Kimms, Alf

A genetic algorithm for multi-level, multi-machine lot sizing and scheduling

Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel, No. 415

Manuskripte
aus den
Instituten für Betriebswirtschaftslehre
der Universität Kiel

No. 415

A Genetic Algorithm for Multi-Level, Multi-Machine Lot Sizing and Scheduling

A. Kimms

No. 415

A Genetic Algorithm for Multi-Level, Multi-Machine Lot Sizing and Scheduling

A. Kimms

October 1996

Alf Kimms
Lehrstuhl für Produktion und Logistik, Institut für Betriebswirtschaftslehre,
Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, 24118 Kiel, Germany
email: Kimms@bwl.uni-kiel.de
URL: http://www.wiso.uni-kiel.de/bwlinstitute/Prod
ftp://ftp.wiso.uni-kiel.de/pub/operations-research

Abstract

This contribution introduces a mixed–integer programming formulation for the multi–level, multi–machine proportional lot sizing and scheduling problem. It also presents a genetic algorithm to solve that problem. The efficiency of that algorithm is due to an encoding of solutions which uses a two–dimensional matrix representation with non–binary entries rather than a simple bitstring. A computational study reveals that the proposed procedure works amazingly fast and competes with a tabu search approach that has recently been published.

Keywords: Multi–level lot sizing, scheduling, genetic algorithms, PLSP

1 Introduction

The problem we are concerned about can be described as follows: Several items are to be produced in order to meet some known (or estimated) dynamic demand without backlogs and stockouts. Precedence relations among these items define an acyclic gozinto–structure of the general type. In contrast to many authors who allow demand for end items only, now, demand may occur for all items including component parts. The finite planning horizon is subdivided into a number of discrete time periods. Positive lead times are given due to technological restrictions such as cooling or transportation for instance. Furthermore, items share common resources. Some (maybe all) of them are scarce. The capacities may vary over time. Producing one item requires an item–specific amount of the available capacity. All data are assumed to be deterministic.

Items which are produced in a period to meet some future demand must be stored in inventory and thus cause item–specific holding costs. Most authors assume that the holding costs for an item must be greater than or equal to the sum of the holding costs for all immediate predecessors. They argue that holding costs are mainly opportunity costs for capital which occurs no matter a component part is assembled or not. Two reasons persuade us to make no particular assumptions for holding costs. First, as it is usual in the chemical industry for instance, keeping some component parts in storage may require ongoing additional effort such as cooling, heating, or shaking. While these parts need no special treatment when processed, storing component parts might be more expensive than storing assembled items. Second, operations such as cutting tin mats for instance make parts smaller and often easier to handle. The remaining "waste" can often be sold as raw material for other manufacturing processes. Hence, opportunity costs may decrease when component parts are assembled. However, it should be made clear that the assumption of general holding costs is the most unrestrictive one. All models and methods developed under this assumption work for more restrictive cases as well.

Each item requires at least one resource for which a setup state has to be taken into account. Production can only take place if a proper state is set

up. Setting a resource up for producing a particular item incurs item–specific setup costs which are assumed to be sequence independent. Setup times are not considered. Once a certain setup action is performed, the setup state is kept up until another setup changes the current state. Hence, same items which are produced having some idle time in–between do not enforce more than one setup action. To get things straight, note that some authors use the word changeover instead of setup in this context.

The most fundamental assumption here is that for each resource at most one setup may occur within one period. Hence, at most two items sharing a common resource for which a setup state exists may be produced per period. Due to this assumption, the problem is known as the proportional lot sizing and scheduling problem (PLSP) [6, 12, 22]. By choosing the length of each time period appropriately small, the PLSP is a good approximation to a continuous time axis. It refines the well–known discrete lot sizing and scheduling problem (DLSP) [4, 8, 15, 24, 30] as well as the continuous setup lot sizing problem (CSLP) [1, 18, 17]. Both assume that at most one item may be produced per period. All three models could be classified as small bucket models since only a few (one or two) items are produced per period. In contrast to this, the well–known capacitated lot sizing problem (CLSP) [3, 7, 10, 14, 23, 26, 27] represents a large bucket model since many items can be produced per period. Remember, the CLSP does not include sequence decisions and is thus a much "easier" problem. An extension of the single–level CLSP with partial sequence decisions can be found in [11]. In [13] a large bucket single–level lot sizing and scheduling model is discussed.

A comprehensive review of the multi–level lot sizing literature is given in [22] where it is shown that most authors do not take capacity restrictions into account and that they make restrictive assumptions such as linear or assembly gozinto–structures. If scarce capacities are considered, the work is mostly confined to single–machine cases. The most general methods are described in [31, 32] where the multi–level CLSP is attacked.

The text is organized as follows: Section 2 gives a precise description of the problem by means of a mixed–integer program. A generic construction procedure is then presented in Section 3. Section 4 refines this procedure and introduces a genetic algorithm. In Section 5 a computational study is performed. Finally, Section 6 summarizes the work.

2 Multi–Level PLSP with Multiple Machines

An important variant of the PLSP is the one with multiple machines (PLSP–MM). Several resources (machines) are available and each item is produced on an item–specific machine. This is to say that there is an unambiguous mapping from items to machines. Of course, some items may share a common machine. Special cases are the single–machine problem for which models and methods

are given in [20, 21], and the problem with dedicated machines where items do not share a common machine. For the latter optimal solutions can be easily computed with a lot-for-lot policy [19].

Let us first introduce some notation. In Table 1 the decision variables are defined. Likewise, the parameters are explained in Table 2. Using this notation, we are now able to present a MIP–model formulation.

Symbol	Definition
I_{jt}	Inventory for item j at the end of period t.
q_{jt}	Production quantity for item j in period t.
x_{jt}	Binary variable which indicates whether a setup for item j occurs in period t ($x_{jt} = 1$) or not ($x_{jt} = 0$).
y_{jt}	Binary variable which indicates whether machine m_j is set up for item j at the end of period t ($y_{jt} = 1$) or not ($y_{jt} = 0$).

