Pergamon

0305-0548(95)00079-8

Computers Ops Res. Vol. 23, No. 9, pp. 857–868, 1996

0305-0548/96 $15.00 + 0.00

SIMULATED ANNEALING AND GENETIC ALGORITHMS FOR SCHEDULING PRODUCTS WITH MULTI-LEVEL PRODUCT STRUCTURE

Jung-Ug Kim† and Yeong-Dae Kim‡

Department of Industrial Engineering, Korea Advanced Institute of Science and Technology, Yusong-gu, Daejon 305-701, Korea

(Received December 1994; in revised form October 1995)

Scope and Purpose—Many manufacturing systems can be considered to be composed of two basic processing units, machining (or fabrication) and assembly, each of which processes various operations for various products. These two units are run or managed relatively separately once a production plan for each unit is determined. We consider a short term production scheduling problem for the production plans of the two units in such manufacturing systems. In general, products that are manufactured in those manufacturing systems have multi-level product structure. That is, those products are composed of several subassemblies and components, and each subassembly may also be composed of sub-subassemblies and components. In this study, operations are aggregated into two basic operations, machining and assembly; and processing units in the manufacturing system are also aggregated into a machining shop and an assembly shop. Since processing capacities of the system must be considered as well as precedence relationships among the items that result from the product structure, it is not easy to solve the problem optimally in a reasonable amount of time. In this paper, we apply two search techniques, simulated annealing and genetic algorithms, which are often used to find near optimal solutions in various optimization problems.

Abstract—We consider a short-term production scheduling problem in a manufacturing system producing products with multi-level product structure, and their components and subassemblies. The problem is to schedule production of components, subassemblies, and final products with the objective of minimizing the weighted sum of tardiness and earliness of the items. We consider due dates of final products, precedence relationships among items, and processing capacity of the manufacturing system. The scheduling problem is solved with simulated annealing and genetic algorithms, which are search heuristics often used for global optimization in a complex search space. To compare the performance of these algorithms with the finite loading method, which is often used in practice, computational experiments are carried out using randomly generated test problems and results are reported.

1. INTRODUCTION

Many manufacturing firms produce products with a multi-level product structure. In such a structure, final products are composed of subassemblies and components, and each subassembly is also made of lower-level subassemblies and/or components. An example of the product structure is shown in Fig. 1. This example shows a case in which the final product ($P\#1$) is composed of two subassemblies ($SA\#1$ and $SA\#3$) and one component ($C\#1$), while subassembly $SA\#1$ is made up of component $C\#2$ and subassembly $SA\#2$, and so on. Because of such product structure, these items have precedence relationships among them.

Since various operations are required to process each of these items (final products, subassemblies and components), planning and scheduling of production for each item in a detailed level may require excessive work. It is common practice to use an aggregate production planning approach, in which similar items or similar processes are aggregated and the resulting aggregated planning

†Jung-Ug Kim is a Ph.D. student at the Department of Industrial Engineering, Korea Advanced Institute of Science and Technology (KAIST). He received a B.A. in Economics from Yonsei University and an M.S. in Industrial Engineering from KAIST. His current research focuses on areas of planning and scheduling of manufacturing systems and transportation systems.

‡Yeong-Dae Kim is an associate professor at the Department of Industrial Engineering, KAIST. He received a B.S. degree from Seoul National University and an M.S. degree from KAIST both in Industrial Engineering, and a Ph.D. degree in Industrial and Operations Engineering from the University of Michigan.

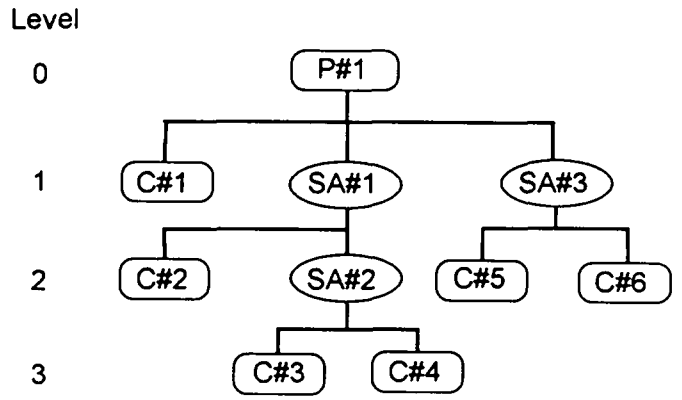

Fig. 1. An example of a product structure.

problem is solved and then the aggregate plan is disaggregated into a detailed schedule. In this paper, operations are aggregated into two basic ones, machining and assembly. Components require machining operations while subassemblies and final products require assembly operations. Similarly, processing units for the operations are also aggregated into a machining shop and an assembly shop. Therefore, it is assumed that the manufacturing system is composed of a machining shop and an assembly shop.

We consider a short term production scheduling problem after such aggregation. The objective is to determine when each item is processed in order to meet given requirements for the final products. It is assumed that due dates of final products are set by the customers or the sales department. If a product is completed later than its due date, a certain amount of tardiness penalty is incurred. Similarly, if it is completed earlier than its due date or delivery date, it also incurs a certain amount of earliness penalty. Likewise, components and subassemblies must be completed when they are needed.

