Ann Oper Res (2010) 179: 57–76
DOI 10.1007/s10479-008-0461-8

Space and time allocation in a shipyard assembly hall

Maud Bay · Yves Crama · Yves Langer · Philippe Rigo

Published online: 7 November 2008

Abstract We present a space and time allocation problem that arises in assembly halls producing large building blocks (namely, a shipyard which assembles prefabricated keel elements). The building blocks are very large, and, once a block is placed in the hall, it cannot be moved until all assembly operations on this block are complete. Each block must be processed during a predetermined time window. The objective is to maximize the number of building blocks produced in the hall.

The problem is modeled as a 3-dimensional bin packing problem (3D-BPP) and is handled by a Guided Local Search heuristic initially developed for the 3D-BPP. Our computational experiments with this heuristic demonstrate that excellent results can be found within minutes on a workstation. We also describe some additional real-life constraints arising in the industrial application, and we show how these constraints can be conveniently and flexibly integrated in the solution procedure.

1 Introduction

The aim of this paper is to present an industrial space allocation and scheduling problem arising in shipyard assembly halls dedicated to the processing of voluminous building blocks, and to demonstrate the efficiency, in this applied context, of algorithmic approaches originally designed for the solution of 3-Dimensional Bin Packing problems.

M. Bay · Y. Crama (✉) · Y. Langer
HEC Management School, University of Liège, Boulevard du Rectorat 7 (B31), 4000 Liège, Belgium
e-mail: Yves.Crama@ulg.ac.be

M. Bay
e-mail: Maud.Bay@ulg.ac.be

Y. Langer
e-mail: Yves.Langer@belpex.be

P. Rigo
Naval Architecture and Transportation Systems (ANAST), University of Liège,
Chemin des Chevreuils 1 (B52-3), 4000 Liège, Belgium
e-mail: Ph.Rigo@ulg.ac.be

This study was carried out at *Aker Yards France* (previously known as *Les Chantiers de l'Atlantique*), one of the major European shipyards located in Saint Nazaire, France, at the mouth of river Loire. This shipyard covers all the activities involved in the shipbuilding process, from the basic pre-design phase to the delivery of sea-ready vessels. Its main product lines consist of passenger ships (big cruise liners and car ferries), LNG tankers, military ships (frigates and logistical ships), etc. From this shipyard came many famous liners such as "*France*" (1912), "*Ile De France*" (1927), "*Normandie*" (1935), and "*Norway*" (1960). Recently, *Aker Yards France* built up the world's largest ocean liner, "*Queen Mary 2*". Its production, completed in 2003, took less than two years.

Building such large ships requires the production and the assembly of tens, or even hundreds of thousands of steel elements and pipes. It includes welding hundreds of kilometers of lines, painting several hundred thousands square meters of surface, and handling subassemblies larger than many townhouses. These complex and numerous activities require careful planning and logistics.

The shipbuilding process has changed radically over the three last decades. Formerly, most of the work took place in a dry dock, with the ship constructed almost piece by piece from the ground up. However, advances in technology and more detailed planning have made it possible to divide the vessel into subunits, called blocks and panels, which integrate utilities and other systems. The blocks are assembled in dedicated halls and are composed of subparts and subassemblies which have been produced in other facilities. After assembly, the blocks are transported to the dry dock where they are fitted together. This process is faster, less expensive and provides better quality management than previous practices. Furthermore, it lends itself to increased use of automation and robotics, which not only decreases costs, but also reduces the workers' exposure to chemical and physical hazards.

This paper focuses on the space allocation and planning decisions concerning the assembly halls. Each hall is fully dedicated to the production of blocks and is divided into several equal-sized rectangular areas (e.g., four areas for the main hall under study), each of which is large enough to contain a few blocks simultaneously (see Sect. 5.1 for additional details). The blocks are voluminous and heavy so that, once the assembly of a block has started, this block cannot be moved until all its required assembly operations have been completed. Then, it is transported out of the hall. The objective, as defined by the shipyard managers, is to maximize the number of building blocks produced in a given hall over a certain time horizon (we return to this point in Sect. 6). In the sequel, we refer to this problem as the *Space and Time Allocation* (STA) problem for shipyard assembly halls.

In Sect. 2, we give a more precise description of the STA model, and we discuss its relation to the 3-Dimensional Bin Packing problem. Our solution approach is actually largely inspired from previous work on bin packing. We describe our heuristic algorithms in Sect. 3 (to test for the existence of feasible solutions) and in Sect. 4 (to find a large feasible subset of blocks). Section 5 presents the results of our computational experiments, including a comparison with the results provided by a standard constraint programming approach. Finally, Sect. 6 explains how additional industrial issues have been taken into account in the solution delivered to the shipyard.

2 The model

We focus on the STA problem associated with an assembly hall subdivided into a number A of identical rectangular areas. Each area has width W, length L and height H. There

is a set of n blocks to be produced in the hall. Each block is viewed as a parallelepiped and is characterized by its geometric dimensions (width w_j, length l_j and height h_j, for $j = 1, 2, \ldots, n$), as well as by its production requirements such as processing time t_j (the number of days needed for its assembly), release date r_j (the date when the required parts are available for assembly), and due date d_j (the date when the block is to be delivered at the dry dock).

In order to be processed, each block should be assigned to an arbitrary area of the hall. The only restrictions are that the assembly of a block j cannot start before its release date r_j, the block must remain in the hall without interruption for t_j time units, and it must leave before its due date d_j. Moreover, each block is always positioned with its sides parallel to the walls of the hall; two blocks cannot overlap physically and cannot be placed on top of each other. As a result of the latter constraint, the height of the blocks and of the hall will not play any role in most of our discussion. (We briefly return in Sect. 6.1 to the practical consequences of the limited height of the hall and to additional constraints on individual blocks.)

Thus, the STA problem consists in orthogonally ordering the blocks into the rectangular areas, without overlap, and so as to respect the time constraints, with the objective to produce the largest possible number of building blocks.

We define six decision variables for each block $j = 1, 2, \ldots, n$:

- $b_j \in \{0, 1\}$, indicating whether block j will be produced in the hall ($b_j = 1$) or whether it will be subcontracted ($b_j = 0$),
- $a_j \in \{1, 2, \ldots, A\}$, indicating the area where block j will be produced,
- x_j and y_j, coordinates representing the position of the upper-left corner of block j in the selected area,
- $o_j \in \{0, 1\}$, indicating the orientation (either longitudinal or transversal) of block j in the selected area,
- s_j, the starting date for the assembly of block j.

A solution, that is to say an assignment of values to the above variables, is *feasible* if the *individual* and the *collective* constraints are met. We call individual constraints those which bear on one block only, regardless of the other blocks. The individual constraints can be modeled as follows:

(1) each block must fit within the width of an area: $x_j \geq 0$ and $x_j + [o_j w_j + (1 - o_j) l_j] \leq W$;
(2) each block must fit within the length of an area: $y_j \geq 0$ and $y_j + [o_j l_j + (1 - o_j) w_j] \leq L$;
(3) each block must fit in its time window: $s_j \geq r_j$ and $s_j + t_j \leq d_j$.

