A Genetic Algorithm-Based Approach to Flexible Flow-Line Scheduling with Variable Lot Sizes

In Lee, Riyaz Sikora, and Michael J. Shaw, Member, IEEE

Abstract—Genetic Algorithms (GA’s) have been used widely for such combinatorial optimization problems as the Traveling Salesman Problem (TSP), the Quadratic Assignment Problem (QAP), and job shop scheduling. In all of these problems there is usually a well defined representation which GA’s use to solve the problem. In this paper, we present a novel approach for solving two related problems—lot-sizing and sequencing—concurrently using GA’s. The essence of our approach lies in the concept of using a unified representation for the information about both the lot sizes and the sequence and enabling GA’s to evolve the chromosome by replacing primitive genes with good building blocks. In addition, a simulated annealing procedure is incorporated to further improve the performance. We evaluate the performance of applying the above approach to flexible flow-line scheduling with variable lot sizes for an actual manufacturing facility, comparing it to such alternative approaches as pair-wise exchange improvement, tabu search, and simulated annealing procedures. The results show the efficacy of this approach for flexible flow-line scheduling.

I. INTRODUCTION

THE n-job, m-machine flexible flow-line scheduling problem with variable lot-sizes is considered in this paper. An example of such a flexible flow-line is the printed circuit board manufacturing system, where each job consists of a number of units (i.e., boards). The flexible flow-line schedule with variable lot sizes under study is defined as follows. Each job consists of splittable lots. The machines in a flexible flow-line are dedicated to processing at most one unit of each job, and the unit can be processed on at most one machine at any time. The flow-line has a limited buffer space available between the machines. Preemption of individual jobs is not allowed. The jobs must be processed in the same sequence by each of the m machines, given the processing times of the unit of each job on each machine. The objective of the above problem is to find the combination of sequence and lot sizes of jobs which minimizes the makespan.1

In traditional approaches, the lot-sizing and sequencing2 problems of the production line are treated as independent sub-problems. Sequencing is usually done after the lot sizes for all the jobs have been found using a suitable heuristic. However, to solve the lot-sizing aspect of the problem requires the exact setup time of each type of job in each period. The determination of these sequence-dependent setup times, however, requires knowledge that can only be found by first determining the sequence. It is the interactive nature of this situation that makes the problem difficult to solve. Because of the interdependence between the two problems, they can benefit from an approach that solves the two problems concurrently.

This paper describes the use of genetic algorithms (GA’s) as a means of jointly determining lot sizes and sequence of jobs in flexible flow-lines. GA’s have been used increasingly to address scheduling problems [4], [11], [13], [18], [20], [36], [56], [61], [62]. Sikora et al. [52] describe a heuristic procedure for integrating the lot-sizing and sequencing decisions in scheduling a flow-line. The procedure taken there, however, does not consider lot-splitting.

The objectives of our research are three fold: 1) to compare the performance of GA’s with those of other heuristics for the problem of sequencing jobs in flexible flow-lines, where the setup times are sequence-dependent and each job consists of a certain number of units; 2) to design a representation for GA’s that can incorporate both lot-sizing and sequencing decisions; and 3) to investigate the effects of combining genetic algorithms with simulated annealing to improve solution quality and computational efficiency.

The novel feature of the GA presented in this paper is the use of explicit building blocks to evolve the chromosome3 from a primitive and lengthy form to a more compact and developed form. In our approach, the GA starts from a lengthy primitive sequence with small initial lot sizes of different jobs and gradually forms optimal lot sizes and sequences concurrently. Similar ideas are used in the Automatically Defined Functions concept in Genetic Programming community where certain portions of the chromosome are frozen, and treated as a single unit in later processing.4

1 Makespan of a set of jobs is defined as the maximum completion time at which all jobs in that job set complete processing.

2 Lot-sizing refers to the problem of determining the number of units of each job to be made in a single batch, and sequencing refers to the problem of determining the order in which these batches are to be made.

3 By a chromosome we refer to a fixed length string having a finite number of possible values at each position. For job sequencing problems, the number of jobs to be processed corresponds to the length of chromosome, and the job type corresponds to the value at each position.

4 We thank an anonymous reviewer for pointing this out.
While GA’s have been shown to be an effective search procedure, especially when the search is complex and difficult [25], a weakness of GA’s for optimization is that sometimes "premature convergence" may occur, i.e., the search process guided by the GA procedure may stop at a local optimum. To enhance the search ability of a GA, a simulated annealing procedure is incorporated for its better control over convergence through the use of an annealing schedule. Recently, several such hybrid approaches have been studied [3], [19], [26], [41]. The rationale behind the hybridization of GA and SA is to exploit the desirable convergence properties of SA, while maintaining the population approach and recombinative power of GA. Such a hybrid method, referred to as the GASA procedure, is shown to further improve the solution quality and computation times for the flexible flow-line scheduling problem under study.

The rest of the paper is organized as follows: In Section II we briefly review the real world manufacturing facility for which the scheduled system is used. In Section III we review the traditional methods used for lot-sizing and sequencing. In Section IV we discuss basic GA theory and GA related research in scheduling. In Section V we evaluate their performance for our application. In Section VI we present the proposed method for jointly solving the lot-sizing and sequencing problems. Section VII illustrates the working of the method by an example and presents empirical results of jointly solving the lot-sizing and sequencing problems. In Section VIII the results of combining GA’s with simulated annealing are shown. The significance of the results is discussed in Section IX.

II. THE DECISION PROBLEM AND ITS CHARACTERISTICS

The motivation of our study comes from the scheduling needs of a real world printed circuit board (PCB) manufacturing plant. Fig. 1 shows the schematic of the production system being studied. It is a Surface Mount Technology (SMT)-based flexible flow-line where the placement of the electronic components on printed circuit boards takes place. It consists of a screen printer for masking and attaching solder on the bare PCB’s, three types of automated chip placement machines, and a solder reflow. Since the machines are connected by conveyors which can hold at most twenty boards at a time, this SMT line has a limited buffer space available between the machines.

Moreover, the set of component tapes fed to chip placement machines has to be changed whenever a different kind of PCB has to be processed by the machine. Normally, the different kinds of boards require components some of which are also needed by the other boards. Therefore, the time required to set up a machine for processing a specific board usually depends on the kind of board processed previously on the machine. In other words, the setup times are sequence dependent.

The problem of scheduling jobs in such a flow-line involves determining the optimal lot sizes into which each job (i.e., a lot of boards) should be split and determining the subsequent sequence that minimizes the makespan.

III. LOT-SIZING AND SEQUENCING METHODS: A BRIEF REVIEW

Typically the problem of scheduling jobs on a flow-line is decomposed into the sub-problems of lot-sizing and sequencing. Bahl et al. [5] give a comprehensive review and categorization of the various lot-sizing methods available in the literature. Some of the most widely used heuristics for lot-sizing are: Silver-Meal Heuristic [53], Least Unit Cost Heuristic (LUCH) [8], and Part Period Balance Heuristic (PPBH) [16].

Karmarkar et al. [33] demonstrated the impact of lot sizes on flow times and waiting times in a deterministic manufacturing cell using a queuing network model that is embedded in an optimization routine that searches for optimal lot sizes. Karmarkar et al.’s study indicates a U-shaped relationship between lot size and flow time. The long flow times resulting from both large and small lot sizes are explained using queuing theory. Large lot sizes tie up machines and cause buildups of queues. Reducing lot sizes at first alleviates this phenomenon, but eventually causes the total workload at machines to increase due to an excessive number of setups. The model provided intuitive insight into job shop behavior depending on the lot-sizing policies. They argued that the real benefits of recognizing this phenomenon should lead to algorithms that compute the lot sizes and determine the sequence of jobs at machines.

