Universidade Estadual de Campinas
Instituto de Computação

Kent Emershon Yucra Quispe

An Exact Algorithm for the Blocks Relocation Problem with New Lower Bounds

Um algoritmo exato para o Problema de Realocação de Blocos usando novos limitantes inferiores

CAMPINAS
2018

Kent Emershon Yucra Quispe

An Exact Algorithm for the Blocks Relocation Problem with New Lower Bounds

Um algoritmo exato para o Problema de Realocação de Blocos usando novos limitantes inferiores

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Eduardo Candido Xavier
Co-supervisor/Coorientador: Profa. Dra. Carla Negri Lintzmayer

Este exemplar corresponde à versão final da Dissertação defendida por Kent Emershon Yucra Quispe e orientada pelo Prof. Dr. Eduardo Candido Xavier.

CAMPINAS
2018

Agência(s) de fomento e nº(s) de processo(s): CAPES

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Y9e

Yucra Quispe, Kent Emershon, 1992-
An exact algorithm for the blocks relocation problem with new lower bounds / Kent Emershon Yucra Quispe. – Campinas, SP : [s.n.], 2018.

Orientador: Eduardo Candido Xavier.
Coorientador: Carla Negri Lintzmayer.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Otimização combinatória. 2. Heurística (Computação). 3. Problema de realocação de blocos. I. Xavier, Eduardo Candido, 1979-. II. Lintzmayer, Carla Negri, 1990-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Um algoritmo exato para o problema de realocação de blocos usando novos limitantes inferiores
Palavras-chave em inglês:
Combinatorial optimization
Heuristic (Computer science)
Blocks relocation problem
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Eduardo Candido Xavier [Orientador]
Cid Carvalho de Souza
Luidi Gelabert Simonetti
Data de defesa: 20-04-2018
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Kent Emershon Yucra Quispe

An Exact Algorithm for the Blocks Relocation Problem with New Lower Bounds

Um algoritmo exato para o Problema de Realocação de Blocos usando novos limitantes inferiores

Banca Examinadora:

- Prof. Dr. Eduardo Candido Xavier
  Instituto de Computação – UNICAMP
- Prof. Dr. Cid Carvalho de Souza
  Instituto de Computação – UNICAMP
- Prof. Dr. Luidi Gelabert Simonetti
  Departamento de Engenharia de Sistemas e Computação – UFRJ

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 20 de abril de 2018

Resumo

O Problema de Realocação de Blocos é um problema importante em sistemas de armazenamento. Um exemplo de entrada para este problema consiste em um conjunto de blocos distribuídos em pilhas, onde cada bloco é identificado por um número que representa sua prioridade de recuperação e todas as pilhas têm um mesmo limite de altura. Apenas blocos no topo de uma pilha podem ser movidos, com dois tipos de movimentos: ou um bloco é recuperado, o que ocorre quando ele tem a mais alta prioridade de recuperação entre os blocos empilhados, ou um bloco é realocado do topo de uma pilha para o topo de outra pilha. O objetivo é recuperar todos os blocos, respeitando sua prioridade de recuperação e executando o menor número de realocações. Resolver este problema é crítico em sistemas de armazenamento, pois economiza tempo e recursos operacionais. Apresentamos dois novos limitantes inferiores para o número de realocações em uma solução ótima. Implementamos um algoritmo de deepening A* usando esses limites inferiores propostos e outros limites inferiores bem conhecidos da literatura. Foi realizado um extenso conjunto de experimentos computacionais mostrando que os novos limites inferiores melhoram o desempenho do algoritmo exato, resolvendo mais instâncias otimamente do que quando usando outros limites inferiores na mesma quantidade de tempo.

Abstract

The Blocks Relocation Problem is an important problem in storage systems. An input instance for this problem consists of a set of blocks distributed in stacks where each block is identified by a retrieval priority number and each stack has the same maximum height limit. Only blocks at the top of a stack can be moved: either a block is retrieved, if it has the highest retrieval priority among the stacked blocks, or it is relocated to the top of another stack. The objective is to retrieve all the blocks, respecting their retrieval priority while performing the minimum number of relocations. Solving this problem is critical in storage systems because it saves operational time and resources. We present two new lower bounds for the number of relocations of an optimal solution. We implemented an iterative deepening A* algorithm using these new proposed lower bounds and other well-known lower bounds from the literature. We performed an extensive set of computational experiments showing that the new lower bounds improve the performance of the exact algorithm, solving to optimality more instances than when using other lower bounds in the same amount of time.

List of Figures

1.1 An instance for the BRP with $S = 6$ (stacks), $H = 3$ (height), and $N = 15$ (blocks). 11

2.1 Original and abstract search space. 19

4.1 Example of auxiliary matrix created to calculate LB-LIS. In this example, assume $t = 1$ and it is block $B_{4,1}$ and that $B_{4,2} = 12$, $B_{4,3} = 11$, $B_{4,4} = 10$, $B_{4,5} = 3$, $B_{4,6} = 7$, $B_{4,7} = 2$ (yellow column). Also assume that $t_1 = 13$, $t_2 = 9$, $t_3 = 8$, $t_5 = 6$, and $t_6 = 5$ (red row). The numbers inside the circles are the size of the longest increasing subsequence for each $B_{4,y'}$, for $y' \geq 2$. 26

4.2 Example of how to compute LB-LIS. 28

5.1 Let I (left) be an instance of the BRP, and consider the set $B = \{11, 12, 13, 14, 15, 16, 17\}$. The instance generated by abState(I, B) is shown to the right where blocks $11, 12, \ldots, 17$ are re-labeled respectively to $1, 2, \ldots, 7$. 30

6.1 Results using LB3, LB4, LB-LIS, and LB-PDB. 41

6.2 Results using LB-LIS, LIS-PDB, and LIS-PDB-L. 43

List of Tables

5.1 Enhancement of the PDB for instances with 10 blocks. 34

6.1 Time to generate the PDB for different values of N'. We also present the number of unique instances in the PDB. 40
6.2 Results of the exact algorithm using different lower bounds: LB3, LB4, LB-LIS, and LB-PDB. "AVG ALL" is the average number of relocations of the best solutions found by each algorithm considering all instances. "Number OPT" is the number of optimal solutions found. "%OPT EASY" (resp. "%OPT HARD") is the percentage of optimal solutions found for easy (resp. hard) instances. "AVG HARD" is the average number of relocations considering hard instances. 41
6.3 Times and values produced by each lower bound for each initial configuration of the 12500 instances in our instance set. We present the sum of the values (SUM) and the average (AVG) among the 12500 instances. 42
6.4 Comparing performances of LB-LIS-PDB with different N'. 42
6.5 Comparing performances of LB-LIS-PDB-L with different k'. 43
6.6 Comparing performances of LB-LIS, LIS-PDB, and LIS-PDB-L. 43

Contents

1 Basics concepts and literature review about the problem **10**
1.1 Introduction 10
1.2 Problem Description and Basic Definitions 11
1.3 Literature Review 12
1.4 Contributions and text organization 14

2 Exact Exponential Time Algorithms to Solve Combinatorial Optimization Problems **15**
2.1 State Space Search 15
2.2 Branch-and-Bound 16
2.3 Iterative Deepening Depth-First Search 17
2.4 Pattern Databases 18

3 Implemented Algorithms **20**
3.1 Iterative Deepening A* Algorithm 20
3.1.1 Branching Strategy at the B&B algorithm 21
3.1.2 Pruning at the B&B algorithm 21
3.1.3 Initial solution 21

4 Lower Bounds **24**
4.1 LB1 24
4.2 LB2 and LB3 24
4.3 LB4 25
4.4 LB-LIS 26

5 Pattern Databases **29**
5.1 Abstract state of the BRP 29
5.2 Building the Pattern Database 30
5.3 Using the PDB to compute a lower bound 33
5.4 Improving the Exact Algorithm 36

6 Computational Results **38**
6.1 Instance Set 38
6.2 Generating the PDB 39
6.3 Algorithm Evaluation with Different Lower Bounds 39
6.4 Improving the Exact Algorithm Using the PDB 42

7 Conclusions and Future Works **44**

Chapter 1

Basics concepts and literature review about the problem

This chapter is structured as follows: in Section 1.1 we explain the importance of the Blocks Relocation Problem (BRP) in storage systems, in Section 1.2 we show a formal description of the problem, in Section 1.3 we review the most important works in the literature for the BRP, and in Section 1.4 we list the contributions of the thesis and the organization of rest of the text.

1.1 Introduction

This work is focused on the *Blocks Relocation Problem (BRP)*, also known as Container Relocation Problem in the literature, which generally emerges from storage systems. In storage systems, there are several types of items to be stored, such as containers and pallets. We will refer to these items as *blocks* throughout the text. We consider that the blocks are stored in a series of stacks, where, at each stack, one block is above another or at the bottom of the stack. This is called the *stacking area*, which is the most common type of storage system for containers. This type of storage only allows one to access the top block of a stack, which can be *relocated* to the top of another stack or can be removed and placed outside the stacking area, in a move called *retrieval*.

Consider a container terminal where blocks have to be retrieved from the stacking area and loaded onto trucks, following a given precedence order called *retrieval priority*. The precedence of these items may be motivated by several factors, such as the arriving order of container transportation trucks or the delivery order of containers in a ship. Given a stacking area and the precedence order of the blocks, the objective of BRP is to minimize the number of relocations in order to retrieve all blocks.

The BRP is known to be NP-Hard [2]. There exist many works that tried to solve the BRP with different approaches. Some of them presented new lower bounds, which are used by exact algorithms to solve the BRP [11, 19], others presented formulations in integer linear programming [2, 18], and some of them presented heuristic approaches [11, 1, 10].

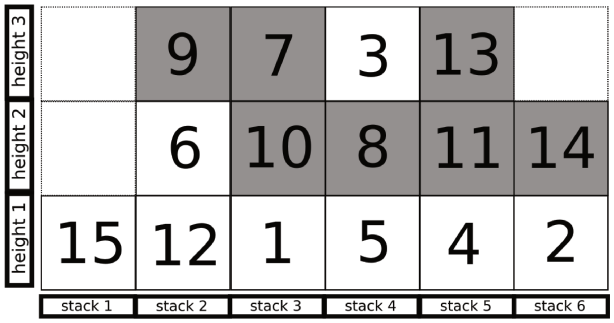


Figure 1.1: An instance for the BRP with $S = 6$ (stacks), $H = 3$ (height), and $N = 15$ (blocks).

1.2 Problem Description and Basic Definitions

Formally, in the BRP we are given N blocks $b_1, b_2, \ldots, b_N$ distributed in a stacking area consisting of S stacks $s_1, s_2, \ldots, s_S$, with a maximum height of H for each stack. The *height* of a stack s_x, denoted by $\text{height}(s_x)$, is the number of blocks stacked on it. Thus, in any possible configuration of the stacking area we must have $\text{height}(s_x) \leq H$ for all $1 \leq x \leq S$. Each block b_i, for $1 \leq i \leq N$, has a *retrieval priority* defined as i, which indicates its retrieval order. Therefore, the lowest value means the highest priority, so b_1 is the first block to be retrieved. In the version of the problem we consider, the retrieval priorities are unique. There are other versions of the problem where different blocks may have a same retrieval priority, but we do not consider them in this work.

We denote by t the block with the highest retrieval priority in the stacking area, also called *target block* of the stacking area, or target block of the instance. We call *target stack* the stack containing block t. Similarly, we denote by t_x the block with the highest retrieval priority in stack s_x, also called *target block of stack* s_x. We also use the notation $\text{B}_{x,y}$ to represent a block that is in stack s_x at height y.

Since we only have access to the block at the top of any stack, there are only two available moves. A *relocation* is a move (or action) $s_x \Rightarrow s_{x'}$ that takes the top block of stack s_x and puts it at the top of stack $s_{x'}$, where it must be valid that $\text{height}(s_{x'}) < H$ before the relocation. Also, a relocation from s_x is allowed only if the target block t is in s_x, so that we can retrieve t latter, after relocating all blocks above it. The BRP problem under this restriction is still NP-hard and this assumption is used in several works on the literature [2]. A *retrieval* is a move (or action) $s_x \Rightarrow s_0$ that removes the top block of stack s_x from the stacking area if such block is t (s_0 is an artificial stack). It is interesting to note that it is an open question the complexity class of the decision version of the problem, where one has to decide if there is a sequence of movements that clear an initial stacking area if $S * H - H + 1 \leq N$.