On the other hand, collective constraints deal with the interaction between the positions of different blocks. Unless we mention otherwise, the only collective constraint is that the blocks may not overlap.

2.1 Relation to the 3-Dimensional Bin Packing problem

The STA problem described above is closely related to the *3-Dimensional Bin Packing problem* (3D-BPP). Recall that in 3D-BPP, we are given a set of n parallelipipeds, each characterized by its width w_j, length l_j and height h_j ($j = 1, 2, \ldots, n$), as well as an unlimited number of identical three-dimensional bins with width W, length L and height H. The 3D-BP problem consists in orthogonally packing all the items in the minimum number of bins; see e.g. Martello et al. (2000).

We have already observed that the height of the assembly hall does not play any active role in the STA problem, since building blocks cannot be stacked upon each other. Time,

therefore, behaves as the third dimension of the model (similar models are mentioned by Dyckhoff 1990). We also note that a "2-dimensional" version of STA is described (without explicitly identifying the industrial environment) by Imahori et al. (2003, 2005). In these papers, the authors observe that each building block has nearly the same width as the assembly hall, so that the x-dimension does not play any active role. (One of the halls at *Aker Yards France* also is of this type.)

A major difference between 3D-BPP and the STA problem is that, in the former, items/blocks must fit in the bin height ($z_j \geq 0$ and $z_j + h_j \leq H$), whereas they must fit in their individual time window in the latter ($s_j \geq r_j$ and $s_j + t_j \leq d_j$). Moreover, the two problems deal with different objective functions (the so-called *knapsack loading* problem or *container packing* problem are variants of 3D-BPP whose objective function is more closely related to the objective of STA; see, e.g., Brunetta and Grégoire 2005, Dyckhoff 1990, or Martello et al. 2000). The distinction between minimizing the number of bins and maximizing the number of blocks will be mitigated, in a first step of our approach, by concentrating on the search for feasible solutions, rather than on the optimization version of the problem (see Sects. 3 and 4).

3D-BPP is a generalization of the well-known (1-Dimensional) Bin Packing problem and it is therefore strongly NP-hard; see e.g. Coffman et al. (1997, 1999), Dyckhoff (1990), Dyckhoff et al. (1997), Garey and Johnson (1979), Lodi et al. (2002), or Martello and Toth (1990) for classifications of bin packing problems and more information about their computational complexity.

Brunetta and Grégoire (2005), Faroe et al. (2003), Martello et al. (2000, 2007) are recent contributions which provide brief surveys of the literature on 3D-BPP. Since the problem is hard, most efficient approaches rely on *local search metaheuristics* for the solution of large-scale instances. (Martello et al. 2007 have tested exact algorithms for values of n up to $n = 50$.) In particular, Faroe et al. (2003) have proposed a *Guided Local Search* (GLS) heuristic for 3D-BPP. In their computational experiments, this approach appears to outperform the best available heuristics for 3D-BPP. It also offers a high degree of flexibility in its implementation, so that it can be easily adapted to variants of the problem involving different objective functions and/or additional constraints (such as the real-world side-constraints discussed in Sect. 6 below). Therefore, the algorithm that we have developed for STA explicitly builds on their work. We now proceed to describe it.

3 Finding feasible solutions

In this section, as in Faroe et al. (2003), we first concentrate on the problem of finding at least one feasible solution for the set of blocks initially given. Of course, if such a feasible solution is found, then no further optimization is needed. We will see in Sect. 4 what should be done in the opposite case.

3.1 General approach

Let X be any solution of the STA problem, that is, any assignment of values to the variables a_j, x_j, y_j, o_j and s_j for $j = 1, 2, \ldots, n$. (We implicitly assume that $b_j = 1$ for all j.) While trying to find a feasible schedule, our local search heuristic strictly enforces the individual block constraints defined in Sect. 2, meaning that X always satisfies the constraints (1)–(3). On the other hand, we do not enforce the collective constraints, but we measure the extent of their violation and these measures are summed in an auxiliary objective function to be

minimized. Without additional real-life collective constraints, the extent of the violations can be measured by the total "volume" (in "square meters × days") of pairwise overlaps between the blocks. Thus, if we denote by $overlap_{ij}(X)$ the volume of the overlap between blocks i and j, then the auxiliary objective function can be formulated as

$$f(X) = \sum_{1 \le i < j \le n} overlap_{ij}(X) \tag{1}$$

Starting from an arbitrary infeasible solution where blocks can overlap, searching for a feasible solution can be achieved by minimizing the function f, since an objective value of zero indicates that all the collective constraints are satisfied.

A typical local search procedure proceeds by moving from the current solution X to another solution X' in a neighborhood $\nu(X)$ whenever this move improves the value of the objective function. Slightly adapting the framework of Faroe et al. (2003) (who do not allow rotating the boxes), we define the neighborhood $\nu(X)$ as the set of all solutions that can be obtained by translating any single block along the coordinate axes or along the timeline, or by a move to the same (relative) position in another area of the hall, or by a ± 90 degree rotation of a block around one of its four corners.[1] A neighbor of X is therefore constructed by assigning a new value to exactly one of the variables x_j, y_j, s_j, a_j or o_j. It is clear that this definition allows to move from any solution to any other solution through a sequence of neighbors.

It is well-known that local search procedures may easily get stuck in a local minimum of poor quality. Several strategies have been proposed to avoid this shortcoming of simple descent algorithms, among which simulated annealing (see e.g. Aarts and Korst 1989), tabu search (see e.g. Glover 1990), variable neighborhood search (see Hansen and Mladenovič 2001), and many others.

Another difficulty with local search procedures is that the neighborhood of any given solution may be quite large (even if continuous, variables like x_j, y_j or s_j can be discretized for practical purposes) and therefore, exploring the neighborhood to find an improving move can be very costly in computing time.

To deal with the above issues, we rely on a the *Guided Local Search* (*GLS*) heuristic, and its accompanying neighborhood reduction scheme called *Fast Local Search* (*FLS*).

3.2 Guided local search

Guided Local Search has its roots in a neural network architecture developed by Wang and Tsang (1991), which is applicable to a class of problems known as *Constraint Satisfaction* problems. The current GLS framework, with the accompanying FLS scheme, has been first proposed by Voudouris (1997) and Voudouris and Tsang (1997, 1999).

Generally speaking, GLS augments the objective function f of a problem to include a set of penalty terms associated with "undesirable features" of a solution, and it considers the new function h, instead of the original one, for minimization by a local search procedure. The local search procedure is denoted *LocalOpt* in our description of *GLS* (see Algorithm 1). Local search is confined by the penalty terms and focuses attention on promising regions of the search space. Each time *LocalOpt* gets caught in a local minimum, penalties are modified and the *LocalOpt* search is called again to minimize the modified objective function. (This

[1] In the case of a rotation around a corner, moving to a neighbor also involves corresponding changes in x_j and in y_j. To simplify the explanation, this technical issue will be ignored in the sequel.

is akin to the use of diversification strategies in tabu search or in variable neighborhood search.)

