ORSA Journal on Computing
Vol. 7, No 4, Fall 1995

0899-1499/95/0704-0443 $01.25
© 1995 INFORMS

Solving a Class Scheduling Problem with a Genetic Algorithm

JEFFREY W. HERRMANN / *Institute for Systems Research, A.V. Williams Building, University of Maryland, College Park, MD 20742; Email: jwh2@eng.umd.edu*

CHUNG-YEE LEE / *Department of Industrial and Systems Engineering, 303 Weil Hall, University of Florida, Gainesville, FL 32611, Email: lee@ise.ufl.edu*

(Received: June 1992; revised: May 1994; accepted: June 1994)

In this paper, we study the one-machine scheduling problem of minimizing the total flowtime subject to the constraint that each job must finish before its deadline, where the jobs to be scheduled fall into different job classes and setups occur whenever the machine processes consecutive jobs from different classes. A new heuristic is proposed for the problem. We investigate the use of a genetic algorithm to improve solution quality by adjusting the inputs of the heuristic. We present experimental results that show that the use of such a search can be successful.

The processing of jobs on a machine or a set of machines often involves a significant cost or time when changing from one job to the next; this cost or time is called the changeover or *setup.* If the setup for a job is independent of the job that was processed before it, the setup time or cost can be included in the processing requirements of that job. If the setup for a job is *sequence-dependent*, that is, the cost or time of the setup depends upon the immediate predecessor, the scheduling problem becomes quite difficult.

One important problem with sequence-dependent setups is the *class scheduling problem*, where the jobs to be processed form a number of job *classes*. There exists no setup between jobs from the same class, although there does exist a special setup, the *class setup*, when a job from one class is processed after a job from another class. These class setups may be sequence-dependent in that they depend upon both the class of the current job and the last class processed. There may also exist a class setup for the first job processed. In this paper, we wish to minimize the sum of the job completion times subject to the constraint that no job may be tardy.

Monma and Potts[23] describe a few examples of class scheduling problems, including paint production machines that are cleaned between the production of different colors, a computer system that must load the appropriate compiler for a type of task, and a limited labor force with workers switching between two or more machines.

This research is motivated by the scheduling of semiconductor test operations. Assembled semiconductor devices must undergo electrical testing on machines that can test a number of different types of semiconductors. If a machine is scheduled to test a lot consisting of devices that are different from the devices tested in the previous lot, various setup tasks are required. These tasks may include changing a handler and load board that can process only certain types of semiconductor packages or loading a new test program for the new part. However, if the new lot consists of circuits that are the same as the previous type, none of this setup is required. This type of change is a sequence-dependent setup that can be modelled by class scheduling.

Since post-assembly testing is the last stage in semiconductor manufacturing, meeting a job's due date is a very important objective for the manager of a test facility. A secondary criterion is the minimization of total flowtime (the sum of the job completion times), which reflects the manager's desire to increase throughput and decrease inventory holding costs.

This is a dual criteria problem, in which the primary criterion is used as a constraint and the secondary criterion is optimized under this restriction. The problem of minimizing the total flowtime subject to deadlines is an old problem. Smith[30] provides an optimal solution technique that repeatedly schedules the longest eligible job last. The class scheduling version of the problem, however, is a more difficult question.

For our problem, finding a feasible schedule is an NP-complete problem. (A schedule is feasible if every job finishes before or at its deadline.) Thus, there exist no exact algorithms to minimize in polynomial time the total flowtime subject to the deadline constraints. (For a discussion of the theory of NP-completeness, see Garey and Johnson.[12]) Thus, we are motivated to try different heuristics. In this paper we develop a multiple-pass heuristic that finds good solutions quickly. The first contribution of our investigation of this problem is the extension of Smith's algorithm into a heuristic which considers the setup times while sequencing the jobs by their deadlines and processing times.

We are also interested in using a genetic algorithm to improve the quality of our solutions. A genetic algorithm is

Subject classifications. Production/scheduling. deterministic, sequencing, sequence-dependent set up.

a heuristic search that has been used to find good solutions to a number of different optimization problems, but genetic algorithms searching for good schedules must overcome the difficulty of manipulating the sequences of jobs. We investigate the use of a genetic algorithm to search a new type of space, the problem space. This type of approach was introduced by Storer, Wu, and Vaccari,[33] who consider alternative search spaces for the general job shop scheduling problem. In this paper, we extend the idea to the problem of one-machine class scheduling.

Our genetic algorithm attempts to adjust the deadlines of the given problem so that our heuristic will find even better solutions. The space of adjusted deadlines that we search forms a *problem space* (problem and heuristic space will be described more fully in Section 4). The second contribution of this paper is our use of this method to improve the scheduling of a single-machine problem.

If we use the principles of Davis,[8] then we can classify our genetic algorithm as a type of hybrid genetic algorithm. However, the only unusual characteristic of our algorithm is the decoding (the bit string does not describe a point in the solution space; instead it must be mapped to a solution via the heuristic). Moreover, the range of hybrid genetic algorithms is so large (for instance, Goldberg[13] describes hybrids differently) that our use of the term *problem space genetic algorithm* is a more precise description of the search. Finally, this problem space exists independently of the genetic algorithm, and the use of this new search space is not limited to our search. Other searches (including steepest descent, simulated annealing, and tabu search) could be used to explore the space. Therefore, we will continue to refer to our search space as a problem space and to our search as a problem space genetic algorithm.

The next section of this paper summarizes some of the relevant literature on class scheduling problems and the dual criteria objective under consideration. In Section 2, we discuss our notation, an example instance of the problem, and a number of basic results. We discuss in Section 3 the heuristic developed for the problem. Our genetic algorithm will employ this heuristic. In Section 4, we present a problem space, introduce genetic algorithms, and discuss the details of the genetic algorithm we developed to search the problem space. Section 5 describes the generation of the sample problems, the computational experiments, and the results. Finally, in Section 6, we present our conclusions and some directions for future research.

