A Promising genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling, and Open-Shop Scheduling Problems

Hsiao-Lan Fang, Peter Ross,
and Dave Corne

DAI Research Paper No. 623

Appears in: Proceedings of the Fifth International Conference on Genetic Algorithms, S. Forrest (ed.), San Mateo: Morgan Kaufmann, 1993, pages 375–382.

A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling, and Open-Shop Scheduling Problems

Hsiao-Lan Fang, Peter Ross, Dave Corne
Department of Artificial Intelligence
University of Edinburgh
Edinburgh, UK
Email: {hsiaolan,peter,dave}@aisb.edinburgh.ac.uk

Abstract

The general job-shop scheduling problem is known to be extremely hard. We describe a GA approach which produces reasonably good results very quickly on standard benchmark job-shop scheduling problems, better than previous efforts using genetic algorithms for this task, and comparable to existing conventional search-based methods. The representation used is a variant of one known to work moderately well for the traveling salesman problem. It has the considerable merit that crossover will always produce legal schedules. A novel method for performance enhancement is examined based on dynamic sampling of the convergence rates in different parts of the genome. Our approach also promises to effectively address the open-shop scheduling problem and the job-shop rescheduling problem.

1 INTRODUCTION

The job-shop scheduling problem (JSSP) is a very important practical problem. Efficient methods of solving it can have major effects on profitability and product quality, but with the JSSP being among the worst members of the class of NP-complete problems (Gary & Johnson 1979) there remains much room for improvement in current techniques. In general, the difficulty of the general JSSP makes it very hard for conventional search-based methods to find near–optima in reasonable time. This has led to recent interest in using genetic algorithms (GAs) to address these problems.

In the general JSSP, there are j jobs and m machines; each job comprises a set of tasks[1] which must each be done on a different machine for different specified processing times, in a given job-dependent order. Eg:, table 1 shows a standard 6×6 benchmark problem (*ie*, $j = 6, m = 6$), from (Muth & Thompson 1963). In

	(m,t)	(m,t)	(m,t)	(m,t)	(m,t)	(m,t)
Job 1:	3,1	1,3	2,6	4,7	6,3	5,6
Job 2:	2,8	3,5	5,10	6,10	1,10	4,4
Job 3:	3,5	4,4	6,8	1,9	2,1	5,7
Job 4:	2,5	1,5	3,5	4,3	5,8	6,9
Job 5:	3,9	2,3	5,5	6,4	1,3	4,1
Job 6:	2,3	4,3	6,9	1,10	5,4	3,1

Table 1: The 6x6 benchmark problem

this example, job 1 must go to machine 3 for 1 unit of time, then to machine 1 for 3 units of time, and so on. A legal schedule is a schedule of job sequences on each machine such that each job's task order is preserved, a machine is not processing two different jobs at once, and different tasks of the same job are not simultaneously being processed on different machines. The problem is to minimise the total elapsed time between the beginning of the first task and the completion of the last task (the *makespan*). Other measures of schedule quality exist, but shortest makespan is the simplest and most widely used criterion. For the above problem the minimum makespan is known to be 55, as in, for example, the schedule shown in figure 1 .

There are two similar benchmarks, of sizes 10×10 and 20×5. The best results on these benchmarks for traditional (B & B – branch & bound search) and GA methods published so far are shown in table 2[2]

The branch & bound method (*eg*: see (Carlier & Pinson 1989)) produces good results but takes considerable computer time even for the 10×10 problem because of the significant amount of schedule generation implicit in the method. (Davis 1985) was the first to suggest and demonstrate the feasibility of using a

[1] Note: what we call a "task" is often called an "operation" in the JSSP literature.

[2] Adapted from (Nakano 1991).

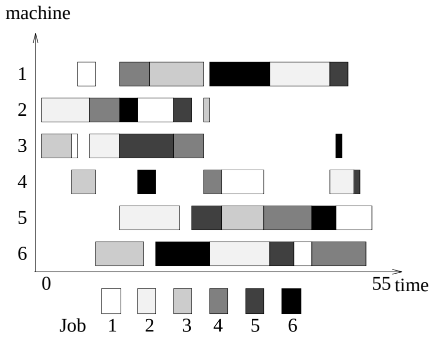

Figure 1: An optimal schedule for 6×6 JSSP benchmark

Paper	Method	6x6	10x10	20x5
McMahon 75	B & B	55	972	1165
Baker 85	B & B	55	960	1303
Carlier 89	B & B	55	930	1165
Nakano 91	GA	55	965	1215

