Transportation Research Part B 139 (2020) 102–131

Contents lists available at ScienceDirect

Transportation Research Part B

journal homepage: www.elsevier.com/locate/trb

Machine learning-driven algorithms for the container relocation problem

Canrong Zhang [a,b], Hao Guan [a,b], Yifei Yuan [a,b], Weiwei Chen [c], Tao Wu [d,*]

[a] Research Center for Modern Logistics, Shenzhen International Graduate School, Tsinghua University, Shenzhen 518055, China
[b] Department of Industrial Engineering, Tsinghua University, Beijing 100084, China
[c] Department of Supply Chain Management, Rutgers Business School-Newark and New Brunswick, Rutgers University, New Jersey 08854, United States
[d] School of Economics & Management, Tongji University, Shanghai 200092, China

ARTICLE INFO

Article history:
Received 27 August 2019
Revised 1 March 2020
Accepted 24 May 2020

Keywords:
Container relocation
branch-and-bound algorithm
beam search
machine learning-driven technique

ABSTRACT

The container relocation problem is one of important issues in seaport terminals which could bring a significant saving on the operating cost even with a slight improvement due to the huge number of containers processed across the world each year. Given a specific layout and container retrieval priorities, the container relocation problem aims to find the optimal movement sequence to minimize the total number of container relocation operations. In this paper, we propose novel machine learning-driven algorithms, which integrate optimization methods and machine learning techniques, to solve the problem. More specifically, we propose a new upper bound method called MLUB that incorporates branch pruners. These pruners are derived from some machine learning techniques through using the optimal solution values of many small-scale instances. The tightened upper bounds generated by MLUB are used subsequently both in the exact branch-and-bound algorithm called IB&B and the hybrid beam search heuristic called MLBS. Moreover, we also provide a tighter lower bound for the problem by additionally considering the interaction between consecutive target containers. Based on the benchmark data published recently in the literature, extensive experiments are conducted to test the performance of the proposed algorithms. The experimental results demonstrate that the proposed algorithms outperform the state-of-the-art algorithms reported in the literature, and some managerial insights regarding the load intensity of the bay and some algorithm parameters such as the look-ahead depth and the beam width are drawn from the results.

1. Introduction

Globalization facilitates a steady rise in the volume of international trade, and the global logistics network is undertaking more and more pressure from the demanding requirement. As an important part of the network, seaborne transportation accounts for more than 60% of the trade amount and maintains a high growth rate in recent years. As a consequence, the seaport container terminal, a critical hub in the seaborne transportation, is facing increasing pressure from the carriers, shippers, and consignees in terms of providing higher-efficiency and more-flexibility services.

* Corresponding author.
E-mail address: danielwu9999@gmail.com (T. Wu).

https://doi.org/10.1016/j.trb.2020.05.017
0191-2615/

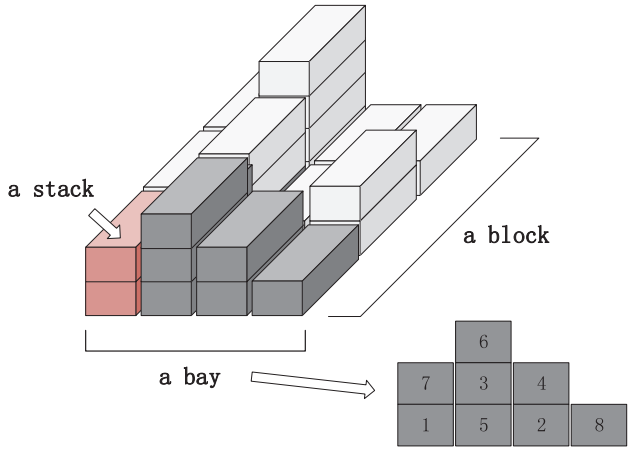

Fig. 1. An illustration of a bay.

Roughly speaking, the operations on the container terminals include four parts: receiving containers for temporary storage before the arrival of destined vessels, the discharging operation, the loading operation, and the pickup operation by consignees to take their containers away from the terminal. This paper focuses on the loading operation, and, more specifically, restricts our focus on the relocation operation during loading. For a more comprehensive understanding of the container terminal operation, the interested readers are referred to Steenken et al. (2004) and the papers surveyed therein.

Before starting the loading operation, containers are transported to the terminal by water or land for temporary storage. To increase the utilization of the yard space, a stack is formed by storing containers vertically, and a horizontal row of stacks forms a bay, while a number of adjacent bays make up a block as shown in Fig. 1. Due to some loading rules required by the stowage plan, for example, heavier containers required to be stacked under lighter ones when loaded onto the vessel, each container will be given a retrieval priority during the loading operation. As the arrival of containers is uncontrollable, containers with higher retrieval priorities are often buried beneath those with lower priorities, leading to the relocation operation inevitable. This paper focuses exactly on this kind of relocation operations occurring during the loading operation.

To be more specific, we assign a smaller number to the container with a higher retrieval priority, and vise versa. For example, as shown in Fig. 1, container 1 should be retrieved first, container 2 should be retrieved next, and so on. When we try to retrieve one container, there may be two cases: if the target container happens to be on the top of its stack, then we can just perform the retrieval operation; otherwise, if some containers are stored on the top of the target one, then the relocation operation is required. For example, as container 7 is stored on the top of container 1, it should be relocated to other stacks. Therefore, the container relocation problem (CRP) studied in this paper aims to give an optimized operation sequence for retrieving containers from a bay with the goal of minimizing the relocation operations as much as possible.

As such relocation operations are time-wasting and unproductive, a method which can generate an operation sequence with fewer relocation operations is appealing due to a huge number of containers handled across the world each year. In addition, the relocation problem distinguishes itself as a difficult one in the combinatorial field because the relocated container causes a chain effect on the subsequent retrieval operations. Due to these two considerations, this paper attempts to propose a novel method, which integrates optimization methods and machine learning techniques, to solve the problem, and validates the method using the benchmark test instances reported on the recently published literature. Accordingly, this paper makes the following contributions:

1. Valuable information regarding the promising nodes selection (called the branch pruner in the later section) is extracted first and then incorporated into a heuristic method called MLUB to generate tight upper bounds. To the best of our knowledge, it is the first attempt to integrate optimization methods and machine learning techniques in the algorithm design for the CRP;
2. An exact branch-and-bound algorithm which relies on the MLUB to determine the branching precedence is proposed. Contrary to using the best lower bound to select the next node to explore, this exact algorithm adopts the best upper bound to select the next node to explore;
3. To handle large-scale instances, we also propose a beam search algorithm that makes use of the MLUB in its beam node selection. In addition, some screening techniques are also proposed and used in the searching process to further enhance its convergence;
4. Based on the benchmark data, the numerical experiments reveal that our proposed algorithms outperform the state-of-the-art algorithms reported in the literature, and some managerial insights regarding the load intensity of the bay and some algorithm parameters such as the look-ahead depth and the beam width are drawn from the results.

The remainder of this paper is organized as follows. Section 2 reviews the related works. Section 3 defines the container relocation problem. Sections 4 and 5 describe the proposed exact branch-and-bound algorithm and the beam search heuris-

tic, respectively, with their machine learning-driven branching pruners presented in Section 6. Results of the numerical experiments are presented in Section 7. Finally, conclusions and suggestions for future research are given in Section 8.

2. Related work

As an essential intermodal interface in the international shipping network, container terminals are crucial to the efficient circulation of cargoes and play an important role in the global competition. In recent years, with more and more research studies dedicated to the quay side of container terminals (Zhen et al., 2017; Wang et al., 2018; Iris and Lam, 2019; Iris et al., 2018), increased pressure has transferred from quay side to yard side. As such, the yard optimization urgently needs to be addressed.

Yard optimization primarily consists of two parts: the yard crane planning and container handling, at the operational level. As summarized in Zehendner et al. (2016), the former delves into the yard crane assignment/scheduling problem to assign yard cranes to storage areas and to define a schedule of storage and retrieval operations for each crane; the latter focuses on the optimization for the handling process of containers, including the storage allocation problem to determine storage locations for incoming containers, the remarshalling/premarshalling problem to reorganize parts of the storage area (a block or a bay) in less busy periods, and the container relocation problem to retrieve all containers from a bay in a given sequence with a minimum number of relocations. For the specific introduction and review of the three problems, we refer the interested readers to the works given by Bortfeldt and Wäscher (2013), Carlo et al. (2014) and Lehnfeld and Knust (2014).

Plenty of studies have been performed on the aforementioned two parts of yard optimization. For the yard crane planning, Zhang et al. (2002) is among the first few research studies that addressed the crane deployment problem, which was formulated as an MIP model and solved by an improved Lagrangean relaxation method. On this basis, the single-crane scheduling (Gharehgozli et al., 2014b; Zheng et al., 2018) and the scheduling with crane interference (Gharehgozli et al., 2014a; Virgile et al., 2018) are widely studied. For the container handling, most of the research studies focus on the storage allocation problem to determine storage locations for incoming containers. Under two cases with different objectives, the container allocation problem was studied in Kim and Kim (2002). In order to assign containers to routes with minimum cost, Bell et al. (2011), Bell et al. (2013) and Wang et al. (2015) studied frequency-based, cost-based and profit-based container assignment problems under maritime liner shipping networks, respectively. Based on Kim et al. (2000) and Zhang et al. (2010), Zhang et al. (2014a) considered the stack height and state-changing magnitude information when interpreting the punishment parameter, and proposed two new “conservative” allocation models for outbound containers in container terminals. Zhang et al. (2014b) presented two new dynamic programming models for the container allocation problem with additional consideration of adjusted probability on the types of arriving containers. By contrast, there are relatively few research studies dedicated to the remarshalling/premarshalling problem (Lee and Chao, 2009; Hottung et al., 2020). The review on the CRP will be presented separately a moment later. In addition, abundant studies were conducted for the integration of the yard crane planning and container handling, for example, Zhang et al. (2003), Zhen (2016) and Jiang and Jin (2017). And there are also some research studies that studied from the strategic and tactical points of view, for example, Liu et al. (2016) and Zhen (2016). Since strategic and tactical decisions are not the focus of this paper, we refer the interested readers to the review by Lee and Song (2017) for more information.

As mentioned previously, the focus of this paper is the container relocation problem. In the following, we will focus on the review on this topic. As an NP-hard problem (Caserta et al., 2012), the CRP is firstly studied by Kim and Hong (2006), in which they proposed a branch-and-bound method to solve the CRP to optimality. As branch-and-bound algorithms can be time-consuming for large-instances, they established a heuristic using “expected number of additional relocations” (ENAR) to determine the stack for relocated containers with the goal to minimize the number of relocations during container loading. Caserta et al. (2011) proposed a “two-dimensional” corridor method, in which a dynamic programming algorithm is used as a subroutine, to tackle the exponential growth of the number of states for large-scale problems, and the validity of this method was proved by comparing with Kim and Hong (2006) in numerical experiments. Forster and Bortfeldt (2012) proposed a heuristic tree search procedure for the CRP that consists of four elements. Zhu et al. (2012) developed iterative deepening A* algorithms to solve the CRP. They presented three different lower bounds (LB) calculation methods and four probe heuristics embedded in the algorithms. The computational results show that their method can solve most cases to the optimum in a time suitable for practical application. On this basis, Tanaka and Takii (2016) proposed a new LB calculation method called LB4 and proved that LB4 is greater than or equal to LB3 established by Zhu et al. (2012). In this paper, we will provide a tighter lower bound, based on LB4, by additionally considering the interaction between consecutive target containers.

Jin et al. (2015) constructed an improved greedy look-ahead heuristic for the unrestricted CRP, which can be split into three levels. The top level keeps executing the relocation and retrieving operation until the layout is empty. The stack to take in the relocated container is given by the middle level using limited tree search and the bottom level applies several heuristic rules to evaluate the leaf nodes generated by the middle level. Jovanovic and Voß (2014) proposed a chain heuristic, based on the classical Min-Max method proposed by Caserta et al. (2011), considering both the relocated container and the container to be relocated next. Ting and Wu (2017) improved the Min-Max heuristic and proposed a virtual relocation heuristic (VRH), which took all the blocking containers above the target container into account at the same time. Galle et al. (2018) improved upon and enhanced an existing binary encoding and using it, formulated the restricted CRP as a binary integer programming problem in which they exploited structural properties of the optimal solution. Later,

Jin (2020) identified two deficiencies contained in the model of Galle et al. (2018) in formulating the "last in, first out" (LIFO) policy, and provided the corrected integer programming formulation. Jovanovic et al. (2019) presented an ant colony optimization algorithm for the CRP, which can be applied to both the restricted and unrestricted CRP and outperform existing methods for the CRP. Bacci et al. (2019) proposed a new beam search approach, called the Bounded Beam Search algorithm, to solve the CRP and compared it with other methods in the literature, showing that it outperforms most of the existing approaches.

As machine learning techniques are incorporated in the design of our algorithm, we also give a brief review on this field. Machine learning is usually used in learning (Siripirote et al., 2015; Wei and Liu, 2013; Allahviranloo and Recker, 2013), estimation (Yin et al., 2015; Zheng and Zuylen, 2013) and prediction (Andres and Nair, 2017) of parameters, as well as to enhance the modeling process (see, e.g., Hofleitner et al. (2012), Sopasakis and Katsoulakis (2016), Kouvelas et al. (2017), Arentze and Timmermans (2004), and Dong et al. (2018)).

The study which adopts machine learning techniques for the container location-related problems is very scant in the literature. Hottung et al. (2020), to the best of our knowledge, is the only published report which proposed a new method called Deep Learning Heuristic Tree Search (DLTS) for the container pre-marshalling problem (CPMP). In the paper, the deep neural networks are used to learn solution strategies through analyzing existing (near-) optimal solutions to CPMP instances, and then the networks are integrated into a tree search procedure to decide which branch to choose next and to prune the search tree. Although Hottung et al. (2020) and our paper utilizes the similar idea, their differences are evident and can be summarized as follows: (i) The CRP studied in this paper considers the retrieval of containers, while the CPMP studied in Hottung et al. (2020) does not; (ii) In terms of the method for generating the feasible solutions (that is, upper bounds in both papers), our method combines a learning mechanism and a heuristic method to yield a feasible solution, while there are no heuristics involved in Hottung et al. (2020) which relies heavily on the deep neural networks; (iii) Hottung et al. (2020) proposes a heuristic method without considering the global optimality, while our paper further incorporates the heuristic into the exact branch-and-bound framework to generate optimal solutions; (iv) The deep neural networks proposed in Hottung et al. (2020) encounter the scalability issue, while this paper handles the scalability issue well.