1. Literature Review

In this section we will summarize some of the relevant research on class scheduling and on the dual criteria problem of minimizing total flowtime subject to job deadlines. We will review the literature on genetic algorithms in Section 4.

One of the first papers on problems with class scheduling characteristics is Sahney,[29] who considers the problem of scheduling one worker to operate two machines in order to minimize the flowtime of jobs that need processing on one of the two machines. Sahney derives a number of optimal properties and uses these to derive a branch-and-bound algorithm for the problem. Gupta[14] defines the class scheduling problem, and Potts[27] and Coffman, Nozari, and Yannakakis[5] also study two-class scheduling problems and develop algorithms to minimize total flowtime in polynomial time. Ho[18] studies the problem of minimizing the number of tardy jobs with two job classes.

Bruno and Downey[3] prove that, for more general class scheduling problems, the question of finding a schedule with no tardy jobs is NP-complete. Monma and Potts[23] prove that many class scheduling problems are NP-complete, including minimizing makespan, maximum lateness, the number of tardy jobs, total flowtime, and weighted flowtime. They also develop a dynamic programming algorithms that are exponential in the number of classes to solve the problem.

Dobson, Karmarkar, and Rummel[9] consider a class scheduling problem with the objective of minimizing flowtime under both the item-flow and batch-flow delivery schedules. Dobson, Karmarkar, and Rummel[10] extend this work to uniform parallel machines. Gupta[15] and Ahn and Hyun[1] study the class scheduling problem of minimizing flowtime and present heuristics to find near-optimal schedules. We will modify the procedure of Ahn and Hyun in order to use it as a comparative heuristic. Mason and Anderson[22] study the problem with the objective of minimizing weighted flowtime and develop a branch-and-bound algorithm using some dominance criteria and a lower bound. The only other dual criteria problem in this area is studied by Woodruff and Spearman;[36] they consider a class scheduling problem with profit maximization and deadlines and use a tabu search to find good solutions.

In the dual criteria literature, the problem of minimizing total flowtime subject to job deadlines (a deadline is a constraint on the completion time) is among the oldest questions, being first studied by Smith.[30] The problem of minimizing the weighted flowtime subject to job deadlines is a strongly NP-complete problem,[20] and different elimination criteria and branch-and-bound techniques have appeared. Potts and Van Wassenhove[28] study the Lagrangian relaxation of the deadline constraints, leading to the discovery of optimal solutions. Work by Posner[26] and Bagchi and Ahmadi[2] present improved varieties of this bound. Many other problems have been studied; see Herrmann, Lee, and Snowdon[17] for a survey of dual criteria problems.

2. Notation and an Optimal Property

In this section we introduce our problem and notation, give an example instance of the class scheduling problem under consideration, and present some basic results.

Our class scheduling problem is the minimization of the total flowtime of a set of jobs where the jobs have deadlines on their completion. The problem can be formulated with the following notation:

J_j	Job j, $j = 1, \ldots, n$
p_j	processing time of J_j
D_j	deadline of J_j
G_i	job class i, $i = 1, \ldots, m$

n_i	number of jobs in G_i
s_{0i}	time of initial setup if first job is in G_i
s_{ki}	time of setup between jobs in G_k and G_i
C_j	completion of J_j
ΣC_j	total flowtime.

The problem is to find a sequence that minimizes the total flowtime (ΣC_j) subject to the deadline constraints ($C_j \leqslant D_j$ for all J_j). We name this problem the *Constrained Flowtime with Setups* problem (*CFTS*). Since job preemption or inserted idle time leads to a non-optimal solution, we will assume that the schedules being considered have neither. Any schedule that is a solution for CFTS will have a number of *batches* or *runs* that are sets of jobs from one class processed consecutively. Before each batch will be a class setup. The problem involves determining the composition and order of batches from different classes.

CFTS is an extension of a one-machine problem studied by Smith.[30] In this problem, which we name the *Constrained Flowtime* problem (*CFT*), there exists no sequence-dependent setup times. An instance that we will use to illustrate our work is described in Example 1.

Example 1. The data in Table I form an instance of a class scheduling problem with five jobs in two job classes. The first three jobs form one class, with the remaining two jobs in the second class. Recall that no setup is required between jobs in the same class. However, a class setup is necessary between jobs of different classes, and this setup is sequence-dependent.

CFTS is an NP-complete problem since finding a feasible schedule for NP-complete.[3] Hence, it is unlikely that any polynomal algorithm to solve the problem exists, and consequently we study the use of heuristics to find good solutions.

We now describe Smith's rule for CFT and an optimal property for CFTS. Our new heuristic extends Smith's rule by taking advantage of the optimal property, which we call Smith's property.

In this and later sections, we will refer to each job that can complete at a given time without violating the job's deadline as being *eligible*. In this problem, we are concerned with deadlines that are constraints on the completion times, and we will create schedules backwards, starting with the last position in the sequence. Thus, we say that a job J_j is eligible at a time t if $D_j \geqslant t$. The job can feasibly complete at this time without violating the deadline constraint.

Table I. Job and Class Data for Example 1

j	1	2	3	4	5
p_j	1	2	2	3	2
D_j	3	16	14	10	18
$G_1 = \{1,2,3\}$	$n_1 = 3$	$s_{01} = s_{21} = 2,\ s_{11} = 0$			
$G_2 = \{4,5\}$	$n_2 = 2$	$s_{02} = 2,\ s_{12} = 1,\ s_{22} = 0$			

Smith's rule for CFT states that if a job is assigned the last position of an optimal schedule, then it must be the longest eligible job.

