Solving a Real-World Problem Using an Evolving Heuristically Driven Schedule Builder

Emma Hart
Department of Artificial Intelligence
University of Edinburgh
5 Forrest Hill
Edinburgh EH1 2QL
emmah@dai.ed.ac.uk

Peter Ross
Department of Artificial Intelligence
University of Edinburgh
5 Forrest Hill
Edinburgh EH1 2QL
peter@dai.ed.ac.uk

Jeremy Nelson
Department of Artificial Intelligence
University of Edinburgh
5 Forrest Hill
Edinburgh EH1 2QL
jeremyn@dai.ed.ac.uk

Abstract

This work addresses the real-life scheduling problem of a Scottish company that must produce daily schedules for the catching and transportation of large numbers of live chickens. The problem is complex and highly constrained. We show that it can be successfully solved by division into two subproblems and solving each using a separate genetic algorithm (GA). We address the problem of whether this produces locally optimal solutions and how to overcome this. We extend the traditional approach of evolving a "permutation + schedule builder" by concentrating on evolving the schedule builder itself. This results in a unique schedule builder being built for each daily scheduling problem, each individually tailored to deal with the particular features of that problem. This results in a robust, fast, and flexible system that can cope with most of the circumstances imaginable at the factory. We also compare the performance of a GA approach to several other evolutionary methods and show that population-based methods are superior to both hill-climbing and simulated annealing in the quality of solutions produced. Population-based methods also have the distinct advantage of producing multiple, equally fit solutions, which is of particular importance when considering the practical aspects of the problem.

Keywords

Evolving schedule builder, heuristics, constraints.

1. Introduction

Studies of the application of evolutionary methods to a variety of benchmark scheduling problems have produced solid evidence that such techniques can be fast and efficient ways of solving many types of scheduling problems, for instance (Lin, Goodman, & Punch, 1997; Cleveland & Smith, 1989; Fang, Ross, & Corne, 1993; Mott, 1990).

However, when attempting to scale these techniques up to tackle some of the real problems faced in industry, we often discover that a number of additional factors and constraints must be incorporated to successfully model the real world, which can complicate the issue considerably. Here we study a very highly constrained problem for a chicken-processing factory that must schedule the arrival times of lorry loads of birds to two factories on a daily basis, scheduling teams of chicken-catching squads, lorries, and drivers in the process. The problem is described in detail in Section 2. The goal is to produce an *automated* scheduling system that produces practical schedules quickly, similar to those currently produced by hand at the factory. Against the grain of most scheduling work, we are not interested in *optimizing* any particular objective while producing the schedules, but in producing a fast and robust system that can produce high-quality *practical* schedules that satisfy a large number of constraints. In trying to formulate a definition of "practical" we are required to add many more constraints to the system, significantly increasing the complexity of the problem. We believe that this problem is common to a large number of industrial scheduling problems, and it is this factor that may hinder the success of many of the techniques successfully proven on artificial problems when applying them to real problems.

An automated scheduling system can be exploited in many ways to directly and indirectly provide benefits to the company. For example, a system that closely replicates current working practices can quickly gain the confidence of its users and perhaps then be used as an exploratory tool to investigate the effects of making changes to current practices. Changing established practice may not, in many cases, be cost-effective in the short-term, and hence a company may be reluctant to implement change. In this case, the automated system is intended to bring an immediate, direct cost benefit to the company by freeing up human resources to be put to better use elsewhere.

The following section describes the scheduling task in more detail. Sections 3, 4, and 5 describe the evolutionary approach we have adopted and the reasoning behind it. This is followed by a set of results found using eight test cases supplied by the factory.

2. Problem Description

The company catches up to 1.3 million live birds a week at farms spread across Scotland and processes them at two factories. The aim is to schedule the catching and delivery of birds using a fixed resource of squads and lorries so as to keep the factories supplied with birds at a constant rate. Each day, a set of *orders* must be processed at each of the two factories.

Each factory runs its own fleet of lorries, which are available in two different sizes, holding either 22 modules or 24 modules of birds. Certain types of birds require a large lorry for delivery, whereas others can fit in either size. Although a lorry may deliver loads to either of the factories during the day, it must deliver its last load to its base factory in the evening.

An *order* consists of m modules of birds, of type t, weight w, to be collected from a farm f and delivered to a factory F. The work defined by the orders must be allocated to teams of catching squads. Lorries and drivers then must be scheduled to meet each load at a farm as it is caught and deliver it to a factory. The factories must be continually supplied with birds throughout the day — however, there are strict regulations governing the period of time the live birds can remain outside the factory before being processed; hence the birds must arrive only at the same rate as the factory can process them. Figure 1 illustrates an example of this process for one factory. A squad leaves its base factory at the beginning of a shift, travels

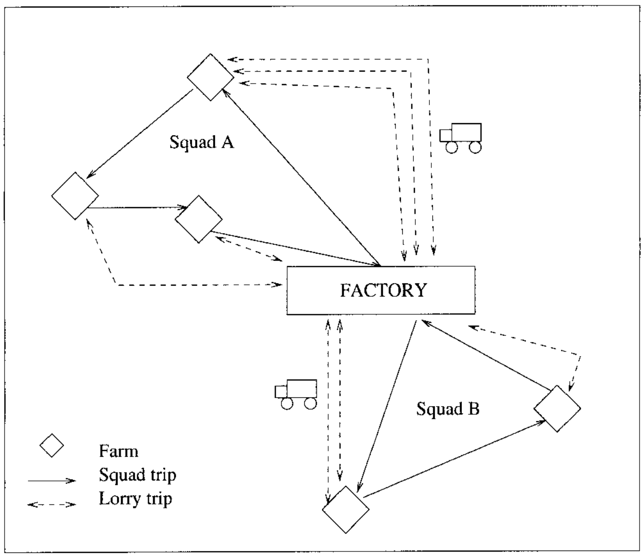

Figure 1. Illustration of squad and lorry schedules for two squads collecting work for a factory.

between a set of farms, catches a variable number of birds at each farm, and returns to its base factory only at the end of the shift.

A *squad* is based at one of the factories and works either part or full time. It rotates between three shifts, classified as *early*, *floating*, or *late*, depending on the time of day the shift begins. There are contractual requirements specifying the minimum and maximum number of modules a squad can catch in one day, and also in one week, and the maximum length of the shifts they can work. Some farms have their own catching squads, and there are restrictions on the order in which squads can visit certain farms due to the prevention of spread of diseases existing at some farms.

Drivers must also be assigned to each lorry trip—a lorry driver can work a maximum of a 13-hour shift, of which only 9 hours can actually be driving. If necessary, contract drivers and/or lorries can be hired to do any extra work, although of course it is preferable to avoid this.

3. Evolving a Scheduler

An often-cited approach to tackling scheduling problems (Syswerda & Palmucci, 1991) is to use a simple chromosome that represents a permutation of tasks to be scheduled, plus a schedule builder that transforms the chromosome into a schedule. This section describes related work that uses this approach, and based on this, discusses some of the factors that must be taken into account when designing both the chromosome and the schedule builder in this particular case.

