The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination

Darrell Whitley Timothy Starkweather Daniel Shaner

Chapter 1

1.1 INTRODUCTION

Scheduling poses a difficult problem in numerous application areas. Realistic scheduling problems involve constraints that often cannot be precisely defined mathematically. As a result, traditional optimization methods are not always applicable for many scheduling and sequencing problems or are not as effective as we would like. Genetic algorithms not only offer a means of optimizing ill-structured problems, but also have the advantage of being a global search technique.

We have developed a genetic operator that generates high-quality solutions for sequencing or ordering problems; the performance of this operator is all the more remarkable because it does not require any heuristic or local optimization information. In fact, as is typical with genetic search, all that is required is that it be possible to obtain some overall evaluation of a sequence relative to other sequences. We refer to this operator as genetic edge recombination. To demonstrate the effectiveness of this operator on a widely known sequencing problem we have applied it to the Traveling Salesman Problem. This problem involves finding the shortest Hamiltonian path or cycle in a complete graph of n nodes. This problem is a classic example of an NP-hard problem; all known methods of finding an exact solution involve searching a solution space that grows exponentially with n, the number of nodes in the graph. In this chapter we report results for 30 and 105 city problems that are superior to our previously reported results on these problems(Whitley et al.,1989).

Although this operator produces excellent results on the Traveling Salesman Problem, the significance of the operator goes well beyond this particular problem. Because the operator only looks at the total evaluation of a sequence, it can be applied to any general sequencing problem. It therefore represents a general approach to sequencing scheduling which produces a global search and which, despite its general purpose nature, produces results that are competitive with other, more domain-specific methods as illustrated in the Traveling Salesman Problem.

We have built a prototype scheduler that is based on a production line scheduling problem encountered at a Hewlett-Packard manufacturing site in Fort Collins, Colorado. Our recent work with this scheduling system looks at various assumptions about the flow of jobs along the production line. A careful analysis of this problem has revealed both new strengths and weaknesses in the operator for specific types of problems. Specifically, we compare a strict First-In-First-Out(FIFO) ordering of jobs to another approach that allows reordering of jobs between machines using heuristic rules. Surprisingly, our data suggest that using the genetic algorithm alone on the more difficult, strict FIFO sequencing problem can produce competitive results with the hybrid algorithm that combines the genetic search with greedy scheduling heuristics and that allows reordering of jobs between machines. This result not only demonstrates the advantages of global search, but it also makes our results more useful for a wider variety of real-world scheduling applications.

We have also studied classes of problems that cause difficulty for genetic search. We show that for a problem with multiple solutions the genetic algorithm may converge to a competitive, yet suboptimal, solution. Specifically, this can occur when two competitive but dissimilar parents reproduce. By dissimilar, we mean that they represent different solutions, which when recombined produce disfunctional or uncompetitive offspring.

Finally, the results presented here are in part based on enhancements we have made to our genetic algorithm implementation. Our algorithm, which we refer to as GENITOR, differs from most standard genetic algorithm implementations in that it employs one-at-a-time recombination where the best strings always survive and it uses a rank-based method of allocating reproductive trials(Whitley,1989; Whitley and Kauth,1988). We have shown that a distributed version of this algorithm can actually produce improved results over the serial algorithm. This is important, not only because of the potential for improved optimization, but also because a distributed implementation can make it possible to obtain results an order of magnitude faster than would be possible with a serial search.

1.2 THE EDGE RECOMBINATION OPERATOR

The key to solving a problem such as the Traveling Salesman Problem using genetic algorithms is to develop an encoding that allows recombination to occur in a meaningful way.

Consider the following tours: [A B C D E F] and [B D C A E F]. If we extract the "link" information in the tour, we have "ab bc cd de ef af" and "bd dc ca ae ef bf" as parent tours. We assume that the tours are circular; the journey finishes again at the initial city, thus forming a Hamiltonian cycle. Also note that edge "ab" has exactly the same value as "ba." All that is important is the value of the link, not its direction. Thus, an ideal operator should construct an offspring tour by exclusively using links present in the two parent structures. On average, these edges should reflect the goodness of the parent structure. There is no random information that might drive the search toward arbitrary links in the search space. Thus, an operator that preserves edges will exploit a maximal amount of information from the parent structures. Operators that break links introduce unwanted mutation.

The edge recombination operator we have developed uses an "edge map" to construct an offspring that inherits as much information as possible from the parent structures. This edge map stores all the connections from the two parents that lead into and out of a city. Since the distance is the same between two cities whether one is coming or going, each city will have at least two and at most four edge associations (two from each parent). An example of an edge map, the operation of the recombination method, and an example of recombination are given in figure 1.1.

The problem that has normally occurred with edge recombination is that cities are often left without a continuing edge(Grefenstette,1987). Thus cities become isolated so that a new edge has to be introduced. Using an edge map we can choose "next cities" in such a way that those cities that currently have the fewest unused edges (and thus appear to have the highest probability of being isolated) have priority for being chosen next. Of course, there must be a connecting edge to a city before it is actually a candidate, but at any point in time there are up to three candidates (and up to four in the case of the initial city), and the candidate with the fewest edge connections is chosen. In the case of a tie the algorithm chooses randomly from among the candidate edges.

Note that the one edge that the edge recombination operator fails to enforce is the edge that goes from the final city to the initial city. On Hamiltonian paths, this is not an issue, but on Hamiltonian cycles it means that a limited amount of mutation is likely to occur (on at most one link in the offspring, i.e., at a rate of 1/n where n is the length of the tour). By allowing different cities to occupy the initial position in different genotypes, this mutation is distributed over the various links.

The edge map stores all the connections from the two parents that lead into and out of a city. Consider these tours: [A B C D E F] and [B D C A E F]. The edge map for these tours is as follows:

A has edges to : B F C E	D has edges to : C E B
B has edges to : A C D F	E has edges to : D F A
C has edges to : B D A	F has edges to : A E B

THE RECOMBINATION ALGORITHM

1. Choose the initial city from one of the two parent tours. (It can be chosen randomly or according to criteria outlined in step 4.) This is the "current city."