Algorithm 1. (Smith's rule for CFT). Let $t = p_1 + \cdots + p_n$. Among the jobs that are eligible at time t, that is, $D_j \geqslant t$, choose the job J_j with the longest processing time p_j. Schedule J_j to complete at t, and solve the remaining problem in a similar manner.

The following property extends Smith's rule to each class in CFTS. This property will then be extended to consider all classes in order to generate approximate solutions to CFTS.

Lemma 1. (Smith's property for CFTS). *For each class in an optimal schedule for CFTS, the only job that could be scheduled to complete at a time t is the longest eligible one.*

Proof. It suffices to show that if two jobs J_i and J_j in the same class are both eligible at time t and J_i is longer than J_j ($p_i > p_j$), then scheduling J_j to complete at time t leads to a non-optimal solution. Suppose we do. Then $C_j = t$, and J_i precedes J_j in the schedule formed. Create a new schedule by interchanging the two jobs. Since J_j is moved to the left, it is still feasible, and the new completion time is less than C_i, the old completion time of J_i ($p_i > p_j$). The completion times of any jobs between J_j and J_i are decreased. Meanwhile, J_i completes when J_j did, but this is feasible since $D_i \geqslant t$. We have therefore created a feasible schedule with less total flowtime, and the original schedule cannot be optimal. QED.

3. The Heuristic

Quick methods of finding good solutions are sometimes effective ways to attack difficult problems. In this section we describe a multiple-pass heuristic that extends the idea of Smith's rule. We illustrate how this heuristic works using Example 1.

Our heuristic finds solutions for CFTS by scheduling jobs in the spirit of Smith's rule, working backwards from the end of the schedule. Since the makespan (the maximum completion time) of the optimal solution is not known, the heuristic starts with a trial makespan. After scheduling all of the jobs, we compute the actual makespan (by removing any idle time) and use this makespan as the starting point for another iteration. We continue this process until some limiting makespan is reached. At this point, another pass of the heuristic yields a schedule with the same makespan or a schedule that is infeasible (because some job or setup starts before time zero).

This heuristic constructs schedules that satisfy Smith's property (Lemma 1). While that lemma applies only to jobs in one class, our algorithm extends the idea of longest eligible job by considering all of the job classes. We schedule the longest job with the minimum wasted time. Wasted time is time spent in a setup and idle time.

This (single-pass) Minimum Waste algorithm schedules an eligible job from the same class as the previously-scheduled job if one exists. Else, it selects a job from the class with the smallest setup or selects the job with the latest deadline. It does this by measuring each job's *gap*: the wasted time incurred by selecting that job. Note that if no class setups exist, this algorithm is the same as Smith's rule (Algorithm 1).

Algorithm 2. (Single-pass) Minimum Waste.

Step 0: Given a completion time t, select for the last job the longest J_j eligible at this time, i.e. $D_j \geqslant t$. Schedule this job to end at t, and reduce t by p_j.

Step 1: (a modification to Smith's rule): Suppose that at time t, a job from class G_i starts. Then, for each unscheduled job J_j, define q_j as the gap between the last possible completion time of J_j and t. If J_j is in class G_k, $q_j = \max\{t - D_j, s_{ki}\}$ (see Remarks below for an explanation of this definition). Let $q = \min\{q_j$ over unscheduled $J_j\}$. Select the longest job J_j with $q_j = q$ and schedule this job to end at $t - q$. Any necessary setup s_{ki} can begin at $t - q$. Reduce t by q and p_j.

Step 2: If there remain unscheduled jobs, return to Step 1.

Step 3: There are no more unscheduled jobs. If the first job in the schedule is in class G_i, a setup of length s_{0i} must end at t. Reduce t by this amount.

Step 4: If $t < 0$, the schedule created is infeasible. Else, compute the actual makespan of the jobs and setups scheduled.

Remarks. In order to motivate the definition of q_j, the gap, let us note that the setup after J_j is s_{ki}, so the job may not complete after $t - s_{ki}$. However, if the deadline $D_j < t - s_{ki}$, the gap will be $q_j = t - D_j$, which is greater than s_{ki}. This gap thus includes the setup and a period of inserted idle time of length $t - D_j - s_{ki}$. Note that if J_j is also in the class G_i, $q_j = 0$ if and only if $D_j \geqslant t$. The algorithm is a type of greedy heuristic, in that it attempts to minimize the setup time or idle time in selecting jobs to be scheduled.

Multiple-Pass Minimum Waste Heuristic. To find a good solution for CFTS, we can use the following procedure that makes use of the single-pass Minimum Waste algorithm.

Step A: Let $t' = \max\{D_j: j = 1, \dots, n\}$.

Step B: Let $t = t'$.

Step C: Perform one pass of the Minimum Waste algorithm (Algorithm 2) with completion time t. This creates a new schedule.

Step D: Let t' be the sum of processing times and setup times of this schedule. If $t' < t$ and the new schedule is feasible, go to Step B. (The smaller makespan may yield another new schedule.)

Step E: If the new schedule was infeasible or $t' = t$, take the last feasible schedule created and remove the inserted idle time, starting all jobs as soon as possible. This schedule is the result of the heuristic. If an infeasible schedule was created on the first pass, then take the sequence of jobs from the schedule and process the jobs in this order, starting at time zero. This will yield a schedule with some violated deadline constraints.

Because the problem of finding a feasible schedule is NP-complete, a single pass of the Minimum Waste algorithm is not guaranteed to find one. Still, as we shall see, it is usually able to find a feasible schedule if one exists. If a feasible schedule exists, it must finish by the maximum deadline, which is the first trial makespan. Initially, the heuristic is concerned with reducing the makespan. Eventually, as the makespan reaches a lower limit, the algorithm concentrates on the flowtime objective through its use of Smith's property to select a job.

