Pergamon 0305-0548(93)E0014-K

Computers Ops Res. Vol. 22, No. 1, pp. 5–13, 1995

0305-0548/95 $7.00+0.00

A GENETIC ALGORITHM FOR FLOWSHOP SEQUENCING

COLIN R. REEVES†
Department of Statistics and Operational Research, Coventry University, Priory Street, Coventry CV1 5FB, England

Scope and Purpose—The concepts of genetic algorithms (GAs) have been applied successfully to problems involving pattern recognition, classification and other problems of concern to the Artificial Intelligence community. There is little reported in respect of application to large combinatorial problems. This paper attempts to apply GAs to one such problem—the $n/m/P/C_{max}$ sequencing problem. GAs have many possible choices of control parameters, but the emphasis here is on showing the feasibility of using GAs for such problems by producing a working algorithm, rather than on a search for the optimal parameter settings.

Abstract—The basic concepts of Genetic Algorithms are described, following which a Genetic Algorithm is developed for finding (approximately) the minimum makespan of the n-job, m-machine permutation flowshop sequencing problem. The performance of the algorithm is then compared with that of a naive Neighbourhood Search technique and with a proven Simulated Annealing algorithm on some carefully designed sets of instances of this problem.

INTRODUCTION

It is now widely recognized that finding optimal solutions to large combinatorial problems is not a practicable option, because of the vast amount of computer time needed to find such solutions. In reality a "good" solution, obtained in reasonably small computer time by a heuristic, is often the only possibility. The development of the concept of NP-completeness and NP-hardness [1] has meant that much research has been concentrated on the design of heuristics. One such problem, known to be NP-hard [2], is the permutation flowshop sequencing problem in which n jobs have to be processed (in the same order) on m machines. The object is to find the permutation of jobs which will minimize the makespan, i.e. the time at which the last job is completed on machine m. The problem investigated in this paper is conventionally given the notation $n/m/P/C_{max}$ [2], and is defined as follows:

If we have processing times $p(i, j)$ for job i on machine j, and a job permutation $\{J_1, J_2, \ldots, J_n\}$, then we calculate the completion times $C(J_i, j)$ as follows:

$$C(J_1, 1) = p(J_1, 1)$$
$$C(J_i, 1) = C(J_{i-1}, 1) + p(J_i, 1) \quad \text{for } i = 2, \ldots, n$$
$$C(J_1, j) = C(J_1, j-1) + p(J_1, j) \quad \text{for } j = 2, \ldots, m$$
$$C(J_i, j) = \max\{C(J_{i-1}, j), C(J_i, j-1)\} + p(J_i, j) \quad \text{for } i = 2, \ldots, n;\ j = 2, \ldots, m$$
$$C_{max} = C(J_n, m).$$

The ideas involved in Genetic Algorithms (GAs) were originally developed by Holland [3] and his associates, and applied to topics of interest to researchers in Artificial Intelligence. Thus there have been applications to classification, pattern recognition and machine learning. (There is a collection of papers and an extensive bibliography in Davis [4].) However, apart from some work on the Travelling Salesman Problem (not totally successful), little has been published on applications to combinatorial optimization.

†Colin R. Reeves is a senior lecturer in Operational Research in the School of Mathematical and Information Sciences at Coventry University. His main research interests are in applications of neural networks to pattern recognition problems, and in heuristic methods for combinatorial optimization, on which he has published several papers. He edited and co-authored the recently published book *Modern Heuristic Techniques for Combinatorial Problems*.

In this paper, the concepts of GAs will first be described and then applied to the solution of the $n/m/P/C_{max}$ sequencing problem. The computational experience gained with this approach will then be compared with results obtained using a simple neighbourhood search, and also by Simulated Annealing. This approach, introduced by Kirkpatrick *et al.* [5], has been applied to many problems of combinatorial optimization in the past few years. The method, and the considerable body of theoretical and experimental knowledge that has accumulated is described fully by van Laarhoven and Aarts [6].