Table 1: Decision Variables for the PLSP–MM

$$\min \sum_{j=1}^{J} \sum_{t=1}^{T} (s_j x_{jt} + h_j I_{jt}) \tag{1}$$

subject to

$$I_{jt} = I_{j(t-1)} + q_{jt} - d_{jt} - \sum_{i \in \mathcal{S}_j} a_{ji} q_{it} \qquad \begin{matrix} j = 1, \ldots, J \\ t = 1, \ldots, T \end{matrix} \tag{2}$$

$$I_{jt} \geq \sum_{i \in \mathcal{S}_j} \sum_{\tau=t+1}^{\min\{t+v_j, T\}} a_{ji} q_{i\tau} \qquad \begin{matrix} j = 1, \ldots, J \\ t = 0, \ldots, T-1 \end{matrix} \tag{3}$$

$$\sum_{j \in \mathcal{J}_m} y_{jt} \leq 1 \qquad \begin{matrix} m = 1, \ldots, M \\ t = 1, \ldots, T \end{matrix} \tag{4}$$

$$x_{jt} \geq y_{jt} - y_{j(t-1)} \qquad \begin{matrix} j = 1, \ldots, J \\ t = 1, \ldots, T \end{matrix} \tag{5}$$

$$p_j q_{jt} \leq C_{m_j t} (y_{j(t-1)} + y_{jt}) \qquad \begin{matrix} j = 1, \ldots, J \\ t = 1, \ldots, T \end{matrix} \tag{6}$$

$$\sum_{j \in \mathcal{J}_m} p_j q_{jt} \leq C_{mt} \qquad \begin{matrix} m = 1, \ldots, M \\ t = 1, \ldots, T \end{matrix} \tag{7}$$

$$y_{jt} \in \{0, 1\} \qquad \begin{matrix} j = 1, \ldots, J \\ t = 1, \ldots, T \end{matrix} \tag{8}$$

Symbol	Definition
a_{ji}	"Gozinto"–factor. Its value is zero if item i is not an immediate successor of item j. Otherwise, it is the quantity of item j that is directly needed to produce one item i.
C_{mt}	Available capacity of machine m in period t.
d_{jt}	External demand for item j in period t.
h_j	Non–negative holding cost for having one unit of item j one period in inventory.
I_{j0}	Initial inventory for item j.
$\mathcal{J}_m$	Set of all items that share the machine m, i.e. $\mathcal{J}_m \stackrel{def}{=} \{j \in \{1,\ldots,J\} \mid m_j = m\}$.
J	Number of items.
M	Number of machines.
m_j	Machine on which item j is produced.
p_j	Capacity needs for producing one unit of item j.
s_j	Non–negative setup cost for item j.
$\mathcal{S}_j$	Set of immediate successors of item j, i.e. $\mathcal{S}_j \stackrel{def}{=} \{i \in \{1,\ldots,J\} \mid a_{ji} > 0\}$.
T	Number of periods.
v_j	Positive and integral lead time of item j.
y_{j0}	Unique initial setup state.

Table 2: Parameters for the PLSP–MM

$$I_{jt}, q_{jt}, x_{jt} \geq 0 \qquad \begin{matrix} j = 1,\ldots,J \\ t = 1,\ldots,T \end{matrix} \tag{9}$$

The objective (1) is to minimize the sum of setup and holding costs. Equations (2) are the inventory balances. At the end of a period t we have in inventory what was in there at the end of period $t-1$ plus what is produced minus external and internal demand. To fulfill internal demand we must respect positive lead times. Restrictions (3) guarantee so. Constraints (4) make sure that the setup state of each machine is uniquely defined at the end of each period. Those periods in which a setup happens are spotted by (5). Note that idle periods may occur in order to save setup costs. Due to (6) production can only take place if there is a proper setup state either at the beginning or at the end of a particular period. Hence, at most two items can be manufactured on each machine per period. Capacity constraints are formulated in (7). Since the right hand side is a constant, overtime is not available. (8) define the binary–valued setup state variables, while (9) are simple non–negativity conditions. The reader may convince himself that due to (5) in combination with (1) setup variables x_{jt}

are indeed zero-one valued. Hence, non-negativity conditions are sufficient for these. For letting inventory variables I_{jt} be non-negative backlogging cannot occur.

3 Construction Principles

There is a generic construction scheme that forms the basis of our heuristic. It is a backward oriented procedure which schedules items period by period starting with period T and ending with period one. We choose here a recurrent representation which enables us to develop the underlying ideas in a stepwise fashion. Now, let us assume that $construct(t, \Delta t, m)$ is the procedure to be defined and $t + \Delta t$ is the period and m is the machine under concern. Again, $\Delta t \in \{0, 1\}$ where $\Delta t = 1$ indicates that the setup state for machine m at the beginning of period $t + 1$ is to be fixed next and $\Delta t = 0$ indicates that we already have chosen a setup state at the end of period t. The symbol j_{mt} will denote the setup state for machine m at the end of period t. Assume $j_{mt} = 0$ for $m = 1, \ldots, M$ and $t = 1, \ldots, T$ initially.

Note, from the problem parameters we can easily derive $\mathcal{P}_j$, the set of the immediate predecessors of item j, and $\bar{\mathcal{P}}_j$, the set of all predecessors of item j. Also, nr_j, the net requirement of item j, and id_{ji}, the internal demand for item i that is directly or indirectly caused by producing one unit of item j, are easy to compute.

Before the construction mechanism starts, the decision variables y_{jt} and q_{jt} are assigned zero for $j = 1, \ldots, J$, $m = 1, \ldots, M$, and $t = 1, \ldots, T$. Remember, given the values for y_{jt} and q_{jt} the values for x_{jt} and I_{jt} are implicitly defined. Furthermore, assume auxiliary variables $\tilde{d}_{jt}$ and CD_{jt} for $j = 1, \ldots, J$ and $t = 1, \ldots, T$. The former ones represent the entries in the demand matrix and thus are initialized with $\tilde{d}_{jt} = d_{jt}$. The latter ones stand for the cumulative future demand for item j which is not been met yet. As we will see, the cumulative demand can be efficiently computed while moving on from period to period. For the sake of convenience we introduce $CD_{j(T+1)} = 0$ for $j = 1, \ldots, J$. The remaining capacity of machine m in period t is denoted as RC_{mt}. Initially, $RC_{mt} = C_{mt}$ for $m = 1, \ldots, M$ and $t = 1, \ldots, T$.

The initial call is $construct(T, 1, 1)$ and initiates the fixing of setup states at the end of period T. Table 3 gives all the details.

The choice of j_{mT} needs to be refined, but at this point we do not need any further insight and suppose that the selection is done somehow. All we need to know is that $\mathcal{I}_{mt} \subseteq \mathcal{J}_m \cup \{0\}$ for $m = 1, \ldots, M$ and $t = 1, \ldots, T$ is the set of items among which items are chosen. Item 0 is a dummy item which will be needed for some methods that will be discussed. We will return for a precise discussion in subsequent sections. As one can see, once a setup state is chosen for all machines at the end of period T, a call of $construct(T, 0, 1)$ is made. Table 4 provides a recipe of how to evaluate such calls.

choose $j_{mT} \in \mathcal{I}_{mT}$.
if $(j_{mT} \neq 0)$
    $y_{j_{mT}T} := 1$.
if $(m = M)$
    $construct(T, 0, 1)$.
else
    $construct(T, 1, m+1)$.