Example 2. This example shows the results of applying the Multiple-Pass Minimum Waste heuristic to Example 1. In the first iteration, $t = 18$, the maximum deadline. The first pass of the Minimum Waste algorithm performs the following calculations (see Table II for complete algorithm): at time 18, no jobs have been scheduled, and the only eligible job is J_5. After choosing J_5, t is reduced by $p_5 = 2$ to 16. For J_1, J_3, and J_4, the waste q_j is the gap until the deadline. Thus, $q_1 = 16 - D_1 = 13$, and similarly for the other two jobs. For J_2, however, $D_2 = 16$, and the deadline gap is zero, but because J_2 is in a different class than J_5, $q_2 = s_{12} = 1$. Thus, J_2 has the smallest waste and is scheduled to end at time 15. After five steps, all of the jobs are scheduled (see Figure 1). There are two units of inserted idle time, however, so the actual makespan of the schedule can be reduced to 16.

When the heuristic repeats the algorithm with the new makespan of 16 (see Table III for calculations), jobs J_2 and J_5 are eligible at time 16. These jobs also have the same processing time, but suppose J_2 is chosen. Then, at time 14, J_3 is eligible and has no gap, as it is in the same class as J_2. However, J_5 has a gap $q_5 = s_{12} = 2$. The algorithm continues in this manner, creating the schedule shown in Figure 2.

The Multiple-Pass Minimum Waste Heuristic again repeats the algorithm, which now begins at the reduced makespan of 15, and a similar schedule can be found if J_2 is chosen at time 15. This schedule has no idle time (see Figure 3), the reduced makespan is also 15, and so the heuristic stops.

4. The Genetic Algorithm

4.1. Problem Space

The original work on problem and heuristic spaces is by Storer, Wu, and Vaccari,[32,33] who define some alternative

Table II. Calculations of the First Pass of the Minimum Waste Algorithm (Initial Makespan = 18)

	Waste					Schedule	
Time	J_1	J_2	J_3	J_4	J_5	J_j	C_j
18	15	2	4	8	0	J_5	18
16	13	1	2	6		J_2	15
13	10		0	3		J_3	13
11	8			2		J_4	9
6	3					J_1	3

Scheduled flowtime: 58.
Reduced makespan: 16.

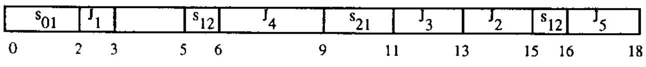

Figure 1. Schedule created on first pass of heuristic.

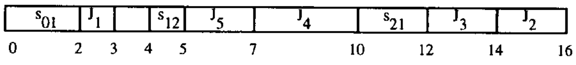

Figure 2. Second schedule.

search spaces for the job shop scheduling problem: the *problem space* and the *heuristic space*. They note that a solution to the problem is the result of applying a heuristic to a problem. Given a problem p, a heuristic h is a function that creates a sequence corresponding to a solution s, i.e., $h(p) = s$. Thus, if one adjusts the heuristic, one creates a different solution. The set of adjusted heuristics is the heuristic space. Likewise, if one adjusts the problem data that are used by the heuristic, one generates a different solution. This set of adjusted problem data is the problem space. The idea is applied to the job shop scheduling problem, and different heuristic searches over the spaces are performed, including hill climbing, genetic algorithms, simulated annealing, and tabu search.

Our research extends this idea by defining a problem space for the one-machine class scheduling problem. If we adjust the deadlines that are the inputs (along with the other problem data) to a pass of the Minimum Waste algorithm, we will create a different schedule. We will use as a problem space for CFTS these *adjusted deadlines*, and we will use one pass of the Minimum Waste algorithm to create a sequence of jobs using the adjusted deadlines. The feasibility (against the actual deadlines) and total flowtime of the sequence can be evaluated by scheduling the jobs to start at time zero with no inserted idle time. The idea is to force jobs to be done earlier or later by decreasing or increasing the deadlines. Note that every solution for CFTS (including the optimal one) is in the range of h:

Theorem 1. *For each solution to an instance of CFTS, there exists a vector of adjusted deadlines that can be mapped to that solution using one pass of the Minimum Waste algorithm.*

Proof. Suppose that σ is a solution (a feasible schedule with no preemption or inserted idle time) for an instance of CFTS. For each job, consider adjusting the deadline so that it equals the job completion time. Then, if we use one pass of the Minimum Waste algorithm with the adjusted deadlines, the job selected for the last position will be the job with the maximum adjusted deadline. This job is the one with the maximum completion time C_i and thus was the last job in σ. It will be scheduled to complete at its adjusted deadline, which is C_i.

Now we are at the start time t of a job J_i and the job with the smallest gap is the unscheduled job J_j that immediately precedes J_i in σ, since the adjusted deadline is C_j, and $C_j \leqslant t$. Any setup necessary between J_j and J_i is already included in the difference between C_j and t. Thus the gap cannot be larger for this job, and the gap for any other job J_k is larger since $C_k < C_j$. This job will be scheduled to complete at its adjusted deadline, which is C_j. If we continue in this manner, all of the jobs will be sequenced in the same order as they were in σ, and we create the same schedule. QED. ■

The more we adjust the deadlines, the more change we create in the schedule. For instance, consider the following examples of applying one pass of the Minimum Waste algorithm to vectors of adjusted deadlines where we have changed only the second and fifth deadlines (Note that the fourth schedule created is infeasible since J_2 completes at time 18, which is greater than the actual deadline: $D_2 = 16$. The adjusted deadline of 19 was used only to sequence the jobs.):

Heuristic (problem) = solution:
Minimum Waste (3, 6, 14, 10, 20) = [1 2 4 3 5], flowtime 46.
Minimum Waste (3, 16, 14, 10, 18) = [1 4 3 2 5], flowtime 50 (original deadlines).
Minimum Waste (3, 17, 14, 10, 16) = [1 5 4 3 2], flowtime 46.
Minimum Waste (3, 19, 14, 10, 17) = [1 4 3 5 2], infeasible ($C_2 = 18$).

