A genetic algorithm to solve the general multi-level lot-sizing problem with time-varying costs

N. Dellaert¹, J. Jeunet²,*, N. Jonard³,⁴

¹Department of Operations Management, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, Netherlands
²Laboratoire de Recherche en Gestion, Université Louis Pasteur, 61 Avenue de la Forêt Noire, 67070 Strasbourg Cedex, France
³CNRS, Bureau d’Economie Théorique et Appliquée, 61 Avenue de la Forêt Noire, 67070 Strasbourg Cedex, France
⁴Maastricht Economic Research Institute on Innovation and Technology, P.O. Box 616, 6200 MD, Maastricht, Netherlands

Received 17 May 2000; accepted 14 June 2000

Abstract

The multi-level lot-sizing (MLLS) problem in material requirements planning (MRP) systems belongs to those problems that industry manufacturers daily face in organizing their overall production plans. However, this combinatorial optimization problem can be solved optimally in a reasonable CPU only when very small instances are considered. This legitimates the search for heuristic techniques that achieve a satisfactory balance between computational demands and cost effectiveness. In this paper, we propose a solution method that exploits the virtues and relative simplicity of genetic algorithms to address combinatorial problems. The MLLS problem that is examined here is the most general version in which the possibility of time-varying costs is allowed. We develop a binary encoding genetic algorithm and design five specific genetic operators to ensure that exploration takes place within the set of feasible solutions. An experimental framework is set up to test the efficiency of the proposed method, which turns out to rate high both in terms of cost effectiveness and execution speed. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Genetic algorithm; Lot-sizing; General product structure

1. Introduction

Material requirements planning (MRP) is an old field of study within business, but it still plays an important part in coordinating replenishment decisions for complex finished goods. There are actually reasons to believe that the rise in consumers’ demands and expectations, and the subsequent increase in product complexity will make the need for production coordinating devices even more accurate. However, we are not only in an era of rising product complexity but also in an era of fierce competition which definitely calls for adequate cost-saving tools. For this certainly MRP is not enough, as its basic philosophy is only to ensure that the right number of components is planned at the right time to meet the demand for end items. MRP therefore only provides a feasible solution to the multi-level production inventory problem, whereas ideally one would aim at a sequence of replenishment quantities through time at the various levels of manufacturing that keeps the total...
relevant cost as low as possible while satisfying the demand for end items. Therefore, determining a proper lot-sizing policy definitely is a key dimension of inventory control, as placing proper batches can allow for significant reductions in inventory-related costs.

Optimal solution algorithms exist for this problem [1], but only very small instances can be solved in reasonable computation time for the problem is NP-hard, not mentioning the mathematical complexity of the technique that might deter many potential users. Several approaches to solve variants of the MLLS problem have been developed, with further assumptions made on the product and/or cost structure (see [2–5]), but execution times remain desperately high. Last, it should also be added that even when the time constraint is made as slack as possible, branch-and-bound algorithms available from standard software packages sometimes fail in finding optimal solutions. Hence heuristic techniques that offer a reasonable trade-off between optimality and computational feasibility are highly advisable.

One alternative, which is often implemented in practice, consists in applying single-level decision rules – just like the economic order quantity – to each level of the product structure (see [6,7]). Though simplicity surely obtains, neglecting the fact that placing a lot for an item somewhere in the product structure often triggers lots for the subcomponents of this item has dramatic consequences in terms of cost effectiveness. Of particular interest are the approaches in which the multi-level nature of the problem is explicitly taken into account. Blackburn and Millen [8] suggested several cost modifications to account for interdependencies among levels of the product structure. Coleman and McKnew [9] developed a four-pass procedure based on the incremental part period algorithm (IPPA). The procedure embeds an original look-down routine used to compare at each level the net benefit resulting from the lumping of each period’s requirement, until the bottom of the product structure is reached. Contrary to both previous approaches, the method developed by Bookbinder and Koch [10] is not only designed to address pure assembly product structures but also general structures, a feature being extremely common in real settings. Dellaert and Jeunet [11] resort to randomization as a means of accounting for interdependencies among stages and achieve fairly good results compared to the previous techniques. However, although these approaches usually outperform sequential methods, they are unable to guarantee an optimal solution.

In this paper, we develop a hybrid genetic algorithm (GA) to solve the MLLS problem with no capacity constraints and no restrictive assumption on the product structure. Our primary incentive for this study is to find a solution method which is relatively moderate in CPU-time and intuitively appealing to potential users for a problem field of which Segerstedt [12] says ‘MRP and other methods without clear capacity constraints will no doubt continue to be used in practical installations for decades to come’. We consider the most general statement of the problem in which costs may vary from one time period to the next. Though the possibility of allowing time-varying costs could be considered a striking assumption, it should be recalled that the cost of carrying items in inventory includes the expenses incurred in running a warehouse, the costs associated with special storage requirements, deterioration, obsolescence and taxes, and primarily the opportunity cost of the money invested which is very likely to fluctuate as a result of changes in investment opportunities. Similarly, the set-up cost attached to replenishment decisions embeds learning effects (getting used to a new set-up, procedures and material has a cost in terms of scrap costs) and evolves in response to changes in the work force, especially when it is subject to frequent turnover.

Our strategy here has been to design specific genetic operators that constrain search to the set of feasible solutions rather than letting the algorithm explore any possibility and relying on sophisticated penalty schemes. To increase search-efficiency, we have further reduced the set of solutions to the MLLS problem by defining adequate bounds on the ordering periods. We first tested our technique against optimality, and then moved to larger problems for which only heuristic methods can be employed. When small instances are considered, the GA almost instantaneously (1 second on average) provides solution of very high quality.
Larger instances confirm the performance of the GA, as it easily beats the sequential techniques we incorporated for the sake of comparison while keeping the computational demand extremely reasonable.

The paper is organized as follows. Section 2 is dedicated to the presentation and mathematical formulation of the MLLS problem. Section 3 gives the building blocks of the genetic algorithm: encoding, feasibility constraints, genetic operators and principles of evolution. Section 4 presents the experimental framework and the parameter settings. Numerical results are discussed in Section 5 and in Section 6 conclusions and practical implications are derived.

2. The multi-level lot-sizing problem

In a manufacturing production system, end items are usually made up with a number of intermediate products which, in turn, consist in combinations of components (purchased parts and raw materials). Each end item is therefore described by a bill of materials, which is the product recipe. When considering the issue of satisfying the demand for end items emanating from customers, the right quantity of each sub-component has to be made available at the right time, and if possible at the lowest cost. As products are associated with holding and set-up costs, different inventory policies lead to different costs and determining an optimal policy is a core concern.