Recently, Ogbu and Smith [7, 9] have shown that Simulated Annealing produces high quality results when applied to the flowshop sequencing problem. Using a different implementation of the basic idea of Simulated Annealing, Osman and Potts [8] have also produced a heuristic with very good performance. According to [9], the two methods are comparable in terms of solution quality and efficiency, so the Osman/Potts version, which is slightly easier to implement, was used as a benchmark for the comparison with a Genetic Algorithm.

There are basically two ways to compare different heuristics. One is to record how long it takes (which in practice approximates to how many evaluations of the objective function are carried out) to produce a solution of a certain standard. Alternatively, one can fix the time allocation, and compare the standard of the solution obtained after this has elapsed.

In this work, the second approach was adopted, since neither of the algorithms have a natural stopping point, and neither of them can guarantee to find a solution of a fixed standard. Further, it may be important to have a method which is time-limited if the sequencing problem has dynamic aspects. Particularly in the case of large problems, by the time the problem is "solved", it may have changed (for instance, more jobs may have arrived), so a method that takes too long may be inappropriate whatever the quality of the solutions it produces for a static problem.

GENETIC ALGORITHMS

The name originates from the analogy between the representation of a complex structure by means of a vector of 1s and 0s, and the idea, familiar to biologists, of the genetic structure of a chromosome. In selective breeding of plants or animals, for example, offspring are sought which have certain desirable characteristics—characteristics which are determined at the genetic level by the way the parents' chromosomes combine. Several genetic "operators" can be identified, the most commonly used ones being crossover (an exchange of sections of the parents' chromosomes), and mutation (a random modification of the chromosome).

In the context of finding the optimal solution to a large combinatorial problem, a Genetic Algorithm works by maintaining a population of M solutions—potential "parents"—whose "fitness values" have been calculated. In Holland's original GA, one parent is selected on a fitness basis (the better the fitness value, the higher the chance of it being chosen), while if crossover is to be applied, the other parent is chosen randomly. They are then "mated" by choosing a crossover point X at random, the offspring consisting of the pre-X section from one parent followed by the post-X section of the other.

For example, suppose we have parents P1 and P2 as follows, with crossover point X; then the offspring will be the pair O1 and O2:

P1	1 0 1 0 0 1 0	O1	1 0 1 1 0 0 1
	X		
P2	0 1 1 1 0 0 1	O2	0 1 1 0 0 1 0

One of the existing population is chosen at random, and replaced by one of the offspring. The *reproductive plan* is repeated as many times as is desired. In another version of this procedure, parents are chosen in strict proportion to their fitness values (rather than probabilistically), and the whole population is changed after every set of M trials, rather than incrementally.

Holland showed that procedures of this type have a property which he called "intrinsic parallelism". To explain this in detail is not the purpose of this paper, but roughly, this says that such plans, while only evaluating one chromosome at each iteration, manage to explore a much larger area of the solution space, at the same time favouring those parts of the space where the

values are above average. In this sense, the GA paradigm can be seen as an "intelligent" way of exploiting information obtained from a random search. Holland's book [3] discusses in great depth the reasons why this type of search is likely to be successful. A more recent book by Goldberg [10] gives a very readable and rather more intuitive account.

Many other variations on this theme are possible: both parents can be selected on a value basis, mutation or other operators can be applied, more than one crossover point could be used, and so on. Furthermore, as well as these methodological decisions, there are parametric choices: different crossover and mutation probabilities, different population sizes and so on. It is also quite possible that the way the initial population is chosen will have a significant impact on the results. These are discussed in a recent survey by Reeves [11], and in greater detail by Goldberg [10].

One GA whose empirical results are very encouraging is that due to Ackley [12]. It differs from Holland's in several respects, but principally in that instead of "reproduction with emphasis", it uses "termination with prejudice". Ackley used the incremental type of reproductive plan with random selection of both parents, but after each mating, the offspring replaces an existing chromosome whose value is worse than average.