Table 2: Some published benchmark results

GA on a simple JSSP, employing an essentially *ad-hoc* set of genetic operators and a memory-intensive chromosome representation, paving the way for future improvements. Meanwhile, the general success of GAs on other kinds of hard scheduling problems, such as the traveling salesman problem (TSP), started to lead to clues for more effective representations and operators for GA approaches. *Eg*: (Whitley *et al* 1989) defined a new edge recombination operator for the TSP, although noted that performance degraded when applied to more typical scheduling problems; (Bagchi *et al* 1991) used problem-specific information in the representation and genetic operators, addressing a limited form of JSSP in which certain batches of tasks must be scheduled continuously. More recently, (Nakano 1991) used a conventional (binary) GA for the JSSP, supplemented with algorithms for interpreting and repairing genomes, and was successful in improving on the performance of some previously reported branch & bound search methods on benchmark problems, though did not improve on the best results found with these methods.

Our approach uses a variant of the ordinal representation introduced in (Grefenstette *et al* 1985) and used for the TSP. This representation has the considerable merit of producing only legal schedules under crossover and mutation. When applied to the JSSP, it produces better results than those of (Nakano 1991) with pleasingly small computational effort, and thus provides a convenient way to handle the rescheduling problem too. The rescheduling problem involves modifying a schedule in process of execution in order to take account of changed, canceled or new jobs. Because this sort of thing happens frequently in the kind of organisation that has to deal with JSSPs, it is as important to find efficient rescheduling algorithms (which hopefully don't involve rebuilding the schedule from scratch) as it is to find effective algorithms for the full JSSP.

2 OVERVIEW

In section 3 we describe our encoding technique, and outline the basic activities of the schedule builder which performs the interpretation of a genome for the JSSP. In section 4 we go on to discuss the application of this approach to *Open-Shop* scheduling, and outline the more sophisticated schedule builder we employ in this latter case. In section 5 we briefly describe the job-shop rescheduling problem, and how it can be addressed via our approach. In section 6 we go on to discuss the qualitative GA dynamics which arise from the representation we use, making points in particular about the redundancy of the representation, and the variation in convergence rates for different genes (or 'chunks' of the genome). This leads us towards introducing a method for combating premature convergence in general GA applications that involve significant variation in gene convergence rates, which is discussed further in section 7. Section 8 presents some basic results: concerning the performance of our basic approach on two benchmark JSSPs, showing how this approach outperforms previously reported GA attempts at this task which we know of; concerning the performance of our basic approach, enhanced by 'gene-variance-based operator targeting', showing improvement on the initial unenhanced results; and concerning performance on a selection of benchmark open-shop-scheduling problems, showing how our approach comes within a few percent (sometimes 0%) of the optimal or best-known solutions for the problems tried. We know of no GA-based efforts on the OSSP with which to compare, so we present these latter results in order to show the potential for a GA approach to open-shop scheduling, and invite fellow GA researchers to experiment with the same problems. At the end of this section, we describe how to obtain the problem definitions for the benchmarks used in this paper. Finally, section 9 summarises our results and discusses the general approach and further work.

3 THE REPRESENTATION

The genotype for a $j \times m$ problem is a string containing $j \times m$ chunks, each chunk being large enough to hold the largest job number (j). A chunk is atomic as far as the GA is concerned. It provides instructions for building a legal schedule as follows: the string of chunks $abc\cdots$ means: put the first untackled task of the a-th uncompleted job into the earliest place where it will fit in the developing schedule, then put the first untackled task of the b-th uncompleted job into the earliest place where it will fit in the developing schedule, and so on. The representation can be seen to encode all active schedules, and also lends itself to obvious extensions which would enable the encoding of necessary or unnecessary delays on machines. The task of constructing an actual schedule is handled by a schedule builder which maintains a circular list of uncompleted jobs and a list of untackled tasks for each such job. Thus the notion of "a-th uncompleted job" is taken modulo the length of the circular list to find the actual uncompleted job. Note: instead of employing a circular list, (Grefenstette *et al* 1985) constrains alleles of the i-th chunk to range from 1 to $N-i+1$ in value; it is unclear how to directly extend this technique to a JSSP (with more than one machine), hence our use of a circular list.