The general problem of sequencing jobs in a flow-shop has been shown to be NP-complete [6], [27]. Although much research has been done over several decades, only a few practical algorithms exist for large problems. Exact methods such as complete enumeration, integer programming, and branch and bound can handle only very small problems. For a problem where the numbers of jobs and machines are large, there is usually a tradeoff between solution quality and computational effort. The problem of sequencing in our application is more complicated because of the sequence-dependent setup times and the presence of limited buffer
space between the machines. Because of the combinatorial complexity of such problems, a more viable method for solving such problems is the use of heuristics [45].

Currently available heuristics for sequencing jobs in a flow-shop may be classified as either constructive heuristics or improvement heuristics. A constructive heuristic builds a sequence of jobs so that once a decision is taken it cannot be reversed. On the other hand, an improvement heuristic starts with any sequence of jobs and then attempts to improve the solution by modifying the sequence. Several constructive heuristics for the sequencing problem have been proposed in the literature [9], [12], [28], [32], [49].

Palmer’s heuristic [49] and Gupta’s heuristic [28] use the concept of slope index for each job. The slope index is a measure of whether a job proceeds from shorter to longer processing times in the sequence. The sequence is then constructed in order of descending slope indices with the idea that jobs which tend to proceed from shorter to longer processing times in the sequence of operations are processed earlier.

Campbell, Dudek, and Smith [9] proposed a heuristic that is a generalization of Johnson’s two-machine algorithm [32]; it generates a set of \( m - 1 \) artificial two-machine problems from an original \( m \)-machine problem, each of which is then solved using Johnson’s two-machine algorithm. The best of \( m - 1 \) solutions becomes the heuristic solution to the \( m \)-machine problem.

There are generally two kinds of improvement heuristics used for the sequencing problem: local search heuristics, which start with an initial solution and incrementally try to improve it, and guided search heuristics, which use some kind of guided search to sample promising portions of the search space. Improvement heuristics can be applied to the sequence obtained from constructive heuristics. Two examples of local search heuristics are hill climbing and steepest descent methods. The hill climbing method chooses a nearby point randomly and moves there if the new state is better than or equal to the current state. (In the context of minimization, the hill is inverted so that the move is downward.) The steepest descent method considers each state in the neighborhood of the current state and selects the best one as the next state. Some of the guided search heuristics that have been used successfully are Simulated Annealing (SA), Tabu Search (TS), and Genetic Algorithms (GA’s).

The most common local search heuristic for sequencing or TSP problems is a pair-wise exchange improvement procedure [12], [38] which tries every pair-wise exchange of positions on the seed solution and chooses the one that gives the best improvement. The pair-wise exchange improvement procedure applied in this study is discussed in Appendix A in detail.

Simulated annealing is a stochastic optimization technique derived from statistical mechanics for finding globally optimal solutions to combinatorial problems. It was originally developed as a simulation model for a physical annealing process of condensed matter [44]. Kirkpatrick et al. [35] were the first to point out the relevance of the simulation techniques from statistical physics to combinatorial optimization problems. A comprehensive discussion of the theory and a review of various applications is given by Van Laarhoven and Aarts [59].

Recently, several studies have been done on the application of simulated annealing to scheduling [43], [46], [47].

In simulated annealing, the rate at which the system is cooled is called an annealing schedule. The annealing schedule involves the determination of an initial temperature, \( T_0 \), a reducing factor, \( r_f \), at which cooling takes place, and a control parameter \( \beta_k \) which is recursively calculated as \( \beta_{k+1} = r_f \beta_k \). If \( C_j \) is the value of a current solution \( j \) and \( C_j \) represents the value of a new solution \( j \) from a neighborhood of \( i \), then the new solution \( j \) is accepted with the probability

\[
\exp \left( -\frac{(C_j - C_i)}{\beta_k} \right)
\]

where \( k \) is the stage index, and \( \beta_k \) is a control (temperature) parameter which decreases until it is close to zero in the final stages. If \( C_j \leq C_i \), the new solution \( j \) is always accepted, whereas if \( C_j > C_i \), the new solution is accepted with probability \( \exp \left( -\frac{(C_j - C_i)}{\beta_k} \right) \). If the temperature is reduced rapidly, the system may freeze at a local minimum. On the other hand, if a slower schedule is used, a global minimum is more likely to be found. The optimal annealing schedule usually must be empirically discovered to achieve a near optimal solution efficiently.

Tabu search is a metaheuristic that guides local search heuristics to explore the solution space beyond local optimality [23]. For complete and formal discussions of tabu search, readers may refer to the papers of Glover [21], [22]. Glover [21] claims that tabu search has its origins in combinatorial procedures applied to nonlinear covering problems in the late 1970’s. Tabu search has been successfully applied to many combinatorial optimization problems including traveling salesman problem, graph coloring, and job shop scheduling [15], [42], [58]. Empirical results indicate that tabu search yields high quality solutions to the above problems with modest computational effort, generally dominating alternative methods compared.

Tabu search helps local search heuristics to continue exploration away from local optima by the mechanism of constraining moves which lead backward and guiding the search toward unexplored regions. Two important elements of tabu search is a tabu list, referred to as the short-term memory, and an aspiration level function. The tabu list, which contains forbidden moves, is appropriately updated by including a new element and removing the oldest one from the tabu list. The size of a tabu list can be fixed or variable depending on the application or the stage of search. An aspiration level is associated with each move. If a current tabu move satisfies an aspiration level (usually the best solution found so far), the tabu status of the move is overridden and the move is allowed. The solution quality and computation times are usually enhanced by incorporating a longer-term memory, whose goal is to diversify the search.

One of the drawbacks of the above-mentioned heuristic approaches for scheduling is that they tend to use different objectives for lot-sizing and sequencing problems. For instance, the objective of lot-sizing is to reduce the total cost, and the objective of sequencing is usually to reduce the makespan. However, the lot-sizing decision has important implications for the makespan of the schedule. By not including the makespan as the objective of lot-sizing, the schedules generated by the above methods may be more likely to be sub-optimal.
IV. GENETIC ALGORITHMS AND RELATED RESEARCH

Genetic algorithms represent a class of algorithms based on a simplified computational model of the biological evolution process [31]. They represent a class of general purpose adaptive search techniques that have the properties of parallel search and an enhanced ability to avoid local optima. For a comprehensive reading on genetic algorithms, readers may refer to Goldberg [25]. Goldberg describes GA’s as a search algorithm based on the mechanics of natural selection and natural genetics. GA’s combine the survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm. In every generation a new set of artificial creatures (chromosomes) is created using bits and pieces of the fittest of the old; an occasional new part is tried for good measure. While randomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to speculate on new search points with improved performance. GA’s should be equipped with the following four components for achieving the desired effect:

1) A chromosomal representation of the solution to the problem;
2) A mechanism to create an initial population of solutions;
3) An evaluation function that rates the solutions in terms of their fitness; and
4) Genetic operators that alter the composition of solutions during a reproduction process.

In addition, in applying GA’s, one needs to decide the various values for the parameters that genetic algorithms use, such as a population size, the probability of crossovers, the probability of mutations, and stopping criteria. Genetic algorithms require a fitness function that assigns a numerical fitness value to each solution. The genetic operators most often used are: a) selection, b) crossover, and c) mutation operators.

Binary tournament selection is used in this paper where two solutions are picked at random and the one with a higher fitness value is selected. This process is repeated to select a second string to mate with the previously selected one. As a result, high-performance strings make more replications for mating than low-performance strings.

After the selection process the crossover operator combines the genetic material of two parents by choosing a position randomly in the string and swapping the portion to the right of that position to produce two offspring. The combined emphasis of selection and the structured, though randomized, information exchange of crossover is the key to the power of GA’s as it helps in combining information from different solutions to discover more useful solutions. Usually crossover is applied with a constant probability $P_c$.