The main practical advantage of this approach is that "good" (i.e. better than average) chromosomes are guaranteed to survive. In Holland's original GA, it is quite possible to discard the best chromosome in moving from one time step to the next. This may be undesirable in the context of an optimization problem. Furthermore, in some experiments where Holland's original algorithm was applied to the $n/m/P/C_{\max}$ problem, the procedure took a long time to converge, so it was decided to develop Ackley's procedure for this problem.

A GENETIC ALGORITHM FOR SEQUENCING PROBLEMS

Chromosomal representation

In order to apply any GA to a sequencing problem, there is an obvious practical difficulty. In most "traditional" GAs, the chromosomal representation is by means of a string of 0s and 1s, and the result of a genetic operator is still a valid chromosome. This is not the case if one uses a permutation to represent the solution of a problem, which is the natural representation for a sequencing problem.

For example, if the parents are as shown below, it is clear that the offspring are "illegitimate".

P1 2 1 3 4 5 6 7 O1 2 1 3 2 5 7 6

X

P2 4 3 1 2 5 7 6 O2 4 3 1 4 5 6 7

Different representations are possible, but it seemed easier to modify the idea of crossover and keep the permutation representation. Two modifications were tried. The first (C1) chose one crossover point X randomly, took the pre-X section of the first parent, and filled up the chromosome by taking in order each "legitimate" element from the second parent.

P1 2 1 3 4 5 6 7 O1 2 1 3 4 5 7 6

C1 X ⇒

P2 4 3 1 2 5 7 6 O2 4 3 1 2 5 6 7

The rationale for C1 is that it preserves the absolute positions of the jobs taken from P1, and the relative positions of those from P2. It was conjectured that this would provide enough scope for modification of the chromosome without excessively disrupting it. Preliminary experiments showed that C1 tended to converge rather too rapidly to a population of identical chromosomes, so a mutation operation was also included. C1 on its own emphasizes *exploitation* over *exploration*; incorporating a mutation helps to restore the balance.

The premature convergence problem is one that has been noted by several workers in applying GAs, and various proposals have been made to prevent it. The method adopted here was to use a adaptive mutation rate P_m. At the outset, a high probability of mutation is allowed; this probability is slowly decreased as long as a reasonably diverse population exists. If the diversity becomes too

small, the original high value is restored. Several measures of diversity were tried, all of which gave similar results. The method adopted was to reset the mutation rate if the ratio v_{min}/v_{mean} exceeded a value D, where v is the makespan for each chromosome.

The other approach (C2) used two crossover points X and Y. Again, the pre-X section of the first parent was retained, and the "legitimate" elements of the post-Y section of the second parent. The gaps were filled in by a random choice from the unused elements.

```
        P1   2 1 3 4 5 6 7        O1   2 1 5 4 3 7 6
C2
              X   Y
        P2   4 3 1 2 5 7 6        O2   4 3 2 5 1 6 7
```

It was expected that this would disrupt the chromosome much more than C1, so that mutation would not be needed, and this appeared to be true in a few experiments. In fact, disruption seemed to be excessive, and as the solutions converged very slowly it was decided to concentrate on C1 rather than try to refine C2.

Selection mechanism

Other variations on Ackley's algorithm were also found to be helpful. The original description calls for random selection of parents, the beneficial effects of genetic search being gained by the replacement of "poor" chromosomes. But it is also possible to allow at least one parent to be selected according to its fitness, with lower-valued chromosomes (in this context) being the favoured ones. But if this idea is adopted, it raises another question: how should we measure "relative fitness"?

The usual method (for maximization) is to measure relative fitness as the ratio of the value of a given chromosome to the population mean. However, in minimization problems, we need to modify this so that low-valued chromosomes are the "good" ones. One approach is to define fitness as $(v_{max} - v)$, but of course we are unlikely to know what v_{max} is. We could use the largest value found so far as a surrogate for v_{max}, but in practice this method was not very good at distinguishing between chromosomes. In essence, if the population consists of many chromosomes whose fitness values are relatively close, the resolution between "good" and "bad" ones is not of a high order.