The schedule builder is straightforward and computationally cheap. It must consider four cases when slotting a task into a developing schedule. For instance, suppose it is asked to slot job 1 into machine 2, with processing time 2. If there is a suitable gap in the schedule for machine 2, it may be possible to fit the task in there with or without compulsory idle time. If no suitable gap exists, that task has to be added to the end of the machine's schedule with or without compulsory idle time. Figure 2 shows the choices. The

```
 With suitable gap,
 idle time needed:   ... not needed:
mc1:.. 11333        mc1:.. 114444
mc2:.. 2 ## 3333    mc2:.. 222## 44

  No suitable gap,
 idle time needed:   ... not needed:
mc1:.. 555111       mc1:.. 66611
mc2:.. 22 5  ##     mc2:.. 22 666##
```

Figure 2: Scheduler builder choices of task placement

symbol "##" shows where the schedule builder would place the task in each case.

4 OPEN-SHOP SCHEDULING

The Open-Shop Scheduling Problem (OSSP) is similar to the JSSP, with the exception that there is no *a priori* ordering on the tasks within a job. The OSSP has a considerably larger search space than the JSSP, and seems to be less heavily addressed in the literature, although it is an important and ubiquitous problem, occurring in any job-shop situation in which tasks for a particular job may be carried out in (almost) any order, such as automotive repairs (tasks) for cars (jobs), or upgrades/repairs (tasks) for PCs (jobs).

Table 3 shows a standard 5×5 benchmark OSSP (that is, $j=5, m=5$) taken from (Beasley 1990). In the

	(m,t)	(m,t)	(m,t)	(m,t)	(m,t)
Job 1:	4,85	1,64	3,31	5,44	2,66
Job 2:	1,7	4,14	2,69	5,18	3,68
Job 3:	4,1	1,74	2,70	5,90	3,60
Job 4:	2,45	4,76	5,13	3,98	1,54
Job 5:	1,80	4,15	2,45	5,91	3,10

Table 3: A 5x5 benchmark OSSP

above example, task 1 of job 1 must go to machine 4 for 85 units of time, task 2 of job 1 must go to machine 1 for 64 units of processing time, and so on, with no restrictions on the order in which the tasks for any job are to be processed. The problem is to generate a valid schedule with minimal makespan. Figure 3 shows a minimum-makespan (300) schedule for the benchmark in table 3.

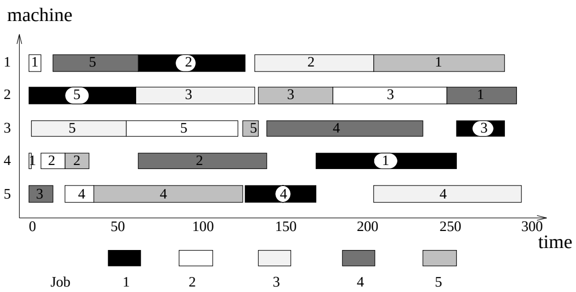

Figure 3: Minimal-makespan schedule for a 5x5 OSSP benchmark

The basic extension of the representation described in section 3 to the OSSP involves a genome $abcd$... meaning: put the ath untackled task of the bth uncompleted job into the earliest place it will fit in the developing schedule, put the cth untackled task of the dth uncompleted job into the earliest place it will fit in the developing schedule, and so on. Whereas previously, for the JSSP, the 'first untackled task' for any particular job was always predetermined owing to the *a priori* ordering on tasks, in this case we need to incorporate an extra gene for each job to encode which of the remaining tasks for a job to choose (since with no predetermined ordering, any may be chosen).

An alternative is to use precisely the same representation as for the JSSP, but change the interpretation of abc... to: heuristically choose an untackled task from the ath uncompleted job and place it in the earliest place it will fit in the developing schedule, heuristically choose an untackled task from the bth uncompleted job and and place it in the earliest place it will fit in the developing schedule, and so on. In this case, at each step the schedule builder looks ahead to find the earliest available slot(s) in the developing schedule into which a non-empty set of tasks from the current job can be placed. If lookahead determines that more than one equally early slots are available, then a simple heuristic is used to choose which task to actually place in which slot. Two simple heuristics we have used are: (a) choose randomly from the available tasks; (b) choose the task with the largest processing time. The random method seems to work best on small problems, but best results are found on larger problems with the "largest-first" heuristic. In general, this lookahead/heuristic method for the OSSP works better than the basic extension to the JSSP approach described in the above paragraph.

