Modeling and Solving a Cyclic and Batching Scheduling Problem with Two Types of Setups

Subhamoy Ganguly
Leeds School of Business, University of Colorado, subhamoy.ganguly@colorado.edu

Manuel Laguna
Leeds School of Business, University of Colorado, laguna@colorado.edu

Abstract

Production systems with closed-loop facilities must deal with the problem of sequencing batches in consecutive loops. We study a problem encountered in a production facility in which plastic parts of several shapes must be painted with different colors to satisfy the demand given by a set of production orders. The shapes and the colors produce a dual-setup problem that to the best of our knowledge has not been considered in the literature. We formulate the problem as a mixed-integer program and discuss the limitations of this approach as a viable solution method. We then describe two alternative solution approaches that are heuristic in nature: one specialized procedure developed from scratch and the other one built in the framework of commercial software. Our computational experiments are designed to assess the advantages and disadvantages of both approaches.

Version: March 8, 2012

1. Introduction

Many production systems with closed-loop facilities must deal with the problem of scheduling batches in consecutive loops (e.g. electrolytic painting of automotive parts in closed conveyors and cyclic painting of metallic furniture). We study such a scheduling problem, which arises in a production system at a manufacturing facility for plastic auto parts. The system combines cyclic and batching scheduling to paint rearview mirrors of varying geometries and colors. The paint line consists of a moving train that forms a continuous loop and contains a fixed number of spaces (positions). A jig is placed in each position in order to hang one or more parts. The sequence in which the parts are hung on the jig determines the number of setups that are incurred. A setup occurs when there is a change in the part geometry and also when there is a change in the color.

Each part geometry requires a special jig but the same jig can be used to paint parts of different colors. Therefore, the jig must be changed every time the geometry of the part to be painted changes. The jigs do not have to be changed when the color changes and the geometry of the parts remains the same. However, a change of color requires the use of a solvent to clean the pipes in which the paint flows. The painting area is located at a fixed position in the line and the problem is to minimize a cost function associated with the setups caused by changing both colors and geometries.

A simple example may be used to illustrate the problem. Suppose that 6 distinct products (labeled A through F) with the characteristics and demand values (in parentheses) shown in Table 1 must be painted. For instance, there is a demand of 3 parts for product A, which has color 1 and geometry 1. Also suppose that the paint line is such that there are 5 positions per loop (PPL). Therefore, 3 loops are necessary to paint all 15 parts (i.e., the sum of all the demand values in Table 1). A possible solution to this problem is shown in Figure 1, where the product labels are used to indicate the characteristics of the parts assigned to each position.

	Geometry		
Color	**1**	**2**	**3**
1	A(3)	B(4)	C(2)
2		D(3)	
3		E(2)	F(1)

Table 1. Characteristics and demand values (number of parts) of 6 products (A to F)

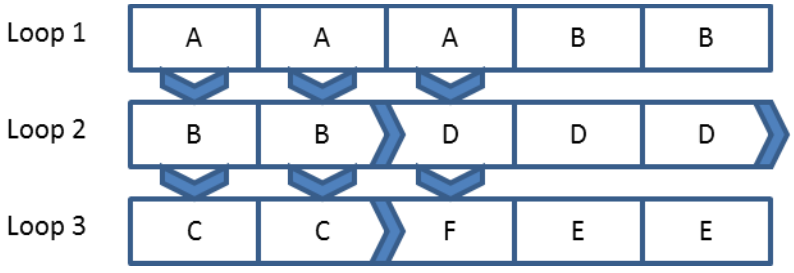

Figure 1. Positioning of 15 parts in 3 loops with $PPL = 5$

The thick arrows in Figure 1 indicate the setups that occur when placing the products in the specified positions. There are a total of three color setups and six jig setups. Solvent and labor costs are incurred when color setups occur. When the jig changes from one loop to the next, a labor cost is incurred because labor is required to make the change. Following the schematic representation of a solution to the problem presented in Figure 1, we will refer to a change of color as a *horizontal* setup while a change of jig as a *vertical* setup.

Scheduling problems with setups have received considerable attention in the operations research literature (Cheng *et al*. 2000, Zhu and Wilhelm 2006, Allahverdi *et al*. 1999, Allahverdi *et al*. 2008); however, the particular problem that we have described above — to the best of our knowledge — has not been discussed, with the exception of Garcia-Sabater *et al*. (2008). This work, however, has a narrow scope because it is limited to describing the problem, introducing a mathematical formulation and solving a single instance of the problem with commercial software. While some work has been devoted to scheduling problems with sequence dependent setups (Szwarc and Gupta 1987, Laguna 1999, Rajendran and Ziegler 2003, Ruiz *et al*. 2005 and Jungwattanakit *et al*. 2009), the extent of sequence dependence addressed in these works is limited to adjacent jobs. In other words, these works have typically assumed that the setup cost incurred by the current job is dependent only on the current job and its immediate predecessor. In contrast, this paper addresses not only the horizontal setup cost due to the current job and its immediate predecessor, but also the vertical setup cost incurred by the current job and the job that occupied the same position in the prior loop. As we will show here, consideration of these two types of setup costs adds significant complexity to the problem.