This general scheme has been adapted to 3D-BPP by Faroe et al. (2003). In their procedure, the *features* of a solution X are the Boolean variables $I_{ij}(X) \in \{0, 1\}$, which indicate whether blocks i and j overlap ($I_{ij}(X) = 1$) or not ($I_{ij}(X) = 0$). The value of $overlap_{ij}(X)$ measures the impact of the corresponding feature on the solution X.

The number of times an "active" feature has been penalized is denoted by p_{ij}, which is initially zero. Thus, the augmented objective function takes the form

$$\begin{aligned} h(X) &= f(X) + \lambda \sum_{1 \le i < j \le n} p_{ij} I_{ij}(X) \\ &= \sum_{1 \le i < j \le n} overlap_{ij}(X) + \lambda \sum_{1 \le i < j \le n} p_{ij} I_{ij}(X) \end{aligned} \tag{2}$$

where λ is a parameter—the only one in this method—that has to be chosen experimentally (see Sect. 5.2).

Intuitively speaking, *GLS* attempts to penalize the features associated with a large overlap, but which have not been penalized very often in the past. More formally, we define a utility function $\mu_{ij}(X) = overlap_{ij}(X)/(1 + p_{ij})$ for each pair of blocks (i, j). At each iteration the procedure adds one unit to the penalty p_{ij} corresponding to the pair of blocks with maximum utility, then it calls $LocalOpt(X, (i, j))$ (see Algorithm 1). In a sense, the search procedure is forced to set a higher priority on these features. Since features with maximum utility keep changing all the time, this guiding principle prevents *GLS* from getting stuck in local minima. The algorithm is run for T time units at most.

Algorithm 1 $GLS(X_0,T)$; {X_0 is the initial solution, T is the time limit}

```
X := X0; {X is the current solution}
X* := X0; {X* is the best available solution}
t = 0; {t measures the run time}
p_ij = 0 for all pairs of blocks (i, j) {initialize the penalties}
repeat
   Select a pair (i, j) with maximum utility;
   p_ij := p_ij + 1; {Increase the penalty of pair (i, j)}
   X := LocalOpt(X, (i, j)) {Run FLS with new penalties}
   if h(X) ≤ h(X*) then
      X* := X;
   end if
until h(X) = 0 or (elapsed time t ≥ T)
return X*
```

3.3 Fast local search

In our implementation, the procedure *LocalOpt* mentioned in Algorithm 1 is a so-called *Fast Local Search* (FLS) procedure adapted from Voudouris and Tsang (1997) and Faroe et al. (2003). The main objective of FLS is to reduce the size of the neighborhoods explored in the local search phase, by an appropriate selection of moves that are likely to reduce the overlaps with maximum utility.

To describe FLS, consider any solution X and any variable m among the variables x_j, y_j, s_j, a_j, o_j with $j \in \{1, \ldots, n\}$. Informally, FLS selects at random a variable m within a list of active variables, as long as this list is not empty (active variables are those which are most likely to lead to an improvement of the current solution). Then, FLS searches within the domain of m for an improvement of the objective function. If no improvement is found, then the variable m becomes inactive and is removed from the list until the end of the current call to *LocalOpt*.

More formally, we define $\nu_m(X)$ as the set of all solutions which differ from X only by the value of variable m. The neighborhood $\nu(X)$ is thus divided into a number of smaller sub-neighborhoods:

$$\nu(X) = \bigcup_m \nu_m(X).$$

Each of the sub-neighborhoods $\nu_m(X)$ can be either *active* or *inactive*. Initially, only some sub-neighborhoods are active. (We will show at the end of this section how the selection process is designed to focus on the maximum utility overlaps.) FLS now continuously visits the active sub-neighborhoods in random order. If there exists a solution X_m within the sub-neighborhood $\nu_m(X)$ such that $h(X_m) < h(X)$, then X becomes X_m; otherwise we suppose that the selected sub-neighborhood will provide no more significant improvements at this step, and thus it becomes inactive. When there is no active sub-neighborhoods left, the FLS procedure is terminated and the best solution found is returned to *GLS*.

The size of the sub-neighborhoods related to the a_j and the o_j variables is relatively small, therefore FLS is set to test all the neighbors of these sets. On the other hand, using an enumerative method for testing the translations along the x, y and s-axes would be very time consuming, especially when areas and/or time windows are large. We may observe, however, that only certain coordinates of such neighborhoods need to be investigated. Indeed, as pointed out by Faroe et al. (2003), all $overlap_{ij}(x)$ functions (respectively $overlap_{ij}(y)$, $overlap_{ij}(s)$) are piecewise linear functions, and will for that reason always reach their minimum in one of their breakpoints or at the limits of their domains. (Thinking of the geometry of the 3D-BP problem, one can easily understand that a best packing arises either when the boxes touch each other along their faces, or when they touch the sides of the bins.) As a result, FLS only needs to compute the values of $f(x)$ (respectively, $f(y)$, $f(s)$) for x (respectively, y, s) at breakpoints or at extreme values. In fact, there are at most four breakpoints for each overlap function, and only the first and the last one are evaluated.

Additionally, since changes in the total overlap function only depend on the value of the selected variable m, most of the terms of this function are constant. Thus, when evaluating the value of $f(X)$ after a move, only the n $overlap_{ij}$ terms depending on m should be computed, and so the computing time for the evaluation of one solution turns out to be linear in n. In the description of FLS (see Algorithm 2), we denote by $h_{partial}(m)$ this "partial" augmented objective function used to compare the impact of fixing m at different values.

The efficiency of FLS directly depends on the number of active sub-neighborhoods. Now, remember that $LocalOpt(X, (i, j))$ is called by GLS after some penalty p_{ij} has been adjusted with the aim to escape local minima. Thus, active sub-neighborhoods should be those which allow moves on the penalized features associated with the overlap of blocks i and j. Accordingly, Faroe et al. (2003) propose to activate the moves on the two blocks i and j, as well as the moves on all blocks that overlap with block i and block j.

A schematic description of FLS is given by Algorithm 2.

Algorithm 2 *LocalOpt*$(X, (i, j))$; {X is the current solution; (i, j) is a pair of blocks }

```
ActiveList := List of the variables associated with the moves applicable to blocks i and j,
and to the blocks overlapping either i or j
while ActiveList ≠ ∅ do
  Pick a variable m in ActiveList
  Let m* be the current value of variable m
  PositionList := List of relevant values of m
  for k := 1 to |PositionList| do
    if h_partial(PositionList(k)) ≤ h_partial(m*) then
      m* := PositionList(k);
    end if
  end for
  Let X' be the solution obtained by setting m := m* in X
  if h(X') ≤ h(X) then
    X := X' {Execute the move}
  else
    Remove m from ActiveList {No improvement}
  end if
end while
return X;
```

4 Selecting the blocks

In the previous section, we described a *GLS* heuristic to find a feasible solution of the STA problem. If *GLS* works as expected, then it should return a space and time allocation with zero overlap (i.e., a feasible solution) when there is one. In general, however, no such feasible solution may exist for the set of blocks initially included in the instance, and we face the problem of selecting a maximum subset of blocks to be scheduled for assembly.