An instance of the BRP consists of the dimensions S and H of the stacking area, the number N of blocks stored, and an initial configuration of these blocks. A solution to an instance of the BRP is a sequence of relocations and retrievals that clears the initial stacking area. Note that the number of retrievals is constant and equal to N, so the goal is to retrieve all the blocks respecting their retrieval priorities using the minimum number of relocations. See Figure 1.1 for an example. In the example, we have $t = 1$ and we can

only relocate blocks 7 and 10, in this order, before retrieving block 1. After relocating blocks 7 and 10, we retrieve block 1, and then the new target block becomes $t = 2$. A possible sequence of relocations to retrieve blocks 1, 2, 3, and 4 is $\mathcal{S} = \{(s_3 \Rightarrow s_1), (s_3 \Rightarrow s_1), (s_3 \Rightarrow s_0), (s_6 \Rightarrow s_3), (s_6 \Rightarrow s_0), (s_4 \Rightarrow s_0), (s_5 \Rightarrow s_3), (s_5 \Rightarrow s_3), (s_5 \Rightarrow s_0)\}$.

Let b_i be a block in some stack s_x and let b_j be some block that is placed below b in the same stack, i.e., the height of b_i in s_x is greater than the height of b_j. If $i > j$, then b_i is called a *blocking block*. In Figure 1.1, blocks $7, 8, 9, 10$, 11, 13, and 14 are blocking blocks.

1.3 Literature Review

Kim and Hong [11] proposed two variations for the BRP. In the first one, each block has a unique retrieval priority and in the second one, two or more blocks can have the same retrieval priority. They proposed a branch and bound algorithm for both variants of the BRP. In the Branch and Bound Algorithm, the order in which the states (which are the instances of the problem) of the state space (all the states that can be reached applying one or more actions over the initial instance) are explored is inspired by the depth-first search and backtracking strategies used in many state space search algorithms. That is, given all the unexplored states, we pick the one reachable from the most recent explored state to be the next one explored. If there exist two or more states that fit this definition, the next state to be explored is the one with minimum sum of the number of performed relocations so far plus the number of blocking blocks in on it.

In the same paper the authors propose a fast heuristic rule as a subroutine to create a solution for the BRP. The *expected number of additional relocations* (ENAR) calculates a valid lower bound in the number of relocations from the current state to the goal state. In order to create a solution for the BRP the authors use ENAR as follows: suppose that we want to relocate the top block tb of the target stack; for that, we can relocate such block to at most $S-1$ other stacks, so we simulate such relocation creating at most $S-1$ new states; now, we calculate the ENAR of each new state; from all the stacks where we can relocate the block tb, we pick a stack in which the ENAR value at the state where the block tb was relocated to such stack is minimum. The authors compared results of those two proposed methods.

Caserta, Schwarze, and VoB [1] proposed a binary representation of the stacking area, which simplifies the transition from the current state of the stacking area to a state generated by a relocation or retrieval. In the binary $(N+S)$x$(N+S)$ matrix M, the first N rows and columns represent the blocks $b_1, b_2, \cdots, b_N$ in the stacking area. The last S rows and columns represent artificial blocks at an artificial height 0 of the stacking area. More formally, the row and column $N+i$ represent the artificial block b_{N+i} with height 0 in the stack s_i. Such binary matrix M is filled as follows: $M_{i,j} = 1$ if the block b_i is below the block b_j at the same stack; otherwise, $M_{i,j} = 0$. They developed a lookahead mechanism (heuristic) that is adapted to this binary representation. The heuristic rule first calculates the target block of a stack (remember that the target block of each stack is the block that has the highest priority); if a stack is empty, then its target block is

$N+1$. The following rules are applied to relocate the block b_i that is at the top of the target stack:

1. Relocate b_i to the stack with the minimum target block such that b_i does not become a blocking block where it is relocated.

2. If there are no stacks with this property, then relocate b_i to a stack with maximum target block.

The authors compare their results with another method created by themselves, called the corridor method [3]. They showed that the running time and the average of the number of relocations is better than the corridor method.

Lee and Lee [15] introduced a new variant of the problem where each relocation have a cost, which is the distance between the two stacks over which the relocation is performed. The authors first present a greedy heuristic, which we describe next. If the target block is at the top of a stack, then we retrieve it; otherwise, we relocate the top block tb of the target stack s_x to the nearest available stack $s_{x'}$ such that $s_{x'} < H$. If we have two stacks available, one stack $s_{x'_1}$ to the left of s_x and another stack $s_{x'_2}$ to the right where $|s_{x'_1} - s_x|$ is equal to $|s_{x'_2} - s_x|$, then pick the stack with less height to relocate tb. The authors then present a method to reduce the number of relocations of the heuristic. Lee and Lee also created a mixed integer program in order to reduce the working time for the crane. The crane is a machine that performs the relocations, the working time is proportional to the distance of two stack for which perform the relocation, for example if we relocated a block of the stack s_i to the stack s_j the working time is $|i-j| * k$. They used CPLEX to solve integer program, which decreased in 5% the working time of the crane compared with the heuristic.

Caserta and Voß [3] presented a new heuristic named corridor method using ideas from genetic algorithms. After that, the same authors introduced a two-dimensional corridor using the corridor method without limiting the height of the stacks [4]. They also proved that the BRP is NP-Hard [2] and presented a binary linear programming model that generates solutions for small instances. To break this limitation, realistic assumptions are introduced, which allows them to create a new binary linear programming model and a new heuristic to get good solutions for medium size instances.

Javanovic and VoB [10] proposed a new heuristic approach, which considers not only the current block to be relocated but also the next block to be relocated. They named this idea Min-Max heuristic. This heuristic reduces on average 5% on the number of relocations comparing with the heuristic proposed by Caserta *et al.* [2].

Exposito and Batista [7] presented two exact algorithms based on the A* search framework, one for the restricted BRP (where we can only relocate the top block of the target stack) and one for the unrestricted one (where we can relocate the top block of any stack). The A* search algorithm for the restricted BRP use as a lower bound the one proposed by Zhu *et al.* [19], which is explained in Chapter 4 (we call it Lower Bound 3). The A* for the unrestricted BRP use as a lower bound the number of blocking blocks only. As an upper bound for the two algorithms they use a simple heuristic, which creates a solution as follows: if the target block is at the top of a stack, then we retrieve it; otherwise,

we relocate the block at the top of the target stack to another random stack s_r where height$(s_r) < T$. With these, they were able to get 62% of optimal solutions in a dataset created by themselves.

Tanaka *et al.* [17] presented a new lower bound based on previous lower bounds that were created by Kim and Hong [11] and Zhu *et al.* [19]. The use of this new lower bound on an exact algorithm results in finding 1.848% more optimal instances than using the previous lower bounds.

Following the same line of research, Tanaka and Mizuno [16] proposed an exact algorithm for the unrestricted BRP with distinct priorities. They proposed three types of dominance properties (for transitive relocation, independent relocations, and retrievals) to eliminate unnecessary nodes in the search tree. They also improved the lower bound proposed by Forster and Bortfeldt [9]. With those two enhancements they solve a large number of benchmark instances, solving more instances to optimality than previous approaches from the literature.

1.4 Contributions and text organization

In this work, we create two new lower bounds for the BRP. We also explore a new approach to tackle the problem by memorizing a part of the search space, named Pattern Database [8]. We use these two lower bounds in a general exhaustive search, iterative deepening A*, which avoids parts of the search space where the optimal solution is not present. The use of the new lower bound, the pattern database, as well as a method to stop the search earlier, saved a significant amount of time in the exhaustive search.

The rest of the text is structured as follows. In Chapter 2 we present all the concepts that are necessary to better explain our contributions. In Chapter 3 we give some details about the algorithms we implemented. In Chapter 4 we present the three most important lower bounds found in the literature and expose one of our new proposed lower bound. In Chapter 5 we explain the concept of pattern databases and expose the other new proposed lower bound. In Chapter 6 we show all the computational results of our experiments using the two new lower bounds proposed comparing such results with existing lower bounds explained in the Chapter 4. At last, in Chapter 7 we draw our conclusions.

Chapter 2

Exact Exponential Time Algorithms to Solve Combinatorial Optimization Problems

In this chapter, we present some exact algorithms used to solve NP-Hard combinatorial optimization problems, such as the BRP. Assuming P $\neq$ NP, there is no polynomial time algorithm to solve these problems, yet some exponential time algorithms are useful in practice to solve them. We first show how a combinatorial problem can be modeled in terms of State Space Search. Then, we will show some basic exact exponential time algorithms, some of them using the State Space Search concept and some of them not. Many researches present exact exponential algorithms as a main structure to solve the Blocks Relocation Problem, as we showed in Section 1.3. The main algorithm that we use in this dissertation is an exact exponential time algorithm where we use our two new lower bounds.

2.1 State Space Search

A *state space* is the set of all possible configurations/states that an instance of the problem can reach when the available actions are applied. Such instance is considered the initial state of the state space. As we mentioned, we can explore the state space by applying actions on a state s of the problem to get to a new state s'. Thus, the state space can be seen as a graph in which two states are connected if there is an action that transforms one state into the other.

A state space comprises six components: S, I, G, actions(s), successor(s, a), and cost(s, a). Set S contains all possible states (including the initial one) that we can reach from the initial state performing one or more actions. Set I contais the initial states of the problem ($I \subseteq S$); for example, one initial state of the BRP is shown in Figure 1.1. Set G contains the goal states of the instance to be solved ($G \subseteq S$). In the case of BRP, there is only one goal state, which is the empty stacking area. The actions function, actions(s), gives the actions that can be performed over a state s. In the BRP, given the target stack s_x if t is the top block then only one action can be performed, which is the retrieval

of t. If t is not the top block, then for each one of the other stacks we have an action corresponding to the relocation of the top block of the target stack to that stack. The successor function, successor(s, a), gives the states reachable when action a is applied over state s. In the case of BRP, each action (relocation or retrieval) reaches only one state when it is applied. The cost function, cost(s, a), gives the cost of transforming a state s into another state $s' \in$ successor(s, a). In the case of BRP, a *retrieval* and a *relocation* have cost 0 and 1, respectively.

Summarizing, for BRP, a configuration/state is any stacking area, the initial state is the initial stacking area, the goal state is the empty stacking area, and the possible actions are retrievals and relocations, which cost 0 and 1, respectively.

State space algorithms explore all the state space searching for a path between an initial state and a goal state. The best-solution path is determined in terms of the size of the path, time, resource consumption, or other aspects according to the problem. Real-world planning problems (e.g., Vehicle routing, crew scheduling, production planning) are combinatorial optimization problems that can be instantiated in terms of state space search.

There exist many algorithms that explore a state space, but the most common methods are Depth First Search (DFS) and Breadth First Search (BFS), which are unfeasible methods for search space problems with exponential number of states in terms of the size of the input instance. For these, we need more sophisticated algorithms, such as A*, Branch-and-Bound, or Minimax.

2.2 Branch-and-Bound

A Branch-and-Bound (B&B) algorithm explores the state space of a problem in order to find the best solution. Since the number of states is normally exponential with respect to the size of the input, it uses bounds and the current best solution to avoid exploration of a portion of the state space.

A B&B algorithm for a minimization problem such as BRP consists of four main parts [5]. Let s be the initial state and g be the goal state of BRP. The first part consists of a *bounding function* for any given state s', which is provided for a given state subspace S' (a set of all states that we can reach when applying one or more actions to the state s'). More formally, let s' be a state reached after a positive number of actions is applied to s. The bounding function is calculated over state s' and it returns a lower bound for the best solution from s' to g, obtainable in the subspace S'. The bounding function is the most important part of any B&B algorithm because low-quality bounding functions cannot contribute to the process of stopping the search. The value of a bounding function for a given state s' is ideally equal to the value of the *best* solution. However, this state s' is an instance of the BRP and, as we mentioned in Chapter 1, solving this instance optimally is an NP-Hard problem. Therefore, the goal is to find a good bounding function using a limited amount of computation (i.e., polynomial time algorithm).

The second part of a B&B is a *strategy for selecting* the next state to be explored. We usually desire to keep the number of explored nodes in the state space search low and

save memory capacity of the computer.

The third part is a *branching rule*, to be applied if the state under investigation cannot be discarded. A branching rule is a subdivision of the search space through the addition of constraints that divide it into some search subspaces. When we apply a branching rule, we are sure that the search subspace is smaller and finite. Typically, the search subspaces are disjoint and, in two different search subspaces, we do not find the same feasible solution.

