

Optimizing production smoothing decisions via batch selection for mixed-model just-in-time manufacturing systems with arbitrary setup and processing times

MESUT YAVUZ†, ELIF AKÇALI*‡
and SÜLEYMAN TÜFEKÇİ§

†Research and Engineering Education Facility,
University of Florida, FL, USA

‡Supply Chain and Logistics Engineering (SCALE) Center,
Department of Industrial and Systems Engineering,
University of Florida, FL, USA

§Department of Industrial and Systems Engineering,
University of Florida, FL, USA

(Revision received August 2005)

This paper is concerned with the production smoothing problem that arises in the context of just-in-time manufacturing systems. The production smoothing problem can be solved by employing a two-phase solution methodology, where optimal batch sizes for the products and a sequence for these batches are specified in the first and second phases, respectively. In this paper, we focus on the problem of selecting optimal batch sizes for the products. We propose a dynamic programming (DP) algorithm for the exact solution of the problem. Our computational experiments demonstrate that the DP approach requires significant computational effort, rendering its use in a real environment impractical. We develop three meta-heuristics for the near-optimal solution of the problem, namely strategic oscillation, scatter search and path relinking. The efficiency and efficacy of the methods are tested via a computational study. The computational results show that the meta-heuristic methods considered in this paper provide near-optimal solutions for the problem within several minutes. In particular, the path relinking method can be used for the planning of mixed-model manufacturing systems in real time with its negligible computational requirement and high solution quality.

Keywords: Production smoothing; Just-in-time manufacturing; Meta-heuristics; Strategic oscillation; Scatter search; Path relinking

1. Introduction

In today's highly competitive industrial environment, low volume high mix (LVHM) production systems are widely used. In the automotive industry, for example, customization of vehicles creates a large variety of products, each with relatively small demand volumes. A similar trend is also observed in contract

*Corresponding author. Email: akcali@ise.ufl.edu

electronics manufacturing. In such LVHM environments, the production facilities are not dedicated to certain products, but shared among a family of products. These types of production systems are known as *mixed-model* systems. The problems that arise in the context of the operation of mixed-model manufacturing systems has attracted significant interest since the 1960s, starting with balancing and sequencing mixed-model assembly lines. For examples, see Thomopoulos (1967, 1970), Macaskill (1972), Dar-el and Cother (1975), Chakravarty and Shtub (1985) and Dar-el and Rabinovitch (1988).

Just-in-time (JIT) advocates having just the right amount of inventory, whether raw materials or finished goods, available to meet the demand of production processes and end customers. Adopting JIT philosophy requires employing a pull system on the shop floor, and the entire system is controlled by the schedule of the final stage of the processing operations. Here, the final stage represents the most downstream level of operations such as assembly of components into end products, and will be referred to as the *final level* in the remainder of the paper. The schedule of the final level determines the demand of the immediate upstream level, and, thereby, its schedule. Since the pull system is applied at every operation level, the schedule of the final level determines the demand pattern for all the operations. In the literature, the problem of scheduling the final level, such that demands for the other operations are stabilized, is known as the *production smoothing problem (PSP)*.

Production smoothing aims at reducing batch sizes and creating a leveled, one-piece-flow of products/parts/materials throughout the entire system. This ideal flow requires that the end products are dispersed over the final schedule as uniformly as possible. For a given end product, this goal is achieved if the cumulative production amount of the product at any time is proportional to the time elapsed since the beginning of the planning horizon. In figure 1, the straight line demonstrates this *ideal schedule* for a given product. The actual production amount, on the other hand, increases when that product is being produced remains constant

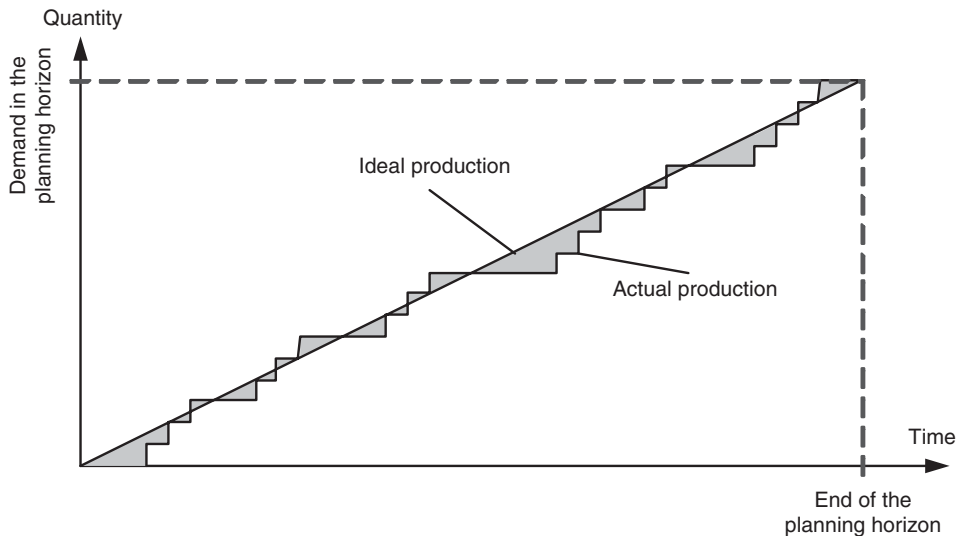


Figure 1. Ideal and actual schedules.

when some other products are being produced. The deviation of an actual schedule from the ideal schedule is captured through the shaded area in the figure. The PSP aims at minimizing this deviation.

The PSP has attracted a lot of interest in the last two decades. In Miltenburg (1989), the problem is formulated as an integer nonlinear programming problem, where the objective function measures the total squared deviation between the actual and ideal schedules for each product at each stage in the processing sequence. Several heuristic and exact solution methods have been provided for this problem (Miltenburg 1989, Miltenburg *et al.* 1990, Kubiak and Sethi 1991, Ding and Cheng 1993, McMullen 1998, 2001a, b, c, McMullen and Frazier 2000, McMullen *et al.* 2000). The most efficient exact method known so far is Kubiak and Sethi's (1991) transformation to the assignment problem.

In all the above-mentioned work, the PSP has been studied for synchronized assembly lines only, where each product takes exactly one unit of processing time on every station and setup times are ignored. Yavuz and Tufekci (2004a) analyse a special case of the PSP in a multi-product environment, where the final operation of the manufacturing system is a single machine, on which the end products have arbitrary non-zero setup and processing times. Solving PSP in such a mixed-model production environment is complicated by the need to optimize setup times between products in addition to stabilizing the production quantities of products and the workload of operations. The authors develop a two-phase solution methodology, where the first phase finds appropriate batch sizes for the products, and the second phase finds a sequence for these batches. We refer to these phases as the *batching problem* (BP) and the *sequencing problem* (SP), respectively. The key concept of the two-phase approach adopted is the takt-time, which helps to define the batch throughput rate of the manufacturing system, i.e. the rate at which batches are completed. For instance, if the takt-time for a manufacturing system is two minutes, then a finished batch departs from the final level of this system every two minutes. That is, a batch of a product is produced every two minutes. Yavuz and Tufekci also show that the SP can be solved efficiently by transforming it to an assignment problem adapted from Kubiak and Sethi's (1991) work, whereas the BP is NP-complete. They propose a parametric heuristic solution procedure that solves the BP through a series of local searches, and show that this procedure promises near-optimal solutions for small-sized problems. This basic local search approach promises good solutions, however it may be trapped in local optima. It is also worth noting that the increase in computation speed has increased our ability to solve hard optimization problems to optimality by enumerative approaches. However, these enumerative approaches may take exponential time in the worst case, and, hence, the development of efficient and effective heuristic solution procedures is still an important area for research.