This approach was therefore abandoned, in favour of a simple ranking mechanism. Sorting the chromosomes is a simple task, and updating the list is also straightforward. The selection of parents was then made in accordance with the probability distribution

$$p([k]) = \frac{2k}{M(M+1)}$$

where $[k]$ is the kth chromosome in ascending order of fitness (i.e. descending order of makespan). This implies that the median value has a chance of $1/M$ of being selected, while the Mth (the fittest) has a chance of $2/(M+1)$, roughly twice that of the median.

The ranking mechanism was also used in selecting the chromosome which is to be "terminated". Ackley chose from those chromosomes whose fitness was below average in the sense of the arithmetic mean. When we already have a ranking of the chromosomes, it seems sensible to choose from chromosomes which are below the median.

Mutation

As already indicated, a mutation operator was used to reduce the rate of convergence. In "traditional" GAs, mutation is applied by "flipping" each element of the chromosome from 0 to 1 (or vice-versa) with a small probability. With a permutation representation, mutation needs to be defined differently.

Two types of mutation were tried. The first, an *exchange* mutation, was a simple exchange of two elements of the permutation, chosen at random. The second, a *shift* mutation, was a shift of one element (chosen randomly) a random number of places to the right or left. A few experiments were made in which *shift* seemed to be better than *exchange*, so this was adopted.

Initial population

The final modification investigated was in the selection of the initial population. Most GAs in other contexts assume that the initial population is chosen completely at random; this was done

in some experiments with the algorithm described here. However, it also seemed worth trying the effect of "seeding" the initial population with a good solution generated by a constructive heuristic. There are several such heuristics known, but the consensus seems to be that the best is the NEH algorithm due to Nawaz *et al.* [13]. Accordingly, another version of the GA was tested in which one solution was obtained from this heuristic, while the remaining $(M-1)$ were generated randomly.

In a few trials of this procedure, the one with the seeded population appeared to arrive at its final solution rather more quickly, with no observed diminution in solution quality. This modification was also therefore included in the final version of the algorithm.

A pseudo-code description of the version finally implemented is given in an Appendix to this paper.

THE PROBLEMS

Most investigators of machine-sequencing problems have tested their heuristics on problem instances in which the processing times have been randomly generated. However, most of these test problems have not been published, so it was necessary to generate some new instances to test these heuristics. Seven sets of six problem instances were generated, varying from 20 jobs on 5 machines to 75 jobs on 20 machines.

There is evidence (e.g. Amar and Gupta [14]) that completely random processing times are unlikely to occur in practice, and Rinnooy Kan [2] suggests that two aspects of non-random structure in processing times are a gradient of times across machines, and a correlation of times within jobs. Further sets of problem instances were generated with these characteristics, using the parameters given in [2] to simulate these aspects of the problem; the problem sizes were as in the first group. Thus there were 126 problem instances in all. In the discussion that follows, instances of type C are those generated randomly using a uniform distribution, while problems with time gradients and job correlations are labelled type A and B respectively.

Results

Both the heuristics were programmed in Pascal, and run on a Sequent S82 computer. As a further comparison, a local neighbourhood search (NS) heuristic was also programmed and used to solve each problem. Initially, all three procedures were allowed the same number of function evaluations: that prescribed by Osman and Potts in their Simulated Annealing algorithm. However, it became apparent that the GA procedure needed more computer time than the others, as it is more complicated to implement, and also incurs fairly substantial "overheads" in terms of population updating. However, this did not affect the results very much, as in many cases the GA reached a solution long before the number of evaluations was exhausted; for the few that did not, the best solution found at the time that the SA algorithm would have stopped was recorded instead.

Further, in a recent paper by Taillard [16], a method is given for implementing the NEH [13] heuristic in an efficient manner. This procedure can easily be adapted to a neighbourhood search, but in so doing, the concept of a fixed number of function evaluations in the sense of [8] is no longer meaningful. The evaluation of the NS procedure was therefore modified too, by starting from an initial random permutation and running it for the same amount of *time* as SA. If, as sometimes occurred, a local optimum was found before this time limit was reached, the procedure was allowed to restart from a different initial solution.