Table 3: Evaluating $construct(T, 1, \cdot)$

for $j \in \mathcal{J}_m$
    $CD_{jt} := \min\left\{CD_{j(t+1)} + \tilde{d}_{jt}, \max\{0, nr_j - \sum_{\tau=t+1}^{T} q_{j\tau}\}\right\}$.
if $(j_{mt} \neq 0)$
    $q_{j_{mt}t} := \min\left\{CD_{j_{mt}t}, \frac{RC_{mt}}{p_{j_{mt}}}\right\}$.
    $CD_{j_{mt}t} := CD_{j_{mt}t} - q_{j_{mt}t}$.
    $RC_{mt} := RC_{mt} - p_{j_{mt}} q_{j_{mt}t}$.
    for $i \in \mathcal{P}_{j_{mt}}$
        if $(t - v_i > 0$ and $q_{j_{mt}t} > 0)$
            $\tilde{d}_{i(t-v_i)} := \tilde{d}_{i(t-v_i)} + a_{ij_{mt}} q_{j_{mt}t}$.
if $(m = M)$
    $construct(t-1, 1, 1)$.
else
    $construct(t, 0, m+1)$.

Table 4: Evaluating $construct(t, 0, \cdot)$ where $1 \leq t \leq T$

The situation when calling $construct(t, 0, m)$ is that the setup state j_{mt} has already been chosen. Remarkable to note, how easy it is to take initial inventory into account. This is due to the backward oriented scheme. Evaluating

$$\min\left\{CD_{j(t+1)} + \tilde{d}_{jt}, \max\{0, nr_j - \sum_{\tau=t+1}^{T} q_{j\tau}\}\right\} \tag{10}$$

makes sure that for an item j no more than the net requirement nr_j is produced. Note, cumulating the production quantities is an easy task which can be done very efficiently. Given the cumulative demand $CD_{j_{mt}t}$, production quantities $q_{j_{mt}t}$ can be determined with respect to capacity constraints. Afterwards, we simply update the $\tilde{d}_{jt}$-matrix to take internal demand into account and proceed. Table 5 describes how to evaluate $construct(t, 1, \cdot)$-calls.

choose $j_{mt} \in \mathcal{I}_{mt}$.
if $(j_{mt} \neq 0)$
    $y_{j_{mt}t} := 1$.
    if $(j_{mt} \neq j_{m(t+1)})$
        $q_{j_{mt}(t+1)} := \min\left\{CD_{j_{mt}(t+1)}, \frac{RC_{m(t+1)}}{p_{j_{mt}}}\right\}$.
        $CD_{j_{mt}(t+1)} := CD_{j_{mt}(t+1)} - q_{j_{mt}(t+1)}$.
        $RC_{m(t+1)} := RC_{m(t+1)} - p_{j_{mt}} q_{j_{mt}(t+1)}$.
        for $i \in \mathcal{P}_{j_{mt}}$
            if $(t+1-v_i > 0$ and $q_{j_{mt}(t+1)} > 0)$
                $\tilde{d}_{i(t+1-v_i)} := \tilde{d}_{i(t+1-v_i)} + a_{ij_{mt}} q_{j_{mt}(t+1)}$.
if $(m = M)$
    $construct(t, 0, 1)$.
else
    $construct(t, 1, m+1)$.

Table 5: Evaluating $construct(t, 1, \cdot)$ where $1 \leq t < T$

These lines closely relate to what is defined in Table 4. Differences lie in the fact that a setup state is chosen for the end of period t but items are scheduled in period $t+1$. For computing production quantities we must therefore take into account that item $j_{m(t+1)}$ may already be scheduled in period $t+1$.

Note, the combination of what is given in Tables 4 and 5 enforces that every item j_{mt} that is produced at the beginning of a period $t+1$ is also produced at the end of period t if there is any positive cumulative demand left. In preliminary tests not reported here we also found out that if capacity is exhausted, i.e. if $RC_{m(t+1)} = 0$ and $CD_{j_{m(t+1)}(t+1)} > 0$, it is best to choose $j_{mt} = j_{m(t+1)}$ in Table 5. In other words, lots are not split.[1] The reason why this turned out to be advantageous is that the setup state tends to flicker otherwise and thus the total sum of setup costs tends to be high. In the rest of this chapter we assume that lots are not split.

Turning back to the specification of the *construct*-procedure, it remains to explain what shall happen when the first period is reached. Table 6 describes how to schedule those items in period 1 for which the machines are initially set up for. In contrast to what is given in Table 5 the initial setup state is known and thus needs not to be chosen.

A call to $construct(0, 0, \cdot)$ terminates the construction phase. What is left is

[1] It is worth to be stressed that lot splitting could be easily integrated by *not* checking for exhausted capacity. All methods based on the described construction scheme may thus be adapted for lot splitting with minor modifications only.

if $(j_{m0} \neq j_{m1})$
$\quad q_{j_{m0}1} := \min\left\{CD_{j_{m0}1}, \frac{RC_{m1}}{p_{j_{m0}}}\right\}.$
$\quad CD_{j_{m0}1} := CD_{j_{m0}1} - q_{j_{m0}1}.$
if $(m = M)$
$\quad construct(0, 0, 1).$
else
$\quad construct(0, 1, m + 1).$

Table 6: Evaluating $construct(0, 1, \cdot)$

a final feasibility test where

$$nr_j = \sum_{t=1}^{T} q_{jt} \tag{11}$$

must hold for $j = 1, \ldots, J$ for being a feasible solution. Eventually, the objective function value of a feasible solution can be determined.

To terminate a run of the construction procedure before period 1 is reached, we can perform a capacity check testing

$$\sum_{j \in \mathcal{J}_m} \sum_{i \in \{\bar{\mathcal{P}}_j \cup \{j\}\} \cap \mathcal{J}_m} p_i id_{ji} CD_{j(t+\Delta t)} > \sum_{\tau=1}^{t+\Delta t} C_{m\tau} \tag{12}$$

which must be false for $m = 1, \ldots, M$ if period $t+\Delta t$ is under concern and thus, when true, indicates an infeasible solution (if there is no initial inventory).

It should be emphasized again, that the construction scheme described above does not necessarily generate an optimum solution. It does not even guarantee to find a feasible solution if there exists one.

4 Genetic Algorithms

A key element of what is assumed to be intelligence is the capability to learn from past experience. Especially when things are done repeatedly, intelligent behavior would avoid doing a mistake more than once and would prefer making advantageous decisions again.