2. Remove all occurrences of the "current city" from the left-hand side of the edge map. (These can be found by referring to the edgelist for the current city.)

3. If the current city has entries in its edgelist go to step 4; otherwise, go to step 5.

4. Determine which of the cities in the edgelist of the current city has the fewest entries in its own edgelist. The city with the fewest entries becomes the "current city." Ties are broken randomly. Go to step 2.

5. If there are no remaining "unvisited" cities, then STOP. Otherwise, randomly choose an "unvisited" city and go to step 2.

AN EXAMPLE

1. The new "child" tour is initialized with one of the two initial cities from its parents. Initial cities A and B both have four edges; randomly choose B.

2. The edge list for B indicates the candidates for the next city are A C D and F. C, D and F all have two edges: the initial three minus B. A now has three edges and thus is not considered. Assume C is randomly chosen.

3. C now has edges to A and D. D is chosen next, since it has fewer edges.

4. D only has an edge to E, so E is chosen next.

5. E has edges to A and F, both of which have one edge left. Randomly choose A.

6. A must now go to F.

The resulting tour is [B C D E A F] and is composed entirely of edges taken from the two parents.

Figure 1.1: The edge mapped recombination operator

Our test over several random recombinations for problems of various sizes indicates that mutation rates are low, typically ranging from 1 to 5 percent.

1.2.1 Edge Recombination and Schema Theory

The theory behind most genetic algorithm implementations assumes that recombination and selection change the sampling rate of hyperplanes in an n-dimensional hypercube, where n refers to the length of some explicit or underlying binary representation of the problem. It is possible to show that our recombination operator in fact manipulates an underlying binary encoding of the Traveling Salesman Problem that largely conforms to the existing theory. Although the purpose of this chapter is to motivate applications largely by way of example, having a mathematical foundation that allows us to account for results not only makes it easier to explain and justify the approach, but it also provides a theoretical base from which to extend and improve the method. We will not review the fundamental theorem of genetic algorithms (rather see Goldberg,1989; Holland,1975; Schaffer,1987; Whitley and Starkweather,1990) but Figure 1.2 does demonstrate how the Traveling Salesman Problem can be represented by a binary encoding and how edge mapped recombination results in the recombination of the underlying binary encoding.

This analysis also reveals another source of mutation (one pointed out to us by David Schaffer). Notice that a mutation occurs in the very last bit position. This is due to "mutation by omission." The edge "ef" occurred in both parents but was not passed on to the offspring. If one parent visits cities [... H C D F ...] and the other parent visits cities [... J C D K ...] then there are two ways of recombining these edges: one that uses the common edge "cd" and one that does not. For example, an offspring containing the fragment [...J C D F...] recombines the two parents and uses the common edge "cd." However, an offspring can inherit the following sequence fragments [...H C J...K D F...] without introducing any edge other than those taken from the two parents. We describe this as mutation by omission. This kind of implicit mutation occurs only if a single common edge is present; if more than one contiguous common edge occurs between two parents, the mutation, if it occurs, will be an explicit one, since at least one common edge will be isolated if the recombination operator does not use all the common edges together.

Despite occasional explicit and implicit mutation, edge recombination is clearly manipulating schemata in this representation space. If the genetic algorithm actually used this binary representation, there would be no clue as to which recombinations result in legal tours. Nevertheless, this characterization of the problem allows us to relate the behavior of our operator to the existing theory.

1.2.2 Results for the Traveling Salesman Problem

Our experiments with the Traveling Salesman Problem involve the use of a serial implementation of GENITOR and a distributed version of GENITOR that allows swapping (or "migration") of individuals between subpopulations. The distributed algorithm uses 10 subpopulations, each of which is performing a typical genetic search except for the fact that copies of the best strings are passed to neighboring subpopulations at regular intervals. The motivation behind the distributed algorithm as well as a more thorough discussion of results for a variety of problems are presented elsewhere(see Whitley and Starkweather,1990).

On 30 city and 105 city Traveling Salesman Problems, the distributed genetic algorithm consistently finds good solutions. The best known solution for the 30 city problem is 420. (Actually, it is 420 if one rounds off the distance between edges before summing; it is 424 if we sum the real valued edges and then round. There has been some confusion about this in the literature. Oliver

The tour [A B C D E F] has the following edges: ab, bc, cd, de, and ef. All possible edges in this six city problem are:

`ab ac ad ae af bc bd be bf cd ce cf de df ef`

The Hamiltonian cycle [A B C D E F] can be represented as a binary mask that indicates which links are present. This is illustrated in the following table.

ab	ac	ad	ae	af	bc	bd	be	bf	cd	ce	cf	de	df	ef
1	0	0	0	1	1	0	0	0	1	0	0	1	0	1

Consider again the tours [A B C D E F] and [B D C A E F]. We now add this second tour as the second binary mask in the table.

ab	ac	ad	ae	af	bc	bd	be	bf	cd	ce	cf	de	df	ef
1	0	0	0	1	1	0	0	0	1	0	0	1	0	1
0	1	0	1	0	0	1	0	1	1	0	0	0	0	1

Now consider the following offspring: [B C D E A F]. We add the mask for this tour as the third binary string in our table.

ab	ac	ad	ae	af	bc	bd	be	bf	cd	ce	cf	de	df	ef
1	0	0	0	1	1	0	0	0	1	0	0	1	0	1
0	1	0	1	0	0	1	0	1	1	0	0	0	0	1
0	0	0	1	1	1	0	0	1	1	0	0	1	0	0

To simplify the binary mask, we remove all those bit positions where neither parent has an edge, and indicate how recombination occurred.

ab	ac	ad	ae	af	bc	bd	be	bf	cd	ce	cf	de	df	ef
1	$\boxed{0}$	-	0	$\boxed{1}$	$\boxed{1}$	$\boxed{0}$	-	0	1	-	-	$\boxed{1}$	-	1
$\boxed{0}$	1	-	$\boxed{1}$	0	0	1	-	$\boxed{1}$	$\boxed{1}$	-	-	0	-	1
0	0	-	1	1	1	0	-	1	1	-	-	1	-	0

Notice that a mutation occurs in the very last bit position. This is due to "mutation by omission." The edge "ef" occurred in both parents but was not passed on to the offspring.