To evaluate the heuristics, each was applied once to the problem instances generated as described above. The results were then averaged over the 6 instances relating to the same problem type and size. An alternative strategy would be to apply each heuristic several times to a single instance and average the results thus obtained. This approach was not adopted here for the following reason. In the case of a naive search, it is *necessary* to start several times from different points in an attempt to have a wider coverage of the solution space, and it is natural to copy this approach for evaluation purposes. However, for more sophisticated methods like SA and GA, this sort of coverage should be achieved from any initial solution, so that the effect of different starting points is much less interesting than the effect of different instances.

Type A problems. The optimal solution of instances with time gradients is rather easy to find, as they tend to be dominated by one machine, and a tight lower bound based on that machine is easily found. In all cases the GA found a solution whose value was identical to the lower bound,

Table 1. Comparison of solution quality

Problem size	Method GA	SA	NS
	Type B Problems		
20/5	0.00	0.02	0.14
20/10	0.03	0.06	0.00
20/15	0.01	0.00	0.03
30/10	0.06	0.21	0.53
30/15	0.01	0.03	0.18
50/10	0.09	0.15	0.72
75/20	0.27	0.44	1.25
Average	0.07	0.13	0.41
	Type C Problems		
20/5	0.36	0.52	1.25
20/10	1.29	1.33	1.41
20/15	1.48	1.64	1.61
30/10	0.65	0.68	1.12
30/15	1.42	1.07	1.89
50/10	1.06	1.30	1.79
75/20	3.02	3.33	4.67
Average	1.33	1.41	1.96

and which is therefore optimal. SA was only slightly inferior—it found the optimal solution in every case but one, as did the naive neighbourhood search. Thus, as the performance was so similar, detailed results on type A problems have not been included in the table of results that follows.

The results of this evaluation for types B and C problems are shown in Table 1, which shows the mean percentage difference of the solutions obtained relative to the best solution known for each instance. (These upper bounds on the value of the makespan were variously obtained from the methods examined here, from a modified version of the NEH heuristic [15], and from some runs of SA with an extremely slow cooling schedule. Both these latter methods took very large amount of computer time.) In each case, the mean is the average over 6 instances.

Type B problems. It is not so easy to find tight lower bounds for instances with correlations within jobs, so the absolute quality of the results is unknown. However, the neighbourhood search heuristic performed almost as well as SA and GA, and there was very high agreement between the two more sophisticated heuristics. In many cases they gave identical results, and even in the few cases where differences occurred, they were very small (the largest individual difference observed was 0.3%); it seems the topography of the solution space for such cases is fairly smooth, so that getting stuck in a deep local minimum is unlikely.

Type C problems. For the randomly generated problems, it can be seen that on average the quality of solution produced is very similar, although there is a degree of variation which was absent in the non-random cases. Comparing GA and SA directly, the largest individual deviation found was 2.4%; GA was slightly better on average in 6 out of the 7 problem groups. Both clearly did better than NS.

TAILLARD'S PROBLEMS

Recently, Taillard [17] has produced a set of test problems which he found to be particularly difficult, in the sense that the best solutions he could find by a very lengthy Tabu Search procedure were still substantially inferior to their lower bounds. The problems range from small instances with 20 jobs and 5 machines to large ones with 500 jobs and 20 machines. There were 10 instances for each size of problem, and all the processing times were generated randomly from a $U(1, 100)$ distribution.

The three procedures tested above were applied to these problem instances with the results seen in Table 2, where the solution quality is measured by the mean percentage difference from Taillard's upper bounds, averaged over the ten instances within each group of problems.