Balance of the paper is organized as follows. In the next section, we present a mathematical programming formulation of the problem and show that it is not practical to use this approach to solve industrial scale problems. Thereafter, we analyze bounds and the landscape of the problem, an analysis that helps us choose the appropriate heuristic methods. Then, we present two alternative heuristic approaches: one based on commercially available software and the other a procedure developed specifically for the problem at hand. Finally, we present experimental results and discuss the relative merits of the alternative heuristic approaches.

2. Notation and Mathematical Programming Formulation

The problem that we described above may be formulated as a mixed integer programming (MIP) model. For this purpose, we employ the following notation that assumes that a product is unique combination of color and geometry:

C : set of colors
G : set of geometries
S : set of products $\subset (C \times G)$
CC : set of pairs of dissimilar colors $\subset (C \times C)$
GG : set of pairs of dissimilar geometries $\subset (G \times G)$
LB_c : lower bound on number of color changeovers, default 0
LB_g : lower bound on number of geometry changeovers, default 0
LB : lower bound on objective-function value (from the best node in the branch-and-bound tree, rounded up to nearest feasible objective value)

$h_{ii'}$: (horizontal) setup cost of changing from color i to i'
$v_{jj'}$: (vertical) setup cost of changing from geometry j to j'
$d_{(i,j)}$: demand for product with color i and geometry j, for $(i,j) \in S$
PPL : parts per loop
n : number of positions, where $n = \sum_{(i,j)\in S} d_{ij}$

The MIP model is formulated with the following variables:

$$x_{ijk} = \begin{cases} 1 \text{ if product } (i,j) \text{ is placed in position } k \\ 0 \text{ otherwise} \end{cases}$$

$$y_{ik} = \begin{cases} 1 \text{ if a part with color } i \text{ is located in position } k \\ 0 \text{ otherwise} \end{cases}$$

$$z_{jk} = \begin{cases} 1 \text{ if a part with geometry } j \text{ is located in position } k \\ 0 \text{ otherwise} \end{cases}$$

$$p_{ii'k} = \begin{cases} 1 \text{ if color changes from } i \text{ to } i' ((i,i') \in CC) \text{ between positions } k \text{ and } k+1 \\ 0 \text{ otherwise} \end{cases}$$

$$q_{jj'k} = \begin{cases} 1 \text{ if geometry changes from } j \text{ to } j' ((j,j') \in GG) \text{ between positions } k \text{ and } k+PPL \\ 0 \text{ otherwise} \end{cases}$$

The mathematical formulation has the following form:

Minimize $\sum_{k=1}^{n-1} \sum_{(i,i')\in CC} h_{ii'} p_{ii'k} + \sum_{k=1}^{n-PPL} \sum_{(j,j')\in GG} v_{jj'} q_{jj'k}$ (1)

Subject to

$\sum_{(i,j)\in S} x_{ijk} = 1$ $k = 1, \dots, n$ (2)

$\sum_{k=1}^{n} x_{ijk} = d_{(i,j)}$ $\forall (i,j) \in S$ (3)

$y_{ik} = \sum_{j:(i,j)\in S} x_{ijk}$ $\forall i: (i,j) \in S, k = 1, \dots, n$ (4)

$z_{jk} = \sum_{i:(i,j)\in S} x_{ijk}$ $\forall j: (i,j) \in S, k = 1, \dots, n$ (5)

$p_{ii'k} \geq y_{ik} + y_{i'k+1} - 1$ $\forall (i,i') \in CC, k = 1, \dots, n-1$ (6)

$q_{jj'k} \geq z_{jk} + z_{j'k+PPL} - 1$ $\forall (j,j') \in GG, k = 1, \dots, n-PPL$ (7)

$x_{ijk} \in \{0,1\}$ $\forall (i,j) \in S, k = 1, \dots, n$ (8)

The objective is to minimize the total cost associated with the changes in color and jigs. Constraint set (2) enforces the assignment of a single product to each available position, while set (3) guarantees that the demand for each product is satisfied. Based on the values of the primary variable x, constraint sets (4) and (5) fully determine the values for the auxiliary variables y and z respectively. Constraint sets (6) and (7) capture the color changeovers p and geometry changeovers q respectively. Constraint set (8) enforces the integrality of the primary binary variables x. Note that the integrality of the auxiliary variables p, q, y, and z are implicitly enforced by the constraint sets (4) through (7).

Our MIP formulation is more general than the one introduced by Garcia-Sabater *et al*. (2008) because it allows for setup costs to depend on the products. Also, our modeling of the changeovers, represented

by constraints (4) through (7), is done without reference to color IDs, as done in Garcia-Sabater *et al*. (2008). While the MIP model above is a valid formulation of the problem, it has a weak LP-relaxation and experimentation shows that it takes a long time to converge to optimality except for very small instances. In preliminary experimentation, the LP relaxation often found optimal non-integer solutions with values of the y variables such as those shown in Table 2, without incurring any color changeover costs.

