ABSTRACT
In this paper we exemplify the generic application of various tabu search methods to different sequencing problems from the field of production planning. Our approach includes the emphasis on implementing neighborhood search based metaheuristics, especially tabu search methods, in a generic and adaptable way, which facilitates an efficient reuse of these software components. We briefly describe a corresponding framework for heuristic search including the application to different \( \mathcal{NP} \)-hard manufacturing problems.

1. INTRODUCTION
In practice there is a great variety of discrete optimization problems. Especially in the field of production planning, there is a large number of different problem types, such as assignment type problems (e.g., assign production tasks to resources) or sequencing problems (e.g., schedule the production of a set of items, or determine a tour sequence). Problems of this kind generally occur in many different situations, however, with respective variations. Since most of these problems are \( \mathcal{NP} \)-hard, heuristics are the primary way to tackle them.

Metaheuristics like tabu search have been successfully applied to a great variety of such problems [1, 2]. In order to apply respective methods to a new type of problem, the corresponding algorithms have to be "materialized" as executable implementations—a non-trivial task if this is done every time starting from scratch. Therefore, we also have to deal with the issue of efficiently building such implementations to bridge the gap into practice, instead of just investigating and solving simplified problems by some ad hoc approach.

Our approach includes the emphasis on building software that implements robust heuristics to solve real world problems. Prerequisites for this are, e.g., mechanisms to easily change some problem characteristics or the pursued objective, which enable an efficient adaptation for problem instances from practice.

Accordingly, we have defined and implemented an object-oriented framework for heuristic search with reusable and adaptable software components.

In this paper we survey the current status of our efforts with the heuristic search framework. For this purpose we consider the application of various tabu search methods from the framework to different \( \mathcal{NP} \)-hard sequencing problems from the field of production planning (pattern sequencing and continuous flow-shop scheduling). We exploit different common neighborhood structures, such as those defined by shift moves, swap moves, and inversion of partial sequences (2-exchange moves). Pattern sequencing problems generally consist of finding a permutation of predetermined production patterns (groupings of some elementary order types) with respect to different objectives. These objectives may represent, e.g., handling costs or stock capacity restrictions. Flow-shop scheduling problems circumscribe an important class of sequencing problems in the field of production planning. The problem considered here is to find a permutation of jobs to be processed sequentially on a number of machines under the restriction that the processing of each job has to be continuous with respect to the objective of minimizing the total processing time (flow time).

2. TABU SEARCH
Building a non-trivial software system requires a solid understanding of the respective domain. Thus, we first define the primary concepts of neighborhood search based metaheuristics (with a focus on tabu search). There are different types of discrete optimization problems \( P \), each with different problem instances \( p \in P \). For every problem instance \( p \), there may exist one or more solution spaces \( S_p \) with solutions \( s \in S_p \). If unambiguous, we may neglect the index \( p \). Solutions are evaluated by an evaluation function \( f : S \to \Gamma \). A fully ordered set \( \Gamma \) represents a simple preference function of a decision maker. The objective is to find a solution \( s \in S \) with minimal evaluation \( f(s) \), or an evaluation as low as possible. (Of course, maximization problems may easily
To maintain information about solutions, there may be one or more solution information functions \( I : S \to \mathcal{F} \), which are called exact, if \( I \) is injective, and approximate otherwise. With this information, one may store a search history (trajectory). For each solution space \( S \), there are one or more neighborhood structures \( N \), that define for each solution \( s \in S \), a linear ordered set of neighbors \( N(s) = \{ n_1(s), \ldots, n_l(s) \} \). To each neighbor \( n(s) \in N(s) \), there is a corresponding move \( \mu(s, n(s)) \) that captures the transitional information from \( s \) to \( n(s) \). Each move \( \mu \) is to be evaluated by an evaluation function \( f(\mu) \); for example, \( \mu(s, n(s)) \) might be evaluated by \( f(s) - f(n(s)) \) or other evaluation functions that return “higher” values for “better” neighbors. A move \( \mu \) is an aggregation of \( |\mu^-| \) negative move attributes \( \mu_1, \ldots, \mu_{|\mu^-|} \), \( \mu_i \in \Psi \) and \( |\mu^+| \) positive move attributes \( \mu_{|\mu^-|+1}^+, \ldots, \mu_{|\mu|}^+ \), \( \mu_i^+ \in \Psi \). The negative (positive) move attributes correspond to those properties of the current solution that are “destroyed” (“created”) when the move under consideration is performed.