As mentioned in Sect. 2.1, this objective function differs from the usual objective function of 3D-BPP. In order to handle it, we rely on the following heuristic assumption:

(HA) *if GLS cannot find a feasible solution of* STA *within a predetermined amount of computing time* T, *then the heuristic assumption is that the instance is* (*probably*) *infeasible*.

As a consequence of this assumption, the search heuristic *GLS* can be used as a "black-box" to carry out feasibility tests. We use the notation $GLS(X, T)$ to indicate the output of procedure *GLS* when it is initialized with the (infeasible) solution X and executed for T time units.

Several procedures have been developed and tested based on this concept. A simple "descent method" is to initialize *GLS* with a randomly generated solution X_0 that includes the entire set of blocks (i.e., set $b_j = 1$ for all $j = 1, 2, \ldots, n$). After a search of T time units, the algorithm is stopped and returns $X_1 = GLS(X_0, T)$, the best solution found (in terms of overlap). Then, one of the blocks with the largest overlap is removed from the solution X_1, and the heuristic *GLS* is restarted from this solution. The entire procedure ends when a solution X_k with zero overlap is found; see Algorithm 3.

A more efficient variant of this procedure, called $MaxBlocks(X, T, Tmax)$, allows adding as well as removing blocks from the current set. Thus, assume that, at any iteration of the procedure, X is a solution (feasible or not) involving some subset of blocks. If the solution

Algorithm 3 *BlockDescent*(X_0, T); {X_0 is the initial solution, T is the time limit in every application of *GLS*}

$X^* := X_0$; $b_j := 1$ for $j = 1, 2, \ldots, n$;
repeat
  Let X be the solution obtained by setting $b_j := 0$ in X^* for a random block j among those such that $\sum_i overlap_{ij}(X^*)$ is maximum;
  $X^* := GLS(X, T)$;
until $f(X^*) = 0$
return X^*;

$GLS(X, T)$ returned by *GLS* is feasible, then this solution is a candidate to be the final optimal solution. So, we record it if it is better than the best incumbent solution X^*, and we try to include an additional block in the set. On the other hand, if $GLS(X, T)$ is not feasible, then a fast post-processing step is performed to produce a feasible solution X': this is achieved by simply removing blocks in a greedy fashion until all overlaps are cancelled. The solution X' is recorded if it is better than the incumbent X^*; then, we remove an overlapping block from $GLS(X, T)$ and the process is repeated. (Depending on the infeasibility level of the solution, we may even want to remove more than one block at a time.) The procedure is stopped after a predetermined amount of computing time *Tmax*, or by any more sophisticated stopping criterion, and it returns the feasible solution X^* involving the largest collection of blocks (i.e., the largest number of variables b_j equal to 1). Note that, thanks to the post-processing phase, $MaxBlocks(X, T, Tmax)$ always generates a feasible solution. A more precise description is given in Algorithm 4. Here, we denote by $|X|$ the number of blocks j such that $b_j = 1$ in a solution X.

Algorithm 4 *MaxBlocks*$(X_0, T, Tmax)$; {X_0 is the initial solution, T is the time limit in every application of *GLS*, *Tmax* is the global time limit for *MaxBlocks*}

$X^* := \emptyset$; $X := X_0$;
repeat
  $X := GLS(X, T)$;
  if $f(X) = 0$ (the current solution is feasible) **then**
    if $|X| > |X^*|$ **then**
      $X^* := X$
    end if
    In X, set $b_j := 1$ for a random block j such that $b_j = 0$; {Add a random block}
  else
    $X' := PostProcessing(X)$; {Generate a feasible solution}
    if $|X'| > |X^*|$ **then**
      $X^* := X'$
    end if
    In X, set $b_j := 0$ for a random block j such that $\sum_i I_{ij} \geq 1$; {Remove a random overlapping block}
  end if
until Stopping criterion: elapsed time $\geq$ *Tmax*
return X^*;

One of our aims in this study, however, was to remain as close as possible to the industrial reality and to the working methods of the shipyard under study. From this point of view, the approaches outlined above suffer from one major drawback: they are likely to have aversion for the largest blocks and to reject such blocks before any others, since they are hardest to allocate. ("Large" may mean here: either large surface $l_j \times w_j$, or large processing requirements t_j.) Scheduling algorithms are known to face similar difficulties when long tasks have to be placed.

In the real-life situation, when the entire set of blocks cannot be produced, the operator in charge of scheduling can either move specific blocks to other assembly halls, or subcontract them to external workshops, or change some of the system parameters (e.g., increase the workforce to reduce processing times, postpone due dates, etc.). However, the shipyard did not provide us with formalized information which could describe all the relevant aspects of these choices, nor with an appropriate weighting scheme to evaluate the preferences among blocks.

For this reason, the software that we have developed allows the operator to change manually the collection of blocks to be allocated. In practice, starting from any solution X_0 (feasible or not), iterative executions of the form $X_{k+1} = GLS(X'_k)$ can be performed at will by the operator, where X'_k is a solution obtained by deliberate modifications of a previous solution X_k. (In particular, by switching any variable b_j from 0 to 1 or from 1 to 0.) This indicates to the operator if a particular set of blocks is feasible or not, and provides the corresponding allocation when there is one.

Finally, it should be observed that, in practice, real instances are likely to be "almost feasible" since the collection of blocks which make up such instances are selected in preliminary planning phases which take into account, at least approximately, the actual production capacity of the assembly halls. Therefore, simple optimization procedures combined with manual updates can quickly lead to good solutions. By offering access to the three procedures mentioned above, the industrial application gives the end-user a broad control of the data and of the computed results, so that he can easily evaluate various situations and take the appropriate decision based on several trials.

5 Computational experiments

This section presents the results of computational experiments with the different procedures described above. The objective of these experiments was to establish guidelines for calibrating the parameters of the procedures, to evaluate their performance on various benchmark instances, and to provide a comparison with the performance of alternative approaches based on a generic solution package (for reasons explained below, we have used a constraint programming package, namely ILOG (2007)). The algorithms, written in C++, have been run on a Pentium 3 GHz with 2 Gb RAM.

5.1 Data and test instances

Let us first provide some additional information regarding the types of problem instances encountered at the shipyard.

The amount of work required by each block depends on the complexity of the block, which typically depends on its position in the ship, on the type of ship under construction, and on many other factors. The processing time is thus extremely variable for different blocks, ranging from 2 to 40 days.

A typical assembly hall consists of four working areas of approximately 70×25 meters. The crane bridge in this hall is able to carry blocks weighing up to 180 tons; blocks of such weights typically have dimensions of the same order as the width of the ship under construction (e.g., 25–40 meters). About 100 blocks are scheduled at once, for a total time horizon of 6 to 9 months.