5 JOB-SHOP RESCHEDULING

Job-shops are beset by the continual need to alter previously worked out schedules in the light of problems which arise. This typically means revising the expected processing time for some job in the schedule, or revising (typically delaying) the start time for a particular task. There is thus a need for efficient methods of rescheduling. If work has not yet begun on the current schedule, then an obvious and simple approach to rescheduling would be to rerun the schedule-finding program (eg: in this case, a GA) from scratch on the changed data. Strict rescheduling, however, means *not* scheduling the entire problem from scratch; rescheduling is thus strictly necessary when either there is not enough time to be able to schedule from scratch, or when part of the current schedule is already in progress. A proper rescheduling method would be to re-use some of the work already done in finding the previous schedule. This might involve augmenting the previous schedule with the new change, and iteratively modifying it until it is acceptable. Another method would be to recover a new, smaller scheduling problem made up from all and only those parts of the previous schedule that are affected by the change.

Rescheduling from scratch is obviously to be avoided in the light of the large processing time required for large problems and the frequency of the need to reschedule. Also, sophisticated use of previous work is very difficult to achieve with a typical GA (although see (Louis *et al* 1993) for a recent attempt at storing schema information in a case base). Nominal use of previous work done could involve seeding; we have not yet tried this. Our representation and schedule builder, however, lend themselves naturally to a method in which we make a smaller scheduling problem, via a simple dependency analysis which finds out which tasks are affected by the changes.

Two kinds of situation are dealt with: a change in the *processing time* of some task (which includes the case of removing a task entirely), and a change in the *start time* of some task (if, for example, a task must be delayed because of problems with a machine or delays in obtaining resources). Input to the rescheduler is simply the genome representing the schedule which must be altered. The user then enters the required modification (to the processing time and/or start time of one or more tasks). With reference to figure 1, suppose we need to increase the processing time of the machine2 task of job1. A simple dependency analysis discovers that the affected tasks are those that occur later in the schedule on machine 2 (as well as the changed task itself), as well as the machine 4, 5 and 6 tasks of job 1. Recursively, other affected tasks are found for each of the initially affected tasks until the complete set of affected tasks is found. Along with values from the previous schedule which contain new available start times for each machine, this set of affected tasks constitutes a reduced JSSP which can be solved by the GA much more quickly than fully rescheduling from scratch. A similar dependency analysis and reduced JSSP formulation is done for the case in which a task's earliest possible start time is shifted.

This method does not guarantee an optimal new schedule; the GA, of course, never guarantees optimality anyway, but the point is that the retention of a fixed (unaffected) portion of the previous (near) optimal schedule might preclude the discovery of an optimal schedule which might otherwise be possible to find by rescheduling from scratch. The strength of this rescheduling method, however, lies in its speed. There is thus a tradeoff between the speed in which a good new schedule can be found via retaining parts of the previous schedule, and the potential advantage of rescheduling from scratch with the (probably low) possibility of evolving a significantly better schedule. Experiments are underway to quantitatively analyse this tradeoff.

6 PERFORMANCE ENHANCEMENTS

On hard problems like the JSSP, GA researchers routinely need to use either problem-specific or problem-type specific performance enhancements to improve performance. These enhancements are interesting because of the light they shed on the dynamics of the GA approach and the aspects of problems which make it hard or easy for GAs to solve them. For example, Nakano's representation is highly redundant (with $2^{mj(j-1)/2}$ genomes representing approximately $j!^m$ distinct schedules) and so leads to the possibility of false competition among genotypes, in which different representations of the same schedule compete against one another, possibly to yield inferior descendants which combine aspects of their parents' representations which do not translate into good building blocks. There is, in fact, very little chance (but see below) of two representations of the same schedule competing in early generations — although there may be a huge number of possible representations of the same schedule, this number is entirely swamped by the number of distinct schedules. However, false competition will still be manifest with different representations of the same *building block* or, to be more correct, the same *forma*. A forma (Radcliffe 1990) can be viewed as any dimension along which two genomes are equivalent. False competition will then be relevant if the schemata in the representation do not directly coincide with the formae which (intuitively) represent the important building blocks; this is typically the case in sophisticated GA applications. Eg; in our case, the forma: "schedules in which the machine2 task of job 1 is scheduled before the machine2 task of job 2" may well be a good building block (ie: have high average fitness), but, since it does not correspond to a particular schema, two schedules which are instances of this forma may well recombine to produce children which are not.

Nakano partially combats false competition with *forcing*, in which he replaces illegal genotypes in the pool with their 'nearest' legal matches. This forces a one-to-one genotype/schedule mapping in a gene pool, eliminating false competition in the selection step (although still typically resulting in illegal schedules after crossover). Nakano hence uses a highly redundant representation with a complex evaluation technique for the basic GA, and then significantly improves performance by using forcing to reduce false competition. Our approach does not require forcing, since the representation always encodes legal schedules, but there is high redundancy (though less high than Nakano's), and we similarly need a way of countering false competition.