The mutation operator randomly changes some of the genetic material in a selected string at a much lower rate ($P_{m}$) than the crossover operator (i.e., $P_{m} \ll P_c$). Mutation is needed to keep the algorithms from getting stuck at a sub-optimal solution by protecting against unrecoverable loss of some potentially important genetic material. In a binary code, mutating a bit corresponds to substituting its value by its complement. In a sequencing problem, two randomly selected positions are interchanged to preserve the feasibility of the solution. The mutation operator plays a secondary role in the search for the solution, but provides the search mechanism for “off-road” solutions.

Special crossover operators for permutation problems (sequencing, TSP, QAP, etc.) have been studied in the literature. Among them, partially matched crossover (PMX) [24], cycle crossover (CX) [48], order crossover (OX) [13], and edge recombination [61] are the most widely used. Appendix B describes the edge recombination operator designed and implemented to solve the lot-sizing and sequencing problems simultaneously.

Recently, several hybridization methods combining both GA’s and simulated annealing have been proposed [3], [19], [40], [41]. Their empirical results show that hybridization methods improve pure GA’s in solution quality, speed of convergence, and consistency on a variety of problems. The major rationale behind the use of hybridization methods is that by adding annealing schedule to GA’s, GA’s may increase the probability of escaping local optima in search of global optima with less computation time. Such hybridization methods have been shown to facilitate the search of solution space by exploiting the desirable convergence properties of SA, while maintaining the population approach and recombinative power of GA.

V. PERFORMANCE EVALUATION OF VARIOUS HEURISTICS

In order to decide the suitability of either local search heuristics or guided search heuristics for our application we compared the performance of various heuristics on some sample problems. In the first set of experiments, we compared four designs of genetic algorithms with a pair-wise exchange improvement procedure. Having identified the best heuristic, in the second set of experiments, we compared it with a simulated annealing procedure, a tabu search, and a repeated pair-wise exchange improvement procedure.

These experiments used a computer simulation to model the SMT flow line (Fig. 1) which produces at most twenty different job types, each job requiring a lot size of 20 to 100, and the same sequence of operations. All jobs are assumed to be available for release into the system at the time of the sequencing decision. The flow-line consists of five different machines, each processing a single operation. The machines are connected by conveyor lines which have a buffer space fixed at twenty between every pair of machines. The sequence-dependent setup times of the job types were generated randomly from the range [5, 25]. The lot sizes for each job type were generated randomly from the range [20, 100]. The processing times for each unit product within a job were generated randomly from the range [0.01, 1.0].

We evaluated the solution quality using an average relative percentage deviation (ARPD) and a maximum percentage deviation (MPD) from the optimal solution or the best known solution returned in all the runs made. ARPD and MPD are

\[ ARPD = \left( \frac{1}{n} \sum_{i=1}^{n} \frac{|x_i - \bar{x}|}{\bar{x}} \right) \times 100 \]

\[ MPD = \max \left( \frac{|x_i - \bar{x}|}{\bar{x}} \right) \times 100 \]

These values are based on real data obtained from the manufacturing facility.
TABLE I
SOLUTION QUALITY AND COMPUTATIONAL TIME FOR TEST PROBLEMS BY GA-BASED METHODS AND PAIR-WISE EXCHANGE IMPROVEMENT PROCEDURE

<table>
<thead>
<tr>
<th>Jobs</th>
<th>Edge (^a)</th>
<th>PMX (^b)</th>
<th>CX (^c)</th>
<th>OX (^d)</th>
<th>PE-Palmer (^e)</th>
<th>PE-CDS (^f)</th>
<th>PE-random (^g)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ARPD</td>
<td>MPD</td>
<td>ARPD</td>
<td>MPD</td>
<td>ARPD</td>
<td>MPD</td>
<td>ARPD</td>
</tr>
<tr>
<td>8</td>
<td>0.19 (12.9)*</td>
<td>0.75</td>
<td>0.27</td>
<td>2.08</td>
<td>1.20 (13.4)</td>
<td>3.02 (46.8)</td>
<td>0.07</td>
</tr>
<tr>
<td>10</td>
<td>0.48 (15.2)</td>
<td>1.17</td>
<td>0.91</td>
<td>3.06</td>
<td>1.80 (13.1)</td>
<td>4.52 (59.4)</td>
<td>0.92</td>
</tr>
<tr>
<td>13</td>
<td>0.70 (29.5)</td>
<td>2.75</td>
<td>0.95</td>
<td>2.45</td>
<td>3.44 (22.7)</td>
<td>5.85 (88.3)</td>
<td>2.36</td>
</tr>
<tr>
<td>16</td>
<td>0.75 (34.5)</td>
<td>6.04</td>
<td>2.46</td>
<td>8.14</td>
<td>3.22 (25.1)</td>
<td>9.75 (53.2)</td>
<td>5.15</td>
</tr>
<tr>
<td>18</td>
<td>0.04 (52.9)</td>
<td>0.28</td>
<td>2.12</td>
<td>3.75</td>
<td>3.33 (45.6)</td>
<td>4.99 (79.3)</td>
<td>4.61</td>
</tr>
<tr>
<td>20</td>
<td>0.19 (58.5)</td>
<td>1.36</td>
<td>2.81</td>
<td>4.21</td>
<td>3.38 (59.8)</td>
<td>5.90 (84.1)</td>
<td>5.01</td>
</tr>
</tbody>
</table>

a: genetic algorithms with edge recombination operator.
b: genetic algorithms with partially matched crossover operator.
c: genetic algorithms with cycle crossover operator.
d: genetic algorithms with order crossover operator.
e: pair-wise exchange improvement procedure using sequence by Palmer heuristic as a seed solution.
f: pair-wise exchange improvement procedure using sequence by Campbell, Dudek and Smith heuristic as a seed solution.
g: pair-wise exchange improvement procedure using sequence by random generation as a seed solution.

* : computational time in seconds.
ARPD: average relative percentage deviation from the best known solution.
MPD: maximum percentage deviation from the best known solution.

calculated by

\[
\text{ARPD} = \left( \frac{1}{n} \sum_{i=1}^{n} \left( \text{Heuristic}_i - \text{Optimal}_i \right) / \text{Optimal}_i \right) / n \times 100,
\]

MPD = max(\(\text{Heuristic}_i - \text{Optimal}_i \times 100\)),

for \(i = 1, 2, \ldots, n\),

where \(\text{Heuristic}_i\) and \(\text{Optimal}_i\) are the heuristic, and optimal or best known solution for problem \(i\), respectively. Both of them measure how close the solutions are to the optimal solutions. ARPD measures the average deviations. MPD measures the maximum deviations.

In the first set of experiments, we considered the following four crossover operators mentioned earlier for a comparative study: 1) Partially Matched Crossover (PMX), 2) Order Crossover (OX), 3) Cycle Crossover (CX), and 4) Edge Recombination. To determine appropriate parameter values for GA’s, sample problems of different job sizes have been generated and tested. In each problem set GA’s operated with a mutation rate of 0.1, a crossover rate of 1.0, a population size set at 5 times the problem size, and a binary tournament selection for the reproduction process. We used an elitist strategy, where the best individual found at each generation is kept unaltered into the next generation. The runs were terminated either after no solution improvement was observed for a fixed number of generations, or after the GA had processed 100 generations.

In addition, we evaluated the performance of the pair-wise exchange improvement procedure. For the pair-wise exchange improvement procedure, the neighborhood of the seed solution is searched sequentially. Each seed solution has \((n^2 - n)/2\) neighbors. Since the final sequence obtained by the pair-wise exchange improvement procedure may be dependent on the initial sequence, we considered three different constructive methods to generate the initial sequence: 1) Palmer heuristic [49], 2) CDS heuristic [9], and 3) random heuristic. Since the Palmer and CDS heuristics are for the problems involving sequence-independent setup times, the sequence-dependent setup times were all averaged to get the required setup times. In the case of the random heuristic the initial sequence was generated randomly. The pair-wise exchange improvement procedure was run until a solution reached a local optimum.

