Solutions to Real-World Instances of PSPACE-Complete Stacking

F.G. König, M.E. Lübbecke, R.H. Möhring, G. Schäfer*, and I. Spenke*

Technische Universität Berlin, Institut für Mathematik, MA 6-1
Straße des 17. Juni 136, 10623 Berlin, Germany
{fkoenig,m.luebbecke,moehring,schaefer,spenke}@math.tu-berlin.de

Preprint 004/2007

April 10, 2007
Revised: August 8, 2007

Abstract. We investigate a complex stacking problem that stems from storage planning of steel slabs in integrated steel production. Besides the practical importance of such stacking tasks, they are appealing from a theoretical point of view. We show that already a simple version of our stacking problem is PSPACE-complete. Thus, fast algorithms for computing provably good solutions as they are required for practical purposes raise various algorithmic challenges. We describe an algorithm that computes solutions within 5/4 of optimality for all our real-world test instances. The basic idea is a search in an exponential state space that is guided by a state-valuation function. The algorithm is extremely fast and solves practical instances within a few seconds. We assess the quality of our solutions by computing instance-dependent lower bounds from a combinatorial relaxation formulated as mixed integer program. To the best of our knowledge, this is the first approach that provides quality guarantees for such problems.

1 Introduction

In this paper, we investigate a dynamic stacking problem that has applications in many production processes with intermediate storage for bulky items. Our work is motivated by a cooperation with PSI-BT [14], a software company that develops planning software for logistics and production processes in steel plants. A crucial task in such a plant is the transportation and intermediate storage of steel slabs over time, which is the prototype of the general stacking problem considered here. It similarly occurs in other settings such as the shunting of rail cars or container stowage.

We keep our problem formulation general enough to provide a concise, theoretically interesting problem class, but still specific enough to be applicable to different industrial scenarios.

* G. Schäfer and I. Spenke were supported by the DFG Research Center Matheon "Mathematics for key technologies."

In practice, many different constraints like precedence relations, time windows, stack heights, etc. have to be respected, and it is not surprising that the literature abounds with heuristics.

However, no general and versatile problem formulation like ours has been presented before, let alone, solutions to them of proven quality.

From a computer science point of view, our problem is related to motion planning and sorting with networks of stacks. The *Towers of Hanoi* puzzle is just a very simple instance of our model. More interestingly, there are similarities to the much more complex *block sliding* puzzles. Many of these puzzles are known to be PSPACE-complete [8]. We show that this is true also for our stacking problem, which rules out the existence of efficient algorithms in general.

Nevertheless, using ideas from dynamic programming and mixed integer programming, well-established techniques from discrete optimization, we compute solutions to all our real-world instances within 5/4 of an optimum.

Motivation. In integrated steel production, steel is casted in a 24/7 operation in slabs, bars of up to 12 meters length and 30 tons weight. These are rolled to sheet metal in a following batch process. Each slab is assigned to a customer order before it is even casted, and is, thus, individual. The rolling at later points in time is in a given order which (also because of technical reasons) typically differs considerably from the casting sequence. In between, slabs are brought by cranes to a storage area where they are piled up on stacks (for up to several days). If slabs are needed much later (several weeks), they may leave the so-called *hot-buffer* and can be temporarily stored in a much larger *cold-buffer*. In both cases, the number of stacks and their height is limited.

In the hot-buffer, ideally, slabs should be stored in such a way that they are readily available according to the planned batch sequence of the rolling process (each batch may request several dozens of slabs in a given order). This conflicts with the limited space, the casting sequence, and the complicating fact, that slabs exiting a strand caster have to be moved away within tight time windows. Time is less crucial in the cold-buffer, but the system inherent non-conformity of input and output sequences becomes much more visible. Obviously, the buffers constitute a major bottleneck, and a high productivity is desired.

Our Contribution. The STACKING PROBLEM we describe captures practical stacking processes accurately, yet it defines a general and versatile problem type. We show that it is PSPACE-complete, even in a quite basic version. It is therefore unlikely that there exists an efficient algorithm to compute solutions of a proven quality for this problem.

We define a state graph for practical stacking problems and develop a state-valuation function which guides a partial exploration of this typically exponential graph on all feasible stackings. The result is a kind of dynamic programming algorithm which computes good solutions for real-world stacking instances fast.

The power of this approach is that it provides sufficient flexibility to model all constraints that are relevant from a practical point of view and at the same time also allows us to use different techniques and heuristics to speed-up the state space exploration. We also report on the effect of a rollout strategy [2], which gives slightly improved solutions at the expense of a considerable increase in running time.