3.1 The Chromosome Encoding

In general, each chromosome represents a permutation of the order in which tasks to be scheduled are to be considered by the schedule builder (SB) (Bierwith, 1995). For example, in a 5×5 job-shop scheduling problem, each chromosome could simply represent a permutation of the 25 operations that must be scheduled (Mattfeld, 1996).

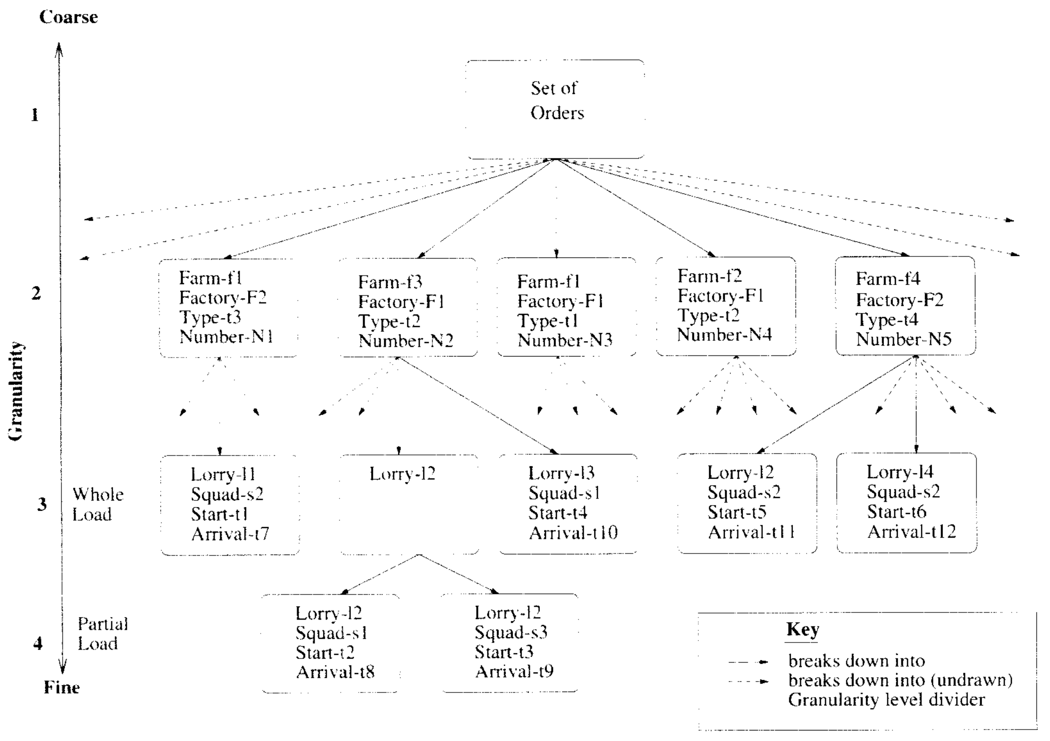
Coarse
Granularity
1
2
3
4
Whole Load
Partial Load
Fine
Set of Orders
Farm-f1 Factory-F2 Type-t3 Number-N1
Farm-f3 Factory-F1 Type-t2 Number-N2
Farm-f1 Factory-F1 Type-t1 Number-N3
Farm-f2 Factory-F1 Type-t2 Number-N4
Farm-f4 Factory-F2 Type-t4 Number-N5
Lorry-l1 Squad-s2 Start-t1 Arrival-t7
Lorry-l2
Lorry-l3 Squad-s1 Start-t4 Arrival-t10
Lorry-l2 Squad-s2 Start-t5 Arrival-t11
Lorry-l4 Squad-s2 Start-t6 Arrival-t12
Lorry-l2 Squad-s1 Start-t2 Arrival-t8
Lorry-l2 Squad-s3 Start-t3 Arrival-t9
Key
breaks down into
breaks down into (undrawn)
Granularity level divider

Figure 2. Levels of order: split granularity.

In the problem presented here, however, there is no clear notion of an *operation* or *task*. At the highest level, we have a collection of orders to schedule (e.g., Fig. 2, levels 1 and 2). An order itself cannot be scheduled in a single step: Each order represents several arrivals at a factory that may be caught by several different squads (e.g., Fig. 2, levels 3 and 4); therefore, each order must be broken down into schedulable tasks. Evolving permutations of *orders* to present to the SB requires that the SB itself must split each whole order into schedulable tasks. These tasks may themselves require ordering to produce good schedules, and hence the SB must do too much work — evolution is taking place at too high a level.

The most obvious candidate for a *task* is a complete lorry load of birds (i.e., 22 or 24 modules), depending on bird type and lorry size (e.g., Fig. 2, level 3). However, scheduling tasks of this size may preclude some of the best solutions. For example, a part-time squad has a maximum load of 64 modules. Assigning two tasks to a part-time squad will result in an underworked squad, catching with 44 or 48 modules, whereas assigning three tasks to the squad will cause it to become overworked. If the task size is reduced to try to resolve this, for example, to half a lorry load (level 4 in Fig. 2), the size of the search space dramatically increases; moreover, the possibility of producing impractical schedules increases also. As up to 60 lorry loads per day may arrive at the factories, even scheduling complete lorry loads results in 60! possible permutations. Many of these permutations are likely to lead to highly impractical schedules, for instance, with squads collecting small loads of birds at many different farms, rather than collecting large loads at few farms. This can be counteracted in a fitness function by introducing penalties to penalize such schedules, but given an already highly constrained problem, this is not desirable (see Section 6.1.2 for a full discussion of the effects of the choice of task size).

We resolve this problem by using a genetic algorithm (GA) that encodes, for every order, a choice of heuristic that should be used to split that order into schedulable tasks. The reasoning behind this is that, wherever possible, the search space is reduced by using a coarse-grained split, with a corresponding increase in the practicality of the schedules. By allowing the GA to increase the size of the search space if necessary, we hope to find an acceptable trade-off between the optimality of the schedule (in terms of satisfying constraints) and the practicality of the schedule.

3.2 The SB

The function of the SB is to transform the chromosome into an actual schedule and is a crucial step in the performance of the whole system. The SB itself can be used to enforce constraints, as well as to produce the actual schedules from the chromosome. Bagchi, Uckun, Miyabe, and Kawamura (1991) note, however, that the schedules produced by an SB are optimal locally (i.e., the schedules produced are constrained by the plans in the SB). More recently, other work has investigated the effect of using some degree of choice within the SB itself to counteract this problem of local optimality. Langdon (1996) used a genetic programming approach to evolve the heuristic to use for a fixed order of tasks. Shaw and Fleming (1997) found that implementing a degree of flexibility into the SB, producing a compromise between a rigid SB and one that assigns tasks randomly, can improve results when trying to optimize costs of schedules in production scheduling.