We performed 10 runs for job size 8, 10, 13, 16, 18, and 20 respectively. The results of 10 runs for each job size were averaged. Results were evaluated using the ARPD and MPD,
and computation times. For job size 8, optimal solutions were used for the calculation of the ARPD and MPD. For larger job sizes, the best solutions were used for the calculation of the ARPD and MPD. The ARPD, MPD, and computation times produced by each of these heuristics are shown in Table I.

In summary, the solution quality resulting from the edge recombination operator is the best among all crossover operators in a variety of job sizes (the difference proved to be statistically significant at a significance level of 0.05 using a t-test except for job size 8). The solution quality of the edge recombination operator is also better than those using the pair-wise exchange improvement procedure at a significance level of 0.05 using a t-test, although it requires more computation time. However, it was observed the initial solutions generated by three constructive heuristics do not affect the final solutions of the pair-wise exchange improvement procedure (the difference proved to be statistically insignificant at a significance level of 0.05 using a t-test).

Fig. 2 shows the average solution quality and efficiency for job size 20 problems associated with various GA crossover operators. The solution quality and efficiency of the edge recombination operator are consistently better than those of the other three GA crossover operators across all generations for job size 20 problems. For other test problems a similar trend has also been observed.

When applying the GA procedure to sequencing problems, it is usually advantageous not to break too many edges in the search process as measured by the disruption rate [61]. The disruption rate is defined as the number of disrupted edges in a child string divided by the total number of edges. For example, suppose that the number of edges in parent string (1 2 3 4 5) is 4. If the sequence of the child string is (1 2 3 5 4), then the disruption rate is 0.5, since edges (3 4) and (4 5) were broken. Each broken edge means that a new, random edge is created. Since the differences in performance resulting from applying various crossover operators appeared to be related to the ability of the operators to preserve edges, we measured the disruption rate of each operator using 30 different test sets, each of which consists of 60 samples randomly chosen. As shown in Table II, the average disruption rate resulting from applying the edge recombination operator is the lowest among them (at a significance level of 0.05 using a t-test).

Having found that the GA using the edge recombination operator performs the best among four GA crossover operators and the pair-wise exchange improvement procedure, we next compare the GA using the edge recombination operator with 1) a simulated annealing procedure, 2) a tabu search procedure, and 3) a repeated pair-wise exchange improvement procedure.

The modified simulated annealing procedure tested in these experiments is discussed in Appendix C in detail. For the simulated annealing procedure, the choice of reasonable annealing parameters was derived empirically through extensive experimentation and observing the effect on the solution quality, the speed at which the process converges, and the quality, the speed at which the process converges, and the solution quality. We set an initial acceptance probability at 0.95 which is decreased geometrically by a factor of 0.88 through the total stages of 40 to 50 depending on the problem sizes. The number of iterations at each stage is linked to the cardinality of the perturbation scheme. On average, it was set at 2 times the cardinality, \( (n^2 - n)/2 \).

The tabu search procedure implemented in this study is discussed in Appendix D. As in the simulated annealing procedure, appropriate parameter values for the tabu list size, aspiration level, and penalty function for longer-term memory were determined through extensive experimentation. The number of stages is set at twice that of the simulated annealing procedure. The neighborhood size at each stage is \( (n^2 - n)/2 \). Pair-wise exchange is used to search the neighborhood of the current solution at each stage.

For the repeated pair-wise exchange improvement procedure, we tried as many initial seed sequences as possible with comparable computation times required by the GA. The best of the locally optimal solutions obtained from the repeated pair-wise exchange improvement procedure was chosen as a final solution to compare with other heuristics. The initial sequences were generated randomly. Each seed solution has \( (n^2 - n)/2 \) neighbors. The repeated pair-wise exchange improvement procedure sequentially searched a neighborhood of a seed solution until a solution reached a local optimum.

We tested the same problem used in the first set of experiments. The results of 10 runs for each job size were averaged. Table III shows the average relative percentage deviation (ARPD), the maximum percentage deviation (MPD), and the computation times produced by each of the heuristics. The solution quality resulting from the edge recombination operator is on average the best among all heuristics in a
variety of job sizes. Figs. 3 and 4 show that between GA and other heuristics, the differences in both ARPD and MPD are generally widening as the job sizes become larger.

The solution quality of the GA procedure with edge recombination is better than that of the repeated pair-wise exchange improvement procedure. The solution quality of the GA procedure with edge recombination operator is also better, with significantly less computation time, than that of the simulated annealing procedure. On average, the solution quality of the GA procedure with edge recombination operator is better than that of the tabu search (TS) procedure. All the above results are statistically significant at 0.05 level of significance using a t-test. Fig. 5 shows that the GA procedure with edge recombination operator dominates the TS and SA procedures over the generations (the iterations of the TS and SA procedures were converted to the number of equivalent generations). Compared to the GA and TS procedures, the SA procedure improves the solution very slowly over the generations. The rapid improvement of solution realized by the TS procedure is due to the pair-wise exchange improvement procedure embedded in the TS procedure. However, the solution quality of the TS procedure is more sensitive to the parameter values than the GA and SA procedures. Our results suggest that GA’s can most efficiently search for global optima without getting trapped in local optima.

In summary, these results indicate that the GA procedure with edge recombination operator is better than the other heuristics considered in this paper with respect to both the solution quality and computation times. The GA procedure with
edge recombination operator also outperforms other heuris-
tics in its ability to maintain the lowest level of solution fluctuations (measured by MPD). It performs better than the pair-wise exchange improvement procedure in solution quality even with comparable computation times. One of the reasons that the edge recombination operator performs best among four crossover operators and other improvement heuristics is that it applies domain knowledge (i.e., the edge map, which represents the allowable sequences of each pair of jobs based on the particular structure of the sequencing problem). The results are very encouraging for the application to real world flow-line scheduling problems, since the timely decision on sequencing and the high quality solution are critical in responding to dynamically changing manufacturing environments. Based on these experimental results, the edge recombination operator is selected for solving lot-sizing and sequencing problems simultaneously.

VI. A METHOD FOR JOINTLY SOLVING THE LOT-SIZING AND SEQUENCING PROBLEMS

In this section we describe a modified genetic algorithm to simultaneously search for optimal lot sizes and sequence. The motivation behind such an approach comes from the intuition that sequencing alone makes it extremely difficult to locate optimal makespan and by taking into account the lot-sizing decision we can improve the makespan by better balancing machine idle times, blocking times, and setup times.

For the job sequencing problem, the basic building blocks are edges (i.e., job orderings) which correspond to the links in the individual string of the GA procedure. By using a primitive representation in which the total demand for each job is randomly split into smaller lots we found that the GA procedure gradually forms clusters of the same job type. Treating these clusters as building blocks, the GA procedure replaces the initial lots with the new aggregate lots.

To solve the lot-sizing and the sequencing simultaneously, we devised a special operator which coalesces the clustered lots of the same type. The coalescing operator can be applied whenever the premature convergence problem arises as the GA procedure approaches a local optimum. The following gives the procedure for the above joint lot-sizing and sequencing approach (JLS) (see bottom of next page).

For illustrative purposes, assume that there are four types of jobs with the following initial lot sizes for each job type.