The bill of materials is commonly depicted as a directed acyclic graph in which a node corresponds to an item and the edge (i, j) between nodes i and j exists if and only if item i is directly required to assemble item j. Item i is fully defined by \( \Gamma^{-1}(i) \) and \( \Gamma(i) \), the sets of its immediate predecessors and successors. The set of ancestors – immediate and non-immediate predecessors – of item i is denoted \( \hat{\Gamma}^{-1}(i) \). Items are numbered in topological order by the integers 1, ..., \( n \) so as to guarantee that any edge (i, j) satisfies \( j > i \). Put another way, items are sorted in increasing level codes and each common part is listed at the lowest level it appears in the product structure. Adopting the convention that finished goods belong to level 0, items are then numbered from 1 to \( P \), starting the labeling from level 0 and moving sequentially to the lowest level. Hence, product 1 is always a finished good whereas item \( P \) is necessarily purchased.

Fig. 1 displays three types of product structures that are often encountered in the literature. Fig. 1(a) represents a pure assembly structure, in which every item has at most one direct successor (\( \# \Gamma(i) \leq 1 \), for all \( i \)). By contrast, Fig. 1(b) shows a pure arborescent structure in which every item has at most one direct predecessor (\( \# \Gamma^{-1}(i) \leq 1 \), for all \( i \)). Fig. 1(c) exhibits a general product structure for which \( \Gamma^{-1}(1) = \{2, 3\} \); \( \Gamma^{-1}(1) = \{2, 3, 4, 5\} \); \( \Gamma(4) = \{2, 3\} \) and \( \hat{\Gamma}(4) = \{1, 2, 3\} \). In this last structure, item 4 is commonly used by item 2 and 3, hence it is said to be a common part.

The MLLS problem consists in finding a sequence of lot sizes that minimizes the sum of set-up and inventory carrying costs, while meeting the demand for end items over a \( T \)-period planning horizon. To formulate the MLLS problem as a mixed integer program we use the symbols summarized in Table 1.

Our formulation follows that of Steinberg and Napier [1]. An optimal solution to the MLLS problem is obtained by minimizing

\[
\sum_{i=1}^{P} \sum_{t=1}^{T} e_{i,t} x_{i,t} + s_{i,t} y_{i,t} + h_{i,t} I_{i,t} \tag{1}
\]

subject to the set of constraints

\[
I_{i,t} = I_{i,t-1} + r_{i,t} + x_{i,t} - d_{i,t}, \tag{2}
\]

\[
d_{i,t} = \sum_{j \in \Gamma(i)} c_{i,j} x_{j,t} + l_{i}, \quad \forall i | \Gamma(i) \neq \emptyset, \tag{3}
\]

\[
x_{i,t} - M y_{i,t} \leq 0, \quad y_{i,t} \in \{0, 1\}, \tag{4}
\]

\[
I_{i,t} \geq 0, \quad x_{i,t} \geq 0. \tag{5}
\]
Table 1
Notations for the MLLS problem

<table>
<thead>
<tr>
<th>Cost parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_{i,t} )</td>
<td>Set-up cost for item ( i ) in period ( t )</td>
</tr>
<tr>
<td>( e_{i,t} )</td>
<td>Unit purchase or production cost for item ( i ) in period ( t )</td>
</tr>
<tr>
<td>( h_{i,t} )</td>
<td>Unit inventory carrying cost per period for item ( i ) in period ( t )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quantity variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_{i,t} )</td>
<td>Gross requirements for item ( i ) in period ( t )</td>
</tr>
<tr>
<td>( x_{i,t} )</td>
<td>Delivered quantity of item ( i ) at the beginning of period ( t )</td>
</tr>
<tr>
<td>( I_{i,t} )</td>
<td>Level of inventory for item ( i ) at the end of period ( t )</td>
</tr>
<tr>
<td>( r_{i,t} )</td>
<td>Scheduled receipts for item ( i ) at the beginning of period ( t )</td>
</tr>
<tr>
<td>( y_{i,t} )</td>
<td>Boolean variable addressed to capture ( i )'s set-up cost in period ( t )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Technical coefficients</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l_i )</td>
<td>Lead time for item ( i )</td>
</tr>
<tr>
<td>( c_{i,j} )</td>
<td>Quantity of item ( i ) required to produce one unit of item ( j ) (production ratio)</td>
</tr>
</tbody>
</table>

The objective function in Eq. (1) is the sum of purchase or production costs, set-up and inventory holding costs for all items over the planning horizon. Note that the possibility of time-varying unit purchase and production costs, inventory costs and set-up costs is allowed for. Eq. (2) expresses the flow conservation constraint for item \( i \). It defines the inventory level for item \( i \) at the end of period \( t \). The variable \( I_{i,0} \) is the initial level of inventory and \( r_{i,t} \) designates the scheduled receipts, which result from previously made ordering decisions and represent a source of the item to meet gross requirements. Gross requirements consist in the external demand when end items are considered, and result from the lot sizes of immediate successors for component items, as stated in Eq. (3). We assume that no component is sold to an outside buyer, i.e. external demands only exist for finished goods. Constraint (4), where \( M \) is a large number, guarantees that a set-up cost will be incurred when a batch is purchased or produced. Finally, constraint (5) states that backlog is not allowed and that production is either positive or zero.

3. The genetic algorithm

In this section, we present the way a genetic algorithm can be customized to address the MLLS problem. We first discuss the issue of encoding – in our case a binary matrix representation – and the feasibility constraints. We then turn to the evolutionary stages of the algorithm and the specific genetic operators that have been designed to increase search efficiency.