We deal with an environment in which the scheduling horizon is divided into discrete time periods. For example, each time period may be a day or a shift. Most of the scheduling literature assumes that due dates and delivery times of jobs have continuous values. The discrete-time version of the earliness and tardiness problem does not seem to have drawn much attention in the literature, while the continuous-time version has been extensively studied (see Baker and Scuder [1] for a comprehensive survey). Such a continuous-time model cannot accurately deal with some practical situations. For instance, shipment may take place periodically. If a job is not ready by the delivery date, then it waits for the next shipment regardless of the actual completion time. Similar examples can be found in a make-to-forecast environment, where due-dates are assigned at the end of a reasonable forecasting sub-period (e.g. week or day) rather than in a continuous fashion.

In this paper, we suggest a solution procedure for the short-term production scheduling problem stated above. Since it is difficult to solve optimally the scheduling problem in a reasonable amount of time, we resort to heuristics. Here, we apply a simulated annealing (SA) algorithm and a genetic algorithm (GA) to our scheduling problem. We review these algorithms and present our implementation of the algorithms for the production scheduling problem. Also, the two search algorithms are compared with *finite loading* (Adam and Surkis [2], Bertrand [3]), which uses the concept of material requirements planning, through a computational experiment on randomly generated test problems.

In the next section, we describe the problem under study. Section 3 presents a brief review of SA algorithms and the application of the algorithms to the problem considered here, while Section 4 deals with those of the GA approach. In Section 5, computational experiments are carried out to find appropriate parameter values to be used in the two search algorithms and the two algorithms with the most appropriate parameter values are compared with the finite loading. Finally, Section 6 concludes the paper with a summary and recommendations for future research.

2. PROBLEM STATEMENTS

We consider a short-term production scheduling problem with the objective of minimizing the weighted sum of discrete earliness and tardiness penalties for the items. Weights of earliness and

tardiness of an item may be estimated from holding and shortage costs per period for the item, respectively (and other relevant costs if they exist). As mentioned earlier, if a product is completed later or earlier than its due date, a certain amount of tardiness or earliness penalty is incurred. Similarly, a penalty may be imposed on components and subassemblies that are not completed on time. An early completion of a component or a subassembly results in additional work-in-process inventory since it should stay in the system before it is put together into a subassembly or a final product for which it is needed. However, since components or subassemblies do not have specific due dates assigned to them, it would be meaningless to define tardiness or to charge the tardiness penalty. Tardiness of these items, even if we could define it, does not cost any real penalty itself but only affects the completion of the final products and that effect can be considered by tardiness or earliness of the final products. Therefore, we consider earliness penalties of all the items, but tardiness penalties of only final products.

In the problem, we must consider precedence relationships among items that result from the product structures of the products. That is, child items should be completed before processing of their parent item can be started. (Here, a child item represents a lower level component that belongs to a parent item.) Also, the resulting production schedules must satisfy constraints of the system capacity. In other words, the sum of processing times of components planned to be processed in a period cannot exceed processing time capacity of the machining shop, and the sum of processing times of items requiring assembly operations should not be greater than the capacity of the assembly shop. It is assumed in this research that due dates of final products are known in advance; processing time of an item does not exceed one period; and each of the items is processed at only one period and cannot be split.

For a more formal statement of the problem considered in this paper, we give a mathematical programming formulation of the problem although it is not used directly for solving the problem in this research. We use the following notation in the formulation.

i	subscript for items requiring machining operations,
j	subscript for items requiring assembly operations,
t	subscript for time periods,
P	set of final products,
$c(i), c(j)$	set of child items (predecessors) of items i and j, respectively,
ε_i^M	earliness penalty of component i for each period early,
ε_j^A	earliness penalty of subassembly (or final product) j,
τ_j^A	tardiness penalty of final product $j, j \in P$,
E_i^M	earliness of component i,
E_j^A	earliness of subassembly (or final product) j,
T_j^A	tardiness of final product $j, j \in P$,
d_j^A	due date of final product $j, j \in P$,
K_t^M	processing time capacity of the machining shop at period t,
K_t^A	processing time capacity of the assembly shop at period t,
p_i^M	processing time of component i,
p_j^A	processing time of subassembly (or final product) j,
X_{it}^M	$= 1$ if component i is processed at period t, and 0 otherwise,
X_{jt}^A	$= 1$ if subassembly (or final product) j is processed at period t, and 0 otherwise.

Now, we give the formulation

$$\text{minimize} \quad \sum_i \varepsilon_i^M E_i^M + \sum_j \varepsilon_j^A E_j^A + \sum_{j \in P} \tau_j^A T_j^A \tag{1}$$

subject to

$$\sum_t t X_{jt}^A - 1 - \sum_t t X_{it}^M = E_i^M \qquad \forall i \in c(j) \tag{2}$$

$$\sum_t t X_{jt}^A - 1 - \sum_t t X_{j't}^A = E_{j'}^A \qquad \forall j' \in c(j) \tag{3}$$