Position	Color 1	Color 2	Color 3
K	0.5	0.5	0.0
***k*+1**	0.0	0.5	0.5

Table 2. Values of y that are feasible for LP relaxation of the MIP model

A structure similar to the one in Table 2 was observed in geometry changeovers as well. To excise such non-integer solutions from the formulation, we added a set of cuts (9 and 10 below) that strengthen the LP relaxation and improve performance. Further, we added constraint sets (11) and (12) to provide lower bounds of color and geometry changeovers. The bounds LB_c and LB_g are obtained using the approach described in the next section. The bound on the objective function value (13) is added during the branch and bound process because, without loss of generality, we consider that both h and v are integers. Therefore, anytime that the lower bound (LB) on the total cost becomes fractional, the value is rounded up to the nearest integer value ($\lceil LB \rceil$), and thus helping with the convergence.

$$\sum_{i \neq i'} p_{ii'k} \geq y_{i'k+1} + \sum_{i \neq i'} y_{ik} - 1 \qquad \forall i' \in C, k = 1, \dots, n-1 \tag{9}$$

$$\sum_{j \neq j'} q_{jj'k} \geq z_{j'k+PPL} + \sum_{j \neq j'} z_{jk} - 1 \qquad \forall j' \in G, k = 1, \dots, n-PPL \tag{10}$$

$$\sum_{k=1}^{n-1} \sum_{(i,i') \in CC} p_{ii'k} \geq LB_c \tag{11}$$

$$\sum_{k=1}^{n-PPL} \sum_{(j,j') \in GG} q_{jj'k} \geq LB_g \tag{12}$$

$$\sum_{k=1}^{n-1} \sum_{(i,i') \in CC} h_{ii'} p_{ii'k} + \sum_{k=1}^{n-PPL} \sum_{(j,j') \in GG} v_{jj'} q_{jj'k} \geq \lceil LB \rceil \tag{13}$$

In the experimental section, we show that despite of our efforts to add cuts and improve bounds, our experience with the MIP formulation has severe limitations as a practical method for solving industrial scale instances of this problem for which $|C| \geq 10$, $|G| \geq 10$, and $PPL \geq 50$. We even considered a simplified MIP formulation (see the appendix) and found similar practical limitations.

3. Bounds and Landscape Analysis

Before embarking on the task of developing an approximate solution method we analyzed the structure of the problem with the goal of gaining additional insights as well as producing bounds for the two types of setups. Understanding the landscape associated with the objective function was part of this analysis.

If we consider that the setup cost is constant and does not depend on the specific colors or geometries involved in the change, then lower bounds on the number of setups of each type (i.e., LB_c and LB_g) may be used to calculate a lower bound on the cost, namely $hLB_c + vLB_g$. (For problems where it is

absolutely necessary to consider variable setup costs, the lower bound could be calculated using the minimum horizontal and vertical costs.) If vertical setups are ignored, a bound on the number of color setups is given by $LB_c = |C| - 1$. That is, in the absence of vertical setups, the optimal way of arranging the parts is by blocking all parts of the same color and sequentially process all blocks in any order, as long as all parts with the same color are processed together.

Now let us disregard the horizontal costs and focus on the vertical costs. A solution may be conceptualized as a two dimensional matrix with number of columns equal to PPL and number of rows equal to n/PPL. (See, for instance, Figure 1 where a solution to a problem with $n = 15$ and $PPL = 5$ is depicted.) The demand associated with each geometry j is given by $d_{(*,j)} = \sum_i d_{(i,j)}$. Define $m_j = mod\left(\frac{d_{(*,j)}}{n/PPL}\right)$. It is trivial to show that $LB_g = 0$ if $m_j = 0$ for all j. In this case, each part geometry is placed in as many columns as needed to meet the demand; each column includes only one geometry, and the number of geometry setups is zero. However, when $m_j \neq 0$ for at least one $j \in G$, geometry setups become unavoidable. This means that when horizontal costs are ignored, the problem of minimizing vertical costs can be reduced to a smaller problem in the following manner. For each geometry $j \in G$, $\left\lfloor \frac{d_{(*,j)}}{n/PPL} \right\rfloor$ gives the number of columns that may be fully occupied by these items without incurring any changeover. The reduced problem then consists of placing m_j units for each $j \in G$ in a matrix with the same number of rows as the original problem, i.e., n/PPL, and a reduced number of columns. The reduced number of columns, or reduced parts-per-loop ($RPPL$), is given by

$$RPPL = PPL - \sum_j \left\lfloor \frac{d_{(*,j)}}{n/PPL} \right\rfloor = \frac{\sum_j m_j}{n/PPL}$$

Moreover, since we are ignoring horizontal costs altogether, the variables and constraints related to color changeovers can be omitted from the formulation shown in the previous section. Thus, ignoring horizontal-costs and considering only geometry-costs allows us to reduce the optimization problem both in scale and complexity. The minimum number of geometry changeovers obtained by solving this reduced model is a true lower bound on the geometry-changeovers (LB_g).

While a lower bound on the total cost can be obtained by computing the number of setups separately, simple examples show that a solution method that focuses on each type of setup one at a time results in solutions that are not only suboptimal but also significantly inferior than those found by strategies that address both setups at the same time. That is, we have verified that a procedure that minimizes one type of setups first and then minimizes the second type (considering only solutions that do not increase the number of setups of the first type) is destined to perform poorly.

