NEW SOLUTION METHODS FOR THE BLOCK RELOCATION PROBLEM

FABIEN TRICOIRE[(1)], JUDITH FECHTER[(2)], AND ANDREAS BEHAM[(2)]

Abstract. This technical report presents new solution methods for the block relocation problem (BRP). Although most of the existing work focuses on the restricted BRP, we tackle the unrestricted BRP, which yields more opportunities for optimisation. Our contributions include fast heuristics able to tackle very large instances within seconds, fast metaheuristics that provide very competitive performance on benchmark data sets as well as a branch-and-bound algorithm that outperforms the best existing exact method and provides new optimal solutions. The branch-and-bound algorithm is adapted to tackle the restricted BRP as well, again outperforming existing methods for that problem.

Keywords: block relocation problem, heuristics, branch-and-bound

1. Introduction

In a world where container port traffic increases steadily [2], the optimisation of logistics at container terminals is more relevant than ever. Containers are typically stored in stacks between their arrival at the terminal and their departure when a vehicle picks them up. Similarly, at steel production factories, slabs are stored in stacks between their production and the time when they are picked up by a vehicle for further delivery. In both contexts, space is limited and the way items are stacked, be they containers or slabs, has an impact on productivity. In both cases it is possible to reorganise stacks with a crane in order to improve productivity. For this reason, optimisation problems related to loading, unloading and premarshalling of stacks in storage areas have been the subject of increasing attention over the last five years, as emphasised in a recent survey by Lehnfeld and Knust [11]. In this report we provide new advances for the block relocation problem (BRP), which is an unloading problem which we succinctly describe now.

A set of n items, usually called blocks or containers, are organised into W stacks and have to be retrieved in a certain order. It is usually considered that item 1 has to be retrieved first, then item 2, and so on until item n. An item may only be retrieved if it is on top of its stack. Otherwise, the items that block it (i.e. are above it) must be relocated to another stack first. A relocation involves taking an item from the top of a stack and putting it on top of another stack. Additionally,

Date: (1) Department of Business Administration, University of Vienna, Oskar-Morgenstern-Platz 1, 1090 Vienna, Austria
fabien.tricoire@univie.ac.at
(2) Heuristic and Evolutionary Algorithms Laboratory, University of Applied Sciences Upper Austria, Softwarepark 11, 4232 Hagenberg, Austria
{judith.fechter|andreas.beham}@fh-hagenberg.at
.

stacks may not exceed a certain height H_{max}. The goal is to retrieve all items while minimising the total number of relocations required to do so.

It is possible to represent the BRP using a graph: to each configuration of the items in the bay, which we call a *state*, we associate a node. To each possible relocation from one state to another, we associate an arc with weight 1. To each possible retrieval of an item, also representing a transition from one state to another, we associate an arc with weight 0. Considering the set V of all nodes (possible states) and the set A of all arcs (possible transitions between those states), we obtain a directed graph $G = (V, A)$. The shortest path from the initial state (source) to the empty state (sink) then yields the optimal solution to the BRP. An example of this graph representation is given in Appendix 6. However the size of V and A grows exponentially with the number of items and the number of stacks. It is therefore impractical to consider this whole graph explicitly. Any optimisation method for the BRP consists in finding a path from source to sink while exploring only a subset of the whole graph.

The contribution of this report is threefold. First, we develop new heuristic operations to quickly build high quality solutions for the BRP. Second, we design a new constructive metaheuristic framework to use these operations in an even more efficient way, trading a bit of CPU effort for increased solution quality. Third, we develop a new exact method for the BRP, which is also applied to the restricted BRP with minor modifications. All methods are validated through experiment, and we establish that all of them improve on the current state of the art.

The remainder of this report is organised as follows. In Section 2, we quickly survey recent literature on the BRP. In Section 3 we present various heuristic methods for the unrestricted BRP. In Section 4 we introduce a simple but effective branch-and-bound algorithm for the unrestricted BRP. We also adapt it to work with the restricted BRP. Section 5 presents experimental validation for our contributed optimisation methods. Finally, we draw some conclusions and suggest further research directions.

2. Literature review

The BRP has received notable attention in the last six years. In their recent survey, Lehnfeld and Knust [11] present an overview of the scientific literature on loading, unloading and premarshalling problems. Using their terminology, the BRP is an unloading problem. They make a difference between *forced moves*, that involve relocating items blocking the next item to be removed, and *voluntary moves*, that involve any other type of relocation. Voluntary moves are also called *cleaning moves* by Petering and Hussein [12]. We want to emphasise this difference here since most of the existing contributions only consider forced moves. Put differently, the assumption is that the only items that may be relocated are those blocking the next item to be retrieved. From now on, we call this assumption A1, as named by Caserta et al. [4]. The BRP under assumption A1 is the restricted BRP, otherwise it is the unrestricted BRP. It should be noted here that under assumption A1, only two situations can occur: (i) Either the next item to be retrieved is on top of a stack, in which case it is retrieved; (ii) Or there are items blocking it, in which case these items are relocated. In that context, the only decision making that can be done is determining to which other stack these items should be relocated.

The BRP is first introduced by Kim and Hong [9]. They consider the now-common BRP setting where every item has a different priority, as well as a setting where priorities are given to groups of items. Their work is under assumption A1, i.e. only forced moves are considered. The authors propose a branch-and-bound algorithm to solve instances with up to 30 items and 6 stacks within less than one hour. They also design a heuristic rule to relocate items, that estimates the expected number of additional relocations needed after a certain relocation and greedily selects the relocation minimising that number. This heuristic produces solutions within one or two seconds and ranges on average between 2% and 16% from the optimum, depending on instance classes.

Lee and Lee [10] present a three-phase heuristic for a BRP where the objective function also considers the distance between stacks. Assumption A1 is also made. In the first phase, a solution is constructed heuristically. In the second phase, the number of relocations is reduced by merging relocations of same items. This is performed using a mixed-integer program (MIP). In the third phase, a consideration of distance between stacks is added to a simplified version of the MIP from phase 2.

Caserta et al. [3] also use assumption A1 and design a corridor method algorithm. The corridor method is a heuristic framework that explores limited amount of solutions within an exact framework, similar to what is done in beam search. In that case, the exact framework is a dynamic programming algorithm.