This study aims to achieve two main research goals. The first goal is to develop efficient and effective meta-heuristic methods to solve a practical manufacturing planning problem. The second goal is to ensure that the proposed solution approach is not only computationally efficient, but also efficient for the entire manufacturing system by achieving low inventory levels. In this paper, we propose a dynamic programming procedure for the exact solution of the BP as well as a new heuristic method based on the continuous relaxation of the BP and three

meta-heuristic techniques, namely strategic oscillation, scatter search and path relinking, to obtain near-optimal solutions for the BP. The performance of the methods is assessed through three performance metrics: computation time, relative deviation from the optimal solution and average inventory level in the entire manufacturing system imposed by the solution.

The remainder of this paper is organized as follows. In section 2, we present a mathematical formulation of the problem. We propose a dynamic programming procedure for the exact solution of the problem in section 3. In section 4, we review existing heuristics for the BP and develop a new heuristic procedure based on the continuous relaxation of the BP formulation. Meta-heuristic techniques are introduced in section 5. In section 6, we develop the experimentation plan, describe our fine-tuning process in the implementation of the meta-heuristic techniques and present the results of our computational study. Finally, in section 7, we present our research findings, and give concluding remarks.

2. Problem formulation

2.1 Preliminaries

Let $N = \{1, 2, \dots, n\}$ be the set of different types of products, indexed by i . Let s_i and p_i denote the setup and unit processing times of product i on the machine, respectively, and d_i the demand for product i in the planning horizon. The batch size and number of batches of product i are denoted by b_i and q_i , respectively. $Q = \sum_{i \in N} q_i$ is the total number of batches to be manufactured in the planning horizon, and T the total available time in the planning horizon. Also, $\lceil x \rceil$ denotes the smallest integer that is greater than or equal to x .

Each batch should be processed within the takt-time, the length of which is obtained by dividing T into Q equal intervals. Both the number of batches and batch sizes are assumed to be integers, which reduces the possibility of satisfying the demand in exact quantities. Therefore, in all the solution procedures, either overproduction or underproduction should be allowed. In our work, we do not allow underproduction, but accept overproduction in minimal quantities. The minimization of the overproduction is achieved through defining a set of *acceptable values* for the number of batches.

Acceptable values of the number of batches q_i for a product i are the integer values that satisfy the demand for the product with minimum overproduction. $q_i = 1$ implies producing all units in a single batch, where $b_i = d_i$ is, thus, an acceptable value for any product. The next acceptable value is found by either decreasing the batch size by one, or increasing the number of batches by one. We propose procedure *Find Acceptable Values* to find all the acceptable values for a decision variable q_i (see figure 2). For example, if demand data for a product i is 15 units, the acceptable values found by the above procedure for q_i are presented in table 1. The first five q_i values are found easily, and it is easy to calculate corresponding b_i values. However, we want to clarify why, for example, 6 is not an acceptable value. If $q_i = 6$, the minimum batch size that satisfies the demand is $b_i = 3$. In this case, $6 \times 3 = 18 > 15$ the demand is met with three units of overproduction. However, for $b_i = 3$, the minimum number of batches that should be produced is $q_i = 5$.

<p>Algorithm Find Acceptable Values 0. Start with an empty list. Set $q_i = 1$ and $b_i = d_i$. 1. Add q_i to the list. IF $(q_i = d_i)$ THEN STOP. 2. IF $(\lceil d_i / (q_i + 1) \rceil < b_i)$ THEN Set $q_i = q_i + 1$ and update b_i ($b_i = \lceil d_i / q_i \rceil$). ELSE Set $b_i = b_i - 1$ and update q_i ($q_i = \lceil d_i / b_i \rceil$). 3. Return to Step 1.</p>

Figure 2. Pseudo-code for algorithm Find Acceptable Values.

Table 1. Acceptable values for $d_i = 15$.

q_i	1	2	3	4	5	8	15
b_i	15	8	5	4	3	2	1

This does not agree with the presumed number of batches ($q_i = 6$). Therefore, $q_i = 6$ is not an acceptable value. In the remainder of this paper, A_i denotes the set of acceptable values of q_i for product i , and a_i the cardinality of this set.

2.2 Mathematical model

The batching problem can be formulated as an optimization model as follows:

$$(BP) \text{ Minimize } F = \sum_{i=1}^n \frac{(\lceil d_i / q_i \rceil)^2 \left((\sum_{h=1}^n q_h)^2 - q_i^2 \right)}{\sum_{h=1}^n q_h}, \tag{1}$$

$$\text{subject to } \left(s_i + p_i \left\lceil \frac{d_i}{q_i} \right\rceil \right) \sum_{h=1}^n q_h \leq T, \quad \forall i, \tag{2}$$

$$q_i \in A_i, \quad \forall i. \tag{3}$$

The objective function (1) is derived from a lower bound to the sequencing phase of the batch PSP (Yavuz and Tufekci 2004a, b). Constraint set (2) ensures that each product is batched in a way that the sum of the setup and processing time of the batch is not longer than the takt-time. Constraint set (3) ensures that the q_i values are restricted to the acceptable values for product i as discussed above.

Note that the decision variables are integer-defined, thus the problem is a discrete optimization problem with nonlinear functions both in the objective function and in the constraints. The ceiling function is used to fulfill the integrality requirements, hence the formulation includes non-smooth functions of the variables. Other structural properties of the model include the non-convexity of the feasible region and the objective function. For further reading on the mathematical investigation of the model, we refer the reader to Yavuz and Tufekci (2004a).

2.3 Neighbourhood structure

A solution $q = (q_1, q_2, \dots, q_n)$ is a vector of the decision variables such that all the decision variables take an acceptable value $q_i \in A_i, \forall i \in N$. A feasible solution

satisfies both constraint sets (2) and (3), whereas an *infeasible solution* satisfies constraint set (3) only. These infeasible solutions are used extensively in our solution methods.