Our choice of representation is highly context sensitive, and leads to front parts of the genotype converge more quickly than later parts. This seems to happen because schemata defined early in the genome correspond more precisely to good formae; that is: a *schema* such as 1,2,□,□,..., *always* corresponds to the forma "first schedule the first task of job1, and then the first task of job2". If it so happens that this forma has high fitness, then this schema will have high fitness. However, schemata defined later in the genotype, such as □,□,□,□,□,□,□,2,□,3, are likely to represent radically different formae in different genomes (contexts) — the sampled mean fitness of such a schema will thus tend towards the mean fitness of the population as a whole. Hence, high-fitness schemata will only be found early in the genome, and these will converge first (providing a 'context' which then leads to high fitness schemata being found a little later in the genome, and so on).

False competition thus leads to differing convergence rates for schemata across the genotype. This effect actually rises quite sharply towards the tail of the genotype owing to the fact that as the context becomes set by convergence in the rest of the genome, the j alleles of any tail-end gene are 'competing' for, and thus multiply representing, fewer and fewer unscheduled tasks.

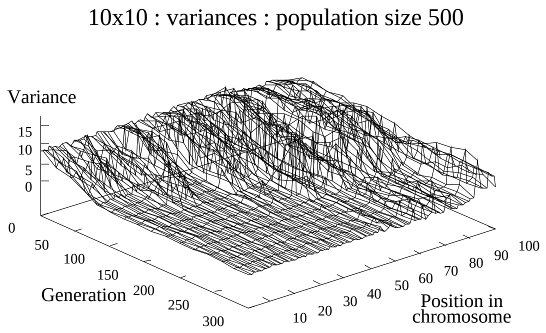

Figure 4: Plot of variance of chunks of the genome with time, and with genome position, on the 10×10 JSSP.

We can visualise the overall effect of this in figure 4, in which we can clearly see gradually decreasing convergence speed as we traverse the chromosome from left to right. This figure shows a plot of the variance of each chunk of the genotype within the pool (size 500) with its position in the genotype, and with generation as the GA operates, for 300 generations of a run on the 10×10 benchmark JSSP. As figure 4 shows, gene convergence rates fall fairly smoothly as a function of position in the genotype. This kind of behaviour should be typical of GA problems where the representation, for whatever reason, is such that there is a variation in 'significance' across the genotype. In the JSSP case, in which large scale problems not only cost significant computational time, but in which the solutions produced might significantly affect profits and/or product quality, we should be able to exploit this effect by using it to inform ways of increasing overall convergence speed and/or solution quality. In section 7, we describe a gene-variance based operator targeting strategy, which is a principled first attempt at doing just this, by making sure that genetic operators are concentrated where and when they seem to be most 'needed'. This initial attempt has led to significant improvement in solution quality.

7 GENE-VARIANCE BASED OPERATOR TARGETING

The situation in figure 4 suggests a strategy to improve solution quality. First, the faster stabilisation of early parts of the genome suggest premature convergence. This is because the fast converging early schemata may not have been adequately tested in the context of good formae that may be (partly) encoded later in the genome. Increasing mutation rates at fast converging sites may thus improve performance; also, this measure should obviate 'wasted' mutation in later, slow-converging parts of the schedule which are still in relatively early stages of exploration. Second, we can expect crossover at early, more stable positions to have minimal effect on sampling adequacy, since this leads only to re-examining schemata over and over again in similar contexts. So, encouraging crossover more at later, less stable positions should lead to more effective exploitation. On the whole, it would seem a good idea to increase the extent to which schemata are effectively sampled in new contexts, in proportion to the degree to which the GA seems 'unsure' about them. Conversely, it would seem a good idea to increase the extent to which new schemata are explored (via mutation) , in proportion to the extent to which schemata defined at the same positions have already been (perhaps prematurely) converged to.

A way of implementing these effects is what we term *gene-variance based operator targeting* (GVOT). This works by measuring the diversity of genes at each position of the genotype in a pool (in our experiments, we do this by sampling statistical variance after every ten generations), and choosing the actual point of crossover or mutation via roulette-wheel selection based on these variances. Sites for N-point uniform crossover are selected probabilistically but according to the square of chunk variance, while order-based mutation positions are selected according to the inverse of chunk variance. Hence, high variance sections are more likely to be chosen for crossover; low variance sections for mutation.