For optimization a class of today's most popular heuristic approaches is known as genetic algorithms. Due to its widespread use and the vast amount of literature dealing with genetic algorithms, e.g. [2, 9, 16, 25, 28, 29, 33], a comprehensive review of research activities is doomed to failure. Thus, we stick to an outline of the fundamental ideas.

The adjective genetic reveals the roots of these algorithms. Adapting the evolution strategy from natural life forms, the basic idea is to start with a set

of (feasible) solutions and to compute a set of new solutions by applying some well–defined operators on the old ones. Then, some solutions (new and/or old ones) are selected to form a new set with which another iteration is started, and so on until some stopping criterion is met. Solutions are represented by sets of attributes, and different solutions are represented by different collections of attribute values. The decision which solutions are dismissed and which are taken over to form a new starting point for the next iteration is made on the basis of a priority rule.

Most authors use notions from evolution theory in this context. The set of solutions an iteration starts with is usually called the parent population while the set of new solutions is the child population. Each iteration represents a generation. A member of a population is an individual or a chromosome, thus we have parent and child individuals (or chromosomes). The attribute values that belong to an individual are called genes. This coins the name of this type of algorithm. The operations for procreating new individuals are applications of so–called genetic operators. Attached to each individual is a fitness value which functions as a priority rule to select the parent individuals for the next generation. This mechanism should simulate what is observed in nature where only the fittest survive and the weak die, good characteristics are inherited and bad ones become extinct.

Up to here, there are many degrees of freedom and thus genetic algorithms are often called meta–heuristics. To develop a method based on the ideas of genetic algorithms for a specific problem, we need to provide some more ingredients. First, we need to specify how to encode a solution of the problem as a set of attributes. Furthermore, a definition of how to compute fitness values needs to be given. Also, we need to define genetic operators. Eventually, the way to select a new parent population must be described. Of minor importance, but not to forget, is a stopping criterion, e.g. a total number of iterations, set by the user. The population sizes denoted as $PARENT$ for the size of the parent population and $CHILD$ for the number of child individuals, respectively, are specified by the user, too.

4.1 Problem Representation

Traditionally, a solution of a given problem is represented as a bitstring, i.e. a sequence of binary values [9]. For many problems this gives not a very compact representation of solutions. So, in some applications genes are chosen to be more complex rather than being binary–valued only. See for instance [5] for an application to job shop scheduling where a gene represents a rule to select a job for scheduling.

For representing a solution of the PLSP, we have chosen a two–dimensional matrix with M rows and T columns. Since we consider a population of matrices,

let each matrix be identified by a unique label

$$k \in \{1, \ldots, PARENT, PARENT+1, \ldots, PARENT+CHILD\}. \quad (13)$$

An entry in row m and column t in the matrix k is a rule $\vartheta_{mtk} \in \Theta$ for selecting the setup state for machine m at the end of period t out of the set $\mathcal{I}_{mt}$. Here, Θ denotes the set of all selection rules which is to be defined. To get things straight, recall that the matrices are the individuals now, and that selection rules are genes.

4.2 Setup State Selection Rules

Though the rules to select setup states is a detail that can be skipped on first reading, it certainly is a significant aspect for the performance of the construction scheme. We will now suggest several rules for selecting a setup state for machine m at the end of period t where $m = 1, \ldots, M$ and $t = 1, \ldots, T$. In the following, let us assume that whenever ties are to be broken, we favor items with a low index by arbitration. If $\mathcal{I}_{mt} = \emptyset$ given the definitions below, then we choose the dummy item $j_{mt} = 0$.

Rule θ_1*:* Maximum Holding Costs

Consider those items for which there is demand in period $t+1$, i.e.

$$\mathcal{I}_{mt} \stackrel{def}{=} \{j \in \mathcal{J}_m \mid CD_{j(t+1)} > 0\} \quad (14)$$

$$\cap \{j \in \mathcal{J}_m \mid nr_j - \sum_{\tau=t+1}^{T} q_{j\tau} > 0\}.$$

When setting machine m up for an item i,

$$\sum_{j \in \mathcal{I}_{mt} \setminus \{i\}} h_j CD_{j(t+1)}$$

are the holding costs that are charged to keep the remaining items in inventory. Note, this is just an estimate which assumes that all $CD_{i(t+1)}$ items can indeed be scheduled in period $t+1$. If this is not true, item i would incur holding costs, too. To keep these costs low we should choose an item causing high holding costs, i.e.

$$j_{mt} \in \left\{ i \in \mathcal{I}_{mt} \mid h_i CD_{i(t+1)} = \max_{j \in \mathcal{I}_{mt}} \{h_j CD_{j(t+1)}\} \right\}. \quad (15)$$

Rule θ_2*:* Minimum Setup Costs

In order to keep setup costs low, we choose

$$j_{mt} = j_{m(t+1)}, \quad (16)$$

if $j_{m(t+1)} \neq 0$ and $\tilde{d}_{j_{m(t+1)}t} > 0$. If this does not hold, we consider the items with demand in period t or period $t+1$, i.e.

$$\mathcal{I}_{mt} \stackrel{def}{=} \{j \in \mathcal{J}_m \mid CD_{j(t+1)} + \tilde{d}_{jt} > 0\} \tag{17}$$

$$\cap\{j \in \mathcal{J}_m \mid nr_j - \sum_{\tau=t+1}^{T} q_{j\tau} > 0\},$$

and choose the one with lowest setup costs. That is,

$$j_{mt} \in \left\{ i \in \mathcal{I}_{mt} \mid s_i = \min_{j \in \mathcal{I}_{mt}} \{s_j\} \right\}. \tag{18}$$

Rule θ_3: Introduce Idle Periods

To enforce keeping a machine idle we allow to choose items for which there is demand in periods prior to t. In this case,

$$\mathcal{I}_{mt} \stackrel{def}{=} \{j \in \mathcal{J}_m \mid CD_{j(t+1)} + \sum_{\tau=1}^{t} \tilde{d}_{j\tau} > 0\} \tag{19}$$

$$\cap\{j \in \mathcal{J}_m \mid nr_j - \sum_{\tau=t+1}^{T} q_{j\tau} > 0\}$$

is the item set under consideration. For $j \in \mathcal{I}_{mt}$ we determine

$$t_j \stackrel{def}{=} \begin{cases} t+1 & \text{, if } CD_{j(t+1)} > 0 \\ \max\{\tau \mid 1 \leq \tau \leq t \wedge \tilde{d}_{j\tau} > 0\} & \text{, otherwise} \end{cases} \tag{20}$$

which is the latest period less than or equal to $t+1$ with demand for item j. Since idle periods bear the risk to lead to an infeasible result, idle periods should not last too long. Hence, we choose

$$j_{mt} \in \left\{ i \in \mathcal{I}_{mt} \mid t_i = \max_{j \in \mathcal{I}_{mt}} \{t_j\} \right\}. \tag{21}$$