The last part of a B&B algorithm is very crucial. It consists of producing a *primal solution*. In general, any heuristic may be used, but if it is not possible to find one, then we can estimate the cost of a solution with an infinite value depending on the problem that we are trying to solve. We use such *primal solution* as an initial upper bound for the BRP and we can update the upper bound each time we find a better solution during the search. The upper bound helps the algorithm to "cut" the search when we are in a branch where the value of the lower bound is greater than the upper bound; this means that we are sure that through this branch we cannot reach a solution of cost better than the cost of the known upper bound (which is a solution, not necessary optimal for the BPR).

2.3 Iterative Deepening Depth-First Search

Iterative Deepening Depth-First Search (IDDFS) is a state space search strategy where the depth of the DFS is limited. Such depth-limited version of DFS runs many times, and in each time we increase the depth until the goal is found (the optimal solution for BRP). The complexity of the algorithm will be given in terms of the branching factor and the depth of the state space search, described next.

The node branching factor (bf) of a problem is the average number of new states generated by applying one single action to a given state. The depth (d) is the length of the shortest sequence of actions that map the initial state into the goal state. The time complexity of a search algorithm in this model of computation is simply the number of states that are expanded in the search. Furthermore, we assume that the space complexity of the algorithm is the number of states that must be stored. Next we present and analyze two classic algorithms in the field of computer science, Breadth-First Search (BFS) and Depth-First Search (DFS), in terms of bf and d. IDDFS takes advantage of BFS and DFS, and we use this algorithm together with a Branch and Bound algorithm as our method to test our two new lower bounds.

Breadth-First Search (BFS) expands all the states that are one action distant from the initial state, then expands all states that are two actions distant from the initial state, then three actions, and so on, until it reaches the goal state. In the worst case, BFS generates bf nodes in the first level, bf^2 nodes in the second level, and so on, until it reaches depth d, where bf^d nodes are generated and a solution is found. Therefore, the worst-case time complexity is $O(bf^{d+1})$. One main characteristic of BFS is that we have to save all nodes expanded before finding a solution, so we have to expand $bf^1+bf^2+\cdots+bf^d$ nodes, which means the space complexity of all of them is $O(bf^{d+1})$. BFS is exponential in the space complexity, therefore it is not practical for some problems because too much memory is

necessary and in practice such memory is not available.

Depth-First Search (DFS) avoids the memory limitation of BFS and it works as follows. The most recently explored node is expanded generating bf new nodes; after that, we have to expand one node among these bf new nodes in an arbitrary order (e.g., lexicography order). We keep this process until no more nodes can be expanded, then we backtrack to the next most recently expanded node, and we repeat this action until the goal node t is found or all states of the state space are explored. The property of the DFS is that we keep in memory only the path between the initial state and the current state where the DFS is, which means that the space complexity of the DFS is $O(d \times bf)$ but note that the worst case time complexity to find a solution is also $O(bf^{d+1})$.

Korf [13] took advantage of those two searches and created a new algorithm named Iterative Deepening Depth-First Search. The algorithm is described as follows: first, perform a DFS with a maximum depth d' calculated with some function (in the case of the BRP, we can assign to d' a lower bound value on the instance) at the beginning of the algorithm; if a goal state is found at a depth d', then we stop the search; otherwise, perform a second DFS with a maximum depth $d'+1$; if a goal state is found at a depth of $d'+1$, then stop the search; otherwise, perform a third DFS with a maximum depth $d'+2$, and so on. IDDFS seems to have a great disadvantage of wasting computation to find the goal depth because it repeats the exploration of the same nodes in different Depth-First Searches. However, its time complexity is the same of BFS and DFS ($O((d-d') \times bf^{d+1})$). IDDFS uses DFS as its main structure, so it needs only to store the nodes which represent the current path analyzed. Since the maximum depth of the path is d, the space complexity is $O(d \times bf)$ and the time complexity remains exponential.

2.4 Pattern Databases

In this section, we explain some concepts needed to present one of our new lower bounds, shown in Chapter 5. A Pattern Database (PDB) stores a collection of solutions of a sub-space state. To explain how such sub-space state is created we will introduce first the concept of abstraction.

An *abstraction* is a function ϕ that maps each state s from the state space $\mathcal{S}$ to some other state *as* from the abstract state space (sub-space state) $\mathcal{AS}$ which preserves paths and goal states. Let a be an action, and let s and s' be states from $\mathcal{S}$. Let $c_a(s, s')$ be the cost of applying an action a to s and obtaining s' as a result at the state space $\mathcal{S}$, and let $c'_a(s, s')$ be the cost of applying an action a to s and obtaining s' as a result at the abstract state space $\mathcal{AS}$. The following must holds:

- If $s \in \mathcal{S}$, then $\phi(s) \in \mathcal{AS}$.
- The cost $c_a(s, s')$ of applying a on s in order to reach s' is smaller than $c'_a(\phi(s), \phi(s'))$ the cost of applying a on $\phi(s) \in \mathcal{AS}$ in order to reach $\phi(s') \in \mathcal{AS}$.

Abstraction replaces one state space with another state space, called *abstract state space*, that is easier to be searched. Figure 2.1 shows the idea and the definition of the

abstraction. Note that the abstract search space is supposed to be smaller than the original search space.

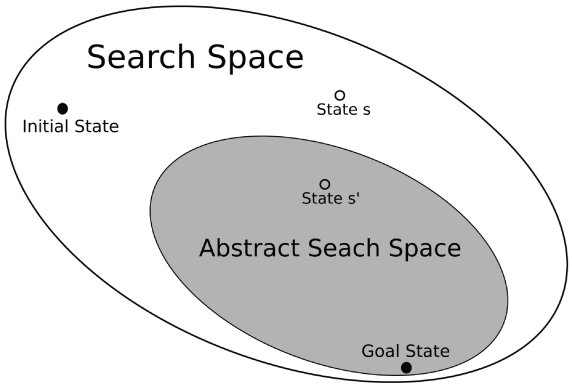


Figure 2.1: Original and abstract search space.

We illustrate the idea of the abstraction with an example using the well-known Tower of Hanoi problem, which consists of three rods that hold a number of different-sized disks (n disks). Initially, the n disks are all stacked on one rod where the largest disk is at the bottom of the stack, the second largest one is above the largest one, and so on, until the smallest disk is at the top. The task is to transfer all the disks to another rod such that they keep the same initial configuration. The constraints to move the disks are that only the top disk on any rod can be moved at any time and that larger disks can never be placed on top of smaller disks. For the 3-rod version, there is a simple deterministic algorithm that provably returns an optimal solution. The minimum number of moves is $2^n - 1$. For the 4-rod version of the problem, there exists a deterministic algorithm for finding a solution and a conjecture that it is optimal, but the conjecture remains open.

Let $\mathcal{S}$ be the set of all states that can be reached from an initial configuration with n disks for the 4-rod Tower of Hanoi and let $\mathcal{AS}$ be the set of all states that can be reached from an initial configuration with m disks for the 4-rod Tower of Hanoi, such that $m < n$. There exists an abstraction function that takes m disks of any state $s \in \mathcal{S}$ and can be mapped to a state $s' \in \mathcal{AS}$. Such function works in the following way. It selects any m disks from s and then it resizes each of them so that they have sizes from 1 to m (keeping the relative sizes). To create a state s', we place these m disks in the same corresponding rod they were in s keeping their relative positions. Suppose that we have $n = 5$ and $m = 3$. We can, for instance, select disks of sizes 2, 4, and 5 from s which are in rods 1, 4, and 4, respectively, with disk 4 above disk 5. Then we resize them to sizes 1, 2, and 3, respectively. To create state s', we place disks 1, 2, and 3 in rods 1, 4, and 4, respectively, with disk 2 above disk 3.

We can compute the PDB with a BFS algorithm starting at the target pattern and until we reach all the states in the *abstract space state*. The complexity of such computation using BFS is exponential in space and time. This technique is frequently used in order have lower/upper bounds in an exact algorithm.

Chapter 3

Implemented Algorithms

In this chapter we present the general idea of the exact algorithm we used in our computational experiments.

3.1 Iterative Deepening A* Algorithm

In this section we describe the exact algorithm we used in our computational experiments, which is the Iterative Deepening A* (IDA*) algorithm. The IDA* was first introduced by Korf [12] in 1985 as a general search method, and has been applied to several problems. For the BRP problem, it was first considered by Zhu et al. [19], and then by other authors such as Tanaka and Takii [17].

In general, these search algorithms use upper and lower bounds in order to avoid the exploration of some parts of the search space where optimal solutions are not present. An outline of the exact algorithm is described below:

1. Given the initial stacking area, calculate a lower bound cost_{cur} and an upper bound UB, which is the cost (number of relocations) of a heuristic solution.

2. Use a branch and bound algorithm (B&B) to search for a solution whose number of relocations is not greater than cost_{cur}. During the execution of the B&B, if a feasible solution with value better than UB is found, then update UB. In the B&B execution, nodes are pruned using cost_{cur} as a maximum cost, i.e., if a node is such that its lower bound is greater than cost_{cur}, then the node is pruned from the search.

3. If no solution with cost equal to cost_{cur} was found, then update cost_{cur} with $\max\{\text{cost}_{\text{cur}}+1, \text{cost}_{\text{new}}\}$, where cost_{new} is the minimum among all lower bounds found in the pruned nodes of the current B&B exploration.

4. If $\text{cost}_{\text{cur}} \geq \text{UB}$, then stop the search since an optimal solution with value UB was found; otherwise, go to 2.

The iterative deepening A* algorithm is shown in Algorithm 1.

Algorithm 1 Pseudocode for the iterative deepening A* algorithm, which receives an initial stacking area state $\mathcal{S}$.

```
1: function IDA*(S)
2:     cost_cur ← Lower Bound for S
3:     UB ← Cost of an initial heuristic solution for S
4:     while cost_cur < UB do
5:         cost_new, UB_new ← B&B(S, cost_cur, 0)
6:         cost_cur ← max{cost_cur + 1, cost_new}
7:         UB ← min{UB, UB_new}
8:     return UB
```

The branch and bound algorithm uses a depth-search strategy. The algorithm is shown in Algorithm 2 and in the three following sections we explain branching, pruning, and how to generate an initial solution for it. Chapter 4 shows the lower bounds.

We use the notation $\mathrm{B}_{x,y}$ to represent a block that is in stack s_x at height y. Recall that t is the target block of the stacking area and t_x is the target block of stack s_x.

3.1.1 Branching Strategy at the B&B algorithm

Consider a node of the search that represents a state $\mathcal{S}$ after k relocations have been done along the current path of the B&B algorithm. The next nodes that are children of $\mathcal{S}$ are generated according to the following criteria:

1. If the block with the highest priority t is at the top of some stack, then the next node is generated with t being retrieved (line 7).

2. Otherwise, we have to relocate the blocking block $\mathrm{B}_{x,h}$ from the top of the target stack s_x, which has height h, to another stack $s_{x'}$. For each each stack $s_{x'}$ such that $height(s_{x'}) < H$ and $s_{x'} \neq s_x$, we create a new node where $\mathrm{B}_{x,h}$ is relocated to $s_{x'}$.

We explore the new nodes in the following order. First, we explore nodes representing states where $\mathrm{B}_{x,h}$ was relocated to a stack $s_{x'}$ such that block $\mathrm{B}_{x,h}$ did not become a blocking block at $s_{x'}$ (lines 17-22). Then we explore nodes representing states such that $\mathrm{B}_{x,h}$ was relocated to $s_{x'}$, but it became a blocking block at $s_{x'}$ (lines 23-28).

3.1.2 Pruning at the B&B algorithm

Let $\mathcal{S}$ be a state after k relocations have been done during the B&B search. We stop the search in the subspace of $\mathcal{S}$ if a lower bound for the number of relocations at state $\mathcal{S}$ plus k is greater than $\mathrm{cost}_{\mathrm{cur}}$. This node can be safely pruned since no solution with $\mathrm{cost}_{\mathrm{cur}}$ will be found from state $\mathcal{S}$ (line 14).

3.1.3 Initial solution

Tanaka and Takii [17] proposed a heuristic to generate a valid solution to the BRP, which is the one we use to generate an initial solution.