A natural representation of a solution to the problem consists of a vector π of size n, where $\pi(k)$ is the index of the product in position k. Simplifying our previous notation, let d_p be the demand of product p (that is, we assume that the index of the product is enough to specify its color and geometry). If for a given problem instance, the demand for each product is 1 (i.e., $d_p = 1$ for all p), then $n = |S|$ and the solution space consists of $n!$ solutions. In general, however, the solution space is given by the following product:

$$\prod_{p=1}^{p=|S|} C_{d_p}^{n-\sum_{p'=0}^{p'=p-1} d_{p'}}$$

Suppose that $h = v = 1$. Then, the total cost associated with a given solution is the same as the total number of setups necessary to implement the solution. An upper bound on the number of horizontal setups (and therefore on the horizontal cost when $h = 1$) is $n - 1$. This occurs in the worst possible scenario, which forces a change of color after every position in the solution. Likewise, $n - PPL$ is an upper bound on the number of vertical setups. This occurs when the parts are arranged in such a way that changing a jig is necessary in every position after each loop. Therefore, when considering unit costs, an upper bound on the total cost is given by $2n - PPL - 1$.

This analysis shows that the natural solution representation provided by the π vector with n positions results in a solution space with a *flat landscape*, that is, one for which many solutions have the same objective function value. For instance, using the data in Table 1 —which corresponds to a problem with 5 products (i.e., $|S| = 5$), a total demand of 15 (i.e., $n = 15$) and a configuration with 5 parts per loop (i.e., $PPL = 5$)— it is easy to verify that, if $h = v = 1$, the number of unique objective function values is only 2.2% of the size of the solution space. The solution space associated with π consists of 1002 solutions. The upper bound on the total cost is 24 (i.e., 2*15-5-1) and the lower bound is 2, resulting in at most 23 unique objective values for 1002 solutions. Similar results are obtained with problems of larger size.

4. Heuristic Solution Methods

In order to tackle instances that are typical in real settings, we approached the problem with metaheuristic technology. Practitioners, whenever possible, tend to prefer the use of commercial software instead of developing specialized procedures for each optimization problem that they face. The effort required to adapt commercial software to produce solutions of acceptable quality, however, depends on the optimization problem of interest. Hence, for certain types of problems, development of a specialized metaheuristic procedure that is tailored to the problem yields better results. In this work, we take both avenues and compare their merits. In particular, we compare the adaptation of the commercial software OptQuest with a specialized VNS (variable neighborhood search) and discuss the pros and cons of following these approaches. The choice of our particular implementation of VNS over other metaheuristics was driven by the structure of the problem landscape discussed in the last section.

4.1. OptQuest Adaptation

OptQuest is a general-purpose optimization system that is built on a platform of metaheuristic technologies (Laguna 2011). The main engine consists of an implementation of scatter search that includes specialized memory structures typically found in tabu search implementations. OptQuest is designed to operate in a black-box structure and therefore it does not take advantage of the specific characteristics of the problem that it is attempting to solve. The main choice that OptQuest users must make relates to the selection of a solution representation. The OptQuest system offers several choices that include continuous, discrete and binary variables. A basic OptQuest implementation is as follows:

1. Define n discrete variables within the range $(1, |S|)$
2. Create an `Evaluate()` method that calculates the objective function value associated with solutions represented by π

The OptQuest Engine calls the `Evaluate()` method every time a solution needs to be evaluated. The optimization problem for OptQuest consists of selecting the value of n discrete variables that are bounded between 1 and $|S|$. Since no information is given to OptQuest about the demand for each product, the system may produce infeasible solutions. Therefore, the `Evaluate()` method must be designed in a way that infeasible solutions are either penalized or are mapped into feasible ones. Note that the absence of this information makes the space where OptQuest searches for a solution much larger than the original. The search space contains $|S|^n$ solutions.

Our mapping from an infeasible solution to a feasible one is straightforward. Given a solution π, we first evaluate its objective function value and at the same time we calculate $parts(p)$, the number of parts of product p in the solution. If $parts(p) > d_p$ then product p has a surplus of parts in the solution. If $parts(p) < d_p$ then product p has a shortage of parts in the solution. Because the solution has n positions and n is equal to the total demand for parts, a feasible solution is one where $parts(p) = d_p$ for all p. We identify the best exchange of a product that has a surplus with a product that has a shortage. The best exchange is the one that either decreases the objective function value the most or increases it the least. After the change is made, the $parts(p)$ values are updated and the procedure searches for the next best exchange. The process ends when there are no products with either surplus or shortages.

1. Calculate the objective function and $parts(p)$

While ($parts(p) \neq d_p$ for at least one p)

2. Find the best exchange of a part corresponding to a product with a surplus and a part corresponding to a product with a shortage
3. Make the change, update the objective function value and $parts(p)$

Figure 2. Feasibility mapping within the `Evaluate()` method

The mapping within the `Evaluate()` method (shown in Figure 2) changes the values of the decision variables for infeasible solutions that are turned into feasible ones. When this occurs, there are two choices: 1) the updated solution along with the objective function value are returned to the OptQuest engine or 2) only the objective function value is returned to the OptQuest search engine. Previous experience with OptQuest and similar mapping strategies have shown that performance is enhanced when the second option is used. The first option leads to a highly focused search that lacks diversity and this is why it is not preferred.

In addition to evaluating the objective function of trial solutions built by OptQuest and performing feasibility mapping, the `Evaluate()` method may be employed to execute neighborhood explorations. For instance, Ugray, *et al.* (2007) combine OptQuest with a gradient-based local solver for nonlinear (NLP) programming problems. In this implementation, the NLP solver not only turns infeasible solutions (constructed by the OptQuest engine) into feasible ones but also searches for improved outcomes.