In Figure 4 we show a graph that illustrates how adjusting just two of the five deadlines of Example 1 can create a number of different schedules. The first, third, and fourth deadlines were not adjusted. Each point in the plane (only non-negative deadlines were considered) corresponds to a pair of values for the second and fifth adjusted deadlines. Each point within a region of the plane is mapped by a pass of the Minimum Waste algorithm to the job sequence denoted by the five-digit sequence shown in that region. The dot marks the point that corresponds to the unadjusted deadlines ($D_2 = 16$, $D_5 = 18$). The best sequences achievable by adjusting these deadlines are 12435 and 15432 (total flowtime = 46), and the only other feasible sequences are

Table III. The Second Pass of the Minimum Waste Algorithm (Initial Makespan = 16)

	Waste					Schedule	
Time	J_1	J_2	J_3	J_4	J_5	J_j	C_j
16	13	0	2	6	0	J_2	16
14	11		0	4	2	J_3	14
12	9			2	2	J_4	10
7	4				0	J_5	7
5	2					J_1	3

Scheduled flowtime: 50.
Reduced makespan: 15.

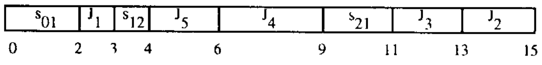

Figure 3. Third and final schedule.

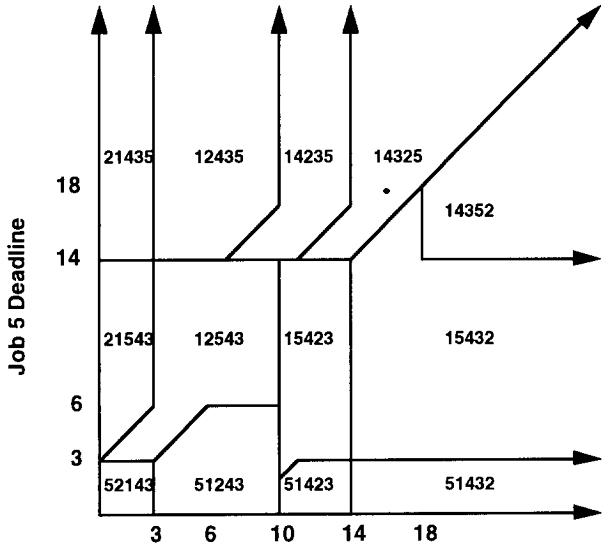

Figure 4. Graph of adjusted deadlines and schedules created.

14235 and 14325 (total flowtime = 50). The optimal solution (which cannot be found by adjusting only the second and fifth deadlines) is 13452, with total flowtime = 43.

Since the actual problem space consists of all of the problem data and there are numerous heuristics that can be used, we can investigate a couple of other spaces and heuristics that might be useful. Our first search was to adjust the job processing times and to use Shortest Processing Time (SPT) rule. However, it is difficult to find feasible solutions since SPT ignores the deadline constraints entirely. We also tried the using the Earliest Due Date (EDD) rule while adjusting the deadlines, but EDD does not give enough attention to the flowtime objective. Sequencing by either SPT or EDD is a fairly dumb heuristic, since neither makes use of the other available information. The Minimum Waste algorithm, however, considers due dates, processing times, and setups, and using it improves our searches. In addition, while it would be possible to use the Minimum Waste algorithm while adjusting the processing or setup times, the effect of these variables on the sequencing of jobs is more indirect than that of the deadlines.

4.2. Genetic Algorithms

A genetic algorithm is a heuristic search, a term that also refers to *smart-and-lucky* searches like tabu search and simulated annealing. These searches are smart enough to avoid becoming trapped in local optima and lucky enough to generally find good solutions. Genetic algorithms use a population of points in the effort to find the optimal solution. Each individual in the population is a string of genes, where each gene describes some feature of the solution. Genetic algorithms mimic the processes of natural evolution, including reproduction and mutation. The most powerful operator of a genetic algorithm is the crossover operator: the recombination of the genes of two parents to create two offspring. This crossover allows the offspring to acquire the good characteristics of different points in the search space.

Most genetic algorithms perform the following steps, stopping when a fixed number of offspring has been created:

Step 0: Form an initial population.
Step 1: Evaluate the individuals in the population.
Step 2: Select individuals to become parents with probability based on their fitness.
Step 3: For each pair of parents, perform a crossover to form two offspring. Mutate each offspring with some small probability.
Step 4: Place the offspring into the population. Return to Step 1.

Holland,[19], Davis,[7,8] and Goldberg[13] provide good descriptions of genetic algorithms. Davis,[6] Oliver, Smith, and Holland,[25] Liepins and Hilliard,[21] Fox and McMahon,[11] Gupta, Gupta, and Kumar,[16] and Storer, Wu, and Vaccari[32] describe the use of genetic algorithms on traveling salesman problems and scheduling problems. Other work on genetic algorithms and scheduling problems are reported in [4, 24, 31, 34, 35]. See Goldberg[13] for additional references on genetic algorithm applications.

4.3. Genetic Algorithm for CFTS

Genetic algorithms are heuristic searches that use a population of points in the effort to find the optimal solution. The stronger members of the population survive, mate and produce offspring that may undergo a mutation. These offspring form a new generation. Genetic algorithms have been used on sequencing problems before, although they cannot use natural crossover techniques in searching the solution space. The advantage of the problem space is that the genetic algorithm can use standard techniques to create offspring.