$$\sum_t t X_{jt}^A - d_j^A = T_j - E_j \qquad \forall j \in P \tag{4}$$

$$\sum_t X_{it}^M = 1 \qquad \forall i \tag{5}$$

$$\sum_t X_{jt}^A = 1 \qquad \forall j \tag{6}$$

$$\sum_i p_i^M X_{it}^M \leq K_t^M \qquad \forall t \tag{7}$$

$$\sum_j p_j^A X_{jt}^A \leq K_t^A \qquad \forall t \tag{8}$$

$$E_i^M, E_j^A, T_j^A \geq 0 \qquad \forall i, j \tag{9}$$

$$X_{it}^M, X_{jt}^A \in \{0, 1\} \qquad \forall i, j, t. \tag{10}$$

Constraints sets (2) and (3) define the earliness of components and subassemblies. Note that items must be processed (and hence completed) just one period earlier than their parent items, in order not to be penalized for earliness or tardiness. Earliness and tardiness of final products are defined with (4). Constraints (5) and (6) ensure that each item (component, subassembly or final product) is processed at one and only one period, while constraints (7) and (8) are included for the capacity constraints, i.e. the total processing time of items processed in each period must not exceed the time capacity of machining shop and assembly shop.

3. SIMULATED ANNEALING ALGORITHM

The SA approach can be viewed as an enhanced version of local optimization or an iterative improvement, in which an initial solution is repeatedly improved by making small local alterations until no such alteration yields a better solution. SA randomizes this procedure in a way that allows occasional *uphill moves* (alterations that worsen the solution) in an attempt to increase the probability of leaving a locally optimal solution [4]. SA algorithms have been used successfully in solving various combinatorial problems including VLSI design [5], multilevel lot sizing [6], quadratic assignment [7–10], scheduling [11–15], cell formation [16, 17], assembly line balancing [18], and process allocation problems [19]. Applications of SA to various problems are surveyed in Koulamas *et al.* [20].

As mentioned earlier, SA attempts to avoid entrapment in a local optimum by sometimes accepting a move that deteriorates the value of the objective function $f(\cdot)$. Starting from an initial solution, SA generates a new solution S' in the neighbourhood of the original solution S. Then, the change in the objective function value, $\Delta = f(S') - f(S)$, is calculated. For a minimization problem, if $\Delta < 0$, the transition to the new solution is accepted. If $\Delta \geq 0$, then the transition to the new solution is accepted with probability, usually denoted by the function, $\exp(-\Delta/T)$, where T is a control parameter called the *temperature*. SA algorithms generally start with a high temperature and then the temperature is gradually lowered. At each temperature, a search is carried out for a certain number of iterations, called the *epoch length*.

In this paper, a solution is encoded to a string of numbers representing priorities of the items. The encoded string has one number for each item. Positions of the numbers in the string correspond to the indexes of the items, while their values denote the priorities of their corresponding items. Since items are classified into two categories in this paper, the strings are logically divided into two substrings, one for components and the other for subassemblies and final products. When encoding the solutions, we use the concept of *random keys* suggested by Bean [21]. In the encoding scheme, values of the numbers in the string are originally selected from uniform random numbers in the region [0, 1]. These values are used as sort keys to decode the string into a feasible solution. Advantages of this encoding scheme are that the scheme is robust to problem structures and it is easy to obtain feasible solutions after perturbing a given solution.

A feasible schedule can be generated from a string of numbers as follows. From a given string, priorities of items are obtained first. (Here, smaller values represent lower priorities.) Final products are scheduled to be processed on their due periods. Among available items of which the parent item has been scheduled, an item with the lowest priority is selected and the period for its processing is determined considering the period for its parent item. If an item cannot be processed because of

(a)

Item	P#1	SA#1	SA#2	SA#3	C#1	C#2	C#3	C#4	C#5	C#6
Sol. Value	0.27	0.74	0.31	0.54	0.79	0.45	0.93	0.65	0.14	0.69

(b)

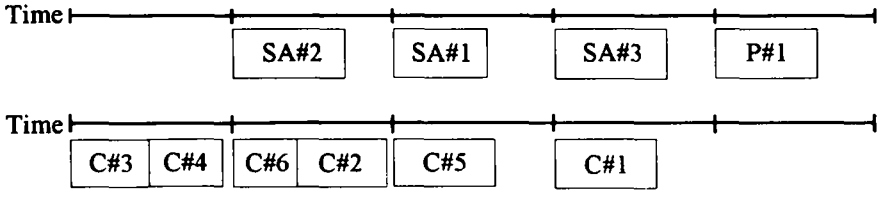

Fig. 2. An example of a solution and corresponding schedule. (a) An example of solution representation. (b) Corresponding schedule.

overload in the period it should be processed, it is scheduled to be processed one (or several) periods earlier. If a feasible schedule cannot be obtained, one of the final products is selected randomly and its processing is delayed by one period and then a new schedule is generated using the above procedure. This schedule revision is repeated until a feasible schedule is acquired. Figure 2 shows an example of the encoding scheme and a feasible solution decoded from a string for the product whose structure is given in Fig. 1.

In search algorithms, the *neighbourhood* of a solution can be obtained with a certain alteration of the current solution. As a simple method for generating neighbourhood, an *insertion* or *interchange* method is commonly used. If the above encoding scheme is used, a number is selected and inserted between certain two consecutive numbers in the former method, while two numbers are selected and exchanged with each other in the latter. As another method, we suggest the *one-position random change method*, in which a number in the string of numbers is selected and its value is altered randomly. Figure 3 shows how neighbourhood solutions are generated by the three methods with a simple example. Although they are not reported in this paper, results of a preliminary test on the three methods revealed that the one-position random method gave better solutions than the other two methods.

