E. H., and Schneider, M. (2018). Picker routing and storage-assignment strategies for precedence-constrained order picking. *Computers & Industrial Engineering*, 123:338–347.