Rule θ_4: Maximum Depth

To avoid infeasibility, it might be a good idea to choose items with a large depth. Thus, taking the items given by (17) into account, the setup state should be chosen using

$$j_{mt} \in \left\{ i \in \mathcal{I}_{mt} \mid dep_i = \max_{j \in \mathcal{I}_{mt}} \{dep_j\} \right\}. \tag{22}$$

Rule θ_5: Maximum Number of Predecessors

Quite similar to rule θ_4 is the rule proposed now. This time, we take the total number of predecessors into account. Again, consider the items defined by (17) and choose

$$j_{mt} \in \left\{ i \in \mathcal{I}_{mt} \;\middle|\; \mid \bar{\mathcal{P}}_i \mid = \max_{j \in \mathcal{I}_{mt}} \{\mid \bar{\mathcal{P}}_j \mid\} \right\}. \tag{23}$$

Rule θ_6: Maximum Demand for Capacity

Determining the capacity utilization of the bottleneck machine also tends to avoid infeasible solutions. Focusing on the items defined in (17) again, we compute

$$cap_j \stackrel{def}{=} (CD_{j(t+1)} + \tilde{d}_{jt}) \cdot \max\left\{ \frac{\sum_{i \in (\mathcal{P}_j \cup \{j\}) \cap \mathcal{J}_m} p_i d_{ji}}{\sum_{\tau=1}^{t} C_{m\tau}} \mid m \in \{1, \ldots, M\} \right\} \tag{24}$$

for $j \in \mathcal{I}_{mt}$. Afterwards, we choose

$$j_{mt} \in \left\{ i \in \mathcal{I}_{mt} \mid cap_i = \max_{j \in \mathcal{I}_{mt}} \{cap_j\} \right\}. \tag{25}$$

Rule θ_7: Pure Random Choice

Last, j_{mt} can be chosen out of the set given by (17) with a pure random choice to give items with no extreme characteristic a chance to be selected.

In summary, we have

$$\Theta \stackrel{def}{=} \{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \theta_7\} \tag{26}$$

and sets of items $\mathcal{I}_{mt}$ to choose among as defined above. This is what is used in our tests. Note, following our arguments for choosing composite priority values for the regret based method in Section 6.3, we have introduced both, rules that tend to give cheap production plans and rules that tend to give feasible plans. In contrast to a composite criterion, the rules given here need less effort to be evaluated. All rules but θ_7 operate deterministically.

4.3 Fitness Values

To compute a fitness value $fitness_k$ for an individual k we call the construction scheme using the selection rules $\vartheta_{mtk} \in \Theta$ for choosing the setup states. Let $fitness_k$ be the objective function value of the production plan that is constructed when matrix k is used (and let $fitness_k = \infty$, if no feasible plan was found using matrix k). It should be clear that due to this definition searching for an individual with utmost fitness in fact means to look for an individual with lowest possible fitness value.

4.4 Genetic Operators

In order to generate a new parent population out of an old one, we employ three different operators. First, a so–called crossover combines two parent individuals to procreate one child individual. Second, mutation introduces non–determinism

into the inheritance. And third, a selection filters the new parent population out of the last generation. The details of these operators shall be given now.

The crossover operates on two matrices, say k_1 and k_2 where $k_1, k_2 \in \{1, \ldots, PARENT\}$. Applying a crossover then cuts the two matrices into four pieces each and puts some of the submatrices together yielding a new matrix $k_3 \in \{PARENT+1, \ldots, PARENT+CHILD\}$ of the same size. For doing so, suppose that two numbers $\hat{m}_{k_3} \in \{1, \ldots, M\}$ and $\hat{t}_{k_3} \in \{1, \ldots, T\}$ are given. More formally, the resulting matrix k_3 is defined as

$$\vartheta_{mtk_3} \stackrel{def}{=} \begin{cases} \vartheta_{mtk_1} & \text{, if } m \leq \hat{m}_{k_3} \text{ and } t \leq \hat{t}_{k_3} \\ \vartheta_{mtk_2} & \text{, if } m \leq \hat{m}_{k_3} \text{ and } t > \hat{t}_{k_3} \\ \vartheta_{mtk_2} & \text{, if } m > \hat{m}_{k_3} \text{ and } t \leq \hat{t}_{k_3} \\ \vartheta_{mtk_1} & \text{, if } m > \hat{m}_{k_3} \text{ and } t > \hat{t}_{k_3} \end{cases} \tag{27}$$

for $m = 1, \ldots, M$ and $t = 1, \ldots, T$.

The mutation stochastically changes some entries of a matrix k. Let $MUTATION \in [0, 1]$ be a (small) probability to change an entry. Furthermore, suppose $prob_{mt} \in [0, 1]$ is drawn at random with uniform distribution where $m = 1, \ldots, M$ and $t = 1, \ldots, T$. Then, the mutation of matrix k is defined as

$$\vartheta'_{mtk} = \begin{cases} \vartheta_{mtk} & \text{, if } prob_{mt} \geq MUTATION \\ \hat{\theta}_{mt} & \text{, otherwise} \end{cases} \tag{28}$$

for $m = 1, \ldots, M$ and $t = 1, \ldots, T$, where $\hat{\theta}_{mt}$ is drawn at random out of Θ with uniform distribution.

The selection of $PARENT$ individuals which form a new parent population is done deterministically choosing those matrices with the highest fitness values. Ties are broken randomly. The effort to find these is the effort of sorting $PARENT+CHILD$ objects. Without loss of generality, we assume the selected individuals be relabeled having unique indices $k = 1, \ldots, PARENT$.

4.5 The Working Principle in a Nutshell

Initially, the genetic algorithm starts with a parent population that is randomly generated by drawing a rule ϑ_{mtk} for each position (m, t) in the matrix k with uniform distribution out of the set of rules Θ where $m = 1, \ldots, M$, $t = 1, \ldots, T$, and $k = 1, \ldots, PARENT$. Then, we compute the fitness values for the matrices $k = 1, \ldots, PARENT$. To do so, we have to execute the construction scheme a total of $PARENT$ times.