When solutions are modified by the NLP solver, the updated variable values are returned to OptQuest. In order to avoid premature convergence of the search, the NLP solver is called selectively. The procedure uses both a distance and a merit filter. As stated by the authors, "the distance filter helps insure that these starting points are diverse, in the sense that they are not too close to any previously found local solution." Likewise, "the merit filter helps insure that the starting points have high quality, by not starting from candidate points whose exact penalty function value is greater than a threshold." In the Ugray, *et al.* (2007) design, OptQuest is treated as an engine whose goal is to provide a set of high quality and diverse starting points for a local optimizer.

We follow a similar approach and in addition to mapping infeasible solutions into feasible ones, we add an improvement procedure, which is run immediately after a feasible solution is identified within the call to the `Evaluate()` method. The improvement method is a simple short-term memory tabu search (see Figure 3). An iteration involves evaluating all possible swaps of different parts. The non-tabu swap that has the best move value (i.e., that decreases the current objective function the most or increases it the least) is executed. A $swap(k_1, k_2)$ is tabu if the parts in positions k_1 and k_2 have been swapped in the last $|S|$ iterations. A tabu swap is executed if it leads to a new incumbent solution. After a swap of the parts in positions k_1 and k_2, the (k_1, k_2) pair is made tabu-active for $|S|$ iterations. The search stops after $|S|$ iterations have been performed without finding a new incumbent solution. Note that the incumbent solution is the best solution found during the current tabu search, that is, during the search that starts in the current call to the `Evaluate()` method. The overall best solution is the best incumbent found after all the OptQuest iterations have been performed.

1. Make current solution the incumbent

While (new incumbent solution found in the last $|S|$ iterations)

2. Find the best non-tabu swap (waving the tabu status if the swap leads to a new incumbent)
3. Make the swap, update the tabu memory and the incumbent solution if appropriate

Figure 3. Short-term tabu search within the `Evaluate()` method

The procedures in Figures 2 and 3 require a very modest implementation effort. In fact, our implementation in C# consists of fewer than 100 lines of code that include both the feasibility mapper and the improvement method. The rest of our code is standard to formulating optimization problems with OptQuest and was in fact adapted from one of the examples that are distributed with the OptQuest engine. We mention this because a discussion that is often omitted, but that we believe is important, relates to the total amount of effort involved in developing solution methods for practical problems. We now turn our attention to the description of a specialized search heuristic that we have developed for the dual-setup problem.

4.2. Variable Neighborhood Search (VNS) Approach

As an alternative to the method described in the previous section, we implemented a multi-start VNS procedure. VNS is based on the notion of creating multiple neighborhoods of varied complexity (Talbi 2009). The main elements of our implementation are:

1. Pseudo-greedy constructions
2. Swap neighborhood
3. Insert neighborhood
4. Shake

The search starts from multiple points that are constructed as follows. The starting point alternates between a completely random solution and a greedy random solution. Our experiments showed that this mix of starting points helped obtain solutions of higher quality by combining the diversity created by totally random solutions with the starting points created by the greedy random solutions. The greedy random construction begins by assigning a random part to the first position. Then, the subsequent positions are filled sequentially by the remaining parts using the following greedy rule. The most preferred part for a position is one which will not result in any changeover, e.g. for position j, the color should match with position $j-1$ (when $j>1$), and the geometry should match with position $j-PPL$ (when $j>PPL$). The next preferred parts are those which avoid one type of changeover, either color or geometry. When no preferred parts are available, a part is chosen at random from those with unfulfilled demand. The primary neighborhood search in our implementation consists of swaps of parts that are not of the same product. We again represent a solution with a vector π of size n, resulting in an upper bound of at most $(n^2-n)/2$ moves. This only occurs when $d_p=1$ for all p. In fact, the total number of swaps —in terms of the demand for each product— is given by the following formula:

$$\sum_{p=1}^{|S|-1} d_p \left(\sum_{p'=p+1}^{|S|} d_{p'} \right)$$

The value of a swap is easy to compute. The calculation consists of at most sixteen simple operations, eight to calculate the "savings" of removing the parts from the current positions and eight to calculate the "cost" of inserting the parts in their new positions. Note that each part has at most four immediate neighboring parts (left, right, up and down, if we picture a solution as given in Figure 1).

In addition to swap, our VNS uses inserts. An insert may be thought of as a partial swap in the sense that a single part is taken from its current position and is transferred to another position in the solution. Here again, we don't have to examine all inserts because some of them do not produce a change of the current solution. For instance, consider the solution depicted in Figure 1 and its representation as a vector of product IDs:

$$\pi = (A, A, A, B, B, B, B, D, D, D, C, C, F, E, E)$$

Clearly, inserting $\pi(1)=A$ between positions 2 and 3 does not change the solution because $\pi(2)=\pi(3)=A$. In fact, the solution does not change either if $\pi(1)$ is inserted between positions 3 and 4.