Several test instances have been generated based on the features of this assembly hall. Each instance contains 100 blocks to be allocated to one of the four areas; the dimensions of each block (spatial and temporal) are compatible with the dimensions of the areas and with the time windows. Thus, the feasibility or infeasibility of each instance is only due to the interactions among the blocks, i.e., to the "collective constraints".

The instances are of 6 different types, labeled by letters from A to F: types A to D correspond to "realistic" instances, type E to highly structured instances, and type F to random instances. The realistic instances are derived from industrial data and are meant to exhibit the main features of real data (shapes of the blocks, processing times, and time windows characteristics). More specifically:

A: Instances A0–A5 are based on real data.

B: Instances B1–B5 are derived from A1 by multiplying the length of each block in A1 by a factor (ranging from 1.06 to 1.10) which increases with the label of the instance. Thus, the blocks in B1 are longer than the blocks in A1, the blocks in B2 are longer than those in B1, and so on.

C: Instances C1–C5 are similarly derived from A1 by multiplying the width of each block in A1 by a factor (ranging from 1.03 to 1.05) which increases with the label of the instance.

D: The multiplicative factors applied when generating instances of type B or C have more impact on large blocks than on small ones. In order to counter this effect, we build a set of instances D1–D5 where the multiplier is applied only to the smaller blocks, thus generating more homogeneous block sizes than in B and C. Instances D1–D3 are increasingly homogeneous. Instance D4 is meant to be difficult: the length (resp., the width) of each block is exactly half the length (resp., the width) of an assembly area, so that two blocks can only fit side by side if they are very accurately adjusted. In instance D5, the length (resp., the width) of each block is exactly equal to the length (resp., the width) of an area, and the time windows are such that it is very clearly impossible to allocate all the blocks.

E: In the instances E1–E5, all the blocks have the same dimensions and durations, and these values increase from E1 to E5. The time window is also identical for all the blocks. Instances E1 and E2 are feasible, and instances E3–E5 are infeasible. In particular, E5 is the instance where all the blocks have the dimensions of an assembly area and duration equal to the planning horizon length, meaning that there are 25 times too many blocks with respect to the availability of the workshop.

F: Instances F0–F5 are randomly generated. The spatial dimensions of the blocks are normally distributed so that on average, each block fills $\frac{1}{16}$ of an assembly area. The durations t_j and the release dates r_j are uniformly distributed, so as to obtain a nearly constant load of the areas. The length of the time windows decreases with the label of the instances, which are therefore increasingly likely to be infeasible.

In total, this yields 32 instances labeled A0–A5, B1–B5, C1–C5, D1–D5, E1–E5 and F0–F5. Some additional hard instances will be introduced in Sect. 5.5.

5.2 The λ parameter

The first experiments were designed to adjust the value of the λ parameter which appears in the definition of the augmented objective function—see equation (2)—and which is the main parameter of the *GLS* procedure (together with its total running time). The value of λ determines to what degree a penalty modifies the augmented objective value and drives the local search out of a local minimum. A large value of λ is supposed to make the search more aggressive, to avoid solutions with penalized features and to favor large jumps in the solution space with limited attention for the overlap term $f(X)$ in the augmented objective function. Small values of λ, on the other hand, may require heavier penalties p_{ij} to escape a local minimum but should result in a more intensive exploration of the neighborhood of the current solution and to a search strategy that is more sensitive to the gradient of $f(X)$. However, small λ values might prevent a broad exploration of the solution space.

We have tested the Guided Local Search algorithm with different values of λ in a broad range from 1 to 9000, and with a high limit (1200 sec.) on its total running time. The results obtained on a representative sample of feasible instances are displayed in Table 1 below. (Similar results were obtained for other instances.) As the computing time of the heuristic is random and may vary from one run to the next on any specific instance, the table displays the mean values of the computing time for 10 executions on each instance, as well as the percentage of the number of trials for which GLS was able to find a solution within 1200 seconds when this percentage is smaller than 100%. (In the latter case, the average computing time is reported for the solved instances only.)

We can see in Table 1 that, for small values of λ (say, λ smaller than 1000), *GLS* does not always reach a feasible solution. On the other hand, the performance of the algorithm does not seem to depend significantly on the choice of λ in the range from 1000 to 9000.

We also performed some experiments where the value of λ was dynamically adapted to the value of the objective functions. But this self-adjusting framework did not yield better results than those obtained with a fixed value of λ.

In the following computational experiments, the value $\lambda = 5000$ was used as default value, since this value led to good results for most of the test instances. In the industrial application, however, the value of λ is a user-parameter which can be changed if it does not provide the expected results, and smaller values of λ are frequently used (see Sect. 6.2).

5.3 Comparison with a constraint programming approach

As explained in Sect. 4, the heart of our approach to STA is the *GLS* algorithm which tests the feasibility of any given set of boxes. A similar approach is typically used by *constraint programming* algorithms, which also rely on iterated solutions of feasibility subproblems to solve optimization problems; see e.g. Focacci et al. (2003), Jussien and Lhomme (2002),

Table 1 Mean execution time of GLS (in seconds) and percentage of solved instances

λ	1	10	50	200	1000	3000	5000	7000	9000
A1	*(0%)	153 (30%)	113 (80%)	69.2	58.1	30.4	35.8	33.5	36.3
A2	106 (70%)	48.6	27.9	15.8	12.5	13.4	15.3	13	10.8
A3	61.9 (90%)	23.7	20	5.3	4	6.8	3.2	5.9	7.5
A4	72.8	20.9	10.1	2.4	1	2.5	4.4	6.6	5.9
B4	*(0%)	519 (10%)	450 (20%)	533 (70%)	468.2	734.3	731.7	526.8	616.7

Van Hentenryck and Michel (2005), Wang and Tsang (1991), etc. Therefore, we decided to compare the performance of our *GLS* algorithm with that of widely available commercial software tools for constraint programming, namely the *CP-Optimizer* (*CPO*) package and the *Solver* package from ILOG (2007). Of course, this comparison is not entirely "fair" to the extent that the GLS algorithm has been specifically tailored to handle STA, whereas the ILOG packages are generic optimization tools. But the virtue of the comparison is precisely to provide a better understanding of the benefits obtained by developping a specialized ad hoc algorithm for the practical problem faced by the shipyard, rather than relying on generic "off-the-shelf" solutions like those provided by ILOG.

Our constraint programming model for STA contains five variables for each block (area, orientation, (x, y)-coordinates, and starting time), as well as the individual constraints and the no-overlap constraints described in Sect. 2. This basic model, however, is too primitive to be efficiently solved. Therefore, we strengthen it with several additional constraints. First, we include a global packing constraint available in *CPO*, in order to ensure that the sum of the three-dimensional encumbrances (surface $\times$ duration) of the blocks placed in each area does not exceed the three-dimensional volume of the area (available surface $\times$ length of planning horizon).