In summary, different types of optimization algorithms including the exact and heuristic ones were proposed in the literature to address the CRP, and several types of machine learning techniques such as the decision tree, the neural network and so on were adopted to parameter estimation and system optimization in a wide range of application. In the CRP, however, the algorithms that combines the optimization methods and the machine learning techniques are very scant. Inspired by the fact that more valuable information regarding the structure of the optimal solution can be extracted and facilitates the solution for large-scale instances, we are motivated to make such an attempt in this regard.

3. Problem description

Generally, the container relocation problem can be divided into two categories by two classification criteria: (i) the restricted and unrestricted problems; (ii) the problem with distinct priorities and the one with duplicate priorities. The former classification depends on the range of containers that can be chosen to be relocated. If only the blocking containers right above the target container can be relocated, it is a restricted CRP; otherwise, it is called unrestricted CRP. The latter classification is determined by the uniqueness of retrieval priorities. The problem with distinct priorities implies a unique retrieval order of containers, and the other specifies precedence relations between groups of containers. In this paper, we consider the restricted CRP in which each container is given a distinct retrieval priority. More specifically, the problem is to determine, arrange and adjust the horizontal and vertical positions of the containers in a bay, in order to obtain an optimal sequence of operations for retrieving containers according to a given retrieval priority, with the objective of minimizing the number of relocations.

For the problem, we make the following assumptions:

1. The container layout and retrieval priority are given beforehand;
2. All containers are uniform in length and height, and they can be described in unit size;
3. There exists a stack height limit for the consideration of security;
4. The relocation operation only occurs within the same bay due to the consideration that the horizontal movement of yard cranes carrying containers is time-consuming;
5. The retrieval priority of each container is unique, and there is no priority shared by multiple containers, i.e., this paper studies the problem with distinct priorities;
6. To better fit the worker's operational convenience, only the containers right above the target container can be relocated, i.e., the restricted CRP is studied in this paper.

4. The exact algorithm (IB&B)

In this section, we propose an exact algorithm, which is essentially an iterative branch-and-bound algorithm (IB&B), for the CRP, and present the lower bound (LB) generating method called LB4 and the upper bound (UB) generating method called MLUB, respectively.

4.1. The framework of the algorithm

Before presenting the framework of the IB&B, we first give a brief discussion on the classical branch-and-bound algorithm (CB&B). The basic idea of the CB&B is dividing and conquering which exploits the relationship between the local lower bound (with respect to the node currently under consideration) and the global upper bound (with respect to the original problem for a minimization objective) to trim some unpromising nodes and facilitate the convergence. To abate confusion and ease exposition, the bounds without any modifiers such as “global” or “local” refer to the global ones throughout the paper, while the bounds with “global” or “local” refer to those with respect to the original problem and the local node, respectively. As mentioned in Morrison et al. (2016), three components can be fine-tuned to improve the performance of the algorithm: the branching strategy, the search strategy, and the pruning rules. In the following, we will present them one by one.

The branching strategy represents how the present node is partitioned to produce new nodes to expand the tree. For the container relocation problem we study, the branching process is straightforward. Given a specific layout of the bay under consideration, if the relocation operation is inevitable, which means the target container is buried under those with lower retrieval priority (called blocking containers), then the blocking containers need to be relocated sequentially from the top down. For the first blocking container, we should initially identify the set of all the stacks that can accommodate the blocking container, and then make the decision about reposition. To be more specific, let c denote the first blocking container that is on the top of the stack where the target container is buried. Then, the set of all stacks that can accommodate the blocking container c can be denoted by $CS(c) = \{s | s \in S, s \neq s(c), h(s) < H\}$, where S denotes the set of stacks indexed by s, $s(c)$ denotes the stack where container c is originally located, $h(s)$ represents the height of stack s, and H is the stack height limit. As a result, we can relocate container c to each stack in the set of $CS(c)$, and update the bay state accordingly, thereby generating $|CS(c)|$ states (or called nodes) in total. In the following, the nodes and states will be used interchangeably. From each newly generated node, the branching process can be restarted for the second blocking container (if any) until the target container is exposed to the top.

The search strategy refers to how to expand the searching tree consisting of the nodes generated by the branching strategy mentioned previously. Under the breadth-first search strategy (BFS), the searching tree is expanded uniformly with the frontier moving forward evenly. Under this strategy, there is no notable difference regarding which node will be treated first, thereby making its implementation straightforward and easy, but it has a significant drawback that is a feasible solution will not be revealed until the frontier cannot move forward any further. To circumvent this drawback, this paper intends to adopt the depth-first search strategy (DFS) which explores from the selected node and goes forward as deep as possible, and backtracks to the most recently visited node if it cannot proceed any further. For this search strategy, which node will be investigated first makes a significant difference. The previous studies usually used the LB as the criterion, that is, the node with the best lower bound will be chosen to be investigated first. It is called the LB-priority strategy in this paper. This strategy may misguide the search process especially when the quality of the lower bound is low. Although the upper bound also suffers the same drawback, a tighter upper bound is more likely to be a good indicator for a better neighborhood. Therefore, in this paper, rather than using the lower bound, we attempt to utilize the UB as the criterion to select the next node. This strategy is called the UB-priority strategy. The comparison between the two strategies will be conducted through numerical experiments in Section 7.3.3.

With respect to the pruning strategy, a conventional way is adopted by exploiting the relationship between the local LB and the global UB. More specifically, there are primarily two situations: (i) if the local LB and UB are equal at one node, then with no need to explore further, this node has arrived at the local optimality and can be used to update the global UB of the original problem whenever possible; (ii) if the local LB of one node plus the number of relocation operation executed so far is greater than or equal to the incumbent global UB of the original problem, then the node certainly cannot produce a better solution and will be pruned.

Similar to Tanaka and Takii (2016), the framework of the IB&B is shown in Algorithm 1 , with the embedded branch-and-bound algorithm (EB&B) showed in Algorithm 2 . Given the layout of a yard bay and the retrieval priority for each container,

Algorithm 1: IB&B.

```
Input: L ← initial layout
LB ← lower bound of L by LB4
UB ← upper bound of L by MLUB
while LB < UB do
    Call EB&B with designated LB and UB
    if there exists feasible solution = LB then
        An optimal solution is found
        break
    end
    Update UB with the better UB obtained from the execution of EB&B (if any)LB = LB + 1
end
return LB
```

Algorithm 2: EB&B.

Input: $L \leftarrow$ initial layout; $LB \leftarrow$ lower bound of L by LB4; $UB \leftarrow$ upper bound of L by MLUB
LayoutList $\leftarrow$ null
FindingFeasibleSolutionFlag $\leftarrow 0$
Identify all relocation choices for L, perform these relocation operations to generate child states, and sort these child states by their upper (lower) bounds for the UB-priority (LB-priority) version of the EB&B
Add L to *LayoutList*
while *LayoutList.size*$! = 0$ **do**
  Get the next operation to perform of the most recently visited layout in *LayoutList*
  if *no more operations to perform* **then**
    Remove the most recently visited layout in *LayoutList*
  end
  else
    Generate a new layout $L1$ by performing the pre-specified relocation operation
    Identify all relocation choices for $L1$, perform these relocation operations to generate child states, and sort these child states by their upper (lower) bounds for the UB-priority (LB-priority) version of the EB&B
    $lb \leftarrow$ lower bound of $L1$ by LB4
    $ub \leftarrow$ upper bound of $L1$ by MLUB
    if $ub = lb$ *and* $lb + LayoutList.size = LB$ **then**
      $FindingFeasibleSolutionFlag = 1$
      break
    end
    else if $ub! = lb$ *and* $lb + LayoutList.size \leq LB$ **then**
      if $ub + LayoutList.size < UB$ **then**
        Update UB by $ub + LayoutList.size$
      end
      Add $L1$ to *LayoutList*
    end
  end
end
if $FindingFeasibleSolutionFlag = 1$ **then**
  return True
end
else
  return False
end

it is able to obtain an LB and an UB (whose generation algorithms will be presented in Sections 4.2 and 4.3, respectively). Different from the CB&B, the EB&B is partially implemented in the framework to search for a specific target in each calling. Initially, the global LB is set as the search target, and all the nodes that satisfy either one of the following conditions will be removed: (i) arriving at the local optimality with an optimal value not equal to the search target; (ii) becoming unpromising when compared to the LB. That is to say, only the nodes with the local LB plus the number of relocation operation executed so far smaller than or equal to the current target value are retained, with the aim to reduce the searching effort. If no feasible solution equal to the target value is found after one search, the global UB will be replaced with the newly found best UB (if any), and the global LB will be increased by 1, then the implementation of the EB&B with the updated global LB and UB will be performed again until the feasible solution is found or the global LB is equal to the global UB. Since IB&B terminates when the global LB equals the global UB, the method generates optimal solutions and is an exact algorithm.

4.2. The method for calculating lower bounds (LB4)

To calculate lower bounds, our proposed algorithms hybridize the LB4 method proposed by Tanaka and Takii (2016), which takes into consideration the combinations of destination stacks for the blocking containers above the target one. We refer the interested readers to Tanaka and Takii (2016) for its details. We notice that, in LB4, the blocking containers on a target container are independent of those of the next (remaining) target container. Actually, the blocking containers of the first target container affect the relocation of the blocking containers of the next (remaining) target container. That is to say, they are interdependent. We, therefore, are able to propose a tighter lower bound called the enhanced LB4 (or ELB4 for short) which is based on LB4 but additionally considers the interaction between two or more consecutive retrieval of target containers. The underlying idea of ELB4 can be illustrated through Fig. 2.

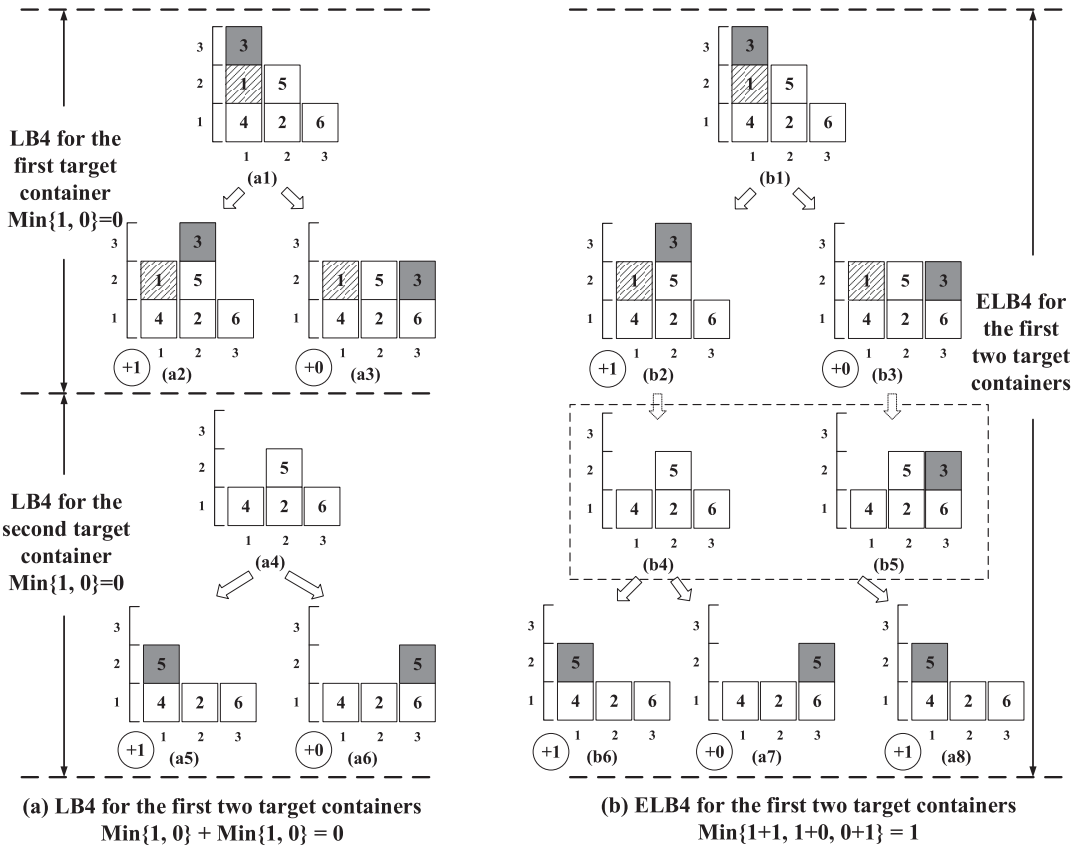

Fig. 2. Comparison between LB4 and ELB4.

As shown in Fig. 2(a), according to LB4, the initial bay status (a1) branches into (a2) and (a3), and then the calculation for LB4 corresponding to the first target container (that is, container 1) is terminated. To calculate the LB4 for the second target container (that is, container 2), container 1 and all the blocking containers above it (that is, container 3) are removed, thereby generating the initial bay status for container 2, which corresponds to (a4), and then the LB4 for container 2 can be calculated. Herein, the lower bounds of relocation numbers required for the first two target containers are calculated independently, and the sum of them are equal to 0. Under ELB4, however, we reserve the container whose priority changes the priority status of the accommodating stack, corresponding to Fig. 2(b5). Starting from (b4) and (b5), we further conduct the LB4 process for container 2, and obtain a tighter LB for the first two containers, which is $ELB4 = 1$. Herein, ELB4 obtains a better lower bound than LB4 because it considers the interaction between two consecutive target containers. We omit the proof process of this claim due to its straightforwardness.

Fig. 2 only demonstrates the interaction between two consecutive target containers. Actually, the interaction can be extended to 3 or more consecutive target containers. For simplicity, in this paper, we only consider the interaction between two consecutive target containers. Although ELB4 is tighter than LB4, our preliminary experiments show that the ELB4 method takes much longer time than LB4 while the improvement on the lower bounds is not that significant relative to the extra time consumed, as discussed in Section 7.1. Considering that the LB needs to be calculated at each node in the branch-and-bound tree, we still use the LB4 method for our proposed algorithms. It is noted that ELB4 is a good choice when high-quality lower bounds are required and long runtime is allowed, and, for the particular case, the number of consecutive target containers whose interactions are considered simultaneously needs to be set to a larger value.

4.3. The method for calculating the machine learning-driven upper bounds (MLUB)

In this section, we propose a machine learning-driven method for the upper bound generation. The method is based on the virtual relocation heuristic (VRH) proposed by Ting and Wu (2017) which in turn is based on the Min-Max method proposed by Caserta et al. (2011). For completeness, the explanation on VRH is given in Appendix based on a given example, while the Min-Max method is proposed below as some definitions are needed in the later sections. Moreover, a more unified formula is proposed in this paper to replace the original one to simplify the implementation. After that, the MLUB for generating the upper bound will be presented.