3.1. A binary matrix representation

The first issue that arises in designing a hybrid genetic algorithm is that of finding a proper encoding. At this stage, we shall exploit a fundamental property of the MLLS problem that allows for a binary encoding. In the MLLS problem, costs are typically concave as a non-zero fixed cost is incurred whenever an order is launched (the set-up cost). Hence, though it might be optimal to place an order at a point in time where there are no net requirements, there is never more than one lot to cover a net requirement. Ordering more than once would imply extra costs (a formal proof is in Veinott [13]). Clearly if this was not the case, only real-coded or floating-point genes could be employed, as quantities in each time period would have to be specified. Hence, concavity implies that there is an optimal solution for which

\[
x_{i,t} \cdot I_{i,t-1} = 0 \quad \text{for all } i \text{ and } t.
\]
Knowing the list of delivery dates is therefore sufficient to derive ordered quantities. This leads to the most natural encoding which consists in representing the list of delivery dates for all items as a $P \times T$ matrix

$$y = \begin{pmatrix} y_{1,1} & \cdots & y_{1,T} \\ \vdots & \ddots & \vdots \\ y_{P,1} & \cdots & y_{P,T} \end{pmatrix},$$

in which $y_{i,t} = 1$ if a set-up for item $i$ is delivered in period $t$, and $y_{i,t} = 0$ otherwise. Searching for an optimal solution to the MLLS problem is therefore equivalent to finding a binary matrix $y$ such that the corresponding quantities minimize the objective function (1). The vector $y_i = (y_{i,1}, \ldots, y_{i,T})$ coding order delivery dates for the single item $i$ is called a string. Thus, in our setting, a chromosome $y$ is a set of binary strings coding delivery dates for all items and periods.

For any chromosome, a decoding procedure is used to convert ordering dates into lot sizes, going sequentially from finished goods to purchased items. This sequential procedure is based on the gross-to-net explosion process, a key element in MRP systems which translates product requirements into component part requirements, taking existing inventories and scheduled receipts into account. Starting from the gross requirements (external demand) for the end item, physical inventory is computed in each period. Physical inventory at the end of any period amounts to the inventory at the end of the previous period augmented with the scheduled receipts and reduced by the gross requirements. When this physical inventory falls below zero, we compute the net requirements, which are exactly what is needed to meet the gross requirements. Lot sizes are such that they cover the net requirements. These lot sizes together with the technical coefficients define the gross requirements for any immediate component of the end item. These steps are summarized in Fig. 2.

As a brief illustration of how the decoding works, consider a stream of gross requirements $d_i = (10, 15, 100, 20, 20)$, scheduled receipts $r_i = (5, 0, 0, 0, 0)$ and an initial level of inventory $I_{i,0} = 12$. We have $z_i = (7, -8, -108, -128, -148)$, hence net requirements equal $b_i = (0, 8, 100, 20, 20)$, and the string $y_i = (1, 0, 1, 0, 0)$ finally yields $x_i = (8, 0, 140, 0, 0)$.

3.2. Feasibility

The check for feasibility follows a logic that is similar to that of the decoding procedure, starting from the end item and sequentially moving to lower levels. A chromosome is said to be feasible if and only if, for all $i = 1, \ldots, P$, the next two conditions are met:

(i) All net requirements for item $i$ can be covered in time. This constraint refers to the possibility that any successor of item $i$ can generate net requirements for item $i$ that simply cannot be covered, due to either item $i$’s lead time or item $i$’s available quantities at the beginning of the horizon.

(ii) All net requirements for item $i$ are covered by proper set-ups. This constraint has to do with the consistency of the delivery dates $y_{i,t}$ with the stream of net requirements for item $i$.

To formally state condition (i), let $z_i$ denote the first possible order delivery date for item $i$. This date can easily be computed from the initial inventories,

for $i = 1$ to $P$
begin
if $\Gamma (i) \neq \emptyset$ (item $i$ is not a finished good)
then compute gross requirements ($d_{i,t}$, equation 3)
compute physical inventory $z_{i,t} = z_{i,t-1} + r_{i,t} - d_{i,t}$, with $z_{i,0} = I_{i,0}$
compute net requirements $b_{i,t} = \max (0, d_{i,t} - r_{i,t} - \max \{0, z_{i,t-1}\})$
compute lot sizes $x_{i,t} = b_{i,t} + \cdots + b_{i,\tau-1}$, $\tau > t$, with $y_{i,t} = y_{i,\tau} = 1$ and $y_{i,t+1} = \cdots = y_{i,\tau-1} = 0$
end

Fig. 2. The decoding procedure.
scheduled receipts and lead times of item $i$’s components, a procedure whose presentation will be omitted for the sake of simplicity. Clearly, no order can take place before $x_i$, and thus no net requirements should appear prior to $x_i$. This is written as

\[ \sum_{t=1}^{x_i-1} b_{i,t} = 0. \]  

(7)

To illustrate a case in which condition (i) fails, consider a finished good (item 1) made from one single component (item 2) with a one-to-one production ratio. Consider the feasibility of chromosome $y = (y_1, y_2)$ displayed in Table 2, in which we also give the net requirements for item 1, the lot sizes resulting from string $y_1$ and the requirements for item 2 computed from the lot sizes of item 1.

As can be seen in Table 2, string $y_1$ implies a net requirement of 27 units for item 2 in period 1 but this requirement cannot be covered due to the lead time of item 2. The first period in which it is possible to receive ordered quantities of item 2 is $x_2 = 2$ if an order is launched in period 1. Thus, the possibility of lumping the first three net requirements of item 1 into a lot in period 3 must be abandoned. Any chromosome embedding the string $y_1$ will systematically be infeasible.

Let us now turn to the formal statement of condition (ii). We shall first define the sequence $\{t_{i,w}; w = 1, \ldots, W_i\}$ of strictly positive net requirements dates for item $i = 1, \ldots, P$. Consider now string $y_i$. As the first net requirement for item $i$ has to be covered, there must be at least one set-up in the time interval between $x_i$ and $t_{i,1}$, where $t_{i,1}$ denotes the first period in which a positive net requirement appears for item $i$. Due to the concavity assumption, we also know that it can never be optimal to cover a net requirement with more than one set-up. Hence between period $x_i$ and period $t_{i,1}$ there can be at most one set-up, which implies that there must be exactly one set-up between $x_i$ and $t_{i,1}$, i.e.

\[ \sum_{t=x_i}^{t_{i,1}} y_{i,t} = 1. \]  

(8)

The concavity assumption actually implies that there cannot be more than one set up between any pair of dates $t_{i,w} + 1$ and $t_{i,w+1}$, that is to say between any pair of consecutive requirements. But there might well be no set-ups at all between $t_{i,w} + 1$ and $t_{i,w+1}$, as net requirements in period $t_{i,w+1}$ can be covered by a set-up anywhere between that period and the first period in which a positive net requirement appears, and we know for sure that a set-up is necessarily ordered between $x_i$ and $t_{i,1}$. Summarizing, this constraint is written as

\[ \sum_{t=t_{i,w}+1}^{t_{i,w+1}} y_{i,t} \leq 1 \quad \text{for all } w = 1, \ldots, W_i. \]  

(9)

The end of the planning horizon has a slightly different status, as between $t_{i,W_i} + 1$ and $T$ the item is not required anymore, which implies that

\[ \sum_{t=t_{i,W_i}+1}^{T} y_{i,t} = 0. \]  

(10)
To illustrate condition (ii) consider again Table 2. Satisfaction of constraints (8)-(10) can easily be checked for item 1, with \( z_1 = 3 \) and \( \{ t_{1,w}; w = 1, \ldots , W_1 \} = \{ 3, 4, 5, 6, 7 \} \). Hence, though string \( y_1 \) meets condition (ii) the chromosome \( y = (y_1,y_2) \) is not feasible.