Furthermore, we add some constraints in order to reduce the large number of symmetric solutions that are inherent to the problem. Note that symmetries arising from 180° rotations of the blocks are automatically ruled out by our definition of variables, since the orientation of each block is simply defined to be either vertical or horizontal (no right-left nor bottom-up symmetry). A second type of symmetry issue is related to the placement of the blocks in an area (vertical symmetry, horizontal symmetry, and 180° rotation of each area); it is reduced by imposing, for each area, that one of the blocks has its upper-left corner placed in the upper-left quadrant of the area. A last symmetry issue derives from the fact that all working areas are identical. This problem occurs in particular when several areas are empty; then, it is indifferent to place the next block in either one of these empty areas. This issue is taken into account by restricting the allowed positions of the first blocks to be placed.

We ran *CP Optimizer* with different sets of parameters on the test instances. Among the search strategies available in *CPO*, only the so-called "restart" strategy was able to deliver significant results. We also used the *Solver* software on the same model; *Solver* allows to implement dedicated search strategies instead of the predefined strategies present in *CPO*. We have experimented with different search strategies which, however, did not prove more efficient than the standard strategies used by *CPO*. Therefore, we do not dwell on the details here and we simply focus on the comparison between *CPO* and *GLS*.

We have analyzed both the quality of the results and the computing times of *GLS* and *CPO*. In our experiments, we concentrated mostly on the feasibility version of STA. Note that for any given instance, the *GLS* heuristic can only reach the conclusion that the instance is feasible or that it is unable to find a solution within the allocated time. On the other hand, the *CPO* software can either find a solution, or prove that the problem is infeasible, or reach the time limit without any conclusion.

Table 2 displays the results obtained on the set of benchmark instances. The computing time of the *GLS* heuristic is random, therefore the times presented in Table 2 for *GLS* are averages over 5 executions. On the other hand, the execution time of *CPO* for a given instance is essentially constant. For both algorithms, a limit of 1200 seconds has been set on the computing time. An objective value of "1" indicates that a feasible solution has been found (in each run of the algorithm), and a "0" means that no solution has (ever) been found within the time limit. For *CPO*, a value "0*" indicates that *CPO* has been able to *prove* that the corresponding instance is infeasible. A computing time larger than 1200 indicates that the time limit has been reached.

Table 2 Comparison *GLS* vs. *CPO*

Instance	Objective		Time	
	GLS	*CPO*	*GLS*	*CPO*
A0	1	0	50	1201
A1	1	1	25	561
A2	1	0	11	1201
A3	1	0	7	1201
A4	1	1	4	119
A5	0	0	1201	1201
B1	1	1	43	568
B2	1	1	39	244
B3	1	0	88	1201
B4	1	0	513	1201
B5	0	0	1201	1201
C1	1	1	1	414
C2	1	0	53	1201
C3	1	0	59	1201
C4	1	0	49	1201
C5	1	0	79	1201
D1	1	0	29	1201
D2	0	0	1201	1201
D3	0	0	1201	1201
D4	0	0	1201	1201
D5	0	0*	1201	1
E1	1	0	1	1201
E2	1	0	284	1201
E3	0	0	1201	1201
E4	0	0	1201	1201
E5	0	0*	1201	1
F0	1	1	1	18
F1	1	1	1	13
F2	1	1	1	11
F3	1	1	1	16
F4	1	0	54	1201
F5	0	0	1201	1201

Clearly, more instances are solved by Guided Local Search than by Constraint Programming within a given time limit. In fact, it never happens that *CPO* finds a solution but *GLS* does not find one. Moreover, whenever both algorithms find a feasible solution, the mean computing time required by *GLS* is always (much) smaller than the time required by *CPO*.

When no feasible solution can be found, either both algorithms reach the time limit without conclusion, or *CPO* proves that the instance is infeasible. This last case is interesting but unfortunately, it occurs very rarely (2 instances out of 32 in our experiments), and only for instances which are "severely" infeasible (recall the description of the instances in Sect. 5.1).

In conclusion, the *GLS* algorithm clearly outperforms the *CPO* algorithm on our set of benchmark instances. Of course, we cannot exclude that a more sophisticated CP model and/or more advanced settings of the *CPO* software would yield better results. But at the very least, the comparison seems to justify the development of our specialized algorithm for the industrial application. These conclusions are actually fully coherent with the observations in recent papers by Martello et al. (2007) or Pisinger and Sigurd (2007), where the authors develop exact algorithms for different variants of bin-packing problems and where CP approaches prove useful for the solution of specific one-bin subproblems, but only in combination with other advanced integer programming techniques.

5.4 Performance of *GLS* as a function of its running time

In the previous section, we have shown that *GLS* is able to solve many feasible benchmark instances within $T = 1200$ seconds. In fact, it is interesting to note that, for the number of blocks considered here, the running time of *GLS* on feasible realistic instances (A0–A5) is actually quite short, in the range of 10 to 50 seconds. This is probably due to the fact, already mentioned above, that real instances are likely to be reasonably easy as the assembly hall is not excessively loaded.

To validate these observations, Table 3 shows the ability of *GLS* to solve a feasible instance within a given time T, when T is relatively short (which must be the case at the shipyard, where the algorithm is meant to be used frequently; see also Sect. 5.5). For each instance, we report the percentage of executions (out of 10 trials) that successfully found a feasible solution. We can see that *GLS* performs quite well for most instances when $T = 40$ seconds or $T = 120$ seconds, except for B4 which is clearly a much harder instance (see Sect. 5.1).

5.5 Optimization procedures

The procedures $BlockDescent(X, T)$ and $MaxBlocks(X, T, Tmax)$ described in Sect. 4 aim at maximizing the total number of blocks produced. They iteratively generate several sets of blocks X, check whether each set is feasible (using the procedure $GLS(X, T)$), modify

Table 3 Percentage of instances solved by *GLS* as a function of its running time

	$T = 120$	$T = 40$	$T = 20$	$T = 10$	$T = 5$
A0	100%	90%	40%	0%	10%
A1	100%	100%	90%	30%	20%
A2	100%	100%	100%	100%	30%
A3	100%	100%	100%	100%	70%
A4	100%	10%	0%	0%	0%
B1	100%	70%	50%	10%	0%
B2	90%	70%	10%	20%	0%
B3	90%	20%	20%	0%	0%
B4	20%	0%	0%	0%	0%
D1	100%	50%	20%	20%	0%
Mean values	90%	55%	39%	25%	12%