Jovanovic and Voss [8] propose a chain heuristic that considers the next two items to be relocated and avoids to relocate the first one to the stack that is the best for the second one. This is under assumption A1. The authors insist on the difference between simple and complex methods. Simple methods typically compute within less than a second. They compare this method to other previous contributions, notably Wu and Ting [14] and Caserta et al. [4], and show that their chain heuristic performs better than the other simple heuristics. The best results reported are from the beam search of Wu and Ting [14], but at the expense of CPU effort.

Wu and Ting [14] develop a beam search algorithm for the BRP with assumption A1. The beam search is based on a breadth-first branch-and-bound algorithm. Using a depth-first scheme instead, the authors also propose a branch-and-bound algorithm. They also propose three heuristic rules to determine where the next item should be relocated.

Tanaka and Takii [13] introduce a new lower bound for the restricted BRP. This lower bound is not applicable to the unrestricted BRP since it relies on the fact that items blocking the next item to remove have to be relocated next. Integrating it into a branch-and-bound algorithm, they show that their new lower bound improves the performance of branch-and-bound on the restricted BRP.

Caserta et al. [4] provide a complexity study of the BRP. They prove that the BRP is NP-hard for any finite H_{max} and $W < n$. They then formulate two MIP models, BRP-I and BRP-II. BRP-I considers all possible relocations while BRP-II follows assumption A1. They also provide a simple example that emphasises that assumption A1 implies losing optimality. We reproduce this example in Figure 1. However, only model BRP-II is tested, so there is no assessment of the cost of assumption A1. Finally, the authors also provide a simple heuristic under assumption A1.

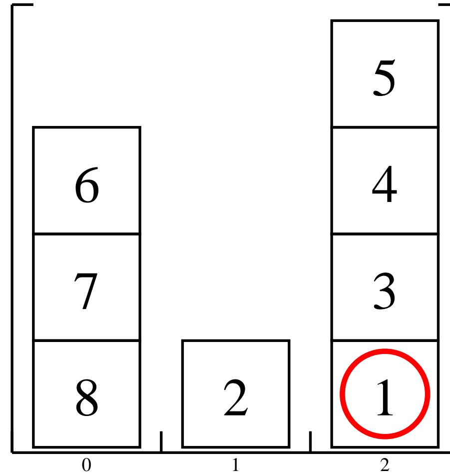

Figure 1: For $H_{max} = 4$, optimal solution with assumption A1 has 6 relocations but optimal solution without A1 has 4 relocations.

Expósito-Izquierdo et al. [6] correct the BRP-II model from Caserta et al. [4], then present a new branch-and-bound algorithm for the restricted BRP. The branch-and-bound is compared to the A* algorithm from Expósito-Izquierdo et al. [5] and results show that the newer method is faster.

Zehendner et al. [15] present an improved mathematical formulation for the restricted BRP. They first correct the BRP-II model from Caserta et al. [4], then provide an alternative model with less variables. This allows them to reduce CPU effort and to solve more instances than with the corrected BRP-II.

Forster and Bortfeldt [7] develop methods for the unrestricted BRP where priorities are given to groups of items rather than to single items. They first develop an improved lower bound, compared to previous contributions simply considering the number of items that need to be relocated. They also develop a construction heuristic that applies what they call BG moves. BG stands for Bad-Good and involves the relocation of an item which was previously blocking another item into a stack where it does not block any item. If BG moves can be performed then they are performed, with a priority given to moves to a non-empty stack. A tree search is also presented. In order to keep CPU effort low, only certain moves are considered, so the method is similar to a beam search. BG moves are always preferred within this tree search procedure. The authors then compare their method to previous contributions, although these previous contributions are considering the restricted BRP.

Petering and Hussein [12] consider the BRP without assumption A1. They first develop a MIP model, called BRP-III, with considerably less variables than BRP-I from Caserta et al. [4]. Although it provides a lower bound of worse quality than BRP-I, BRP-III still performs better on average. Still, the authors conclude that a mathematical programming approach is not sufficient for real-world use, and

propose a look-ahead heuristic called LA-N. The N refers to the degree of looking-ahead. Instead of attempting to relocate items from only the stack of the next item to retrieve, LA-N looks at the stacks of the next N items to retrieve. If there is a promising voluntary move in one of these stacks, it is performed. Voluntary moves are considered promising if they involve relocating an item that needs to be relocated sooner or later, in a way that involves never having to relocate it again. By definition, LA-1 only considers forced moves. LA-N is compared to fast methods from previous contributions (Kim and Hong [9], Aydin and Ünlüyurt [1], Caserta et al. [3], Lee and Lee [10]) and shows to be competitive on some instance sets while dominating other methods on other instance sets. Overall it is safe to conclude that it is the best heuristic among all those compared.

Expósito-Izquierdo et al. [5] develop an A* algorithm for the BRP with and without assumption A1. They obtain optimal solutions that are inconsistent with those reported in Caserta et al. [4], and report the inconsistency. However their solutions are always checked for feasibility and sometimes better than those reported by Caserta et al. [4], which suggests the errors are in the latter. They also develop a knowledge-based heuristic and compare it to known optimal solutions as well as to the solutions obtained by Forster and Bortfeldt [7]. The knowledge-based heuristic is better on average, but not systematically.

Most contributions cited above focus on the restricted BRP, i.e. under assumption A1. However it is difficult to find a good reason for that assumption other than making the problem more tractable. In the following, we focus on the unrestricted BRP. Since the branch-and-bound algorithm from Section 4 can very easily be adapted to tackle the restricted BRP, we do perform such an adaptation. Therefore we also report results on the restricted BRP.

3. New methods for the block relocation problem

We look at two kinds of operations for voluntary moves, which we name *safe moves* and relocating *decreasing sequences*. We also consider forced moves, as abundantly described in the literature. In that case, we follow the LA-1 heuristic described by Petering and Hussein [12], which is similar to the LA heuristic presented by Caserta et al. [4]. We now describe safe moves and decreasing sequences, then we explain how they can be combined into heuristics. For that purpose, we use the following notation:

- Γ is the set of stacks.
- $top(s)$ is the item on top of stack s.
- $tops() = \{top(s) | s \in \Gamma\}$ is the set of all items that have no item above them.
- $lowest(s)$ is the lowest-index element in stack s. For instance if stack s has items 6, 4, 2, 7 in that order then $lowest(s) = 2$. If s is an empty stack then $lowest(s) = n + 1$ (n being the index of the final item to remove).
- NR is the number of relocations that have already been performed to reach the current state.
- $lb(i)$ is 0 if i is well-placed, i.e. is not placed above an item of lower index; 1 otherwise.
- $sr(i)$ is 0 if i can be placed on top of a stack, other than its current stack, without creating any conflict; 1 otherwise.