In order to further reduce the solution set, note that the last period in which an item is required might differ from the last period of the planning horizon, and symmetrically the first possible period of production for any item usually does not coincide with the first period of the horizon. This is ignored by the general MILP formulation of the MLLS problem but allows for substantial reductions of the set of solutions. The option of releasing an order for purchased items in the first period must be considered (even if initial inventory levels cover the first net requirements) where costs are time-varying. Thus delivery of materials can at best start in period \( 1 + l_i \). Optimal delivery dates for purchased item \( i \) belong to \( \{ 1 + l_i, \ldots , \beta_i \} \), where \( \beta_i \) is the last period in which item \( i \) is required.\(^1\) As for a manufactured item, production can be initiated when all predecessors are simultaneously available. If we let \( \varphi_j \) denote the smallest \( t \) such as \( I_{t_j} > 0 \) with \( j \in I^{-1}(i) \), then at best \( i \) is delivered in period \( l_i + \max_{j \in I^{-1}(i)} \{ \varphi_j \} \).\(^2\) To illustrate, consider product 1 in Fig. 1(c), where \( I^{-1}(1) = \{ 2, 3 \} \). Suppose that \( \varphi_2 = 2 \), \( \varphi_3 = 4 \) and \( l_1 = 1 \). Obviously, production of item 1 cannot be launched in period 2 since at this date no inventory of predecessor 3 is available. Thus, production is released in period 4 \( (4 = \max(2, 4)) \) and becomes available at date \( 4 + l_1 = 5 \). Hence, optimal delivery dates for manufactured item \( i \) belong to the set

\[
\{ l_i + \max_{j \in I^{-1}(i)} \{ \varphi_j \}, \ldots , \beta_i \}.
\]

In our genetic algorithm, only feasible chromosomes will be considered. Each chromosome will have to pass the feasibility test before entering the population. By doing so, we avoid the complication of defining sophisticated penalty functions to guide search and storing useless infeasible chromosomes in the population.

### 3.3. Genetic search operators

As there is no natural definition of genetic operators for the MLLS problem with the binary representation we have adopted, we have designed five problem-specific search operators. When an operator is applied to a feasible chromosome, it can happen that the altered chromosome is not feasible anymore. A simple repairing procedure is then applied to make the chromosome anew feasible, by implementing adequate changes so that each string in the altered chromosome satisfies constraints (8)-(10). In case infeasibility arises from the violation of constraint (7), the application of the operator is blindly cancelled. An example of how each operator works is given in the appendix.

#### 3.3.1. Single-bit mutation

Mutation is usually defined as a change in a (random) number of bits in a chromosome. In our single-bit mutation, a unique bit undergoes mutation. This amounts to considering each chromosome with given probability, selecting a random pair \((i^*,t^*)\) such that \( i^* \in \{ 1, \ldots , P \} \) and \( t^* \in \{ z_{r^*}, \ldots , t_{r^*} \} \), and changing its value from 0 to 1 or the opposite. This, in turn, can possibly trigger a series of changes in different places in the chromosome in order to maintain its feasibility.

The mutated string should satisfy constraints (8)-(10). If mutation takes place in \( \{ z_{r^*}, \ldots , t_{r^*} \} \), a symmetric mutation has to be performed in this same interval so as to ensure that there is exactly one set-up between \( z_{r^*} \) and \( t_{r^*} \) as stated in constraint 8. Consider now \( t^* \in \{ t_{i^*,w} + 1, \ldots , t_{i^*,w+1} \} \), for \( w = 1, \ldots , W_{t^*} - 1 \). When \( y_{r^*,t^*} = 0 \), there are two possibilities. If there is already a bit equal to 1 in this interval, it is swapped with \( y_{r^*,t^*} \); otherwise, \( y_{r^*,r^*} \) can be set to 0 without violating the feasibility constraint (9). When \( y_{r^*,t^*} = 1 \), we know that there is no other set-up in this interval and the bit can be set to 0. We finally examine \( i^* \)'s ancestors to see whether the altered chromosome is feasible, applying the repairing procedure when necessary.

\(^1\) Actually, it is enough to consider the lot-for-lot solution, for any other lumping would generate a last positive net requirement prior to \( \beta_i \).

\(^2\) The stream of dates \( \{ \varphi_j \} \) obtains starting from the bottom of the product structure and proceeding from one level to the next, until level 0 is reached.
Violation of constraint (7) for any ancestor entails the selection of a new random pair \((i^p, t^p)\).

### 3.3.3. Inversion

We implement the following inversion. A random pair \((i^p, t^p)\) is selected and \(y_{i^p, t^p}\) is compared to its neighbors \(y_{i^p, t^p-1}\) and \(y_{i^p, t^p+1}\). When \(y_{i^p, t^p}\) and exactly one of its neighbors differ, we exchange their values. When both neighbors are different from \(y_{i^p, t^p}\), one of them is randomly selected for exchange. The same inversion is performed on item \(i^w\)'s predecessors. We finally apply the repairing procedure to the altered chromosome.

### 3.3.4. Period crossover

The classical crossover is a one-point crossover that combines two chromosomes on the basis of one cross site randomly selected. Genetic material is swapped between two chromosomes to produce a pair of offsprings. Our crossover is a period crossover that chooses a point in time \(t^p\) randomly selected in the planning horizon. To produce offsprings, we combine the first periods of one parent’s strings with the last periods of the second parent’s strings, the appropriate corrections being made when cumulative lead times have to be incorporated (more details are given in the appendix). Again, we apply the repairing procedure when necessary.

### 3.3.5. Product crossover

In the product crossover operator, the crossing point is an item chosen at random. To illustrate the functioning of this specific crossover, let us consider the product structure in Fig. 3 and assume the crossing point \(i^p\) is item 5. We can replace string \(y_5\) by \(y'_5\) in parent \(y\) only if \(y'_5\) fits feasibility conditions (8)–(10) (see Section 3.2), given the stream of positive net requirements dates \(\{t_{5,w}\}\) in parent \(y\).