In general, SA algorithms are specified by several parameters and/or methods, namely the initial temperature t_0, the epoch length L, the rule specifying how the temperature is reduced, and the termination condition. The initial temperature is often chosen in such a way that the fraction of accepted uphill moves in a trial run of the annealing process becomes approximately F_0, a parameter to be selected. That is, after the mean increase of objective value $\bar{\Delta}$ is computed with uphill moves only, the value of t_0 is obtained from the equation, $\exp(-\bar{\Delta}/t) = F_0$. In our algorithm, the number of transitions in the trial run is set to be equal to 20 times the number of items.

The epoch length denotes the number of moves made with the same temperature. The epoch length can be set as qs_N, where q is a parameter to be determined and s_N is the number of neighbourhood solutions for a given solution. We set s_N to be the number of items in our algorithm. In SA algorithms, the temperature should be decreased in such a way that the cooling process would not take too long. The method, which appears to be the most common in the current literature, specifies the temperature with $t_k = rt_{k-1}$, during the kth epoch ($k = 1, 2, \ldots$), where r is a parameter, called the *cooling ratio*, with a value less than 1. Higher cooling ratios correspond to a slower cooling process, and therefore more moves are required before the process is frozen (terminated). Different methods for decreasing the temperature are presented by Osman and Potts [12] and Van Laarhoven *et al.* [15].

As a criterion to terminate the algorithm we use one given by Johnson *et al.* [4], in which a counter

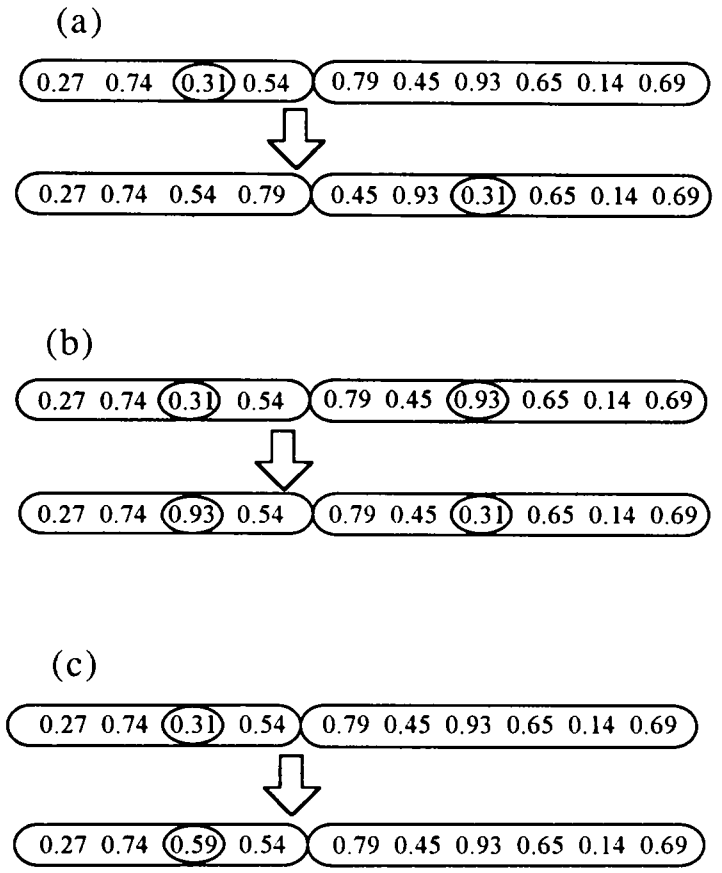

Fig. 3. Methods for neighbourhood generation in SA. (a) Insertion. (b) Interchange. (c) One-position random change.

is incremented by one when an epoch is completed with the fraction (or percentage) of accepted moves less than a predetermined limit, $F_{\min}$. This counter is reset to 0 when a new incumbent solution is found. In our algorithm, the search procedure is stopped when the counter reaches 5. Also, our SA algorithm is terminated when the computation time exceeds the predetermined time limit to prohibit excessive computation time. Of course, there are other criteria, for example, the one suggested by Kouvelis and Chiang [8].

4. GENETIC ALGORITHM

A genetic algorithm (GA) is a search technique for global optimization in a complex search space. The algorithm mimics the process of natural evolution by combining the concept of *survival of the fittest* among solutions with a structured, yet randomized information exchange and offsprings creation [22]. GAs have been used for many difficult optimization problems such as scheduling [23–25], assignment [26, 27], assembly line balancing [28], machine-component grouping [29], and facility layout problems [30].

In a simple GA, a candidate solution (called a *chromosome*) is represented by a string of numbers called *genes*. A chromosome's potential as a solution is measured by the *evaluation function*, which evaluates a chromosome with the objective function value. A judiciously selected set of chromosomes is called a *population* and the population at a given time is called a *generation*. The underlying fundamental mechanism of GAs consists of three main operations: *reproduction*, *crossover* and *mutation*. Chromosomes resulting from these operations applied to those in the current population, often known as *offspring* or *children*, form a population of the next generation. The procedure of generating a next generation is repeated for a certain number of times, which can be determined in various ways.