This is the first time that a proof of the solution's quality for a stacking problem of this versatility is given. We do so by computing an instance-dependent lower bound using a mixed integer program. On our industrial data we obtain solutions that are less than 25% off optimum, usually considerably better.

Related Work. Practical stacking problems have been addressed in the literature a lot, like in steel production [7, 13], container terminals [5, 12], in the very general area of generic planning problems [6], and several more. All of these papers significantly simplify the problems, in particular the precedence constraints implied by stacking. Methodologies applied range from simulated annealing [7], simulations [5], genetic algorithms [13], state space evaluations [3, 10], but, to the best of our knowledge, no principally exact approaches have been suggested.

2 Problem Definition and Hardness Results

We formally describe the STACKING PROBLEM, omitting a few practically relevant details; we comment on these where appropriate. Let $[n] := \{1, \ldots, n\}$.

Incoming items. We have a set $I := [n]$ of incoming items that arrive on m parallel inputs over time. Each item $i \in I$ is associated with a *time window* $[r_i, d_i] \subseteq \mathbb{R}^+$; r_i is the *release time* and d_i the *due time* of item i. An item i must be removed from the input within its time window. We may either move it to one of the k *buffer stacks* or directly to one of the *target stacks* defined below. We assume that at any time at most one item is available at every input.

Buffer stacks. Let $\mathcal{S} := \{S_1, \ldots, S_k\}$ be a set of k *buffer stacks*. Stack $S \in \mathcal{S}$ can hold up to $h(S)$ items. We write $i \rightsquigarrow_S j$ iff i lies, not necessarily directly, on top of j in S. We use the same notation for the target stacks defined below. An allocation of items to stacks $C := (S_1, \ldots, S_k)$ with their respective positions is called a *configuration*. The initial configuration C_0 need not be empty. The set of items that are allocated to the buffer stacks in C_0 is denoted by J. The entire set of items is thus $V := I \cup J$.

Stacking constraints. Items may not be placed arbitrarily on top of each other. We model stacking restrictions by a *conflict graph*. Let $G := (V, A)$ be a directed graph with vertex set V and arc set A. Item $i \in V$ cannot be placed *directly* on top of item j iff $(i, j) \in A$. We call a configuration $C = (S_1, \ldots, S_k)$ *feasible* if every buffer stack $S \in \mathcal{S}$ contains at most $h(S)$ items and for all $i, j \in V$ such that i lies directly on top of j, we have $(i, j) \notin A$. We assume that the initial configuration C_0 is feasible.

Target stacks. We are given a set $\mathcal{T} := \{T_1, T_2, \ldots, T_\ell\}$ of *target stacks*. Each target stack $T_i \in \mathcal{T}$ specifies an order (from first to last) in which the respective items have to be collected. Every item $i \in V$ occurs in at most one target stack; define $t_i \in \mathcal{T}$ as i's target stack and let $t_i := \emptyset$ if no such target stack exists. Items for no more than w target stacks may be collected in parallel, capturing the notion of limited space in the exit area of the buffer.

Once we have started collecting items for a target stack, we may only move away, or *dispose*, this target stack when all of its items have been collected. Additionally, the order in which we dispose target stacks must obey precedence constraints defined by a partial order $\prec_{\mathcal{T}}$ on $\mathcal{T}$: if $T_1 \prec_{\mathcal{T}} T_2$ for target stacks $T_1, T_2 \in \mathcal{T}$ then T_2 can only be disposed after T_1 is disposed.

Moves and objective. We have four possible kinds of moves; (a) an item can be moved from an input to a buffer stack, (b) from an input to its target stack (a *direct move*), (c) from the top of a buffer stack to the top of another buffer stack (this is called *restacking*), and (d) from the top of a buffer stack to a target stack.

A move is feasible if it respects all the conditions like time windows, height bounds, stacking conflicts, accessibility of target stacks, etc. defined above. Transport times are known, but important for feasibility only, since they are small as compared to pickup and drop-off times for items. Thus, our goal is to build all target stacks with a minimum number of feasible moves, starting from the initial configuration C_0. Naturally, this includes determining the exact order in which target stacks are started and disposed. The *feasibility version* asks whether some feasible sequence of moves exists to build all target stacks.

Theorem 1. *The* STACKING PROBLEM *is in PSPACE.*

Proof. We exploit Savitch's Theorem [11] which states the equivalence between the complexity classes PSPACE and NPSPACE. We can represent any configuration of the STACKING PROBLEM in polynomial space. Moreover, all possible moves from a given configuration can be computed in polynomial time. We can therefore perform a nondeterministic search using only polynomial space: In each step we choose one of the possible moves nondeterministically and only keep track of the current state. Savitch's Theorem states that any such nondeterministic PSPACE algorithm can be transformed into a deterministic one. □

We show that the feasibility version of the STACKING PROBLEM is PSPACE-complete using the *nondeterministic constraint logic model of computation (NCL)* introduced by Hearn and Demaine [8]. The NCL model is an alternative view on the complexity class PSPACE. It does not need to adopt a two-player view (like in reductions from QUANTIFIED BOOLEAN FORMULAS), and is thus much better suited to our problem.