This can be seen as a specific instance of an idea which should be of more general use in GA performance enhancement on hard problems, particularly where there is a significant variance in convergence rates at different sites in the genotype. In many other kinds of problem we can't expect smooth changes in variance across the genotype; this would not occur in the JSSP, for instance if (unusually) task processing times were to grow as a function of advancing position in the job sequence. However, whenever significant variation in convergence rate does occur (smooth or not), the GVOT strategy, targeting operators solely on the basis of dynamically sampled variance, should work just as well.

This performance enhancement method complements those discussed in, for example, (Booker 1987) and (Eshelman & Schaffer 1991), which present ways of improving performance by, eg, encouraging recombination between adequately 'different' genomes (incest prevention), and avoiding wasted crossover operations by only recombining the 'reduced surrogate' of two parents (the smaller genome made up of those sites at which the parents are different). There are complex interactions between such methods and GVOT. Roughly speaking, GVOT slows down convergence of otherwise fast-converging schemata in order to wait for other schemata to catch up, while encouraging vigorous recombination to more effectively test the latter; incest prevention in conjunction with reduced-surrogate recombination, on the other hand, will partially reproduce this effect to the extent that less converged schemata will be more likely to be present in the reduced surrogates of parents which are far enough apart to sanction recombination. The latter method, however, does not 'slow down' fast-converging schemata (which GVOT does via targeting mutation at fast-converging sites). We intend extensive experimentation to tease out the relative effectiveness of these methods in conjunction with, and other than, GVOT on problems with highly context sensitive genome representations. Our feeling is that GVOT, owing to the direct selective targeting of operators according to convergence rates, will be more and more effective the more varied the schemata convergence rates are in the application.

GVOT is less effective (though still produces better results), for example, with the representations we discuss above for the OSSP. This is because the plot analogous to figure 4 for the OSSP is rather more flat; because of much higher epistasis in the OSSP case (low-variance highly fit schemata only begin to occur at relatively long defining lengths), schemata sampled in earlier generations have a less significant advantage over others than in the JSSP case, and hence there is reduced variation in convergence rates.

8 RESULTS

The JSSP results below all involve population sizes of 500, using rank-based selection with elitism and a fixed crossover rate, running for 300 generations (unless otherwise specified), hence involving 150,000 evaluations. The comparative figures for Nakano involve the same number of evaluations, though based on 1,000 generations with populations of size 150. The raw fitness of a chromosome was taken to be the makespan of the schedule it represents. The OSSP results similarly involve rank-based selection with elitism but use adaptive crossover and runs of 1,000 generations. The two smaller OSSPs were tackled with populations of size 100, while the rest were tackled with populations of size 200. We found that results did not vary significantly across changes in crossover rate and adaptation regime. The reported JSSP experiments used a crossover rate of 0.6 and adaptive mutation (starting at 0, rising by 0.001 per generation), while the OSSP experiments use adaptive crossover (starting with p_C at 0.6, falling by 0.002 per generation, with a limit of 0.2) and adaptive mutation at $1 - p_C$. Typically, order-based mutation (swap alleles between two randomly chosen genes) was used. For the OSSP, the mutation rate was the probability of mutating a genome; so, for example, where p_M (ie: $1 - p_C$) was 0.6, this roughly translates to a bit-mutation rate of, for example, 0.012 for the 10×10 OSSP (divide by half the genome length).

GVOT involves calculating a measure of the diversity of alleles of a gene (or chunk of genes) within a population. We are still experimenting to find the most suitable measure of this diversity. Both JSSP-with-GVOT and OSSP-with-GVOT results use statistical variance of the numerical value of the alleles as a simple approximation to this measure; we are also investigating the use of allele entropy as a more well-founded information-theoretic measure of the diversity of alleles. In addition, we are experimenting with different ways of using the diversity measure to target operators. For the JSSP with GVOT, the method we used was roulette-wheel selection of crossover points based on variance (mutation sites based on inverse variance). For the OSSP with GVOT, we employed what we term *multiform* crossover, in which the probability of swapping genes between parents at a particular site is adjusted (from the normal 0.5, for uniform crossover) in accordance with the relative variance at that site.

Our main results are that we have been able to find better solutions on benchmark JSSPs than previous GA-based methods and have thus closed the gap somewhat between GA-based approaches and the best solutions so far found with branch & bound search.

In the two following tables, 'average' figures refer to the mean result over 10 trials; these are not available for Nakano's technique. Also, Nakano's 'without-forcing' result on the 10 × 10 benchmark is read from a graph in (Nakano 1991), hence our estimated error margin.

Table 4 summarises our results *without* gene-variance based operator targeting (GVOT), compared with Nakano's results (where available) *without* forcing, showing how, the representation we describe leads to better results when false competition is highly evident in both approaches[3].