Algorithm 2 Pseudocode for the branch and bound algorithm. It receives a stacking area state $\mathcal{S}$, the target cost cost_{cur} of the solution to be found, and the number reloc of relocations already performed. It returns the minimum of all lower bounds found, and the best upper bound found.

```
1:  function B&B(S, cost_cur, reloc)
2:      cost_new ← ∞
3:      UB_new ← ∞
4:      if S is empty then
5:          UB_new ← reloc
6:          return cost_new, UB_new
7:      if t is at the top of a stack then
8:          Retrieve t from S
9:          return B&B(S, cost_cur, reloc)
10:     Let s_x be the stack of the block t
11:     Let y be the height of the block t
12:     h ← height(s_x)
13:     LB ← Lower bound for S + reloc
14:     if LB > cost_cur then
15:         cost_new ← LB
16:         return cost_new, UB_new
17:     for x' ∈ {1, ..., S} and s_x' ≠ s_x do
18:         if height(s_x') < H and B_x,h < t_x' then
19:             S' ← New state by relocating block B_x,h to stack s_x'
20:             cost_x', UB_x' ← B&B(S', cost_cur, reloc + 1)
21:             cost_new ← min{cost_new, cost_x'}
22:             UB_new ← min{UB_new, UB_x'}
23:     for x' ∈ {1, ..., S} and s_x' ≠ s_x do
24:         if height(s_x') < H and B_x,h > t_x' then
25:             S' ← New state by relocating block B_x,h to stack s_x'
26:             cost_x', UB_x' ← B&B(S', cost_cur, reloc + 1)
27:             cost_new ← min{cost_new, cost_x'}
28:             UB_new ← min{UB_new, UB_x'}
29:     return cost_new, UB_new
```

Let $\mathrm{B}_{x,y} = t$ and $\mathrm{B}_{x,h}$ be the block to be relocated, where $h = \mathit{height}(s_x)$. There are two possibilities. If $\mathrm{B}_{x,h}$ can be relocated to a stack without generating a blocking block, then we relocate it to one such stack $s_{x'}$ that has a target block $t_{x'}$ of highest priority. If $\mathrm{B}_{x,h}$ always becomes a blocking block after relocation, then let stacks s_{x_1} and s_{x_2} be stacks such that $s_{x_1} \neq s_x$, $s_{x_2} \neq s_x$, $\mathit{height}(s_{x_1}) < H$, $\mathit{height}(s_{x_2}) < H$, and whose target blocks have the lowest and second lowest retrieval priorities, respectively. If $\mathit{height}(s_{x_1}) = H - 1$ and s_{x_2} exists, then stack s_{x_2} is selected as the destination stack; otherwise, stack s_{x_1} is selected.

We create an initial solution where relocations are done following these rules until the target block can be retrieved. The process then continues to the next target block until all blocks are retrieved.

Chapter 4

Lower Bounds

In this chapter, we present lower bounds that we use in the experiments with the branch and bound algorithm. First, we present some lower bounds from the literature [11, 19, 17] in Sections 4.1 to 4.3. In Section 4.4 we show a new combinatorial lower bound that we proposed and then in Section 5.3 we present another new lower bound based on the use of pattern databases. We emphasize that the new lower bounds we present in this thesis work for the restricted variation of the BRP where blocks have unique priorities.

4.1 LB1

This lower bound was presented by Kim and Hong [11] and it was named *Lower Bound 1* (LB1) throughout the literature. The main observation used in this lower bound comes from the concept of blocking blocks. Given a current state of the stacking area, LB1 is defined as the number of blocking blocks in all the stacks of the stacking area. Remember that b is a blocking block if there is another block $a < b$ in the same stack of b and positioned below it. This is a valid lower bound because at least such blocks must be relocated to clear the stacking area.

4.2 LB2 and LB3

Lower Bound 2 (LB2) was proposed by Zhu *et al.* [19]. For a given state of the stacking area, LB2 is defined as the number of blocking blocks in the stacking area (LB1) plus the number of blocking blocks above the target block t that even after relocation are still blocking blocks.

More formally, suppose that the target block t is $\mathrm{B}_{x,y}$ (it is in stack s_x at height y). This means that we have to relocate the blocks $\mathrm{B}_{x,y+1}, \mathrm{B}_{x,y+2}, \ldots, \mathrm{B}_{x,h}$, where $h = height(s_x)$. We thus check if each one of these blocks keeps the status of being a blocking block even after we relocate them, regardless of which stack it is relocated to. A block $\mathrm{B}_{x,y'}$, for $y' \in [y+1, h]$, will be a blocking block after relocation if $\mathrm{B}_{x,y'} > t_{x'}$ for all $x' \in \{1, 2, \ldots, S\}$ such that $s_{x'} \neq s_x$.

Lower Bound 3 (LB3) was also proposed by Zhu *et al.* [19] as an improvement over LB2. It is computed in an iterative process as follows. First, we compute the number

of blocking blocks above t in the target stack and add to this value the amount of these blocking blocks that remain being a blocking block even after relocating them, in a process similar to LB2. Then, we remove these blocks and the target block, so that a new target block is identified in this new configuration. Then the calculation is repeated for the blocks above the new target block. This process is repeated until all blocks are removed. The value of LB3 is obtained by adding all these values.

In the computational experiments, the use of LB3 in the exact algorithm resulted in more optimal solutions being found than when using LB1 and LB2. The algorithm to compute LB3 has time complexity $O(SN)$, where S is the number of stacks and N is the number of blocks in the instance.

4.3 LB4

Using the same previous notation, suppose that the target block t is $\mathrm{B}_{x,y}$. We have to relocate blocks $\mathrm{B}_{x,y+1}, \mathrm{B}_{x,y+2}, \ldots, \mathrm{B}_{x,h}$, where $h = \mathit{height}(s_x)$. After relocating a block $\mathrm{B}_{x,y'}$ where $y' \in [y+1, h]$ to some stack $s_{x'}$, there are two possibilities: (1) $\mathrm{B}_{x,y'}$ becomes a blocking block in $s_{x'}$ or (2) it does not become a blocking block in $s_{x'}$, which means that $\mathrm{B}_{x,y'}$ becomes the new target block of stack $s_{x'}$. In this last case, it may happen that one of the other blocks to be relocated, $\mathrm{B}_{x,y+1}, \mathrm{B}_{x,y+2}, \ldots, \mathrm{B}_{x,y'-1}$, becomes a blocking block if relocated to $s_{x'}$, blocking the new target block of this stack, which is $\mathrm{B}_{x,y'}$. Tanaka and Takki [17] used this observation to create a new lower bound, which they denote by LB4.

We give a brief overview of how LB4 is computed. First, the height limit H is relaxed for the stacks $s_{x'}$ where $\mathrm{height}(s_{x'}) < H$. These stacks are the ones for which blocks above t can be relocated to. We denote these stacks by $s_{u_1}, s_{u_2}, \ldots, s_{u_k}$ where $\mathrm{height}(s_{u_i}) < H$ and $s_x \neq s_{u_i}$ for $1 \leq i \leq k$. Now the algorithm performs an enumeration process relocating each blocking block above t in the following order, first $\mathrm{B}_{x,h}$, then $\mathrm{B}_{x,h-1}$ and so on. We want to find the number of these blocks that remain being a blocking block even after relocating them. In the enumeration process, the current block is relocated either to a stack where it becomes a blocking block, or to a stack where it becomes a new target block, creating two possible branches in the enumeration tree. In the first case, the current block can be relocated to any stack where it becomes a blocking block, since the target block of that stack does not change. In the second case Tanaka and Takki [17] show that we can only consider the case where the current block is relocated to the stack with minimum target block value.

Let b be the number of blocking blocks above the target stack $t = \mathrm{B}_{x,y}$, i.e., $b = h - y$. After enumerating all $O(2^b)$ possibilities of relocations of the blocking blocks above t, we consider the relocation of blocks $\mathrm{B}_{x,y+1}, \mathrm{B}_{x,y+2}, \ldots, \mathrm{B}_{x,h}$ that resulted in the least number of blocking blocks. We add to LB4 this value and b, and as in LB3, remove the current target block and all blocks above it, and repeat the process to the next target block. We repeat this process until all blocks are removed.

The algorithm to compute LB4 has time complexity $O(S \log S + 2^b \log S)$, where S is the number of stacks and b is the number of blocks above the current target block.

4.4 LB-LIS

In this section, we present a new lower bound, which we call LB-LIS. Given a state of the problem (stacking area), let $B_{x,y}$ be the target block t. In this iteration we have to relocate the blocks $B_{x,y+1}, B_{x,y+2}, \ldots, B_{x,h}$, where $h = height(s_x)$. Let $t_1, \ldots, t_{x-1}, t_{x+1}, \ldots, t_S$ be the target blocks of each remaining stack.

We create an auxiliary binary matrix M, where $B_{x,y+1}, B_{x,y+2}, \ldots, B_{x,h}$ are the labels of the rows (first column of color yellow in Figure 4.1) and $t_1, \ldots, t_{x-1}, t_{x+1}, \ldots, t_S$ are the labels of the columns (bottom row of color red in Figure 4.1). We fill this matrix in the following manner: for each $y' \in [y+1, h]$, and $x' \in [1, S] \setminus \{x\}$, if $B_{x,y'}$ is greater than $t_{x'}$, or if $height(s_{x'}) = H$, then $M[B_{x,y'}, t_{x'}] = 1$; otherwise $M[B_{x,y'}, t_{x'}] = 0$. The idea of such matrix is to show in which stack other than s_x, $B_{x,y'}$ is also a blocking block if relocated to it. In Figure 4.1, columns labels go from left to right and rows labels go from bottom to top.

(5)	2	0	0	0	0	0
(4)	7	0	0	0	1	1
(4)	3	0	0	0	0	0
(3)	10	0	1	1	1	1
(2)	11	0	1	1	1	1
(1)	12	0	1	1	1	1
		13	9	8	6	5

Figure 4.1: Example of auxiliary matrix created to calculate LB-LIS. In this example, assume $t = 1$ and it is block $B_{4,1}$ and that $B_{4,2} = 12$, $B_{4,3} = 11$, $B_{4,4} = 10$, $B_{4,5} = 3$, $B_{4,6} = 7$, $B_{4,7} = 2$ (yellow column). Also assume that $t_1 = 13$, $t_2 = 9$, $t_3 = 8$, $t_5 = 6$, and $t_6 = 5$ (red row). The numbers inside the circles are the size of the longest increasing subsequence for each $B_{4,y'}$, for $y' \geq 2$.

For each $B_{x,y'}$ where $y < y' \leq h$, we create a sequence $\text{Seq}(B_{x,y'}) = (B_{x,y'}, B_{x,y'-1}, \ldots, B_{x,y+1})$, where y is the height of the target block. Then, we calculate the size of the Longest Increasing Subsequence (LIS), not necessarily consecutive, taking $\text{Seq}(B_{x,y'})$ as input.

In Figure 4.1, if $B_{x,y'} = 2$, then $\text{Seq}(B_{x,y'}) = (2, 7, 3, 10, 11, 12)$ and thus $\text{LIS}(\text{Seq}(B_{x,y'})) = 5$. If $B_{x,y'} = 7$, then $\text{Seq}(B_{x,y'}) = (7, 3, 10, 11, 12)$ and $\text{LIS}(\text{Seq}(B_{x,y'})) = 4$. If $B_{x,y'} = 3$, then $\text{LIS}(\text{Seq}(B_{x,y'})) = 4$, and so on. The numbers inside the circles in Figure 4.1 show the size of the LIS for each $B_{x,y'}$.

Notice that, for each $B_{x,y'}$, we know in how many stacks it does not become a blocking block after relocating it: it is the number of zeros in its corresponding row, denoted by $\text{zeros}(y')$. If we consider only the blocks that belong to the LIS of $\text{Seq}(B_{x,y'})$, no matter how the relocations of those blocks are performed, it will generate at least $\text{reloc}(x, y') = \max\{0, \text{LIS}(\text{Seq}(B_{x,y'})) - \text{zeros}(y')\}$ blocking blocks. Let $\text{reloc}(t)_{max}$ be the maximum

value among all values of reloc(x, y') for $y' \in [y+1, h]$. A valid lower bound is thus created taking this maximum value plus the number of blocking blocks above t in the target stack. We then remove t and all blocking blocks that are above t from the stacking area and repeat this process considering the new target block (similarly to what is done in LB3). The value of LB-LIS is obtained summing all these partial values.

The algorithm to compute LB-LIS has time complexity $O(Sb + S^2)$, where S is the number of stacks and b is the number of blocks above the current target block.