Afterwards, a population of $CHILD$ individuals with unique indices $k = PARENT+1, \ldots, PARENT+CHILD$ is generated using the crossover operation. The two parent individuals that are combined to create a new child individual k are randomly chosen out of $\{1, \ldots, PARENT\}$ with uniform distribution. The values $\hat{m}_k$ and $\hat{t}_k$ used as parameters for the crossover operators are integral random numbers which are drawn out of $[1, \ldots, M]$ and

$[1, \ldots, T]$, respectively, with uniform distribution. Mutation of all child individuals is done next. Eventually, the fitness values for the matrices $k = PARENT+1, \ldots, PARENT+CHILD$ are computed executing the construction scheme $CHILD$ times. Finally, the parent population for the next generation is selected having (new) indices $k = 1, \ldots, PARENT$. The process is repeated starting with the generation of new child individuals until some stopping criterion is met.

The production plan with the lowest objective function value found during all iterations is given as a result.

4.6 An Example

Consider the gozinto–structure given in Figure 1 and the parameters in Table 7. Furthermore, assume $M = 2$, $m_1 = m_4 = 1$, $m_2 = m_3 = 2$, and $C_{1t} = C_{2t} = 15$ for $t = 1, \ldots, 10$. Let us suppose that $s_2 < s_3$ holds. For illustrating the construction of a production plan we do not need any information about holding costs. Furthermore, assume a matrix k filled with selection rules as given in Table 8.

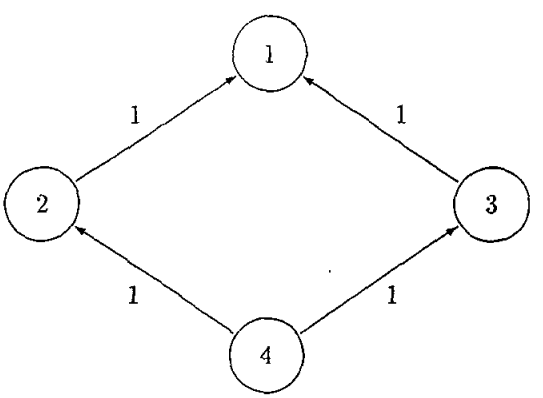

Figure 1: A Gozinto–Structure with Four Items

A protocol of running the construction scheme is shown in Table 9. Figure 2 depicts a plan that could be the outcome when completing the protocol.

Some interesting points shall be explained in a little more detail.

Step 2: The set of items $\mathcal{I}_{2T}$ to choose among is empty. Hence, we choose the dummy item 0.

Step 5: Item 1 is chosen again, because lot splitting is not allowed.

d_{jt}	$t=1$	...	5	6	7	8	9	10	p_j	v_j	y_{j0}	I_{j0}
$j=1$					20			20	1	1	1	0
$j=2$									1	1	1	0
$j=3$									1	1	0	0
$j=4$						5			1	1	0	0

Table 7: Parameters of the Example

ϑ_{mtk}	$t=1$	2	3	4	5	6	7	8	9	10
$m=1$	θ_7	θ_4	θ_1	θ_7	θ_3	θ_5	θ_6	θ_3	θ_5	θ_4
$m=2$	θ_2	θ_6	θ_4	θ_1	θ_7	θ_3	θ_5	θ_1	θ_2	θ_7

Table 8: A Matrix of Setup State Selection Rules

Step 6: Item 2 is chosen due to the selection rule $\vartheta_{29k} = \theta_2$ which chooses the item with the lowest setup costs. Remember, we have assumed $s_2 < s_3$.

Step 9: To set machine 1 up at the end of period 8, we use the selection rule $\vartheta_{18k} = \theta_3$ which may introduce idle periods. In the item set $\mathcal{I}_{18}$ we have both, item 1 and item 4, because there is demand for item 1 in period 7 and demand for item 4 in period 8. Since idle time is kept as short as possible, item 4 is chosen.

Step 10: The selection rule to be employed is $\vartheta_{28k} = \theta_1$ which chooses the item with the maximum holding costs. For item 3 being the only one with cumulative demand, $\mathcal{I}_{28} = \{3\}$ and no other item is contained in the item set. As one can see, the capacity of machine 2 in period 9 is used up by item 2 which was scheduled in Step 8. Thus, item 3 cannot be scheduled in period 9, but in period 8 (Step 12).

5 Experimental Evaluation

To test the performance, the genetic algorithm is applied to the small PLSP-MM-instances which are defined in [22]. All tests are conducted running a C-implementation on a Pentium computer with 120 MHz. This test-bed consists of a collection of 1,080 instances with $J = 5$ and $T = 10$ which is small enough to be solved optimally with standard solvers and large enough to construct non-trivial instances. A full factorial experimental design is used where different levels of the following parameters are combined: M (the number of machine), C (the complexity of the gozinto-structure), $(T_{macro}, T_{micro}, T_{idle})$ (the demand pattern), $COSTRATIO$ (the ratio of setup and holding costs), and U

Step	$(t, \Delta t, m)$	$\vec{d}_{jt}$	$\vec{CD}_{j(t+\Delta t)}$	$\mathcal{I}_{mt}$	j_{mt}	$q_{j_{mt}(t+\Delta t)}$
1	(10,1,1)	(20,0,0,0)	(0,0,0,0)	{1}	1	
2	(10,1,2)	(20,0,0,0)	(0,0,0,0)	$\emptyset$	0	
3	(10,0,1)	(20,0,0,0)	(20,0,0,0)			$q_{1T} = 15$
4	(10,0,2)	(20,0,0,0)	(5,0,0,0)			
5	(9,1,1)	(0,15,15,0)	(5,0,0,0)	{1}	1	
6	(9,1,2)	(0,15,15,0)	(5,0,0,0)	{2,3}	2	$q_{2T} = 0$
7	(9,0,1)	(0,15,15,0)	(5,0,0,0)			$q_{19} = 5$
8	(9,0,2)	(0,15,15,0)	(0,15,15,0)			$q_{29} = 15$
9	(8,1,1)	(0,5,5,20)	(0,0,15,0)	{1,4}	4	$q_{49} = 0$
10	(8,1,2)	(0,5,5,20)	(0,0,15,0)	{3}	3	$q_{39} = 0$
11	(8,0,1)	(0,5,5,20)	(0,0,15,20)			$q_{48} = 15$
12	(8,0,2)	(0,5,5,20)	(0,5,20,5)			$q_{38} = 15$
13	(7,1,1)	(20,0,0,15)	(0,5,5,5)	{4}	4	
14	(7,1,2)	(20,0,0,15)	(0,5,5,5)	{3}	3	
...						