More formally, we are given an undirected graph $\hat{G} = (\hat{N}, \hat{E})$ with non-negative integer weights on the edges and integral minimum inflow constraints for the nodes. A *configuration* of $\hat{G}$ corresponds to an orientation of the edges such that for every node the sum of the weights of all incoming edges is at least the minimum inflow constraint of that node. A move from one configuration to another corresponds to the reversal of a single edge such that all inflow constraints remain satisfied. We consider the decision problem whether, starting from a given initial configuration $\hat{C}_0$, there exists a sequence of moves such that a designated edge can be reversed. This problem is referred to as CONFIGURATION-TO-EDGE. It is PSPACE-complete even if $\hat{G}$ is a planar AND/OR graph, i.e., consists of the very simple OR and AND nodes only, depicted in Fig. 1 (a) and (b).

Theorem 2. *The feasibility version of the* STACKING PROBLEM *is PSPACE-complete, even if there are no incoming items, each buffer stack can hold at most three items, and only one item is requested at a target stack.*

Proof. We present a polynomial-time reduction from CONFIGURATION-TO-EDGE with an AND/OR graph $\hat{G} = (\hat{N}, \hat{E})$ to our STACKING PROBLEM. To this end, we construct AND and OR gadgets that consist of two buffer stacks each. Abusing notation slightly, we have one item $e \in I$ for every edge $e \in \hat{E}$, and every buffer stack $S \in \mathcal{S}$ can hold at most two or three items. We consider an edge e directed outward a node u iff item e is placed on the buffer stacks corresponding to u.

Consider an OR node $u \in \hat{N}$, see Fig. 1(a). Our OR gadget consists of two buffer stacks S_u^1 and S_u^2 on which only the items a, b or c can be placed, see Fig. 1(c). We enforce this restriction by placing dummy items z_u^1 and z_u^2 at the bottom of S_u^1 and S_u^2, respectively, and adding arcs (x, z_u^1) and (x, z_u^2) for all $x \notin \{a, b, c\}$ to our stacking conflict graph. Imposing a height constraint of two, we can place at most two items in $\{a, b, c\}$ on these stacks. Edge $x \in \{a, b, c\}$ is directed outward of u iff x is placed on S_u^1 or S_u^2, that is, at least one edge must be directed inward. Clearly, this gadget implements an OR node.

Next consider an AND node $u \in \hat{N}$, with incident edges a, b and c, see Fig. 1(b). Our AND gadget again consists of two buffer stacks S_u^1 and S_u^2. Again with the help of dummy items, we enforce the following placement restrictions. There are two helper items h_b and h_c which can be placed on top of each other, and at the bottom of S_u^1 or S_u^2. The heavy item a can be placed only at the bottom of S_u^1 or S_u^2. The light items b and c can be placed only on their corresponding helper item h_b and h_c, respectively. A height bound of three on S_u^1 and S_u^2 now guarantees that edge a can be directed outward only, if edges b and c are directed inward. One easily checks that exactly all feasible edge configurations are represented by a feasible item configuration, so this gadget implements an AND node.

For some initial configuration $\hat{C}_0$ and a designated edge $e \in \hat{E}$, CONFIGURATION-TO-EDGE now reduces to the decision problem whether, starting from the buffer stack configuration corresponding to $\hat{C}_0$, there exists a sequence of moves such that the item e can eventually be moved. This decision problem can be simulated by letting the dummy item that is hidden by e in the initial buffer stack configuration be the only item that is requested at the target stack. $\square$

In the PSPACE-completeness proof above we exploit crucially that complex restacking operations may be necessary in order to access a particular item. The

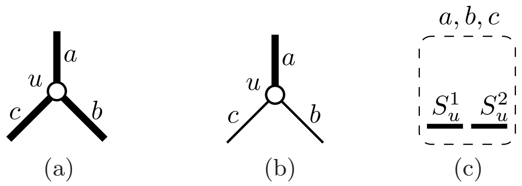

Fig. 1. Illustration of an OR (a) and AND (b) node. Bold edges have weight two, light edges have weight one; the minimum inflow constraint is two. For the OR node, one edge can be directed outward iff at least one of the other two edges is directed inward. For the AND node, edge a may be directed outward iff b and c are directed inward. The gadget for both nodes is depicted in (c).

key ingredients seem to be that we start with a non-empty initial configuration and that every item has only a very limited number of buffer stacks onto which it can be placed, i.e., the stacking conflict graph is rather dense. The next theorem shows that the problem remains hard even if we remove these assumptions.