	10×10	20×5
Average sol'n without GVOT (Fang *et al*)	985	1225
Best sol'n without GVOT (Fang *et al*)	960	1213
Best sol'n without forcing (Nakano 91)	1160(±10)	—

Table 4: Our approach vs. Nakano's, without GVOT

With performance-enhancements in place, our results using GVOT are compared with Nakano's results using forcing in table 5. It can also be noted that our best solutions without GVOT are marginally better than Nakano's *with* forcing.

	10×10	20×5
Average sol'n with GVOT (Fang *et al*)	977	1215
Best sol'n with GVOT (Fang *et al*)	949	1189
Best sol'n with forcing (Nakano 91)	965	1215

Table 5: Our approach vs. Nakano's, with GVOT

Although improvement in solution quality is modest as a percentage (though significant considering that these solutions may be very close to optimal anyway), the real advantage of our technique over previous GA methods is the combination of its apparent promise and the straightforwardness of applying it (arising from the absence of any need to repair invalid genomes). We also feel that it significantly improves on other techniques in terms of computational complexity, though unfortunately we cannot yet provide more quantitative results with regard to comparative speed because of a lack of available figures for comparison; however we can report that experiments on the 10×10 JSSP take less than 25 minutes of CPU time and the 20×5 JSSP less than 30 minutes, with our experiments implemented in C and run on a Sun-4 (without using GVOT, CPU time drops by about 30%).

We also experimented with one-point *vs* uniform crossover, and adaptive *vs* fixed order-based mutation rates. The graphs in 5 show our results on the 10×10 and 20×5 JSSP benchmarks, comparing different GA variants. *Fixed-1P* employs a fixed mutation rate per chromosome of 0.05 and one-point crossover; *one-point* employs a mutation rate per chromosome which begins at 0 and increases by 0.001 in each generation (stopping at 0.1); *uniform* employs the same adaptive mutation strategy as *one-point* and uses uniform crossover; finally, *GVOT* is as *uniform*, except for the use of N-point uniform crossover (where N is half the genome length) with GVOT.

Initial experiments with a (pseudo-)parallel GA with migration every 20 generations show improved average solution quality, as do experiments with larger population sizes (though obviously at the expense of time); but more work is needed properly to investigate and quantify these aspects.

Our initial results for a set of benchmark open shop scheduling problems are shown in table 6. Results for the two smaller problems were obtained with the 'break-ties-randomly' heuristic, while the results for the larger problems were obtained with the 'largest-

[3]It is difficult for us to quantitatively compare our with approaches other than Nakano's since we have not yet found other reported GA approaches which use the benchmarks.

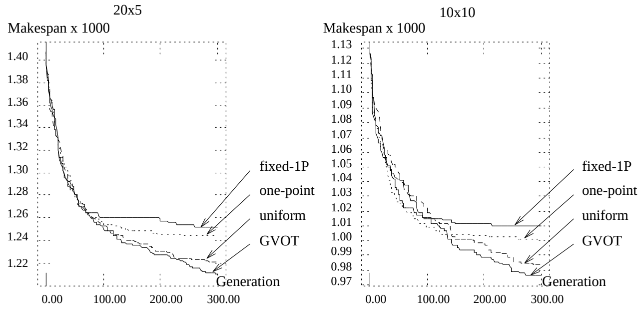

Figure 5: Relative performance of different variants on the 10×10 and 20×5 JSSP benchmarks

first' heuristic. For the two smaller problems and two larger problems, 'Best Known' is the optimal solution; for the rest, it is the best known solution. All OSSP experiments involved use of GVOT, which produced reliably better results than without GVOT, though less markedly so than with the JSSP.

OSS Problem	Best Known	Results: mean/best
4×4	193	193 / 193
5×5	300	302.2 / 300
7×7	438	447.1 / 439
10×10	645	679.5 / 669
15×15	937	980.0 / 969
20×20	1155	1235.1 / 1213

Table 6: Results on benchmark OSSPs

The JSSP benchmark problems used in this paper can be obtained from (Muth & Thompson 1963). The OSSP problems referred to in table 6 can be obtained via (Beasley 1990). The OR library referred to in the latter article is an electronic library from which may be obtained benchmarks for a wide range of OR problems. These are distributed in the form of Pascal code which generates the problems. Researchers wishing to compare with our results will need to know that the problems referred to in table 6 are each the *problem No. 1* of their specified size. Alternatively, problem data may be obtained directly from us.