Table 9: A Protocol of the Construction Scheme of the Genetic Algorithm

(the capacity utilization). For each parameter level combination, 10 instances are generated using common random numbers. It turned out that 1,033 out of the 1,080 instances have a feasible solution. For more details about the test–bed, we refer to [22]. The mutation probability is chosen to be 0.1. The method parameters are chosen as $PARENT = 20$, $CHILD = 10$, and 98 being the number of generations. This gives a total of 1,000 runs of the construction phase.

To find out which parameter levels have what effect on the performance, we aggregate the data. Table 10 focuses on the number of machines. As we can see, additional machines increase the average deviation from the optimum, but reduce the infeasibility ratio. In both cases, the effect is remarkably large. Only small changes are measured for the run–time performance.

	$M = 1$	$M = 2$
Average Deviation	17.72	21.90
Infeasibility Ratio	20.39	14.67
Average Run–Time	0.08	0.11

Table 10: The Impact of the Number of Machines on the Performance

Table 11 examines the impact of the gozinto–structure complexity on the performance. It becomes clear that a high complexity has a drastic negative effect on both, the average deviation from the optimum as well as the infeasibility

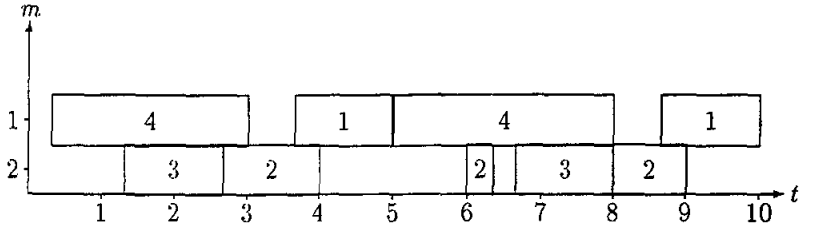

Figure 2: A Possible Outcome of the Run in the Protocol

ratio. The run-time performance is not affected.

	$C = 0.2$	$C = 0.8$
Average Deviation	18.21	21.80
Infeasibility Ratio	13.71	21.46
Average Run-Time	0.10	0.10

Table 11: The Impact of the Gozinto-Structure Complexity on the Performance

For different demand patterns, Table 12 shows that many positive entries in the demand matrix have a dramatic effect on the average deviation from the optimum and on the infeasibility ratio. The results turn out to be very poor. The run-time performance, however, does not change.

	$(T_{macro}, T_{micro}, T_{idle}) =$		
	$(10, 1, 5)$	$(5, 2, 2)$	$(1, 10, 0)$
Average Deviation	36.68	22.19	6.69
Infeasibility Ratio	32.19	18.21	2.51
Average Run-Time	0.10	0.10	0.10

Table 12: The Impact of the Demand Pattern on the Performance

An investigation of different cost structures is performed in Table 13. Clearly, this parameter level has a significant impact on the average deviation from the optimum. While low setup costs give the worse result, high setup costs give only second best results. The best average deviation is reached for a balanced cost structure. The infeasibility ratio and the run-time performance are almost unaffected by different costs.

	$COSTRATIO =$		
	5	150	900
Average Deviation	26.20	15.78	17.66
Infeasibility Ratio	17.39	17.68	17.49
Average Run–Time	0.10	0.10	0.10

Table 13: The Impact of the Cost Structure on the Performance

The capacity utilization is studied in Table 14. The best average deviation from the optimum is gained for a high utilization. However, the infeasibility ratio grows quickly when the capacity utilization is increased. For $U = 70$, four out of ten instances cannot be solved. Once more, the run–time performance remains stable.

	$U = 30$	$U = 50$	$U = 70$
Average Deviation	19.75	21.41	17.60
Infeasibility Ratio	0.00	13.13	42.01
Average Run–Time	0.10	0.10	0.10

Table 14: The Impact of the Capacity Utilization on the Performance

In summary, the genetic algorithm is unable to solve 181 out of the 1,033 instances in the test–bed. This corresponds to an overall infeasibility ratio of 17.52%. The average run–time is 0.10 CPU–seconds. The average deviation from the optimum objective function value is 19.89%.

The most important method parameters of the genetic algorithm are the sizes of the parent and the child population. Hence, Table 15 gives some insight into what happens if these values are varied. All other parameters are kept as they are.

PARENT	*CHILD*	Average Deviation	Infeasibility Ratio	Average Run–Time
20	10	19.89	17.52	0.10
200	10	113.73	0.87	0.26
200	100	14.75	8.71	1.10

Table 15: The Impact of the Population Sizes on the Performance

It turns out that increasing the population sizes reduces the infeasibility ratio

remarkably. Only nine out of 1,033 instances are left unresolved when we choose $PARENT = 200$ and $CHILD = 10$. With respect to the average deviation from the optimum, it becomes clear that the parent population should not be chosen too large in comparison with the child population. Since the genetic algorithm works very fast, it is no problem to evaluate a large number of calls to the construction scheme. For 98 generations where the parent population contains 200 individuals and each child population contains 100 individuals we have to execute the construction scheme 10,000 times which can be done in round about one second.

To prove that the genetic algorithm indeed makes a contribution, we briefly report about the results of a disjunctive arc based tabu search which has recently been described in [21] where the single–machine case is considered only. A straightforward extension to multiple machines is presented in [22]. Applying the tabu search procedure to the same test–bed and evaluating 1,000 production plans gives the results provided in Table 16.

	Average Deviation	Infeasibility Ratio	Average Run–Time
Disjunctive Arc Based Tabu Search	17.59	35.62	0.50

Table 16: Results of the Disjunctive Arc Based Tabu Search

We can see that the genetic algorithm clearly dominates the tabu search procedure in terms of run–time performance and in terms of the ability to find feasible solutions. The average deviation from the optimum result is slightly better for the tabu search if both procedures evaluate the same number of production plans. But, since the genetic algorithm is much faster, it offers the opportunity to evaluate more plans within the same computation time which may reduce the deviation from the optimum. For a fair comparison we run the genetic algorithm with $PARENT = 20$ and $CHILD = 10$ again evaluating 500 generations this time. Also, we used the parameters $PARENT = 200$ and $CHILD = 100$ evaluating 40 generations. In both cases the average run–time of the genetic algorithm is 0.48 which is almost the same computational effort that is spent on the tabu search. In the former case, the average deviation from the optimum result is 19.89 which means that nothing changes for such a small population. In the latter case the average deviation increases to 30.65 which is rather poor. Hence, we cannot state that the genetic algorithm dominates the tabu search with respect to the deviation from the optimum result.