Assume we keep \(y_4\) unchanged whereas \(y_5\) is replaced by \(y'_5\). Strings \((y_4, y'_5)\) will generate new requirements for item 7 for which there is little chance that \(y_7\) still fits conditions (8)–(10). As items 4 and 5 have an ancestor in common (item 7), it might be wise not only to replace \(y_5\) by \(y'_5\) but also \((y_4, y_6, y_7)\) by \((y'_4, y'_6, y'_7)\) when \(y_4\) meets the feasibility conditions when dates \(\{t_{4,w}\}\) are considered.

To produce an offspring, crossing item \(i^p\) as well as all items with which it shares at least one ancestor must have strings \(y'_p\) that meet the feasibility conditions when dates \(\{t_{w}\}\) in parent \(y\) are considered. The corresponding streams from parent \(y\) then replace the original strings in parent \(y\). A second offspring can be produced the same way, provided the symmetric condition is satisfied. The crossover procedure ends with a check of feasibility condition (7).\(^3\)

### 3.4. Evolution: Selecting, searching and stopping

The initial population \(\mathcal{P}\) is generated by repeated application of the single mutation operator to the chromosome coding delivery dates for the lot-for-lot policy, a replenishment rule which consists in placing an order each time a net requirement appears. The evaluation of chromosomes is achieved through the computation of the associated cost. Once the decoding procedure has been applied to each chromosome in the population, values \(I_{t_i}\) of ending inventory are computed for all items and periods and the total cost \(C(y)\) resulting from

\(^3\)It should be noted that in the numerical experiments we conducted, illegal offsprings never appeared.
from the delivery dates given by chromosome $y$ is derived (according to Eq. (1)). On the basis of this ‘anti-fitness’ evaluation function, we build the probability for an individual to become a member of the next generation. Once the cost $C(y^*)$ associated with chromosome $y^*$ has been calculated, the probability for individual $y^*$ to be a member of the new population is

$$\frac{\exp\left[-\lambda(C(y^*)-C_{\text{min}})/(C-C_{\text{min}})\right]}{\sum_{y \in \mathcal{P}} \exp\left[-\lambda(C(y)-C_{\text{min}})/(C-C_{\text{min}})\right]}$$

(11)

where $\lambda$ is a positive constant, $C_{\text{min}} = \min_{y \in \mathcal{P}} C(y)$ is the lowest cost in the current population and $C = \sum_{y \in \mathcal{P}} C(y)/|\mathcal{P}|$ is the population average cost. The re-scaling that $C$ and $C_{\text{min}}$ operate on costs permits to focus on relative cost differences rather than absolute ones, hence avoiding extra parameter tuning when moving from one problem to another. The logistic formulation in expression (11) is inspired from a type of strategy used to assign lifetime values for chromosomes in varying population size (see for instance [14]). Chromosomes of minimum cost have the largest probability to belong to the next generation, and the larger $\lambda$ is the more likely it is that they actually do so – hence the stronger the selective pressure is. We add a small amount of elitism to our GA by always keeping the best individual from one generation to the next.

Evolution then works as follows. The search operators are sequentially activated with given probabilities on the current population to produce recombined individuals which found the next generation. Selection then operates on the current population and the chosen chromosomes join the genetically modified ones to form the new generation. The rest of the evolution is cyclic repetition of the previous steps.

Two categories of termination conditions generally exist, respectively based on the structure of the chromosomes (search stops if chromosomes become identical) and on the progress made by the algorithm in a predefined number of generations (if the improvement is smaller than some epsilon, search is terminated). In the problem we consider, convergence towards a single chromosome cannot be guaranteed, for the optimal solution may not be unique. Consequently, we use this second stopping rule, denoted GA*$n$, in which search stops if no significant improvement is recorded during the last $n$ generations. As this rule hardly offers any control on computation time, we also consider the rule GA*N in which search is terminated after a fixed number of generations $N$.

4. Experimental design

The purpose of the following experiments is first to test the algorithm against optimality in small-sized MLLS problems. In order to derive optimal solutions, we have used Steinberg and Napier’s formulation and tried to solve the resultant mixed integer linear program with the general algebraic modeling system (GAMS), a standard optimization package. We then performed simulations for medium-sized instances to assess the cost performance of the GA.

4.1. The test problems

Two sets of experiments were considered, in which we first compared the performance of the GA to solutions provided by GAMS for small instances (10 products over 12 periods) and then examined the performance of the algorithm for medium-sized product structures (involving 50 products) over extended planning horizons. In both phases, product structures were defined in terms of complexity. We used the complexity index proposed by Kimms [15] which is defined in the following way. Recall products are numbered in topological order by the integers $1,\ldots,P$ and let $P(k)$ be the number of products at level $k$, with $k = 0,\ldots,K$ ($K+1$ is the depth of the structure). The total number of items obviously equals $\sum_{k=0}^{K} P(k)$, which by definition is also $P$. The most complex – in the sense of having the largest number of product interdependencies – structure is obtained when each item enters the composition of all the items located at higher levels in the product structure. By contrast, the simplest structure obtains when each item enters the composition of exactly one item belonging to a higher level. Kimms [15] defines the complexity of a product structure as

$$C = \frac{A - A_{\text{min}}}{A_{\text{max}} - A_{\text{min}}}.$$
where \( A = \sum_{i=1}^{P} \# \Gamma(i) \) is the actual number of arcs in the structure. There is of course a minimal number of arcs such as the product structure is connected, which we denote \( A_{\text{min}} \) and is equal to \( P - P(0) \). Conversely there is a maximum number of arcs denoted \( A_{\text{max}} \) that the graph can contain, and which is written as

\[
A_{\text{max}} = \sum_{k=0}^{K-1} \{ P(k) \cdot \sum_{j=k+1}^{K} P(j) \}.
\]

Structures for which the number of arcs equals the minimum number of arcs \( A = A_{\text{min}} \) are necessarily assembly structures with a zero \( C \)-value, whereas structures such as \( A = A_{\text{max}} \) satisfy \( C = 1 \). The \( C \)-index is therefore bounded from below and above, whereas the traditional index of Collier [16] is not.