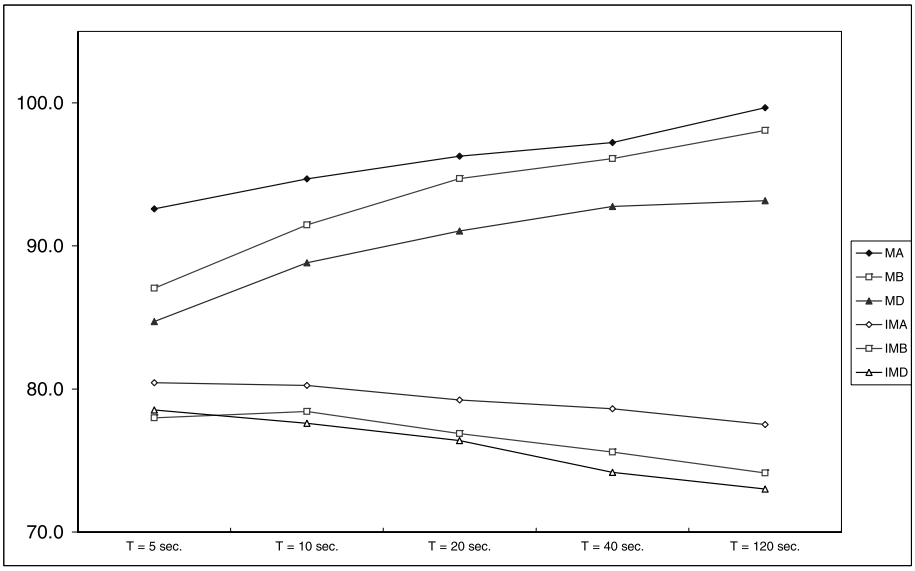

Fig. 1 Performance of $MaxBlocks(X, T, Tmax)$ for different values of T

it accordingly, and eventually return the best solution found in the process. Of course, we expect a larger time parameter T to provide more certainty about the feasibility or infeasibility of a current solution X (cf. hypothesis (HA) in Sect. 4, and the observations reported in Sect. 5.4). Thus, it is intuitively better to set a rather large value for this parameter. On the other hand, when the iterative calls $GLS(X, T)$ are long, a smaller number of solutions are analyzed within the same total computing time $Tmax$, and this reduces the likelihood to reach the best solutions. Therefore, we have tested the trade-off between these two antagonistic impacts for the procedure $MaxBlocks(X, T, Tmax)$, which is the more efficient of the two optimization procedures.

Figure 1 displays the results obtained when $MaxBlocks(X, T, Tmax)$ is executed with a total time limit $Tmax = 120$ seconds and with various values of T in a range from 5 to 120 seconds. Note that, as a result of these parameter settings, $GLS(X, T)$ can be called from 1 to more than 120 times in any given experiment. For each value of T, the vertical axis shows the average number of blocks in the best solution found by $MaxBlocks(X, T, Tmax)$ for several classes of instances.

More precisely, the graph labeled MA (resp., MB, MD) shows the mean value of the results over the instances A0–A5 (resp., B1–B5, D1–D4). Each individual instance was solved 10 times for each value of T. The complexity of the individual instances has been further increased by multiplying the width of each block by a constant factor 1.5, thus giving rise to more difficult instances IA0–IA5, IB1–IB5 and ID1–ID4, respectively. The mean results obtained for the latter instances are also displayed in Fig. 1: see graphs IMA, IMB, IMD, respectively.

We can observe that two behaviors appear depending on the relative complexity of the instances. Instances A to D are rather low complexity instances for the optimization problem, and each of these instances includes many feasible subsets of blocks; so, the main challenge in this context is essentially to identify correctly these feasible subsets and, as we already observed in Sect. 5.4, larger values of T produce the best performance. For harder

instances (IA, IB, ID), however, the performance of the algorithm slightly increases when T decreases. In this case, each local search phase $GLS(X, T)$ is very short, but its repetitive execution allows to reach good solutions.

The previous observations suggest that running *MaxBlocks* for a longer time should yield even better results. Accordingly, some experiments have also been conducted with a very large time limit ($Tmax = 6000$ seconds) and led to an improvement of the best solution by about 4% (i.e., 4 blocks) on some instances. It should be stressed, however, that such long running times are excessive in the industrial context, where the algorithms are usually allowed to run for one or two minutes only.

To conclude, we can say that $GLS(X, T)$ displays a good performance on feasible instances when T is around 120 seconds. On the other hand, for infeasible instances, the optimization procedure $MaxBlocks(X, T, Tmax)$ allows *GLS* to escape local minima. In this case, a trade-off has to be found when setting the parameter T. In practice, a total computing time of $Tmax = 120$ seconds with $T = 20$ seconds appears to offer a good compromise.

6 Industrial issues

The algorithms described in this paper have been developed for three different workshops at *Aker Yards France*. At the time of this writing, they have been in daily use for several months at the shipyards. As compared to the "academic" and rather abstract description of the problem that we gave in previous sections, tailoring the algorithms to their industrial environment required several adaptations and raised new questions that we now proceed to discuss.

6.1 Additional constraints

Various side-constraints have to be considered in order to increase the practical relevance of the STA model. Fortunately, the *GLS* framework proved flexible enough to incorporate most of these constraints without too much additional effort.

For example, in practice, it may be necessary to restrict or to impose the position of certain blocks (e.g., because these blocks are already in process when the planning process is launched, or because some required handling or production equipment is only available in a particular area, etc.). Such individual constraints on blocks are easily handled by the *GLS* algorithm: forbidden positions and infeasible neighbors are simply not generated during the search. Thus, in practice, the end-user may fix the value or reduce the domain of any variable when using the software (including b_j-variables; see also Sect. 4).

More complex collective constraints also appeared in the real-life situation. In particular, for the assembly hall described in the previous section, each working area has a single door, and the crane bridge can only carry the blocks up to a certain height C. As a result, it may happen that a tall block obstructs the door or stands otherwise in the way, and some finished blocks may not be deliverable in time because there is no feasible passageway to carry them out of the hall.

Here again, the *GLS* approach proved "generic" enough to deal with this issue. For each generated solution X, we added to the objective function $h(X)$ a new penalty term which accounts for exit difficulties:

$$\begin{aligned} g(X) &= h(X) + e(X) \\ &= \sum_{i<j} overlap_{ij}(X) + \lambda \sum_{i<j} p_{ij} I_{ij}(X) + \sum_{i,j} exit_{ij}(X), \end{aligned}$$

where $exit_{ij}(X)$ measures the overlap between block i and the "exit path" for block j. The exit path for j is restricted by security constraints which impose to use a straight path, and thus it is determined by:

- the longitudinal interval $[x_j, x_j + o_j w_j + (1 - o_j) l_j]$;
- the transversal interval $[0, y_j + (1 - o_j) w_j + o_j l_j]$, as the doors are at position $y = 0$;
- the vertical interval $[C - h_j, C]$, since each block can be carried up to the height of the crane bridge;
- the completion date $s_j + t_j$ of block j;
- the area a_j where block j is produced.

Note that the value of the *exit* terms could somehow be scaled in relation to the $h(X)$ values, but this did not appear to be useful in our procedure, as the new penalty terms proved sufficient to drive the objective function to zero.

Additional collective constraints arise when a family of related blocks have to be produced for a ship. For example, all the blocks that include the emergency boats require similar production equipments, and it is convenient to allocate them to a same zone of the halls. In a similar way, two blocks that are adjacent in the ship structure may need to be produced next to each other in the assembly hall, so as to allow a fine positioning of connecting elements like members or piping tracks. An easy way to cope with the latter requirement is to define a super-block that includes the two (or more) adjacent blocks and to replace the individual blocks by this super-block in the data of the problem. For the first situation, however, this method is too restrictive, and we prefered instead to define a "distance" constraint that limits the relative distance between two blocks.