The basic paradigm of tabu search is to use information about the search history to guide local search approaches to overcome local optimality [1]. In general, this is done by a dynamic transformation of the local neighborhood, which may lead to performing deteriorating moves when all improving moves of the current neighborhood are set tabu. The various tabu search strategies differ especially in the way the tabu criteria are defined, taking into consideration the information about the search history (performed moves, traversed solutions). A general description of a tabu search frame may be presented as shown in Figure 1 for a given starting solution \( s \) and a generic tabu criterion, which is represented by a component TabuMemory. A neighbor, or a corresponding move, is called admissible, if it is not tabu or if an aspiration criterion is fulfilled. The aspiration criterion may override a possibly inappropriate tabu status. For example, one may allow all moves that lead to a neighbor with a better objective function value than encountered so far. Sometimes, it has proved to be appropriate to incorporate the means to diversify the search into new regions of the search space to a tabu search method. This requires a meaningful mechanism to detect situations in which the search might be trapped in a certain area of the solution space.

### Static Tabu Search

The most commonly used tabu search method is to use a recency-based memory that stores moves, more

---

**Figure 1** Generic tabu search heuristic

\[
\begin{align*}
\text{TabuSearch}_{S,N,\text{TabuMemory}}(s) : \\
\text{initialize TabuMemory} \\
\text{while stopping criterion not fulfilled} \\
& \quad s' = \text{BestAdmissibleNeighbors}_{S,N,\text{TabuMemory}}(s) \\
& \quad \text{TabuMemory.add}(\mu(s, s')) \\
& \quad s = s' \\
& \quad \text{if ( escape triggered by TabuMemory )} \\
& \quad \text{perform a diversifying move}
\end{align*}
\]

---

Exactly move attributes, of the recent past. The basic idea of such static tabu search approaches is to prohibit an appropriately defined inversion of performed moves for a given period (defined by a tabu list length). Considering a performed move \( \mu \), here we follow the following generic procedure: the positive move attributes \( \mu_{|\mu^-|}^+, \ldots, \mu_{|\mu|}^+ \) are stored in the tabu memory. To obtain the current tabu status of a move to a neighbor, one must check whether the negative move attributes of the examined move are contained in the tabu list. As a move generally consists of more than one move attribute, there are different ways to define the tabu criterion. One may use a parameter tabu threshold that defines the number of attributes of a move that have to be contained in the tabu list in order to regard this move as tabu.

### Strict Tabu Search

Strict tabu search embodies the idea of preventing cycling to formerly traversed solutions. That is, the goal of strict tabu search is to provide necessity and sufficiency with respect to the idea of not revisiting any previously visited solution. Accordingly, a move is classified as tabu if and only if it leads to a neighbor that has already been visited during the previous part of the search.

There are two primary mechanisms to accomplish the tabu criterion: First, we may exploit logical interdependencies between the sequence of moves performed throughout the search process, as realized by the reverse elimination method. Second, we may store information about all solutions visited so far. This may be carried out either exactly or, for reasons of efficiency, approximately. For our purpose this may be accomplished by using a hash function that defines a non-injective transformation from the set of solutions to integer numbers [3]. As the hash code of two different solutions may be the same whenever a so-
called collision occurs, moves might be unnecessarily set tabu in some cases. However, as our own experiments have shown, this random effect mostly does not affect the search negatively.

Reactive tabu search
Reactive tabu search aims at the automatic adaptation of the tabu list length in static tabu search [4]. This frees the developer from decisions concerning such parameters, and thus provides appropriate default behavior. The basic idea is to increase the tabu list length when the tabu memory indicates that the search is revisiting formerly traversed solutions. A specific algorithm may be described as follows: Define $\eta_N$ as the average number of negative move attributes for a neighborhood structure $N$. We start with a tabu list length $l$ equal to $\eta_N$ and increase it to $\min\{\max\{1 + \eta_N, l \times \delta\}, u\}$ (for a parameter $\delta > 1$; e.g., $\delta = 1.1$) every time a solution has been repeated, taking into account an appropriate upper bound $u$ (e.g., to guarantee at least one admissible move). If there has been no repetition for a certain number of iterations, we decrease the list length appropriately to $\max\{1 - \eta_N, l \times \delta^{-1}\}, \eta_N\}$. To accomplish the detection of a repetition of a solution, one may apply a trajectory-based memory as described for strict tabu search.

As noted above, and especially for reactive tabu search, noticed by [4], it may be appropriate to include moves for diversifying moves whenever the tabu memory indicates that we may be trapped in a certain region of the search space. As a corresponding trigger mechanism, one may use the combination of at least $\alpha$ solutions each having been traversed $\beta$ times, with two parameters $\alpha$ and $\beta$ (e.g., $\alpha = \beta = 3$).