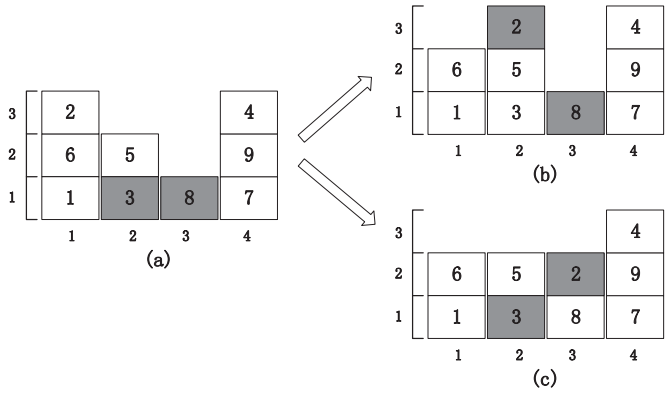

Fig. 3. Selection principle for Type-1 stacks.

Let us first define the priority number of a non-empty stack s as $p(s) = \min\{p(c)|s(c) = s\}$, where $p(c)$ is the retrieval priority of container c and the priority number of the stack is expressed as the highest priority (i.e., the smallest priority number) among all the containers belonging to that stack. If stack s is empty, $p(s)$ is set to $N + 1$. Assume c denotes the container to be relocated, and then the available stacks to accommodate the relocated container can be divided into two categories according to whether $p(s)$ is larger than $p(c)$ or not. For the stack whose priority number is larger than $p(c)$, relocating the container onto it will not cause future relocation, then we call such a relocation a well-relocated one and call the corresponding stack a Type-1 stack. On the contrary, if the future relocation of the relocated container is inevitable, we call the accommodating stack a Type-2 stack.

The Min-Max method first checks the existence of the well-relocated stacks when choosing a stack to accept the relocated container. On the one hand, if there is more than one well-relocated stack, the stack with the highest priority will be chosen, that is, $s^* = arg\ \min\{p(s)|s \in S, p(s) > p(c), h(s) < H\}$, where S and H denote the set of stacks and the stack height limit, respectively. This selection policy is aimed at reserving more well-relocated stacks for the future blocking containers. As shown in Fig. 3(a), the current target container is 1 and the container to be relocated is container 2. Under the assumption that the stack height is limited to 3, the feasible receiving stacks are stacks 2 and 3, both of which are well-relocated ones. If container 2 is relocated to stack 2, stack 3 remains Type-1 stack for the next blocking container 6 as shown in Fig. 3 (b). If container 2 is relocated to stack 3, however, neither the two stacks meet the well-relocated condition for next relocation and container 6 is bound to be relocated at least twice as shown in Fig. 3(c).

On the other hand, if none of the available stacks meets the well-relocated condition, the stack with the lowest priority is selected, that is, $s^* = arg\ \max\{p(s)|s \in S, p(s) < p(c), h(s) < H\}$. This selection policy tries to put aside the relocated container so that it has no or little impact on the retrieval for the immediately subsequent several target containers. In summary, under the Min-Max rule, if there exist stacks with lower priority than that of the container to be relocated, the one with the highest priority will be chosen; otherwise, the stack with the lowest priority will be chosen. We can unify the policy for the two scenarios by introducing a new stack priority of stack s denoted by $p^*(s)$ as follows.

$$p^*(s) = \begin{cases} p(s) - p(c) & if\ p(s) > p(c) \\ 2N + 1 - p(s) & otherwise \end{cases} \tag{1}$$

where N is the total number of containers at the beginning, $p(s)$ is the highest container priority of stack s and set to $N + 1$ when the stack is empty. The use of $2N + 1$ ensures that the $p^*(s)$ value of stacks satisfying the well-relocated condition is always smaller than that of stacks not satisfying the condition. With the introduction of $p^*(s)$, the aforementioned two policies applied to two different scenarios have been unified by adopting the smallest number of $p^*(s)$ no matter whether they are Type-1 or Type-2 stacks.

The basic idea of the MLUB is shown in Fig. 4(b) which is essentially an enhanced version of the look-ahead method. As shown in Fig. 4(a), we expand from node 1 to get a complete set of searching branches for the first D layers. Assume that at the Dth layer, there are a total number of K states. And then by using the VRH method proposed by Ting and Wu (2017), we can get at most K upper bounds (denoted by ub), among which, $\min\{ub\} + D$ will be set as the UB of the initial node. It is evident that, with the increase of D, the calculation time of this method increases exponentially. Accordingly, we are motivated in this paper to eliminate some branches which are unlikely to be selected in the D layers in terms of generating good-quality UBs in order to reduce the searching efforts. To the end, a branch pruner (or called filter) extracted from the machine learning-driven methods will be proposed to prune unpromising branches and expedite the convergence. As shown in Fig. 4(b), if branch $1 \rightarrow 4$ is pruned by the filter in the first layer, and branches $2 \rightarrow 7$ and $3 \rightarrow 9$ are pruned in the second layer, the number of nodes to calculate UBs will decrease dramatically. To the best of our knowledge, it is the first paper attempting to increase the searching efficiency of the algorithm by adopting the learning mechanism for the CRP. How to generate the filter by the learning mechanism will be introduced in Section 6.

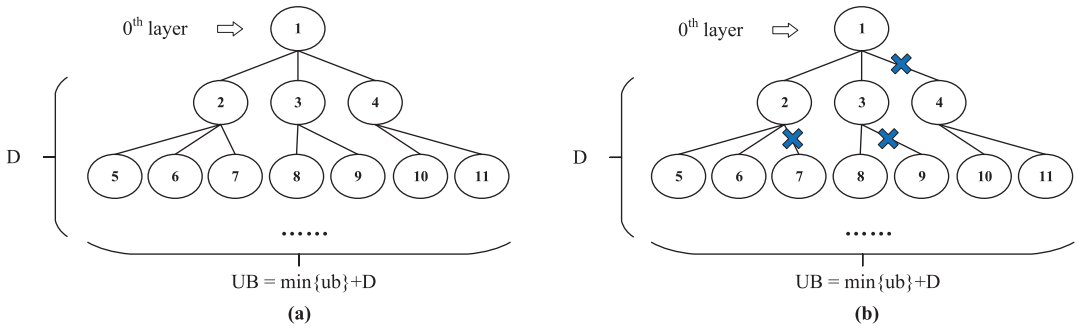

Fig. 4. (a): Illustration of the look-ahead method without branch pruners; (b) Illustration of the look-ahead method with branch pruners.

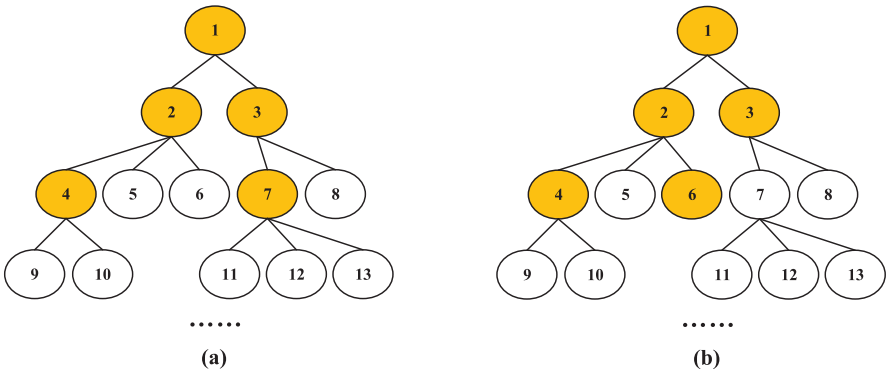

Fig. 5. Illustration of the MLBS.

5. The beam search algorithm (MLBS)

Even though the exact algorithm has been enhanced as presented in the previous section, it still encounters difficulty when solving large-scale instances. We, therefore, have to resort to heuristic algorithms to handle such large-scale instances. By well utilizing the branching process provided by the branch-and-bound algorithm, we propose the MLBS in the following section.

5.1. The framework of the algorithm

Beam search is a classical heuristic method, which is similar to the breadth-first tree search, with the difference lying in the number of nodes to explore. More specifically, the beam search strictly restricts the number of reserved nodes equal to or smaller than the beam width at each level to reduce the searching space. The reserved nodes at each level are called beam nodes and only the beam nodes can create their successors while others will be discarded. There are two types of beam search algorithms in terms of how to generate succeeding beam nodes: the interdependent one and the independent one. The former one generates only one descendant from each beam node while the latter one selects next-generation beam nodes from all “legal” descendant nodes. As shown in Fig. 5, suppose the beam width is 2, node 1 can branch into nodes 2 and 3, node 2 has 3 successors, and node 3 has 2 successors. Fig. 5(a) and (b) schematically illustrate the node selection process by the interdependent and independent beam searches, respectively.

In this paper, we use the independent beam search method, that is, all the descendent nodes in each layer are put together and compared with each other to screen out the beam nodes. The pseudocode of the MLBS is shown in Algorithm 3 . In the algorithm, the evaluation function is particularly important as it directly determines which node will be left for the subsequent exploration and which will be discarded immediately. We adopt the UB obtained by the MLUB presented in Section 4.3 as the evaluation function. The smaller the UB, the better the quality of the node, and the more likely it is to get a better solution. The LB of each node as calculated by the LB4 presented in Section 4.2 is also calculated at each level to discard unpromising nodes with the help of the UB. In addition, a set of screening techniques presented in Section 5.2 will also be used to further reduce the candidate nodes for beam node selection. The global UB will be updated whenever a better UB is found. Once the global UB equals LB, the optimal solution has been found and the search terminates. Another termination condition is when the beam node set becomes empty in one certain level, UB will be the final solution provided by the algorithm.

Algorithm 3: MLBS.

```
Input:LB ← lower bound of L by LB4
UB ← upper bound of L by MLUB
if LB = UB then
    return UB
end
else
    Retrieve the target container until the target container is inaccessible
    L ← the layout after retrieving until the target container is inaccessible
    i ← 0; C_i ← ∅; flag ← 1
    Add L to C_i
    while flag = 1 do
        i = i + 1
        for each beam node in beam node set C_i do
            Get the blocking container B on the top of the target container
            for each optional stack do
                Relocate B to the stack and get a new layout L1
                lb ← lower bound of L1 by LB4
                ub ← upper bound of L1 by MLUB
                if lb + i < UB then
                    Add the new layout to beam node set C_(i+1)
                end
                if ub + i < UB then
                    UB = ub + i
                end
                if UB = LB then
                    flag = 0
                end
            end
        end
        Delete duplicate nodes, sort the remaining nodes in C_(i+1) by their upper bounds in an ascending order, and
        only retain the first W nodes
        for each beam node in beam node set C_(i+1) do
            Retrieve the target container until the target container is inaccessible
        end
        if C_(i+1) is empty then
            flag = 0
        end
    end
    return UB
end
```

5.2. Screening techniques

It is evident that the quality of the beam nodes has a significant impact on the quality of the algorithm, which makes it necessary for us to provide high-quality node pool for selection. To this end, a set of screening techniques is presented as follows.

1. Calculate the UB and LB of the node simultaneously. If the UB equals the LB, then this node needs not to be added into the candidate set for beam nodes selection. In this case, the node has arrived at the local optimality, with no need to explore further, and can be used to update the global UB if possible;
2. If the LB of one node plus the number of relocation performed so far is greater than or equal to the current global UB, this node cannot produce a better solution, and therefore will not enter the candidate set for beam nodes selection;
3. During the execution, some beam nodes might be exactly the same in a certain layer as shown in Fig. 6, in which the stack limit is set to 3, and states (f) and (g) in the figure are actually the same. Therefore, before sorting the nodes, we will perform a de-duplication screening to ensure that all the nodes in the candidate set are distinctive;
4. In order to deal with the situation when there are multiple nodes with the same UB in the candidate set, we propose to select the node whose relocation leads to the smallest $p^*(s)$ value defined by Eq. (1), contrary to the random selection adopted in Ting and Wu (2017).

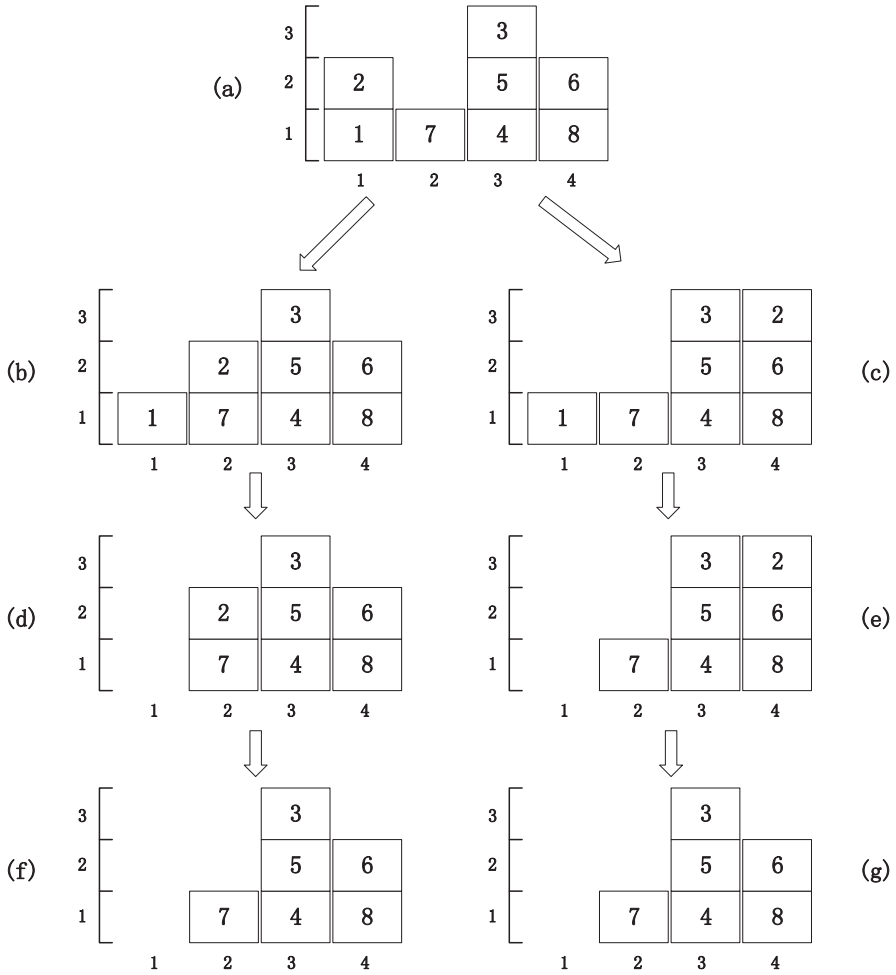

Fig. 6. Illustration of the same beam nodes.

6. The machine learning-driven branch pruners

In this section, we will present the machine learning-driven branch pruners, called filters for short, to facilitate the convergence of the upper bound.