Other people have reported successful use of a GA to evolve heuristic choice in scheduling applications. For example, Dorndorf and Pesch (1995) describe a "priority-rule based genetic algorithm," where the *i*th gene in a chromosome denotes a dispatch rule used to resolve conflicts in the *i*th iteration of a scheduling algorithm. In a similar vein, Fang, Ross, and Corne (1993) and Fang (1993) investigated "evolving heuristic choice" for open-shop problems, in which they evolve the choice of heuristic to decide which operation of the job defined in the permutation should be scheduled next, with promising results. Norenkov and Goodman (1997) extend this approach in an algorithm called the "heuristic combination method," in which a GA is used to optimize the choice and sequence of application of a set of heuristic rules for schedule synthesis.

In this case, as we aim to produce a system that replicates a human scheduler, it is obviously vital to include as much of his/her knowledge as possible in the SB. The potential problem of producing suboptimal schedules by exactly reflecting factory decision making is irrelevant in this case, as we wish to produce schedules that look like those currently made by hand. The crux of the problem lies with accurately formulating the heuristics used by the human. Often, the reasoning behind why a particular decision was taken cannot be easily encapsulated — it comes with years of experience and an "intuition" that cannot be expressed on paper. We resolve this problem by expressing as much information as we can glean from current factory practice as a set of rules and by exploiting the searching capabilities of the GA to combine these rules into a sensible SB that fits the features of the problem being considered. A sequence of heuristic choices is evolved that dictates which heuristic to use to place a task into the partially built schedule.

In summary, we shall use a GA to evolve a *strategy* for producing schedules, rather than the schedule itself. We circumvent the inherent problems of the "permutation+SB" approach by evolving the optimal method of dividing the high-level work into schedulable tasks and then evolving a unique SB for each test case. This is described in detail in Sections 5 and 6.

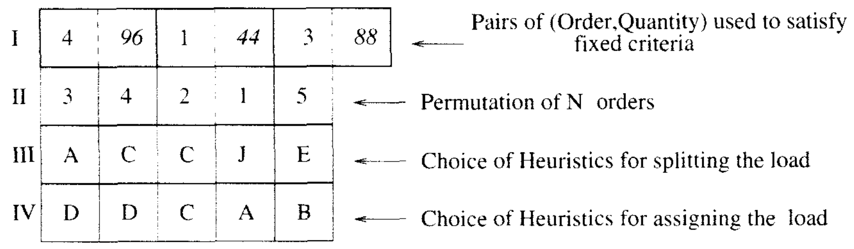
I
4
96
1
44
3
88
Pairs of (Order,Quantity) used to satisfy fixed criteria
II
3
4
2
1
5
Permutation of N orders
III
A
C
C
J
E
Choice of Heuristics for splitting the load
IV
D
D
C
A
B
Choice of Heuristics for assigning the load

Figure 3. The chromosome representation for *GA*(*Assign*).

4. Dividing Into Subtasks

Trying to evolve an SB that assigns tasks to squads and simultaneously schedules lorries, drivers, and factory arrivals in one step is extremely complex. We can simplify the problem somewhat by breaking it down into two stages:

1. Split each order into tasks, according to a specific heuristic for each order, and assign those tasks to squads in a manner that minimizes the number of constraint violations, using the evolved heuristic choice—*GA*(*Assign*).
2. Assign start times to each of those squads in a manner that optimizes the arrival times of loads to the factory and minimizes the number of lorries required—*GA*(*Timer*).

For each stage, we run a separate GA. The best assignment found by *GA*(*Assign*) is used as input to *GA*(*Timer*), which evolves a start time for each squad. In the general case, hierarchical decomposition of a problem in this manner runs the risk of producing suboptimal solutions, where the performance of the second stage, in this case *GA*(*Timer*), is constrained by the output of the preceding stage [i.e., *GA*(*Assign*)]. Ryu, Hwang, Choi, and Cho (1997) tackle this problem by using an iterative process in which the cycle of steps is repeated until no further improvement in performance can be obtained. We investigate thoroughly the extent to which *GA*(*Timer*) is actually constrained by *GA*(*Assign*) and whether such an iterative approach is necessary or beneficial in this particular instance.

5. *GA*(*Assign*)

This section describes the chromosome representation adopted and explains in detail the steps taken to deal with the constraints associated with the problem. Although the final representation appears highly problem specific, the principles underlying its design have much more general applicability to other problems. In particular, the methodology used to handle constraints (described in Section 5.2) is extendible to other highly constrained problems, and the manner in which domain knowledge is dealt with (see Section 5.3), is transferable to other real-world applications. Figure 3 shows the structure used to indirectly represent how the orders should be split into tasks and assigned to squads for a single factory. Each chromosome consists of two identical structures, one for each factory. A detailed explanation of each part of the representation is given in the following sections.

5.1 Constraints

GAs that are based on constraint satisfaction or penalty functions can often be hard to get to work as desired (Richardson, Palmer, Leipin, & Hilliard, 1989; Michalewicz & Janikow, 1991). Michalewicz and Janikow (1991) provide a mechanism for constraint satisfaction that does not allow infeasible chromosomes to exist in the population. In this case this mechanism is neither necessary nor desirable: In this problem generation of feasible solutions is not difficult (see Section 6.2; Section 6.1.1 and Table 4; and Table 12); and as Richardson et al. (1989) point out

> The foundations of GA theory, however, say that GAs optimise by combining partial information from all the population. Therefore, the infeasible solutions should provide information and not just be thrown away.

Richardson et al. (1989) provide a generic technique for constructing a penalty function. This method cannot be applied to this problem as it requires either a priori knowledge of the search space or at least certain features of it to be calculable (Michalewicz & Janikow, 1991). We approached the problem by simplifying the constraints and manually assigning penalty function weights.

5.2 Handling Constraints

We deal with the potentially large number of constraints in three ways: handle the *unavoidable* criteria; simplify by removal of constraints; and incorporate the hard and soft constraints into a penalty function.

5.2.1 Dealing With *Unavoidables* Several of the constraints concern events that must happen in the schedule; for instance, "a full-time squad must deliver a *large* load of roaster birds to factory A to open the factory." We can guarantee that such constraints are met by encoding on the chromosome the manner in which the constraint should be satisfied. Thus, in section I, pairs of genes specify an *order* and *quantity* for satisfying each of the *unavoidable* criteria. Thus, we retain some flexibility by allowing the GA to evolve the *method* by which each criterion is specified. In Figure 3, the first pair of genes (4, k) in section I specifies that criterion (1) is met by using k modules of birds from order 4.

5.2.2 Removal Through Interpretation Several constraints can be removed by interpreting the schedule in a certain manner—this applies especially to those constraints introduced to preserve the "practicality" of schedules. For instance, the constraint that a squad should not be allowed to "double-back" (i.e., visit two farms in the sequence A, B, A) can be removed by scheduling *all* work done at farm f by squad s in a contiguous block.

5.2.3 Penalty Function The remaining constraints are incorporated into a penalty function that penalizes each violation of a constraint in a schedule. These constraints may be classified as "hard" or "soft" and are suitably weighted to reflect this. The penalties used, and their associated weights, are defined in Table 1. A schedule must satisfy all the hard constraints (e.g., [OVERLOAD] and [UNDERLOAD]) and attempts to minimize violations of the soft constraints. A *feasible* schedule must satisfy all hard constraints, and it is preferable to minimize the number of violations of soft constraints incurred by a solution. A schedule cannot be accepted by the factory unless it is at least feasible.