An application of a GA can be characterized by: (1) a chromosomal representation (encoding

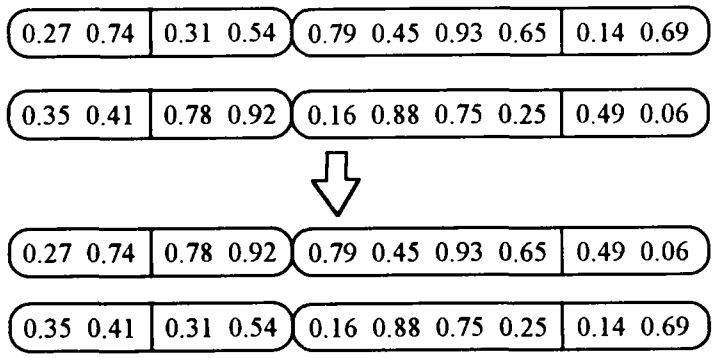

Fig. 4. A simple example of crossover.

scheme) of a solution; (2) an initial population; (3) an evaluation function for rating solutions in terms of their fitness; (4) genetic operators (reproduction, crossover and mutation) that determine the composition of offspring for the next generation; and (5) a termination rule. In the following, we describe how these factors are determined or selected in our implementation of GAs.

4.1. Encoding scheme

A solution is represented by the same method as that for the simulated annealing algorithms described in the previous section. A chromosome representing a solution has one gene for each of the items. The position of a gene corresponds to the index of an item while its value represents the priority.

4.2. Initial population

Gupta *et al.* [24] reported that the use of seeded populations provides little advantage but a little faster convergence. Therefore, we generate an initial population randomly in this research.

4.3. Evaluation function

Mapping an original objective function value to a fitness value that represents relative superiority (or inferiority) of chromosomes is a feature of an evaluation function. In a problem with the objective function to be minimized, a certain transformation is needed since we usually maximize fitness in GAs. In our algorithm, we use the *reciprocal method* in which the following transformation is applied: $f_i = \sum_j g_j/g_i$, where f_i and g_i are the transformed and the original objective function values of chromosome i, respectively.

In addition, it is needed to scale an objective function value into a fitness value in order to diversify the population. We use the *linear scaling* scheme suggested by Anderson and Ferris [28]. In the scheme, the scaled fitness value for a chromosome i is obtained with $F_i = mf_i + c$, where f_i is the raw fitness value and m and c are constants. The scaled values, F_i, $i = 1, \ldots, N$, constitute a scaled fitness distribution that satisfies $\sum_{i=1}^{N} F_i = 1$, $\max F_i = \lambda\left(\sum_{i=1}^{N} F_i/N\right)$, and $F_i \geq 0$ for all i, where λ is called the *scaling factor*. A detailed procedure for determining m, c and λ is presented by Anderson and Ferris [28].

4.4. Reproduction

Reproduction is a process in which individual chromosomes are copied according to their scaled fitness function values, i.e. chromosomes with a higher fitness value would have more of their copies at the next generation. This can be done by randomly selecting (with replacement) and copying chromosomes with probabilities that are proportional to the fitness values of the chromosomes.

4.5. Crossover

After reproduced chromosomes constitute a new population, crossover is performed. Crossover introduces new chromosomes by recombining current genes. Two chromosomes are randomly selected without replacement, and substrings after a crossover point is interchanged with a probability called the *crossover rate*. The crossover point is randomly selected in between two consecutive numbers in the string. Crossover is applied to each of the two substrings representing

items requiring machining and those requiring assembly. This process is repeated until all chromosomes in the population are mated. Figure 4 shows an example in which new chromosomes are created by interchanging the substrings to the right of the crossover points, which are indicated by a vertical bar (|) in the figure.

4.6. Mutation

Mutation introduces random changes to the chromosomes by altering the value of a gene with a probability called the *mutation rate*. In our application, genes of a chromosome are considered one by one and the value of the gene being considered is altered to another value with probability equal to the mutation rate. This operation is applied to all chromosomes in the population.

4.7. Termination rate

In our implementation of the GA, the search procedure is terminated when the best objective function value found so far is not updated for a predetermined number of generations. It can also be terminated when the computation time exceeds a predetermined time limit.

Various control parameters (such as crossover rate, mutation rate, population size, and the limits for the number of generations) play a crucial role for a successful implementation of a GA. Performance of a GA is significantly affected by these parameter values. We test several values for each of these parameters in the next section.

5. COMPUTATIONAL EXPERIMENTS

To implement the two search algorithms described in the previous sections, we determine values of various parameters. In this section, we first select a few sets of parameter values that give good results for each algorithm through a series of experiments using a number of randomly generated test problems. Then, using 150 randomly generated test problems the two algorithms are compared with a method called finite loading, which is often used in practice for production scheduling [2, 3]. In the finite loading method, items are assigned to time periods starting with final products. First, processings of final products are assigned to their due dates (periods) and other items are assigned working down in the product structure. If addition of an item to a period makes the capacity limit violated, processing of the item is shifted further forward to a time period where sufficient capacity is available. If a feasible schedule cannot be generated, one of the final products is assigned to periods some time (usually one period) after their due periods and the above procedure is repeated.