6.1. Adopted datasets

The datasets chosen to generate exact solutions for filter generation generally need to meet the following two conditions: (i) the instances must be small-scale, so that the exact solution can be relatively easy to obtain; (ii) the instances must be representative and diversified as well, so that the extracted knowledge, to a large extent, will be meaningful and useful. Based on the aforementioned conditions, we choose, from Zhu et al. (2012), the datasets of the form $H-S-N$, including 3-7-18, 3-7-19, 3-7-20, 3-8-21, 3-8-22, 3-8-23, 3-9-24, 3-9-25, 3-9-26, 3-10-27, 3-10-28, 3-10-29, 4-6-20, 4-6-21, 4-6-22, 4-6-23, 4-7-24, 4-7-25, 4-7-26 and 4-7-27, where H is the stack height limit, S is the total number of stacks, and N is the initial number of containers, which varies between $H(S-1)$ to $HS-1$. Besides, we choose, from Caserta et al. (2011), the datasets of the form $T-S$, including 3-3, 3-4, 3-5, 3-6, 3-7, 3-8, 4-4, 4-5, 4-6 and 4-7, where T is the stack height, S is the number of stacks, and the number of containers is by default TS, meaning that each stack contains T containers at the initial state. There is no stack height limit in the original dataset in Caserta et al. (2011), while in order to make it more consistent with the reality and comparable with the recent research, we assume the stack height limit is the initial stack height plus 2, i.e., $H=T+2$. The dataset from Zhu et al. (2012) represents the situation where the bay is relatively full and randomly placed, while the dataset from Caserta et al. (2011) considers the situation where the bay is relatively empty and evenly placed. Each dataset in Zhu et al. (2012) contains 100 instances and each dataset in Caserta et al. (2011) consists of 40 instances, leading to a total number of $20\times 100+10\times 40=2400$ instances. The exact solutions of the aforementioned

small-scale instances will be generated by enumeration, represented as the optimal operation sequences, from which the filter will be constructed.

6.2. Random forest based branch pruner

The random forest is a classifier that utilizes multiple decision trees to train and predict samples. As the basis of the decision tree, information entropy is an index to quantitatively describe uncertainty from the perspective of probability theory. The more chaotic the system, the greater the uncertainty of the system, the greater the information entropy, and vice versa. Classification by reasonable properties, however, can effectively reduce the chaos of the whole system, thereby reducing the information entropy of the system. In view of this observation, given a dataset that contains data objects with properties describing the data objects, we can extract the classification effectiveness under each property, primarily due to the identification of the change ratio of information entropy relative to the initial status without classification. There are multiple algorithms such as ID3 and C4.5 supporting the implementation of random forest. Since the relative change in the entropy rather than the absolute change is considered in this paper, the C4.5 algorithm which focuses more on the information gain ratio better fits our case, and, therefore, is adopted in this paper. Accordingly, the property leading to the most effective classification is firstly selected as the current dividing point of the decision tree. Then we exclude this property and repeat the process at the frontier node in the current decision tree until the establishment of the whole tree is completed. Different decision trees are established for the samples drawn by the replacement sampling method, and then aggregated to a random forest, where the modal number of the results under all decision trees in the random forest will be adopted as the final prediction output of the random forest.

Based on the random forest, the proposed branch pruner is actually a classifier which classifies stacks into several categories, and only keeps the target category of stacks and discards the rest to achieve the goal of pruning. Accordingly, each relocation in the optimal operation sequence of the instances introduced in Section 6.1 corresponds to a data object, and we need to find reasonable properties to describe the data object. To this end, we first need to define the representation of the bay for each configuration of containers piled up in the stacks. With the definition, we are able to identify the optional stacks for each container to be relocated, among which, the best stack will be chosen and the corresponding index will be recorded. It should be noted that when defining the representation of the bay, the information on the size of the bay should not be incorporated, as the information on size may prevent it from being extended to larger-scale instances. To represent the bay, we introduce the following two indicators, both of which are unitless so that the size-independent requirement can be met.

1. $R1(s)$: the ratio to represent the relationship between the selection preference of stack s and the total number of optional stacks that can accommodate the container to be relocated;
2. $R2(s)$: the ratio of the number of the blocking containers to the total number of containers in the same stack s.

Taking Fig. 7 as an example, the current target container is container 1 and container 9 is to be relocated. With the assumption that the stack height limit is 3, the set of optional stacks to accommodate container 9 is S_3={2, 4, 5, 6}, and $p^*(2) = 18$ (as $p^*(2) = 2N + 1 - p(2) = 2 * 12 + 1 - 7 = 18$ according to the definition of $p^*(s)$ as shown in Eq. (1)), $p^*(4) = 17$ (whose calculation is the same as $p^*(2)$), $p^*(5) = 1$ (as $p^*(5) = p(5) - p(c) = 10 - 9 = 1$ according to Eq. (1)) and $p^*(6) = 19$ (whose calculation is the same as $p^*(2)$), then we can get a stack selection sequence by sorting the $p^*(s)$ values in an ascending order, i.e. {5, 4, 2, 6}. Given the stack selection preference ranking, we can calculate $R1(s)$ for each stack as: $R1(2) = 2/3,\ R1(4) = 1/3,\ R1(5) = 0/3 = 0,\ R1(6) = 3/3 = 1$.

The calculation of $R2$ is straightforward. As shown in Fig. 7, considering container 12 is a blocking container in stack 2, and the total number of containers in stack 2 is 2, thereby $R2(2) = 1/2$. Following the same fashion, we can calculate $R2(4) = 0,\ R2(5) = 1/2,$ and $R2(6) = 0$.

To validate the significance of the two indicators, we calculate the values of $R1(s)$ and $R2(s)$ for different stacks in the 30 datasets introduced in Section 6.1, and then we observe the average values, with respect to the two indicators, of the stacks that are chosen in the exact solution (labeled 1 as shown in Figs. 8 and 9), and those that are not in the exact solution (labeled 0). It is evident that the average value of stacks labeled 1 are always below those of stacks labeled 0, which implies that the two indicators are effective to differentiate the selected stacks from the unselected ones.

With the previous description, we are able to describe a stack and further a bay. As a bay is composed of many stacks, we can divide stacks according to the two ratios $R1(s)$ and $R2(s)$ and count the number of stacks belonging to different stack types to describe the bay. To this end, our problem turns out to determining two dividing points, or thresholds, for the two ratios, respectively. The average of the two ratios for the 30 datasets in terms of different stack types (as before, 0 represents the stack not chosen in the exact solution, and 1 represents the chosen stack), are shown in Fig. 10.

According to the distribution of Fig. 10, we choose 0.5 as the dividing point for $R1(s)$, so that the pruner can cut off at least half of the candidate branches, and 1/3 as the dividing point for $R2(s)$. As a consequence, the stacks can be divided into $2 \times 2 = 4$ categories according to their ratio values against their respective dividing points. For each bay, we assign the stacks to different stack categories, and add up the number of stacks belonging to different categories to generate the representation of the bay. More specifically, an array of 1×4 named TEMP is introduced to represent the bay, with the first element counting the number of stacks with $R1(s)$ within [0,0.5] and $R2(s)$ within [0,1/3], the second element corresponding to $R1(s)$ within [0,0.5] and $R2(s)$ within (1/3,1], the third to $R1(s)$ within (0.5,1] and $R2(s)$ within [0,1/3], and the last one to

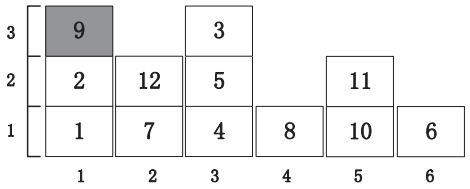

Fig. 7. Illustration of the tie breaking principle.

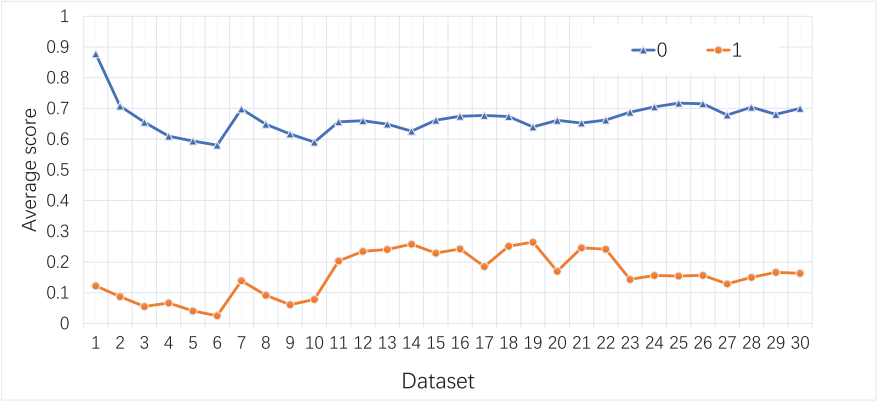

Fig. 8. The average score of $R1(s)$ for different datasets.

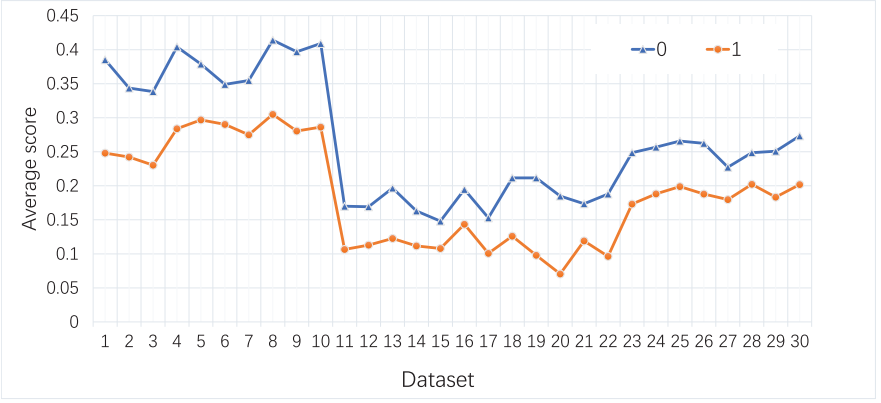

Fig. 9. The average score of $R2(s)$ for different datasets.

$R1(s)$ within (0.5,1] and $R2(s)$ within (1/3,1]. For the example as shown in Fig. 7, its ratio values are $R1(2) = 2/3$, $R1(4) = 1/3$, $R1(5) = 0/3 = 0$ and $R1(6) = 3/3 = 1$ for the first indicator, and $R2(2) = 1/2$, $R2(4) = 0$, $R2(5) = 1/2$ and $R2(6) = 0$ for the second indicator. Therefore, TEMP=[1,1,1,1] is obtained for the particular case as shown in Fig. 11. Assume stack 5 is chosen in the exact solution, the corresponding outcome will be labeled 2, as stack 5 falls into the second element in array TEMP. In order to eliminate the impact of scalability, we will normalize the data again to adopt TEMP = [0.25, 0.25, 0.25, 0.25] rather than the original form TEMP = [1,1,1,1]. That is to say, the scalability issue is handled through the normalized TEMP.

For each relocation, we will have a normalized TEMP and an element index corresponding to the optimal solution. We record all these data from the running on the 30 datasets and generate 6693 records. In order to determine the proper number of decision trees in each random forest, we varied the number of decision trees in the range {1,...,1000} with incremental interval set to 25. We observed that the estimation errors keep the same when the number of decision trees is larger than 50, and therefore, we set this number to 100 in the subsequent experiments. By randomly dividing the data into two sets according to the ratio of 7 to 3, we form the training set and testing set, respectively. The classifier is constructed by Random Forest in RStudio software, and, as a consequence, the prediction accuracy is 80.5% for the testing set. The runtime for generating the classifier by invoking the RStudio software is 6.5 min, which can be ignored when compared to the total runtime of the test dataset. After the random forest based branch pruner is constructed, by injecting the TEMP of the current

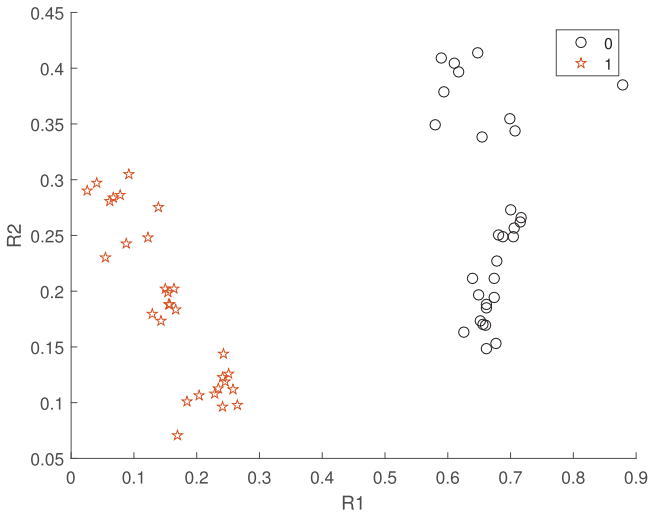

Fig. 10. The average values of two ratios for the 30 datasets in terms of different stack types.

R1 \ R2	[0, 1/3]	(1/3, 1]
[0,0.5]	1st element stack 4 R1(4) = 1/3 R2(4) = 0	2nd element stack 5 R1(5) = 0 R2(5) = 1/2
(0.5,1]	3rd element stack 6 R1(6) = 1 R2(6) = 0	4th element stack 2 R1(2) = 2/3 R2(2) = 1/2

Fig. 11. The example of TEMP for the case as shown in Fig. 7.

bay, for example, TEMP = [0.25, 0.25, 0.25, 0.25] for the case as shown in Fig. 7, into the random forest, it will identify the element number of the array to indicate which category of stacks will proceed to the branching process and which will be pruned at this stage. For example, if the returned value is 3, then it means that only the third category of branches (in this case, corresponding to stack 6) will be retained and all other categories of branches (in this case, corresponding to stacks 2, 4 and 5) will be discarded. Through this process, the random forest based branch pruner fulfills the pruning function.

6.3. Association rules based branch pruner

The branch pruner based on random forest has primarily encountered the following two shortcomings which will be further illustrated in the numerical experiments: (i) the time for recording all the required data and the time to generate the classifying results from the RStudio file is relatively long, which, to a certain extent, increases the running time of the algorithm; (ii) the numerical experiments also show that the classification accuracy is not high when it applies to large-scale problems. The possible reason is that the statistical indicators, i.e. $R1(s)$ and $R2(s)$, may not be representative enough, and the normalization operation with the aim to eliminate the impact of scalability may remove some important features. Due to these considerations, we attempt to find some more intuitive and more efficient rules to guide the pruning, that is, the association rules based pruner in this subsection. The basic idea is that we divide the stacks into two categories as we did in the Min-Max method presented in Section 4.3, then we retain partial stacks to represent each type with the aim to reduce the search space, and finally the selection recommendations will be generated by the association rules based on the remaining data.