Our genetic algorithm searches the space of adjusted deadlines. We use a binary coding for the adjusted deadlines and a single pass of the Minimum Waste algorithm as the heuristic. For this genetic algorithm, we use many of the ideas presented in Davis.[8]

In the problem space, each point is a vector of integers that are deadlines used as input for one pass of the Minimum Waste algorithm. We will use a binary representation of the points in problem space. In the population of the genetic algorithm, each individual is a string of bits. Each successive six-bit substring represents a deadline for a specific job. The integer decoded from this binary number ranges from zero to 63 and linearly maps to a real number in the range from zero to the maximum deadline in the given problem data. This discretization reduces the problem space but still allows the deadlines to vary significantly with respect to each other.

The adjusted deadlines are used as input to a single pass of the Minimum Wage algorithm, which outputs a sequence of jobs. The algorithm uses the largest of the adjusted deadlines as the initial makespan and schedules the jobs accordingly, using the actual job processing and class setup times where necessary but using the adjusted deadlines to determine when a job is eligible. If necessary, the algorithm can start jobs before time zero.

Using the actual problem data and the sequence of jobs output from one application of the Minimum Waste Algorithm, we can create a schedule of jobs that starts at time zero and has no inserted idle time. We evaluate the original string of bits by computing the feasibility and the total flowtime of this schedule.

Since we cannot guarantee that the algorithm will produce a schedule with no tardy jobs, we use a penalty function to make undesirable those individuals in the population that yielded infeasible schedules (with respect to the actual deadlines). This penalty function is $F = \Sigma T_j^2$, where $T_j = \max\{0, C_j - D_j\}$, Our objective function f is defined as $f = \Sigma C_j + rF$.

In order to encourage solutions with good total flowtime (regardless of feasibility) at the beginning of the search and to encourage feasibility near the end of the search, we start the search with the constant r small and increase it periodically.

The initial population includes one individual (the dummy, or seed) that we create by dividing each actual deadline by the maximum deadline, multiplying by 63, rounding down to the nearest integer, and converting this integer (which is in the range 0 to 63) to its binary representation. The remaining individuals in the initial population are constructed by mutating the bits in the initial (dummy) chromosome. The mutation rate is set at fifty percent (0.5). Note that using an initial mutation rate of 0.5 is equivalent to choosing a random chromosome from the entire search space.

Let us illustrate this procedure using Example 1, the problem we introduced earlier (see Table I for problem data). Also, let us define $[x]$ as the greatest integer less than or equal to x.

Mapping the deadlines to the bit strings yields the dummy, shown in Table IV. Let us create another member on the initial population. If two bits, the fourth of the fourth substring and the first of the last substring, are flipped in the mutation, we have the point in problem space shown in Table V. Performing one pass of the Minimum Waste algorithm on the new deadlines (see Table VI) yields the sequence [1 5 4 3 2], from which we create the feasible schedule shown in Figure 5, with a makespan of 15 and a total flowtime of 46.

After some experimentation we decided to use a steady-state genetic algorithm that created offspring by repeatedly selecting from a set of four genetic operators: one-point crossover, uniform crossover, and two types of mutation. *Small mutation* used a low probability (two percent per bit) of flipping a bit in the string; *large mutation* used a higher probability (fifty percent). The search used tournament selection for selecting the parents necessary for the crossover or mutation, and duplicate bit strings were not allowed in the population. Tuning was performed in order to determine some good settings for various algorithm parameters.

These parameters included the population size, the mutation rate, the operator fitness, and the rate of change of the penalty coefficient r. The rate of increase in the penalty coefficient affected the solution quality slightly. The mutation rate, population size, and relative probabilities of operator selection affected solution quality more significantly, with a smaller mutation rate (two percent, as mentioned earlier), smaller population sizes, and a higher fitness for mutation yielding better solutions. These factors imply that the search can easily find good neighborhoods (especially since a point corresponding to the original problem data is included in the initial population) but needs to spend time hunting for a better solution. Thus, a search that incorporates some kind of local search at the end of the genetic algorithm may be useful.

Table IV. Bit Representation of the Dummy

D_j	3	16	14	10	18
$[63\ D_j/18]$	10	56	49	35	63
Bits	001010	111000	110001	100011	111111

Table V. A Point in the Problem Space

Bits	001010	111000	110001	100111	011111
Integer	10	56	49	39	31
D_j	2.85	16	14	11.14	8.85

5. Empirical Testing

5.1. Problem Generation

In order to test the heuristics and the genetic algorithm described above, it is necessary to create a testbed of problems. We describe in this section how we can create problems that have at least one feasible solution and problems where finding a feasible solution is more difficult.

The problems in the first problem set have 30 jobs in four classes, with random processing times in the range [1, 20] and sequence-dependent class setups in the range [0, 5]. For this set, we want to determine random deadlines in order to insure that some feasible schedule did exist.

Table VI. Application of the Minimum Waste Algorithm

	Waste					Schedule	
Time	J_1	J_2	J_3	J_4	J_5	J_j	C_j
16	13.15	0	2	4.86	7.15	J_2	16
14	11.15		0	2.86	5.15	J_3	14
12	9.15			2	3.15	J_4	10
7	4.15				0	J_5	7
5	2.15					J_1	2.85

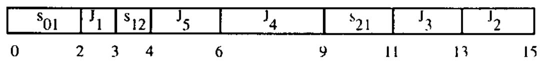

Figure 5. Schedule corresponding to new bit string.

We use the following procedure: after computing the random class setup times, each job is given a random processing time, and an initial completion time is computed by scheduling it after all previously-constructed jobs. This first-generated-first-served schedule yields a makespan that becomes an upper bound for the deadlines, and each job is given a deadline determined by sampling a random variable uniformly distributed between the job completion time (in this schedule) and the makespan, i.e. the interval $[C_j, C_{max}]$. Thus, the initial sequence is a feasible solution.