Theorem 3. *The problem of deciding whether a given target sequence can be built without any restacking operations is NP-hard if all buffer stacks have a uniform height bound of $h \geq 6$, even if we start with empty buffer stacks and there are no conflicts between items.*

Proof. We reduce MUTUAL EXCLUSION SCHEDULING [9] for permutation graphs to our STACKING PROBLEM. In this problem, we are given a permutation graph $\hat{G}(\Pi) = (\hat{N}, \hat{E})$ of n nodes and a positive integer h. Jansen [9] proved that for every fixed $h \geq 6$, the decision problem whether there exists a partition of $\hat{N}$ into t independent sets of size at most h is NP-hard.

Let $\Pi = \langle \pi_1, \ldots, \pi_n \rangle$ be the permutation of $\langle 1, \ldots, n \rangle$ that defines the permutation graph $\hat{G}(\Pi)$. We define an instance of our STACKING PROBLEM as follows: Items appear at the input in the order $\langle \pi_1, \ldots, \pi_n, n+1 \rangle$ and are requested at the target stack in the order $T = \langle n+1, n, \ldots, 1 \rangle$. The buffer stacks are empty initially and have the same height bound h. It is not difficult to verify that the target stack T can be built without any restacking operations, i.e., using $2n+1$ moves, iff $\hat{N}$ can be partitioned into $t = k$ independent sets of size at most h. $\square$

3 Guided Graph Search

We use graph terminology to specify the state space which is searched for a solution to the STACKING PROBLEM: We associate a node with each possible buffer configuration C and define the neighborhood of a node as the set of all configurations that can be constructed from C by one feasible move. The notion of feasibility of a move includes that *after* the move it must still be possible to remove all items from the inputs respecting their time windows. The due time to remove items from their respective input in order to be able to feasibly serve all inputs can be determined by a simple backward-calculation.

Note that the actual nodes of the state graph do not only consist of a buffer configuration C but also comprise a time stamp t and a notion of progress made. This progress includes the items already moved from the inputs, say A, and the set of target stacks for which items are currently being collected Ω. We call $\Sigma := (C, t, A, \Omega)$ a *state*; every state corresponds to exactly one node in the state graph.

Obviously, it is impossible to generate even only a significant part of the state graph—already the number of possible buffer configurations is huge. Thus we use a *valuation function* on the buffer states to tell us which parts of the graph are "interesting". This approach is related to the idea of approximate cost-to-go functions in dynamic programming [1].

The definition of a proper valuation function depends on the nature of the instances to be solved. We give two examples in Sections 3.1 and 3.2. The first one yields an optimal algorithm for the Towers of Hanoi problem. The latter one leads to good solutions for instances of our real-world application and may well be suitable for other applications involving the storage of items on stacks.

The power of this approach is two-fold: On the problem formulation side, all constraints regarding the structure of the stacks, travel times and the necessity to respect time windows when removing items from the inputs can easily be modeled by modifying the rules with which a node's neighborhood is created; on the solution side, different techniques can be used for the exploration of the graph allowing fine-tuning of the algorithm towards speed, robustness, precision, flexibility or other goals.

3.1 The Towers of Hanoi Example

We formulate the Towers of Hanoi problem as a STACKING PROBLEM and solve it optimally by our algorithm using a suitable valuation function. In this case, there are no arriving items and no constraints for the number and height of the stacks; only stacking constraints have to be respected.

There are n discs $0, 1, \ldots, n-1$ with diameters $d(0) > d(1) > \cdots > d(n-1)$ and a larger disc may never be stacked on a smaller one. In the beginning, all discs are stacked feasibly on S_1 and the goal is to stack all of them feasibly on S_3 by only moving one disc at a time.

Then following iterative algorithm leads to the unique optimal solution [15]. First, arrange the three stacks S_1 (source), S_2, and S_3 (target) into a triangle with S_1 on the left, S_2 on top and S_3 on the right. Apply two rules:

($R1$) If n is even, move even discs clockwise and odd discs counter-clockwise only; if n is odd, do it vice versa.

($R2$) In move k, starting at 1, move the disc corresponding to the power of the right-most (i.e., least-significant) 1-bit in the binary representation of k.

These two rules imply the following two facts; we omit their proofs.

($F1$) In the unique optimal solution, a disc i is never moved to a non-empty stack S_ℓ, such that the difference between i and the top-most disc j in S_ℓ is even.