In the test problems, the number of components ranges from 100 to 140 (100, 120 and 140) while the number of final products and subassemblies is 40. Product structure for each final product is randomly generated and the number of item levels in the structure ranges from 3 to 5. The length of the planning horizon is set to 30, and other required data, such as processing times of the items and due dates of the final products, are generated randomly from discrete uniform distributions. Due dates of final products range from 21 to 30, while the processing times of components range from 0.05 to 0.4 (in periods) and those of subassemblies and final products range from 0.3 to 0.7. (The processing time capacity of each shop is 1.) The earliness penalties of the items (ε_i^M, ε_j^A) in the test problems are proportional to the processing times of the items including those of their descendant items if they exist, and the tardiness penalty of a final product (τ_j^A) is set 10 times the earliness penalty of the product. (Here, we assume that the inventory holding cost and the shortage cost of an item are proportional to the value (added) of the item and that the value (added) is proportional to the processing time.) All the heuristics were coded in C language and run on a personal computer with a Pentium processor.

5.1. Determination of parameter sets

Since it is well known that performance of a SA algorithm is affected by various parameters, we carefully select values of these parameters. For this purpose, various combinations of values for three parameters (r, q, F_0) were tested on 15 problems with a CPU time limit of 10 minutes. We tested three levels for r (0.75, 0.8 and 0.85), three levels for q (1, 3 and 5) and four levels for F_0 (0.1, 0.2, 0.3 and 0.4). In the SA algorithms, the one-position random change method was used for neighbourhood generation, since it was better than the other two methods as discussed in Section 3.

Table 1. Mean RDIs of SA algorithms

		F_0			
r	q	0.1	0.2	0.3	0.4
0.75	1	0.540	0.320	0.430	0.531
	3	0.322	0.330	0.295	0.289
	5	0.176	0.218	0.214	0.275
0.8	1	0.495	0.573	0.323	0.363
	3	0.173	0.220	0.238	0.337
	5	0.233	0.273	0.323	0.291
0.85	1	0.526	0.457	0.330	0.398
	3	0.210	0.229	0.128	0.322
	5	0.281	0.321	0.237	0.317

Table 2. Mean RDIs of GAs

		P_{crs}			
P_{mut}	λ	0.6	0.7	0.8	0.9
0.001	2.2	0.528	0.378	0.467	0.414
	2.5	0.610	0.433	0.474	0.398
	2.8	0.582	0.480	0.453	0.361
0.005	2.2	0.530	0.344	0.264	0.303
	2.5	0.314	0.259	0.296	0.372
	2.8	0.387	0.457	0.460	0.389
0.01	2.2	0.402	0.387	0.396	0.289
	2.5	0.337	0.344	0.312	0.401
	2.8	0.420	0.355	0.384	0.346

The test results are given in Table 1. Since optimal solutions cannot be obtained in a reasonable amount of time, we compare the solutions with the *relative deviation index* (RDI), which is defined as $(T_a - T_B)/(T_W - T_B)$ for parameter set a, where T_a, T_B, and T_W are the solutions of parameter set a, the parameter set which gave the smallest solution value, and of the parameter set which gave the largest solution value, respectively. Therefore, the index has a value between 0 and 1.

From the table, we can see the algorithm with $(r, q, F_0) = (0.85, 3, 0.3)$ was better than the others. In most cases, SA algorithms with larger values of q gave better performance. In other words, SA algorithms gave better solutions when a wide solution space could be searched for a transition. For a comparison with other algorithms, we select five parameter sets, $(r, q, F_0) = (0.75, 5, 0.1)$, (0.75, 5, 0.3), (0.8, 3, 0.1), (0.85, 3, 0.1) and (0.85, 3, 0.3), since they gave better solutions than others on average.

We also selected appropriate parameter values in GAs. In order to determine the termination condition, we tested several sets of values for crossover rate (P_{crs}), mutation rate (P_{mut}), and scaling factor (λ) while fixing the population size at 80 and plot solution improvements against the number of generations. From this test, we found that improvements were made once in very many generations until the very late stages of the search procedure. Considering these results, we let the search be terminated when there is no improvement for 100 consecutive generations or when the computation time reaches 10 minutes.

Using this termination condition, we tested several values for each of the above parameters. Included in the test were combinations of four levels for P_{crs} (0.6, 0.7, 0.8 and 0.9), three levels for P_{mut} (0.001, 0.005 and 0.01), and three levels for λ (2.2, 2.5 and 2.8). The test results are given in Table 2. The table shows that the algorithm with $(P_{crs}, P_{mut}, \lambda) = (0.7, 0.005, 2.5)$ was better than the others. The scaling factor and crossover rate had less effect on performance than the mutation rate. Since it is important in an implementation of GAs to keep the balance between preserving good current solutions and exploring new solutions, the crossover rate and mutation rate should not be considered separately. (There are interaction effects between these two rates.) Generally, computation time was not much affected by the values of the parameters. We select five parameter sets, $(P_{crs}, P_{mut}, \lambda) = (0.7, 0.005, 2.5)$, (0.8, 0.005, 2.2), (0.8, 0.005, 2.5) (0.9, 0.005, 2.2) and (0.9, 0.01, 2.2) for GAs to be included in the final tests.