This shows that the set of "valid" inserts depends on the configuration of the current solution. The number of inserts however is in the same order of magnitude as the number of swaps. The VNS operates by applying the primary neighborhood to the initial solution. The procedure keeps iterating as long as the incumbent solution improves. (The definition of incumbent is the same as in the OptQuest adaptation and refers to the best solution found from the current starting point.) When no more improvement is possible, the neighborhood is switched to inserts. Again, this neighborhood remains operational until either a new incumbent solution is found or the entire insert neighborhood is explored and no improvement of the incumbent solution is achieved. If a new incumbent solution is found, then the neighborhood search reverts back to the primary mechanism (that is, swaps). However, if the exploration of the insert neighborhood ends without improving the incumbent solution then the shake mechanism is invoked. Our "shaking" strategy simply consists of applying random inserts to the incumbent solution. The number of items involved in the random inserts increases from 1 up to 4 as the number of shakings progresses. The entire process ends when there no new incumbents are found after a predetermined number of shakes, which in our case we have set to 40.

Compared to the OptQuest adaptation, the VNS implementation requires more effort. However, it is still based on fairly straightforward mechanisms and the tuning involved choosing the right mix of starting points (alternating between greedy random and totally random), the number of starts (500), the number of shakes (40), and the number of random inserts to be made during the shakes (1 to 4). The merit of the two approaches is analyzed in the following section.

5. Computational Experiments

Performances of the multi-start VNS procedure and the Optquest application with the tabu search `Evaluate()` method were compared employing randomly generated instances of various complexities listed in Table 2. For the small and medium size instances (total demand of 30 and 40 parts) we have been able to obtain optimal solutions by solving the MIP formulation (augmented with the (9)-(13) cuts) presented in Section 2 with CPLEX. Thus, for these instances, we have been able to verify the heuristic procedures' ability to reach optimality. The approximate time to find these optimal solutions with a CPLEX 11.2.1 solver running on a Windows Enterprise Server with 3.16 GHz dual processors and 32 GB RAM is given in the last column of Table 2.

Type	Demand	Colors	Geometries	Products	Cplex Time
A	30	3	3	5	3 min
B	40	3	3	8	30 min
C	40	5	5	8	10 hours
D	60	5	5	15	—

Table 2. Characteristics of problem instances

We generated ten instances of each type and solved them with $PPL = 5$ and $PPL = 10$. The results found by setting $h = v = 1$ are summarized in Table 3 (for $PPL = 5$) and Table 4 (for $PPL = 10$). OptQuest was set to perform 1000 iterations while VNS was set to execute 500 starts.

		Multistart VNS			OptQuest/TS			
Prob	**Opt**	**Iter**	**Time**	**Obj**	**Iter**	**Time**	**Obj**	**OQ-VNS**
A-01	6	3	0.99	6	140	1.59	6	0
A-02	4	0	0.12	4	9	0.21	4	0
A-03	3	0	0.13	3	30	0.27	3	0
A-04	9	0	0.08	9	20	0.26	9	0
A-05	6	2	0.37	6	29	0.27	6	0
A-06	5	0	0.12	5	2	0.20	5	0
A-07	7	3	0.45	7	35	0.27	7	0
A-08	9	0	0.16	9	20	0.26	9	0
A-09	3	0	0.18	3	12	0.22	3	0
A-10	6	0	0.10	6	11	0.22	6	0
B-01	7	1	1.40	7	42	0.42	7	0
B-02	5	12	3.46	5	791	4.86	5	0
B-03	5	0	0.28	5	158	2.01	5	0
B-04	4	10	3.43	4	483	3.21	4	0
B-05	5	0	0.26	5	24	0.33	5	0
B-06	4	4	1.86	4	83	0.59	4	0
B-07	4	34	10.75	4	11	0.26	5	1
B-08	6	2	1.05	6	338	2.77	6	0
B-09	5	2	0.62	5	182	1.91	5	0
B-10	4	8	2.52	4	143	1.88	4	0
C-01	11	0	0.78	11	11	0.26	11	0
C-02	14	5	2.11	14	55	0.44	14	0
C-03	12	0	0.24	12	1	0.17	12	0
C-04	9	0	0.29	9	21	0.31	9	0
C-05	18	0	0.27	18	7	0.23	18	0
C-06	14	0	0.21	14	32	0.35	14	0
C-07	12	0	0.15	12	3	0.21	12	0
C-08	15	0	0.16	15	122	0.67	15	0
C-09	16	2	0.53	16	2	0.21	16	0
C-10	8	0	0.22	8	1	0.16	8	0
D-01	—	33	25.90	12	115	3.54	14	2
D-02	—	307	90.01	13	394	12.45	14	1
D-03	—	200	47.96	14	544	15.43	15	1
D-04	—	85	289.15	13	341	10.13	15	2
D-05	—	334	47.89	12	505	14.41	13	1
D-06	—	11	13.07	13	259	7.68	14	1
D-07	—	114	7.78	13	598	16.45	14	1
D-08	—	36	9.68	14	233	6.83	15	1
D-09	—	23	3.17	15	942	25.62	16	1
D-10	—	19	96.05	14	523	14.20	15	1