3. FRAMEWORK FOR HEURISTIC OPTIMIZATION

First, we briefly describe a framework for heuristic search; for a more detailed discussion we refer to [5]. The framework is primarily based on genericity; that is, common behavior is factored out and grouped in generic classes, applying static type variation that enables one to keep respective concerns flexible. For example, a local search component may be parameterized by the neighborhood used. Specific neighborhood types, which may be used to instantiate the component as a specific application, must meet a number of requirements defined by the generic component. This may be called conceptual inheritance: all type parameters of the component represent specializations of such an abstract type with respective syntactic and semantic requirements. In C++, generic classes are defined by using the template mechanism (sometimes termed as static polymorphism), which enables one to leave certain types and values unspecified until the code is actually instantiated and used (compiled). With this, in connection with traditional object-oriented mechanisms such as encapsulation, etc., adaptation of concerns such as the representation of data, the implementation of algorithms, and the tabu criterion used is possible. There are two primary concerns that are reused here. The first concern regards the different heuristics incorporated. The second, but equally important, concern deals with the predefined component collaboration between different components, especially the interface between heuristics and different problem-specific components. As the framework users—the developers of application-specific software—may orient on the framework architecture, they are freed from some of the harder tasks; ideally, one must only add some basic, well-defined functionality.

According to the brief discussion of tabu search concepts above, the abstract problem-specific types involved from a specification perspective (with $X$ representing the problem type) would be a problem class $X.P$, and one or more solution classes $X.S$ for all local search based heuristics. Tabu search heuristics may require for each solution class $X.S$ corresponding classes $X.S.I$, which encapsulate the different solution information functions $I$. For each solution class $X.S$, one or more corresponding neighborhood classes $X.S.N$ have to be implemented. For each neighborhood class $X.S.N$, one or more classes $X.S.N.I$, which encapsulates the transitional information of a move $\mu$ as corresponding move attributes, may be necessary. Having defined such classes, we may instantiate a problem-specific application of a tabu search method as, e.g., $\text{TabuSearch}< X.S, X.S.N, \text{ReactiveTabuCriterion}< X.S.N.I, X.S.I, \ldots >$.

The application of the framework to the problems considered here is also based on the idea of factoring out the common aspects of the different problems into general base classes. For example, we must code the solution representation, i.e., a sequence of a set of objects, in a corresponding class $\text{Sequencing.S}$. Then, the solution information classes $\text{Sequencing.S.I}$ are common for all problem types.

Furthermore, different reasonable neighborhood structures may also be implemented only once. More exactly, a neighborhood base class $\text{Sequencing.S.N}$ covers common aspects of a neighborhood for sequencing problems, while this class is specialized
for the specific neighborhood structures that are described below. For this, consider a permutation \( \Pi \) as a chain of \( n \) objects, each connected by an edge, with two dummy objects that represent a virtual fixed first (0) and last \( (n+1) \) object. A 2-exchange move is defined for a pair \((p_1, p_2)\) with \( 0 \leq p_1 \) and \( p_1 + 2 \leq p_2 \leq n \) as the deletion of the edges \((\pi_{p_1}, \pi_{p_1+1})\) and \((\pi_{p_2}, \pi_{p_2+1})\) and the inclusion of the edges \((\pi_{p_1}, \pi_{p_2})\) and \((\pi_{p_1+1}, \pi_{p_2+1})\). Such a move represents the inversion of the partial sequence between position \( p_1 + 1 \) and position \( p_2 \). The idea for a shift move is to allow the shift of some object to another position. A shift move of the object at position \( p_1 \) to another position \( p_2 \), with \( 1 \leq p_1 \leq n \), \( 1 \leq p_2 \leq n+1 \), \( p_1 \neq p_2 \), \( p_1 + 1 \neq p_2 \), may thus be defined as the deletion of the edges \((\pi_{p_1-1}, \pi_{p_1+1})\), \((\pi_{p_1}, \pi_{p_1+1})\), \((\pi_{p_1-1}, \pi_{p_1+1})\), \((\pi_{p_2}, \pi_{p_2+1})\), \((\pi_{p_1}, \pi_{p_2+1})\). Correspondingly, we may define a swap move for a pair \((p_1, p_2)\) with \( 1 \leq p_1 < p_2 \leq n \) as the swap of the objects \( \pi_{p_1} \) and \( \pi_{p_2} \) while the remaining schedule remains identical. Thus, a swap move corresponds to the deletion of the edges \((\pi_{p_1-1}, \pi_{p_1+1})\), \((\pi_{p_1}, \pi_{p_1+1})\), \((\pi_{p_2-1}, \pi_{p_2+1})\), \((\pi_{p_2}, \pi_{p_2+1})\), \((\pi_{p_1}, \pi_{p_2+1})\). Move attributes \( \mu_k \), i.e., edges \((p_1, p_2)\), are represented by a common class \textsc{Sequencing} \_\textsc{S}\_\textsc{N}\_\textsc{I}. As the treatments of general sequences and moves are common for all problem types, the implementation of the tabu search methods is fully independent from the specific problem under consideration.