Now, consider the following example with $n=2$ products. Let $d_1=15$ and $d_2=20$; $s_1=s_2=1$, $p_1=p_2=1$ and $T=50$ min. The above procedure proposed for finding the acceptable values implies $q_1 \in A_1 = \{1, 2, 3, 4, 5, 8, 15\}$ and $q_2 \in A_2 = \{1, 2, 3, 4, 5, 7, 10, 20\}$. Any pair of these acceptable values is a solution, for example (1, 1), (5, 5) and (5, 20) are all solutions. (5, 5) is a feasible solution, since the batch sizes are 3 and 4 and these batches take 4 and 5 min, where the length of the takt-time is $50/(5+5) = 5$ min, therefore both batches can be processed within the takt-time. Similarly, (5, 20) requires 4 and 2 min to process the batches, however the takt-time is too short ($50/(5+20) = 2$ min), and thus this solution is infeasible.

A solution $q^1 = (q_1^1, q_2^1, \dots, q_n^1)$ is a *neighbour* of $q^0 = (q_1^0, q_2^0, \dots, q_n^0)$ if and only if exactly one variable value is different in these vectors, and the categorical distance between the values of this decision variable is at most ρ , where ρ is a user-defined integer that is greater than or equal to one. Let us denote the set of neighbour solutions of a solution q^0 with $NS(q^0, \rho)$ and consider $q^0 = (5, 5)$ in the above example. Assuming $\rho=2$, the acceptable values to consider are $\{3, 4, 8, 15\}$ for q_1 and $\{3, 4, 7, 10\}$ for q_2 . The set of the neighbour solutions is constructed by using the value of one variable in the given solution and replacing the other one with its alternative acceptable values, one by one. Here the neighbour solutions set of q^0 is $NS((5, 5), 2) = \{(3, 5), (4, 5), (8, 5), (15, 5), (5, 3), (5, 4), (5, 7), (5, 10)\}$. With this definition, a solution may have at most $2 \times \rho \times n$ neighbours.

We identify two particular solutions. The first one is the *origin*, where each decision variable takes its lowest possible value, that is $q_i = 1, \forall i \in N$. The second one is *the farthest corner* of the solution space, where every decision variable takes its largest value, that is $q_i = d_i, \forall i \in N$. If we relax integrality of batch sizes, and let $r_i = q_i/Q$ where $0 \leq r_i \leq 1$ such that $\sum_{i \in N} r_i = 1$ denotes the proportion of the number of batches of a certain product to the total number of batches, and assume these proportions (r_i 's) are fixed, then the objective function (1) becomes $\sum_{i \in N} (d_i/r_i)^2 (1 - r_i^2)/Q$. This shows that larger Q values are expected to yield better solutions. We can intuitively argue that the global optimum may be located in the vicinity of the farthest corner of the solution space. Therefore, guiding the search process towards this farthest corner might help us to find the global optimum.

3. Exact solution with dynamic programming (DP)

We note that, given a fixed Q value, the objective function (1) simplifies to $F' = \sum_{i=1}^n (\lceil d_i/q_i \rceil)^2 (Q^2 - q_i^2)/Q$, which is separable in q_i variables. If the vector $q^* = (q_1^*, q_2^*, \dots, q_n^*)$ is an optimal solution to the problem with $\sum_{i=1}^n q_i^* = Q$, then the sub-vector $(q_2^*, q_3^*, \dots, q_n^*)$ should also be optimal to the problem with $\sum_{i=2}^n q_i^* = Q - q_1^*$. Otherwise, the vector q^* cannot be an optimal solution. Thus, the principle of optimality holds for the problem, and we can build the optimal solution by consecutively deciding on q_i values. Let product index i correspond to the stage index in the DP formulation and the pair (i, R) represent the states.

We define the following recursive equation:

$$F(i, R) = \begin{cases} 0, & \text{if } i = 0, \\ \min_{q_i} \left\{ F(i - 1, R - q_i) + \left(\left\lceil \frac{d_i}{q_i} \right\rceil \right)^2 (Q^2 - q_i^2) / Q |s_i + \left\lceil \frac{d_i}{q_i} \right\rceil p_i \leq \frac{T}{Q} \right\}, & \text{if } i > 0. \end{cases}$$

In this formulation, the final state is (n, Q) , and the solution to the problem, $F(n, Q)$, can be found using a forward recursion. The number of states in a stage is bounded by $\sum_{i \in N} d_i$. Computing the optimal cost of reaching a state in the i th stage may require at most a_i calculations, therefore in this forward recursive structure, the modified problem can be solved by performing at most $\sum_{i \in N} a_i \sum_{i \in N} d_i$ calculations. To solve the BP, we need to consider every possible value of $n \leq Q \leq \sum_{i \in N} d_i$. Therefore, the time complexity of this DP procedure is $O(\sum_{i \in N} a_i (\sum_{i \in N} d_i)^2)$.

4. Heuristic solution methods

In this section, we first review the existing solution procedures for the BP. We then present a new heuristics procedure that is based on the continuous relaxation of the BP formulation.

4.1 Existing solution procedures

Yavuz and Tufekci (2004a) propose a parametric heuristic procedure that relies on first finding an initial feasible solution, and, then, improving the solution with successive local searches (see figure 3). In the first stage, the initial solution is set to the origin, that is $q_i = 1, \forall i \in N$. If this solution is not feasible, then constraints (2) are examined. The constraint with $\max\{s_i + p_i \lceil d_i / q_i \rceil : i \in N\}$ is identified as the most violated constraint, and the value of q_i is increased. Yavuz Tufekci (2004a)

Algorithm Parametric Heuristic Search (*SearchDepth*, *MoveDepth*, *Eligible*)

1. Set Current Solution as $q_i = 1, \forall i \in N$ and perform a feasible solution search by increasing values one at a time. If no feasible solutions can be found, stop. Otherwise set Current Solution as this solution.
2. Evaluate all *SearchDepth*-step *EligibleNeighbors* of the Current Solution. If the Best Neighbor is not null, then move to the *MoveDepth*-step neighbor that leads to the Best Neighbor. If this new solution is feasible then repeat step 2. Otherwise go to step 3.
3. Search for a feasible solution by increasing q_i values one at a time. If a feasible solution is found, then move to that feasible solution, and go to step 2. Otherwise go to step 4.
4. Return to the last visited feasible solution. If the Best Feasible Neighbor is not null, then move to the *MoveDepth*-step feasible neighbor that leads to the Best Feasible Neighbor, and go to step 2. Otherwise stop and return the Best Solution.

Figure 3. Pseudo-code for algorithm Parametric Heuristic Search.

prove that this search policy guarantees finding a feasible solution, if one exists. The second stage of the approach is based on the general local search idea.

The procedure takes three parameters. The first parameter represents the search depth, such that if the search depth is one, then only the immediate neighbours of the current solution are evaluated; if the search depth is two, then the neighbours' neighbours are also evaluated. The second parameter represents the proximity of the neighbour solution that will be moved to after all the neighbours are evaluated, and the third parameter specifies whether infeasible solutions are used or not. The authors compare four methods, the first three of which evaluate feasible solutions only. The first method (PSH1) uses a search depth of one and moves to the best neighbour evaluated. The second method (PSH2) sets the search depth to two and moves to the immediate neighbour which leads to the best neighbour evaluated. The third method (PSH3) also sets the search depth to two, but it moves directly to the best neighbour found. The fourth method (PSH4) evaluates all one-step neighbours, including the infeasible ones. If the neighbour that yields the lowest objective value is infeasible, then the move to this infeasible solution is allowed, but the next move is forced to be a move to a feasible solution.