We count, separately, the number of choosing Type-1 stacks (not necessarily with the highest stack priority) and choosing Type-1 stacks with the highest stacks priority for relocation in the exact solution for each instance covered in the 30 datasets. The result is shown in Fig. 12, in which the right-side vertical coordinate “Ratio” refers to the ratio of the number

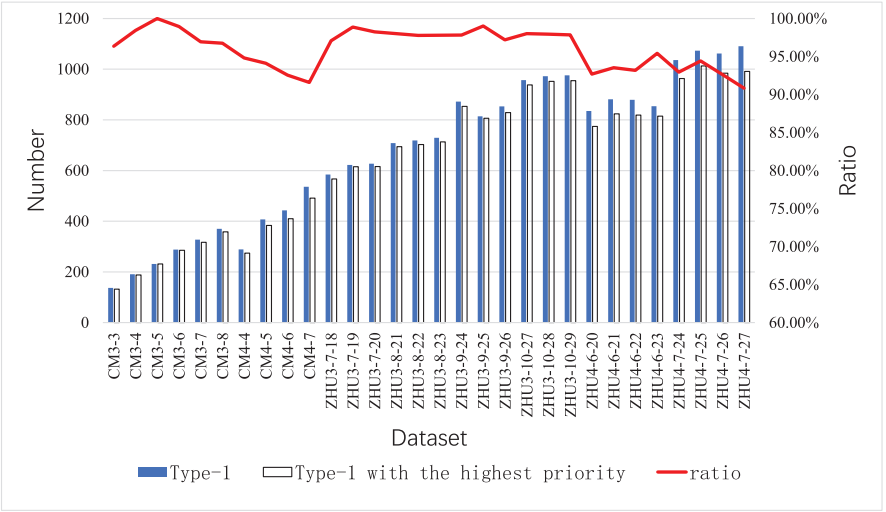

Fig. 12. Type-1 stack selection statistics.

Table 1
The selected features.

Id	Definition
N1	a binary variable, 1, if there are Type-1 stacks; 0, otherwise
N2	the priority-based ranking number of the stack chosen in the exact solution among all the Type-2 stacks. For example, in Fig. 7, the priorities of Type-2 stacks 2, 4 and 6 are 7, 8 and 6, respectively, and, therefore, their priority-based ranking numbers are 2, 3 and 1, respectively. If stack 2 is chosen in the exact solution, then N2=2.
N3	the number of Type-2 stacks
N4	the difference between the number of Type-2 stacks and the priority ranking number of the stack chosen in the exact solution, i.e. N3-N2
N5	the priority ranking number of the container to be relocated among all the containers above the target one. For example, in Fig. 7, the two blocking containers 9 and 2 have ranking number 2 and 1, respectively.
N6	a binary variable, 1, if the container to be relocated has the largest priority number among all the containers above the target one; 0, otherwise
N7	the number of containers above the target one
N8	the difference between the stack height limit and the height of the stack recommended by the Min-Max rules

of Type-1 stacks with highest priority selected in the exact solution to the number of Type-1 stacks selected in the corresponding exact solution for each dataset. From the result, we find that the ratio of each dataset is more than 90%, with the average equal to 95.73%. Based on this observation, we, to a large extent, only need to keep the Type-1 stack with the highest stack priority for further consideration and discard the rest to reduce the search space.

As for the selection of Type-2 stacks for further considerations, because statistical results are not that intuitive as in the selection of Type-1 stacks, we will resort to the association learning method to generate the recommendation. The features used in the learning method are shown in Table 1, most of which are extracted directly from those widely used in the upper bound calculation methods reported in the published literature, including the stack priority ranking, stack height, container ranking and other aspects. Roughly speaking, the 8 features can be categorized into 4 types, with the first type (including N1) related to Type-1 stacks, the second type (including N2 to N4) associated with Type-2 stacks, the third type (including N5-N7) pertaining to the stack where the target container is located, and the fourth type (including N8) suggesting a preferred stack. Therefore, the 8 features are both representative and diversified, providing a relatively complete representation for the bay.

We collect the data from all the operations related to the Type-2 stacks before the number of blocking containers equals 0 for the 30 datasets introduced in Section 6.1 as after the number equals 0 there will be no more relocation operations. For each operation in the exact solution, we recorded all the values of the features as shown in Table 1 and accumulated 3498 records in total. In order to make the data records processable for association rules, we define multiple 0-1 variables for possible values of each feature, and the number of variables of each feature depends on the chosen data. We need to conduct statistical analysis for the chosen data first before determining the number of variables of each feature. For the 3498 data records used in this paper, the possible values and the number of variables of each feature obtained through

Table 2
The variables of each feature for the accumulated 3498 records.

Id	Possible values	Variables	Number of variables (N)
N1	{0,1}	hastypeone/notypeone	2
N2	{1,2,...,8}	typetworankN	8
N3	{1,2,...,8}	typetwonumN	8
N4	{0,1,...,5}	rankdiffN	6
N5	{1,2,3}	blockconrankN	3
N6	{0,1}	ismaxupid/notmaxupid	2
N7	{1,2,3,4}	blockconnumN	4
N8	{1,2,...,6}	heightdiffN	6

Table 3
Rules list in the first execution of the Apriori algorithm.

LHS	RHS	support	confidence
{}	{blockconnum1}	0.6134934	0.6134934
{}	{rankdiff0}	0.6409377	0.6409377
{}	{blockconrank1}	0.7238422	0.7238422
{}	{ismaxupid}	0.8624929	0.8624929
{blockconnum1}	{blockconrank1}	0.6134934	1.0000000
{blockconrank1}	{blockconnum1}	0.6134934	0.8475513
{blockconnum1}	{ismaxupid}	0.6134934	1.0000000
{ismaxupid}	{blockconnum1}	0.6134934	0.7113026
{rankdiff0}	{ismaxupid}	0.5457404	0.8514719
{ismaxupid}	{rankdiff0}	0.5457404	0.6327478
{blockconrank1}	{ismaxupid}	0.6134934	0.8475513
{ismaxupid}	{blockconrank1}	0.6134934	0.7113026
{blockconnum1, blockconrank1}	{ismaxupid}	0.6134934	1.0000000
{blockconnum1, ismaxupid}	{blockconrank1}	0.6134934	1.0000000
{blockconrank1, ismaxupid}	{blockconnum1}	0.6134934	1.0000000

blockconnum corresponding to N7 in Table 1; blockconrank to N5; ismaxupid to N6; rankdiff to N4.

statistics are listed in Table 2. Using RStudio software and applying Apriori association learning algorithm to the generated data, we first extract the rules as shown in Table 3 with both support and confidence greater than 0.5.

As we want to use rules to guide branch pruning, what we are concerned about is which features can deduce the "rankdiff" feature, which means the difference between the number of Type-2 stacks and the ranking sequence of the selected stack in the exact solution. The "rankdiff" features serve as the indicator for the selected stack, which, to a certain extent, can eliminate the influence of the stack size. For example, "rankdiff0" means the difference is 0, meaning the selected stack is the one with the lowest stack priority in Type-2 stacks. Each row in Table 3 represents a rule, in which the left-hand side (LHS) and the right-hand side (RHS) both indicate the set of features. For example, the fifth row represents the rule that the feature "blockconnum1" leads to the feature "blockconrank1", namely, the chosen stack with the feature "blockconnum1" has the feature "blockconrank1" at the same time, for which the support is 0.6134934 and the confidence is 1.0000000. From the results, we observe two rules leading to the "rankdiff" feature. As the first one dominates the second one, we can only consider the first one which means that the stack with the lowest stack priority needs to be kept.

Then we delete the "rankdiff0" data and run a further analysis on the remaining 1256 records. We get 80 rules by Apriori association learning algorithm, whose support and confidence are both greater than or equal to 0.5. After checking the rules leading to "rankdiff", we obtain the results as shown in Table 4. As other rules as shown in Table 4 can be dominated by the first rule, it convinces us to keep the stack with the second lowest priority and discard the rest. Considering that only 64 records are left after deleting "rankdiff1" data, we will stop there and select the two stacks with first two lowest stack priorities to represent the Type-2 stacks. The total runtime for applying the Apriori association learning algorithm in the RStudio software is less than 1 min. In conclusion, at most three stacks are reserved each time, which include the Type-1 stack with the highest stack priority and the Type-2 with first two lowest stack priority.

7. Numerical experiments

All experiments are coded in Java, and run on a personal computer with Intel(R) Core i7-7500U CPU @2.70GHz and RAM-8GB.

7.1. Test on the calculation methods for the lower bounds

In this subsection, we compare the performance of the LB4 and the newly proposed ELB4. In order to compare the performance of different calculation methods for the LB on large-scale problems, we choose, from Zhu et al. (2012), the

Table 4
The association rules leading to rankdiff1.

LHS	RHS	support	confidence
{}	{rankdiff1}	0.949045	0.949045
{blockconnum1}	{rankdiff1}	0.589968	0.969895
{blockconrank1}	{rankdiff1}	0.675159	0.966933
{ismaxupid}	{rankdiff1}	0.838376	0.950361
{heightdiff1}	{rankdiff1}	0.880573	0.975309
{blockconnum1, blockconrank1}	{rankdiff1}	0.589968	0.969895
{blockconnum1, ismaxupid}	{rankdiff1}	0.589968	0.969895
{blockconnum1, heightdiff1}	{rankdiff1}	0.566083	0.982044
{blockconrank1, ismaxupid}	{rankdiff1}	0.589968	0.969895
{blockconrank1, heightdiff1}	{rankdiff1}	0.643312	0.984166
{heightdiff1, ismaxupid}	{rankdiff1}	0.784236	0.973320
{blockconnum1, blockconrank1, ismaxupid}	{rankdiff1}	0.589968	0.969895
{blockconnum1, blockconrank1, heightdiff1}	{rankdiff1}	0.566083	0.982044
{blockconnum1, heightdiff1, ismaxupid}	{rankdiff1}	0.566083	0.982044
{blockconrank1, heightdiff1, ismaxupid}	{rankdiff1}	0.566083	0.982044
{blockconnum1, blockconrank1, heightdiff1, ismaxupid}	{rankdiff1}	0.566083	0.982044

blockconnum corresponding to N7 in Table 1; blockconrank to N5; ismaxupid to N6; heightdiff to N8; rankdiff to N4.

Table 5
The comparison of the calculation methods for LBs.

Dataset	LB4		ELB4		GAP_R	GAP_T
	Relocation1	Time1 (s)	Relocation2	Time2 (s)		
7-10-63	4605	1.111	4796	1.833	4.15%	64.99%
7-10-64	4706	0.766	4920	1.329	4.55%	73.50%
7-10-65	4892	1.128	5070	1.451	3.64%	28.63%
7-10-66	4914	0.891	5074	1.164	3.26%	30.64%
7-10-67	5086	0.984	5277	1.354	3.76%	37.60%
7-10-68	5073	0.797	5237	1.191	3.23%	49.44%
7-10-69	5186	0.906	5363	1.349	3.41%	48.90%
Average	4923	0.940	5105	1.382	3.71%	47.67%

GAP_R = (Relocation2-Relocation1)/Relocation1 * 100 GAP_T = (Time2-Time1)/Time1 * 100

datasets of the form $H-S-N$ (whose specific meaning is illustrated in Section 6.1), including 7-10-63, 7-10-64, 7-10-65, 7-10-66, 7-10-67, 7-10-68 and 7-10-69, and there are 100 instances for each dataset. As shown in Table 5, the quality of LBs calculated by ELB4 has improved by 3.71% when compared to LB4, but the time has increased by 47.67%. It is observed that the improvement is costly in terms of the increase in runtime. Considering that the LB needs to be calculated at each node in the branch-and-bound tree, we decide to use LB4 instead for the subsequent experiments to save the total runtime.

7.2. Test on the calculation methods for the upper bounds

In this section, we will test the calculation methods for the upper bounds, that is, MLUB. As several variants of MLUB are involved, to abate the confusion, we make the following definitions:

- Complete-VRH: the MLUB without adopting pruners in its look-ahead process, that is, the look-ahead method in conjunction of the heuristic method proposed in Ting and Wu (2017);
- RF-VRH: the MLUB adopting the random forest pruner in its look-ahead process;
- AR-VRH: the MLUB adopting the association rules pruner in its look-ahead process.

7.2.1. The look-ahead method with and without the branch pruner

In order to compare the performance of different calculation methods for the LB on large-scale problems, we still resort to dataset 7-10-63, 7-10-64, 7-10-65, 7-10-66, 7-10-67, 7-10-68 and 7-10-69 from Zhu et al. (2012). For the convenience of comparison, we take a small look-ahead depth for each of the three methods, i.e., $D = 1$, and the results are shown in Table 6.

For comparison, a calculation method based on the PR4 heuristic proposed in Zhu et al. (2012) (denoted by Zhu) and the VRH proposed in Ting and Wu (2017) (denoted by VRH), are also included in this experiment. In Table 6, "Relocation" represents the sum of the UB for the 100 instances obtained by the corresponding method for the considered dataset, and "Time" represents the total running time.

From the perspective of the quality of the UB (evaluated by the value of "Relocation"), the VRH gets better UBs than the method proposed in Zhu et al. (2012), and most of the UBs obtained by Complete-VRH, RF-VRH and AR-VRH are better than

Table 6
The comparison of the calculation methods for UBs ($D = 1$).