- LB is a valid lower bound on the number of necessary relocations. In its simplest form: $LB_1 = \sum_{i \in \{1..n\}} lb(i)$. Forster and Bortfeldt [7] introduce a more advanced version (LB_2) which is $1 + LB_1$ if $\nexists i \in tops()|lb(i)+sr(i) < 2$, LB_1 otherwise.
- $height(s)$ gives the number of items in s.

3.1. **Safe relocations.** We introduce the concept of *conflict*: a conflict occurs when a given item is positioned above at least one item of smaller index. In other words, when an item blocks at least one other item. At any given stage of the solution process, $NR + LB$ gives us a lower bound on the final solution quality. Keeping that in mind, we focus on relocations that do not increase $NR+LB$, which we call *safe*.

We define safe 1-relocates as relocations that involve moving an item i from stack s to stack t such that $i < lowest(t) \wedge i > lowest(s)$. A safe 1-relocate adds 1 to NR while subtracting 1 from LB, the end result being that $NR + LB$ stays the same. When several safe 1-relocates can be performed and we need to decide which one is the best, we use a heuristic rule similar to what is found in LA. More precisely, the safe 1-relocate of item i from stack s to stack t has an impact $d_{it} = lowest(t) - i$, and the safe 1-relocate that minimises d_{it} is considered to be the best for heuristic purpose. We note here that safe 1-relocates are similar to the BG moves described in Forster and Bortfeldt [7].

We also define safe 2-relocates as relocating item i from s to t such that (i) there is no possible safe 1-relocate, (ii) $i < lowest(t)$ (i.e. i does not block any other item once relocated), (iii) this relocation creates a possibility to safely 1-relocate another item. Safe 2-relocates are safe with LB_2 but not with LB_1, which is too myopic. A safe 2-relocate is characterised by moving item i from s to t in order to allow to safely 1-relocate item j to s. Such a relocation is only a safe 2-relocate if $i < lowest(t) \wedge j < lowest(s \setminus \{i\})$. The impact of this safe 2-relocate is then defined as $d'_{ijst} = lowest(s \setminus \{i\}) - j + lowest(t) - i$. Again, the heuristically best safe 2-relocate is the one that minimises impact.

3.2. **Decreasing sequences.** We note here that none of the moves described above allows to solve the instance described in Figure 1 optimally. Similarly, no fast heuristic from the literature allows to find it. In the following we propose a new type of move aimed at dealing with that case.

We consider the case where s is the stack of the next item to retrieve, with $top(s) > lowest(s)$, i.e. $top(s)$ needs to be relocated. We look at the decreasing sequence $DS(s)$ of consecutive elements starting from $top(s)$ downward that does not include $lowest(s)$. In the case of Figure 1, this sequence is $DS(2) = (5, 4, 3)$. If there is a stack t where $top(s)$ can be safely 1-relocated, then the whole sequence can also be safely 1-relocated there, assuming there is enough height available. If not, then it may still be interesting to make room so that it becomes possible. For example in Figure 1 the optimal solution involves relocating item 2 to stack 0 then relocating $DS(2)$ to stack 1. Given $DS(s)$ and a target stack t where we want to relocate it, we can estimate the cost of enabling such a relocation by evaluating the impact on LB_1. In order to do that, we first determine the sequence $u \in t$ of items which have to be relocated so that the modified stack $t' = t \setminus u$ can receive $DS(s)$ without creating any conflict, i.e. such that $lowest(t') > top(s) \wedge height(t') + |DS(s)| \leqslant H$.

The impact of this relocation is estimated as $\sum_{i \in u} 1 - lb(i) + sr(i)$. In other words, relocating an item that needed to be relocated anyway is rewarded while relocations that create conflicts are penalised. The sequence of relocations that minimises impact is considered to be the best one.

3.3. **Combinations of these operators.** We propose a general heuristic scheme that considers the following types of operations, in this order of priority:

(1) Retrieve items that can be retrieved without any relocation
(2) Perform voluntary moves if possible
(3) Perform forced moves

Step 1 is obvious and step 3 is done in a similar fashion as with the LA heuristic as explained above, so all that remains to specify is step 2. The operators introduced above have different time complexities and aim at dealing with different situations. We apply them using different heuristics for voluntary moves.
SM-1. iteratively performs the best safe 1-relocate until there are no safe 1-relocates
SM-2. iteratively tries to perform the best safe 1-relocate; if there is none, it tries to perform the best safe 2-relocate
DSEQ. iteratively tries to perform the best decreasing sequence relocate
SmSEQ-1. iteratively tries to perform the best safe 1-relocate; if there is none, it tries to perform the best decreasing sequence relocate
SmSEQ-2. iteratively tries to perform the best safe 1-relocate; if there is none, it tries to perform the best safe 2-relocate; if there is none, it tries to perform the best decreasing sequence relocate

3.4. **Rake search: a new constructive metaheuristic framework.** In preliminary experiments, we made the observation that the first steps taken in constructing a solution, i.e. the first relocations, are crucial to the quality of the final solution. Conversely, if the first steps are well taken, it is not very hard to find a good solution afterwards. This was also observed when trying to build solutions by hand. We now propose a solution framework that exploits this aspect of the BRP.

We present a new constructive metaheuristic framework, inspired by beam search and tree search in general, which we call *rake search*. It relies on the idea of completing a partial solution with a quick construction heuristic. Rake search starts as a breadth-first tree search procedure, generating level after level of the tree. Once a level that has w nodes or more is reached, the tree search stops and each node is used as a starting point for various construction heuristics. The best solution found is kept.

In our case, at any given node of the search tree, we start by retrieving all the items that can be retrieved without requiring any relocation. Once this is done, many relocations are possible, each giving rise to a different branch of the search tree. Some of these are generated, thus producing the next level of the search tree. More precisely, we generate all safe 1-relocates and all safe 2-relocates. We also generate all possible forced moves, i.e. relocating the item on top of the stack where the next item to retrieve is, to any other stack.