Table 3. Description of the algorithms included in the test

Algorithms	Parameter values		
SA algorithms			
SA1	$r = 0.75$	$q = 5$	$F_0 = 0.1$
SA2	$r = 0.75$	$q = 5$	$F_0 = 0.3$
SA3	$r = 0.80$	$q = 3$	$F_0 = 0.1$
SA4	$r = 0.85$	$q = 3$	$F_0 = 0.1$
SA5	$r = 0.85$	$q = 3$	$F_0 = 0.3$
Genetic Algorithms			
GA1	$P_{crs} = 0.7$	$P_{mut} = 0.005$	$\lambda = 2.5$
GA2	$P_{crs} = 0.8$	$P_{mut} = 0.005$	$\lambda = 2.2$
GA3	$P_{crs} = 0.8$	$P_{mut} = 0.005$	$\lambda = 2.5$
GA4	$P_{crs} = 0.9$	$P_{mut} = 0.005$	$\lambda = 2.2$
GA5	$P_{crs} = 0.9$	$P_{mut} = 0.01$	$\lambda = 2.2$

Table 4. Performance of the algorithms

		CPU time (seconds)		
Algorithms	RDI	$n = 100$	$n = 120$	$n = 140$
SA1	0.159 (0.24)	422.1 (8.25)	588.2 (6.03)	594.3 (7.92)
SA2	0.172 (0.24)	454.0 (13.46)	588.9 (5.37)	594.2 (8.07)
SA3	0.195 (0.26)	329.0 (10.56)	502.8 (14.26)	589.0 (4.43)
SA4	0.175 (0.25)	448.8 (12.54)	585.9 (3.19)	588.8 (4.26)
SA5	0.161 (0.25)	476.2 (12.10)	586.3 (2.36)	589.1 (4.79)
GA1	0.402 (0.32)	597.9 (0.65)	598.8 (0.81)	598.9 (1.09)
GA2	0.396 (0.30)	598.2 (0.81)	598.9 (0.88)	598.7 (1.00)
GA3	0.401 (0.30)	598.3 (0.73)	598.8 (0.84)	598.4 (1.05)
GA4	0.403 (0.30)	598.1 (0.78)	598.9 (0.91)	598.9 (1.06)
GA5	0.384 (0.29)	598.1 (0.77)	598.1 (1.13)	598.9 (1.10)
FL	0.830 (0.33)		less than 1	

Figures in the table denote means and standard deviations (in parentheses) of RDIs and CPU times.

5.2. *Comparison of the algorithms*

In this subsection, we compare three algorithms for solving the problem, the finite loading method (FL), SA algorithms and GAs with parameter sets selected in the previous section. Table 3 gives a brief description of the individual SA algorithms and GAs included in the comparison. For the comparison, we randomly generated 50 problems for each of three levels for the number of components (100, 120 and 140) with a method similar to the one explained at the beginning of this section. The results of the tests are shown in Table 4, which includes the means and standard deviations of RDIs and CPU times for each problem size. Since the performance of the algorithms was indifferent to the problem sizes, the results were combined to show the overall mean and standard deviation.

In most problems, SA algorithms and GAs performed better than FL. Although SA algorithms and GAs require longer computation time than FL, which needed less than 1 s, this may be compensated for by improvements in the solution. Noticeably, SA algorithms outperformed GAs in our problems. A reason for the inferiority of GAs seems to be that we could not take full advantage of the merits of GAs in this specific implementation on the scheduling problem considered here. In GAs, it is expected that good properties of the solution structure are inherited from generation to generation and that better solutions be acquired by exchanging portions of the solutions. In our solution representation, however, information on the periods in which items are processed may not be well preserved due to the capacity constraints, although information on the sequence of processing may be preserved.

Attention should be given to the fact that the results of the above comparison are not general or complete. We cannot argue from these results that SA algorithms may outperform GAs in all or most combinatorial optimization problems. Also, we do not claim our computational experiments are complete and perfect even for the problem considered here, since there may be other ways of selecting parameter sets and their values and other measures can be used to evaluate performance of search algorithms. Moreover, there are still many ways of improving the two algorithms if characteristics of the problem are well studied. However, we can argue that with currently well

known techniques for the two search algorithms and with an equal amount of work for application of these to the problem considered here, SA algorithms may work better than GAs.

6. CONCLUDING REMARKS

We considered a short-term production scheduling problem for products with multi-level product structures, with the objective of minimizing the weighted sum of discrete earliness and tardiness of components, subassemblies and final products. Since it is difficult to solve the problem optimally in a reasonable amount of time, we applied simulated annealing and genetic algorithms. The two search algorithms were compared with a commonly used approach called finite loading after a series of tests for finding the most appropriate values for various parameters needed in the algorithms. The results showed that the two search algorithms performed better than the finite loading method and the simulated annealing algorithm performed better than the genetic algorithm although it cannot be argued that our results are final.

We considered the circumstances in which each of the items can be and should be processed at only one period. However, in many real systems, two or more periods are required for an item, or it may be better to split processing of an item even though it can be done in a single period. Studies on these cases are needed to obtain better schedules in real production systems. Since the performance of SA algorithms and GAs is significantly affected by the parameters needed in the algorithms, they should be carefully determined. However, it is not easy to adapt a search algorithm to a given problem, i.e. to find parameter values that are most appropriate for a given problem (time-consuming and tedious work is needed to obtain the best values). Research into the methodology of adapting SA algorithms or GAs to certain types of problems may be needed for better use of these search algorithms.