Table 1. The penalties applied in the fitness function for *GA(Assign)*.

Constraint	Description	Importance	Weight
OVERLOAD	A squad must not collect more than max_t modules	*Hard*	10
UNDERLOAD	A squad must collect at least min_t modules	*Hard*	10
BOTH SIDES	A squad should not work on north and south of the Forth Estuary in the same day	*Soft*	3
TOTAL FARMS	Minimize the total number of farms a squad visits	*Soft*	1
SMALL LOAD	A squad should collect at least l modules from each farm it visits	*Hard*	10

The values of the penalties above were arrived at somewhat arbitrarily. Values that worked were easy to find, and those that worked best were adopted. The actual fitness of a solution is given by $(1 + \sum_{i=1}^{i=N} w_i N_i)^{-1}$, where N is the number of violations of constraint i, and w_i is its associated penalty; hence, a solution that incurs no penalty has fitness 1.

5.3 Evolving Heuristic Choice

In Figure 3, section II of the chromosome represents a permutation of the N orders that must arrive at factory f_i and determines the order in which they will be scheduled. The chromosome sections marked III and IV in Figure 3 encode the heuristics that should be used to split and assign each order, respectively. We interpret this as meaning that the ith order in section II should be split according to the heuristic denoted by the ith gene in section III and scheduled according to the heuristic denoted by the ith gene in section IV. Tables 2 and 3 define the sets of heuristics used. These heuristics were chosen to represent the rules used by the human scheduler when constructing schedules. For example, Table 2 defines heuristics that cover all ways in which we know the human scheduler currently splits orders, and Table 3 tries to capture some of the rules used by the human scheduler (somewhat intuitively) to ensure that schedules are practical. For instance, heuristics E and F in Table 3 implicitly minimize the total time spent in traveling by a squad (note, however, that this is not always desirable, as it could result in loads arriving at the factory too closely spaced).

5.4 Feasibility and Repair

The choice of assignment heuristic indicated at locus i may not be feasible as a method of splitting order i for two reasons. First, the choice of *order* and number of *modules* (o, m) used to satisfy the fixed criteria in section I takes precedence when constructing an assignment and hence may lead to an infeasible heuristic being assigned in section III to split the remaining loads of order o. Second, the chosen splitting heuristic may be incompatible with the size of the order under consideration — for example, an order containing more than 110 modules cannot be split by heuristic A. Before the chromosome is interpreted, it is checked for such anomalies, and any invalid assignments are repaired by randomly selecting another valid heuristic and altering the chromosome to represent this.

However, an assignment of work to squads constructed from the strategy represented by part IV of the chromosome may still be *infeasible*; for instance, a squad may be overworked

Table 2. Heuristics for splitting an order.

Heuristic	Description
A	If farm order <= 110, use all this
B	Split into single loads of 22 (but split first load into 10 + 12)
C	Split into tasks of size 44
D	1 task of size 66, split rest into single loads
E	Split into tasks of size 66 66
F	1 task of size 88, split rest into single loads
G	Split into tasks of size 88
H	1 task size 110, split rest into single loads
I	If order == 110, split into 56 and 54
J	If order == 110, split into 66 and 44

Table 3. Heuristics for assignment.

Heuristic	Description
A	Assign each part-load to the first squad found that satisfies all the constraints
B	Try and assign the load to a squad that is already doing this type of work (taking into account constraints)
C	Assign the load to an unused squad (if possible)
D	Assign the load to a squad already working (if possible)
E	Assign the load to a squad already going to this farm
F	Assign the load to a squad already going to the same *complex*
G	Assign the loads randomly

as a result of too many tasks being assigned to it. The chromosome is not repaired to correct this—the resulting assignment is simply heavily penalized via the penalty function. Repairing the chromosome in this case is judged too costly, as the infeasibility of the choice of assignment heuristic does not become clear until the preceding operations have been assigned. Others (e.g., Smith & Tate, 1993) have noted that maintaining infeasible chromosomes in the population can ensure that parts of the search space are not cut off. Smith and Tate used an adaptive penalty function that gradually reduced the number of infeasible chromosomes allowed—in this case, we find that a feasible population is rapidly produced, and hence this is not necessary.

5.5 Output of *GA*(*Assign*)

The result of applying *GA*(*Assign*) is thus a complete assignment of squads to work. The sequence in which a squad visits the farms assigned to it is also implicitly determined by this GA via the permutation in part II: The first farm (f_1) that is assigned to a squad s as the chromosome is interpreted from left to right is the first farm it visits; the next *new* farm to be assigned to this squad (i.e., different from f_1) is visited second; and so on. If at iteration t an order is assigned to a squad from a farm it is already visiting, the modules from this order are merely added to the modules already due to be collected at this farm.

Table 4. Results of experiments using *GA(Assign)* averaged over 10 trials.

Test Case	Minimum Fitness	Average Fitness	Maximum Fitness	% of Feasible Chromosomes in Initial Population
1	0.200	0.200	0.200	29
2	0.250	0.250	0.250	19
3	0.500	0.500	0.500	10
4	0.500	0.500	0.500	5
5	0.125	0.137	0.167	15
6	0.250	0.316	0.333	11
7	0.111	0.113	0.125	6
8	0.083	0.117	0.125	16.5

6. Experiments

Experiments were performed in eight test cases provided by the factory. Cases 1 to 4 were deemed "easy" to solve by the current scheduling expert, whereas cases 5 to 8 were found to be very difficult to solve by hand. Experiments with each test case were repeated 10 times and the results averaged. Experiments were repeated using a GA, a simple hill-climber (HC), a population-based hill-climber (Pop-HC), and simulated annealing (SA). For *GA(Assign)* and the Pop-HC, the population size was 100. Crossover and mutation operators were applied independently to each section of the chromosome. A two-point crossover operator was applied to sections I, III, and IV of the chromosome and PMX crossover to the permutation in section II (Mott, 1990). Mutation was applied with a probability 0.02 to each gene in each of sections I, III, and IV—an order-based mutation operator, that picks two loci at random and exchanges their alleles, was applied to the permutation of orders. All experiments were run for 10,000 evaluations, with a steady-state reproduction strategy. For the HC experiments, mutation was applied with probability 0.2, with the resulting solution accepted if it was fitter or of equal fitness to the current solution.

6.1 Results

The average, best, and minimum fitness found in 10 trials of *GA(Assign)* are shown in Table 4. In *every* case a feasible schedule is found, and in every case the assignments are judged to be acceptable by the expert. Here we define "acceptable" to mean that the schedules were similar to those currently produced by hand at the factory and could be employed in practice. Note that the maximum fitness is always unknown and differs from case to case. In each case, the final penalty results only from the unavoidable fact that at least one squad must visit more than one farm and hence can never be zero. The average fitness found by *GA(Assign)* is contrasted with that found using the SA, HC, and Pop-HC in Table 5. For SA, HC, and Pop-HC, the number in parentheses gives the probability that the result is *not* significantly different from the result found by the GA, calculated using Student t-tests.