($F2$) In the unique optimal solution, a disc is only moved to an empty stack if every other feasible move violates Fact ($F1$).

We use $(F1)$ and $(F2)$ to define a valuation function. Let $e(C)$ be the number of empty stacks in a configuration C and $v(S_\ell) = 1$ if the difference between the two topmost discs in stack S_ℓ is even, and 0 otherwise. Let $w > 1$ be an arbitrary number making $e(C)$ a tie-breaker enforcing Fact $(F2)$.

$$val^{hanoi}(C) := w \cdot \left(\sum_{\ell=1}^{3} v(S_\ell) \right) - e(C) \tag{1}$$

Let C_1 and C_2 be two neighbored configurations in the unique optimal sequence of moves. Then $val^{hanoi}(C_2)$ is the unique minimum among all $val^{hanoi}(C)$, where C is reached from C_1 by a feasible move. Applying our greedy graph search we thus obtain the unique optimal sequence of moves.

It is easy to see that time and space required by our search are of the same order as the known lower bounds (see [4]) of $\Theta(2^n)$ and $\Theta(n)$, respectively.

3.2 The Application-Driven Greedy Approach

Given a buffer configuration $C = (S_1, \ldots, S_k)$ and the set of target stacks $\mathcal{T}$ to be compiled, there is a natural lower bound on the number of moves needed: We count one move for each item that will be moved from a buffer to a target stack plus one move for each item which has to be moved out of the way; we will call the latter the number of *false positions* in C.

In order to know more precisely which items must be "moved out of the way", we determine in a preprocessing step a linear extension $<_\mathcal{T}$ of the partial order $\prec_\mathcal{T}$ in which we will begin collecting items for target stacks. This is done by a simple heuristic, which first orders all target stacks according to the latest release time of one of their items, and then ensures $T_i <_\mathcal{T} T_j$ for all T_i, T_j with $T_i \prec_\mathcal{T} T_j$ by delaying T_j until directly after T_i in $<_\mathcal{T}$ if necessary. Now assume without loss of generality that $T_1 <_\mathcal{T} T_2 <_\mathcal{T} \ldots$

We now define the following partial precedence order on the items: For two items i, j, we write $i \prec j$ iff $t_i <_\mathcal{T} t_j$ and $\{t_i, t_j\} \not\subseteq \Omega$, or $t_i = t_j$ and $j \rightsquigarrow_{t_i} i$. The number of false positions for item i on stack S is then

$$false(i, S) := |\{j \in S \ : \ j \rightsquigarrow_S i \ \wedge \ i \prec j\}| \ . \tag{2}$$

From any configuration C without false positions, we can build all required target stacks without a single restacking operation. Thus, we define a valuation function which deems those buffer configurations interesting which have few false positions.

We introduce a valuation functions val^{app} on states $\Sigma = (C, t, A, \Omega)$:

$$val^{app}(\Sigma) := \sum_{\theta=\gamma}^{|\mathcal{T}|} d^{\gamma-\theta} \cdot val_\theta^{app}(\Sigma), \tag{3}$$

where where γ is the smallest index of a target stack not yet disposed and $d \leq 1$ is some discount factor determining how fast the weight of false positions decreases for target stacks to be compiled in the future,

$$val_\theta^{app}(\Sigma) := \sum_{\ell=1}^{k} \sum_{i \in S_\ell \cap T_\theta} false(i, S_\ell) \qquad \forall \theta = \gamma, \ldots, |\mathcal{T}| \tag{4}$$

accounts for the total number of false positions for items in T_θ.

Note that the sum of all false position in a configuration constitutes a lower bound on the number of necessary moves only with respect to $<_\mathcal{T}$. Theoretically, starting to collect items for target stacks in a different order may lead to a better solution. Yet we have found that the number of false positions computed with respect to $\prec_\mathcal{T}$ only is too weak of a bound to be of any help in the state space search.

In view of some uncertainty in related processes and the strict bounds on computation time prescribed by the slab stacking application, the most important features of the used algorithm are speed and the ability to react to changes in data. The latter can naturally be achieved by defining the state of the buffer after a change in data as a new initial state and rerunning the algorithm, making the speed of the algorithm an even more important goal for practical purposes.

Thus, a fast greedy variant of the graph search procedure is used, which in each current node of the state graph computes the valuation function for all neighboring nodes and picks the one with best value. To break ties, we prefer moves which take less time and try to maintain as many unused stacks in the buffer as possible. Somewhat surprisingly, this simple and extremely fast search strategy leads to good solutions to practical instances, as described in Section 5.