In order to determine the performance of the genetic algorithm on minimizing the flowtime when feasible solutions are harder to locate, a number of additional problem sets are created. In addition to the problem set described earlier, which includes problems that were known to have a feasible solution, we generate 30-job and 50-job problems with tighter deadlines. The 30-job problems have four job classes and the 50-job problems ten job classes. Tighter deadlines are achieved by extending the range of values that a random deadline could take. Let us define a value k that can range from zero to one. The deadline for J_j is taken from the interval $[kC_j, C_{max}]$, where the C_j and C_{max} are from the original generated schedule. If $k = 1$, this plan is the same as the original one, and the generated problem is guaranteed to have a feasible schedule. If $k = 0$, all of the deadlines vary equally, and there may exist no feasible schedule. As k decreases from one to zero, the problems we generate have a higher probability of having fewer feasible schedules. We generated problems with $k = 0$, 0.2, and 1. Since the Multiple-Pass Minimum Waste Heuristic cannot find feasible solutions for some of these problems, we will see if the genetic algorithm can find solutions that are feasible.

5.2. Results

In this section we discuss the results of our experiments with the solution procedures on the generated problem sets. Since the optimal solutions are not known and no good lower bound can be determined, we measure the performance of the solution procedures relative to each other. Each procedure was run once on each of the problems in the problem sets. The procedures include the Multiple-Pass Minimum Waste Heuristic and the problem space genetic algorithm.

Since no algorithms for this problem have been previously introduced, we implemented for comparison purposes a version of the heuristic that Ahn and Hyun[1] use to reduce the total flowtime in unconstrained class scheduling problems. They proposed an iterative heuristic that starts with an initial feasible sequence where the jobs in each class are in SPT order (since they are not concerned with deadlines) and applies both a forward and backward procedure to it, repeating the steps until no strict improvement is found. Each of the forward and backward procedures interchanges different subschedules where the second subschedule consists of jobs from one class and the first subschedule has no jobs from this class. If the interchange reduces the total flowtime, the subschedules are switched; this maintains the class SPT property.

Our version of this algorithm, called the Modified Ahn & Hyun heuristic, uses one pass of the Minimum Waste algorithm to form the initial schedule. In addition, a potential swap of two subschedules is performed only if the swap reduces the total flowtime and maintains deadline feasibility.

The results (see Table VII) show that the genetic algorithm can find solutions that are much better than those found by the Multiple-Pass Minimum Waste Heuristic and are slightly better than those that Modified Ahn & Hyun heuristic produces. The genetic algorithm needs more time to find good solutions on the larger problems, although additional tuning may help improve the performance of the search.

The searches for the 30-job problems were for 2000 iterations using the following parameter settings: population size of 10, all operator fitnesses equal, increase of 50 in r every 10 individuals. The searches for the 50-job problems were for 3000 iterations using the same population size, no large mutations, and an increase of 50 in r every 50 individuals.

We observed that if the Multiple-Pass Minimum Waste Heuristic is unable to generate a feasible solution, the Modified Ahn & Hyun heuristic and the problem space genetic algorithm cannot locate a feasible solution. Thus our results are reported only for those problems where feasible schedules were created.

Table VII. Total Flowtime Performance of Heuristics on Problems Where a Feasible Was Found

Problem Set	Problems	Jobs	k	Performance[a] of Heuristics Genetic Algorithm	Modified Ahn & Huun
Newprob4	10	30	1	0.8562	0.9130
Prob2a	4	30	0.2	0.9099	0.9346
Prob50z	10	50	1	0.8739	0.8755
Prob2c	8	50	0.2	0.8796	0.8914

[a]Performance is average ratio to solution found by the Multiple-Pass Minimum Waste Heuristic.

6. Conclusions and Future Directions

This paper has two contributions: it introduces an extended heuristic for the dual criteria class scheduling problem that we call CFTS, and it describes a problem space genetic algorithm used to find good solutions. The problem is to minimize the total flowtime subject to deadline constraints. In this paper we present a multiple-pass heuristic for finding good solutions and discuss problem space and the genetic algorithm. Finally, we describe our experimental results, in which we compared the genetic algorithm to some heuristic approaches. From these results we make the following conclusions:

The Multiple-Pass Minimum Waste heuristic performs well at minimizing the total flowtime of CFTS. Though not an exact procedure, it is usually able to find feasible, high-quality solutions.

A genetic algorithm that searches a problem space of the Minimum Waste algorithm for CFTS can find solutions with lower total flowtime. This genetic algorithm includes a penalty function for infeasible points that increases the cost of tardiness as the search progresses.

Acknowledgments

This material is based upon work supported under a National Science Foundation Graduate Fellowship and work funded by Harris Semiconductor and National Science Foundation Grant DDM-9201627. We also wish to thank the referees and the area editor, Darrell Whitley, whose constructive comments helped us significantly improve the paper.