9 CONCLUSIONS AND FURTHER WORK

We present a promising new representation for GA approaches to the JSSP, and described novel techniques for analysing the GA dynamics in terms of the variation in gene variance across the genotype, and targeting operator positions according to dynamically sampled measures of gene convergence rates. Our approach improves on the results obtained from other GA methods we know of, and brings us closer to closing the gap in solution quality between the best solutions found by branch & bound search and those found by GA approaches so far.

The approach also conveniently handles rescheduling in the job-shop problem, and seems promising for application to the open shop scheduling problem. More tests are needed, however, before we can report a thorough comparison of our method against other techniques, and before we can determine the efficacy of our method when applied to real-world problems (benchmark problems are unrepresentative of the true difficulty of the general JSSP; the same might also be true of most real-life JSSPs!). In this vein, further work is under way to more thoroughly test the performance of our technique on the benchmarks, and on a set of real world JSSPs which we are planning to collate.

Finally, we hope to have shown further promise for GA-based approaches to job-shop problems, and hope and expect that further improvements will be reported (by us and others) via the use of various problem-specific heuristic improvements, as well as via approaches based on different genome representations. For example (Grefenstette 1987) discusses the general idea of incorporating problem-specific knowledge into various parts of the GA, while (Beasley *et al* 1993), describes a GA approach to combinatorial problems based on an epistasis reducing representation, which may be of use for the JSSP.

Acknowledgments

We gratefully acknowledge the constructive criticism and comment received from three anonymous referees, from which this paper has benefited, and we hope that we have satisfactorily accommodated their concerns and suggestions. Many thanks also to the China Steel Corporation, Taiwan, R.O.C., for financial support of Hsiao-Lan Fang.

References

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

D. Beasley, D. R. Bull, & R. R. Martin (1993). Reducing Epistasis in Combinatorial Problems by Expansive Coding. In S. Forrest (ed), *Genetic Algorithms; Proceedings of the Fifth International Conference (GA93)*, Morgan Kaufmann, San Mateo, CA.

J. E. Beasley (1990). OR-Library: Distributing test problems by electronic mail. In *Journal of the Operational Research Society*, Vol 41, pp. 1069–1072.

L. Booker (1987). Improving Search in Genetic Algorithms. In L. Davis (ed) *Genetic Algorithms and*

Simulated Annealing, San Mateo: Morgan Kaufmann, 1987, pages 61–73.

J. Carlier & E. Pinson (1989). An algorithm for solving the job-shop problem. In *Management Science*, 35(2):164–176, February 1989.

L. Davis (1985). Job shop scheduling with genetic algorithms. In J. J. Grefenstette, (ed) *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 136–140. San Mateo: Morgan Kaufmann, 1985.

L. J. Eshelman & J. D. Schaffer (1991). Preventing Premature Convergence in Genetic Algorithms by Preventing Incest. In R. K. Belew & L. B. Booker (eds) *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo: Morgan Kaufmann, 1991, pages 115–120.

M. R. Gary & D. S. Johnson (1979) *Computers and Intractability: a Guide to the Theory of NP-Completeness*. Freeman.

J. J. Grefenstette, R. Gopal, B. Rosmaita, & D. Van Gucht (1985). Genetic algorithms for the traveling salesman problem. In J. J. Grefenstette (ed) *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 160–168. San Mateo: Morgan Kaufmann, 1985.

J. J. Grefenstette (1987). Incorporating Problem-Specific Knowledge into Genetic Algorithms. In L. Davis (ed) *Genetic Algorithms and Simulated Annealing*, San Mateo: Morgan Kaufmann, 1987, pages 43–60.

S. Louis, G. McGraw, & R. O. Wyckoff (1993). Case-based reasoning assisted explanation of genetic algorithm results. *J. Expt. Theor. Artif. Intell.*, Vol. 5, 1993, pp. 21–37.

J. F. Muth & G. L. Thompson (1963). *Industrial Scheduling*. Prentice Hall, Englewood Cliffs, New Jersey, 1963.

R. Nakano (1991). Conventional Genetic Algorithms for Job-Shop Problems. In R. K. Belew & L. B. Booker (eds) *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo: Morgan Kaufmann, 1991, pages 474–479.

N. J. Radcliffe (1990). Equivalence Class Analysis of Genetic Algorithms. In *Complex Systems* Volume 5, No. 2, pp.183–205, 1990.

D. Whitley, T. Starkweather, & D'Ann Fuquay (1989). Scheduling problems and traveling salesmen: The genetic edge recombination operator. In J. D. Schaffer (ed) *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, pages 133–140. San Mateo: Morgan Kaufmann, 1989.