Other collective constraints could certainly be included in the model by taking full advantage of the flexibility of the *GLS* framework.

6.2 Robustness

The optimization procedures *BlockDescent* or *MaxBlocks* described in Sect. 4, just like the guided local search procedure *GLS*, always start from an initial solution X_0. A drawback of this approach is that the structure of X_0 can confine *GLS* to an area of the solution space that can be difficult to escape (especially for small values of λ), and therefore, the search process may not reach the very best solution.

However, in a dynamic industrial setting, this apparent drawback turns out to be an advantage. Indeed, it may be very costly, or practically impossible for the company to frequently readjust the schedules and the allocation of blocks to the halls. By generating new solutions from previous ones, the *GLS* procedure actually ensures that the structure of previous solutions can be preserved when the production plans are updated. (This is to be contrasted with various methods proposed for rectangle packing problems arising in VLSI design, which typically rely on construction strategies and for which a slight modification of the data may lead to major perturbations of the solution.) As a consequence, it may prove rewarding to run *GLS* with a relatively small value of λ in the industrial context.

6.3 Dispatching and subcontracting blocks

The aim of the methods described in this paper was to maximize the number of blocks produced in a particular assembly hall. Compared to the situation faced by the shipyard, this is actually a rather gross simplification. The shipyard is indeed composed of two distinct assembly halls dedicated to the production of building blocks. Additionally, blocks (or parts

of blocks) can be processed at a *panel line*, another assembly hall involving completely different processes. Finally, blocks can also be subcontracted to outside firms, but this is usually more costly and should be avoided as much as possible.

Our algorithms have been easily adapted, and are currently used, for several halls consisting of rectangular areas of heterogenous sizes. These halls are viewed as distinct production units and are handled independently of each other. An interesting topic for additional research, however, would be to *simultaneously* deal with the global issue of dispatching the blocks among the assembly halls, as well as with the subcontracting decisions. This would require, in particular, to handle cost data (for handling, subcontracting, ...) which were not available to us in the framework of the project reported here.

7 Conclusion

In this paper, we have presented a real-world space and time allocation problem arising in a large shipyard, and we have modeled it as a 3-dimensional bin packing problem. We have demonstrated the practical efficiency and usefulness in this industrial context of the GLS approach proposed by Faroe et al. (2003) for the 3D-BPP. This generic approach allows incorporating various real-life constraints and led to the successful implementation of a flexible and robust application which is now in daily use at the shipyard.

Acknowledgements We wish to thank *Aker Yards France* for their support in this work and for numerous helpful comments regarding the shipbuilding process. We also thank several ANAST colleagues and three anonymous referees for their constructive remarks on earlier versions of the paper.

References

Aarts, E., & Korst, J. (1989). *Simulated annealing and Boltzmann machines—a stochastic approach to combinatorial optimization and neural computing*. New York: Wiley.

Brunetta, L., & Grégoire, Ph. (2005). A general purpose algorithm for three-dimensional packing. *INFORMS Journal on Computing*, *17*(3), 328–338.

Coffman, E. G., Garey, M. R., & Johnson, D. S. (1997). Approximation algorithms for bin packing: A survey. In D. S. Hochbaum (Ed.), *Approximation algorithms for NP-hard problems*. Boston: PWS Publishing Company.

Coffman, E. G., Galambos, G., Martello, S., & Vigo, D. (1999). Packing approximation algorithms: Combinatorial analysis. In D.-Z. Du & P. M. Pardalos (Eds.), *Handbook of combinatorial optimization*. Dordrecht: Kluwer Academic.

Dyckhoff, H. (1990). A typology of cutting and packing problems. *European Journal of Operational Research*, *44*, 145–159.

Dyckhoff, H., Scheithauer, G., & Terno, J. (1997). Cutting and packing. In M. Dell'Amico, F. Maffioli, & S. Martello (Eds.), *Annotated bibliographies in combinatorial optimization*. New York: Wiley.

Faroe, O., Pisinger, D., & Zachariasen, M. (2003). Guided local search for the three-dimensional bin-packing problem. *INFORMS Journal on Computing*, *15*(3), 267–283.

Focacci, F., Laburthe, F., & Lodi, A. (2003). Local search and constraint programming. In F. Glover & G. Kochenberger (Eds.), *Handbook of metaheuristics*. *International Series in Operations Research & Management Science*. Dordrecht: Kluwer Academic.

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: a guide to the theory of NP-completeness*. New York: Freeman.

Glover, F. (1990). Tabu search: a tutorial. *Interfaces*, *20*(4), 74–94.

Hansen, P., & Mladenovič, N. (2001). Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, *130*, 449–467.

ILOG (2007). *CP Optimizer user's manual and reference manual*. ILOG, Paris.

Imahori, S., Yagiura, M., & Ibaraki, T. (2003). Local search algorithms for the rectangle packing problem with general spatial costs. *Mathematical Programming*, *97*, 543–569.

Imahori, S., Yagiura, M., & Ibaraki, T. (2005). Improved local search algorithms for the rectangle packing problem with general spatial costs. *European Journal of Operational Research*, *167*, 48–67.

Jussien, N., & Lhomme, O. (2002). Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, *139*, 21–45.

Lodi, A., Martello, S., & Monaci, M. (2002). Two-dimensional packing problem: a survey. *European Journal of Operational Research*, *141*, 241–252.

Martello, S., Pisinger, D., & Vigo, D. (2000). The three dimensional bin packing problem. *Operations Research*, *48*(2), 256–267.

Martello, S., Pisinger, D., Vigo, D., Den Boef, E., & Korst, J. (2007). Algorithm 864: General and robot-packable variants of the three-dimensional bin packing problem. *ACM Transactions on Mathematical Software*, *33*(1), 1–12.

Martello, S., & Toth, P. (1990). *Knapsack problems—algorithms and computer implementations*. New York: Wiley.

Pisinger, D., & Sigurd, M. (2007). Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem. *INFORMS Journal on Computing*, *19*(1), 36–51.

Van Hentenryck, P., & Michel, L. (2005). *Constraint-based local search*. Cambridge, MA: The MIT Press.

Voudouris, C. (1997). *Guided local search for combinatorial optimization problems*. Ph.D. Thesis, Department of Computer Science, University of Essex, Colchester, United Kingdom.

Voudouris, C., & Tsang, E. (1997). Fast local search and guided local search and their application to British Telecom's workforce scheduling problem. *Operations Research Letters*, *20*, 119–127.

Voudouris, C., & Tsang, E. (1999). Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, *113*, 469–499.

Wang, C. J., & Tsang, E. (1991). Solving constraint satisfaction problems using neural-networks. In *Proceedings of IEE second international conference on artificial neural networks*, pp. 295–299.