4.2 A new heuristic procedure

In this paper, we introduce a new heuristic procedure developed to solve the BP. The procedure is based on relaxing the integrality requirements in the BP and solving the relaxed problem (see figure 4). An optimal solution to the relaxed problem satisfies the relationship $q_i^*/q_j^* = (d_i/d_j)^{2/3}$ between all (q_i, q_j) pairs. This further allows us to construct a solution to the relaxed problem for a given Q value. See the algorithm in figure 4 and the details of the derivation in appendix A.

The simplified problem can be solved in $O(n)$ time, and yields the optimal number of total batches (Q^*). Using the r_i^* values, this solution can be converted to an optimal solution of the continuous problem ($q_i^* = r_i^* Q^*, \forall i \in N$).

In order to use this solution as an initial solution for our meta-heuristic methods, we need to first round the q_i^* values to the nearest acceptable values and then check the feasibility of the rounded solution. If it is not feasible, then the algorithm performs a local search to obtain a feasible solution. The closest feasible solution can now be used as an initial solution in a meta-heuristic approach.

Algorithm Continuous Relaxation Heuristic

0. Relax the integrality requirements in the problem.

1. Build r_i^* vector using the relationships $\frac{q_i^*}{q_j^*} = \left(\frac{d_i}{d_j}\right)^{\frac{2}{3}}$ and $r_i^* = \frac{q_i^*}{\sum_{j=1}^n q_j^*}$.

2. Set $Q^* = \min_i \left\lceil \frac{T - p_i \frac{d_i}{r_i^*}}{s_i} \right\rceil$.

3. Obtain continuous solution vector q^* using $q_i^* = r_i^* Q^*, \forall i$.

4. Round the q_i^* values to the nearest acceptable values for all i .

5. Check if the rounded solution is feasible. If not, then search for a feasible solution in the neighborhood.

Figure 4. Pseudo-code for algorithm Continuous Relaxation Heuristic.

5. Meta-heuristic techniques

Meta-heuristic techniques are based on local search strategies, and require first finding an initial solution and then moving to an improving neighbour solution through a local search framework. In contrast to local search approaches, meta-heuristics do not necessarily stop when no improving neighbour solutions can be found. They perform moves to worsening solutions in order to prevent premature convergence to a local optimum solution. Meta-heuristics can employ some problem-specific heuristics in the initialization step, and use these solutions as starting points for the local searches. For more on meta-heuristic techniques, we refer the reader to Reeves (1993) and Osman and Laporte (1996).

The BP is an integer nonlinear programming problem with non-smooth functions. The existence of non-smooth, nonlinear functions in the constraints makes the feasible region non-convex. Therefore, it is highly probable that most of the neighbour solutions are infeasible, and finding a feasible neighbour to evaluate becomes a difficult task. In the literature, there exist some meta-heuristic techniques which are able to work well with sparse and non-convex feasible regions. In this paper, we consider strategic oscillation, scatter search and path relinking.

5.1 Strategic oscillation

The idea behind strategic oscillation (SO) is to drive the search towards and away from boundaries of feasibility (Kelly *et al.* 1993, Dowsland 1998). SO operates by moving with local searches until hitting a boundary of feasibility. Then, it crosses over the boundary and proceeds into the infeasible region for a certain number of moves. Then, a search in an opposite direction, which results in re-entering the feasible region, is performed. Crossing the boundary from feasible to infeasible and from infeasible to feasible regions continuously during the search process creates some form of oscillation, which gives its name to the method (Kelly *et al.* 1993, Dowsland 1998, Glover 2000, Amaral and Wright 2001).

We implement the SO method in a multi-start manner, where the starting solutions are called seed solutions. For the generation of these seed solutions we apply three of the four problem-specific heuristic methods proposed in Yavuz and Tufekci (2004a) (PSH1, PSH3 and PSH4), and the continuous relaxation heuristic developed in section 4.2. Here, we choose the problem-specific heuristic methods with respect to their time requirement, as reported in Yavuz and Tufekci (2004a). Furthermore, we take the farthest corner of the solutions space defined in section 2.3 as an additional seed solution. These seed solutions are compared to each other and duplicates are eliminated before initiating the search process. During the search process, the numbers of moves in the feasible and infeasible regions are limited by the parameters *NFM* and *NIM*, respectively. For the termination of the method, we use two criteria simultaneously. The first criterion is set on the number of iterations, such that at least *MaxIters* iterations are performed. The second criterion is based on the relative improvement obtained in the most recent iterations, such that if the iterations are providing significant improvements the procedure does not terminate even if the number of iterations has exceeded *MaxIters*. This criterion is identified by parameter *RelativeImprovement* which is actually a pair of two sub-parameters

```

Algorithm S0
Find seed solutions using problem specific heuristics. Add the farthest corner of the solution
space to the list of seed solutions.
For each seed solution{
  Repeat until Termination Criteria are satisfied.{
    Perform a local search for NFM moves.
    Cross over the boundary and perform a local search for NIM moves into
    the infeasible region.
    Perform a local search for a feasible solution.
  }
  Perform successive local searches on the current solution to find a local optimum.
}
}

```

Figure 5. Pseudo-code for algorithm S0.

(*Number, Percentage*), where *Number* is the number of the most recent iterations to be followed and *Percentage* is the limit of the percent relative improvement.

We consider only one-step neighbours, but control the neighbourhood size by a parameter, *Range*. The parameter *Range* corresponds to ρ presented in the definition of the neighbour structure. We present the algorithmic structure of our SO implementation in figure 5.

5.2 Scatter search and path relinking

Scatter search (SS) and path relinking (PR) are evolutionary methods that have been proven to yield promising results in solving combinatorial optimization problems (Glover 1998, 1999).

SS is an evolutionary method that constructs new solutions by combining currently known solutions. It maintains a population of solutions on hand, creates candidate solutions using the current population on each iteration, and selects the fittest solutions to be kept in the population for the next iteration. The original form of SS, as proposed in Glover (1977), consists of three stages. First, a starting set of solutions is generated by using heuristic methods designed for the problem and a subset of the best solutions is selected to be the *reference set*. Second, subsets of the reference set are used to create new solutions as linear combinations of the current reference solutions. The linear combinations are chosen to produce points both inside and outside the convex regions spanned by the reference solutions. Such linear combinations generally yield non-integer solutions, therefore a rounding process is employed to obtain integer-valued solution vectors when necessary. Third and finally, the candidate solutions created in the second phase and the original reference solutions are evaluated and the fittest solutions are randomly selected for the next iteration's reference set. The last two phases are repeatedly applied for a pre-determined number of iterations.