Dataset	Zhu (Zhu et al., 2012)		VRH (Ting and Wu, 2017)		Complete-VRH		RF-VRH		AR-VRH	
	Relocation	Time(s)	Relocation	Time(s)	Relocation	Time(s)	Relocation	Time(s)	Relocation	Time(s)
7-10-63	5912	0.016	5748	0.031	5684	0.062	5739	0.546	5712	0.048
7-10-64	6111	0.016	5963	0.036	5910	0.093	5958	0.484	5933	0.094
7-10-65	6386	0.016	6277	0.032	6216	0.062	6259	0.500	6235	0.116
7-10-66	6307	0.016	6206	0.063	6161	0.046	6200	0.446	6171	0.080
7-10-67	6668	0.015	6516	0.031	6488	0.078	6517	0.406	6496	0.070
7-10-68	6706	0.031	6589	0.047	6576	0.047	6595	0.282	6579	0.046
7-10-69	6964	0.020	6811	0.030	6811	0.051	6811	0.071	6811	0.040

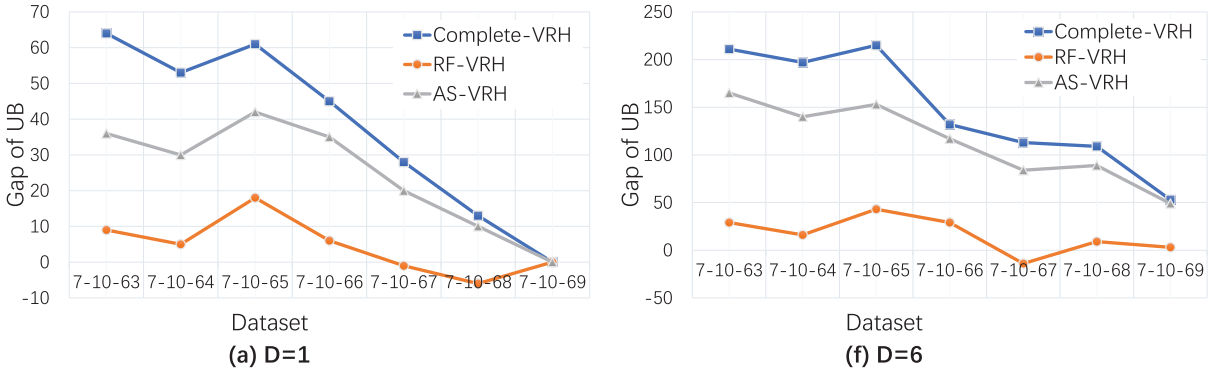

Fig. 13. The gap of UB under different datasets.

that of the VRH, which implies that the application of the look-ahead method can significantly improve the quality of the UB. Among the methods, the UBs of the Complete-VRH are the tightest, which should be taken for granted, as it explores all the possibilities of relocation. With pruning strategies adopted to improve the solution efficiency, the UB quality of RF-VRH and AR-VRH is slightly worse than that of Complete-VRH. Paradoxically, RF-VRH has slightly worse results than VRH for the datasets 7-10-67 and 7-10-68, and it is probably because the initial state of bay is relatively full and the look-ahead method is conducted for only 1 layer.

In terms of the running time, the method in Zhu et al. (2012), which produces the worst UB, has the shortest running time. Among the other four methods, using VRH alone has the shortest running time, followed by AR-VRH. The running time of RF-VRH is the longest, since there are few available relocation stacks and branches under these datasets, and the random forest based filter needs some operating time to solve and output prediction results, restricting the advantage of pruning. As the number of alternative stacks increases and the look-ahead depth increases, the advantage of pruning gradually emerges, which is further verified in Section 7.2.2.

7.2.2. Sensitivity analysis

As can be seen from the above analysis, how full the bay is (that is, the load intensity) and how deep the look-ahead depth is have certain influences on the performance of the look-ahead method and different pruning mechanisms. In this subsection, sensitivity analysis is further carried out for both parameters.

As the situations with $D = 0$ (i.e. using VRH without look-ahead) and $D = 1$ are considered in 7.2.1, $D = 2, 3, 4, 5, 6, 7, 8, 9$ are selected for numerical experiments in this subsection. Partial results for $D = 2, 4, 6, 8$ are shown in Table 7 as other results display the same trend.

Under different load intensities, the quality of the UB obtained by the three look-ahead methods is firstly analysed. As shown in Fig. 13 (where only D=1 and 6 are shown as other depths reveal the same trend), the y-coordinate represents the UB obtained by VRH minus the UB obtained by the corresponding method, e.g., the y-coordinates of the points on line "Complete-VRH" represent the UB obtained by VRH minus the UB obtained by Complete-VRH. It can be found that, with the increase of the container number (i.e. the decrease of available stacks for each relocation), the gap between the UB obtained by the three look-ahead methods and that by VRH decreases, which implies that, if there are more available stacks for relocation (i.e., low load intensity of the bay), it is more likely to get tighter UBs by the look-ahead methods.

Under different look-ahead depths, the UBs obtained by Complete-VRH and AR-VRH are all tighter than that obtained by VRH, and the quality of the UBs increases with the look-ahead depth. The RF-VRH also displays the same trend. However, when the number of containers is large (i.e., the bay is relatively full), the UB quality of RF-VRH will be partially worse than that of VRH and the solution performance will be unreliable. In terms of the running time, the efficiency of the three look-ahead methods, i.e. Complete-VRH, RF-VRH and AR-VRH, decreases with the look-ahead depth, and the decreasing rate of the Complete-VRH is obviously faster than the other two methods with pruning strategies.

Table 7
The comparison of the calculation methods for UBs ($D = 2, 4, 6, 8$).

Dataset	$D = 2$						$D = 4$					
	Complete-VRH		RF-VRH		AR-VRH		Complete-VRH		RF-VRH		AR-VRH	
	Relocation	Time(s)	Relocation	Time(s)	Relocation	Time(s)	Relocation	Time(s)	Relocation	Time(s)	Relocation	Time(s)
7-10-63	5647	0.210	5739	1.068	5690	0.080	5590	0.797	5737	1.499	5642	0.225
7-10-64	5854	0.187	5959	0.710	5901	0.640	5807	0.468	5951	1.234	5854	0.162
7-10-65	6165	0.110	6241	0.926	6187	0.107	6110	0.313	6229	0.985	6154	0.181
7-10-66	6130	0.078	6192	0.734	6140	0.111	6098	0.242	6182	0.989	6113	0.202
7-10-67	6461	0.078	6511	0.719	6471	0.106	6437	0.141	6524	0.734	6456	0.084
7-10-68	6565	0.083	6595	0.394	6572	0.060	6529	0.078	6583	0.594	6539	0.096
7-10-69	6797	0.047	6810	0.313	6797	0.047	6781	0.047	6806	0.375	6781	0.062
Dataset	$D = 6$						$D = 8$					
	Complete-VRH		RF-VRH		AR-VRH		Complete-VRH		RF-VRH		AR-VRH	
	Relocation	Time(s)	Relocation	Time(s)	Relocation	Time(s)	Relocation	Time(s)	Relocation	Time(s)	Relocation	Time(s)
7-10-63	5537	2.109	5719	1.828	5583	0.416	5493	10.357	5692	2.985	5546	1.090
7-10-64	5766	1.203	5947	1.484	5823	0.323	5736	4.628	5955	2.545	5799	0.682
7-10-65	6062	0.875	6234	1.561	6124	0.314	6021	3.346	6215	2.504	6081	0.624
7-10-66	6074	0.406	6177	1.250	6089	0.178	6035	1.421	6182	1.874	6062	0.414
7-10-67	6403	0.234	6530	0.812	6432	0.188	6374	0.530	6544	1.263	6411	0.302
7-10-68	6480	0.187	6580	0.750	6500	0.135	6447	0.360	6574	1.187	6469	0.266
7-10-69	6758	0.093	6808	0.578	6762	0.084	6741	0.156	6814	0.958	6745	0.127

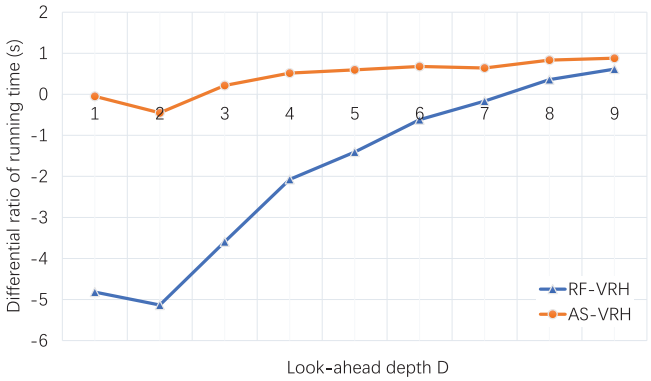

Fig. 14. The comparison of running time under different depths.

As shown in Fig. 14, the y-coordinate represents the ratio of the difference of total solution time between RF-VRH/AR-VRH and Complete-VRH under different depths to the solution time of Complete-VRH. More specifically, when the running time of RF-VRH/AR-VRH is less than Complete-VRH, the y-coordinate is positive, otherwise it is negative, which means, when this value is positive, the branch pruner of the corresponding method has an improved effect on the solution efficiency, and the larger the value, the more significant the improvement. It shows that the improvements of solution efficiency of random forest based and association rules based filters for the look-ahead VRH both increase with D. When D reaches a certain value, the y-coordinate reaches positive values, and pruning through the two filters can improve the solution efficiency, reflective of the advantage of pruning.

In summary of this section, using the look-ahead method to improve VRH can effectively enhance the quality of the UB, and further pruning by random forest and association rules would slightly decrease the quality of the UB compared with the complete look-ahead UB, but still ensure that the UB quality is higher than that of VRH for most cases. Considering the time consumed by the construction of pruning filters, the solution efficiency is lower than Complete-VRH when the look-ahead depth is small or there are only a few available stacks for relocation. However, with the increase of D, and the decrease of the load intensity of the bay, the improvement of pruning filters on the solution efficiency emerges. Among the two filters, the experimental results show that the association rules based filter is more reliable than the random forest based filter in both UB quality and solution efficiency. In addition, it should be noted that, though the quality of UB improves with the increase of D, running time also increases, and the increase of running time will be further amplified when the obtained UB is applied to the calculation of each node by the IB&B. Therefore, there is a trade-off between the UB quality and solution efficiency in the searching framework of the IB&B, which will be further verified in Sections 7.3 and 7.4.

7.3. Test on the exact method IB&B

7.3.1. Test on performance

We compared the IB&B with the CB&B based on the datasets adopted in Zhu et al. (2012) in terms of the number of nodes explored to get optimal solutions as well as the running time, and, to avoid the impact of the quality of UB, the method based on the PR4 heuristic (without look-ahead method and branch pruner) in Zhu et al. (2012) is applied to the calculation of upper bounds. As each dataset contains 100 instances, we report the average results in Table 8, from which we see that the IB&B performs better than the CB&B in terms of both performance indicators.

7.3.2. Sensitivity analysis

As the look-ahead depth can affect the number of nodes explored as well as the search time, we conduct the sensitivity analysis to observe the impact of the look-ahead depth D on the two performance indicators under the UB-priority branching strategy. The experimental results are shown in Table 9, from which we see that the running time increases when the look-ahead depth D increases, and the number of nodes explored has the trend to decrease when D increases, but the reduction is not evident when compared with the total number. We, therefore, choose $D = 1$ in the subsequent experiment regarding the comparison between the UB-priority and LB-priority branching strategies.

7.3.3. Branching strategies

We compare the UB-priority with the LB-priority (using LB4) branching strategies based on the datasets adopted in Zhu et al. (2012) in terms of the number of nodes explored as well as the running time for generating the optimal solution. The results are shown in Table 10, from which we see that the UB-priority branching strategy can effectively reduce the total number of nodes explored, and the runtime of the UB-priority branching strategy is relatively shorter when compared with the LB-priority branching strategy, except for RF-VRH, which is because of the long time consumed to solve and output prediction results by the random forest based filter. It is noted that although the UB-priority branching strategy outperforms

Table 8
Test on the IB&B (adopting PR4 heuristic (Zhu et al., 2012) to calculate UBs).

Dataset	Optimal solution	CB&B		IB&B		GAP_N	GAP_T
		Node number1	Time1(s)	Node number2	Time2(s)		
5-6-25	1674	108042	25.935	90648	10.814	−16.10%	−58.30%
5-6-26	1859	219609	18.481	187820	15.544	−14.48%	−15.89%
5-6-27	1888	108099	11.124	83777	7.052	−22.50%	−36.61%
5-6-28	1969	120795	12.806	56709	5.823	−53.05%	−54.53%
5-6-29	2113	498046	48.592	249971	22.238	−49.81%	−54.24%
5-7-30	2011	756815	74.103	618241	60.160	−18.31%	−18.82%
5-7-31	2186	1908334	185.981	1016409	111.438	−46.74%	−40.08%
5-7-32	2268	1067219	108.531	640883	62.371	−39.95%	−42.53%
5-7-33	2340	1557448	122.869	1021655	83.952	−34.40%	−31.67%
5-7-34	2454	768616	83.128	656812	72.836	−14.55%	−12.38%
6-6-30	2295	5295414	617.146	5241436	601.926	−1.02%	−2.47%
6-6-31	2476	7505345	982.455	6251112	919.500	−16.71%	−6.41%
6-6-32	2682	20614627	2924.011	12945238	1821.959	−37.20%	−37.69%
6-6-33	2873	39792068	5909.837	12472842	1799.120	−68.65%	−69.56%
6-6-34	2920	33474780	3495.543	23438601	2422.007	−29.98%	−30.71%

GAP_N = (Node number2-Node number1)/Node number1 * 100 GAP_T = (Time2-Time1)/Time1 * 100

Table 9
Sensitivity analysis of the look-ahead depth.

Dataset	Optimal solution	Complete-VRH					
		$D = 1$		$D = 2$		$D = 3$	
		Node number	Time(s)	Node number	Time(s)	Node number	Times(s)
5-6-25	1674	79381	10.427	80666	18.053	80471	35.486
5-6-26	1859	188418	20.686	192124	38.321	187551	78.233
5-6-27	1888	82674	3.781	81488	17.092	81489	33.772
5-6-28	1969	42290	5.373	42271	7.839	42170	13.701
5-6-29	2113	249221	28.771	249021	57.281	249227	126.971
Dataset	Optimal solution	RF-VRH					
		$D = 1$		$D = 2$		$D = 3$	
		Node number	Time(s)	Node number	Time(s)	Node number	Times(s)
5-6-25	1674	93602	98.444	95672	162.354	101240	273.520
5-6-26	1859	189354	198.148	190087	290.504	187843	450.690
5-6-27	1888	82791	81.043	82759	201.705	83269	345.579
5-6-28	1969	46889	41.883	46929	65.770	47025	101.363
5-6-29	2113	249405	222.192	249999	562.925	250036	711.764
Dataset	Optimal solution	AR-VRH					
		$D = 1$		$D = 2$		$D = 3$	
		Node number	Time(s)	Node number	Time(s)	Node number	Times(s)
5-6-25	1674	78469	9.725	79754	12.266	79784	18.034
5-6-26	1859	188864	15.741	192595	24.025	188022	36.596
5-6-27	1888	82498	8.328	81486	10.269	81486	16.410
5-6-28	1969	42267	5.191	42266	6.834	42276	7.648
5-6-29	2113	249222	21.947	248928	30.760	249134	50.099

the LB-priority branching strategy and fits our test data very well, there is no guarantee that the former strategy outperforms the latter one for general cases. We believe that the relationship between the two strategies deserves deep investigation.

7.4. Test on the heuristic method MLBS

7.4.1. Solution quality of the heuristic method

We test our method on the datasets adopted in Caserta et al. (2011) with the look-ahead depth $D = 3$ and the beam width $W = 5$. The results are shown in Table 11 where the CM represents the method proposed by Caserta et al. (2011). It can be seen from the table that Complete-VRH can generate better results than VRH, and using the branch pruners can improve the efficiency of the algorithm but at the slight cost of solution quality compared with Complete-VRH. In fact, when the look-ahead depth is increased to $D = 4$, all the results generated by AR-VRH become better than those by VRH as shown in Table 11.