4.1.1. First phase: Test against optimality

Obtaining optimal solutions within reasonable CPU is a pretty difficult task, even for small instances. The first test phase therefore only involved 10-item product structures over a 12-period planning horizon. We controlled the complexity index \( C \) so that it took values within the set \( \{0.00, 0.25, 0.50, 0.75\} \). For the sake of clarity, we have set the number of end-items \( P(0) \) to one, although the GA can easily handle product structures in which there are multiple finished goods. Without loss of generality we assumed a one-to-one production ratio in each case and set the depth of the structure \((K + 1 \text{ in our formalism) to four levels throughout, a very reasonable value for 10-item structures. Demand in each period for the end item was randomly drawn from a uniform distribution ranging from 0 to 200. Similarly, initial inventories for all items were randomly chosen in the range \([0, 200]\) and scheduled receipts were set to zero for all items in all periods, again without loss of generality. Cost parameters were chosen so as to guarantee increasing set-up costs and decreasing holding costs along any branch of the product structure tree. This is in accordance with the standard assumptions of value-added holding costs and highest set-up costs for raw materials needing major transformations versus lowest-order costs for end items only requiring assembly. An order cost in the range \([45, 55]\) and a holding cost varying from 0.04 and 0.07 were assumed for all purchased items, in each period. For all other items, costs satisfy

\[
s_{i,t} = 0.5 \min_{j \in \Gamma^{-1}(i)} (s_{j,t}) \quad \text{with} \quad \Gamma^{-1}(i) \neq \emptyset,
\]

\[
h_{i,t} = 2 \sum_{j \in \Gamma^{-1}(i)} (h_{j,t}) \quad \text{with} \quad \Gamma^{-1}(i) \neq \emptyset.
\]

Finally, lead times were randomly chosen and set to either 0 or 1. For each value of the commonality index, we performed five replications of costs and demands, hence a total number of \( 4 \times 5 = 20 \) problems was examined in the first phase.

4.1.2. Second phase: Performance evaluation for medium-sized problems

The second phase of the experiment involved larger product structures embedding 50 items and longer planning horizons (24 and 36 periods were considered). Again we set \( C \in \{0.00, 0.25, 0.50, 0.75\} \). We considered a unique end-item together with a one-to-one production ratio and a structure depth of 10 levels, a reasonable value given the number of components. Carrying costs were computed in terms of random echelon cost uniformly drawn over \([0.2, 4]\), as in Afentakis and Gavish [17]). Set-up costs for each item in each period were randomly selected from a uniform distribution over \([100, 120]\). For each value of the commonality index set-up costs were then multiplied by a scaling factor so as to avoid the lot-for-lot and the unique-lot solutions. Demand in each period uniformly selected over \([0, 180]\). The planning horizon was set to 24 and 36 periods. All lead times were randomly set to either 0 or 1 period. We generated 5 tests (replications of costs and demands) for each value of the complexity index \( C \) and planning horizon, hence a total of \( 4 \times 2 \times 5 = 40 \) cases was considered in the second phase.

4.2. Parameters setting of the genetic algorithm and other lot-sizing rules

The genetic algorithm requires a number of parameters to be specified. We kept the population size equal to 50 chromosomes in all experiments, as it is

\footnote{We did not consider a \( C \)-value of 1 as we believed the corresponding product structures would be too intricate to be realistic.}
generally acknowledged (see [18]) that further increases in the population size do not necessarily lead to significant improvements. The values of the number of generations for each stopping rule were based on intuitive and practical considerations (including numerous test runs).

In all experiments, we reported the results of rule GA0 (the initial population) in order to highlight the improvements reached within a positive number of generations. The GA generally showed strong signs of cost convergence after 100 generations in the smallest instances we considered. For larger problems we kept track of the performance of GA200 and GA300, and of course that of GA*50 which turned out to achieve the best overall results. Each of the five operators was applied with a constant probability equal to 0.1, and we arbitrarily set the selection parameter $\lambda$ in Eq. (11) to 25, which corresponds to a fairly strong selective pressure.

Though few heuristic methods have been designed for time-varying cost parameters, we had to incorporate some for the sake of comparison. We thus reported the lot-for-lot solution (L4L) that does not require any specific cost assumption to be implemented. We also considered the ‘best item ordering cycle’ (BIOC) that consists in ordering each item every $\theta$ period, where $\theta$ is the ordering cycle that leads to the minimum cost amongst costs associated with the various possible values of the ordering cycle (from one period to $T$ periods). Note that $\theta$ obviously varies from one component to another.

5. Results

This section is devoted to the presentation and discussion of experimental data produced in the two test phases.

5.1. Small instances

Table 3 displays the results of the first test phase for each C-value. Techniques are sorted in increasing cost order, the cost index being the ratio of the solution provided by each technique over the solution provided by GA*50, which was used as a benchmark. Whenever a GA was used, we ran the experiment 20 times in order to obtain statistically robust conclusions and reported the average value of the lowest cost individual over the 20 runs.

It should first be said that GAMS proved unable to find any optimal solution. Even when the constraints on resources and iterations were made as slack as possible, 8 hours were not enough for GAMS to reach an optimum. Hence, the strategy we adopted was to limit the execution time of the branch-and-bound algorithm to 5 minutes, therefore gaining some control over the length of the simulation experiment. The results reported in Table 3 are solutions provided by GAMS after exactly a 5 minutes run time. The best overall performance is achieved by GA*50, the GA for which search stops when no improvement is reached within 50 consecutive generations. Rule GA*50 is closely followed by GA100, which provided solutions that remain within a 1% distance from the best ones (0.26% on average). GAMS offers solutions of a much poorer
That sometimes the cost ratio of GAMS to GA\(_{50}\) is lower than 100 should be no surprise. The random generation of the initial population together with the particular sequence of random events shaping the search process create variability in the performance of the GA which sometimes keep it away from the optimal solution. There is however clear evidence that the GA is never far away.

The sequential BIOC algorithm stands out as a good compromise achieving a decent performance (7.7% excess cost on average) at the expense of a negligible demand in computer time. Needless to say the lot-for-lot policy is outperformed by all methods, with an excess cost rate approaching 36%. This is worse than the outcome of pure random search in the initial population GA0, which produces excess cost rates of about 28% (hence the magnitude of improvement).

Note how efficiently GA\(_{50}\) and GA100 have improved initial solution quality through selection and application of the five genetic operators. The fact that GA\(_{50}\) outperforms GA100 is simply due to a longer search, for exploration goes on as long as it keeps paying off.