4. APPLICATIONS

Our framework has successfully been applied to certain permutation like manufacturing problems as follows.

Pattern Sequencing

The task of pattern sequencing is to construct an optimal permutation of the rows (=patterns) of a given \( n \times n \) matrix \( C \) with respect to some given objective function \([6, 7, 8]\). The interpretations of these matrices are generally based on the assumption that the columns represent some elementary "orders" and that the patterns represent predetermined groupings of these orders. Objective functions considered here differ from traveling salesman like problems by the non-locality of the evaluation; i.e., the evaluation of a permutation cannot be computed by using values that only depend on patterns positioned next to each other. In the following we consider two different objective functions.

First, one may consider restrictions with respect to, e.g., a limited number of containers to be served simultaneously (provided that each order is assigned to one container) or restrictions concerning stock capacity. Such limitations may be represented by the objective of minimizing the maximum number of simultaneously open order stacks (PSP-SOS). For this, an order is called open if its production has been started but not yet been finished. That is, an order \( j \) is defined open at position \( i \) of the pattern sequence if \((\sum_{i=1}^{n} c_{x_{i,j}})\sum_{i=1}^{n} c_{x_{i,j}} > 0 \). Whether PSP-SOS is \( \mathcal{NP} \)-hard or efficiently solvable is open. Yanasse \([7]\) gives an example in the field of wood cutting.

Second, starting from the same assumptions as for the PSP-SOS, with an additional frequency vector \( \alpha \), with \( \alpha_i \) representing the frequency of pattern \( i \), an alternative objective is to find a sequence of order patterns with respect to the following policy: As there may be some handling costs connected with each open order, we may strive to minimize the average length each order is open, the so-called average order spread (PSP-AOS) that is defined for a permutation \( \Pi \) as

\[
AOS(\Pi) = \frac{1}{m} \sum_{j=1}^{m} \left( \max_{i} \left\{ \alpha c_{x_{i,j}} \right\} - \frac{1}{\min \{k|c_{x_{k,j}} > 0\}} \right)
\]

Madsen \([6]\) gives an example from the field of cutting stock problems, where the patterns are results of a prior optimization step with the aim to minimize waste of material. PSP-AOS is \( \mathcal{NP} \)-hard.

Continuous Flow-shop Scheduling

Flow-shop scheduling problems focus on processing a given set of jobs, where each job has to be processed in an identical order on a given number of machines. Each machine can process only one job at a time. The type of problem considered here, the continuous flow-shop scheduling problem, has the additional restriction that the processing of each job has to be continuous, i.e., once the processing of a job begins, there must not be any waiting times between the processing of any consecutive tasks of this job. This may be due to technological restrictions or lack of intermediate storage capacity between the processing stages (machines). Such problems occur, e.g., in chemical or steel production processes. These problems are generally termed as continuous, no-wait or no-intermediate-queue flow-shop problems.

The parameters \( t_{ij} \), \( 1 \leq i \leq n \), \( 1 \leq j \leq m \), denote the processing time of job \( i \) on machine \( j \). Continuous processing of a job generally determines an inevitable delay \( d_{ik} \), \( 1 \leq i \leq n \), \( 1 \leq k \leq n \), \( i \neq k \), on the first
The goal of minimizing the makespan of the schedule, on the first machine until the end of the last job on the last machine, leads to a problem that can be formulated as an asymmetrical traveling salesman problem and so may be tackled by corresponding methods from the stock of algorithms for traveling salesman problems. On the other hand, here we consider the objective to construct a permutation \( \pi \) of the jobs that minimizes the total processing time (flow time):

\[
F(\pi) = \sum_{i=2}^{n} (n+1-i) d_{\pi(i-1),\pi(i)} + \sum_{i=1}^{n} \sum_{j=1}^{m} t_{ij}.
\]

The respective problem is \( NP \)-hard. For a brief literature survey we refer to [9].