Glover (1998) defines several embellishments to this approach. A *Diversification Generator* can be developed to generate a collection of diverse trial solutions, using an arbitrary trial solution (or seed solution) as an input. The diversification generator is employed in the first phase where the solutions from the problem-specific heuristic methods are processed to obtain a rich initial

reference set. An Improvement Method can be used to transform a trial solution into one or more enhanced trial solutions. This improvement method can be a local search procedure which is successively applied until a local optimal solution is found. A Reference Set Update Method can be employed to build and maintain a reference set consisting of the b best solutions found, organized to provide efficient accessing by other parts of the method. A Subset Generation Method can be used to operate on the reference set, to produce a subset of its solutions as a basis for creating combined solutions. Finally, a Solution Combination Method can be built to transform a given subset of solutions produced by the Subset Generation Method into one or more combined solution vectors.

We now give a description of our SS implementation. For initialization, we apply three of the four problem-specific heuristic methods proposed in Yavuz and Tufekci (2004a) (PSH1, PSH3 and PSH4), and the continuous relaxation heuristic developed in section 4.2. Note that we also use the same problem-specific heuristics in the initialization of the SO method.

Having established a set of seed solutions, we use our diversification generator to create the initial reference set. We use two alternative modes of the diversification generator. The first mode is similar to the mutation operator used in genetic algorithms (Holland 1975, Goldberg 1989, Reeves 1997). That is, the seed solution vector is taken as the input and, starting with the first variable, a diversified solution is created for each variable. This is achieved by replacing the variable's value with its 100th next acceptable value. If $a_i < 100$, the mod operator is used in order to obtain an acceptable value with index value between 1 and a_i . Here, 100 is arbitrarily selected, and any significantly large integer such as 50, 200 or 500 can also be chosen. The second mode, on the other hand, does not process seed solution vectors. It performs a local search for each decision variable and identifies solutions that maximize the value of that particular decision variable. This mode of diversification yields a total of n alternative solutions and enables us to explore extreme corners of the feasible region. The parameter representing the selection of the diversification mode is *Diversification*, and it has four levels. At level 1 no diversification is applied, at level 2 only a corner search is applied, at level 3 only the first mode of the diversification generator is used and, finally, at level 4 both modes are used. Depending on the mode selection in the application of the algorithm, the number of diversified solutions may be less than the size of the reference set. In this case, the empty slots in the reference set can be filled in the consecutive iterations.

The size of the reference set is represented by parameter b . In our implementation we keep one infeasible solution in the reference set at all times. This infeasible solution is the farthest corner of the solution space discussed in section 2.3.

The most important aspect of subset generation is the subset size, which determines the number of subsets generated. We create all subsets with two elements, subsets with three elements that contain the best solution, subsets with four elements that contain the best two solutions and subsets with k elements ($5 \leq k \leq b$) that contain the k best solutions. The use of these subset types is controlled by parameters *SubsetType*₁ through *SubsetType*₄, which take *true* or *false* values. If a parameter takes value *false*, the associated subset type is not generated and if it takes value *true*, all the subsets of that type are generated.

For the solution combination mechanism used in SS, we take the weight centre of the solutions in the subset under consideration as the basis. Each solution

in the subset is treated as an original solution and a line from the weight centre to the original solution, the *inner line segment*, is drawn. The line is extended as long as its length, yielding the *external line segment* and the end point. Let parameters *NIC* and *NEC* represent the number of internal and external combinations, respectively. *NIC* equidistant linear combinations on the inner line segment and *NEC* equidistant linear combinations on the external line segment are created. The weight centre, the external end points, internal and external combinations give a total of $(NIC + NEC + 1) \times SubsetSize + 1$ combination solutions, for a certain *SubsetSize*.

Using the improvement method on combined solutions and updating the reference set accordingly are common in both the initial and iterative phases. However, performing a local search on every solution obtained may be impractical. *LSinPreProcess* is the parameter that represents local search usage in the initial phase. If *LSinPreProcess* = 0, no local search is applied. If *LSinPreProcess* = 1, local search is only applied at the end of the initial phase, on the solutions that are stored in the reference set. If *LSinPreProcess* = 2, a local search is applied for every trial solution considered. *LStoRefSetPP* is the parameter representing the update frequency of the reference set and takes the values of *true* or *false*. *LStoRefSetPP* = *true*, every time a solution is evaluated, it is compared to the solutions in the reference set and, if necessary, the reference set is updated. This requires that every move performed during the local search is considered for the reference set. If *LStoRefSetPP* = *false*, only the final result of the local search, a local optimum, is tried for the reference set. We apply the same ideas to the iterative phase with parameters *LSinIterations* and *LStoRSIters*.

For the termination of the algorithm we have one criterion only. If the reference set is not modified on a given iteration, it cannot be modified on subsequent iterations either. Therefore, we keep track of the solutions in the reference set and immediately terminate if the reference set is the same before and after an iteration. This criterion does not require a parameter.

The SS and PR methods differ in the solution combination mechanism they employ. In SS, the combined solutions are analytically related to the parent solutions. In PR, on the other hand, the combination of the parent solutions is based on the neighbourhood function defined for the problem. Therefore, we use the same structure for our SS and PR implementations. The differences between our implementation of the two methods involve the subset generation and solution combination mechanisms.

The subset generation mechanism used for PR considers the subsets with two solutions only. These solutions are used as origin and destination points in the solution combination mechanism. Based on the acceptable values, we measure the distance between the origin and the destination with a categorical distance measure. If q^1 and q^2 are the origin and destination vectors, and we define the function $Position(q_i)$ as an integer function which returns the position of variable i 's value in A_i , then the distance between these two vectors is defined as $\sum_{i \in N} |Position(q_i^1) - Position(q_i^2)|$, where $|x|$ is the absolute value of x . Starting from the origin, the neighbour solutions that decrease this distance by one are considered and the best *NTS* solutions are stored in a list, where *NTS* is the parameter standing for the number of temporary solutions. In the next step, each solution in this list is considered as the origin, and again the neighbour solutions

<p>Algorithm SS/PR</p> <p><u>Initialization</u></p> <p>Initialize the Reference Set with seed solutions, using problem specific heuristics.</p> <p>For each seed solution{</p> <p style="padding-left: 2em;">Create all diversified solutions of the seed solution on hand.</p> <p style="padding-left: 2em;">For each diversified solution{</p> <p style="padding-left: 4em;">Find a local optimum using the Improvement Method.</p> <p style="padding-left: 4em;">Update the Reference Set.</p> <p style="padding-left: 2em;">}</p> <p>}</p> <p><u>Improvement</u></p> <p>Generate subsets of the Reference Set.</p> <p>For each subset{</p> <p style="padding-left: 2em;">Create combinations of the solutions in the subset.</p> <p style="padding-left: 2em;">For each combination {</p> <p style="padding-left: 4em;">Find a local optimum using the Improvement Method.</p> <p style="padding-left: 4em;">Update the Reference Set.</p> <p style="padding-left: 2em;">}</p> <p>}</p> <p>Iterate until Termination Criteria are satisfied.</p>
--

Figure 6. Pseudo-code for algorithm SS/PR.

that further decrease the distance by one are evaluated. This is repeated until the destination solution is reached, while keeping *NTS* best solutions between the steps. $NTS = 1$ represents a single path between the origin and the destination. However, $NTS > 1$ can be considered as *NTS* parallel paths that are built between the origin and the destination solutions. All the other mechanisms explained for SS, and the associated parameters, are used for the PR method. The generic algorithm in figure 6 presents our implementation of the SS and PR methods.