Once a level of at least w nodes is reached, each of the nodes (partial solutions) is used as a starting point for each of heuristics SM-1, SM-2, SmSEQ-1, SmSEQ-2. Assuming there are exactly w nodes before this step, $4w$ solutions are constructed in total. In practice there are more than that since we produce a whole level of

the search tree before stopping, instead of stopping as soon as there are exactly w nodes.

4. Exact solution of the BRP

We develop a simple branch-and-bound algorithm for the BRP. For any given node of the branch-and-bound tree, i.e. state of the stacks or partial solution, we first retrieve all items that can be retrieved without any relocation. At this point, LB_1 (see Section 3) gives us a valid lower bound on the final cost (total number of relocations) of the solution. Given a valid upper bound UB, obtained for instance by using one of the heuristics mentioned above, a node of the branch-and-bound tree can be pruned as soon as $NR + LB \geqslant UB$. The successors of a given node are then obtained by exploring the various possible relocations. In order to speed up the search we introduce a look-ahead mechanism that assesses the changes in NR and LB induced by a relocation, and only generates the new node if it cannot already be pruned using UB. This avoids unnecessary memory allocations of many nodes of the branch-and-bound tree. In order to create a new node, implementation possibilities are either (i) copying the parent node then modifying the copy, which involves memory allocation, or (ii) storing the sequence of branching decisions at each node and applying them to the starting solution (root node) when processing a node, then reversing them, which involves a lot of writing to memory. Either way, our look-ahead mechanism allows to reduce the amount of these costly operations.

We note here that if the tree is explored in a best-first fashion and disregarding our look-ahead mechanism, our algorithm is in fact equivalent to the A* algorithm proposed by Expósito-Izquierdo et al. [6]. However we find that running a depth-first search yields better results.

We also tried using LB_2 by Forster and Bortfeldt [7] since it is a stronger lower bound than LB_1. It actually decreases the performance of our algorithm strongly. The reason is that in most cases $LB_1 = LB_2$, while in a few cases only it holds that $LB_2 = LB_1 + 1$. However LB_2 takes longer to compute and overall, the benefit does not outweigh the cost of this computation.

This branch-and-bound algorithm can easily be adapted to solve the restricted BRP: instead of trying every possible relocation at a given node, we only try relocating the item on top of the stack where the next item to be retrieved is. Therefore we also implement a version of our algorithm for the restricted BRP.

5. Experiments

Our algorithms are implemented using the Python programming language and run with PyPy, which is usually faster than the vanilla Python interpreter but still typically slower than other compiled languages such as C or Java. The CPU is an Intel Xeon Processor E5-2670 v2 (25M Cache, 2.50 GHz) with a 4-GB RAM limit. The operating system is Linux. We note here that the RAM limitation only makes a difference for our best-first B&B algorithm. For the depth-first B&B all runs were completed using an initial limit of 1GB, which was never reached. We increased the limit to allow the best-first version to be compared to the A* algorithm from Expósito-Izquierdo et al. [5], which uses 4 GB, once we noticed that the best-first B&B was sometimes running out of memory. The best-first B&B still runs out of memory in many cases. The heuristics have very low memory requirement.

Although there exist many contributions using the same set of instances, a direct comparison of all of them is not possible for two reasons:

- Some studies consider the restricted BRP, others the unrestricted BRP. The restricted BRP considers additional constraints so the solutions that are optimal for the unrestricted BRP are not necessarily feasible for the restricted BRP.
- Even though the instances are the same, the height limit H_{max} is not always the same. Some authors consider that there is no height limit, some consider it to be $2H - 1$, where H is the height of the initial configuration, and others use $H + 2$.

5.1. **Heuristics for the unrestricted BRP.** We first compare our work to contributions on the unrestricted BRP. In a first step of experiments, we compare it to the LA-N heuristic of Petering and Hussein [12]. We implemented their LA-N heuristic and use some of its components for our heuristics. Rake search is run with $w = 50$ and its results are reported in column RS-50. Each heuristic is deterministic and is run once on each test instance. Instances are grouped into classes whose name is HxS, where H is the height of each stack in the initial configuration and S is the number of stacks. Each class has 40 instances. For these runs, $H_{max} = 2H - 1$. Table 1 summarises these results. Aside from rake search, every run finishes instantly. In the worst case rake search takes less than five seconds so there is little point in comparing CPU times.

Rake search consistently provides the best results, at the cost of a few seconds of CPU time. Among the "fast heuristics", i.e. all except rake search, SM-2 provides the best results on average, suggesting that safe relocations are a good idea. Except for the smallest instance size, one of our heuristics always obtains the best result among all fast heuristics. DSEQ is the worst method overall, suggesting that relocating decreasing sequences is not necessarily relevant. We note here that even though these instances are of a size matching currently cited applications of the BRP, they are all rather small.

In a second set of experiments, we compare our methods to the tree search from Forster and Bortfeldt [7] (TS) as well as the knowledge-based heuristic of Expósito-Izquierdo et al. [5] (KB). The authors from both contributions provide results for two settings, unlimited H_{max} and $H_{max} = H + 2$, so we provide results for these two settings as well.

It appears that the results from both previous contributions are sometimes erroneously rounded. For instance, for $H_{max} = H + 2$ and instance class 3x4, Expósito-Izquierdo et al. [5] report an average objective value of 6.02. Since it is an average over 40 instances, this means that the sum of objective values over these 40 instances is 241, which is on average 6.025, which should be rounded to 6.03 and not 6.02. This kind of mistake appears a few times and we suppose it is due to limitations in floating point number representation on computers. We report here the corrected values.