Execution times are not given in detail in Table 3, for they were extremely low. The lot-for-lot policy and GA0 have negligible requirements which never exceeded 0.005 seconds. The average CPU per replication amounts to 1.11 seconds for GA\(_{50}\) (with standard deviation 0.22 seconds) and 0.72 seconds for GA100 (with standard deviation 0.03 seconds). (The longer execution times associated with GA\(_{50}\) originate in the larger number of generations that GA\(_{50}\) performs, compared to GA100.) By contrast, GAMS reached solutions within the 5 minutes delay whose optimality was never guaranteed. We therefore end up with completion times that, from a practical standpoint, are extremely reasonable and remain far away from the time spans required by GAMS for achieving low-quality solutions.

In order to get some insights into the workings of the GA, we also kept track of the performance of the five genetic operators involved in this study. For each operator we recorded the number of fitness improvements. We then computed the relative contribution of each operator to the total number of improvements. Table 4 gives the success rates (in percent) obtained with GA100 and GA\(_{50}\) in all test cases.

From Table 4 it is clear that search is not performed with equal efficiency by all operators. The three mutation-based operators stand out in terms of successful search intensity whereas both crossover exhibit lower rates. The reason for this is twofold. First, our crossovers participate in the process of variety creation, which is a key dimension of evolutionary search. In the early stages of the process, they perform global search and generate chromosomes that local search operators are likely to improve upon. Second, as the population converges, crossover tends to recombine identical chromosomes, which surely produces no improvement. By contrast, local search operators necessarily provoke changes, and thus have a chance to improve.

Before turning to larger cases, note that although standard optimization techniques can theoretically be employed to find an optimal solution to the MLLS problem, even in small-sized cases does the GA show its virtue, providing a quicker and better solution than GAMS. Supposedly, the tendency of the GA to outperform other techniques should be confirmed when larger instances are considered.

### 5.2. Medium-sized problems

Table 5 summarizes the results of the second test phase. Techniques are arrayed by increasing cost order, the cost index being the ratio of the cost over the solution provided by GA\(_{50}\), which was used as a benchmark. Each GA was replicated 10 times and the average minimum values are reported in Table 5.

The results of the second test phase confirm the quality of GA-based techniques. Compared to the first phase, convergence is much slower and far from sufficient when only a small number of generations (100 or 200 generations) is considered. The genetic algorithm consistently improves with the number of generations, GA\(_{50}\) reaching a 37% improvement over GA100 for the 24-period horizon and an improvement of 51% for 36 periods. Obviously, when the problem size is increased the

---

5 That sometimes the cost ratio of GAMS to GA\(_{50}\) is lower than 100 should be no surprise. The random generation of the initial population together with the particular sequence of random events shaping the search process create variability in the performance of the GA which sometimes keep it away from the optimal solution. There is however clear evidence that the GA is never far away.
number of generations needed to reach a satisfactory outcome also rises. As expected, controlling the number of consecutive improvements leads to the best overall performances in spite of an additional computational burden. Table 6 displays the time requirements of the GA at its various stages.

We choose not to show the execution times of BIOC and of the L4L policy, as they remained steadily below 0.01 second in all the 40 test cases. By contrast, GAs are more greedy and CPUs increase with the number of generations, the maximum value being attached to GAH50, the solution which also maximizes cost-efficiency. Execution times are all increasing functions of the complexity index and the length of the planning horizon. Still, with CPUs that are between half a minute and three minutes, GAs produce remarkably cost-efficient solutions while formulating very reasonable computational demands. Although we chose not to report them here, it is worth mentioning that the success rates of the operators were approximately the same as in small instances.

We tested for factor influence using several non-parametric procedures at the 1% significance level. Tests were performed on the samples of cost ratios benchmarked against the solution provided by BIOC, so as to focus on changes in the ability of the GA to outperform a standard heuristic procedure. The Kruskal–Wallis non-parametric ANOVA was used to appraise the influence of the planning horizon.

Table 4
Relative rate of success for the five genetic operators in the GA

<table>
<thead>
<tr>
<th>Rule</th>
<th>C-value</th>
<th>Genetic operators</th>
<th>Cumulative mutation</th>
<th>Inversion</th>
<th>Period crossover</th>
<th>Product crossover</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mutation</td>
<td>mutation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GA100</td>
<td>0.00</td>
<td>30.81</td>
<td>26.89</td>
<td>32.32</td>
<td>5.47</td>
<td>4.50</td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>33.80</td>
<td>32.21</td>
<td>25.93</td>
<td>4.61</td>
<td>3.45</td>
</tr>
<tr>
<td></td>
<td>0.50</td>
<td>34.43</td>
<td>29.45</td>
<td>30.26</td>
<td>3.35</td>
<td>2.52</td>
</tr>
<tr>
<td></td>
<td>0.75</td>
<td>33.83</td>
<td>28.85</td>
<td>30.97</td>
<td>3.47</td>
<td>2.88</td>
</tr>
<tr>
<td>GA*50</td>
<td>0.00</td>
<td>31.61</td>
<td>28.59</td>
<td>32.09</td>
<td>4.22</td>
<td>3.49</td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>34.83</td>
<td>34.70</td>
<td>24.63</td>
<td>3.14</td>
<td>2.70</td>
</tr>
<tr>
<td></td>
<td>0.50</td>
<td>34.75</td>
<td>34.65</td>
<td>25.03</td>
<td>2.70</td>
<td>2.87</td>
</tr>
<tr>
<td></td>
<td>0.75</td>
<td>33.39</td>
<td>33.23</td>
<td>28.89</td>
<td>2.63</td>
<td>1.86</td>
</tr>
</tbody>
</table>

Table 5
Cost index values for the second test phase

<table>
<thead>
<tr>
<th>Horizon</th>
<th>C-value</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GA*50</td>
</tr>
<tr>
<td>24 periods</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>100.00</td>
</tr>
<tr>
<td></td>
<td>0.50</td>
<td>100.00</td>
</tr>
<tr>
<td></td>
<td>0.75</td>
<td>100.00</td>
</tr>
<tr>
<td>36 periods</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>100.00</td>
</tr>
<tr>
<td></td>
<td>0.50</td>
<td>100.00</td>
</tr>
<tr>
<td></td>
<td>0.75</td>
<td>100.00</td>
</tr>
</tbody>
</table>
Table 6
Execution time for the GAs in CPU seconds

<table>
<thead>
<tr>
<th>Horizon</th>
<th>C-value</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GA*50</td>
<td>GA300</td>
</tr>
<tr>
<td>24 periods</td>
<td>0.00</td>
<td>47.02</td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>77.32</td>
</tr>
<tr>
<td></td>
<td>0.50</td>
<td>75.18</td>
</tr>
<tr>
<td></td>
<td>0.75</td>
<td>70.17</td>
</tr>
<tr>
<td>36 periods</td>
<td>0.00</td>
<td>131.51</td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>177.25</td>
</tr>
<tr>
<td></td>
<td>0.50</td>
<td>189.30</td>
</tr>
<tr>
<td></td>
<td>0.75</td>
<td>214.54</td>
</tr>
</tbody>
</table>

horizon length. Though we rejected the null hypothesis that GA100, GA200 and GA300 are equally efficient under a 24 and a 36 periods horizon, both GA0 and GA*25 produce results which are not significantly affected by the planning horizon. The relative performance of each rule is significantly affected by complexity. The results are robust across the runs of the GA. They change substantially from a replication of costs and demands to another. This should be no surprise for there might be very different product structures, as well as costs and demand patterns, for a given C-value.