Figure 1.2: The following illustrates how a Traveling Salesman Problem can be mapped to an underlying binary encoding and also indicates how edge mapped recombination acts on the underlying binary encoding.

et al.(1987) report an optimal solution of 424, but the actual tour they give is not optimal: it has a length of 424 if we first round off the distance associated with the edges and then sum. This explains discrepancies we noted in a previous publication[Whitley et al.,1987].) Using the serial version of GENITOR, a population of 2,000, and allowing up to 70,000 recombinations, we found the optimal solution of 420 on 28 out of 30 runs. On the other two runs we found the near-optimal solution of 421. We were somewhat surprised by these results. In our previous experiments we did not find solutions this consistently or this good. The only difference between these results and our previous results is a difference in population size. We had limited our experiments to populations of 100 to 500. Within this range, we found that populations of approximately 200 to 300 seem to work best. We went back and tested the larger population sizes only after obtaining the results from the distributed genetic algorithm. We ran the larger population sizes for comparative purposes, and thus were surprised by the results. From an applications point of view, this represents one of the traps that we must be careful to avoid. We originally did not push the genetic algorithm harder because we were already impressed by the results we had obtained.

For the distributed version of GENITOR we used 10 subpopulations of size 200 each, allowed up to 7,000 recombinations in each subpopulation, and swapped individuals between the subpopulations every 1,000 recombinations. The overall amount of work between the two algorithms is very comparable. The overhead of the distributed algorithm is roughly offset by the additional cost associated with sorting the larger population in the serial algorithm. Using the distributed algorithm, we found the optimal solution on 30 out of 30 runs. Thus, we were able to find the best known solution on every run.

On the 105 city problem we only present results for the distributed algorithm with migration. We used 10 subpopulations of size 1,000 each and allowed up to 200,000 recombinations in each subpopulation. Swapping occurred after every 10,000 recombinations. The best known solution (using the sum of the real-valued edges, and then rounding) is 14,383. We matched the best known solution on 15 out of 30 runs. This is 50 percent of the runs; on the remaining 50 percent of the runs, the solution found was always within 1 percent of the best known.

1.3 SEQUENCING PROBLEMS

Our development of a theoretically well-founded genetic operator for the Traveling Salesman Problem is a major innovative step forward in the application of genetic algorithms to sequencing problems. The only information used was feedback about the total length of the tours that were evaluated. Although these results are impressive, the implications are far more dramatic for other kinds of sequencing problems where the problems are too ill structured to apply the kinds of heuristics traditionally used to solve Traveling Salesman Problems. Since the genetic recombination operator requires so little information, it is ideally suited to the many and varied kinds of sequencing problems that occur in scheduling.

The use of genetic algorithms lends itself to a general purpose system in the sense that the core of the genetic simulator remains the same from one application to the next. What varies from one application to the next is (1) the representation scheme and (2) the evaluation function. However, a broad class of scheduling problems can be viewed as sequencing problems. By optimizing the sequence of processes or events that are fed into a simple schedule builder, optimization across the entire problem domain can be achieved. Schedules have not always been viewed this way because there has not existed a general purpose mechanism for optimizing sequences that only requires feedback about the performance of a sample sequence. However, as our results on the Traveling Salesman Problem indicate, we have this capability. Thus, a “genetic” approach to scheduling has

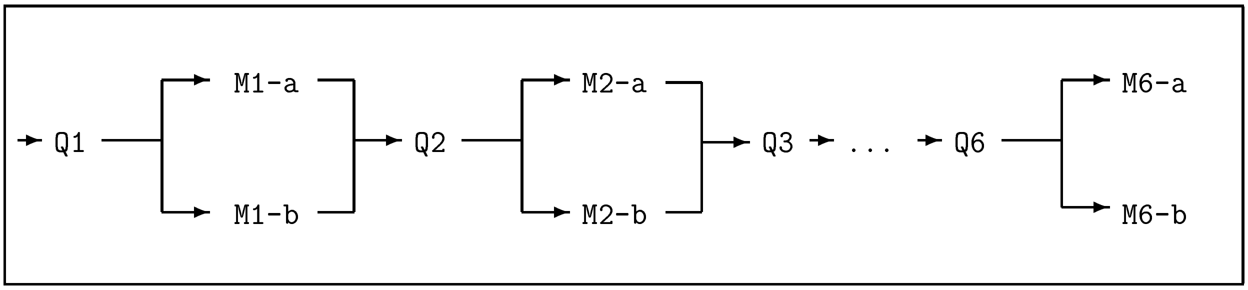

Figure 1.3: Layout of the production line.

the potential to produce some very general scheduling techniques, and could be the foundation of a general purpose approach to sequence scheduling.

1.3.1 A Production Line Scheduler

We have developed a prototype system for optimizing scheduling problems. We have previously discussed a preliminary version of this system that used genetic recombination in conjunction with scheduling heuristics(Whitley at al.,1989). In this chapter, we show improved results on this type of problem as well as new results for a different version of the scheduling problem that uses no heuristics and enforces a strict FIFO ordering of jobs. We also look at variations on this same problem and clarify some of the assumptions of our model.

We have used the edge recombination operator to drive a prototype system designed to solve a production line scheduling problem modeled after a board assembly line at Hewlett-Packard in Fort Collins, Colorado. The problem is characterized by the following attributes:

- There are six workcells (i.e., machine groups) in sequence, and each completes a specific operation. Each workcell receives work from a single input queue and sends completed work to a single output queue. The scheduling dependencies are therefore six workcells deep. In order words, if we let A, B, C, D, E, and F represent the six workcells in sequence, then clearly the scheduling of workcell F is dependent on the scheduling of workcell E, which is dependent on D, which is dependent on C, which is dependent on B, which is dependent on the initial scheduling of workcell A. We stress the depth of these dependencies because the genetic algorithm must generate schedules that are sensitive to distant and indirect dependencies.

- Each workcell contains two identical machines operating independently and in parallel.