It can be seen that SA and GA again out-perform NS on all except the smallest problems. In comparing SA and GA directly, there was again little to choose between them overall in that their distance from the upper bounds were very similar. On the whole, SA performed very slightly better

Table 2. Comparison of solution quality

Problem size	Method GA	SA	NS
	Taillard's Benchmarks		
20/5	1.61	1.27	1.46
20/10	2.29	1.71	2.02
20/20	1.95	0.86	1.10
50/5	0.45	0.78	0.79
50/10	2.28	1.98	3.21
50/20	3.44	2.86	3.90
100/5	0.23	0.56	0.76
100/10	1.25	1.33	2.69
100/20	2.91	2.32	3.98
200/10	0.50	0.83	3.81
200/20	1.35	1.74	6.07
500/20	−0.22	0.85	9.07
Average	1.50	1.42	3.24

Table 3. Comparison of solution effort

Problem size	Method GA	SA	NS
	Taillard's Benchmarks		
20/5	31.8	42.8	45.0
20/10	34.7	51.3	43.4
20/20	23.0	36.3	59.4
50/5	40.8	60.3	65.8
50/10	55.2	73.6	57.4
50/20	52.7	82.2	63.0
100/5	60.0	53.8	79.7
100/10	52.6	76.4	81.5
100/20	45.9	86.0	80.6
200/10	50.9	76.7	100
200/20	63.1	93.0	100
500/20	66.5	91.3	100

than GA, but the figures show that GA did better for the larger problems. It is also worth pointing out that for the last set, GA actually found 9 solutions (out of 10) which improved on Taillard's upper bounds.

Another factor of interest is the effective amount of computation that each method needed. The actual computer time allowed for each problem was the same for all methods, as explained earlier, but in many cases, the final solution was obtained well before the limit was reached. Table 3 shows the amount of computer time used (as a percentage of this limit) for each method to reach its final solution, averaged over the 10 problems in each group.

These figures show that GA generally reached its final solution rather more quickly than SA or NS. (In the case of NS on the largest problems, the improvement had clearly not ceased even when the limit was reached.) This is not unimportant, especially for the larger problems where the actual computer time needed was considerable—nearly 2 h of CPU time on a mainframe was needed for each of the 500/20 problems. The implication of these figures is that were the limit to be reduced substantially the performance of GA would be far less affected than SA.

CONCLUSION

The main aim of this research was to explore the potential of Genetic Algorithms. However, as a subsidiary observation, we see that for certain types of problem, it may not be worth using sophisticated procedures, since a simple neighbourhood search can obtain solutions of comparable quality very easily. This dependence of solution quality on problem structure in combinatorial optimization is not one that has been adequately addressed in the literature. For problems where structure is not apparent, a naive method does less well, and we would make the following observations in respect to the other methods.

The overall implication of the studies carried out is that SA and GA produce comparable results for the flowshop sequencing problem for most sizes and types of problem, but that GA will perform relatively better for large problems, and that it will reach a near-optimal solution rather more quickly. This is encouraging, for Ogbu and Smith [9] regard SA as out-performing all other heuristics.

Moreover, it should be pointed out that the Osman/Potts SA heuristic is the result of a substantial amount of experimentation in order to arrive at the parameter settings recommended. No experimentation of this kind has as yet been done on the GA heuristic reported here; the parameters were simply set at what seemed "sensible" values after some rather small-scale experiments on one parameter at a time using some 20/5 problems. One of the characteristics of GAs is their robustness in respect of parameter settings, so that similar performance might be expected for a wide range of choices of population size, mutation rate and so on. Nevertheless, there is some experience (e.g. Grefenstette [18]) that would suggest that worthwhile gains may be achieved by trying to optimize in parameter space. And of course, it is possible that another type of GA might do even better; in particular it should be pointed out that this implementation is a "pure" GA—i.e. no problem-specific information is used. Davis [19] has recently suggested that "hybrid" GAs where use is made of such information may perform better. All these factors are being investigated.

Furthermore, GAs lend themselves very well to parallel implementation: there is no reason why the evaluation of new chromosomes needs to be done sequentially. This would offer further speed-up in solution times, and a parallel version of the GA described above is also being explored.

In conclusion, Genetic Algorithms provide a variety of options and parameter settings which still have to be fully investigated. This research has demonstrated the potential for solving machine-sequencing problems by means of a Genetic Algorithm, and it clearly suggests that such procedures are well worth exploring in the context of solving large and difficult combinatorial problems.