Closer analysis of the results highlights the following facts.

6.1.1 Variation in Difficulty of Test Cases Although a good solution is always found, the ease with which each case is solved varies quite considerably across the problem set. Problems 1 to 4 are solved reliably, whereas problems 5 to 8 appear to be solved with more

Table 5. Average results obtained over 10 trials. Numbers in parentheses show the probability that the null hypothesis is true; that the result is not significant.

Test Case	GA	HC	SA	Pop-HC
1	0.200	0.200	0.200	0.200
2	0.250	0.250	0.237 *(0.166)*	0.250
3	0.500	0.500	0.500	0.500
4	0.500	0.250 *(0.331)*	0.431 *(0.092)*	0.500
5	0.137	0.111 *(0.547)*	0.128 *(0.215)*	0.128 *(0.215)*
6	0.317	0.167 *(0.062)*	0.270 *(0.050)*	0.308 *(0.628)*
7	0.113	0.113	0.111 *(0.331)*	0.121 *(0.124)*
8	0.117	0.091 *(0.10)*	0.120 *(0.822)*	0.124 *(0.395)*

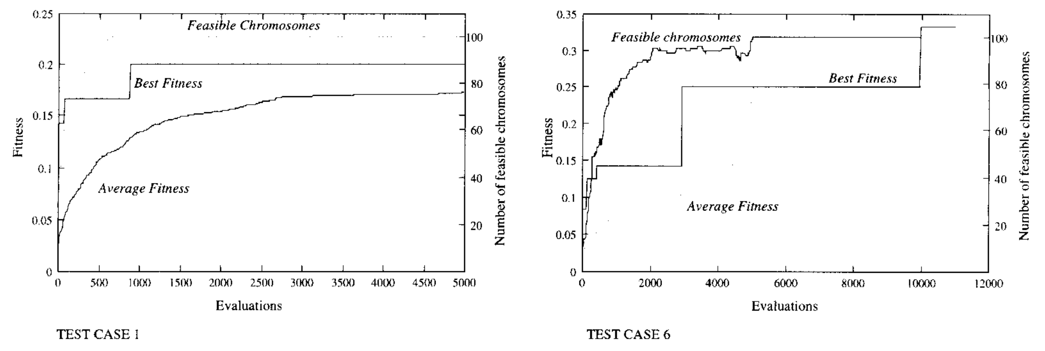

Figure 4. Best and average fitnesses.

difficulty, as shown in Table 4. Examining the number of feasible solutions produced on average in the initial populations shows that, in fact, in all cases *feasible* solutions can be randomly generated, although in all cases the fitness of these solutions is improved through search. The search is clearly more difficult in some cases; for example, see Figure 4, which shows the average and best fitness of the population as *GA(Assign)* runs, for cases 1 and 6. It also depicts the number of feasible chromosomes present in the population and the stepwise nature of the improvements in fitness.

6.1.2 Benefit From Evolving Choice of Split Heuristic Table 6 shows the distribution in the frequency with which a splitting heuristic was chosen across all the orders for each test case. The most commonly used heuristic in each case is shown in boldface (see Table 2 for the actual heuristics). The distribution is clearly not random and highlights the importance of selecting a suitable task size. In general, the most common choice is to use a coarse-grained split.[1] As briefly mentioned in Section 3.1, there is a trade-off between the level of granularity of the order split and the amount of work that has to be done in searching the space of possible task assignment: The finer the granularity, the more choices there are of possible assignments, hence the more work that has to be done in exploring the possible combinations (see Fig. 2).

1 Regardless of the split heuristic used for the purpose of assigning work to squads, ultimately (at the timing stage) single lorry loads have to be scheduled (i.e., a lorry load is the atomic unit).

Table 6. Frequency of application of splitting heuristic.

Test Case	% of Heuristic Being Applied to Split an Order									
	A	B	C	D	E	F	G	H	I	J
1	**84**	3	11	1	1	0	1	0	0	0
2	**78**	6	2	1	14	0	0	0	0	0
3	**44**	6	38	0	0	0	10	0	1	1
4	35	4	**43**	1	9	1	3	0	3	3
5	**50**	10	19	1	8	0	0	10	0	0
6	**79**	12	9	0	0	0	0	0	0	0
7	**47**	29	14	0	0	0	6	3	1	0
8	**43**	33	3	9	1	0	2	1	0	0

Table 7. Contrast in performance between *GA(Assign)* and modifications of *GA(Assign)* in which the split heuristic and the permutation were fixed.

Test Case	GA	GA+FixSplit (single lorryload)	GA+FixOrder
1	0.200	0.167	0.200 (0.0)
2	0.250	0.139	0.250
3	0.500	0.458	0.500
4	0.500	0.188	0.500
5	0.137	0.091	0.123
6	0.317	0.152	0.250
7	0.113	0.111	0.111
8	0.117	0.099	0.122

For example, Table 7 contrasts the results found when *GA(Assign)* is forced to fix the order split granularity at a single lorry load with those of allowing the *GA(Assign)* to evolve a choice of split granularity (as described above). In only two cases, (cases 7 and 3) are the average results comparable in quality, and we find that for these cases, the evolved choice of split is often the fine-grained split (see also Table 2).

Again, we emphasize the advantages of a flexible scheduler for a system in which the work to be scheduled can vary so much from day to day.

6.1.3 Benefit From Evolving Choice of Assignment Heuristic The distribution in frequency for the choice of assignment heuristic in Table 8 shows a less distinct preference for choice. In this case, the frequencies are more evenly distributed and do not deviate significantly from the 14% frequency that would be expected if the choice were completely random.

***6.1.4* Permutation of Orders** Table 7 also contrasts the results using a version of *GA(Assign)* in which the permutation of orders to be considered by the SB was fixed throughout the GA run. We see that for 50% of the cases, this has no effect. Of the remaining cases,

Table 8. Frequency of application of assignment heuristic.

Test Case	% of Heuristic Being Applied to Assign an Order						
	A	B	C	D	E	F	G
1	15	**19**	12	15	14	13	11
2	8	14	13	13	13	**28**	12
3	16	16	10	13	**18**	16	11
4	21	16	4	**25**	13	10	11
5	8	22	15	9	**23**	16	7
6	14	**18**	18	12	13	13	14
7	17	**21**	8	14	18	11	11
8	**26**	16	10	15	11	10	12

evolving a permutation is beneficial in three (cases 5, 6, and 7), whereas in case 8, a better result is obtained by using a fixed permutation. This last case is rather surprising, suggesting that the choice of permutation of orders used to test the effect of a fixed permutation was rather fortuitous and that the GA failed to evolve a permutation of equal quality. This variation in result also highlights the variation in problem difficulty from day to day. On balance, we conclude that treating orders in a flexible manner by using a permutation is advantageous, as it produces an equal or superior result in seven of the eight cases investigated.