- Each machine has a "cost" associated with the type of work being done. In general, these are assumed to be the following: $1 per time unit for idling, $5 per time unit for processing, and $10 per time unit for setup.

- Twenty different types of products are produced in a single schedule. Each product has a fixed setup time and processing time at each operation.

Ten units of each type of product are to be assembled. Figure 1.3 illustrates the layout of the production line. M refers to Machine, and Q stands for Queue. Q1 is the initial job sequence optimized by the genetic algorithm. M1-a and M1-b are two identical machines that together make up the first workcell.

The operation of this hypothetical line has been modeled in two ways. In one case, a FIFO model is used such that a workcell simply dequeues the next item from its input queue and sets up for the job as required. In the FIFO model any two adjacent workcells in the production sequence behave as two machines coupled by a conveyor belt. When a time unit is simulated, the next available job in the preceding queue is assigned to the next available machine. No attempt is made to give the next available job to a machine already set up for that machine. In the FIFO model, the only optimization is the optimization of the sequence in which jobs are supplied to the first workcell. In other words, only Q1 is optimized.

The HYBRID model involves optimizing the initial sequence of jobs (i.e., Q1) in conjunction with greedy heuristics for reordering jobs between workcells. Each workcell has random access to the jobs in the preceding queue as well as knowledge about the parallel machines and the "supplying" workcell. Whenever possible, the HYBRID model attempts to select a unit identical to the last and will wait idle if a "like unit" will become available before a different setup can be accomplished. This implies, for example, that machines M2-a and M2-b use locally available information to select jobs from Q2 in a different order from that in which they arrived in Q2.

The strategy behind the HYBRID/greedy genetic scheduler implementation is as follows. (1) Using the edge recombination operator, the genetic algorithm produces a "primary schedule" indicating the sequence in which jobs are loaded onto the first workcell. (2) The "secondary" rule-based scheduler then generates a full schedule. The rules for the secondary scheduler are relatively simple since the problem is highly constrained by the availability of jobs as they come off the first machines.

In both the FIFO and the HYBRID model, the evaluation of the resulting schedule is used to assign a fitness value to the initial sequence (i.e., Q1) which was generated by the genetic algorithm. This "fitness" value is used to rank the sequences and thus to allocate reproductive trials in a population of initial "job sequence" genotypes. Since the FIFO scheduler is not allowed to reorder jobs between machines, the FIFO scheduling task is more difficult.

For the HYBRID model, the solution space is highly constrained once the initial machine is scheduled. If a good sequence of jobs can be found, then the remaining machines can be scheduled using a few simple rules that act as a greedy scheduler with some lookahead capabilities. The greedy scheduler uses the initial job sequence to produce a complete schedule that is then evaluated. This evaluation is then assigned as the "performance" value associated with the initial sequence. Thus, the goal of recombination was to find a sequence that optimized the performance of the final schedule in the environment of the secondary greedy scheduler.

The rules used to implement the secondary scheduler of the HYBRID model do not actually look ahead but rather use a "wait and see" approach. If a machine is already set up to run "Job X" and a job X is available, then it is scheduled. If a machine is set up to run "Job X" and X is unavailable, then an "idle time" is introduced. The scheduler will wait and see if this idle time can be filled later by another job. If X becomes available before another job can be set up, then the scheduler waits for X. However, if at some later point in time X is still not available and another Job Y could have been set up and started, then the idle time is "back-filled" by scheduling a setup for the new Job Y. If multiple jobs could be set up in place of X (the unavailable job the machine is currently set up to run), then further comparisons are made. First, the machine decides which jobs have the shortest setup time. If two or more jobs have identical setup times, the one that has the shorter execution time is chosen. The only restriction on the selection of the new job type is that if there is more than one type of job in the input queue, a machine will not set up to run the same job type being run by the corresponding parallel "like-machine." For example, the only way that M1-b will set up to run a job type that M2-a is already running is if that is the only job type in the input queue. This restriction overrides the "short-setup-time" criteria for selecting a new

WORKCELL	AVERAGE PROCESSING TIME	AVERAGE SETUP
1	5.20	3.40
2	6.20	5.55
3	4.70	3.90
4	4.70	3.90
5	4.70	3.90
6	4.70	3.90

Table 1.1: Scheduling Problem-1

job type.

Both the FIFO and HYBRID models attempt to keep all machines busy at all times. In both models, the machines in the workcell will set up to do whatever work is available at almost every opportunity (subject to the restrictions just outlined). There is no attempt to determine if a machine should remain idle rather than set up or do work; machines are idle only if there is no work to do.

Another key part of adapting our genetic recombination operator for scheduling applications was to encode the problem as a sequencing problem. Using A, B, C and so on to represent jobs, we might have an actual sequence something like the following: [A A A B B A A C C]. Note that job types reoccur. Our strategy for encoding the problem was simply to ignore the fact that some of the jobs are the same and thus can change positions without altering the schedule. Thus, [A A A B B A A C C] is encoded as [A1 A2 A3 B1 B2 A4 A5 C1 C2]. This means we are not using all the information available to us. The results indicate that the operator works well even without this additional information. Although the results are positive, it is most likely possible to devise a version of the edge recombination operator that exploits this additional information. We have not vigorously explored this possibility, although this could perhaps solve some of the difficulties we discuss later in this chapter.

1.3.2 The Scheduling Results

We evaluated our scheduler on numerous problems. We found that the most difficult of these problems involved schedules with a uniform number of jobs. While a uniform number of jobs is not typical of scheduling problems, we have found that this makes the problem more difficult. If there is a bias such that there are significantly more of one type of job than others, then the scheduling task is usually easier. This appears to make intuitive sense. If there are one or two jobs that occupy large blocks of time, then these jobs very much constrain the solution space. Large blocks of time can be allocated to the most frequently occurring job types, while any resulting "holes" are filled with small jobs.

The following discussion pertains to a problem we will refer to as problem-1. This problem has 200 job batches, where there are 10 different job types and 20 of each type. Setup times were varied from 1 to 10 time units, and processing times from 1 to 20 time units, as shown in Table 1.1. Workcells 3 through 6 have identical characteristics, while 1 and 2 vary significantly.