An additional parameter, the *lookahead* f, is introduced to speed up the calculation of the valuation function for each buffer state, especially for instances where data is available for many future target stacks: Instead of considering *all* values val_θ^{app} in (3), we only take the first $f+1$ values into account; i.e., we replace $|\mathcal{T}|$ by $\gamma + f$ in (3). The effects of different choices for f for the practical instances are described in Section 5.

Note that this valuation function favors buffer states in which items can be moved to a target stack, but does not yet encourage the algorithm to actually do so. Thus, we introduce an additional rule for the choice of a neighbor, always preferring a state which moves an item to a target stack over one that does not. The order in which the algorithm starts to build the target stacks is computed in a preprocessing step by the following simple rule: Build those target stacks first, for which all items have arrived at the buffer the earliest while respecting precedence constraints on the target stacks where necessary.

In Section 5, we also report on the effects of combining our greedy algorithm with a rollout strategy [2]. Due to space limitations, we only sketch the basic idea here: If we have reached a state Σ in our state graph, we run the greedy algorithm from each neighboring state and then continue with the state that returns the mimimum number of moves. Clearly, this approach entails computational overhead, but may improve the solution quality.

4 Lower Bounds from a Combinatorial Relaxation

In order to assess the quality of our solutions we compute lower bounds on the optimal number of moves per instance. If the initial configuration is non-empty, one may count how many items on top of an item are needed later and thus have to be moved away. This bound is used for the cold-buffer instances below.

A more elaborate technique is needed for the hot-buffer instances. We formulate a mixed integer program (MIP) and derive a bound from its linear programming (LP) relaxation. However, an MIP which captures the STACKING PROBLEM in full detail is far from being computationally tractable. In fact, in order to represent an exponential solution one would have to work explicitly with an exponential number of variables.

Instead, we devise a non-obvious but surprisingly useful *combinatorial* relaxation of the problem, which is mainly based on the time windows at the inputs: We assume a sufficiently large number $k \geq n$ of stacks. This implies that we need not take care of infeasible buffer configurations because each item can use its own stack. As a consequence, we disregard the concept of stacks and conflicts at all, except for target stacks. The objective is to maximize the number of direct moves from the inputs to the target stacks. For this relaxation, the MIP computes a best possible linear extension $<_{\mathcal{T}}$ of the partial order $\prec_{\mathcal{T}}$ in which to begin collecting items for target stacks.

Even though the MIP is deprived the essential stacking character, it turns out that the LP relaxation gives a good lower bound in practice, see the hot-buffer instances in Section 5. The MIP is stated in appendix A

5 Computational Results

We tested two different sets of industrial instances supplied by PSI-BT: *hot-buffer instances* with few buffer stacks and tight time windows and *cold-buffer instances* with many more buffer stacks, but without incoming items. In both cases, all target stacks need to be built with a minimum number of moves. All experiments were performed on a standard PC running Linux.

Hot-buffer instances. The time horizon for these instances is up to 12 hours. All instances have 3 strand caster inputs, 3 target stacks can be built in parallel. The number of buffer stacks varies between 14 and 16. The crane can transport one item at a time. The numbers of needed moves are stated in Table 1.

The runtime for the greedy algorithm is less than 0.1 seconds per move. The optimality gap is below 12% and can be improved to less than 10% using a rollout strategy. An optimal integer solution to the MIP was computed using the commercial solver CPLEX 10.1 within some minutes to a few hours. No integer solution was found for instances `v3` and we used the LP bound instead.

Unfortunately, no data about the crane transports that are nowadays performed by the manual operators is available. However, the steel plant states that

Table 1. Solution quality for the hot-buffer instances of the greedy algorithm (and rollout enhancement). LB refers to the MIP lower bound described in Section 4.

	v1	v2a	v2b	v3
LB	325	496	251	732
greedy	362	561	264	794
gap (%)	10.2	11.6	4.9	7.8
rollout	359	538	260	785
gap (%)	9.5	7.8	3.5	6.8

at least about half of all transports are restacking operations, while in our solutions they account to less than 20%. This suggests the buffer performance could be greatly improved by the implementation of our algorithm in practice.

Cold-buffer instances. All instances have 50 buffer stacks, and 5 target stacks can be built simultaneously. There are no incoming items and therefore our MIP lower bound does not apply. The crane can transport several items, up to a maximum weight limit, simultaneously. Again, numbers of needed moves are given in Table 2.