6. Computational study

In this section, we first describe our experimental design. We then present our fine-tuning approach. We conclude this section by discussing the results from our computational experiments.

6.1 Design of experiments

In our study, we consider 10, 15 and 20 product problems with average demand of 750, 500 and 375, respectively, which can be solved by the dynamic programming procedure in reasonable times.

We use three experimental factors, the setup time to processing time ratio α , T relaxation percentage β and diversification level γ . $\gamma \in \{0, 1\}$ is used to create test cases in which different products are diversified in terms of demand, processing time and setup time. $\gamma = 1$ reflects the diversified case, and $\gamma = 0$ reflects the undiversified case where the products are very similar to each other. Demand values are randomly and uniformly generated between the minimum and maximum values, where

maximum demand is twice as large as the average demand for diversified instances and 20% over the average demand for the instances with similar products. The ratio of maximum demand to minimum demand is 50 and 1.5 for these two types of instances, respectively.

We use α to denote the ratio between the expected values of s_i and p_i for the diversified instances. We first create p_i according to a uniform distribution between $(0, 5]$, and then s_i according to a uniform distribution between $[(1 - 0.1 \times \gamma) \times p_i \times \alpha, (1 + 0.1 \times \gamma) \times p_i \times \alpha]$. We let $\alpha \in \{100, 10, 1\}$ for our experiments.

The total available time should allow at least one setup per product, that is $T \geq T_{LB} = \sum_{i \in N} (d_i p_i + s_i)$. On the other hand, T should be limited with $T < T_{UB} = \sum_{i \in N} d_i (p_i + s_i)$. We create T with $T = \beta T_{UB} + (1 - \beta) T_{LB}$, where $\beta \in \{0.2, 0.5, 0.8\}$.

Allowing three different values for the parameters α and β and two different values for γ results in 18 different problem sets. In order to obtain reliable results, 25 instances are created for each problem set, giving a total of 450 test instances for each n value and 1350 test instances in total.

6.2 Fine-tuning

Generally, there are several parameters used in a meta-heuristic method. The computational requirement and the solution quality of a meta-heuristic method heavily depend on the values of the parameters. Therefore, special care should be taken in determining the parameters' values. In other words, the parameters should be *fine-tuned*. The fine-tuning process can be seen as a supervised learning process, in that the user selects a set of test instances of the problem and experiments with different values of the parameters on this set of selected instances. The parameters are set to their most promising values and the method is used to solve new instances of the problem.

The parametric structure of our computer code is quite flexible in terms of testing alternative strategies for a method. Based on the results from our preliminary experiments with the methods, we identify five, six and four critical parameters for the SO, SS and PR methods, respectively. We identify two levels for each critical parameter and adopt a half-factorial design strategy for the experiments. That is, we fine-tune three meta-heuristic methods with 16 experiments for SO, 32 experiments for SS and eight experiments for PR.

We use 20% of the test instances (five problems for each problem setting presented in the previous section) for fine-tuning. That is, the most promising methods according to their performance on the fraction of the test instance will be used on the entire set of test instances. To circulate the significance of the difference between the tested levels of a parameter, we apply paired t -tests. We denote the mean values of computation time and relative deviation from the optimal solution measures with μ_l^t and μ_l^d , respectively, for the l th level of the parameter. One hypothesis per measure is built. For all t -tests, we use a confidence level of 95%.

Combining all the parameters we denote the SO method with SO(*MaxIters*, *NFM*, *NIM*, *Range*, (*Number*, *Percentage*)). After fine-tuning the method, we obtained SO($3N - 20, 45 - N, 10, 1, (5, 0.01\%)$). We represent the SS method with SS(*b*, *SubsetSize*, *NIC*, *NEC*, *LSinPreProcess*, *LSinIterations*). The fine-tuning process yielded the following combination: SS($N + 10, 3, 3, 2, 1, 1$). Finally, the PR

method is represented with $PR(b, NTS, LSinPreProcess, LStoRSIters)$. The most promising combination found in the fine-tuning process is $PR(N + 20, \lceil 3N/8 \rceil, 1, false)$.

6.3 Results

In evaluating the computational performance of our solution methods, we consider three performance measures. The first two measures are computational time and percent deviation from the optimal solution. These two measures represent the trade-off between the solution quality and time required to obtain this solution. We define a third measure to extend our analysis beyond this trade-off, by mapping the solutions found to the BP to the average inventory level of the manufacturing system. Recall that reducing the average inventory level is the ultimate objective in the JIT philosophy. Results from solving the test instances with three meta-heuristic methods and an exact method, for the computation time and percent deviation from the optimum measures, are summarized in table 2.

We analyse the difference of the methods using pairwise comparisons of both computational time and percent deviation from the optimal solution measures. A total of 12 null hypotheses are built and all of them are rejected at the 95% confidence level by two-tailed paired t -tests. That is, all the methods are significantly different from each other in terms of both the solution time and deviation from the optimal solution. The ordering of the methods is $\mu'_{DP} > \mu'_{PR} > \mu'_{SS} > \mu'_{SO}$ for the solution time and $\mu^d_{DP} < \mu^d_{PR} < \mu^d_{SS} < \mu^d_{SO}$ for the deviation measure.

The dynamic programming procedure proposed for the exact solution of the problem requires extensive computational time. In the worst case, we see that it solves 20 product problems in approximately 40 min, which is far beyond our goal of solving the problem within several minutes.

Since this study is not concerned with solving the BP with problem-specific heuristic methods, the performance of these methods is not reported. However, we note that our continuous relaxation heuristic that runs very fast (under 1% of a

Table 2. Summary of results.

n	Method	Time (s)			Deviation (%)	
		Avg.	Min.	Max.	Avg.	Max.
10	DP	302.07	6.63	1279.07		
	SO	1.67	0.64	5.22	0.137	4.219
	SS	3.28	0.57	16.14	0.121	4.219
	PR	3.43	0.55	40.53	0.009	1.156
15	DP	482.03	15.09	1849.66		
	SO	8.71	2.41	30.78	0.229	6.227
	SS	13.75	1.94	73.70	0.099	6.227
	PR	16.84	2.02	272.922	0.008	1.293
20	DP	631.88	11.18	2247.64		
	SO	22.32	6.39	94.52	0.285	3.845
	SS	40.23	4.70	195.81	0.138	3.138
	PR	52.18	5.47	599.88	0.020	1.363

Table 3. Average inventory levels for the solutions found by each method.

n	Method			
	DP	SO	SS	PR
10	49.5	49.7	49.6	49.5
15	75.8	76.1	76.0	75.8
20	102.4	102.8	102.6	102.5
Avg.	75.9	76.2	76.1	75.9

second), is used in finding initial solutions for the meta-heuristic methods and contributes to diversify the initial solutions.