A theoretical limit can be calculated for problem-1 which establishes a lower bound on the cost function. The theoretical limit assumes that there is only one setup per job type per workcell, and no idle time. The theoretical limit for problem-1 has an associated cost of 35,110. If we assume a single line of machines is used rather than paired machines, the theoretical limit based on the assumption of one setup per job and no idle time does not change. Nevertheless, we can prove that in fact it is impossible to produce a schedule with a cost of less than 36,442 because no machine later in the sequence can finish before any prior machine in the sequence. The situation is more complex when paired machines are involved because we can now only make the following, less restrictive observation: at each workcell, at least one of the paired machines must finish after all preceding machines in the sequence are done. More than likely, the “real limit” has an associated cost of at least 36,400, and it may actually be somewhat above this amount.

Because this problem is fairly well structured, we were able to generate competitive schedules by hand. We did this as an iterative process, generating what appeared to be good schedules by hand and then evaluating them using the same schedule evaluation that the genetic algorithm uses. In this way, we were able to construct fairly tight schedules. For the HYBRID model we were able to develop a schedule with an associated cost of 40,363, and for the more difficult FIFO model we were able to construct a solution with an associated cost of 41,694. In all likelihood, the optimal solution for the HYBRID model has a lower cost than the optimal solution for the FIFO model, simply because job reordering between machines results in greater flexibility in the schedules we can generate. Moreover, a good HYBRID model schedule is not necessarily a good FIFO model schedule. The 40,363 HYBRID model schedule, when run through the FIFO scheduler, produces a schedule with an associated cost of 45,800.

Figures 1.4 and 1.5 show several different kinds of experiments. Two versions of the FIFO and two versions of the HYBRID algorithm were run. For each model, a set of experiments was conducted using a “single-population” genetic algorithm with a population of 50 and one using a distributed genetic algorithm with 7 subpopulations of 50. (We also ran serial populations of 350 for comparison; the end results were almost identical with the distributed genetic algorithm, although the distributed algorithm produced faster initial reductions in error, even after the results were rescaled for comparative purposes.) Figure 1.6 gives an overlay of the four sets of experiments with respect to the theoretical limit. Actually, in one sense the FIFO and HYBRID schedulers are solving two distinct versions of the scheduling problem, and so, in terms of the optimization results, we are comparing apples and oranges. On the other hand, from an applications perspective of the end result, we are ultimately solving the same scheduling task. Thus, from this perspective the comparison is an interesting one.

Figure 1.6 shows that both of the FIFO runs overtake the HYBRID algorithm after approximately 10,000 recombinations. In the serial runs with small population sizes, the HYBRID algorithm produces slightly better results than the FIFO in the end. However, when the more powerful distributed algorithm is used, the FIFO actually produces the best results. Furthermore, the comparisons to the “iterative/hand-generated” results are very interesting with respect to the two types of models. By using the HYBRID model, the distributed genetic algorithm produced an average result of 40,952 over 30 runs. Thus, its average behavior is just above the hand-generated solution of 40,363. However, using the distributed genetic algorithm to optimize the FIFO model we obtained an average result of 40,804, whereas our best hand-generated solution was 41,694.

1.3.3 Significance of the Scheduling Results

From a scheduling/application point of view, the results suggest that there is no significant difference between the FIFO and HYBRID scheduling systems. From an implementation and computational point of view, however, there are real differences. First, the FIFO model is computationally more efficient than the HYBRID model, since we no longer have the overhead of evaluating the small rule-base used to implement the greedy reordering heuristics. Second, the FIFO converges to competitive solutions in fewer recombinations. The FIFO runs are close to the final results by 15,000 recombinations, whereas the HYBRID runs do not catch up until after roughly 25,000 recombinations. The faster FIFO evaluation and convergence means that approximate results can