REFERENCES

1. M. R. Garey and D. S. Johnson, *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco (1979).
2. A. H. G. Rinnooy Kan, *Machine Scheduling Problems: Classification, Complexity and Computations*. Martinus Nijhoff, The Hague (1976).
3. J. H. Holland, *Adaptation in Natural and Artificial Systems*. Univ. of Michigan Press, Ann Arbor (1976).
4. L. Davis (Ed.), *Genetic Algorithms and Simulated Annealing*. Pitman, London (1987).
5. S. Kirkpatrick, C. D. Jelatt and M. P. Vecchi, Optimisation by simulated annealing. *Science* **220**, 671–679 (1983).
6. P. J. M. van Laarhoven and E. H. L. Aarts, *Simulated Annealing: Theory and Applications*. Kluwer, Dordrecht (1987).
7. F. A. Ogbu and D. K. Smith, The application of the simulated annealing algorithm to the solution of the $n/m/C_{max}$ flowshop problem. *Computers Ops Res.* **17**, 243–253 (1990).
8. I. H. Osman and C. N. Potts, Simulated annealing for permutation flow-shop scheduling. *Omega* **17**, 551–557 (1989).
9. F. A. Ogbu and D. K. Smith, Simulated annealing for the permutation flow-shop problem. *Omega* **19**, 64–67 (1991).
10. D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison–Wesley, Reading, Mass. (1989).
11. C. R. Reeves, An introduction to genetic algorithms. In *Operational Research Tutorial Papers* (Edited by A. G. Munford and T. C. Bailey). Operational Research Society, Birmingham (1991).
12. D. H. Ackley, An empirical study of bit vector function optimisation. In *Genetic Algorithms and Simulated Annealing* (Edited by L. Davis), pp. 170–204. Pitman, London (1987).
13. M. Nawaz, E. E. Emscore Jr and I. Ham, A heuristic algorithm for the m-machine, n-job flow-shop sequencing problems. *Omega* **11**, 91–95 (1983).
14. A. D. Amar and J. N. D. Gupta, Simulated versus real life data in testing the efficiency of scheduling algorithms. *IIE Trans.* **18**, 16–25 (1986).
15. C. R. Reeves, Improvements to the NEH heuristic for flowshop scheduling. Coventry Polytechnic Working Paper SOR91/2 (1991).
16. E. Taillard, Some efficient heuristic methods for the flow shop sequencing problem. *Eur. J. Opl Res.* **47**, 65–74 (1990).
17. E. Taillard, Benchmarks for basic scheduling problems. *Eur. J. Opl Res.* In press.
18. J. J. Grefenstette, Optimization of control parameters for genetic algorithms. *IEEE Trans. Syst. Man Cybernet.* **SMC-16**, 122–128 (1986).
19. L. Davis (Ed.), *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York (1991).

APPENDIX

A pseudo-code version of the version of the GA used to obtain the results reported above is given below.

(* set parameters *)

$M := 30$;
$P_c := 1.0$;

```
P_m^init := 0.8;
θ := 0.99;
D := 0.95;
(* select_initial_poppulation *)
generate(NEH_sequence);
evaluate(NEH_sequence);
pop_no. := 1;
repeat
     pop_no. := pop_no. + 1;
     generate(random_sequence);
     evaluate(random_sequence);
until pop_no. := M;
(* this creates an initial population of size M*)
sort(population);
calculate(population_statistics);
P_m := P_m^init;
repeat
     if random_no. < P_c then
     begin
          select(parent_1) using fitness_rank distribution;
          select(parent_2) using uniform distribution;
          choose(crossover_poin);
          crossover;
     end;
     if random_no. < P_m then mutate;
     evaluate(new_sequence);
     select(old_sequence) from unfit_members;
     delete(old_sequence) from population;
     insert(new_sequence) into population;
     update(population_statistics);
     P_m := θP_m
     if v_min/v_mean > D then P_m := P_m^init
unit cpu_time > max_cpu_time;
```