6.1.5 Choice of Evolutionary Technique Table 5 compares the performance of a range of evolutionary techniques in the eight cases. The SA appears to perform the most poorly when looking at the test set as a whole and the two population-based methods, that is, *GA(Assign)* and Pop-HC show the best overall performance. The GA and Pop-HC give equivalent results on the easier cases 1 to 4, and we find that the GA outperforms Pop-HC in cases 5 and 6, whereas Pop-HC outperforms *GA(Assign)* in cases 7 and 8. Closer analysis of the assigments produced shows that all four methods always produce *feasible* assignments; however, those produced by SA and HC were often rejected by the human scheduler as "impractical." Thus, although these schedules *could* be used by the factory, the better solutions produced by *GA(Assign)* and/or HC were always preferred.

We conclude that a population-based approach is advantageous; however, further work is required to determine whether a GA has any significant advantage over a simple HC approach. A population-based approach also has the advantage of being able to produce multiple solutions with the same fitness: Analyzing the final population at the end of each test showed that in a single population many different solutions, each with the same fitness, existed. This allows the possibility of human input in choosing among equally fit schedules and, as discussed in Section 3.2, can have an important bearing on the quality of the results produced by *GA(Timer)*.

6.2 Is *GA(Assign)* Solving a Nontrivial Problem?

It is important to ask whether a GA is actually doing any work (i.e., that the problem is nontrivial); the purpose of this section is to illustrate that this is the case in this instance.

One hundred random solutions were generated for each case (see Table 9). Note that for case 1, random generation produces as good a solution as *GA(Assign)*. Also, for cases 3, 4, 7, and 8 the fitness of the best randomly generated solution is 50% or greater of the maximum

Table 9. Contrast in performance between randomly generated solutions and those evolved using *GA(Assign)*.

Test Case	Generate 100 Solutions Randomly		After Evolution
	Worst (Fitness)	Best (Fitness)	Best Solution (Fitness)
1	0.0008	0.2	0.2
2	0.0005	0.091	0.25
3	0.001	0.250	0.5
4	0.001	0.250	0.5
5	0.0008	0.071	0.267
6	0.001	0.111	0.333
7	0.0006	0.067	0.125
8	0.001	0.071	0.125

evolved fitness for each case. This could explain why SA and HC (see start of Section 6) all found comparable solution fitnesses: Because random initialization could discover them, fitnesses of that sort of level are not all that difficult to find.

It can be seen that in most cases the GA improves the fitnesses of the populations, and hence the search procedure is worthwhile. Although occasionally the problem is straightforward, mostly it is nontrivial; this demonstrates the need for a flexible system—the advantage of evolved strategies over solutions.

It should be noted that although in most initial randomly generated populations there are members that are moderately fit, the density of good solutions is low, and the spread of solution fitnesses and variance in the population is high. This is illustrated by Figure 5. The charts in Figure 5 show a bar for each unique fitness value that members of the initial population have. In each there is a wide spread of values, with no more than five members of a population having the same fitness value.

7. *GA(Timer)*

The *GA(Timer)* takes the output from *GA(Assign)* and attempts to formulate a factory schedule, by assigning a time at which each load is to be caught. The process is considerably simplified by the factory rules that a squad must continue working without a break once it has begun its shift. This means that it is only necessary to assign a start time for a squad—then, for a fixed sequence of farms ($f_i \cdots f_n$), given the number of loads to be caught at each farm, the travel time between farms, the travel time from farm to factory, and the catching time required for each load, we have all the necessary information to calculate the arrival time of each load at the factory. The sequence in which farms are considered is determined by *GA(Assign)*, that is, the first farm assigned to a squad by *GA(Assign)* is the first farm visited, and so on. The process for a single squad is illustrated in Figure 6.

A straightforward direct representation is used in which the chromosome has N genes, one for every squad taking part in the schedule. All factory arrivals are timed to the nearest 15 minutes. Each gene thus has an integer value a in the range 0 to 88, which is interpreted as meaning that the squad starts work at (15*a) minutes after midnight. The latest time a squad can start work is 22:00 hours, hence the upper limit on the value of a is 88, i.e., 1320 minutes after midnight.

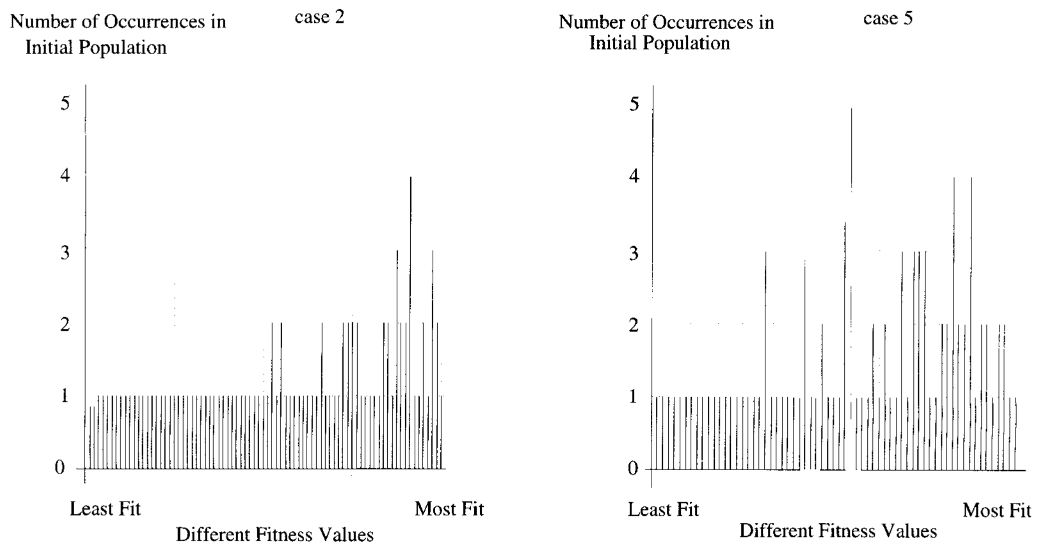

Figure 5. Graphs of cases 2 and 5, showing the diversity of fitnesses in the initial populations. In case 2 there are 80, and in case 5 there are 64, different fitness values. Note that the overall density of fit solutions, although relatively high (compared to the number of unfit solutions), is low in absolute terms (compared with the fitnesses *after* evolution).

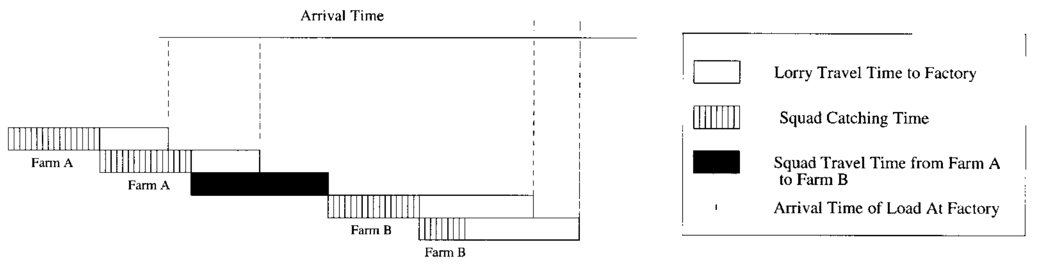
Arrival Time
Farm A
Farm A
Farm B
Farm B
Lorry Travel Time to Factory
Squad Catching Time
Squad Travel Time from Farm A to Farm B
Arrival Time of Load At Factory

Figure 6. An example schedule for a squad.