Acknowledgement—We would like to thank two anonymous referees for valuable comments that have substantially improved the paper.

REFERENCES

1. K. R. Baker and G. R. Scudder, Sequencing with earliness and tardiness penalties: a review. *Opns Res.* **38,** 22–36 (1990).
2. N. Adam and J. Surkis, A comparison of capacity planning techniques in a job shop control system. *Mgmt Sci.* **23,** 1011–1015 (1977).
3. J. W. M. Bertrand, The effect of workload dependent due-dates on job shop performance. *Mgmt Sci.* **29,** 799–816 (1983).
4. D. Johnson, C. Aragon, L. McGoech and C. Schevon, Optimization by simulated annealing: an experimental evaluation; part I, graph partitioning. *Opns Res.* **37,** 865–892 (1989).
5. S. Kirkpatrick, C. D. Gelatt, Jr. and M. P. Vecchi, Optimization by simulated annealing. *Science* **220,** 671–680 (1983).
6. R. Kuik and M. Salomon, Multi-level lot sizing problem: evaluation of a simulated annealing heuristics. *Eur. J. Opl Res.* **45,** 25–37 (1990).
7. D. T. Connolly, An improved annealing scheme for the QAP. *Eur. J. Opl Res.* **46,** 93–100 (1990).
8. P. Kouvelis and W. Chiang, A simulated annealing procedure for single row layout problems in flexible manufacturing systems. *Int. J. Prod. Res.* **30,** 717–732 (1992).
9. P. S. Laursen, Simulated annealing for the QAP—optimal tradeoff between simulation time and solution quality. *Eur. J. Opl Res.* **69,** 238–243 (1993).
10. M. R. Wilhelm and T. L. Ward, Solving quadratic assignment problems by simulated annealing. *IIE Trans.* **19,** 107–119 (1987).
11. D. E. Jeffcoat and R. L. Bulfin, Simulated annealing for resource-constrained scheduling. *Eur. J. Opl Res.* **70,** 43–51 (1993).
12. I. H. Osman and C. N. Potts, Simulated annealing for permutation flow-shop scheduling. *Omega, Int. J. Mgmt Sci.* **17,** 551–557 (1989).
13. J. Sridhar and C. Rajendran, Scheduling in a cellular manufacturing system: a simulated annealing approach. *Int. J. Prod. Res.* **31,** 2927–2945 (1993).
14. A. J. Vakharia and Y. L. Chang, A simulated annealing approach to scheduling a manufacturing cell. *Naval Res. Logist.* **37,** 559–577 (1990).
15. P. J. M. Van Laarhoven, E. H. L. Aarts and J. K. Lenstra, Jobshop scheduling by simulated annealing. *Opns Res.* **40,** 113–125 (1992).
16. W. H. Chen and B. Srivastava, Simulated annealing procedure for forming machine cells in group technology. *Eur. J. Opl Res.* **75,** 100–111 (1994).
17. V. Venugopal and T. T. Narendran, Cell formation in manufacturing systems through simulated annealing: an experimental evaluation. *Eur. J. Opl Res.* **63,** 409–422 (1992).
18. G. Suresh and S. Sahu, Stochastic assembly line balancing using simulated annealing. *Int. J. Prod. Res.* **32,** 1801–1810 (1994).
19. S. Sofianopoulou, Simulated annealing applied to the process allocation problem. *Eur. J. Opl Res.* **60,** 327–334 (1992).
20. C. Koulamas, S. R. Antony and R. Jean, A survey of simulated annealing applications to operations research problems. *Omega, Int. J. Mgmt Sci.* **22,** 41–56 (1994).

21. J. C. Bean, Genetic algorithms and random keys for sequencing and optimization. *ORSA J. Computing* **6,** 154–160 (1994).
22. D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley (1989).
23. J. E. Biegel and J. J. Davern, Genetic algorithms and job shop scheduling. *Comps and Ind. Eng.* **19,** 81–91 (1990).
24. C. G. Gupta, Y. P. Gupta and A. Kumar, Minimizing flow time variance in a single machine system using genetic algorithms. *Eur. J. Opl Res.* **70,** 289–303 (1993).
25. E. S. H. Hou, A genetic algorithm for multiprocessor scheduling. *IEEE Trans. Parallel and Distributed Systems* **5,** 113–120 (1994).
26. K. C. Chan and H. Tansri, A study of genetic crossover operations on the facilities layout problem. *Comps and Ind. Eng.* **26,** 537–550 (1994).
27. G. Levitin and J. Rubinovitz, Genetic algorithm for linear and cyclic assignment problem. *Comps and Opns Res.* **20,** 575–586 (1993).
28. E. J. Anderson and M. C. Ferris, Genetic algorithms for combinatorial optimization: the assembly line balancing problem. *ORSA J. Computing* **6,** 161–173 (1994).
29. V. Venugopal and T. T. Narendran, A genetic algorithm approach to the machine-component grouping problem with multiple objectives. *Comps and Ind. Eng.* **22,** 469–480 (1992).
30. K. Y. Tam, Genetic algorithms, function optimization, and facility layout design. *Eur. J. Opl Res.* **63,** 322–346 (1992).