The runtime of the MLBS using the three aforementioned methods for calculating UBs is shown in Table 12, from which we see that, with the increase of problem sizes, the computing time of the method using Complete-VRH increases rapidly, and the advantage of using the branch pruner becomes more evident. From Tables 11 and 12, we can see that using Complete-VRH in the MLBS can generate better results than using VRH at the cost of runtime, and incorporating branch

Table 10
Comparsion between LB- and UB-priority branching strategies.

Dataset	Optimal solution	LB-priority		UB-priority							
				VRH (Ting and Wu, 2017)		Complete-VRH		RF-VRH		AR-VRH	
		Node number	Time(s)	Node number	Time(s)	Node number	Time(s)	Node number	Time(s)	Node number	Time(s)
5-6-25	1674	90648	23.623	81501	4.559	79381	10.427	93602	98.444	78469	9.725
5-6-26	1859	187820	44.158	189863	7.535	188418	20.686	189354	198.148	188864	15.741
5-6-27	1888	83777	22.591	82583	9.506	82674	3.781	82791	81.043	82498	8.328
5-6-28	1969	56709	16.461	46532	3.399	42290	5.373	46889	41.883	42267	5.191
5-6-29	2113	249971	62.333	249297	10.003	249221	28.771	249405	222.192	249222	21.947
5-7-30	2011	618241	169.187	605804	31.269	599098	92.116	608706	543.962	599506	62.769
5-7-31	2186	1016409	270.020	967282	47.085	966409	222.438	988183	1115.593	966331	123.099
5-7-32	2268	640883	154.024	629694	25.843	635582	182.848	628137	869.474	632391	104.292
5-7-33	2340	1021655	239.517	1014436	35.957	1014732	168.512	1014860	1593.988	1014738	80.521
5-7-34	2454	656812	172.892	612972	29.474	619442	102.265	615714	539.604	620843	93.604

Table 11
Average relocation number of different UBs on CM dataset.

Bay info			Average relocation number							
Stack height	Stack number	Container number	CM	VRH	$D = 3$			$D = 4$		
			(Caserta et al., 2011)	(Ting and Wu, 2017)	Complete-VRH	RF-VRH	AR-VRH	Complete-VRH	RF-VRH	AR-VRH
5	4	20	16.6	15.45	15.425	15.425	15.425	15.4	15.425	15.425
5	5	25	18.8	19	18.95	18.925	18.95	18.725	18.925	18.9
5	6	30	22.1	22.22	22.25	22.275	22.3	22.2	22.25	22.2
5	7	35	25.8	24.37	24.35	24.425	24.37	24.35	24.425	24.37
5	8	40	30.1	27.8	27.75	27.775	27.775	27.725	27.775	27.775
5	9	45	33.1	30.52	30.52	30.55	30.55	30.52	30.55	30.52
5	10	50	36.4	33.4	33.375	33.45	33.375	33.25	33.4	33.325
6	6	36	32.4	31.2	31.2	31.375	31.25	31.15	31.375	31.175
6	10	60	49.5	46.15	46.025	46	46.075	45.925	46	45.975
10	6	60	102	79.97	78.5	81.475	78.35	78.375	81.475	78.65
10	10	100	128.3	113.25	112.325	115.575	113.175	111.9	114.85	112.425

The results in column “CM (Caserta et al., 2011)” are copied directly from the referred paper.

Table 12
Average running time of different UBs when $D = 3$ on CM dataset.

Bay info			Average runing time(s)				
Stack height	Stack number	Container number	CM (Caserta et al., 2011)	VRH (Ting and Wu, 2017)	Complete-VRH	RF-VRH	AR-VRH
5	4	20	0.5	0.011	0.016	0.084	0.012
5	5	25	0.8	0.018	0.049	0.215	0.027
5	6	30	0.8	0.031	0.137	0.305	0.056
5	7	35	1.43	0.041	0.414	0.431	0.077
5	8	40	1.46	0.045	0.793	0.631	0.076
5	9	45	1.41	0.056	1.947	0.839	0.101
5	10	50	1.87	0.098	3.806	1.272	0.156
6	6	36	1.74	0.081	0.469	0.702	0.130
6	10	60	1.95	0.230	16.047	5.552	0.572
10	6	60	4.73	0.914	4.838	4.773	1.597
10	10	100	6.34	4.946	290.335	66.570	9.920

The results in column "CM (Caserta et al., 2011)" are copied directly from the referred paper.

Table 13
Test results of different UBs when $D = 3$ on the dataset in Lee and Lee (2010).

Case	LL (Lee and Lee, 2010)		VRH (Ting and Wu, 2017)		Complete-VRH		RF-VRH		AR-VRH	
	Number	Time(s)	Number	Time(s)	Number	Time(s)	Number	Time(s)	Number	Time(s)
R011606_0070_001	48	6304.280	37	0.012	37	0.108	37	0.347	37	0.010
R011606_0070_002	47	11081.030	38	0.405	38	5.750	38	2.395	38	0.523
R011606_0070_003	40	5501.920	38	0.067	38	7.552	38	2.395	38	0.226
R011606_0070_004	88	9026.420	45	0.515	45	21.741	45	4.153	45	1.116
R011606_0070_005	54	9107.970	40	0.219	40	1.428	40	1.050	40	0.303
R011608_0090_001	100	13268.670	61	0.895	61	357.731	61	19.244	61	1.985
R011608_0090_002	101	11134.630	61	0.088	61	0.569	61	0.508	61	0.168
R011608_0090_003	126	21583.130	65	2.316	65	323.351	65	10.451	65	1.836
R011608_0090_004	88	7042.380	61	0.416	61	5.945	61	2.508	61	0.478
R011608_0090_005	92	13738.000	60	1.160	59	69.677	59	4.370	59	2.298
U011606_0070_001	55	17326.310	55	0.015	55	1.601	55	1.571	55	0.066
U011606_0070_002	60	11243.400	58	0.012	58	1.962	58	1.015	58	0.068
U011608_0090_001	85	21586.810	76	0.054	76	0.594	76	0.362	76	0.038
U011608_0090_002	90	8021.310	78	0.023	78	3.098	78	1.207	78	0.045

The results in column "LL (Lee and Lee, 2010)" are copied directly from the referred paper.

Table 14
Sensitivity anlysis of the look-ahead depth.

Dataset	Complete-VRH					
	$D = 1$		$D = 2$		$D = 3$	
	Average relocation	Time(s)	Average relocation	Time(s)	Average relocation	Time(s)
5-4	15.475	0.013	15.45	0.022	15.425	0.016
5-5	18.95	0.031	18.925	0.049	18.95	0.049
5-6	22.225	0.071	22.275	0.092	22.25	0.137
5-7	24.425	0.081	24.35	0.173	24.35	0.414
5-8	27.775	0.127	27.75	0.312	27.75	0.793
5-9	30.575	0.176	30.6	0.522	30.525	1.947
5-10	33.425	0.246	33.35	0.878	33.375	3.806
6-6	31.225	0.132	31.075	0.225	31.2	0.469
6-10	46.125	1.092	45.975	4.016	46.025	16.047
10-6	79.15	1.565	78.875	3.128	78.5	4.838
10-10	113.525	16.983	112.575	64.088	112.325	290.335
Dataset			RF-VRH			
	$D = 1$		$D = 2$		$D = 3$	
	Average relocation	Time(s)	Average relocation	Time(s)	Average relocation	Time(s)
5-4	15.475	0.052	15.475	0.065	15.45	0.084
5-5	18.95	0.078	18.975	0.098	19.025	0.215
5-6	22.325	0.114	22.375	0.156	22.325	0.305
5-7	24.5	0.155	24.35	0.199	24.35	0.431
5-8	27.8	0.163	27.775	0.259	27.775	0.631
5-9	30.52	0.199	30.52	0.330	30.55	0.839
5-10	33.425	0.252	33.4	0.438	33.425	1.272
6-6	31.3	0.247	31.25	0.342	31.425	0.702
6-10	46.175	0.944	45.975	1.524	46	5.552
10-6	79.925	1.742	80.1	2.169	79.875	4.773
10-10	113.875	9.164	113.925	18.035	114.125	66.570
Dataset			AR-VRH			
	$D = 1$		$D = 2$		$D = 3$	
	Average relocation	Time(s)	Average relocation	Time(s)	Average relocation	Time(s)
5-4	15.45	0.010	15.425	0.012	15.425	0.012
5-5	18.95	0.022	18.95	0.022	18.9	0.027
5-6	22.275	0.034	22.3	0.040	22.2	0.056
5-7	24.35	0.044	24.37	0.077	24.37	0.077
5-8	27.775	0.053	27.775	0.062	27.775	0.076
5-9	30.575	0.080	30.55	0.085	30.52	0.101
5-10	33.375	0.105	33.375	0.111	33.325	0.156
6-6	31.225	0.093	31.25	0.113	31.175	0.130
6-10	46.1	0.285	46.075	0.418	45.975	0.572
10-6	79.25	1.174	78.35	1.314	78.65	1.597
10-10	113.1	6.350	113.175	7.368	112.425	9.920

Table 15
Sensitivity analysis of beam width when $D = 2$.

Beam width	Complete-VRH		RF-VRH		AR-VRH	
	Average relocation	Time(s)	Average relocation	Time(s)	Average relocation	Time(s)
5	33.35	0.878	33.4	0.438	33.375	0.111
10	33.325	1.300	33.4	0.888	33.35	0.186
20	33.325	2.110	33.35	1.180	33.3	0.397

pruners into the UB calculation methods can save the runtime significantly. To make the conclusion more convincing, we also test the algorithm on the datasets adopted in Lee and Lee (2010) with the look-ahead depth $D = 3$ and the beam width $W = 5$. The results are as shown in Table 13.

7.4.2. Sensitivity analysis

As the look-ahead depth D and beam width W may greatly affect the UB quality as well as running time, we will conduct the sensitivity analysis on the two factors. To analyse the influence of the look-ahead depth, keeping the beam width $W = 5$ unchanged and varying the values of D for Complete-VRH, RF-VRH and AR-VRH, we obtain the result as shown in Table 14. From these sensitivity analyses, we can see that, as the look-ahead depth increases, the solution quality is not notably improved. Therefore, in view of the requirement of solution efficiency, the optimal depth $D = 1$ is adoptable.

As for the influence of beam width, setting the look-ahead depth D to 2 and varying the beam width W for the MLBS, we obtain the result as shown in Table 15 for the dataset 5–10 of Caserta et al. (2011). It can be seen from the result that,

Table 16
Effect of deleting duplicate nodes.

Bay info			Average relocation number	
Stack height	Stack number	Container number	No deletion	Deletion
5	4	20	15.45	15.45
5	5	25	18.925	18.925
5	6	30	22.275	22.275
5	7	35	24.375	24.35
5	8	40	27.75	27.75
5	9	45	30.6	30.6
5	10	50	33.35	33.35
6	6	36	31.075	31.075
6	10	60	46.075	45.975
10	6	60	79	78.875
10	10	100	112.65	112.575

Table 17
Effect of using the Min-Max rules to break ties.

Bay info			Random selection		Min-Max selection
Stack height	Stack number	Container number	Range	Number	Number
5	4	20	2	15.463	15.45
5	5	25	3	18.95	18.925
5	6	30	7	22.25	22.275
5	7	35	6	24.43	24.35
5	8	40	4	27.775	27.75
5	9	45	3	30.598	30.6
5	10	50	6	33.423	33.35
6	6	36	8	31.155	31.075
6	10	60	5	46.033	45.975
10	6	60	18	78.975	78.875
10	10	100	18	112.82	112.575

with the increase of beam width, the quality of the solution is improved, but the improvement is not significant. To save the running time, we will set the beam width to a relatively small number.

7.4.3. Screening techniques analysis

We compare the average relocation number of algorithms with and without deleting duplicate nodes, leading to the result shown in Table 16, from which we can see that solution quality can be improved by deleting duplicate nodes.

In Ting and Wu (2017), when the upper bounds for some nodes are the same, a random selection method is used to break ties. We conduct numerical experiments to invoke the random selection on the CM datasets 10 times to get the average relocation number, as well as the results under the same conditions except using the Min-Max rules to break ties, and the comparison result is shown in Table 17. As can be seen from the table, the average number of relocation using the Min-Max method to break ties is significantly lower than that of using random selection method for most cases. The range in the table represents the difference between the best solution and the worst solution in the 10 random experiments and we find that the performance of the random selection method varies greatly for different problem sizes. With the increase of problem scale, the range increases significantly, implying that the random selection to break tie is not a reliable method.

8. Conclusions

For an important restricted container relocation problem in container terminals, this paper proposes a new calculation method for upper bounds, which incorporates the look-ahead method and machine learning to facilitate the convergence. Furthermore, the machine learning-driven upper bounds are injected into an exact branch-and-bound algorithm and a beam search method to enhance their respective performance. The numerical experiments show that the proposed machine learning-driven UB, the exact algorithm and the heuristic display the better performance than the state-of-the-art methods reported in the literature based on the benchmark test instances.

Further research can be done in the following directions: (i) considering the relocation across bays and combining the operation time of yard cranes; (ii) extending the framework of the algorithms to the unrestricted container relocation problem; and (iii) considering the container relocation problem with duplicate priorities.

Acknowledgments

This work was supported by the National Key R&D Program of China under grant No. 2018AAA0101705, and the National Natural Science Foundation of China under grant No. 71872092.

Appendix A. The virtual relocation heuristic (VRH)

One of the drawback of the Min-Max method is that it only considers one container at a time. To overcome this drawback, Ting and Wu (2017) proposed a virtual relocation heuristic (VRH) to calculate the upper bound, which takes into consideration all blocking containers above the target one simultaneously. Here we will give an example to specifically present the two stages of this algorithm, as shown in Fig. 15.

In the first stage of the VRH, we initially rank the blocking containers above the target container in a descending order by the priority number and obtain the container sequence to be relocated Q = {8, 7, 6, 5}, then determine the well-relocated stacks for the blocking containers sequentially according to the Min-Max condition. As shown in Fig. 15(a), stack 3 satisfies the well-relocated condition for container 8, then the container will be virtually relocated to stack 3. It should be noted that when relocating containers by the Min-Max condition, the first-in-last-out (FILO) principle of containers should be taken into account, i.e., containers originally placed in the lower layer should be above the upper-layer containers after relocation. This can be realized by recording the original minimum layer number of all the accommodated containers in a particular stack s, denoted by $t_{\min}(s)$. As shown in Fig. 15(a), with container 8 relocated to stack 3, the $t_{\min}(s)$ of the stack s is updated from the initial stack height limit 5 to the original layer number of container 8, which is 4. Assume the next container c needs to be relocated to this stack, and then $t(c) < t_{\min}(s)$ must be satisfied, i.e., the original layer number of container c is smaller than the minimum layer number of all the accommodated containers in this stack. As in Fig. 15(b), for the next container 7, as stack 3 satisfies the well-relocated principle, container 7 can be relocated to stack 3, and its corresponding $t_{\min}(s)$ value is updated to 2 accordingly. For the following container 6 and container 5, though stack 3 meets the well-relocated condition, the condition $t(c) < t_{\min}(s)$ is not satisfied, which violates the FILO principle, implying that these two containers will not be processed in this stage. To summarize the first stage of VRH, all blocking containers above the target container is ranked by the priority number in a descending order, and then according to the well-located condition of the Min-Max method and the FILO principle, the well-located stack for each blocking container is searched sequentially. If no stack satisfies the both conditions for one container, this container will be put aside and relocated in the next stage.