Consider another example presented in Figure 4.2a. The target block is $\mathrm{B}_{4,2}$ (blue block), and the two blocks above it, $\mathrm{B}_{4,3}$ and $\mathrm{B}_{4,4}$, are blocking blocks. So, we create matrix M (to the left of Figure 4.2a), with $\mathrm{B}_{4,3}$ and $\mathrm{B}_{4,4}$ as rows (in yellow), and in the bottom of the columns (in red) we have the target blocks of each one of the other stacks. We fill the matrix according to the rule explained before. In this case, reloc$(4,3) = 1 - 1 = 0$ and reloc$(4,4) = 2 - 1 = 1$. So, for now LB-LIS has value $1 + 2$, which is the maximum reloc value plus the number of blocking blocks above the target block.

In the next iteration, after removing the target block and blocking blocks above it, we have the state presented in Figure 4.2b. All blocks $\mathrm{B}_{3,2}, \ldots, \mathrm{B}_{3,6}$ are blocking blocks. Using these blocks as rows and the remaining target blocks as columns, we construct matrix M (to the left in Figure 4.2b). We have reloc$(3,6) = 3 - 2 = 1$, reloc$(3,5) = 2 - 1 = 1$, reloc$(3,4) = 2 - 2 = 0$, reloc$(3,3) = 1 - 0 = 1$, and reloc$(3,2) = 1 - 1 = 0$. So, LB-LIS is incremented by $1 + 5$, being now equal to 9. Proceeding in this manner, the final value of LB-LIS is 22, while LB1 (grey blocks) is 17, LB3 is 18, and LB4 is also 22.

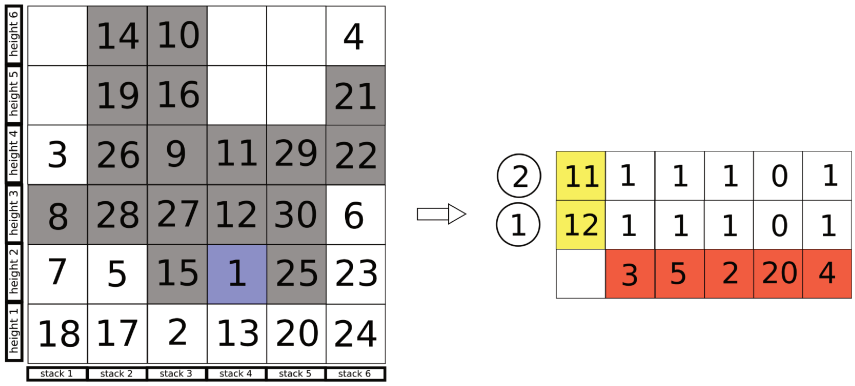
Now we prove the correctness of this lower bound.

Theorem 4.4.1 *LB-LIS is a valid lower bound for the BRP.*

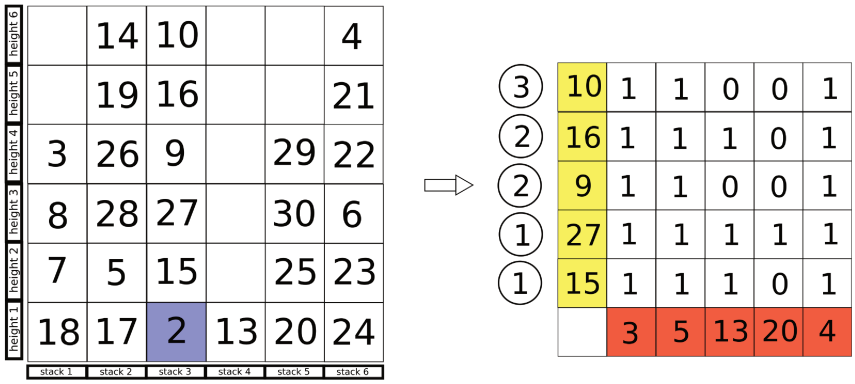
Proof. Let I be an instance of the BRP problem, and for each block b_i of this instance, let $r(b_i)$ be the number of relocations b_i requires in an optimal solution OPT. The cost of OPT is thus $\sum_{i=1}^{N} r(b_i)$. To show that LB-LIS is a valid lower bound, we are going to show that we can split the value of LB-LIS among all blocks, namely we will find lis(b_i) for each $i = 1, \ldots, N$, such that $\sum_{i=1}^{N} \mathrm{lis}(b_i) \leq \sum_{i=1}^{N} r(b_i)$.

The value of LB-LIS is the sum of *partial values* of the lower bound computed in a series of iterations, where on each iteration we have a stacking area state $\mathcal{S}$, and its corresponding target block $t = \mathrm{B}_{x,y}$. Let h be the height of the stack containing t. We compute the number of relocations for blocks above t as described previously, i.e., the maximum value of reloc(x, y') for $y < y' \leq h$ plus the number of blocking blocks above t. After that, we remove t and all blocking blocks above it and proceed to the next iteration with a new state and target block. Let R be the partial value computed in one iteration of the computation of LB-LIS, and let $\mathrm{B}_{x,y+1}, \ldots, \mathrm{B}_{x,h}$, be the blocking blocks of this iteration. For each block $\mathrm{B}_{x,y'}$, for $y < y' \leq h$, we assign lis$(\mathrm{B}_{x,y'}) = R/(h - y)$, i.e., we split the value of R equally among all blocks involved in this iteration. First notice that each block b_i of the instance is assigned a value lis(b_i) only once, since after being considered in some iteration it is removed from the instance. So we only need to show that $R \leq \sum_{y'=y+1}^{h} r(\mathrm{B}_{x,y'})$.

In any iteration, the target block and its corresponding blocking blocks are in the same relative configuration as in the initial state. The only difference is that some other blocks



(a) Instance of the BRP and how to calculate the LB-LIS for the target block 1 (blue color).



(b) Same instance after retrieving block 1 and removing all blocks above it. Now we can calculate LB-LIS for the target block 2 (blue color).

Figure 4.2: Example of how to compute LB-LIS.

may have been removed in previous iterations. Let $r'(b_i)$ be the number of relocations suffered by a block b_i in an optimal solution considering the state of the current iteration as the initial state. One can easily see that

$$\sum_{y'=y+1}^{h} r'(\mathrm{B}_{x,y'}) \leq \sum_{y'=y+1}^{h} r(\mathrm{B}_{x,y'}).$$

Since to retrieve the current target block t, $\mathrm{reloc}(t)_{max}$ accounts for the number of blocking blocks that will still be blocking blocks after relocation in the current stacking area state, we have that

$$R = \mathrm{reloc}(t)_{max} + \#(\text{blocking blocks above } t) \leq \sum_{y'=y+1}^{h} r'(\mathrm{B}_{x,y'})$$

and the result follows. □

Chapter 5

Pattern Databases

A Pattern DataBase (PDB) [6] basically consists of precomputed optimal solutions for small instances of the problem that are stored in a hash table. This way, when exploring some node during the search for an optimal solution, if this node corresponds to a solution stored in the PDB, we can stop the exploration and obtain its optimal solution from the PDB. In order to increase the usefulness of the PDB, generally one represents solutions in an abstract manner, such that several nodes are represented by a same solution in the PDB. We notice that the use of PDB and abstract states as a caching strategy of precomputed solutions was used before for the BRP problem by Ku and Arthanari [14].

Our contributions in this chapter are showing how to use PDBs to create a new lower bound for the BRP, the presentation of an algorithm to generate the PDB together with a prove of its correctness, and some ideas on how to save memory space using the PDB.

In the following sections, we describe some definitions necessary to understand how to generate and use the PDB for the BRP.

5.1 Abstract state of the BRP

We start by describing how to represent a state of a stacking area in an abstract manner, such that several stacking areas correspond to a same abstract state. The abstract state we use is essentially the same one used by Ku and Arthanari [14].

We present a function abState(I, B) that takes as input an instance I of the BRP with N blocks and a subset B of the blocks ($B \subseteq \{b_1, b_2, \ldots, b_N\}$). The function generates a new instance I' containing only the blocks in B and preserving their relative positions. The blocks in $N \setminus B$ are removed and the blocks in B are relabeled from 1 to $|B|$, keeping the priority order. Therefore, this function creates a new smaller instance that is based on the original instance I. See Figure 5.1 for an example.

Another important observation is that in the BRP all relocations cost 1 no matter what is the distance between stacks. Therefore, the order of the stacks are not relevant, and many states of the stacking area are equivalent if we always sort the stacks by the following general criterion: first place the empty stacks to the right and then rearrange the remaining stacks in ascending order by the value of the target block in each stack. An *abstract state* for an instance I of the BRP, denoted by abs(I), consists of the same

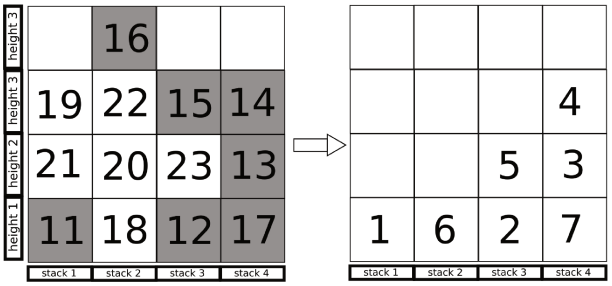


Figure 5.1: Let I (left) be an instance of the BRP, and consider the set $B = \{11, 12, 13, 14, 15, 16, 17\}$. The instance generated by $\text{abState}(I, B)$ is shown to the right where blocks $11, 12, \ldots, 17$ are re-labeled respectively to $1, 2, \ldots, 7$.

instance I but with the stacking area organized in the order described above.

We assume that $\text{abState}(I, B)$ always generate an abstract state of the resulting instance, i.e., it is equivalent to $\text{abs}(\text{abState}(I, B))$. So in the example of Figure 5.1, the stack 2, containing block 6, would become the rightmost stack. The pattern database for the BRP consists of optimal solutions for all abstract states representing instances with at most $|B|$ blocks.

5.2 Building the Pattern Database

In general, a PDB is built by running a breadth-first search backwards from the goal state (the empty stacking area) until all states in the abstract state space are reached. In our case, we want to generate all optimal solutions for instances with up to N' blocks, for given values S (number of stacks) and H (maximum stack height) for the stacking area.

We use the following steps to build the PDB for the BRP. Let I be an instance where the stacking area is empty and the artificial stack (s_0) has N' ordered blocks on it (block i above $i-1$, for $i \in \{2, 3, \cdots, N'\}$). The main steps of the algorithm that creates the PDB are:

1. Insert $(I, 0)$ into a queue.
2. Remove instance (I', k) from the queue.
3. If $\text{abs}(I')$ was not visited:
   - Generate at most S new instances I'_x where the top element from the artificial stack s_0 is relocated to each stack s_x inside the stacking area such that $height(s_x) < H$. Insert the abstract state of these new instances into the queue with cost k, i.e., insert $(\text{abs}(I'_x), k)$ into the queue.
   - Generate $S - 1$ new instances I'_x where we relocate a top block of some stack s_x (except the stack of the target block) to the stack of the target block of I'. Insert the abstract state of these new instances into the queue but with cost $k + 1$, i.e., insert $(\text{abs}(I'_x), k + 1)$ into the queue.
4. Mark $\text{abs}(I')$ as visited and set its cost as k.

5. If the queue is empty, then finish the algorithm; otherwise, go to step 2.

A detailed pseudocode of the algorithm used to create the PDB is presented in Algorithm 3.

Algorithm 3 Pseudocode of a Breath First Search algorithm used to generate the PDB consisting of all abstract instances with N blocks and S stacks with maximum height H.

```
1:  function BFS-PDB(N', S, H)
2:      PDB ← ∅
3:      Let Q be an empty queue
4:      Let I be the state representing an instance with empty stacks s_0^I, ..., s_S^I
5:      for i ← 1 to N' do
6:          Push block b_i into s_0^I
7:      Insert (I, 0) into Q
8:      while Q is not empty do
9:          (I', k) ← dequeue(Q)
10:         if I' is not in PDB then
11:             if s_0^{I'} is not empty at I' then
12:                 for x ∈ {1, ..., S} such that height(s_x^{I'}) < H do
13:                     Relocate the top block from s_0^{I'} to s_x^{I'}
14:                     Insert a copy of (I', k) into Q
15:                     Relocate the top block from s_x^{I'} to s_0^{I'}
16:             Let s_{x*}^{I'} be the target stack of the stacking area I'
17:             for x ∈ {1, ..., S} such that s_x^{I'} ≠ s_{x*}^{I'} do
18:                 if height(s_x^{I'}) > 0 and height(s_{x*}^{I'}) < H then
19:                     Relocate the top block from s_x^{I'} to s_{x*}^{I'}
20:                     Insert copy of (I', k + 1) into Q
21:                     Relocate the top block from s_{x*}^{I'} to s_x^{I'}
22:             Insert I' into the PDB with value k
23:     return PDB
```

Now we prove the correctness of the algorithm. First notice that the algorithm generates all possible states of N' blocks in a stacking area of sizes S and H.