Table 2. Solution quality for the cold-buffer instances of the greedy algorithm (and rollout enhancement). LB is the simple lower bound counting false positions.

	aa1	aa2	aa3	aa4	ab1	ab2	ab3	ab4	ab5	ab6	ab7	ab8	ab9	ab10
LB	30	52	109	109	105	111	108	103	104	103	110	110	113	104
greedy	36	64	128	126	131	135	126	125	119	126	129	130	129	118
gap (%)	20.0	23.1	17.4	15.6	24.8	21.6	16.7	21.4	14.4	22.3	17.3	18.2	14.2	13.5
rollout	34	60	119	120	124	125	117	114	115	113	122	122	120	113
gap (%)	13.3	15.4	9.2	10.1	18.1	12.6	8.3	10.7	10.6	9.7	10.9	10.9	6.2	8.7

For these instances, more than 50% of the overall moves are restacking moves. Our algorithm needs less than 3 seconds per move. The solution quality is below 25%, despite the large share of restacking moves.

In all our experiments, the computational overhead produced by the rollout strategy was enourmous (increase in runtime by a factor of more than 1000), while the gain in quality is marginal—the application of this method in practice is therefore not conceivable.

6 Discussion

A major difficulty in developing better lower bounds is to capture the problem-inherent restacking operations.

The Towers of Hanoi example shows that exponentially many restacking operations may be needed; also our PSPACE-completeness proof relies crucially on these operations. On the other hand, in our practical instances we observed only a rather small number of restacking moves. It would be very interesting to characterize the hardness of an instance by means of its "restacking complexity". Yet, in an ongoing work, we have experienced that such a characterization is not obvious at all, even for two buffer stacks only.

One may argue that a solution quality of 25% off optimum is rather weak. Three comments are in order here. First, we conjecture that the quality of the computed solutions is much better than what we are able to prove. Second, in the area of approximation algorithms, an approximation factor of 5/4 is considered to be very good for a problem that does certainly not admit a PTAS. Third, in recent years we have witnessed tremendous advances in the area of computational combinatorial optimization, in particular mixed integer programming. We believe that striving for close-to-optimal solutions is a necessary and fruitful step in advancing the field even further.

We hope that our results encourage further investigations concerning the applicability of discrete optimization to PSPACE-complete problems. The practical relevance of these approaches in industry is evident. In fact, a prototype implementation of our algorithm is used to evaluate the buffer performance of two steel plants already. Animated visualizations of our stacking plans were shown to several practitioners, who were quite impressed by the low share of restacking operations and the anticipation of slabs needed on target stacks in the future.

References

1. D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.
2. D.P. Bertsekas, J.N. Tsitsiklis, and C. Wu. Rollout algorithms for combinatorial optimization. *Journal of Heuristics*, 3(3):245–262, 1997.
3. B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. *Proceedings of the 14th National Conference on AI*, pages 714–719, 1997.
4. P. Cull and E.F. Ecklund. Towers of hanoi and analysis of algorithms. *The American Mathematical Monthly*, 90:407–420, 1985.
5. R. Dekker, P. Voogd, and E. van Asperen. Advanced methods for container stacking. *OR Spectrum*, 28:563–586, 2006.
6. M. Fox and D. Long. Progress in AI planning research and application. *CEPIS*, 3:10–25, 2002.
7. J. Hansen and J. Clausen. Crane scheduling for a plate storage. Technical Report 1, Informatics and Mathematical Modelling, Technical University of Denmark, 2002.
8. R.A. Hearn and E.D. Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1–2):72–96, 2005.
9. K. Jansen. The mutual exclusion scheduling problem for permutation and comparability graphs. *Information and Computation*, 180:71–81, 2003.
10. D. McDermott. A heuristic estimator for means ends analysis in planning. *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems*, pages 142–149, 1996.
11. W.J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

12. D. Steenken, S. Voß, and R. Stahlbock. Container terminal operation and operations research - a classification and literature review. *OR Spectrum*, 26:3–49, 2004.
13. L. Tang, J. Liu, A. Rong, and Z. Yang. Modelling and a genetic algorithm solution to the slab stack shuffling problem in implementing steel rolling schedulings. *International Journal of Production Research*, 40(7):1583–1595, 2002.
14. `http://www.psi-bt.com`.
15. T. Walsh. The towers of hanoi revisited, moving the rings by counting the moves. *Information Processing Letters*, 15(2):64–67, 1982.