Among the meta-heuristic methods we consider in this study, we see that a trade-off between solution quality and solution time exists. PR requires the longest computational time and yields the best solution quality. If solutions are needed to be obtained in less time, then either SS or SO can be selected, depending on the desired solution quality.

As far as the BP is considered, all three meta-heuristic methods prove effective with respect to the relative deviation from the optimal solution. We further test the performance of the methods by comparing the average inventory levels implied in the system, by the batching solutions. For measuring the total average inventory level, we use $B = \sum_{i \in N} b_i$ (for details on deriving this measure, see appendix B). The results in table 3 show that the exact method yields the lowest average inventory level. We have also built six hypotheses for the indifference of the methods and rejected all but one hypothesis. The results give an order of the methods as $B_{DP} \equiv B_{PR} < B_{SS} < B_{SO}$ for the 95% confidence level, where B_x is the B value calculated for the solution found by method x . This result shows that our PR implementation is as helpful as the exact method in minimizing the average inventory level of the system.

7. Conclusions

This paper addresses the production smoothing problem that arises in mixed-model just-in-time manufacturing systems, where products have arbitrary setup and processing times and the limit on the total available time necessitates manufacturing the products in batches. We particularly focus on the batching problem that aims at finding optimal batch sizes of the products. We present a dynamic programming algorithm for the exact solution of the problem, develop a new heuristic based on the continuous relaxation of the problem and propose meta-heuristic solution methods using strategic oscillation, scatter search and path relinking.

The dynamic programming procedure is used to solve up to 20 product problems. This is a significant improvement over the size of the problems solved in the literature previously. However, the worst case computational time of the dynamic programming algorithm is found to be excessive, rendering its use in practical settings infeasible. The computational results show that the meta-heuristic methods

considered in this paper provide near-optimal solutions for the problem within several minutes. Our test results show that path relinking deviates from the optimal solution by at most 1.4% while providing average deviations under 0.1% for the test instances considered in this study. Considering the time performance of the path relinking method, we claim that it can be used in almost real-time production scheduling in a mixed-model manufacturing environment. Further analysis of the solutions reveals that the inventory level implied in the system by the solution of the batching problem using path relinking is statistically indifferent from that of the optimal solution. This result confirms the efficacy of the path relinking method for the problem in practical settings.

Focusing on the dynamic programming algorithm and reducing its computational time is possible with the implementation of effective bounding strategies. Improvements can be obtained by reducing both the number of dynamic programs successively solved and the time needed to solve a dynamic program. We consider this improvement as a future research direction on the topic. Another possible extension to this work would be to consider a more complicated manufacturing environment such as a flow shop, as the final level of the manufacturing system.

Appendix A: Relaxing the integrality constraints

If we relax the integrality requirement of q_i variables and ignore the constraints, we obtain

$$\text{Minimize } F' = Q \sum_{i \in N} \left(\frac{d_i}{q_i} \right)^2 - \frac{\sum_{i \in N} d_i^2}{Q}.$$

We first consider problems with $n=2$ products. For a given Q value, we can express the objective function in terms of only one variable, q_1 , as follows:

$$\text{Minimize } F' = Q \left(\left(\frac{d_1}{q_1} \right)^2 + \left(\frac{d_2}{Q - q_1} \right)^2 \right) - \frac{\sum_{i \in N} d_i^2}{Q}.$$

The second part of this function is constant. Therefore, we can drop the second part, as well as the Q constant in the first part, and state the objective function:

$$\text{Minimize } F' = \left(\frac{d_1}{q_1} \right)^2 + \left(\frac{d_2}{Q - q_1} \right)^2.$$

In order to find the minimum value of this function, we check the first and second derivatives with respect to q_1 :

$$\frac{dF'}{dq_1} = -\frac{2d_1^2}{q_1^3} + \frac{2d_2^2}{(Q - q_1)^3},$$

$$\frac{d^2F'}{dq_1^2} = \frac{6d_1^2}{q_1^4} + \frac{6d_2^2}{(Q - q_1)^4} > 0, \quad 0 < q_1 < Q.$$

The second derivative is positive in the definition domain of q_1 , thus the objective function is convex in q_1 and the minimizer can be found by setting the first derivative to 0:

$$\begin{aligned} \frac{dF'}{dq_1} &= 0 \\ \Rightarrow -\frac{2d_1^2}{q_1^3} + \frac{2d_2^2}{(Q - q_1)^3} &= 0 \\ \Rightarrow -\frac{2d_1^2}{q_1^3} + \frac{2d_2^2}{q_2^3} &= 0 \\ \Rightarrow \frac{q_1}{q_2} &= \left(\frac{d_1}{d_2}\right)^{2/3}. \end{aligned}$$

Now consider problems with $n > 2$. If we decide on all but two of the q_i values, then the final two variable values can be set to their optimal levels using the above relationship. Since we can select the two variables that will be decided on last, arbitrarily, then the relationship can be generalized to the $n > 2$ case with relative ease. The optimal level of variable q_i is found as follows:

$$\begin{aligned} q_j &= q_i \left(\frac{d_j}{d_i}\right)^{2/3}, \quad \forall j \\ \Rightarrow Q &= \sum_{j \in N} q_j = q_i \sum_{j \in N} \left(\frac{d_j}{d_i}\right)^{2/3} \\ \Rightarrow q_i &= \frac{Q}{\sum_{j \in N} (d_j/d_i)^{2/3}}, \quad \forall i. \end{aligned}$$

This result enables us to define optimal ratios of the variable values to the sum of the variable values:

$$r_i^* = \frac{q_i^*}{Q}.$$

Furthermore, we can calculate the r_i^* values in a pre-processing step and simplify the problem as follows.

Simplification of the objective function

$$\begin{aligned} \text{minimize } \frac{\sum_{i=1}^n (d_i/q_i)^2 (Q^2 - q_i^2)}{Q} &= \text{minimize } \frac{\sum_{i=1}^n (d_i/(r_i^* Q))^2 (1^2 - r_i^{*2}) Q^2}{Q} \\ &= \text{minimize } \frac{\sum_{i=1}^n (d_i/r_i^*)^2 (1^2 - r_i^{*2})}{Q} \\ &\equiv \text{maximize } Q. \end{aligned}$$

Simplification of the constraints

$$\begin{aligned}
 & s_i + p_i \frac{d_i}{q_i} \leq \frac{T}{Q}, \quad \forall i \in N \\
 \Rightarrow & \left(s_i + p_i \frac{d_i}{q_i} \right) Q \leq T, \quad \forall i \in N \\
 \Rightarrow & s_i Q + p_i \frac{d_i}{r_i^*} \leq T, \quad \forall i \in N \\
 \Rightarrow & Q \leq \frac{T - p_i(d_i/r_i^*)}{s_i}, \quad \forall i \in N.
 \end{aligned}$$

Appendix B: Deriving the average inventory levels

The solution to the batching phase is taken as an input by the sequencing phase. The batches are sequenced in such a way that q_i batches of product i are produced, each consisting of b_i units. Thus, the number of products to be manufactured is $q_i b_i$, for each i . Assuming a constant demand for the planning horizon of Q periods, demand per period is $q_i b_i / Q$. An ideal schedule would schedule product i to every Q/q_i th period, to produce b_i units. In this case, the inventory level of product i would be as depicted in figure B1, where the average inventory level for product i is $b_i/2$.