In the second stage of the VRH, the remaining containers in the first stage will be ranked according to the priority number in an ascending order and the sequence Q={5, 6} is obtained, which means container 5 should be considered first and container 6 follows. In this case, if container 5 is relocated to a stack that has accommodated other containers before, it is bound to be inserted into the middle of those containers that have been virtually relocated in the first stage. As shown in Fig. 16(a), if container 5 is relocated to stack 3, it needs to be inserted between containers 8 and 9. Here we can determine the target stack based on virtual relocation index (VRI) of the stack s as follows.

$$VRI_s = \begin{cases} \min\{l_{sc} - p(c),\ p(c) - u_{sc}\} & if\ U \le 1 \\ -N + (p(c) - u_{sc}) & otherwise \end{cases} \tag{2}$$

where U denotes the number of blocking containers above the inserted container c. For container 5 in Fig. 16(a), as there are containers 7 and 8 above it, both of which are blocking containers relative to container 5, U=2. N is the initial total number of containers. In terms of stack s, l_{sc} is the highest priority (i.e. the minimum priority number) of all containers below the inserted container c, and u_{sc} is the highest priority (i.e. the minimum priority number) of all containers above the inserted container c. For Fig. 16(a), $l_{35} = \min\{9\} = 9$, and $u_{35} = \min\{7, 8\} = 7$. Therefore, $VRI_3 = -9 + 5 - 7 = -11$ is obtained for stack 3, $VRI_2 = \min\{2 - 55 - 0\} = -3$ is obtained for stack 2, and then the stack with the highest VRI will be selected for

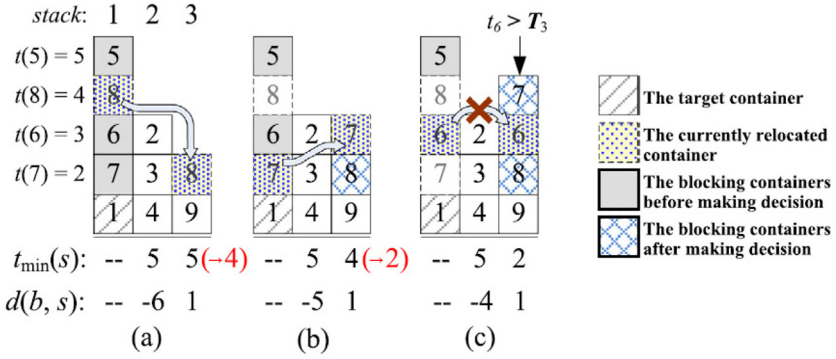

Fig. 15. The first stage of the VRH (Ting and Wu, 2017)

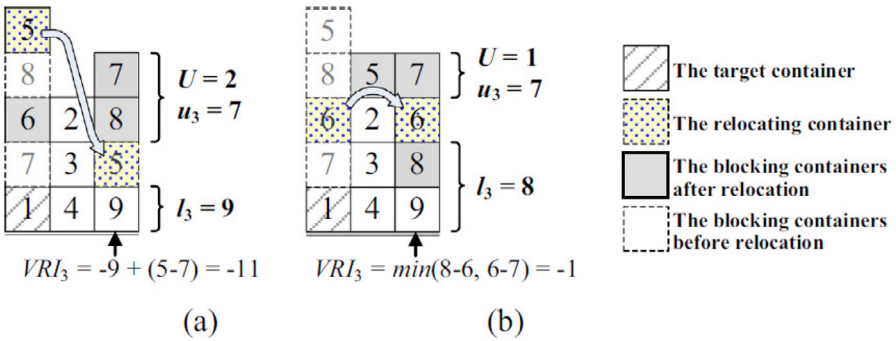

Fig. 16. The second stage of the VRH (Ting and Wu, 2017)

relocation, which means container 5 will be relocated to stack 2. In the same way, we can get the recommended relocation stack for container 6 is stack 3. Repeat the above operation until all containers are extracted, and then a relatively tight UB is obtained.

References

Allahviranloo, M., Recker, W., 2013. Daily activity pattern recognition by using support vector machines with multiple classes. Transp. Res. Part B-Methodol. 58, 16–43.

Andres, M., Nair, R., 2017. A predictive-control framework to address bus bunching. Transp. Res. Part B-Methodol. 104, 123–148.

Arentze, T.A., Timmermans, H.J.P., 2004. A learning-based transportation oriented simulation system. Transp. Res. Part B-Methodol. 38 (7), 613–633.

Bacci, T., Mattia, S., Ventura, P., 2019. The bounded beam search algorithm for the block relocation problem. Comput. Oper. Res. 103, 252–264.

Bell, M.G.H., Liu, X., Angeloudis, P., Fonzone, A., Hosseinloo, S.H., 2011. A frequency-based maritime container assignment model. Transp. Res. Part B-Methodol. 45 (8), 1152–1161.

Bell, M.G.H., Liu, X., Rioult, J., Angeloudis, P., 2013. A cost-based maritime container assignment model. Transp. Res. Part B-Methodol. 58, 58–70.

Bortfeldt, A., Wäscher, G., 2013. Constraints in container loading c a state-of-the-art review. Eur. J. Oper. Res. 229 (1), 1–20.

Carlo, H.J., Vis, I.F.A., Roodbergen, K.J., 2014. Transport operations in container terminals: literature overview, trends, research directions and classification scheme. Eur. J. Oper. Res. 236 (1), 1–13.

Caserta, M., Schwarze, S., Voß, S., 2012. A mathematical formulation and complexity considerations for the blocks relocation problem. Eur. J. Oper. Res. 219 (1), 96–104.

Caserta, M., Voß, S., Sniedovich, M., 2011. Applying the corridor method to a blocks relocation problem. OR Spectrum 33 (4), 915–929.

Dong, C.J., Shao, C.F., Clarke, D.B., Nambisan, S.S., 2018. An innovative approach for traffic crash estimation and prediction on accommodating unobserved heterogeneities. Transp. Res. Part B-Methodol. 118, 407–428.

Forster, F., Bortfeldt, A., 2012. A tree search procedure for the container relocation problem. Comput. Oper. Res. 39 (2), 299–309.

Galle, V., Barnhart, C., Jaillet, P., 2018. A new binary formulation of the restricted container relocation problem based on a binary encoding of configurations. Eur. J. Oper. Res. 267 (2), 467–477.

deGharehgozli, A.H., Laporte, G., Yu, Y., Koster, R., 2014. Scheduling twin yard cranes in a container block. Trans. Sci 49 (3), 686–705.

deGharehgozli, A.H., Yu, Y., Koster, R., Udding, J.T., 2014. An exact method for scheduling a yard crane. Eur. J. Oper. Res 235 (2), 431–447.

Hofleitner, A., Herring, R., Bayen, A., 2012. Arterial travel time forecast with streaming data: a hybrid approach of flow modeling and machine learning. Transp. Res. Part B-Methodol. 46 (9), 1097–1122.

Hottung, A., Tanaka, S., Tierney, K., 2020. Deep learning assisted heuristic tree search for the container pre-marshalling problem. Comput. Oper. Res. 113, 1–13. 104781

Iris, C., Christensen, J., Pacino, D., Ropke, S., 2018. Flexible ship loading problem with transfer vehicle assignment and scheduling. Transp. Res. Part B-Methodol.. 111, 113–134.

Iris, C., Lam, J.S.L., 2019. Recoverable robustness in weekly berth and quay crane planning. Transp. Res. Part B-Methodol. 122, 365–389.

Jiang, X.J., Jin, J.G., 2017. A branch-and-price method for integrated yard crane deployment and container allocation in transshipment yards. Transp. Res. Part B-Methodol. 98, 62–75.

Jin, B., 2020. On the integer programming formulation for the relaxed restricted container relocation problem. Eur. J. Oper. Res. 281 (2). 475–C482

Jin, B., Zhu, W., Lim, A., 2015. Solving the container relocation problem by an improved greedy look-ahead heuristic. Eur. J. Oper. Res. 240 (3), 837–847.

Jovanovic, R., Tuba, M., Voß, S., 2019. An efficient ant colony optimization algorithm for the blocks relocation problem. Eur. J. Oper. Res 274 (1), 78–90.

Jovanovic, R., Voß, S., 2014. A chain heuristic for the blocks relocation problem. Comput. Ind. Eng 75, 79–86.

Kim, K.H., Hong, G.P., 2006. A heuristic rule for relocating blocks. Comput. Oper. Res. 33 (4), 940–954.

Kim, K.H., Kim, H.B., 2002. The optimal sizing of the storage space and handling facilities for import containers. Transp. Res. Part B-Methodol. 36 (9), 821–835.

Kim, K.H., Park, Y.M., Ryu, K.R., 2000. Deriving decision rules to locate export containers in container yards. Eur. J. Oper. 124 (1), 89–101.

Kouvelas, A., Saeedmanesh, M., Geroliminis, N., 2017. Enhancing model-based feedback perimeter control with data-driven online adaptive optimization. Transp. Res. Part B-Methodol. 96, 26–45.

Lee, C.Y., Song, D.P., 2017. Ocean container transport in global supply chains: overview and research opportunities. Transp. Res. Part B-Methodol. 95, 442–474.

Lee, Y., Chao, S.L., 2009. A neighborhood search heuristic for pre-marshalling export containers. Eur. J. Oper. Res. 196 (2), 468–475.

Lee, Y., Lee, Y.J., 2010. A heuristic for retrieving containers from a yard. Comput. Oper. Res. 37 (6), 1139–1147.

Lehnfeld, J., Knust, S., 2014. Loading, unloading and premarshalling of stacks in storage areas: Survey and classification. Eur. J. Oper. Res. 239 (2), 297–312.

Liu, M., Lee, C.Y., Zhang, Z., Chu, C., 2016. Bi-objective optimization for the container terminal integrated planning. Transp. Res. Part B-Methodol. 93, 720–749.

Morrison, D.R., Jacobson, S.H., Sauppe, J.J., Sewell, E.C., 2016. Branch-and-bound algorithms: a survey of recent advances in searching, branching, and pruning. Discret. Optim. 19, 79–102.

Siripirote, T., Sumalee, A., Ho, H.W., Lam, W.H.K., 2015. Statistical approach for activity-based model calibration based on plate scanning and traffic counts data. Transp. Res. Part B-Methodol. 78, 280–300.

Sopasakis, A., Katsoulakis, M.A., 2016. Information metrics for improved traffic model fidelity through sensitivity analysis and data assimilation. Transp. Res. Part B-Methodol. 86, 1–18.
Steenken, D., Voß, S., Stahlbock, R., 2004. Container terminal operation and operations research c a classification and literature review. OR Spectrum 26 (1), 3–49.
Tanaka, S., Takii, K., 2016. A faster branch-and-bound algorithm for the block relocation problem. IEEE Trans. Autom. Sci. Eng. 13 (1), 181–190.
Ting, C.J., Wu, K.C., 2017. Optimizing container relocation operations at container yards with beam search. Transp. Res. Part E-Logist. Transp. Rev. 103, 17–31.
Virgile, G., Cynthia, B., Patrick, J., 2018. Yard crane scheduling for container storage, retrieval, and relocation. Eur. J. Oper. Res 271 (1), 288–316.
Wang, S., Liu, Z., Bell, M.G.H., 2015. Profit-based maritime container assignment models for liner shipping networks. Transp. Res. Part B-Methodol. 72, 59–76.
Wang, T.S., Wang, X.C., Meng, Q., 2018. Joint berth allocation and quay crane assignment under different carbon taxation policies. Transp. Res. Part B-Methodol. 117, 18–36.
Wei, D., Liu, H., 2013. Analysis of asymmetric driving behavior using a self-learning approach. Transp. Res. Part B-Methodol. 47, 1–14.
Yin, K., Wang, W., Bruce Wang, X.B., Adams, T.M., 2015. Link travel time inference using entry/exit information of trips on a network. Transp. Res. Part B-Methodol. 80, 303–321.
Zehendner, E., Feillet, D., Jaillet, P., 2016. An algorithm with performance guarantee for the online container relocation problem. Eur. J. Oper. Res. 259 (1), 48–62.
Zhang, C., Chen, W., Shi, L., Zheng, L., 2010. A note on deriving decision rules to locate export containers in container yards. Eur. J. Oper. 205 (2), 483–485.
Zhang, C., Liu, J., Wan, Y., Murty, K.G., Linn, R.J., 2003. Storage space allocation in container terminals. Transp. Res. Part B-Methodol. 37 (10), 883–903.
Zhang, C., Wan, Y.W., Liu, J., Linn, R.J., 2002. Dynamic crane deployment in container storage yards. Transp. Res. Part B-Methodol. 36 (6), 537–555.
Zhang, C., Wu, T., Kim, K.H., Miao, L., 2014. Conservative allocation models for outbound containers in container terminals. Eur. J. Oper. Res. 238 (1), 155–165.
Zhang, C., Wu, T., Zhong, M., Zheng, L., Miao, L., 2014. Location assignment for outbound containers with adjusted weight proportion. Comput. Oper. Res 52, 84–93.
Zhen, L., 2016. Modeling of yard congestion and optimization of yard template in container ports. Transp. Transp. Res. Part B-Methodol. 90, 83–104.
Zhen, L., Liang, Z., Zhuge, D., Lee, L.H., Chew, E.P., 2017. Daily berth planning in a tidal port with channel flow control. Transp. Res. Part B-Methodol. 106, 193–217.
VanZheng, F., Zuylen, H., 2013. Urban link travel time estimation based on sparse probe vehicle data. Transp. Res. Part C-Emerg. Technol. 31, 145–157.
Zheng, F.F., Man, X.Y., Chu, F., Liu, M., Chu, C.B., 2018. A two-stage stochastic programming for single yard crane scheduling with uncertain release times of retrieval tasks. Int. J. Prod. Res 1–16.
Zhu, W., Qin, H., Lim, A., Zhang, H., 2012. Iterative deepening a* algorithms for the container relocation problem. IEEE Trans. Autom. Sci. Eng. 9 (4), 710–722.