Table 3. Results with $PPL = 5$

		Multistart VNS			OptQuest/TS			
Prob	**Opt**	**Iter**	**Time**	**Obj**	**Iter**	**Time**	**Obj**	**OQ-VNS**
A-01	6	0	0.14	6	2	0.20	6	0
A-02	3	0	0.12	3	148	1.36	3	0
A-03	3	0	0.08	3	37	0.26	3	0
A-04	8	0	0.06	8	1	0.16	8	0
A-05	6	0	0.11	6	3	0.20	6	0
A-06	4	0	0.10	4	2	0.20	4	0
A-07	6	0	0.11	6	15	0.25	6	0
A-08	8	0	0.08	8	1	0.16	8	0
A-09	4	0	0.07	4	10	0.22	4	0
A-10	7	0	0.10	7	18	0.25	7	0
B-01	5	6	1.90	5	699	3.92	5	0
B-02	5	8	2.03	5	74	0.50	5	0
B-03	5	2	0.65	5	49	0.41	5	0
B-04	3	1	0.81	3	125	1.73	3	0
B-05	4	0	0.29	4	71	0.50	4	0
B-06	4	6	1.70	4	99	0.60	4	0
B-07	3	0	0.25	3	133	1.75	3	0
B-08	4	2	0.89	4	708	3.69	4	0
B-09	4	0	0.31	4	32	0.34	4	0
B-10	4	6	1.73	4	269	2.36	4	0
C-01	10	0	0.34	10	180	1.95	10	0
C-02	15	1	0.40	15	37	0.36	15	0
C-03	10	15	3.70	10	159	1.77	10	0
C-04	8	10	2.68	8	91	0.53	9	1
C-05	16	2	0.63	16	88	0.55	16	0
C-06	14	9	2.46	14	2	0.21	14	0
C-07	11	0	0.19	11	80	0.52	11	0
C-08	13	11	3.19	13	12	0.24	14	1
C-09	16	0	0.27	16	16	0.28	16	0
C-10	6	0	0.19	6	23	0.31	6	0
D-01	—	33	304.46	14	82	2.31	15	1
D-02	—	307	67.42	13	84	2.53	16	3
D-03	—	200	258.95	15	998	25.89	17	2
D-04	—	85	178.29	15	60	1.67	17	2
D-05	—	334	106.01	13	200	5.78	15	2
D-06	—	11	112.27	13	412	10.20	15	2
D-07	—	114	119.68	14	253	7.21	16	2
D-08	—	36	96.08	16	75	1.99	17	1
D-09	—	23	372.48	17	65	1.69	18	1
D-10	—	19	405.61	15	498	12.42	16	1

Table 4. Results with $PPL = 10$

Tables 3 and 4 show that the Multistart VNS is able to find all known optima. OptQuest fails to match the optimal solution to problem B-07 with $PPL = 5$ and C-04 and C-08 with $PPL = 10$ The specialized VNS procedure finds better solutions than the OptQuest adaptation for all instances in the D set. Our

conjecture is that the flat nature of the landscape is a primary reason behind the superior performance of VNS; in fact, VNS performed better than other metaheuristics that we have attempted on this problem. Navigating different neighborhoods, which is the essence of VNS, helps explore diverse areas of the flat landscape. For the same reason, i.e. flatness of the landscape, multiple starts make the procedure more effective. In fact, depending on the size and complexity of the problem instance, the number of starts to be used is an important parameter to set. The **Iter** column in Tables 3 and 4 refer to the earliest iteration, i.e. start, that found the best solution. For smaller and simpler instances (types A, B, and C), an optimal solution was found within 50 iterations, while for the larger type-D instances, better solutions were found even after 300 iterations.

In terms of computational times, both procedures performed at a similar level for problem types A, B and C. Computational times for the VNS in problem instances of type D are significantly larger than the OptQuest times. This is because VNS finds the best solutions late in the search and each VNS start is computationally more demanding that an OptQuest iteration. This may seem like an unfair comparison but we have verified that additional OptQuest iterations do not improved the results reported in Tables 3 and 4 for the instances type D.

We performed one more experiment with significantly larger instances. Specifically, we generated a set of ten problem instances with 10 colors, 10 geometries, 50 products and a total demand of 300 parts. Hence, the average demand is 6 parts per product. The PPL was set to 50, and the number of starts for VNS and the number of iterations for OptQuest were both set to 100. As expected, the gap between the quality of the solutions found with VNS and the quality of the solutions found with OptQuest widen. In 4 out of the 10 instances, OptQuest's solutions had at least 10% more setups than VNS solutions and in 8 out of 10 the number of setups in the OpQuest solutions was at least 5% more than in the VNS solutions. The maximum difference in the number of setups was 18%, which occurred once. VNS required 2.5 more computational time than OptQuest but both procedures are capable of delivering solutions within the timeframe needed in the application (which does not require an online real-time response).

6. Conclusions

We have described a sequencing problem with two types of setups. A MIP formulation of the problem is presented and used to find optimal solutions to a set of problem instances. We also describe the adaptation of OptQuest and the development of a multistart VNS as two alternative heuristic approaches to the problem. While the MIP formulation works well for very small size instances, heuristic approaches are essential to obtain reasonably good solutions for industrial scale problem instances. The OptQuest adaptation required less effort to develop compared to the multistart VNS, which is tailored to the problem of interest. We show that the problem inherently has a very flat landscape, which motivated the choices we made to create a broader exploration of diverse regions of the landscape; navigating multiple neighborhoods through VNS, and using multiple starts proved to be effective strategies

In our implementation, both the OptQuest adaptation and the multistart VNS performed reasonably well for small size instances. However, for larger scale problems, the multistart VNS performed better, indicating that for this type of problem, it might be worthwhile to invest in a heuristic custom-made for the problem. Experiments show that for large problems, savings of at least 10% are likely and savings of at least 5% are very likely when employing the specialized VNS procedure instead of the OptQuest adaptation. However, these predicted savings must be thought of as the benefits that help amortize the difference in development cost between the specialized method and the adaptation of the commercial software.