References

1. B.-H. AHN and J.-. HYUN, 1990. Single Facility Multi-Class Job Scheduling, *Computers and Operatıons Research 17:3*, 265–272.
2. U. BAGCHI and R.H. AHMADI, 1987. An Improved Lower Bound for Minimizing Weighted Completion Times with Deadlines, *Operations Research 35*, 311–313.
3. J. BRUNO and P. DOWNEY, 1978. Complexity of Task Sequencing with Deadlines, Set-Up Times and Changeover Costs, *SIAM Journal of Computing 7:4*, 393–404.
4. G. CLEVELAND and S. SMITH, 1989. Using Genetic Algorıthms to Schedule Flow Shop Releases, in *Proceedings of the Third International Conference on Genetıc Algorithms*, Morgan Kaufmann, San Mateo, California.
5. E.G. COFFMAN, A. NOZARI and M. YANNAKAKIS, 1989. Optimal Scheduling or Products with Two Subassemblies on a Single Machine, *Operatıons Research 37:3*, 426–436.
6. L. DAVIS, 1985. Job Shıp Scheduling with Genetic Algorithms, in *Proceedıngs of an International Conference on Genetic Algorithms and theır Applıcatıons*, J. Grefenstette (ed.), Lawrence Erlbaum Associates, Hillsdale, N.J.
7. L. DAVIS (ed.), 1987. *Genetic Algorithms and Simulated Annealing*, Pitman Publishing, London.
8. L. DAVIS (ed.), 1991. *Handbook of Genetıc Algorıthms*, Van Nostrand Reinhold, New York.
9. G. DOBSON, U.S. KARMARKAR and J.L. RUMMEL, 1987. Batching to Minimize Flow Times on One Machine, *Management Science 33:6*, 784–799.
10. G. DOBSON, U.S. KARMARKAR and J.L. RUMMELL, 1989. Batching to Minimize Flow Times on Parallel Heterogeneous Machines, *Management Science 35:5*, 607–613.
11. B.R. FOX and M.B. MCMAHON, 1990. *Genetic Operators for Sequencıng Problems*, Planning and Scheduling Group, McDonnell Douglas Space Systems, Houston, Texas.
12. M.R. GAREY and D.S. JOHNSON, 1979. *Computers and Intractabılity: a Guide to the Theory of NP-Completeness*, Freeman, San Francisco.
13. D.E. GOLDBERG, 1989. *Genetic Algorithms ın Search, Optimizatıon, and Machine Learning*, Addison-Wesley, Reading, Massachusetts.
14. J.N.D. GUPTA, 1984. Optimal Schedules for Single Facility with Classes, *Computers and Operations Research 11*, 409–413.
15. J.N.D. GUPTA, 1988. Single Facility Scheduling with Multiple Job Classes, *European Journal of Operatıonal Reserach 8*, 42–45.
16. M.C. GUPTA, Y.P. GUPTA and A. KUMAR, 1993. Minimizing Flow Time Variance in a Single Machine System Using Genetic Algorithms, *European Journal of Operational Research 70:3*, 289.
17 J.W. HERRMANN, C.-Y. LEE and J.L. SNOWDON, 1993. A Classification of Static Scheduling Problems, in *Complexıty ın Numerical Optimızation*, P.M. Pardalos (ed.), World Scientific, River Edge, N.J., pp. 203–253.
18. J.C. HO, 1992. Minimızing the Number of Tardy Jobs with Two Jobs Classes, Division of Business, Northeast Missouri State University.
19. J.H. HOLLAND, 1975. *Adaptation in Natural and Artificıal Systems*, University of Michigan Press.
20. J.K. LENSTRA, A.H.G. RINNOOY KAN and P. BRUCKER, 1977. Complexity of Machine Scheduling Problems, *Annals of Discrete Mathematıcs 1*, 343–362.
21. G.E. LIEPINS and M.R. HILLIARD, 1989. Genetic Algorithms: Foundations and Applications, *Annals of Operations Research 21*, 31–58.
22. A.J. MASON and E.J. ANDERSON, 1991. Mınimizing Flow Time on a Single Machine with Job Classes and Setup Times, *Naval Research Logistics 38*, 333–350.
23. C.L. MONMA and C.N. POTTS, 1989. On the Complexity of Schedulıng With Batch Setup Times, *Operations Research 37:5*, 798–804.
24. R. NAKANO and T. YAMADA, 1991. Conventional Genetic Algorithms for Job Shop Problems, in *Proceedıngs of the Fourth Internatıonal Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, San Mateo, California.
25. I.M. OLIVER, D.J. SMITH and J.R.C. HOLLAND, 1987. A Study of Perturbation Crossover Operators on the Traveling Salesman Problem, in *Genetic Algorithms and their Applications: Proceedings of the Second Internatıonal Conference on Genetic Algorithms*, J. Grefenstette (ed.), Erlbaum Associates, Hillsdale, N.J.
26. M.E. POSNER, 1985. Minimizing Weighted Completion Times with Deadlines, *Operations Research* 33, 562–574.
27. C.N. POTTS, 1991. Scheduling Two Job Classes on a Single Machine, *Computers and Operations Research 18:5*, 411–415.
28. C.N. POTTS and L.N. VAN WASSENHOVE, 1983. An Algorithm for Single Machine Sequencing with Deadlines to Minimize Total Weighted Completion Time, *European Journal of Operational Research 12*, 379–387.
29. V.K. SAHNEY, 1972. Single-Server, Two-Machine Sequencıng with Switching Time, *Operations Research 20:1*, 24–36.
30. W.B. SMITH, 1956. Various Optimızers for Single-Stage Production, *Naval Research Logistics Quarterly 3*, 59–66.
31. T. STARKWEATHER, S. MCDANIELS, K. MATHIAS, C. WHITLEY and D. WHITLEY, 1991. A Comparison of Genetic Sequencing Operators, in *Proceedıngs of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, San Mateo, California.
32. R.H. STORER, S.Y.D. WU and R. VACCARI, 1990. Local Search in Problem and Heuristic Space for Job Shop Scheduling, Department of Industrial Engineering, Lehigh University.

33. R.H. STORER, S.Y.D. WU and R. VACCARI, 1992. New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling, *Management Science 38:10*, 1495–1509.
34. G. SYSWEDA, 1991. Schedule Optimization Using Genetic Algorithms, in *Handbook of Genetic Algorithms*, L. Davis (ed.), Van Nostrand Reinhold, New York.
35. D. WHITLEY, T. STARKWEATHER, and D. SHANER, 1991. The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Optimization, in *Handbook of Genetic Algorithms*, L. Davis (ed.), Van Nostrand Reinhold, New York.
36. D.L. WOODRUFF and M.L. SPEARMAN, 1992. Sequencing and Batching for Two Classes of Jobs with Deadlines and Setup Times, *Production and Operations Management 1:1*, 87–102.