6 Conclusion

We have presented a mixed-integer programming model for multi-level, multi-machine lot sizing and scheduling. A heuristic for this problem has been introduced using the idea of genetic algorithms. Rather than working on bitstrings, the proposed genetic algorithm operates on two-dimensional matrices with non-binary entries. The genetic algorithm was proven to dominate a recently proposed tabu search method in terms of run-time performance and in terms of the ability to find feasible solutions. In terms of the average deviation from the optimum objective function value, the genetic algorithm gives competitive results.

Acknowledgement

This work was done with partial support from the DFG-project Dr 170/4-1. We are indebted to Andreas Drexl for his insightful comments.

References

[1] BITRAN, G.R., MATSUO, H., (1986), Approximation Formulations for the Single-Product Capacitated Lot Size Problem, Operations Research, Vol. 34, pp. 63–74

[2] DAVIS, L., (ed.), (1991), Handbook of Genetic Algorithms, New York, van Nostrand Reinhold

[3] DIABY, M., BAHL, H.C., KARWAN, M.H., ZIONTS, S., (1992), A Lagrangean Relaxation Approach for Very-Large-Scale Capacitated Lot-Sizing, Management Science, Vol. 38, pp. 1329–1340

[4] DINKELBACH, W., (1964), Zum Problem der Produktionsplanung in Ein- und Mehrproduktunternehmen, Würzburg, Physica, 2nd edition

[5] DORNDORF, U., PESCH, E., (1995), Evolution Based Learning in a Job Shop Scheduling Environment, Computers & Operations Research, Vol. 22, pp. 25–40

[6] DREXL, A., HAASE, K., (1995), Proportional Lotsizing and Scheduling, International Journal of Production Economics, Vol. 40, pp. 73–87

[7] EPPEN, G.D., MARTIN, R.K., (1987), Solving Multi-Item Capacitated Lot-Sizing Problems Using Variable Redefinition, Operations Research, Vol. 35, pp. 832–848

[8] FLEISCHMANN, B., (1990), The Discrete Lot-Sizing and Scheduling Problem, European Journal of Operational Research, Vol. 44, pp. 337–348

[9] GOLDBERG, D.E., (1989), Genetic Algorithms in Search, Optimization, and Machine Learning, Reading, Addison–Wesley

[10] GÜNTHER, H.O., (1987), Planning Lot Sizes and Capacity Requirements in a Single–Stage Production System, European Journal of Operational Research, Vol. 31, pp. 223–231

[11] HAASE, K., (1993), Capacitated Lot–Sizing with Linked Production Quantities of Adjacent Periods, Working Paper No. 334, University of Kiel

[12] HAASE, K., (1994), Lotsizing and Scheduling for Production Planning, Lecture Notes in Economics and Mathematical Systems, Vol. 408, Berlin, Springer

[13] HAASE, K., KIMMS, A., (1996), Lot Sizing and Scheduling with Sequence Dependent Setup Costs and Times and Efficient Rescheduling Opportunities, Working Paper No. 393, University of Kiel

[14] HINDI, K.S., (1996), Solving the CLSP by a Tabu Search Heuristic, Journal of the Operational Research Society, Vol. 47, pp. 151–161

[15] VAN HOESEL, S., KOLEN, A., (1994), A Linear Description of the Discrete Lot–Sizing and Scheduling Problem, European Journal of Operational Research, Vol. 75, pp. 342–353

[16] HOLLAND, J.H., (1975), Adaptation in Natural and Artificial Systems, Ann Arbor, The University of Michigan Press

[17] KARMARKAR, U.S., KEKRE, S., KEKRE, S., (1987), The Deterministic Lotsizing Problem with Startup and Reservation Costs, Operations Research, Vol. 35, pp. 389–398

[18] KARMARKAR, U.S., SCHRAGE, L., (1985), The Deterministic Dynamic Product Cycling Problem, Operations Research, Vol. 33, pp. 326–345

[19] KIMMS, A., (1994), Optimal Multi–Level Lot Sizing and Scheduling with Dedicated Machines, Working Paper No. 351, University of Kiel

[20] KIMMS, A., (1996), Multi–Level, Single–Machine Lot Sizing and Scheduling (with Initial Inventory), European Journal of Operational Research, Vol. 89, pp. 86–99

[21] KIMMS, A., (1996), Competitive Methods for Multi–Level Lot Sizing and Scheduling: Tabu Search and Randomized Regrets, International Journal of Production Research, Vol. 34, pp. 2279–2298

[22] KIMMS, A., (1996), Multi–Level Lot Sizing and Scheduling — Methods for Capacitated, Dynamic, and Deterministic Models, Ph.D. dissertation, University of Kiel

[23] KIRCA, Ö., KÖKTEN, M., (1994), A New Heuristic Approach for the Multi–Item Dynamic Lot Sizing Problem, European Journal of Operational Research, Vol. 75, pp. 332–341

[24] LASDON, L.S., TERJUNG, R.C., (1971), An Efficient Algorithm for Multi–Item Scheduling, Operations Research, Vol. 19, pp. 946–969

[25] LIEPINS, G.E., HILLIARD, M.R., (1989), Genetic Algorithms: Foundations and Applications, Annals of Operations Research, Vol. 21, pp. 31–58

[26] LOTFI, V., CHEN, W.H., (1991), An Optimal Algorithm for the Multi–Item Capacitated Production Planning Problem, European Journal of Operational Research, Vol. 52, pp. 179–193

[27] MAES, J, VAN WASSENHOVE, L.N., (1988), Multi–Item Single–Level Capacitated Dynamic Lot–Sizing Heuristics: A General Review, Journal of the Operational Research Society, Vol. 39, pp. 991–1004

[28] MÜHLENBEIN, H., GORGES–SCHLEUTER, M., KRÄMER, O., (1988), Evolution Algorithms in Combinatorial Optimization, Parallel Computing, Vol. 7, pp. 65–85

[29] REEVES, C., (1993), Modern Heuristic Techniques for Combinatorial Problems, Oxford, Blackwell

[30] SALOMON, M., KROON, L.G., KUIK, R., VAN WASSENHOVE, L.N., (1991), Some Extensions of the Discrete Lotsizing and Scheduling Problem, Management Science, Vol. 37, pp. 801–812

[31] TEMPELMEIER, H., DERSTROFF, M., (1996), A Lagrangean–Based Heuristic for Dynamic Multi–Level Multi–Item Constrained Lotsizing with Setup Times, Management Science, Vol. 42, pp. 738–757

[32] TEMPELMEIER, H., HELBER, S., (1994), A Heuristic for Dynamic Multi–Item Multi–Level Capacitated Lotsizing for General Product Structures, European Journal of Operational Research, Vol. 75, pp. 296–311

[33] WHITLEY, D., (1993), Foundations of Genetic Algorithms 2, Morgan Kaufmann