Theorem 5.2.1 *For any given stacking area state I' with N' blocks and stacking area sizes S and H,* $\text{abs}(I')$ *is generated by the algorithm.*

Proof. Let $(m_1, m_2, \ldots, m_f)$ be an optimal sequence of movements (relocations or retrievals) that clears the state I'. Consider an initial state where all blocks are stacked in the artificial stack s_0 in increasing order from bottom to top. The inverse sequence of these movements $(m_f^{-1}, m_{f-1}^{-1}, \ldots, m_1^{-1})$ applied to this initial state consists only of insertions from s_0 to a stack s_x or relocations of a block from one stack to the current target stack. For simplicity, rename the movements $(m_f^{-1}, m_{f-1}^{-1}, \ldots, m_1^{-1})$ to $(m'_1, m'_2, \ldots, m'_f)$, i.e., $m_i^{-1} = m'_{f-i+1}$ for $i = f, \ldots, 1$. We can prove by induction on f that the algorithm will generate the state $\text{abs}(I'_f)$ representing the application of movements $(m'_1, m'_2, \ldots, m'_f)$ applied to an initial state with blocks stacked in ascending order at the artificial stack

s_0. For $f = 1$, since the algorithm generates all states with the top element in s_0, block $b_{N'}$, relocated to each stack s_x, one of these states corresponds to abs(I'_1). Now assume that the algorithm generated a state corresponding to abs(I'_{f-1}), where movements $(m'_1, m'_2, \ldots, m'_{f-1})$ were applied. This state is saved in the queue and when it is evaluated, the algorithm will perform all types of movements possible, either relocating a stacked block to the top of the target stack, or relocating a block from s_0 to each possible stack s_x. One of these movements corresponds to m'_f, so abs(I') is generated. □

Since the algorithm generates all possible stacking area states, we just need to prove that when it marks a state I' as visited, then the cost k is the value of an optimal solution for this instance.

Theorem 5.2.2 *Let I' be a state with N' blocks and stacking area sizes S and H. If the algorithm marks* abs(I') *as visited with cost k, then k is the optimal number of relocations to clear state I'.*

Proof. Notice that the algorithm inserts the blocks in the stacking area in inverse order, from block $b_{N'}$ to b_1. Let ℓ be the number of blocks inserted so far in the stacking area in a given iteration of the algorithm. We prove by induction on ℓ that all states containing the first ℓ blocks in the stacking area, i.e., blocks $b_{N'}$, $b_{N'-1}$, ..., $b_{N'-\ell+1}$, are visited in ascending order by their cost, so that the first time the algorithm visits a state I' it has the smallest possible cost.

For the base case consider $\ell = 1$. The algorithm generates states containing just $b_{N'}$ with cost 0, which is the minimum possible.

Now consider a state I^* with $\ell = N'$ blocks in the stacking area. For this state I^*, consider an optimal sequence of movements $(m_1, m_2, \ldots, m_f)$ that clears I^*, and let $(m'_1, m'_2, \ldots, m'_f)$ be the inverse of this optimal sequence. Let $(m'_1, m'_2, \ldots, m'_j)$ be the movements of this inverse sequence with movements just before block b_1 is inserted in the stacking area, i.e., m'_{j+1} is a relocation of b_1 from stack s_0 to some stack s_x, and movements $m'_{j+2}, \ldots, m'_f$ are relocations of blocks from the stacking area to the target stack s_x.

Assume for the purpose of induction that the state I^1, corresponding to the application of $(m'_1, m'_2, \ldots, m'_j)$, has optimal cost and is inserted in the queue before other states containing blocks $b_{N'}, \ldots, b_2$ and higher cost. The state I^2 corresponding to the application of $(m'_1, m'_2, \ldots, m'_j, m'_{j+1})$ has the same cost of I^1, since it derives from I^1 and a relocation from s_0 to s_x does not change the cost of this new state. When state I^2 is evaluated by the algorithm it generates all new states corresponding to relocations of blocks from other stacks to the target stack s_x, and one of these corresponds to $(m'_1, m'_2, \ldots, m'_{j+2})$ which is inserted in the queue with the cost increased by one. It is not hard to see that the algorithm will generate state I^* after I^2 applying relocations $m'_{j+2}, \ldots, m'_f$ and this state will have optimal cost, since I^2 has optimal cost and the inverse of $m'_{j+2}, \ldots, m'_f$ corresponds to relocations of the optimal solution to clear I^*. So every state I^* with N' blocks is visited in increasing order of cost, so that the first time I^* is visited it has optimal cost. □

Depending on the values of N', S, and H, the previous algorithm generates a database with very large size. To reduce the database size we performed some improvements over it. Our first enhancement was to take advantage of the abstract states of the BRP domain, that is, we sorted the stacks in order to keep only one representative for each set of similar stacking areas. Our second enhancement was to compactly represent a state with only two numbers of 64 bits. The first number represents the heights of the N' blocks $b_1, b_2, \ldots, b'_N$ in the stacking area: the ith digit (from left to right) of this number represents the height of the block b_i minus 1. For example, if the blocks are $b_1, b_2, \ldots, b_{10}$ and their heights are $1, 2, 3, 1, 1, 1, 2, 3, 4, 2$ respectively, then the first number is 0120001231 (we discard leading zeros, therefore this number is only 120001231). The second number represents the stack in which each block is stacked and it is kept in a similar manner as the number representing heights. Our third enhancement was to delete from the database all states for which the total number of relocations is 0. Our fourth and last enhancement was to delete instances of the PDB in the following way. For two instances, A and B, if after consecutive retrievals on A we arrive at B, then we delete A from the PDB because the two instances are equivalent in cost. So before consulting an instance in the PDB, we first retrieve all possible blocks from it that can be retrieved without relocations. In Table 5.1 we present the sizes of the generated PDB with $N' = 10$ blocks, for given values of S and H, after each enhancement.

5.3 Using the PDB to compute a lower bound

In this section we show how to use the PDB to find a new lower bound to the BRP. We denote this new lower bound by LB-PDB. Let I be an instance with N blocks and S stacks with height limit H. Let $g_1, g_2, \ldots, g_{\lceil N/N' \rceil}$ be a partition of the N blocks in groups of N' blocks in order, where for each $j = 1, \ldots, \lceil N/N' \rceil$, $g_j = \{b_{(j-1)N'+1}, \ldots, b_{jN'}\}$. Let G_j be the union of the first j parts, i.e., $G_j = g_1 \cup \ldots \cup g_j$ and let $I_j = \text{abState}(I, G_j)$.

We define $\text{OPT}(I)$ as the minimum number of relocations needed to retrieve all blocks of any given instance I and $\text{REL}(I)$ as an optimal sequence of relocations and retrievals for I. We denote by PDB a function that consults the pattern database and returns the number of relocations to solve some instance I, if $\text{abs}(I)$ is present in the database. Note that $\text{PDB}(\text{abState}(I, g_j)) = \text{OPT}(\text{abState}(I, g_j))$.

Recall LB2, the lower bound proposed by Zhu *et al.* [19] and explained in Chapter 4. We have $\text{LB2}(b) = 0$ if b is not a blocking block, $\text{LB2}(b) = 1$ if b is a blocking block but it is not a blocking block after relocation, and $\text{LB2}(b) = 2$ otherwise. We define

$$\text{LB2}(g_j) = \sum_{b \in g_j} \text{LB2}(b).$$

At last for each $j = 1, \ldots, \lceil N/N' \rceil$ define

$$\text{LB-PDB}(I_j) = \sum_{i=1}^{j} \max\{\text{LB2}(g_i), \text{PDB}(\text{abState}(I, g_i))\},$$

Table 5.1: Enhancement of the PDB for instances with 10 blocks.

S	H	First	Second	Third	Fourth
6	3	182 MB	92 MB	77 MB	41 MB
6	4	698 MB	353 MB	296 MB	177 MB
6	5	1400 MB	683 MB	573 MB	365 MB
6	6	1900 MB	954 MB	800 MB	526 MB
6	7	2300 MB	1200 MB	961 MB	645 MB
7	3	184 MB	93 MB	77 MB	42 MB
7	4	701 MB	355 MB	296 MB	177 MB
7	5	1400 MB	685 MB	573 MB	365 MB
7	6	1900 MB	955 MB	800 MB	527 MB
7	7	2300 MB	1200 MB	961 MB	646 MB
8	3	185 MB	93 MB	77 MB	42 MB
8	4	701 MB	355 MB	296 MB	177 MB
8	5	1400 MB	685 MB	573 MB	365 MB
8	6	1900 MB	955 MB	800 MB	527 MB
8	7	2300 MB	1200 MB	961 MB	646 MB
9	3	185 MB	93 MB	77 MB	42 MB
9	4	701 MB	355 MB	296 MB	177 MB
9	5	1400 MB	685 MB	573 MB	365 MB
9	6	1900 MB	955 MB	800 MB	527 MB
9	7	2300 MB	1200 MB	961 MB	646 MB
10	3	185 MB	93 MB	77 MB	42 MB
10	4	701 MB	355 MB	296 MB	177 MB
10	5	1400 MB	685 MB	573 MB	365 MB
10	6	1900 MB	955 MB	800 MB	527 MB
10	7	2300 MB	1200 MB	961 MB	646 MB

and

$$\text{LB-PDB}(I) = \text{LB-PDB}(I_{\lceil N/N' \rceil}).$$

We show that this is a valid lower bound for $\text{OPT}(I)$ in Theorem 5.3.1. First, we need some intermediary results. We can write $\text{OPT}(I_j)$, the optimal solution value to retrieve blocks of the first j parts $G_j = g_1 \cup \ldots \cup g_j$, as

$$\text{OPT}(I_j) = |R_0(G_{j-1})| + |R_1(g_j)| + |R_2(g_j)|,$$

where $R_0(G_{j-1}) \subseteq \text{REL}(I_j)$ is the sequence of relocations that are performed over blocks in G_{j-1} to retrieve all blocks in G_{j-1}, $R_1(g_j) \subseteq \text{REL}(I_j)$ is the sequence of relocations that are performed over blocks in g_j when retrieving all blocks in G_{j-1}, and $R_2(g_j) \subseteq \text{REL}(I_j)$ is the sequence of relocations that are performed over blocks in g_j to retrieve these blocks of g_j, after the blocks in G_{j-1} were retrieved.

The following three claims are direct results from the definitions above.

Claim 1 *The sequence of relocations $R_0(G_{j-1})$ is a solution for I_{j-1}, so*

$$\text{OPT}(I_{j-1}) \leq |R_0(G_{j-1})|.$$

Claim 2 *The sequence of relocations $R_1(g_j)$ and $R_2(g_j)$ together are a solution for* $\text{abState}(I, g_j)$*, so*

$$\text{PDB}(\text{abState}(I, g_j)) \leq |R_1(g_j)| + |R_2(g_j)|.$$

Claim 3 *The sequence of relocations $R_1(g_j)$ and $R_2(g_j)$ are applied to the whole instance G_j to remove items in G_{j-1} and g_j, so*

$$\text{LB2}(g_j) \leq |R_1(g_j)| + |R_2(g_j)|.$$

Theorem 5.3.1 *For any j such that $1 \leq j \leq \lceil N/N' \rceil$ we have* $\text{LB-PDB}(I_j) \leq \text{OPT}(I_j)$.

Proof. The proof is by induction on j. In the base case consider $j = 1$. Since $G_1 = g_1$, and this set contains the smallest blocks, they can only block each other, so by definition we must have $\text{LB2}(g_1) \leq \text{OPT}(I_1)$. Since the PDB saves optimal solutions to abstract states we have $\text{PDB}(\text{abState}(I, g_1)) = \text{OPT}(I_1)$. So we conclude that

$$\text{LB-PDB}(I_1) = \max\{\text{LB2}(g_1), \text{PDB}(\text{abState}(I, g_1))\} \leq \text{OPT}(I_1).$$

Now consider $j \geq 2$. We have

$$\begin{aligned}
\text{LB-PDB}(I_j) &= \text{LB-PDB}(I_{j-1}) + \max\{\text{LB2}(g_j), \text{PDB}(\text{abState}(I, g_j))\} \\
&\leq \text{OPT}(I_{j-1}) + \max\{\text{LB2}(g_j), \text{PDB}(\text{abState}(I, g_j))\} \\
&\leq \text{OPT}(I_{j-1}) + |R_1(g_j)| + |R_2(g_j)| \\
&\leq |R_0(G_{j-1})| + |R_1(g_j)| + |R_2(g_j)| \\
&= \text{OPT}(I_j) \ ,
\end{aligned}$$

where the first inequality follows from the inductive hypothesis, the second one from Claims 2 and 3, and the third inequality follows from Claim 1. □

5.4 Improving the Exact Algorithm

In this section we show how to use the PDB and abstractions states to speed up the B&B algorithm. We avoid the exploration of state nodes whose solutions are precomputed in the PDB and we use a hash table to save solutions of abstract states corresponding to nodes that were completely explored during the B&B search. These ideas were used before by Ku and Arthanari [14].