Those results are summarised in Tables 2 and 3. Rake search (RS) provides more often than not a better solution than TS or KB, while keeping CPU effort low (again below five seconds for the worst case). Both TS and KB take between 0 and 60 seconds depending on the cases. For unlimited H_{max}, RS provides better

HxS	LA-1	LA-2	LA-3	LA-S-1	RS-50	SM-1	SM-2	DSEQ	SmSEQ-1	SmSEQ-2
3x3	5.08	5.08	5.08	5.08	**4.95**	5.10	5.12	6.22	5.17	5.20
3x4	6.30	6.25	6.28	6.28	**6.03**	6.25	6.22	7.17	6.28	6.22
3x5	7.05	7.00	6.95	6.97	**6.85**	6.92	7.00	7.47	6.92	6.95
3x6	8.45	8.40	8.47	8.47	**8.28**	8.45	8.38	8.97	8.40	8.35
3x7	9.32	9.28	9.22	9.25	**9.12**	9.22	9.25	9.53	9.22	9.25
3x8	10.72	10.60	10.65	10.82	**10.30**	10.70	10.70	10.88	10.62	10.60
4x4	10.90	10.43	10.38	10.38	**9.80**	10.22	10.25	12.75	10.53	10.47
4x5	13.55	13.32	13.22	13.03	**12.28**	12.93	12.72	14.38	12.93	12.97
4x6	14.45	14.30	14.15	14.03	**13.40**	14.05	13.82	15.20	13.95	13.95
4x7	16.62	16.43	16.40	16.38	**15.43**	16.20	15.93	16.80	15.95	16.00
5x4	16.45	15.85	15.75	15.75	**14.68**	15.72	15.60	19.70	16.07	16.05
5x5	20.25	20.00	19.77	19.70	**17.95**	19.32	19.23	23.05	19.60	19.73
5x6	23.52	23.07	22.98	22.55	**21.20**	22.18	21.85	25.27	22.02	22.07
5x7	25.68	25.25	25.23	24.77	**22.75**	24.18	23.50	26.38	24.02	23.73
5x8	28.65	28.60	28.27	27.82	**25.90**	27.52	27.07	29.93	27.15	27.12
5x9	31.85	31.35	31.18	30.70	**28.65**	30.18	29.82	31.93	30.18	30.18
5x10	34.50	34.10	33.62	33.45	**31.18**	32.62	32.02	34.30	32.48	32.25
6x6	34.15	33.55	32.90	32.48	**28.82**	31.82	31.12	35.77	31.60	31.35
6x10	48.65	48.45	47.60	46.62	**42.25**	45.30	44.20	48.58	45.60	44.95
10x6	89.75	86.85	86.70	84.95	**72.75**	85.17	81.28	98.35	83.70	82.78
10x10	126.75	125.62	123.75	119.95	**101.38**	113.83	109.03	128.03	114.03	112.15

Table 1: Performance of our heuristics compared to $LA-N$ methods on benchmark instances from Caserta et al. [3]. $H_{max} = 2H - 1$.

results than the other two methods in 12 cases while KB does so on 5 cases and TS never does. For $H_{max} = H + 2$, RS provides better results in 15 cases while KB does so in 2 cases and TS never does. For $H_{max} = H + 2$, both KB and TS seem to be hitting a limit around instances of size 10x10: they are systematically better than all of our fast heuristics for all instance sizes except for that one, which happens to be the largest one. This is also the only case where TS reaches its CPU budget limit of one minute. This decrease in performance on larger instances also applies for TS when H_{max} is unlimited. Overall RS provides the best performance and also does not hit any computational limit for instances of this size.

5.2. **Scalability of the fast heuristics.** One issue regarding the existing data sets is that they are relatively small. In order to assess the robustness and scalability of our fast heuristics, we now provide experimental results on larger data sets with up to 100 stacks and 10,000 items. Since we implemented the LA-N heuristic from Petering and Hussein [12], we can compare it with our heuristics. For $S \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$ we generate 40 random instances with S stacks and S^2 items, uniformly distributed among those stacks in a random order. We then compare our fast heuristics as well as the LA-N using those instances. We do not include the heuristics by Forster and Bortfeldt [7] and Expósito-Izquierdo et al. [5] since they already hit a limit for 10 stacks and 100 items. H_{max} is set to $2H - 1$. Table 4 reports average values for this experiment. Table 5 reports

HxS	KB	TS	RS-50	SM-1	SM-2	DSEQ	SmSEQ-1	SmSEQ-2
3x3	**4.95**	**4.95**	**4.95**	5.03	5.03	6.30	5.10	5.13
3x4	**6.03**	6.05	**6.03**	6.23	6.18	7.00	6.20	6.15
3x5	6.88	**6.85**	**6.85**	6.90	6.95	7.45	6.90	6.90
3x6	**8.28**	**8.28**	**8.28**	8.35	8.30	8.98	8.33	8.30
3x7	**9.10**	9.20	9.13	9.20	9.23	9.43	9.15	9.20
3x8	10.38	10.45	**10.30**	10.60	10.60	10.85	10.55	10.55
4x4	**9.70**	9.90	9.78	10.13	10.15	12.35	10.40	10.35
4x5	12.30	12.63	**12.28**	12.80	12.68	14.13	12.80	12.85
4x6	**13.38**	13.70	13.40	13.90	13.75	15.10	13.85	13.85
4x7	15.60	15.78	**15.43**	16.15	15.93	16.70	15.90	15.95
5x4	**14.68**	15.00	14.70	15.88	15.65	18.85	16.03	16.03
5x5	18.03	18.63	**17.95**	19.38	19.23	22.63	19.60	19.65
5x6	**21.08**	21.83	21.20	22.13	21.85	25.00	21.98	21.98
5x7	23.28	23.58	**22.70**	24.08	23.43	26.28	23.98	23.65
5x8	26.65	26.73	**25.90**	27.40	26.98	29.83	27.00	27.00
5x9	29.40	29.45	**28.63**	30.03	29.60	31.95	30.00	29.90
5x10	31.70	31.85	**31.18**	32.53	32.03	34.08	32.30	32.08
6x6	28.98	29.68	**28.85**	31.78	31.08	35.63	31.43	31.28
6x10	42.45	43.60	**42.18**	45.18	43.95	48.48	45.50	44.65
10x6	76.03	75.65	**72.70**	84.78	81.48	97.23	84.08	83.05
10x10	104.03	116.90	**101.03**	113.25	108.60	127.73	114.25	112.35

Table 2: Performance of our heuristics compared to the tree search of Forster and Bortfeldt [7] (TS) and the knowledge-based heuristic of Expósito-Izquierdo et al. [5] (KB). Unlimited H_{max}.

the corresponding average CPU times in seconds. First of all, we note that methods SM-2, DSEQ, SmSEQ-1 and SmSEQ-2 are much slower when instance size increases. However a CPU effort of around a minute to solve an instance with 10,000 items is still reasonable, especially with an interpreted language. Second of all, there are several hints that relocating whole decreasing sequences is in fact a good idea when instance size increases: above size 70x70, all the methods using this type of relocation provide better solutions than all other methods. For instances of size 30x30 and above, DSEQ always provides better solutions than all LA-N methods. Moreover, the methods providing the best results overall are very clearly SmSEQ-1 and SmSEQ-2, which do consider relocating decreasing sequences, even though they only do it when no safe relocate is found. That appears to be the winning combination, as it is also faster than DSEQ, probably due to the fact that evaluating how to relocate decreasing sequences is more CPU intensive than evaluating safe relocates.