6. Conclusion

Though the world of manufacturing is experiencing ongoing change both in terms of consumer needs and the organization of planning and control systems, a number of issues did not fade out. The necessity of carefully selecting adequate inventory control policies certainly is one of these issues. Solving the uncapacitated MLLS problem is left a daily concern for the vast majority of companies engaged in manufacturing production. But the computational requirements imposed by optimizing routines tend to preclude their implementation in realistic size settings. Actually, despite the fact that the MLLS problem is encountered very often in practical installations, most solution methods are faced with severe difficulties in case the problem size grows. One problem in selecting a lot-sizing procedure is that substantial reductions in inventory-related costs can generally be achieved only by using increasingly complex procedures. Optimal models not only demand mathematical prerequisite but also prove unable to solve problems pertaining to complex product structures. For time-varying cost parameters, there is even a lack of simple heuristic rules.

By contrast, the relative simplicity of the heuristic offered in this work, together with its cost-efficiency, makes it an appealing tool to industrials, as documented by our simulation results. We designed specific genetic operators and purposefully constrained search to the set of feasible solutions. The first experimental test phase appraised the effectiveness of the GA in reference to the solutions provided by GAMS. In most cases the best performance was achieved by GA*50. Not only did the GA prove to be a very good cost-saving device, but it also maintained a computational requirement much less stringent than that needed by GAMS. For larger problems, the GA again showed its ability to improve on initial solutions within reasonable time frames.

From a practical perspective, the heuristic offered in this work is significantly easier to understand and implement than optimal mathematical methods. It offers a very satisfactory trade-off between algorithmic complexity and optimality, and as such we believe it can be a powerful near-optimal method for actual settings as well as a useful benchmark to evaluate future heuristic methods.

Appendix. Example of operators

Let us consider the product structure in Fig. 3. Lead times \( \{ l_i \}_{i=1,...,7} \) equal \{1, 1, 2, 0, 1, 1, 0\} and the gross requirements for item 1 (end item) over a 10-period horizon are given by \{200, 100, 20, 50, 10, 40, 65, 100, 40, 30\}. Initial inventories \( \{ I_{1,0}, \ldots, I_{7,0} \} \) equal \{270, 100, 130, 200, 0, 0, 0\}. All scheduled receipts are set to zero except for item 5 for which we have \( r_{5,1} = 25 \). We assume a one-to-one production ratio.

Fig. 4 displays what obtains when the five genetic operators are applied on initial chromosomes.
Fig. 4. Example of operators.
which correspond to feasible solutions to the problem with the general product structure in Fig. 3 over ten periods. Positive net requirements are symbolically represented by stars.

Assume the pair \((i^*, t^*) = (3, 5)\) in chromosome 1 is selected for mutation. In the single-bit mutation, this triggers a mutation from 1 to 0 in period 3 to satisfy condition (7) as a single order is necessary to cover the first positive requirement (which occurs in period 5). In the cumulative mutation, the same mutations take place for item 3. We now perform a mutation from zero to one to items 4 and 5 (the predecessors of item 3) in period \(t^* - l_p = 5 - 2 = 3\). This entails a mutation from 1 to 0 for item 4 in period 5 to satisfy condition (7). String \(y_4\) is left unchanged since \(y_{5,3}\) already equally one. String \(y_6\) is no longer feasible since new net requirements have been generated for this item. The first net requirement appears earlier than before and needs to be covered in time. Periods 2 and 3 are candidates for an order and period 2 is finally selected at random. There is now only one net requirement in time interval \{4, 5, 6\} but two orders. One of them is selected at random for cancellation. Bit \(y_{6,6}\) is therefore set to 0.

Assume the pair \((i^*, t^*) = (3, 4)\) is selected for inversion. Bit \(y_{3,4}\) can either be swapped with bit \(y_{3,5}\) or bit \(y_{3,3}\). Suppose bit \(y_{3,5}\) is randomly chosen for swapping. Bit \(y_{3,4}\) then takes the value of \(y_{3,5}\) which is zero and bit \(y_{3,4}\) is set to one. Predecessors of item 3, say items 4 and 5, undergo the same inversion in periods \(t^* - l_3 = 4 - 2 = 2\) and its neighbouring period 3. Bits \(y_{4,2}\) and \(y_{5,2}\) are respectively, swapped with \(y_{4,3}\) and \(y_{5,3}\). The inversion leads to new requirements for item 7 and some mutations have to be performed to satisfy the feasibility conditions.

Fig. 4 exhibits an example of period cross-over when the crossing site is period 9. The first string of the offspring combines the first 9 ordering periods for item 1 in chromosome 2 and the last ordering period of item 1 in chromosome 1. Let \(t^*_i\) be the crossing period for item \(i\). We have \(t^*_9 = 9\). The crossing period for any other item \(i\) \((i > 1)\) is given by \(t^*_i = \min_{j<i} (t^*_j - l_j)\). For example, we have for item 2 and 3, \(t^*_2 = t^*_3 = t^*_9 - l_9 = 9 - 1 = 8\). For item 4, we have \(t^*_4 = \min(t^*_2 - l_2, t^*_3 - l_3) = \min(8 - 1, 8 - 2) = 3\), etc. One can easily check that each string in the offspring is feasible regarding the new positive requirements.

Assume item 5 is selected as a crossing point to implement the product cross-over. To produce an offspring, we simply combine \((y_1, y_2, y_3)\) from chromosome 1 with strings \((y_4, y_5, y_6, y_7)\) from chromosome 2. One can easily check the feasibility of the resultant offspring. Let us note that the symmetric offspring is not feasible.

References