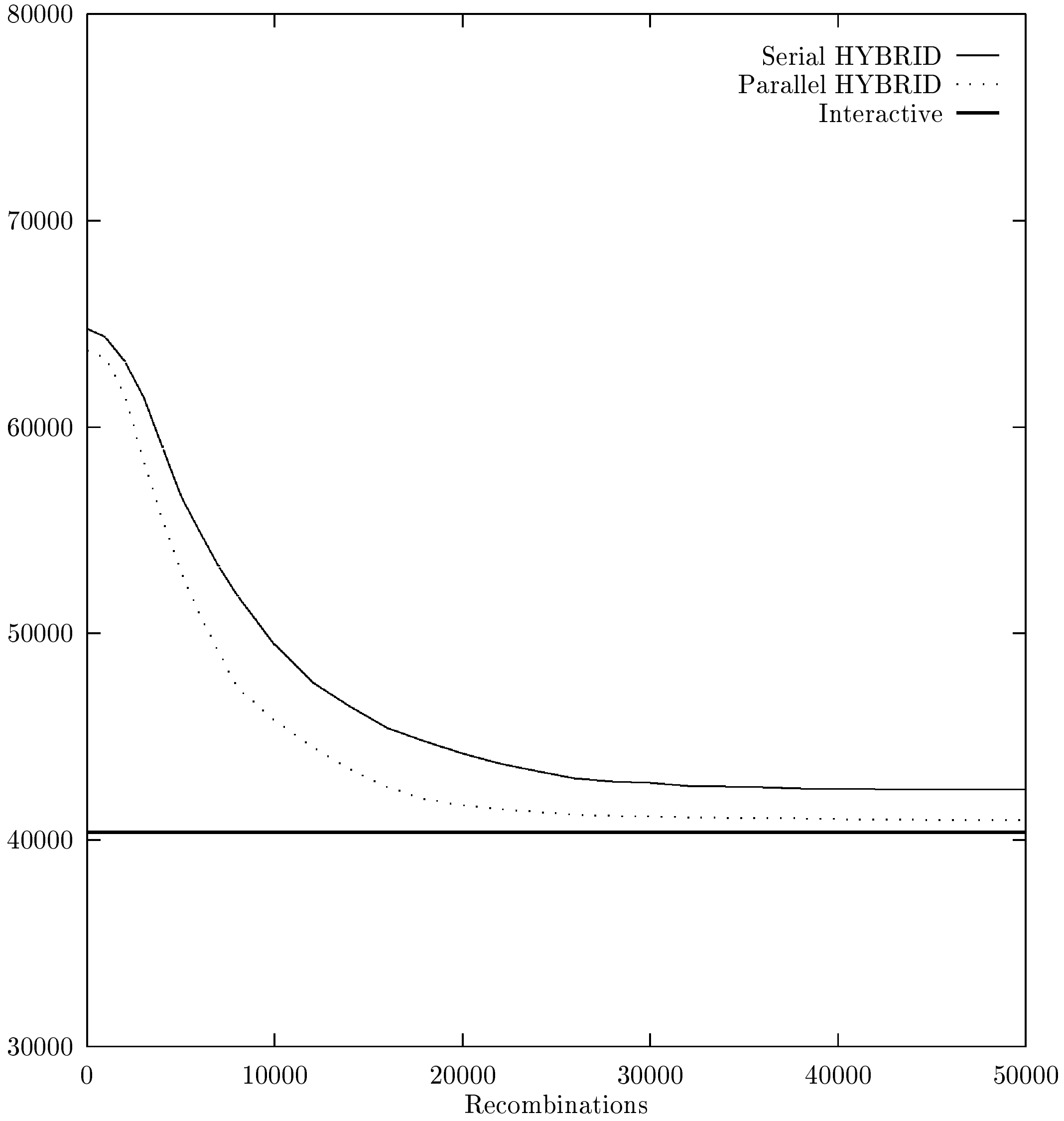

Figure 1.4: The results of using the HYBRID model in conjunction with a serial genetic algorithm (dashed line) and the distributed genetic algorithm (solid line). The interactively hand-generated solution (40,363) in this case is slightly better than the average solution found by the genetic algorithm. The average value of the solutions produced by the serial algorithm is 42,455, whereas the average value of the solutions produced by the distributed algorithm is 40,952. These results are averaged over 30 runs.

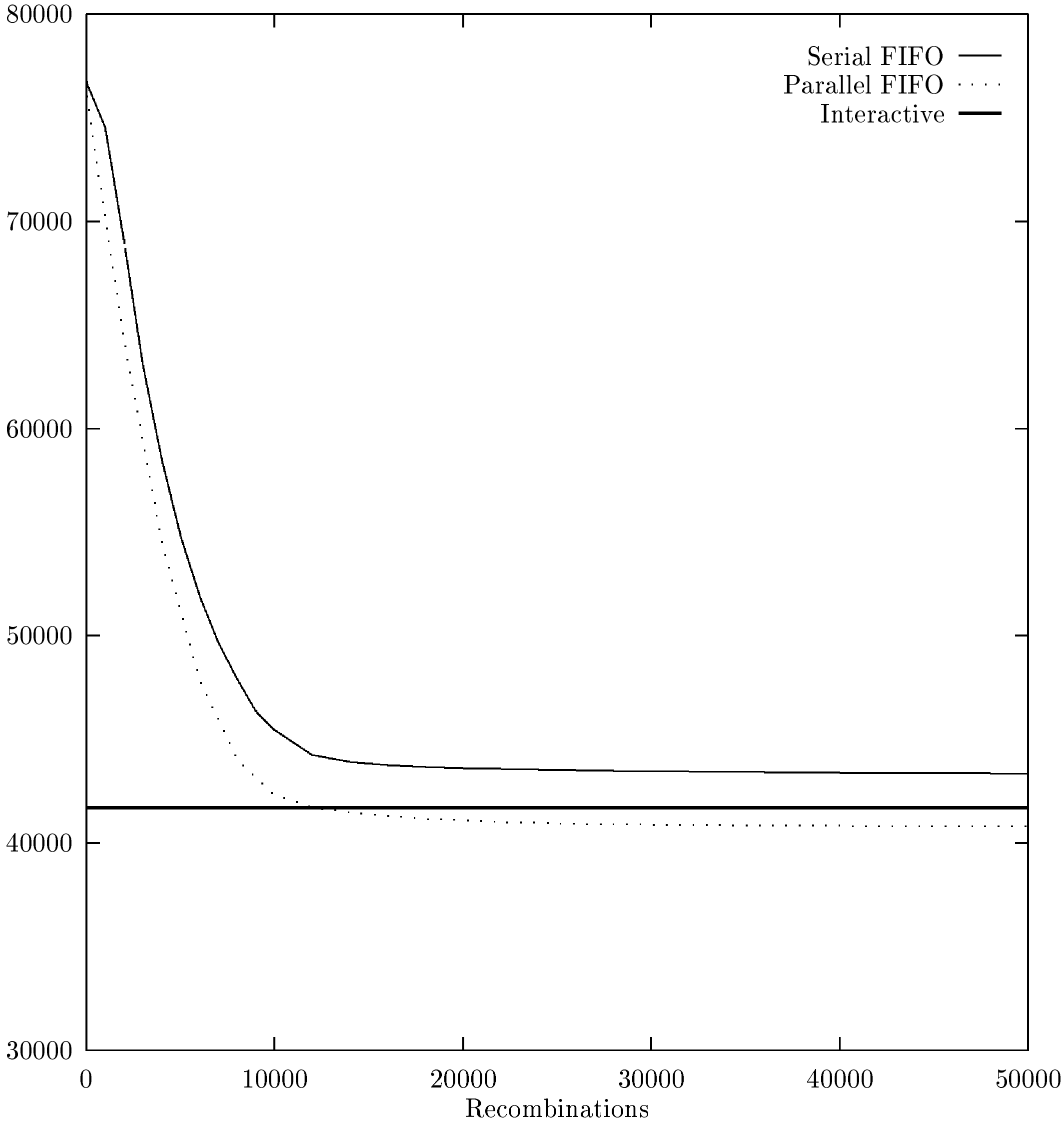

Figure 1.5: The results of using the FIFO model in conjunction with a serial genetic algorithm (dashed line) and the distributed genetic algorithm (solid line). In this case, the distributed FIFO genetic scheduler on average found solutions better than the interactively hand-generated solution (41,694) for this problem. The average value of the solutions produced by the serial algorithm is 43,330, whereas the average value of the solutions produced by the distributed algorithm is 40,804. These results are averaged over 30 runs.