Some preprocessing of the input data can narrow down the allele range for each squad, with the effect that the search space is considerably reduced. For example, the squads allocated to satisfy the "unavoidable" criteria (see Section 5.2) in *GA(Assign)* have a predefined time window in which they must start. A predetermined number of loads must be waiting at the factory when it opens in the morning, and the last load must arrive half an hour before the factory closes. Squads are selected at random to satisfy these criteria, and the starting times are fixed.

The GA only evolves start times for each squad. When the factory schedule has been built, lorry assignment is done by the following iterative process:

1. Sort the tasks in chronological order of arrival at the factory;
2. Set start time(*lorry l*) = 0 for all lorries;

Table 10. Constraints applied to *GA(Timer)*.

Constraint	Description	Importance
LATE	Loads arriving after a factory closes	*Hard*
EARLY	Only two loads must arrive before a factory opens	*Hard*
SIMULTANEOUS	The number of loads arriving at the same moment in time	*Soft*
STANDARD DEVIATION	The deviation of arrival times from the ideal schedule in which loads arrived with even gaps throughout the day	*Soft*
LORRIES	The number of *extra* lorries that must be contracted in to deliver the loads	*Soft*

3. For each unassigned task *i*:

```
consider each lorry l with start time > 0:
    if (start time (task i) > finish time (lorry l)
        allocate (task i) to lorry l
        finish time (lorry l) = finish time (task i)
```

4. If still not assigned (*task i*):

```
assign (task i) to a new lorry (l + 1)
start time (lorry l) = start time (task i)
finish time (lorry l) = finish time (task i)
```

5. Repeat from step 3 until all tasks assigned

This method is preferred over an evolutionary one because the majority of assignments of lorry to task made at random would be infeasible, and hence the GA would spend most of its time searching for feasible rather than good solutions. Each factory schedule is evaluated for quality using a weighted penalty function that penalizes violations of constraints. As in *GA(Assign)*, the constraints can be "hard" or "soft"—acceptable schedules must not violate *any* hard constraints, and all schedules must minimize the number of violations of soft constraints. The constraints used are detailed in Table 10. The values for the weights of the constraints were arrived at easily and arbitrarily (as in Section 5.2.3). Values that worked were quick to find, and those that worked well were adopted.

8. Experiments

For every test case, the best assignment found by running *GA(Assign)* was used as input to *GA(Timer)*, which was run for 5000 evaluations. The experiments with each test case were

Table 11. The worst and best penalties found from the *GA(Timer)* using a set of input assignments for each test case having equal fitness but different assignments. Note: the lower the value of the significance (column 4), the more significant the value.

Test Case	Best	Worst	Significance
1	300	330	0.421
2	280	338	0
3	283	315	0.196
4	306	330	0.001
5	211	218	0.708
6	314	368	0.001
8	269	296	0.373

repeated 10 times. Two-point crossover and a mutation rate of 0.02 were used, with a steady-state reproduction strategy. On a good laptop PC, the speed was around 10 evaluations per second.

8.1 Results

The *GA(Timer)* evolves acceptable schedules in every case. Here, acceptable means that the factory is supplied with birds throughout the day and is never either idle or oversupplied. In each case the final penalty results only from the standard deviation of the times of the load arrivals—although not possible to prove, it is reasonable to assume that there cannot be an optimum schedule in which loads arrive at precisely even intervals throughout the day, given the widely spread geographical locations of the farms. In practice, a situation often arises where two loads arrive at once, or a few loads arrive closely spaced. Also, the lorry schedules produced use fewer lorries than those available at the factory, representing a possible cost saving to the factory.

8.2 Relationship Between Assignments and Factory Schedules

Although the *GA(Timer)* evolves satisfactory schedules, the question of whether they are constrained by the optimality of the input assignment must be addressed. The exact sequence with which a squad visits farms, as well as the allocation, can affect the resulting schedules.

The *GA(Timer)* could be improved by considering permutations of the possible sequences in which each squad visits its allocated set of farms. In the current version, the sequence used for each squad is simply the order in which *GA(Assign)* considered each farm; thus if the permutation of orders in *GA(Assign)* specified "ABCDE," and squad S was assigned work at farms B and E, the *GA(Timer)* assumes a sequence "BE." By examining other permutations, it may be possible to achieve better schedules.

As noted in Section 6.1.5, running *GA(Assign)* generally results in a population with several different chromosomes of equal fitness. Given a set of assignments $A(i)$ of equivalent fitness for each test case i, we use each member of A as input to the *GA(Timer)* and find the average quality of the schedule produced over 10 runs. Table 11 shows the penalty assigned to the best and worst quality schedules found in each set $A(i)$ and the probability that the results are not significant.

The table shows that in several cases, the factory schedule can be significantly improved by judicious choice of input schedule. As the schedules are produced very quickly, a simple improvement to the system would be to evolve factory schedules from a selection of the fittest chromosomes in the final *GA(Assign)* population and present the user with the fittest timing schedule found and the assignment schedule needed to produce this.

Another appealing feature from the user's point of view would be to evolve a timing schedule for every member of the assignment population that has reached the "best" fitness. The output of the scheduling system is then a set of pairs of (assignment + timings) from which the user can choose which pair best represents a satisfactory trade-off between factory schedules and assignment schedules.

9. Conclusion

The combination of two GAs—*GA(Assign)* and *GA(Timer)*—has resulted in a robust system that produces schedules that are *acceptable* to the factory for every test case. The eight test cases used varied in complexity and were representative of typical siutations that could occur in the factory. This system was able to produce feasible schedules that could be put into working practice by the company, and were similar to those already used. Results were compared using a GA, SA, a Pop-HC, and a simple HC, all based on the same problem representation. All the search methods produced *feasible* schedules that satisfied hard constraints, but those produced by the GA and Pop-HC were judged to be superior by the factory experts because they represented more practical solutions.

Although this paper discusses a very specific application, several of the design concepts can be extended to other scheduling problems, particularly those involving complex real-world data and constraints. In particular, we highlight the benefit of employing the following techniques:

- Division of a complex problem into smaller subproblems.
- Separation of the most important constraints from the penalty function and instead incorporating them into the chromosome representation.
- Evolving an SB that uses heuristic choice as a means of dealing with complex domain knowledge.

Dividing the problem into smaller subtasks and solving each separately simplifies the search space considerably and hence increases efficiency. However, we must recognize that this may compromise the solution quality because the quality of the final output is dependent on the quality of the output from each individual subtask.