**Sequence-dependent Setups**

In the context of production planning with respect to the MRP II concept [10], the consideration of sequence-dependent setups is often neglected. However, the explicit recognition of sequence-dependent setups may be regarded as particularly useful for environments with a tight schedule. Specific problems with respect to determining a reasonable sequencing often occur as sub-problems, part of an superordinate production planning process. The primary properties of such problem are similar. However, usually there are specific differences, e.g., with respect to the objective function or special ordering restrictions.

The generic problem may be defined as follows: In the context of production scheduling with respect to different resources, \( s_{ijk} \) may be defined as the fraction of resource \( k \) used to change over from product \( i \) to product \( j \). The task is to sequence the products for all resources with respect to an objective function (e.g., a specific cost function or surrogates as the makespan) and situation-specific restrictions.

**5. RESULTS**

As the various methods are implemented and put together in a straightforward way without any fine tuning by using the same application skeleton and just changing the strategies (i.e., the components) to be compared, we obtain a way to fairly compare different heuristics by controlled and unbiased experiments. This addresses some of the criticism given by [11] and [12], and allows one to do controlled, unbiased experimentation.

Referring to a more detailed description of the results to [8, 9], we summarize the primary findings for pattern sequencing and continuous flow-shop scheduling. Here we neglect corresponding experimental results for production planning problems with sequence-dependent setups.

**Pattern Sequencing**

We applied the heuristics described above for the larger instances of the problem sets described and used in [13]. The data represent heuristically determined solutions (patterns) of randomly generated cutting stock problems. There are four problem sets for \( n = 50 \) and four problem sets for \( n = 60 \) (altogether 797 problem instances). All data sets were regarded as both PSP-SOS and PSP-AOS.

We restricted ourselves to the neighborhood defined by 2-exchange moves. Whereas we evaluated the neighbors for PSP-AOS by the change of the objective function value, this strategy does not provide enough information for PSP-SOS to guide local search methods into promising search regions. For instance, computational experiments have shown that steepest descent often immediately terminates with the starting solution, as there is no move that leads to an improvement of the objective function value, i.e., a direct decrease of the maximum number of open stacks. However, there is often a sequence of moves that eventually reduce this maximum value. Therefore, we modified the neighborhood evaluation by considering not only the maximum value but also the most significant difference between two solutions [8].

For static tabu search, we used a tabu duration of 50 for 1000 iterations; moves are classified tabu when both attributes of this move are contained in the tabu list. We applied strict tabu search by approximate trajectory for 1000 iterations, strict tabu search by approximate trajectory including five random escape moves after every 100 iterations for 1000 iterations, and reactive tabu search including five random escape moves after every 100 iterations for 1000 and 5000 iterations. The solution quality of the simple tabu search approaches has been found to be unsatisfactory. The primary reason for this might be the inability to diversify the search into unexplored regions of the search space as the neighborhood generally includes \( O(n^2) \) solutions with minor changes of the objective function value. On the other hand, the results have shown the superiority of tabu search methods with the incorporation of simple diversification steps by random escape moves. To summa-
rize the computational experiments, reactive tabu search with the inclusion of escape moves led to overall best results. Simple tabu search approaches alone were not competitive, so we may conclude that diversification has been the crucial component of the heuristic strategies considered here. Concerning the considered pattern sequencing problems, the computational results achieved (regarding solution quality and running times) are superior to those presented in the literature (cf. [8]). This may rule out the exclusive application of simple priority rules as examined in [14, 15].

Continuous Flow-shop Scheduling

We applied the heuristics described above for the benchmark data sets with up to 200 jobs (10 problem instances each) of [16], which have been generated for unrestricted flow-shop scheduling. We considered the three neighborhoods that are described above. The evaluation of potential moves is defined as the change of the objective function value.

The findings from these results may be summarized as follows [9]: The neighborhood structure determined by shift moves might be assumed to be the most effective. The use of good starting solutions (instead of starting with the identity permutation) led for the deterministic methods at the most to better results. Reactive tabu search did not lead to better results than those of static tabu search with the corresponding neighborhood and construction method. In view of the considered flow-shop scheduling problem, our implementation allows the treatment of considerably larger problem instances than previously reported in the literature.

6. CONCLUSIONS

Designing and implementing reusable software components is possible. Building a component library and reusing and adapting these components to specific applications is not effortless (“Nothing is for free”). Nevertheless, this approach may be regarded as the primary way to enable the broad application of the methods under consideration in practice. This addresses, e.g., some of the concerns given in [17], where Reisman criticizes the lack of relevance to practice for the overall majority of research in the field of flow-shop scheduling. Furthermore, the next research steps should not be to improve the objective function value of the considered objectives to the last percent, but to transfer the findings described in this paper to decision support systems dealing with real world problems, not necessarily restricted to manufacturing.

References