References

Allahverdi, A., J. N. D. Gupta, T. Aldowaisan (1999) "A Review of Scheduling Research involving Setup Considerations," *OMEGA The International Journal of Management Sciences,* vol. 27, no. 2, pp. 219-239.

Allahverdi, A., C. T. Ng, T. C. E. Cheng and M. Y. Kovalyov (2008) "A Survey of Scheduling Problems with Setup Times or Costs," *European Journal of Operational Research,* vol. 187, no. 3, pp. 985-1032.

Cheng, T. C. E., J. N. D. Gupta and G. Wang (2000) "A Review of Flowshop Scheduling Research with Setup Times," *Production and Operations Management,* vol. 9, no. 3, pp. 262-282.

Garcia-Sabater, J. P., C. Andres, C. Miralles and J. J. Garcia-Sbater (2008) "An Application Oriented Approach for Scheduling a Production Line with Two Dimensional Setups, in *Proceedings of the 11th International Workshop on Project Management and Scheduling*, F. Şerifoğlu and Ü. Bilge (eds.), Istanbul, Turkey, pp. 90-93.

Jungwattanakit, J., M. Reodecha, P. Chaovalitwongse and F. Werner (2009) "A Comparison of Scheduling Algorithms for Flexible Flow Shop Problems with Unrelated Parallel Machines, Setup Times and Dual Criteria," *Computers and Operations Research,* vol. 36, no. 2, pp. 358-378.

Laguna, M. (1999) "A Heuristic for Production Scheduling and Inventory Control in the Presence of Sequence-dependent Setup Times," *IIE Transactions,* vol. 31, no. 2, pp. 125-134.

Laguna, M. (2011) "OptQuest: Optimization of Complex Systems," White Paper, OptTek Systems Inc., http://www.opttek.com/white-papers.

Rajendran, C. and H. Ziegler (2003) "Scheduling to Minimize the sum of Weighted Flowtime and Weighted Tardiness of Jobs in a Flowshop with Sequence-dependent Setup Times," *European Journal of Operational Research,* vol. 149, no. 3, pp. 513-522.

Ruiz, R., C. Maroto, T. and J. Alcaraz (2005) "Solving the Flowshop Scheduling Problem with Sequence Dependent Setup Times using Advanced Metaheuristics," *European Journal of Operational Research,* vol. 165, no. 1, pp. 34-54.

Szwarc, W. and J. N. D. Gupta (1987) "A Flow-Shop Problem with Sequence-Dependent Additive Setup Times," *Naval Research Logistics,* vol. 34, no. 5, pp. 619-627.

Talbi, G. (2009) *Metaheuristics – From Design to Implementation*, John Wiley & Sons: Hoboken, pp. 150-154.

Ugray, Z., L. Lasdon, J. Plummer, F. Glover, J. Kelly and R. Martí (2007) "Scatter Search and Local NLP Solvers: A Multistart Framework for Global Optimization," *INFORMS Journal on Computing*, vol. 19, no. 3, pp. 328–340.

Zhu, X. and W. E. Wilhelm (2006) "Scheduling and Lot Sizing with Sequence-Dependent Setup: A Literature Review," *IIE Transactions,* vol. 38, no. 11, pp. 987-1007.

Appendix

If we consider that h and v do not depend on the particular color and geometry changeover, then there is no need to track specific color or geometry changes. That is, all that needs to be determined is whether there was a change or not from one position to the next. Therefore, the y variables in our MIP formulation may be redefined as follows:

$y_k = \begin{cases} 1 \text{ if a color change occurs between position } k \text{ and } k+1 \\ 0 \text{ otherwise} \end{cases}$

$z_k = \begin{cases} 1 \text{ if a geometry change occurs between position } k \text{ and } k+PPL \\ 0 \text{ otherwise} \end{cases}$

This results in the following simplified model:

Minimize $\quad h\sum_{k=1}^{n-1} y_k + v\sum_{k=1}^{n-PPL} z_k$

Subject to

$\sum_{(i,j)\in S} x_{ijk} = 1$	$k = 1, \dots, n$
$\sum_{k=1}^{n} x_{ijk} = d_{(i,j)}$	$\forall (i,j) \in S$
$y_k \geq x_{ijk} + \sum_{i':(i,i')\in CC} x_{i'jk+1} - 1$	$\forall i: (i,j) \in S, k = 1, \dots, n-1$
$z_k \geq x_{ijk} + \sum_{j':(j,j')\in GG} x_{ij'k+PPL} - 1$	$\forall j: (i,j) \in S, k = 1, \dots, n-PPL$
$x_{ijk} \in \{0,1\}$	$\forall (i,j) \in S, k = 1, \dots, n$

Although this model is more compact than the one in Section 2, it showed similar limitation as a viable solution approach for practical problems.