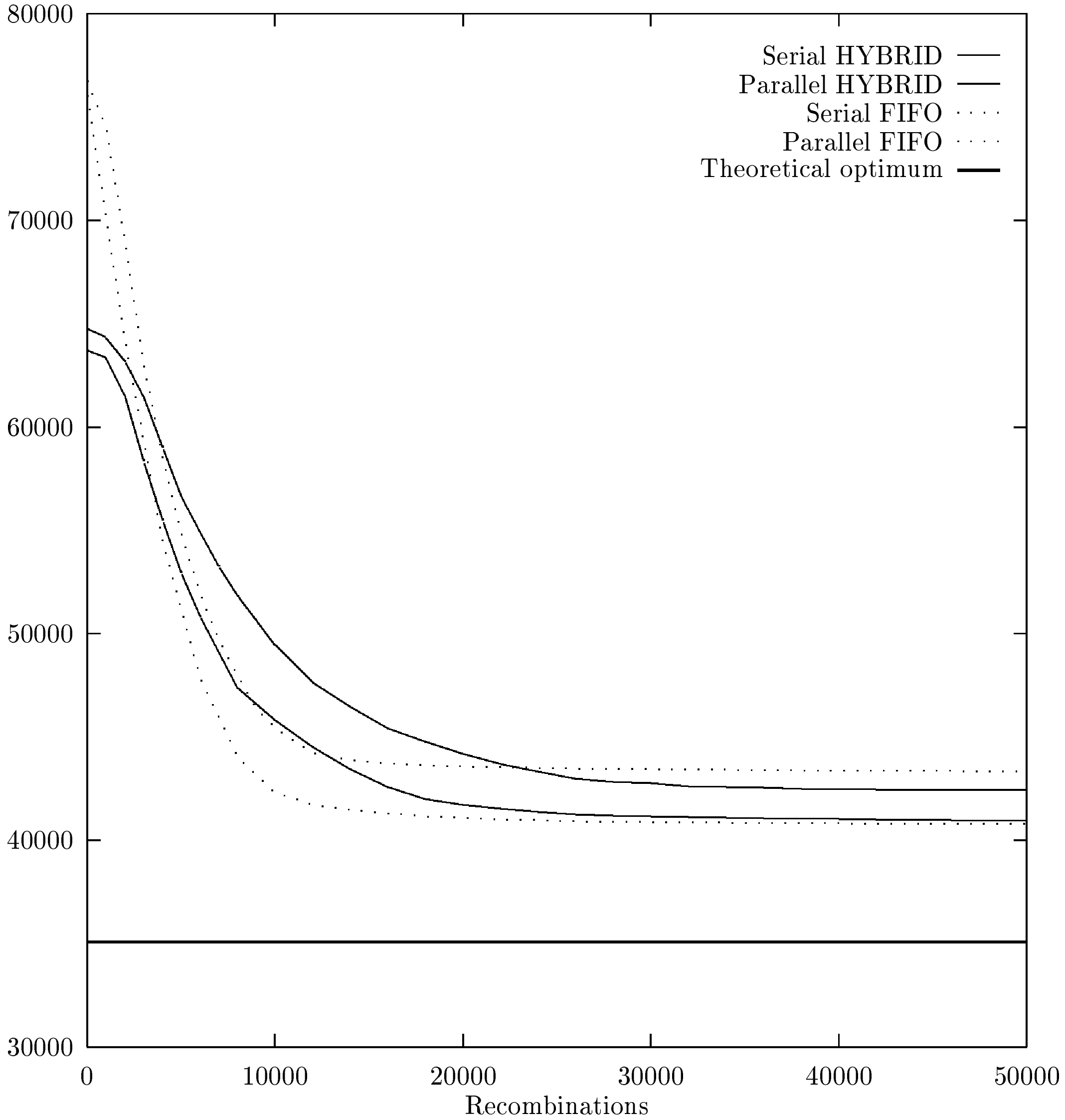

Figure 1.6: Overlay of the results of the FIFO model and the HYBRID model from the previous two figures. The two dotted lines represent the two FIFO results, whereas the two solid lines represent the HYBRID results. The theoretical optimum is also shown. In both cases the distributed solution is the result that is closer to the theoretical optimum.

be obtained two to three times faster than is possible with the HYBRID method.

The third difference, and certainly the most important one, is that FIFO is a more realistic model of many scheduling tasks. To implement the HYBRID algorithm would require human intervention or a sophisticated automated reordering system. The FIFO simply conveys work from one workcell to the next.

At first it may seem unreasonable that the FIFO scheduler should produce results competitive with the HYBRID system. However, we believe that the FIFO scheduler is competitive because it relies on global search. That is, what's good from a local, greedy point of view may not be good from the point of view of developing a good overall schedule. When tracing the FIFO schedules by hand we noted that the genetic algorithm appeared to miss optimization opportunities that the greedy heuristics would have certainly caught. However, when we traced the consequences of these "missed opportunities," we found that if the local improvement had been made it would have degraded the overall schedule because of dependencies that occurred later in the schedule.

The somewhat surprising results of the FIFO implementation is good news from our perspective because it suggests that we should not attempt to build heuristics into the scheduling system, but rather should let the genetic algorithm do the work. This keeps the implementation and the system simple and reduces computation costs.

1.4 ANALYZING THE SCHEDULER USING WORST CASE PROBLEMS

We have spent a good deal of time trying to decide how our evaluation function affects the schedules generated. Our analyses suggest that this problem is actually one that is relatively difficult to solve by genetic recombination. We have also conducted experiments studying the effect of using two machines in parallel. Our tests, though not exhaustive, suggest that running two machines in parallel while at the same time assuming that paired machines form a single workcell with a single queue between workcells makes the scheduling problem harder. In general, to obtain good schedules, the two parallel lines should specialize. That is, all "right-hand side" machines should do job A, while the "left-hand side" machines should do job B. However, to do this, the initial sequence of jobs must be perfectly interleaved to make the next available job arrive in the proper sequence to be consumed by the proper machine.