The observation that stacks can be rearranged and represent a same state suggests that in the B&B algorithm we may visit equivalent states many times. Therefore, we can avoid the exploration of parts of the search space if we keep track of equivalent states. Algorithm 4 shows the pseudocode for the improvement that is described next. Every time we completely explore a subtree of the search space, with a root representing a state with k' relocations done previously, we can save the abstract state of this root node and the optimal solution for it (lines 31 to 33). During the exploration of another part of the search space, if we explore another node representing the same abstract state, we can stop the exploration of this subtree (lines 7 and 8), since we already know its optimal value. In this approach, we save all abstract states where at most k' relocations were done previously in a hash table.

Another improvement is to use the PDB, where we precompute optimal solutions with up to N' blocks. When exploring a new node in the B&B, where it represents a state with N' blocks, we check if the abstract state of this node was already saved in the PDB (lines 4 to 6) and use its precomputed optimal solution.

Algorithm 4 Pseudocode for the branch and bound algorithm using the information of the PDB and abstraction states. It receives a stacking area state $\mathcal{S}$, the target cost cost_{cur} of the solution to be found, the number reloc of relocations already performed, and constants k' and N'. It returns the minimum of all lower bounds found, and the best upper bound found.

1: **function** B&B($\mathcal{S}$, cost_{cur}, reloc, k', N')
2: $\quad\text{cost}_{\text{new}} \leftarrow \infty$
3: $\quad\text{UB}_{\text{new}} \leftarrow \infty$
4: $\quad$**if** $\mathcal{S}$ has N' blocks **then**
5: $\quad\quad\text{UB}_{\text{new}} \leftarrow \text{reloc} + \text{PDB}(abs(\mathcal{S}))$
6: $\quad\quad$**return** $\text{cost}_{\text{new}}, \text{UB}_{\text{new}}$
7: $\quad$**if** reloc $\leq k'$ and $abs(\mathcal{S}) \in$ hashTable **then**
8: $\quad\quad$**return** hashTable($abs(\mathcal{S})$)
9: $\quad$**if** t is at the top of a stack **then**
10: $\quad\quad$Retrieve t from $\mathcal{S}$
11: $\quad\quad$**return** $\text{B\&B}(S, \text{cost}_{\text{cur}}, \text{reloc}, k', N')$
12: $\quad$Let s_x be the stack of block t
13: $\quad$Let y be the height of block t
14: $\quad h \leftarrow height(s_x)$
15: $\quad\text{LB} \leftarrow$ Lower bound for $\mathcal{S} + \text{reloc}$
16: $\quad$**if** $\text{LB} > \text{cost}_{\text{cur}}$ **then**
17: $\quad\quad\text{cost}_{\text{new}} \leftarrow \text{LB}$
18: $\quad\quad$**return** $\text{cost}_{\text{new}}, \text{UB}_{\text{new}}$
19: $\quad$**for** $x' \in \{1, \ldots, S\}$ and $s_{x'} \neq s_x$ **do**
20: $\quad\quad$**if** $height(s_{x'}) < H$ and $\text{B}_{x,h} < t_{x'}$ **then**
21: $\quad\quad\quad\mathcal{S}' \leftarrow$ New state by relocating block $\text{B}_{x,h}$ to stack $s_{x'}$
22: $\quad\quad\quad\text{cost}_{x'}, \text{UB}_{x'} \leftarrow \text{B\&B}(\mathcal{S}', \text{cost}_{\text{cur}}, \text{reloc} + 1, k', N')$
23: $\quad\quad\quad\text{cost}_{\text{new}} \leftarrow \min\{\text{cost}_{\text{new}}, \text{cost}_{x'}\}$
24: $\quad\quad\quad\text{UB}_{\text{new}} \leftarrow \min\{\text{UB}_{\text{new}}, \text{UB}_{x'}\}$
25: $\quad$**for** $x' \in \{1, \ldots, S\}$ and $s_{x'} \neq s_x$ **do**
26: $\quad\quad$**if** $height(s_{x'}) < H$ and $\text{B}_{x,h} > t_{x'}$ **then**
27: $\quad\quad\quad\mathcal{S}' \leftarrow$ New state by relocating block $\text{B}_{x,h}$ to stack $s_{x'}$
28: $\quad\quad\quad\text{cost}_{x'}, \text{UB}_{x'} \leftarrow \text{B\&B}(\mathcal{S}', \text{cost}_{\text{cur}}, \text{reloc} + 1, k', N')$
29: $\quad\quad\quad\text{cost}_{\text{new}} \leftarrow \min\{\text{cost}_{\text{new}}, \text{cost}_{x'}\}$
30: $\quad\quad\quad\text{UB}_{\text{new}} \leftarrow \min\{\text{UB}_{\text{new}}, \text{UB}_{x'}\}$
31: $\quad$**if** reloc $\leq k'$ and $abs(\mathcal{S}) \notin$ hashTable **then**
32: $\quad\quad\text{hashTable}(abs(\mathcal{S})) = \text{cost}_{\text{new}}, \text{UB}_{\text{new}}$
33: $\quad$**return** $\text{cost}_{\text{new}}, \text{UB}_{\text{new}}$

Chapter 6

Computational Results

In this chapter we present the experimental analysis of the algorithms to the BRP. First we show the set of instances used in the experiments (Section 6.1), then we present some results regarding the construction of the PDB (Section 6.2), and then in Section 6.3 we evaluate the algorithms using the different lower bounds. In Section 6.4 we present the results of the improved exact algorithm using the PDB to stop the exploration of the search space earlier, as well as the use of a hash table to save solutions already computed of abstract states until a pre-defined depth of the search tree.

6.1 Instance Set

For the computational experiments, we downloaded and used the dataset of instances of Zhu *et al.* [19]. These are the instances used in other experiments such as the ones performed by Tanaka and Takki [17]. According to Zhu *et al.* [19], the dataset is generated in the following manner. For fixed values of N (number of blocks), S (number of stacks), and H (height limit), it is generated a random permutation of the first N integers. The blocks are inserted in stacks in the order given by this permutation. For each block in the permutation, it is selected a random stack and the block is assigned to it if the stack has fewer than H blocks. If the stack already has H blocks, then the instance is discarded and the process is restarted. An instance is generated after each element of the permutation is assigned to a stack.

The instances were generated with the following values: for $S = 6$ and each $H \in \{3, 4, 5, 6, 7\}$ it was generated 300 instances, with a total of 1500 instances for $S = 6$; for $S = 7$ and for each $H \in \{3, 4, 5, 6, 7\}$ it was generated 400 instances, with a total of 2000 instances for $S = 7$; for $S = 8$ and for each $H \in \{3, 4, 5, 6, 7\}$ it was generated 500 instances, with a total of 2500 instances; for $S = 9$ and for each $H \in \{3, 4, 5, 6, 7\}$ it was generated 600 instances, with a total of 3000 instances for $S = 9$; and for $S = 10$ and for each $H \in \{3, 4, 5, 6, 7\}$ it was generated 700 instances, with a total of 3500 instances for $S = 10$. Therefore, a total of 12500 instances were generated. For each fixed pair of S and H values, if it was generated X instances, then there are $X/(H - 1)$ instances for each value of N, where N is between $H(S - 1)$ and $HS - 1$.

The algorithms were implemented in C++ programming language and we used a

computer with Intel Xeon E3-1230 CPU (3.30 GHz) and 32GB of memory to execute Algorithm 1 over all instances, just varying the lower bound it uses. The source code of the implementation is available at "URL" (available after publication).

6.2 Generating the PDB

In Table 6.1 we present information about the generation of the PDBs for different values of N', i.e., different number of blocks in the abstract instance. We generated PDBs with $N' \in \{5, 8, 10\}$. We tried to generate a PDB with $N' = 11$ but that resulted in the process being killed by the operating system, since the system became out of memory. For $N' = 10$ it took almost 7 hours to generate the PDB.

Our instance set contains instances with different values of S and H, so we have to generate PDBs for these different combinations, with $S \in \{6, 7, 8, 9, 10\}$ and $H \in \{3, 4, 5, 6, 7\}$. In Table 6.1 we show the number of unique abstract instances generated and the time to generate them.

We can see that the number of unique instances, as well as the time to generate the PDB, increases very fast with N'. The time to generate a PDB for $N' = 5$ is less than one second, increases to some seconds for $N' = 8$, and becomes several minutes for $N' = 10$, being half an hour for several of the combinations of S and H.

Since the largest PDB that we could generate was with $N' = 10$, we used this one in our experiments. Notice that the larger the value of N' in the PDB, the better the search algorithm tends to be, since in the exploration of the search space, as soon as we have an abstract state with N' blocks, we can stop the search in that subspace as its optimal solution is saved in the PDB.

Later, in Section 6.4, for completeness we present results of the execution of the exact algorithm with different values of N', showing the impact it has on it.

6.3 Algorithm Evaluation with Different Lower Bounds

In this section we present results of the exact algorithms using different lower bounds.

In the first experiment, we used the exact algorithm given in Algorithm 1 to solve all instances, where the only difference was which lower bound was used in the branch and bound algorithm (Algorithm 2) in order to prune nodes. We compared the use of the lower bound LB3 [19], LB4 [17], and our two new lower bounds named LB-LIS and LB-PDB. For each one of the 12500 instances, we set a time limit of 5 minutes for executing the algorithms.

In Figure 6.1 we show the results of this first experiment, where we present in the y-axis the percentage of instances solved to optimality by each algorithm. Notice that we split the instances into two categories: easy instances, with $S \in \{6, 7, 8, 9, 10\}$ and $H \in \{3, 4, 5\}$, and hard instances, with $S \in \{6, 7, 8, 9, 10\}$ and $H \in \{6, 7\}$. This split was done based on the fraction of solutions that could be solved within the time limit.

Table 6.1: Time to generate the PDB for different values of N'. We also present the number of unique instances in the PDB.

		Unique instances			Time (sec)		
S	H	$N'=5$	$N'=8$	$N'=10$	$N'=5$	$N'=8$	$N'=10$
6	3	153	38058	2141418	0	1	105
6	4	243	109278	9246126	0	4	460
6	5	339	173694	19086798	0	8	970
6	6	339	220734	27569118	0	10	1428
6	7	339	255294	33812958	0	12	1796
7	3	153	38065	2158505	0	1	138
7	4	243	109285	9264725	0	5	460
7	5	339	173701	19105397	0	8	963
7	6	339	220741	27587717	0	10	1435
7	7	339	255301	33831557	0	12	1815
8	3	153	38065	2159161	0	1	138
8	4	243	109285	9265381	0	5	469
8	5	339	173701	19106053	0	8	938
8	6	339	220741	27588373	0	10	1464
8	7	339	255301	33832213	0	13	1846
9	3	153	38065	2159170	0	1	145
9	4	243	109285	9265390	0	5	494
9	5	339	173701	19106062	0	8	1032
9	6	339	220741	27588382	0	10	1533
9	7	339	255301	33832222	0	13	1928
10	3	153	38065	2159170	0	1	148
10	4	243	109285	9265390	0	5	501
10	5	339	173701	19106062	0	8	1044
10	6	339	220741	27588382	0	10	1549
10	7	339	255301	33832222	0	13	1940

We can see that the performance of the algorithm using LB3, LB4, LB-LIS, and LB-PDB is similar for the easy instances, with around 95% of all easy instances being solved to optimality. However, for the hard instances, the performance of the algorithm using LB-LIS is superior than when using the other lower bounds.