5.3. **Exact methods.** The only two exact methods for the unrestricted BRP that we are aware of are model BRP-III from Petering and Hussein [12] and the A* algorithm by Expósito-Izquierdo et al. [5]. BRP-III was applied to instances with up to nine containers only, and is likely unable to solve bigger instances. We compare our branch-and-bound (B&B) to the results of the A* for the unrestricted

HxS	KB	TS	RS-50	SM-1	SM-2	DSEQ	SmSEQ-1	SmSEQ-2
3x3	4.98	4.98	**4.95**	5.10	5.13	6.23	5.18	5.20
3x4	**6.03**	6.05	**6.03**	6.25	6.23	7.18	6.28	6.23
3x5	6.88	**6.85**	**6.85**	6.93	7.00	7.48	6.93	6.95
3x6	**8.28**	**8.28**	**8.28**	8.45	8.38	8.98	8.40	8.35
3x7	9.15	9.23	**9.13**	9.23	9.25	9.53	9.23	9.25
3x8	10.38	10.40	**10.30**	10.70	10.70	10.88	10.63	10.60
4x4	**9.78**	9.93	9.83	10.40	10.33	12.90	10.75	10.73
4x5	**12.33**	12.65	**12.33**	13.15	12.93	14.73	13.13	13.13
4x6	**13.35**	13.70	13.43	14.13	13.93	15.33	14.05	14.03
4x7	15.48	15.78	**15.43**	16.35	16.00	16.93	16.03	16.05
5x4	15.45	15.53	**14.90**	16.33	16.13	19.98	16.85	16.75
5x5	18.68	18.80	**18.13**	19.60	19.40	23.93	19.90	20.00
5x6	21.63	22.08	**21.38**	22.53	22.25	26.75	22.68	22.70
5x7	23.45	23.58	**22.95**	24.78	24.15	27.50	24.65	24.38
5x8	26.45	27.03	**26.18**	27.95	27.58	30.40	27.73	27.70
5x9	29.13	30.05	**28.83**	31.05	30.55	33.10	31.03	30.98
5x10	31.90	32.25	**31.35**	33.60	33.10	35.60	33.15	32.98
6x6	30.40	31.13	**29.58**	32.80	32.33	39.60	33.45	33.18
6x10	44.08	44.48	**43.15**	47.10	45.80	50.63	47.05	46.63
10x6	85.45	83.03	**78.10**	90.85	88.10	112.00	93.08	93.38
10x10	121.50	125.38	**106.88**	121.28	115.98	144.70	123.50	121.73

Table 3: Performance of our heuristics compared to the tree search of Forster and Bortfeldt [7] (TS) and the knowledge-based heuristic of Expósito-Izquierdo et al. [5] (KB). $H_{max} = H + 2$.

HxS	LA-1	LA-2	LA-3	LA-S-1	SM-1	SM-2	DSEQ	SmSEQ-1	SmSEQ-2
10x10	125.97	124.50	122.30	116.85	115.67	**109.03**	123.65	112.45	110.65
20x20	747.60	747.12	740.60	708.83	673.55	644.25	751.60	636.30	**628.88**
30x30	2123.50	2124.90	2133.28	2051.05	1955.95	1821.17	2047.22	1768.40	**1750.20**
40x40	4426.10	4444.65	4450.65	4399.00	4252.95	3840.62	4237.40	3642.45	**3607.75**
50x50	7808.62	7855.60	7864.48	7962.77	7665.48	6901.95	7463.55	6443.65	**6418.12**
60x60	12584.92	12702.10	12726.30	13159.80	12667.17	11286.65	11616.35	10094.73	**10068.88**
70x70	18537.42	18611.47	18756.85	19907.25	19188.22	16886.00	17069.22	14766.45	**14752.75**
80x80	26058.65	26102.70	26084.70	28286.62	27218.92	24219.75	23681.83	20483.53	**20466.65**
90x90	35114.90	35214.97	35327.53	39008.00	37701.43	33162.03	31850.03	27136.70	**27107.78**
100x100	45891.32	46202.40	46443.53	52205.05	50259.47	44245.45	40930.12	35176.25	**35137.70**

Table 4: Performance of our heuristics compared to $LA - N$ methods on large randomly generated instances.

BRP of Expósito-Izquierdo et al. [5]. They consider two settings: unlimited H_{max} and $H_{max} = H + 2$. We provide results for both settings. The B&B starts with an upper bound which is the best solution obtained by our five fast heuristics. We report results for two versions of our branch-and-bound algorithm: depth-first and best-first. The A* from Expósito-Izquierdo et al. [5] runs for 24 hours, whereas our

HxS	LA-1	LA-2	LA-3	LA-S-1	SM-1	SM-2	DSEQ	SmSEQ-1	SmSEQ-2
10x10	0.00	0.00	0.01	0.01	0.01	0.02	0.02	0.02	0.02
20x20	0.00	0.00	0.01	0.01	0.02	0.05	0.06	0.03	0.04
30x30	0.01	0.01	0.01	0.02	0.05	0.17	0.33	0.14	0.15
40x40	0.03	0.03	0.03	0.05	0.13	0.60	1.42	0.51	0.57
50x50	0.06	0.06	0.06	0.09	0.29	1.69	4.44	1.46	1.59
60x60	0.10	0.11	0.11	0.18	0.61	4.06	11.18	3.55	3.81
70x70	0.17	0.19	0.20	0.34	1.21	9.33	25.55	7.81	8.34
80x80	0.29	0.30	0.31	0.53	2.06	18.59	52.55	15.42	16.41
90x90	0.42	0.44	0.47	0.81	3.27	33.19	98.60	27.74	29.34
100x100	0.63	0.66	0.68	1.23	5.14	57.80	174.21	49.46	51.86

Table 5: CPU effort of our heuristics compared to $LA - N$ methods on large randomly generated instances.