<table>
<thead>
<tr>
<th>Job type</th>
<th>Initial lot sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>20</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>40</td>
</tr>
<tr>
<td>D</td>
<td>30</td>
</tr>
</tbody>
</table>

In other words, there are twenty items of job type A to be made, ten items of job type B to be made, etc. The machines have to be set up when the type of job changes and the time for setting up the machine depends on the type of job that was processed last. The problem of scheduling the above jobs involves determining the optimal lot sizes into which each job should be split and determining the subsequent sequence that minimizes the makespan. For example, if we decide to start with an initial lot size of 10 for each job then we get the following representation for the primitive string:

<table>
<thead>
<tr>
<th>Alleles</th>
<th>Job type</th>
<th>Lot size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>C</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>C</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>D</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>D</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
<td>10</td>
</tr>
</tbody>
</table>

A schedule then would be the sequence in which the above lots have to be processed. An example of a solution might be

Genotype\(^6\): (3 4 5 1 9 10 8 7 2 6)

Phenotype\(^7\): ((B 10)(C 10)(C 10)(A 10)

(10)(D 10)(D 10)(C 10)(A 10)(C 10))

\(^6\)The genotype is a solution structure used by the GA.

\(^7\)The phenotype maps the genotype to the actual solution structure in the problem domain.
Each pair in the phenotype indicates the job type and corresponding lot sizes. The main idea behind our approach is to first use the above primitive string representation for a certain number of generations to enable the JLS procedure to find good clustering of genes. In our example, same job types clustering together indicate that they should form good building blocks. Whenever a convergent problem arises, the best individual of the population is chosen and clusters of the same job type are coalesced into a single lot. In other words, good building blocks replace the primitive genes in the original string. In the above example of the phenotype, the genes that are underlined can be coalesced into single genes giving the following string representation (see bottom of next page).

Note that, as a result of coalescing, the string length has been reduced from 10 to 7. Using the new representation the JLS procedure is run again by retaining the above best solution in the initial population and generating the rest of the individuals randomly. Even though the coalescing operator changes the previous structure of the string, the JLS procedure preserves prior knowledge (i.e., the best sequence and lot sizes so far) and lets the new population evolve incrementally.

VII. APPLYING JLS TO SCHEDULING

A more detailed example would be helpful to show how the solution structure evolves in jointly determining the lot sizes and the sequence. In this example, a sample problem with five machines (Fig. 1) and eight job types was considered. Initial lot sizes for each job type were generated randomly from the range [20, 100]. Processing times of the job types on the different machines were generated randomly from [0.01, 1.0], sequence-dependent setup times of the job types were generated randomly from [0.5, 2.5], and buffer sizes between machines are set to 20. The primitive string representation was obtained by splitting the jobs into initial lot sizes of 10. The edge recombination operator was used as a crossover operator and binary tournament selection was used for a reproduction process. The probability of crossover was fixed at 1.0 and the probability of mutation was kept at 0.1. After every 50 generations the best individual in the population was coalesced, as explained in the previous section, to get a new string representation. Fig. 6 summarizes the evolutionary process of the string and the resulting improvement in the solution.

The primitive string (i.e., the number of job lots) at the beginning of the JLS procedure has a size of 57 and the best individual has a makespan of 431.47 time units. After 50 generations the makespan is reduced to 413.58. As shown in Fig. 6, the sequence has clusters of the same job types. Treating these as building blocks, coalescing is done and the string length is reduced to 42. Note that the JLS procedure start with a new solution structure as a result of coalescing the old solution structure. Except for the best individual, the rest of the initial population is replaced by random members. The reason for the use of a random replacement scheme is that the JLS procedure needs to maintain as much diversity as

The JLS procedure:
1. Initialize input parameters—lot size, setup time, and processing time of each job, and job types;
2. Split each original lot size into initial lot sizes;
3. Assign a new type identification to each newly created job to get the chromosome representation;
4. Start the GA procedure with a randomly created population of size \( n \);
   \[
   \text{epoch} = 0;
   \]
   repeat;
   \[
   \text{compute the fitness of each member in the old population, old_pop; /*evaluation*/}
   \]
   repeat;
   \[
   \text{stochastically select 2 parents with probability proportional to their fitness; /*reproduction*/}
   \]
   perform edge recombination on 2 parents with probability \( P_c \)
   to generate 2 offspring; /*crossover*/
   mutate a random bit with probability \( P_m; /*mutation*/
   place the 2 offspring in the new population, new_pop;
   until the new population is filled with \( n \) individuals;
   \[
   \text{gen} = \text{gen} + 1;
   \]
   \[
   \text{old_pop} = \text{new_pop};
   \]
   \[
   \text{epoch} = \text{epoch} + 1;
   \]
   until \( \text{epoch} \geq \text{max_epoch}; /*termination*/
5.1. If same types of jobs cluster and \( \text{gen} < \text{max_gen} \), then coalesce the clusters of the same job type of the best individual to get a new chromosome representation and go to step 4;
5.2. If there are no clusters of the same job type and \( \text{gen} \leq \text{max_gen} \), then output the best sequence, lot sizes, and makespan. Stop.
possible in the initial population of the new solution structure. In our preliminary test, the above replacement scheme worked very well compared with other population mixing schemes. This process of coalescing and generating a new population after every 50 generations is repeated until the evolution of the string converges. As can be seen, after 300 generations, the string length is reduced to 17 and the makespan is reduced to 379.16.

We also ran the JLS procedure without coalescing with the primitive string representation for 300 generations to illustrate the benefit of evolving the string. The JLS procedure without coalescing converged to a local optimum after only 50 generations, even though the entire population (except the best individual) was replaced by random members after every 50 generations. The final sequence obtained had a makespan of 409.82 time units.

To underscore the importance of lot-sizing in the scheduling process, we ran the GA procedure without breaking the jobs into smaller lots. In other words, the string length in this case was 8 because there were eight job types. The best sequence obtained after 300 generations had a makespan of 408.62 time units. Thus, the approach where the string is also allowed to evolve gives the best results.

The reason why the lot splitting gives a shorter makespan despite the increases in the number of setups can best be understood by analyzing machine setup times, blocking times, and idle times for the two schemes: with lot splitting and without lot splitting. Figs. 7–9 illustrate the results graphically. While the total setup times have been increased from 59 to 120 time units, the total machine blocking times have been reduced from 262 to 15 and the total machine idle times from 184 to 14 by implementing the JLS procedure (i.e., GA’s with lot

---

<table>
<thead>
<tr>
<th>Generation</th>
<th>Chromosome for Best Sequence</th>
<th>Makespan</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32320161667248361477831881268933815512778983854382265887</td>
<td>431.472</td>
</tr>
<tr>
<td>50</td>
<td>[diagram]</td>
<td>413.578</td>
</tr>
<tr>
<td>100</td>
<td>[diagram]</td>
<td>402.339</td>
</tr>
<tr>
<td>150</td>
<td>[diagram]</td>
<td>393.140</td>
</tr>
<tr>
<td>200</td>
<td>[diagram]</td>
<td>388.817</td>
</tr>
<tr>
<td>250</td>
<td>[diagram]</td>
<td>382.341</td>
</tr>
<tr>
<td>300</td>
<td>[diagram]</td>
<td>379.185</td>
</tr>
</tbody>
</table>

Fig. 6. Evolution of the chromosome.
splitting). This example shows that the reduced blocking and idle times compensate for the increased setup times. Another important observation is that the difference between maximum blocking/idle time and minimum blocking/idle time on the machines is greatly reduced, indicating a more balanced flow line.

In applying the JLS procedure, there is an issue of determining the initial lot size used in the starting solution. For instance, in the above example, the initial lot size was chosen as 10. The logical questions to ask then are how to determine the initial lot size and whether the performance is sensitive to this parameter in terms of solution quality and computational efficiency. A good statistic to determine the best initial lot size is the ratio of total setup times to average unit processing times—which we call an average capacity consumption by a setup. Initial lot sizes are determined with the ratio of the initial lot size to the average capacity consumption by a setup (i.e., the LC ratio) as 1/3, 2/3, 4/3, and 8/3, respectively (i.e., initial lot size 5, 10, 20, and 40 respectively). We performed five runs for job sizes 5, 7, 8, and 9 respectively.