A The MIP

Because of a time discretization there are pseudo-polynomially many variables; they have the following meaning:

- $y_i = 1 \Leftrightarrow$ item i is moved directly from its input s_i to its target t_i
- τ_i^{in} is the point in time at which item i is removed from s_i; and τ_i^{out} is the point in time at which it is moved to its target stack t_i
- b_q and e_q denote the points in time at which assembly of target stack T_q is started and ended, respectively
- $x_{q\tau} = 1 \Leftrightarrow$ target stack T_q is being assembled at time τ

- let $\theta := \max_{i \in I} d_i$ denote the latest due time of any item
- M is a constant chosen to be large enough; any $M \geq \theta$ will suffice

Every item has to be moved at least twice, except if it is moved directly from its input to its target stack. This is expressed in the objective function (5a). The reasoning of the constraints is as follows:

- constraint (5b) enforces for each target stack T_q, if item j is above i on T_q that i cannot be moved before j
- by (5c) an item cannot be moved to its target stack earlier than it is moved from its input; for a direct move these times coincide because of (5d)
- the correct ordering of each target stack T_q is preserved via (5e)
- constraint (5f) reflects the partial order among target stacks
- an item can be moved to a target stack T_q only if T_q is being assembled; this is ensured by (5g) and (5h)
- constraints (5i), (5j) and (5k) ensure the correct setting of $x_{q\tau}$ variables: If target stack T_q is being assembled at time τ we force $x_{q\tau} = 1$. Otherwise, constraints (5l) and (5m) force $x_{q\tau} = 0$. For this (technical) construction we need binary variables b_q^τ and e_q^τ (not to be confused with b_q and e_q)
- the purpose of (5i)–(5m) is to limit the number of simultaneously assembled target stacks to w; expressed by (5n)
- the remaining constraints (5o)–(5t) describe the respective variables' domains, in particular the items' time windows

Most constraints (involving time) can be strengthened via minimal times δ needed to move items around. This is omitted to simplify the presentation, but used in our computations. Note that the mixed integer program computes a best possible feasible ordering in which to assemble the target stacks.

Table 3. Mixed integer program for obtaining a combinatorial lower bound on the optimal number of moves for the stacking problem

$$\min \quad 2n - \sum_{i=0}^{n} y_i \tag{5a}$$

$$\text{s.t.} \quad \tau_i^{\text{in}} \le \tau_j^{\text{in}} + (1 - y_j)M \qquad \forall i, j \in I: \ t_i = t_j \ \wedge \ i \rightsquigarrow_{t_i} j \tag{5b}$$

$$\tau_i^{\text{in}} \le \tau_i^{\text{out}} \qquad \forall i \in I \tag{5c}$$

$$\tau_i^{\text{out}} \le \tau_i^{\text{in}} + (1 - y_i)M \qquad \forall i \in I \tag{5d}$$

$$\tau_{T_{q,p-1}}^{\text{out}} \le \tau_{T_{q,p}}^{\text{out}} \qquad \forall T_q \in \mathcal{T}, \quad p = 1, \ldots, |T_q| - 1 \tag{5e}$$

$$e_q \le e_r \qquad \forall T_q, T_r \in \mathcal{T}: \ T_q \prec T_r \tag{5f}$$

$$b_{t_i} \le \tau_i^{\text{out}} \qquad \forall i \in I \tag{5g}$$

$$\tau_i^{\text{out}} \le e_{t_i} \qquad \forall i \in I \tag{5h}$$

$$1 \le b_q - \tau + b_q^{\tau}(M + 1) \qquad \tau = 0, 1, \ldots, \theta, \ \forall q \in \mathcal{T} \tag{5i}$$

$$1 \le \tau - e_q + e_q^{\tau}(M + 1) \qquad \tau = 0, 1, \ldots, \theta, \ \forall q \in \mathcal{T} \tag{5j}$$

$$e_q^{\tau} + b_q^{\tau} \le 1 + x_{q\tau} \qquad \tau = 0, 1, \ldots, \theta, \ \forall q \in \mathcal{T} \tag{5k}$$

$$b_q \le \tau + (1 - x_{q\tau})M \qquad \tau = 0, 1, \ldots, \theta, \ \forall q \in \mathcal{T} \tag{5l}$$

$$\tau \le e_q + (1 - x_{q\tau})M \qquad \tau = 0, 1, \ldots, \theta, \ \forall q \in \mathcal{T} \tag{5m}$$

$$\sum_{q \in \mathcal{T}} x_{q\tau} \le w \qquad \tau = 0, 1, \ldots, \theta \tag{5n}$$

$$r_i \le \tau_i^{\text{in}} \qquad \forall i \in I \tag{5o}$$

$$\tau_i^{\text{in}} \le d_i - 1 \qquad \forall i \in I \tag{5p}$$

$$0 \le b_q \qquad \forall q \in \mathcal{T} \tag{5q}$$

$$e_q \le \theta \qquad \forall q \in \mathcal{T} \tag{5r}$$

$$y_i \in \{0, 1\} \qquad \forall i \in I \tag{5s}$$

$$b_q^{\tau}, \ e_q^{\tau}, \ x_{q\tau} \in \{0, 1\} \qquad \tau = 0, 1, \ldots, \theta, \ \forall q \in \mathcal{T} \tag{5t}$$