HxS	A*	depth-first	best-first
3x3	**40**	**40**	**40**
3x4	**40**	**40**	**40**
3x5	**40**	**40**	**40**
3x6	**40**	**40**	**40**
3x7	**40**	**40**	**40**
3x8	**40**	**40**	**40**
4x4	**40**	**40**	**40**
4x5	**40**	**40**	**40**
4x6	**40**	**40**	38
4x7	5	**39**	37
5x4	**40**	**40**	33
5x5	25	**37**	21
5x6	1	**30**	15
5x7	1	**25**	16
5x8	0	**17**	11
5x9	0	**13**	6
5x10	0	**9**	7
6x6	0	**5**	0
6x10	**0**	**0**	**0**
10x6	**0**	**0**	**0**
10x10	**0**	**0**	**0**

Table 6: Results of our depth-first and best-first branch-and-bound algorithms versus A* method from Expósito-Izquierdo et al. [5]. $H_{max} = H + 2$. Each row represents the number of instances solved within the allotted time budget, per instance size.

branch-and-bound algorithm only runs for one hour. These results are reported in Tables 6 and 7, where each line indicates how many instances of a certain size can be solved by each algorithm. The depth-first branch-and-bound is dominating the other two methods. This being said, our best-first algorithm also provides better

HxS	A*	depth-first	best-first
3x3	**40**	**40**	**40**
3x4	**40**	**40**	**40**
3x5	**40**	**40**	**40**
3x6	**40**	**40**	**40**
3x7	**40**	**40**	**40**
3x8	**40**	**40**	**40**
4x4	**40**	**40**	**40**
4x5	**40**	**40**	**40**
4x6	**40**	**40**	38
4x7	5	**39**	37
5x4	**40**	**40**	34
5x5	25	**36**	23
5x6	1	**31**	18
5x7	1	**26**	21
5x8	0	**20**	14
5x9	0	**16**	8
5x10	0	**13**	12
6x6	0	**6**	2
6x10	**0**	**0**	**0**
10x6	**0**	**0**	**0**
10x10	**0**	**0**	**0**

Table 7: Results of our depth-first and best-first branch-and-bound algorithms versus A* method from Expósito-Izquierdo et al. [5]. Unlimited H_{max}. Each row represents the number of instances solved within the allotted time budget, per instance size.

performance than the A* overall, with a few exceptions. This is noteworthy because they are basically the same algorithm and our branch-and-bound only runs for one hour instead of 24 hours for the A*. There is also the fact that all our code is in Python, which is typically slower than Java, the language with which the A* was implemented. The only algorithmic difference between the two algorithms is the look-ahead mechanism when evaluating new successors, and apparently it is enough to make up for a much shorter CPU budget as well as a supposedly less efficient implementation. Using better starting upper bounds may also help B&B.

We now evaluate the quality of our algorithm when it is modified to solve the restricted BRP. Results are available from Expósito-Izquierdo et al. [6] and from Zehendner et al. [15]. The results from Expósito-Izquierdo et al. [6] do not allow a very interesting comparison since a small subset of instances is considered and they can all be solved within a few seconds. We observe a similar behaviour on these instances. The instance that requires the highest CPU effort among all considered is 4-6-1, which requires 17.476 seconds for the branch-and-bound algorithm by Expósito-Izquierdo et al. [6] and 11.819 seconds for our B&B. All other instances are solved in less than a second in Expósito-Izquierdo et al. [6] and in less than 2 seconds by our B&B. We can provide more detailed results in comparison with the

HxS	MIP	depth-first B&B
3x3	**40**	**40**
3x4	**40**	**40**
3x5	**40**	**40**
3x6	**40**	**40**
3x7	**40**	**40**
3x8	**40**	**40**
4x4	**40**	**40**
4x5	**40**	**40**
4x6	**40**	**40**
4x7	**40**	**40**
5x4	**40**	**40**
5x5	**39**	**39**
5x6	33	**36**
5x7	24	**32**
5x8	9	**28**
5x9	5	**19**
5x10	2	**17**
6x6	7	**10**
6x10	**2**	1
10x6	**0**	**0**
10x10	**0**	**0**

Table 8: Restricted BRP: Results of our depth-first branch-and-bound algorithms versus MIP approach method from Zehendner et al. [15]. $H_{max} = H + 2$. Each row represents the number of instances solved within the allotted time budget, per instance size.

mixed-integer programming (MIP) approach of Zehendner et al. [15], since the authors report how many instances are solved over all 40 instances of each size. Their approach also has a time limit of one hour. They also report the average CPU time required to solve *non-trivial* instances, where trivial instances are those for which the initial lower bound is equal to the initial upper bound. However, our approach uses different bounds than theirs and instances that are trivial for the MIP are not necessarily trivial for our B&B and vice-versa. For that reason we only report the number of instances solved within the allotted time. They only consider the setting where $H_{max} = H + 2$, so we only report results for this setting. Since previous experiments showed that our depth-first B&B is better, this is the one we use for that experiment. Table 8 reports these results. Overall, the depth-first B&B solves more instances than the MIP approach of Zehendner et al. [15] within the same time budget. There is one case, for size 6x10, where the MIP solves 2 instances and our B&B only solves one.

6. Conclusions and perspectives

In this report we have presented several contributions for the block relocation problem. Even though most of the existing work concerns the restricted BRP, we

focused our work on the unrestricted BRP, which yields more opportunities for optimisation. We first introduced fast heuristics that provide solutions for instances of realistic size within an instant. We then integrated them into a new construction metaheuristic framework, named rake search, in order to provide even better solutions within a few seconds. We also showed that when considering larger instances, our fast heuristics provide a very significant improvement over existing fast methods, with around 30% improvement for the largest instances considered, all within a few seconds. This experiment also emphasised the value of considering sequences of items with increasing priorities, which allows to conduct less relocations overall by relocating such a sequence as a unique entity. We also developed a branch-and-bound algorithm which is the best exact approach to date for the unrestricted BRP. This allowed us to compute previously unknown optimal solutions for a number of instances. This algorithm could possibly be improved by computing better combinatorial bounds, which is a direction for future research. We also modified this algorithm to solve the restricted BRP, which allowed us to provide results that are better than those from the literature.