Fig. 7. Illustration of blocking time change by lot splitting.

Fig. 10 shows the impact of initial lot sizes on solution quality used by the JLS procedure. Intuitively, smaller initial lot sizes generally result in better solutions at the cost of higher computation time; however, as highlighted in Figs. 10 and 11, there was no significant difference in solution quality

<table>
<thead>
<tr>
<th>Blocking time on each machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
</tr>
<tr>
<td>160</td>
</tr>
</tbody>
</table>

- w/o splitting
- w/ splitting

The GASA procedure:
1. Initialize input parameters—lot size, setup time, and processing time of each job, and job types;
2. Split each original lot size into initial lot sizes;
3. Assign a new type identification to each newly created job to get the chromosome representation;
4. Start the GA procedure with a randomly created population of size $n$;
   - epoch = 0;
   - repeat;
     - compute the fitness of each member
       - in the old population, old_pop; /*evaluation*/
     - repeat;
       - stochastically select 2 parents with probability proportional to their fitness; /*reproduction*/
       - perform edge recombination on 2 parents with probability $P_c$
         - to generate 2 offspring; /*crossover*/
       - mutate a random bit with probability $P_m$; /*mutation*/
       - if fitness of new_solution $\geq$ fitness of current_solution
         - then accept new_solution
       - else if random $(0, 1) \leq \text{AP}(1)(rt)^{\text{gen}-1}$
         - then accept new_solution
       - else accept current_solution;
     - place the 2 offspring in the new population, new_pop;
     - until the new population is filled with $n$ individuals;
   - if fitness of best member in new_pop $>$ fitness of best member in old_pop
   - perform one stage of the pair-wise interchange improvement procedure;
   - gen = gen + 1;
   - old_pop = new_pop;
   - epoch = epoch + 1;
   - until epoch $\geq$ max_epoch; /*termination*/
5.1. If same types of jobs cluster and gen $<$ max_gen, then coalesce the clusters of the same job type of the best individual to get a new chromosome representation and go to step 4;
5.2. If there are no clusters of the same job type and gen $\geq$ max_gen, then apply simulated annealing procedure to the best solution found so far;
5.3. Output the best sequence, lot sizes, and makespan. Stop.
such a hybrid approach, referred to as the GASA procedure, is implemented on two levels: a main-level genetic algorithm, used to search globally, and a second-level modified simulated annealing procedure, used to improve upon the solution of the genetic algorithms. The following gives the procedure for the GASA approach (see bottom of previous page).

The simulated annealing procedure is embedded in 1) the mutation operator of the JLS procedure and 2) the pair-wise exchange improvement procedure. The simulated annealing procedure embedded in the mutation operator and the pair-wise exchange improvement procedure allow the GASA procedure to escape the local minima at early generations by adjusting an annealing schedule and to efficiently hill-climb as the temperature approaches zero. In the GASA procedure, the mutation module gets the child string as an input and performs the simulated annealing procedure. The mutation operator randomly picks two positions of the input string and exchanges the value of both positions. If an improvement is observed, then the mutated string is returned as an output to the new generation. Alternatively, when the fitness decreases, the mutated string is accepted with probability $P(1)(t^{k-1})$. In case the GASA procedure terminates prematurely due to a maladjusted annealing schedule, the simulated annealing procedure was applied to the best solution found so far.

To evaluate the GASA procedure, we first need to compare the solutions generated to the corresponding optimal solutions. However, the flow-shop scheduling problem with lot-sizing and sequencing is too complicated to identify the optimal solutions. To that end, we first test the GASA procedure on the $n$-jobs, two-machine sequencing problems, where Johnson's algorithm [32] is known to generate optimal solutions.

To transform our previous problems to Johnson's job, 2-machine problems, the condition of a limited buffer space has been relaxed to allow an infinite buffer space between machines. In addition, the setup times are now independent of the job sequence. The processing time for each job is calculated as follows

$$P_{mn} = s_{mn} + p_{mn} \times l_{n},$$

where $s_{mn}$ represents a setup time for $n$th job on the $m$th machine, $p_{mn}$ a unit processing time for $n$th job on the $m$th machine, and $l_{n}$ lot size for $n$th job.

The optimal lot sizes and sequences are obtained by applying Johnson's algorithm, which serves as the benchmark for evaluating the performance of the GASA procedure. We tested 10 different job size problems. Based on extensive preliminary tests, the mutation rate was set at 0.1, the initial acceptance probability at 0.95, and the reducing factor at 0.99. At each generation the acceptance probability was reduced geometrically.

Table IV shows the experimental results of the GASA procedure, the JLS procedure, and the pair-wise exchange improvement procedure. For the pair-wise exchange improvement procedure, coalescing was not performed since it leads to worse solutions. While the JLS procedure achieves optimal solutions in seven problems and the pair-wise exchange heuristic only one out of the ten problems, the GASA procedure reaches the optimal solutions in all test problems. The JLS
procedure results in a smaller deviation from the optimal solutions. The computation time increases significantly in the pair-wise exchange improvement procedure compared to the GASA procedure and the JLS procedure especially in larger problems. For the 30-job problem, the GASA procedure takes significantly less computation time (182.8) than the JLS procedure (321.4) and the pair-wise exchange improvement procedure (896.4). While the improved solution quality from applying the GASA procedure over the JLS procedure is anticipated, the reduced computation times of the GASA procedure are a surprise and call for further explanation.

Fig. 12 shows the improvement of solution quality over time for the GASA procedure, the JLS procedure, and the pair-wise exchange improvement procedure for 12-job problem. The GASA procedure shows highest efficiency, maintaining superior solution quality over the generations compared to the JLS procedure and the pair-wise exchange improvement procedure. The JLS procedure eventually reaches the optimal solution, but takes longer. For other test problems, a similar trend has also been observed.

In the second set of experiments we tested the performance of the GASA, JLS, and tabu search (TS) procedures on the problem set given in Section VII. For the tabu search procedure, we used the same parameter values as in Section V. The maximum number of generations for the GASA and JLS procedures was set at 300. The maximum number of iterations for the TS procedure was set at $6 \times 10^4$. Unlike the GASA procedure, the TS procedure did not use a coalescing scheme, since the coalescing scheme frequently leads to worse solutions. The probable explanation is that the TS procedure searches from a single sequence and is more likely to coalesce a bad sequence, while GA-based methods search from a population of sequences with a better capability of identifying a good sequence.

The results are summarized in Table V. Consistently, the advantage of the GASA procedure becomes more pronounced compared to the JLS procedure as the job size becomes larger (at a significance level of 0.05 for job sizes 14 and 16, 18 using a $t$-test). The solution quality of the GASA procedure is also better than the TS procedure in all test problems with less computation times (the difference is statistically significant at the significance level of 0.05 using a $t$-test except for job size 18). It was also observed that on average approximately 4% improvements in makespan were obtained by considering joint lot-sizing and sequencing.