An ideal approach would consider a continuous evaluation cycle, as shown in Figure 7, where there is some form of feedback from one subtask to another, and each subtask is iteratively evolved until a satisfactory solution is reached. An example of successful implementation of an iterative algorithm is given by Ryu, Hwang, Choi, and Cho (1997). In this case, the aim would be to improve the assignments based on the ability of the timing module to produce a high-quality schedule from a given assignment. However, Table 12 shows the *average* number of evaluations in each test case before *GA(Assign)* evolves a population that consists of feasible chromosomes. This varies considerably, and hence an iterative scheme could result in *timing* schedules being evolved for infeasible *assignments* in much of the early reproductive cycles, and consequently, an unnecessary expenditure of energy evaluating infeasible schedules. A sensible and efficient method of combining information from both evolutionary processes would have to be established. An alternative methodology would be

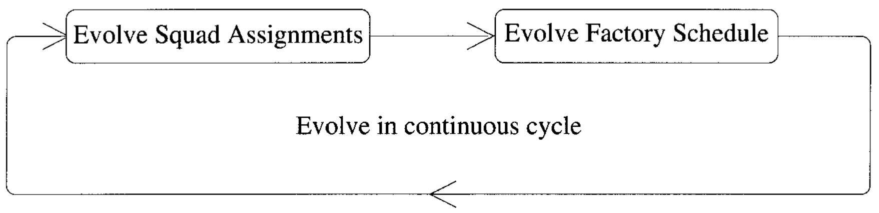

Figure 7. The ideal evolutionary model for schedule construction.

Table 12. Variation in number of evaluations until *GA(Assign)* produces a feasible population.

Test Case	1	2	3	4	5	6	7	8
Evaluations before 100% of Population Feasible	711	3690	1012	2028	660	5514	1100	1315

to treat the assignment constraints and the factory constraints as different objectives within a single GA and search for a satisfactory trade-off between the two quantities. A multiple objective GA approach is currently being investigated by the authors.

In section 5.2, we described three methods of dealing with constraints that further simplified the problem structure and reduced the search space. In particular, we significantly decreased the size of the search space by separating the most important constraints from the penalty function and using a chromosome representation that guarantees that these constraints will be met. The way in which they are met is encoded on the chromosome, and the GA searches for the optimal way to meet the constraints. This ensures that all feasible regions of the search space are explored while improving the efficiency of the search by cutting off infeasible regions. This places less burden on the penalty function, and as a result, the inherent difficulties usually associated with penalty functions in finding suitable weights are avoided.

Finally, we emphasize the importance of the heuristically driven evolving SB in the success of this application. Although the value of using a GA to find optimal combinations of heuristics as a method of solving complex scheduling problems has been previously noted (e.g., Norenkov & Goodman, 1997) in solving job-shop scheduling problems, we stress that the method is particularly advantageous when tackling real-world problems. Often, a large amount of domain knowledge can be gathered from an expert, but it is generally complex and often difficult to formulate into exact procedures for solving problems. By encapsulating the knowledge in the form of heuristics, and using a GA to construct an SB via an optimal combination of these heuristics, we considerably simplify the task. The key advantage, however, lies in the flexibility that such a system provides—real problems do not follow an exact pattern from day to day, and each schedule may require a subtly different approach. Using the heuristically driven SB, a unique SB is built each time a problem is solved, which takes account of the individual characteristics of each problem. This results in a much more robust system that copes well with variations within the data and hence can be used by the factory as a reliable tool.

References

Bagchi, S., Uckun, S., Miyabe, Y., & Kawamura, K. (1991). Exploring problem specific recombination operators for job-shop scheduling. In R. Belew & L. Booker (Eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms* (pp. 10–17). San Mateo, CA: Morgan Kaufmann.

Bierwith, C. (1995). A generalized permutation approach to job-shop scheduling with genetic algorithms. *Or Spektrum*, *17*(2-3), 87–92.

Cleveland, G. A., & Smith, S. F. (1989). Using genetic algorithms to schedule flow shop releases. In J. D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications* (pp. 160–169). San Mateo, CA: Morgan Kaufmann.

Dorndorf, U., & Pesch, E. (1995). Evolution based learning in a job shop scheduling environment. *Computers and Operations Research*, *22*(1), 25–40.

Fang, H.-L. (1993). *Genetic algorithms in timetabling and scheduling*. Unpublished doctoral dissertation, Department of Artificial Intelligence, University of Edinburgh.

Fang, H.-L., Ross, P., & Corne, D. (1993). A promising genetic algorithm approach to job-shop scheduling, rescheduling, and open-shop scheduling problems. In S. Forrest (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms* (pp. 375–382). San Mateo, CA: Morgan Kaufmann.

Langdon, W. (1996). Scheduling maintenance of electrical power transmission networks using genetic programming. In J. Koza (Ed.), *Late-breaking papers, genetic programming 96* (pp. 107–116). Stanford Bookstore, Stanford, CA.

Lin, S.-C., Goodman, E. D., & Punch, W. F. (1997). A genetic algorithm approach to dynamic job-shop scheduling problems. In T. Bäck (Ed.), *Proceedings of the Seventh International Conference on Genetic Algorithms* (pp. 481–489). San Francisco, CA: Morgan Kaufmann.

Mattfeld, D. (1996). *Evolutionary search and the job shop*. Heidelberg: Physica-Verlag.

Michalewicz, M., & Janikow, C. Z. (1991). Handling constraints in genetic algorithms. In R. Belew & L. Booker (Eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms* (pp. 151–157). San Mateo, CA: Morgan Kaufmann.

Mott, G. (1990). *Optimising flowshop scheduling through adaptive genetic algorithms*. Unpublished master's thesis, chemistry part II thesis, Oxford University.

Norenkov, I., & Goodman, E. (1997). Solving scheduling problems via evolutionary methods for rule sequence optimization. In P. K. Chowdry, R. Roy, & R. K. Pant (Eds.), *Soft computing in engineering design and manufacturing*. London: Springer.

Richardson, J., Palmer, M., Leipin, G., & Hilliard, M. (1989). Some guidelines for gas with penalty functions. In J. D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications* (pp. 191–197). San Mateo, CA: Morgan Kaufmann.

Ryu, K. R., Hwang, J., Choi, H. R., & Cho, K. K. (1997). A genetic algorithm hybrid for hierarchical reactive scheduling. In T. Bäck (Ed.), *Proceedings of the Seventh International Conference on Genetic Algorithms* (pp. 497–505). San Francisco: Morgan Kaufmann.

Shaw, K., & Fleming, P. (1997). Which line is it anyway—Use of rules and preferences for schedule builders in genetic algorithms for production scheduling. In D. Corne & J. L. Shapiro (Eds.), *Selected Papers: AISB Workshop on Evolutionary Computing, Lecture Notes in Computer Science*. Berlin: Springer-Verlag.

Smith, A., & Tate, D. (1993). Genetic optimization using a penalty function. In S. Forrest (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms* (pp. 499–505). San Mateo, CA: Morgan Kaufmann.

Syswerda, G., & Palmucci, J. (1991). Application of genetic algorithms to resource scheduling. In R. Belew & L. Booker (Eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms* (pp. 502–508). San Mateo, CA: Morgan Kaufmann.