Acknowledgements

The work described in this paper was done within the COMET Project Heuristic Optimization in Production and Logistics (HOPL), #843532 funded by the Austrian Research Promotion Agency (FFG). The authors also want to thank Richard F. Hartl, Sophie N. Parragh and Martin Romauch for their valuable feedback.

References

[1] C. Aydin and T. Ünlüyurt. Improved rehandling strategies for the container retrieval process. *Journal of advanced transportation*, 46(4):378393, 2012.
[2] The World Bank. http://data.worldbank.org/indicator/is.shp.good.tu/, 2014.
[3] M. Caserta, S. Voss, and M. Sniedovich. Applying the corridor method to a blocks relocation problem. *OR Spectrum*, 33(4):915–929, 2011.
[4] M. Caserta, S. Schwarze, and S. Voss. A mathematical formulation and complexity considerations for the blocks relocation problem. *European Journal of Operational Research*, 219(1):96–104, 2012.
[5] C. Expósito-Izquierdo, B. Melin-Batista, and J.M. Moreno-Vega. A domain-specific knowledge-based heuristic for the blocks relocation problem. *Advanced Engineering Informatics*, 28(4):327 – 343, 2014.
[6] C. Expósito-Izquierdo, B. Melin-Batista, and J.M. Moreno-Vega. An exact approach for the blocks relocation problem. *Expert Systems with Applications*, 42(1718):6408 – 6422, 2015.
[7] F. Forster and A. Bortfeldt. A tree search procedure for the container relocation problem. *Computers & Operations Research*, 39(2):299 – 309, 2012.
[8] R. Jovanovic and S. Voss. A chain heuristic for the blocks relocation problem. *Computers & Industrial Engineering*, 75:79 – 86, 2014.
[9] K.H. Kim and G.-P. Hong. A heuristic rule for relocating blocks. *Computers & Operations Research*, 33(4):940 – 954, 2006.
[10] Y. Lee and Y.-J. Lee. A heuristic for retrieving containers from a yard. *Computers & Operations Research*, 37(6):1139 – 1147, 2010.

[11] J. Lehnfeld and S. Knust. Loading, unloading and premarshalling of stacks in storage areas: Survey and classification. *European Journal of Operational Research*, 239(2):297 – 312, 2014.

[12] M. Petering and M. Hussein. A new mixed integer program and extended look-ahead heuristic algorithm for the block relocation problem. *European Journal of Operational Research*, 231(1):120 – 130, 2013.

[13] S. Tanaka and K. Takii. A faster branch-and-bound algorithm for the block relocation problem. In *Automation Science and Engineering (CASE), 2014 IEEE International Conference on*, pages 7–12, Aug 2014.

[14] K.C. Wu and C.J. Ting. A beam search algorithm for minimizing reshuffle operations at container yards. In *Proceedings of the 2010 international conference on logistics and maritime systems*, Busan, Korea, 2010.

[15] E. Zehendner, M. Caserta, M. Feillet, S. Schwarze, and S. Voss. An improved mathematical formulation for the blocks relocation problem. *European Journal of Operational Research*, 245(2):415 – 422, 2015.

Graph representation of the block relocation problem

As mentioned in Section 1, any BRP instance is associated to a graph where the nodes correspond to various states of the bay and the arcs represent possible transitions between those states. A given arc has a weight of 1 if it corresponds to a relocation, while removing an item from the bay is associated to a null weight. We also consider the initial state of the bay as the source node and the state associated to an empty bay as the sink node. Solving the BRP is then equivalent to finding the shortest path from source to sink. Without loss of generality, we consider that if a given state allows to remove the next item to be removed, then that is the only transition that is allowed from that state (node).

We now illustrate this representation with a small instance considering 3 items and 3 stacks. The first stack contains items 1 and 3, the second stack contains items 2, the third stack is empty. This instance is represented in Figure 2, where the source is leftmost and the sink is rightmost.

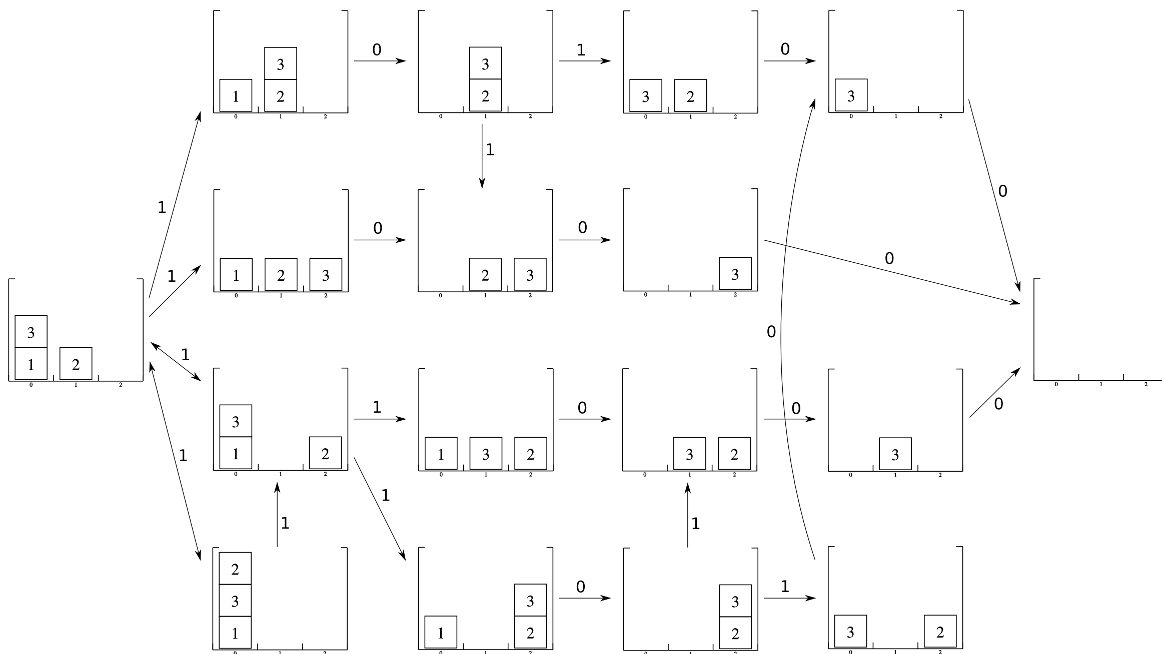

Figure 2: Full graph for a small instance of the BRP with 3 items and 3 stacks.