The performance differences in ARPD and MPD are illustrated in Figs. 13 and 14, which show that the JLS procedure performs worse than the TS procedure on larger problems (job sizes 16 and 18), while the GASA procedure performs consistently better than the TS procedure over all problems. This can be explained by the fact that the GASA procedure facilitates over the generation the search toward promising so-
TABLE IV
SOLUTION QUALITY AND COMPUTATIONAL TIME FOR n-JOB, TWO-MACHINE TEST PROBLEMS

<table>
<thead>
<tr>
<th>Jobs</th>
<th>GASA a</th>
<th>JLS b</th>
<th>PE c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ARPD</td>
<td>MPD</td>
<td>ARPD</td>
</tr>
<tr>
<td>7</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>(6.2)*</td>
<td>(8.1)</td>
<td>(2.4)</td>
</tr>
<tr>
<td>8</td>
<td>0.00</td>
<td>0.00</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>(7.1)</td>
<td>(10.4)</td>
<td>(3.7)</td>
</tr>
<tr>
<td>9</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>(12.5)</td>
<td>(9.6)</td>
<td>(4.3)</td>
</tr>
<tr>
<td>10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>(15.8)</td>
<td>(20.2)</td>
<td>(7.1)</td>
</tr>
<tr>
<td>12</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>(18.4)</td>
<td>(22.8)</td>
<td>(15.9)</td>
</tr>
<tr>
<td>15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>(24.3)</td>
<td>(28.6)</td>
<td>(21.8)</td>
</tr>
<tr>
<td>17</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>(21.9)</td>
<td>(32.1)</td>
<td>(43.1)</td>
</tr>
<tr>
<td>20</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>(51.3)</td>
<td>(72.6)</td>
<td>(96.2)</td>
</tr>
<tr>
<td>25</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>(93.5)</td>
<td>(162.0)</td>
<td>(296.4)</td>
</tr>
<tr>
<td>30</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>(182.8)</td>
<td>(321.4)</td>
<td>(896.4)</td>
</tr>
</tbody>
</table>

a: genetic algorithms incorporating simulated annealing with initial lot size 20.
b: genetic algorithms with initial lot size 20 with coalescing.
c: random pair-wise improvement procedure with initial lot size 20.
* : computational time in seconds.

IX. DISCUSSIONS AND CONCLUSION

In summary, these experimental results show that the GASA procedure has the added advantages of incorporating simulated annealing that helps avoid unnecessary searches, resulting in improvements over both the solution quality and computation times. The improvement over the solution quality becomes more pronounced for larger problems.

Fig. 12. Performance of the GASA procedure, the JLS procedure, and the pair-wise exchange improvement procedure over the generations (number of iterations in pair-wise exchange improvement procedure has been converted to the number of generations).

Job Size: 12
TABLE V
SOLUTION QUALITY AND COMPUTATIONAL TIME FOR TEST PROBLEMS

<table>
<thead>
<tr>
<th>Jobs</th>
<th>GA procedure</th>
<th>JLS procedure</th>
<th>TS procedure</th>
<th>Non splitting*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ARPD</td>
<td>MPD</td>
<td>ARPD</td>
<td>MPD</td>
</tr>
<tr>
<td>5</td>
<td>0.02</td>
<td>0.08</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>(14.7)*</td>
<td></td>
<td>(18.5)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.14</td>
<td>0.40</td>
<td>0.02</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td>(34.6)</td>
<td></td>
<td>(44.8)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.09</td>
<td>0.38</td>
<td>0.01</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>(33.9)</td>
<td></td>
<td>(41.2)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0.09</td>
<td>0.25</td>
<td>0.19</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>(125.3)</td>
<td></td>
<td>(137.5)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.04</td>
<td>0.29</td>
<td>0.15</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>(144.6)</td>
<td></td>
<td>(166.3)</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0.02</td>
<td>0.09</td>
<td>0.28</td>
<td>0.41</td>
</tr>
<tr>
<td></td>
<td>(184.8)</td>
<td></td>
<td>(214.4)</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.02</td>
<td>0.11</td>
<td>0.68</td>
<td>1.42</td>
</tr>
<tr>
<td></td>
<td>(312.1)</td>
<td></td>
<td>(294.8)</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>0.14</td>
<td>0.72</td>
<td>0.99</td>
<td>1.67</td>
</tr>
<tr>
<td></td>
<td>(642.7)</td>
<td></td>
<td>(592.9)</td>
<td></td>
</tr>
</tbody>
</table>

a: genetic algorithms with original lot size.
* : computational time in seconds.

Fig. 13. Performance of the GASA procedure, the JLS procedure, and the tabu search procedure with regard to ARPD.

GA procedure to flexible flow-line scheduling with variable lot sizes, the GA procedure with edge recombination operator performs better than the GA procedure with other types of crossover operators, simulated annealing, tabu search, and repeated pair-wise exchange improvement procedures; 2) the solution quality and computation time of the GA-based method with lot splitting for jointly solving the lot-sizing and sequencing problems (i.e., the JLS procedure) are significantly better than the GA-based method without lot splitting; and 3) the combination of the JLS procedure with simulated annealing (i.e., the GASA procedure) can further improve the solution quality with less computation time.

Genetic algorithms represent a class of computational methods equipped with a variety of genetic operators of different characteristics. Our experiments indicate that, for solving the sequencing problem, the GA procedure with edge recombination operator is better than all other heuristics, including the GA procedures with other types of crossover operators. This can be explained by the fact that the edge recombination operator incorporates the domain-specific knowledge which takes into account the allowable sequences of each pair of jobs based on the edge map.

The joint lot-sizing and sequencing procedure demonstrates very promising properties for solving the real world flexible flow-line scheduling problem because of its superior solution quality and efficiency to other competitive improvement heuristics. By taking advantage of the genetic representation used by the GA procedure, where job lots and sequence are
Fig. 14. Performance of the GASA procedure, the JLS procedure, and the tabu search procedure with regard to MPD.

represented by chromosomes, the JLS procedure converges to a schedule that identifies both the best sequence and the lot sizes of job types. In addition to the performance advantages demonstrated by our experiments, a side benefit of this approach is that it clearly shows the benefits of lot splitting even at the cost of additional setups.

While the GA-based JLS procedure has demonstrated good performance for solving the joint lot-sizing and sequencing problem, doing better than the pair-wise exchange improvement procedure used by other researchers, we showed that the performance of the JLS procedure can be further improved by the incorporation of simulated annealing implemented in the GASA procedure. This improvement is especially pronounced when the problem size is larger, due in large part to the greater need to get away from local optimal solutions and to fine-tune the solutions. As important, the GASA procedure also results in faster computation times.

APPENDIX

A. Pair-Wise Exchange Improvement Procedure

The most common local search method for sequencing or TSP problems is a pair-wise exchange improvement procedure which tries every pair-wise exchange of positions on the seed solution and chooses the one that gives the best improvement. We define a neighborhood as a set of solutions that can be generated from a current solution. Suppose that \((1, 2, 3, \ldots, n)\) is a current sequence and that the integers \(i\) and \(j\) are generated where \(i \leq n\), \(j \leq n\), and \(i \neq j\). The pair-wise exchange procedure will generate a new sequence \((1, 2, \ldots, (i-1), j, \ldots, (j-1), i, \ldots, n)\) by interchanging jobs in \(i\) and \(j\) positions. For this neighborhood search, each sequence generates \(n(n-1)/2\) new sequences. The best of the neighbors is selected if it is better than or equal to the current sequence. Otherwise the search is stopped. We adopted an ordered search which examines jobs in position \((i, j)\) in the order \((1, 2), (1, 3), \ldots, (1, n), (2, 3), \ldots, (n-1, n)\).

B. Edge Recombination Operator

In this study we used a modified version of the edge recombination operator which was originally developed by Whitley et al. [61]. The edge recombination operator should construct an offspring by exclusively using edges present in the parent strings to inherit as much information as possible from the parent structures. To store all the connections from two parents, an edge map is created. For example, for the two parent strings given below, the indicated edge map is created.