In Table 6.2 we show more details about the comparison of using LB3, LB4, LB-LIS, and LB-PDB in the exact algorithm. In this table, "AVG ALL" is the average of the number of relocations of all 12500 instances, "Number OPT" is the number of instances solved to optimality among all 12500 instances, "OPT EASY" is the percentage of instances solved to optimality among the easy instances category, "OPT HARD" is the percentage instances solved to optimality among the hard instances category, and "AVG HARD" is the average of the number of relocations among all the hard instances category. Note that the use of the lower bound LB-LIS resulted in solving 186 more instances to optimality than LB3 and solving 942 more instances to optimality than LB-PDB.

It is interesting to note that LB4 from Tanaka and Takki [17] is considered the best lower bound up to date, however in our experiments the results of the exact algorithm

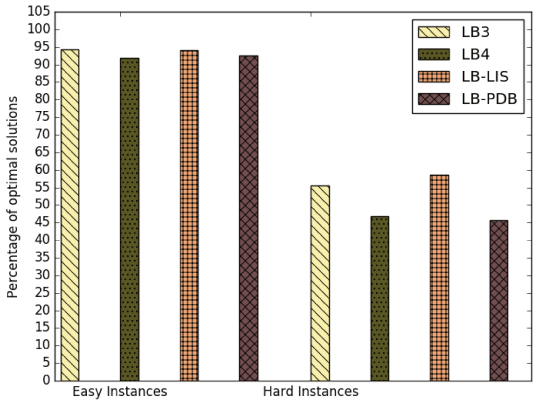


Figure 6.1: Results using LB3, LB4, LB-LIS, and LB-PDB.

measures	LB3	LB4	LB-LIS	LB-PDB
AVG ALL	30.848	31.02	30.627	31.148
Number OPT	9274	8551	9460	8518
%OPT EASY	94.283	91.83	94.116	92.483
%OPT HARD	55.646	46.78	58.661	45.676
AVG HARD	42.368	42.709	41.943	42.944

Table 6.2: Results of the exact algorithm using different lower bounds: LB3, LB4, LB-LIS, and LB-PDB. "AVG ALL" is the average number of relocations of the best solutions found by each algorithm considering all instances. "Number OPT" is the number of optimal solutions found. "%OPT EASY" (resp. "%OPT HARD") is the percentage of optimal solutions found for easy (resp. hard) instances. "AVG HARD" is the average number of relocations considering hard instances.

using it only outperforms the results of LB-PDB. This can be explained by the fact that although LB4 provides tighter lower bounds, it is expensive to compute, since it is an exponential time algorithm, while the other lower bounds are polynomial time bounded. To show this, we computed the values produced by each lower bound for each initial configuration of the 12500 instances in our instance set, as well as the times needed for this computation. The results are presented in Table 6.3. In this table, for each lower bound, we present: (1) the sum of the lower bounds (SUM LB) computed for all instances, (2) the average value of the lower bound (AVG LB) considering all instances, (3) the sum of the times (SUM time) in milliseconds to compute the lower bound for all instances, and (4) the average time (AVG time) to calculate the lower bound. From this table we can see that indeed LB4 generates tighter lower bounds with values being approximately 0.3% higher than the ones computed by LB-LIS, however it is much slower, taking ten times more than LB-LIS.

measures	LB3	LB4	LB-LIS	LB-PDB
SUM LB	323321	333273	332072	323321
AVG LB	25.865	26.661	26.565	25.865
SUM time (ms)	32.526	674.417	64.919	125.984
AVG time (ms)	0.002	0.054	0.005	0.010

Table 6.3: Times and values produced by each lower bound for each initial configuration of the 12500 instances in our instance set. We present the sum of the values (SUM) and the average (AVG) among the 12500 instances.

6.4 Improving the Exact Algorithm Using the PDB

Given that the exact algorithm obtained the best result with LB-LIS, we developed two other variations of the algorithm using the PDB and the concept of abstract states, as explained in Section 5.4 and detailed in Algorithm 4.

The first variation, which we denote by LIS-PDB, uses the PDB to stop earlier the evaluation of subspaces. During the exploration of the search space, if a node being explored has N' blocks, then the optimal solution for it was already computed and is saved in the PDB. So we can stop the search at every node $\mathcal{S}$ if it has N' blocks.

In the second variation, which we denote by LIS-PDB-L, we use the Algorithm 4. In this case we use the PDB to evaluate nodes with N' blocks, as well as a hash table to save optimal solutions of solved states in which at most k' relocations were applied.

First we performed an experimental evaluation of the algorithms using different values of $N' \in \{5, 8, 10\}$ and $k' \in \{5, 8, 10, 12, 15\}$. For this experiment we used a subset of the instance set, since running all variations in all algorithms would took a month to complete the experiments. So, for each combination of S and H, we selected at random 20% of the instances with these values, which means we have a dataset with 20% of all the 12500 instances of the original instance set. The results are presented in Tables 6.4 and 6.5. For this experiments we also set a time limite of 5 minutes.

From Table 6.4, as expected, we see that the algorithm finds more optimal solutions as N' increases. In fact, the values of the best solutions found by all variations of the algorithm are the same, as we can see by the same values of AVG ALL. However, as N' increases the algorithm can prove that the solution returned is optimal. That is the reason why the number of optimal solutions increases as N' increases despite the fact that the solutions values found by all algorithms are the same.

measures	$N' = 5$	$N' = 8$	$N' = 10$
AVG ALL	30.700	30.700	30.700
Number OPT	1897	1914	1929
%OPT EASY	94.666	95.166	95.416
%OPT HARD	58.314	59.157	60.076
AVG HARD	42.080	42.080	42.080

Table 6.4: Comparing performances of LB-LIS-PDB with different N'.

From Table 6.5 we see that the algorithm finds more optimal solutions as k' increases.

We tried to execute the algorithm with $k' = 20$ but the process was killed during execution by the operating system due to a out of memory error.

measures	$k' = 5$	$k' = 8$	$k' = 10$	$k' = 12$	$k' = 15$	$k' = 18$
AVG ALL	30.783	30.777	30.771	30.762	30.760	30.741
Number OPT	1829	1835	1831	1840	1849	1869
%OPT EASY	94.083	94.166	94.166	94.333	94.58	95.08
%OPT HARD	53.639	54.022	53.716	54.252	54.712	55.780
AVG HARD	42.239	42.229	42.216	42.200	42.195	42.162

Table 6.5: Comparing performances of LB-LIS-PDB-L with different k'.

Given the previous results we used $k' = 18$ and $N' = 10$ in the experiments of the improved exact algorithm. The results shown in Figure 6.2 are a comparison among LB-LIS, and the two improved versions LIS-PDB, and LIS-PDB-L. We give more details of this comparison in Table 6.6. Note that the algorithm using LIS-PDB solved 200 more instances to optimality than when using LB-LIS, and the same algorithm solved 219 more instances to optimality than LIS-PBD-L.

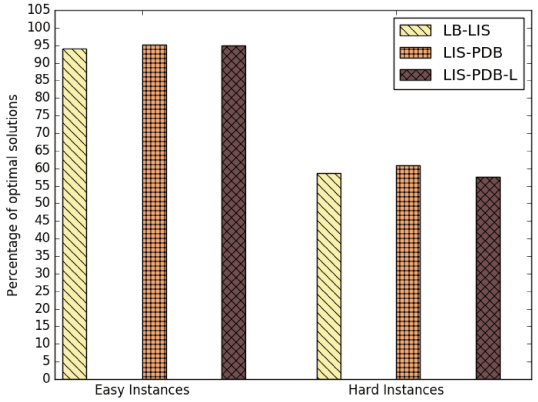


Figure 6.2: Results using LB-LIS, LIS-PDB, and LIS-PDB-L.

measures	LIS	LIS-PDB	LIS-PDB-L
AVG ALL	30.627	30.627	30.642
Number OPT	9460	9660	9441
%OPT EASY	94.116	95.216	94.950
%OPT HARD	58.661	60.723	57.600
AVG HARD	41.942	41.942	41.972

Table 6.6: Comparing performances of LB-LIS, LIS-PDB, and LIS-PDB-L.

We thus conclude that LIS-PDB is the best of all the techniques proposed in this thesis. We note that this algorithm solved approximately 5% more hard instance than the exact algorithm using the previously best lower bound LB3 found in the literature.

Chapter 7

Conclusions and Future Works

In this thesis we presented two new lower bounds for the BRP problem. One of them is denoted by LB-LIS and uses the idea of longest increasing subsequences to calculate the amount of blocking blocks that will keep the property of being blocking blocks even after relocation. The other lower bound is denoted by LB-PDB, and takes the maximum between LB2 (Zhu *et al.* [19]) and partial solutions saved in a pattern database in series of steps to calculate a valid lower bound. We used an exact algorithm to solve a set of 12500 instances, where for each instance it was set a time limit of 5 minutes for the execution of the exact algorithm. First we run the exact algorithm just varying the lower bound it uses during the search. The algorithm found more optimal instances when using LB-LIS, compared to other lower bounds of the literature, such as LB3 [19] and LB4 [17]. As was done by Ku and Arthanari [14], we used the concepts of abstract states and pattern databases as a way to save solutions during the exploration of the search tree. We precomputed a PDB containing solutions for instances with $N' = 10$ blocks and used a hash table to save completely explored nodes during the search with a depth of at most $k' = 18$. The use of the PDB improved the exact algorithm, as it was able to solve more instances to optimality, however the use of the hash table did not improve the algorithm. We believe the access cost of the hash table does not compensate its use.

Further work includes establishing if such lower bound, which is applied to the restricted BRP (when the relocations can be performed only in the stack where the target block is), can also be applied to the unrestricted BRP. Further work also might explore the creation of new lower bounds using different techniques than the ones showed here or the improvement of our techniques.

Bibliography

[1] Marco Caserta, Silvia Schwarze, and Stefan Voß. A new binary description of the blocks relocation problem and benefits in a look ahead heuristic. In *EvoCOP*, pages 37–48. Springer, 2009.

[2] Marco Caserta, Silvia Schwarze, and Stefan Voß. A mathematical formulation and complexity considerations for the blocks relocation problem. *European Journal of Operational Research*, 219(1):96–104, 2012.

[3] Marco Caserta and Stefan Voß. A corridor method-based algorithm for the pre-marshalling problem. In *Workshops on Applications of Evolutionary Computation*, pages 788–797. Springer, 2009.

[4] Marco Caserta, Stefan Voß, and Moshe Sniedovich. Applying the corridor method to a blocks relocation problem. *OR spectrum*, 33(4):915–929, 2011.

[5] Jens Clausen. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

[6] Joseph C Culberson and Jonathan Schaeffer. Searching with pattern databases. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 402–416. Springer, 1996.

[7] Christopher Expósito-Izquierdo, Belén Melián-Batista, and J Marcos Moreno-Vega. A domain-specific knowledge-based heuristic for the blocks relocation problem. *Advanced Engineering Informatics*, 28(4):327–343, 2014.

[8] Ariel Felner, Richard E Korf, and Sarit Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[9] Florian Forster and Andreas Bortfeldt. A tree search procedure for the container relocation problem. *Computers & Operations Research*, 39(2):299–309, 2012.

[10] Raka Jovanovic and Stefan Voß. A chain heuristic for the blocks relocation problem. *Computers & Industrial Engineering*, 75:79–86, 2014.

[11] Kap Hwan Kim and Gyu-Pyo Hong. A heuristic rule for relocating blocks. *Computers & Operations Research*, 33(4):940–954, 2006.

[12] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.

[13] Richard E Korf. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI/IAAI*, pages 700–705, 1997.

[14] Dusan Ku and Tiru S Arthanari. On the abstraction method for the container relocation problem. *Computers & Operations Research*, 68:110–122, 2016.

[15] Yusin Lee and Yen-Ju Lee. A heuristic for retrieving containers from a yard. *Computers & Operations Research*, 37(6):1139–1147, 2010.

[16] Shunji Tanaka and Fumitaka Mizuno. An exact algorithm for the unrestricted block relocation problem. *Computers & Operations Research*, 95:12–31, 2018.

[17] Shunji Tanaka and Kenta Takii. A faster branch-and-bound algorithm for the block relocation problem. *IEEE Transactions on Automation Science and Engineering*, 13(1):181–190, 2016.

[18] Elisabeth Zehendner, Marco Caserta, Dominique Feillet, Silvia Schwarze, and Stefan Voß. An improved mathematical formulation for the blocks relocation problem. *European Journal of Operational Research*, 245(2):415–422, 2015.

[19] Wenbin Zhu, Hu Qin, Andrew Lim, and Huidong Zhang. Iterative deepening a* algorithms for the container relocation problem. *IEEE Transactions on Automation Science and Engineering*, 9(4):710–722, 2012.