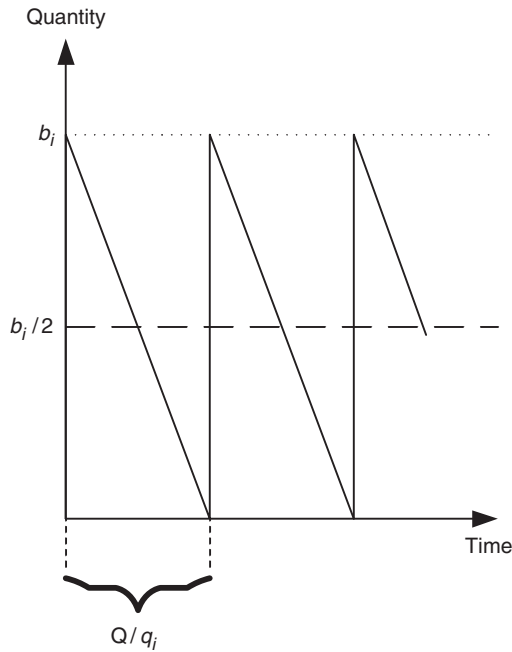


Figure B1. Average inventory level for product i .

The total average inventory level is $\sum_{i \in N} b_i/2$. We can eliminate the constant in the denominator and use $\sum_{i \in N} b_i$ as a measure of the average inventory level in the system.

References

- Amaral, A.R.S. and Wright, M., Experiments with strategic oscillation algorithm for the pallet loading problem. *Int. J. Prod. Res.*, 2001, **39**, 2341–2351.
- Chakravarty, A.K. and Shtub, A., Balancing mixed model lines with in-process inventories. *Man. Sci.*, 1985, **31**, 1161–1174.
- Dar-el, E.M. and Cothor, R.F., Assembly line sequencing for model mix. *Int. J. Prod. Res.*, 1975, **13**, 463–477.
- Dar-el, E.M. and Rabinovitch, M., Optimal planning and scheduling of assembly lines. *Int. J. Prod. Res.*, 1988, **26**, 1433–1450.
- Ding, F.Y. and Cheng, L., A simple sequencing algorithm for mixed-model assembly lines in just-in-time production systems. *Oper. Res. Lett.*, 1993, **13**, 27–36.
- Dowland, K.A., Nurse scheduling with tabu search and strategic oscillation. *Eur. J. Oper. Res.*, 1998, **106**, 393–407.
- Glover, F., Heuristics for integer programming using surrogate constraints. *Dec. Sci.*, 1977, **8**, 156–166.
- Glover, F., A template for scatter search and path relinking. In *Selected Papers from the Third European Conference on Artificial Evolution*, Lecture Notes in Computer Science, edited by J.K. Hao, E. Lutton, E. Ronald, M. Schoenauer and D. Snyers, Vol. 1354, pp. 13–54, 1998 (Springer: Berlin).
- Glover, F., Scatter search and path relinking. In *New Ideas in Optimization*, edited by D. Corne, M. Dorigo and F. Glover, pp. 297–316, 1999 (McGraw Hill: New York).
- Glover, F., Multi-start and strategic oscillation methods—Principles to exploit adaptive memory: a tutorial on unexplored opportunities. In *Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research*, edited by M. Laguna and J.L. Gozales Velarde, pp. 1–24, 2000 (Kluwer Academic: Dordrecht).
- Goldberg, D.E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, 1989 (Addison-Wesley: Reading, MA).
- Holland, J., *Adaptation in Natural and Artificial Systems*, 1975 (The University of Michigan Press: Ann Arbor, MI).
- Kelly, J.P., Golden, B.L. and Assad, A.A., Large-scale controlled rounding using tabu search with strategic oscillation. *Ann. Oper. Res.*, 1993, **41**, 69–84.
- Kubiak, W. and Sethi, S., A note on ‘level schedules for mixed-model assembly lines in just-in-time production systems’. *Man. Sci.*, 1991, **37**, 121–122.
- Macaskill, J.L.C., Production-line balances for mixed-model lines. *Man. Sci.*, 1972, **19**, 423–434.
- McMullen, P.R., JIT sequencing for mixed-model assembly lines with setups using tabu search. *Prod. Plan. & Con.*, 1998, **9**, 504–510.
- McMullen, P.R., An ant colony optimization approach to addressing a JIT sequencing problem with multiple objectives. *Art. Intell. Eng.*, 2001a, **15**, 309–317.
- McMullen, P.R., An efficient frontier approach to addressing JIT sequencing problems with setups via search heuristics. *Comp. Ind. Eng.*, 2001b, **41**, 335–353.
- McMullen, P.R., A Kohonen self-organizing map approach to addressing a multiple objective, mixed-Model JIT sequencing problem. *Int. J. Prod. Econ.*, 2001c, **72**, 59–71.
- McMullen, P.R. and Frazier, G.V., A simulated annealing approach to mixed-model sequencing with multiple objectives on a just-in-time line. *IIE Trans.*, 2000, **32**, 679–686.
- McMullen, P.R., Tarasewich, P. and Frazier, G.V., Using genetic algorithms to solve the multi-product JIT sequencing problem with set-ups. *Int. J. Prod. Res.*, 2000, **38**, 2653–2670.

- Miltenburg, J., Level schedules for mixed-model assembly lines in just-in-time production systems. *Man. Sci.*, 1989, **35**, 192–207.
- Miltenburg, J., Steiner, G. and Yeomans, S., A dynamic programming algorithm for scheduling mixed-model just-in-time production systems. *Math. Comp. Mod.*, 1990, **13**, 57–66.
- Osman, I.H. and Laporte, G., Metaheuristics: a bibliography. *Ann. Oper. Res.*, 1996, **63**, 513–623.
- Reeves, C.R., *Modern Heuristic Techniques for Combinatorial Problems*, 1993 (Wiley: New York).
- Reeves, C.R., Genetic algorithms for the operations researcher. *INFORMS J. Comp.*, 1997, **9**, 231–250.
- Thomopoulos, N.T., Line balancing-sequencing for mixed-model assembly. *Man. Sci.*, 1967, **14**, B59–B75.
- Thomopoulos, N.T., Mixed model line balancing with smoothed station assignments. *Man. Sci.*, 1970, **16**, 593–603.
- Yavuz, M. and Tufekci, S., The single-level batch production smoothing problem: an analysis and a heuristic solution. Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL, No. 05, 2004a.
- Yavuz, M. and Tufekci, S., Some lower bounds on the mixed-model level-scheduling problems, in *Proceedings of the 10th International Conference on Industry, Engineering and Management Systems*, 2004b.