<table>
<thead>
<tr>
<th>Parent 1</th>
<th>1 2 3 4 5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent 2</td>
<td>9 4 10 6 5 2 8 1 3 7</td>
</tr>
</tbody>
</table>

edge map:

<table>
<thead>
<tr>
<th>Jobtype</th>
<th>1 2 3 4 5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent 1</td>
<td>1 2 3 4 5 6 7 8 9 10</td>
</tr>
<tr>
<td>Parent 2</td>
<td>9 4 10 6 5 2 8 1 3 7</td>
</tr>
</tbody>
</table>

In the above the blank cell means that it has been left without any continuing edges. Using edge map we always choose the job type which has the fewest edges since it has the higher probability of being isolated. Suppose the child string is initialized with one of two final job types of parent strings. In our example both job type 10 and job type 7 have two edges. So we make a random choice. Suppose that 7 has been chosen randomly. Then job type 7 is removed from the edge map. Next, candidate job types are 6 and 3 both of which have the same number of edges. Suppose that 6 is chosen randomly. Both candidate job type 5 and job type 10 have 2 unused edges. Then, suppose that 10 is chosen randomly. Next, candidate job type 9 is chosen because it has fewer number of edges. Job type 8 is chosen next since it is the only candidate. Continue this process until we get complete sequence. The resulting sequence is composed of edges entirely from the two parent strings:

Child 1: 1 3 4 5 2 8 9 10 6 7

C. Simulated Annealing Procedure

For the simulated annealing procedure (SA) implemented in our study, we used an acceptance probability function, which is independent of the change in the objective function value. Matsuo et al. [43] argued that the benefit of using the above acceptance probability function is that it facilitates designing experiments to empirically identify the best annealing schedule. Matsuo et al. [43] and Ogbu and Smith [46] showed that a simulated annealing approach with an acceptance probability function which is independent of the change in the objective function value, provided as good quality solutions as the
conventional (Metropolis) scheme in much less computation times. The simulated annealing algorithm works as follows:

1. Initialization—parameters of annealing schedule and an initial state.
2. Select a generation mechanism—a simple prescription to generate a transition from a current state to another state by a small perturbation.
3. Evaluate the new state; Compute $\Delta E =$ (value of current state – value of new state).
4. If new state is better than current state, make it current state; otherwise probabilistically accept (or reject it).
5. Based on stopping rules, either stop or continue iterations at step 2.

To determine an appropriate functional form of the acceptance probability, $AP_{ij}$, we considered linear and geometric forms. Based on extensive preliminary tests, the functional form of the acceptance probability, $AP_{ij}$, is set to decrease geometrically until the value of $AP_{ij}$ approaches 0 as we reach the stage limit. This probability is given by the function

$$AP_{ij}(k) = \frac{AP(1)}{AP(1) + (1 - AP(1))\lambda^{k-1}}$$

where $k = 1, 2, 3, \ldots, K$. Here $AP(1)$ is the initial value of the acceptance probability at stage 1, if ($\lambda < 1$) the reducing factor, $C$ the makespan, and $K$ the number of stage. The acceptance probability is constant at each stage. The acceptance probability given by the above two functions is 1 for all movements that result in better solutions and $AP(1)/\lambda^{k-1}$ for worse solutions. To specify the order in which neighbors are searched, random search and sequential search methods were considered. Since sequential search method performed slightly better (the difference is statistically insignificant at a significance level of 0.05 using a $t$-test), the search for the neighborhood of the current solution is implemented sequentially using the pairwise exchange improvement procedure. In addition to current solution, the best solution found so far is maintained for the case of a worse final solution.

D. Tabu Search Procedure

Three major components of the tabu search procedure used in this study are 1) a tabu list, 2) an aspiration level, and 3) a longer-term memory structure. For a detailed description of tabu search techniques applied to the flow-shop problem, refer to Widmer and Hertz [60] and Taillard [57]. The general procedure of the tabu search implemented in this study is described as follows:

Step 1. Initialize the number of stage, a tabu list, an aspiration level, and a longer-term memory structure.
Step 2. Select an initial sequence, and set it to the current solution.
Step 3. Evaluate all neighborhood of the current solution.
Step 4. Select as a new current solution a solution that minimizes the makespan among a set of candidates that do not belong to the tabu list or the tabu status of which was overridden by the aspiration level.
Step 5. Update the tabu list, the aspiration level, and the longer-term memory, and increment the number of stage by 1.
Step 6. If the new current solution is better than the best solution found up to now, set the best solution to the new solution.
Step 7. Return to step 3, if the number of stage does not exceed the maximum number of stage; Otherwise, return the best solution and stop.

As in the simulated annealing procedure, the neighborhood is defined by pair-wise exchange. The size of the neighborhood is $n(n-1)/2$. The number of stage is 80 to 100 depending on the problem sizes. Since the performance of the tabu search procedure is sensitive to the tabu list size, the tabu list size needed experimental tuning. If the tabu list size is large, the procedure can forbid too many moves, whereas if the tabu list size is small, the procedure may get trapped into cycles. Taillard [58] suggests that the tabu list size has to be set to about $(n+m)/2$. Based on our preliminary test, the tabu list size was set to $(n+m)/2$. In the tabu list, vector $(i, j)$ position $(i)$, position $(j)$ is maintained to prevent any swap in the near future which exchange job type $i$ and $j$ occupying position $(i)$ and position $(j)$ respectively [42].

The tabu list was updated according to the first come first out (FIFO) rule. To choose a promising move, the tabu status is overridden if an aspiration level is satisfied. In our implementation, any tabu status is overridden if the move leads to better solution than the best solution found so far. For the longer-term memory structure, we followed the procedure suggested by Taillard [58]. However, we adjusted the parameter values to achieve the best possible results. To penalize the repetition of the same exchange over a long term, we accumulated the number of times at which a job is moved to the left of its current position over time. The more frequently a job is moved to the left of its current position, the more the job is prevented from moving to the left in the future. In the evaluation of the neighborhood, we added the value of penalty function $Pf(j)$, to the makespan of the candidate solutions ($P$ refers to a parameter value and $f(j)$ refers to the frequency at which a job $j$ is moved to the left). Based on extensive preliminary tests, $P$ is calculated by $0.01\Delta_{max} \sqrt{n/(\Delta_{max})}$, refers to the maximal decreases of the makespan between two successive solution achieved until stage $k$).

REFERENCES

In Lee was born in 1959 in Angang, Korea. He received the B.B.A. degree from Keimyung University, Korea, and the MBA degree from the University of Texas, Austin. He is currently pursuing the Ph.D. degree in information systems at the University of Illinois, Urbana-Champaign.

His research interests are in the development of decision support systems, business process reengineering, and AI applications to manufacturing.

Mr. Lee is a member of the Decision Sciences Institute (DSI), the Institute for Operations Research and the Management Sciences (INFORMS), and the Association of Computing Machinery (ACM).

Riyaz Sikora was born on May 23, 1966, in Hyderabad, India. He received the B.E. degree in electronics and communication engineering from Osmania University, India, and the Ph.D. degree in information systems from the University of Illinois, Urbana-Champaign, specializing in artificial intelligence.

He is currently an Assistant Professor, Department of Industrial and Manufacturing Systems Engineering, University of Michigan, Dearborn. His research interests are in the area of manufacturing information systems with emphasis on the development and application of agent-based framework for systems integration, intelligent manufacturing, and application of artificial intelligence techniques.

Dr. Sikora serves on the editorial board of the International Journal of Computational Intelligence and Organizations, and is a member of the Association of Computing Machinery (ACM), the American Association of Artificial Intelligence (AAAI), the Association for Information Systems (AIS), the Institute for Operations Research and the Management Sciences (INFORMS), and the Decision Sciences Institute (DSI).

Michael J. Shaw (S’82–M’85) has been a Professor, Information Systems and Technology, Department of Business Administration, University of Illinois, Urbana-Champaign, since 1984. He is also a research faculty member at the Beckman Institute for Advanced Science and Technology, University of Illinois. In 1990, he was a Visiting Associate Professor, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA. He has been doing research in areas related to management information technology, manufacturing decision support, business process design, intelligent decision support systems, and artificial intelligence.