To use a simple case as an example, if we assume that every machine processes every job at the same rate and that all jobs have identical setup costs, then to achieve a tight schedule with minimal setups requires a sequence such as the following [A B A B A B .... C D C D C D]. In other words, this will cause the "right hand machine" to consume a job of type A, the "left-hand machine" to consume a job of type B, the "right-hand machine" next to consume a job of type A, and so on. Thus, a very simple scheduling task has a relatively complex (though highly regular) sequence of jobs. If we construct a problem of 200 jobs, with 20 different types of jobs and 10 jobs of each type, *and we assume that all processing times and all setup times for all jobs are the same,* then we create a situation in which there are numerous solutions. Furthermore, the multiple solutions for this schedule do not recombine well. Since we assume that all processing times and all setup times require five time steps, we will refer to this as the "All-5s" problem. A perfectly packed solution for the All-5s problem can be achieved which has a cost of 36,075. We stress that this is an artificial problem designed to create difficulties for the genetic algorithm. Nevertheless, it also provides useful insight into the nature of genetic search which impacts potential applications.

To illustrate why such a problem causes difficulties, consider a scaled-down version of the All-5s problem where we have four job types (A, B, C, and D) and we have two jobs of each type.

All of the following sequences are perfect solutions to this problem: [A B A B C D C D], [A C A C B D B D], [C B C B A D A D], and so on. However, if we try to recombine some of these different potential solutions, we typically obtain uncompetitive offspring. For example, we might recombine [A B A B C D C D] and [A C A C B D B D], both of which are optimal solutions. When recombining these "parents" it is possible to produce the offspring [A B D C A B C D], which is in fact the worst possible schedule. When dissimilar solutions are recombined and uncompetitive offspring are produced, the result from a theoretical point of view is high variance in the fitness values associated with hyperplane samples in the search space. This in turn means that the genetic algorithm is receiving inconsistent feedback about what are good sequence combinations (i.e., good hyperplanes) in the problem space. We have observed the same kind of difficulties in other problems(Whitley et al.,1990) while Baker and Grefenstette(1989) discuss from a theoretical perspective the problem of hyperplane samples that are characterized by a high variance in the associated set of fitness values for that hyperplane. Interestingly, the compromise solution to this problem is simply to block all jobs of the same type together; this compromise solution has an associated cost of 42,075. While there are multiple ways in which the jobs may be blocked, the "order" in which the blocked jobs occur is not important. Therefore, reordering the blocked jobs during recombination does not degrade performance in the offspring. The result of this strategy is that each machine in each workcell runs half of each job type. The schedule is perfectly packed, but two setups per job type are required.

Our past experience with other problems where it is possible to recombine dissimilar parents suggests that when this occurs we can obtain better results running several small searches instead of one larger search. This is because as larger populations are used, a greater number of dissimilar solutions are likely to be contained in the population.

We ran the All-5s problem using populations of 50 and a population of 500 using the paired machines configuration. On 30 runs the FIFO model using a population of 50 produced an average schedule of 39,923. Although we did not run a large number of experiments to confirm the following observation, based on runs using a population of 500 it appears that increasing the population size actually degrades performance toward the compromise solution. Again, this in not unexpected given that larger populations will contain more dissimilar solutions. The computational cost of running a population of 500 is actually comparable to running 10 populations of 50 for one-tenth the number of recombinations. Therefore, rather than increasing population size, if we take the best of 10 runs using a population of 50, we are much better off: an average of 37,628 is obtained in our experiments. Despite doing our best to torture the genetic algorithm with an extremely unfavorable problem, the results are respectable.

We next ran a set of experiments aimed at removing the obstacle that causes the difficulty for the genetic algorithm. We assume exactly the same All-5s problem, but we now assume that only a single line of machines is used. For the genetic algorithm the problem is much easier.

For a single line of machines (again assuming all jobs have the same processing time and setup costs), the sequence would be merely [A ... A, B ... B, etc.]. Although this is an easy problem (for people who happen to be good at finding order in complex patterns), the genetic algorithm must "discover" that this is an easy problem by finding that it should order the jobs in batches. GENITOR finds the the optimal solution within 10,000 recombinations on every run (30 runs each, FIFO and HYBRID) using populations of 50. This is not an amazing accomplishment, but it is reassuring that the genetic algorithm also finds this to be a trivial problem to solve.

1.5 DISCUSSION AND CONCLUSIONS

Our results on Traveling Salesman Problems are excellent, and, in general, we would not expect difficulties on problems sequencing unique items or events. For example, we are currently working on a real-world scheduling application where we must determine the sequence in which customer orders are to be filled. These orders involve different degrees of product mix (e.g., 50 units of product A, 20 units of product C, etc.) where there are approximately 500 products. The ability to fill an order is constrained by current production (not all products are being produced at any point in time) and by the products in storage (which we wish to minimize). Furthermore, in general we want to move the product from the production line into a truck for shipping. Therefore, once we start to fill an order, we wish to fill it as quickly as possible. Assuming we fill 10 orders per hour (which is realistic given the problem we are working on), then on average, we must be fill and load each order for shipping in under an hour.

Although there are some subtle difficulties in solving such a problem, we view the underlying scheduling problem in much the same way that we view the Traveling Salesman Problem. Each order is unique, and we wish to find the sequence in which the orders should be filled. Our objective is to minimize the amount of time it takes to fill an order while at the same time minimizing the amount of product that must go in and out of storage. The exact balance to strike between these two goals will depend on the evaluation function.

In general, we feel that our scheduler will generate high-quality solutions to problems in which the objects or events to be sequenced are unique. What about scheduling problems in which the objects or events to be scheduled are not unique? Our experiments show that we can indeed create scheduling problems that are unfavorable for the genetic algorithm. We can even explain these problems in terms of the underlying theory of genetic algorithms. Nevertheless, unfavorable conditions do not prevent the genetic algorithm from finding competitive solutions, although they may prevent it from finding an optimal solution. Problems most likely to cause difficulty are those that have multiple dissimilar solutions that do not "recombine well." We suspect that the problems most likely to be difficult for the genetic algorithm may be those that are not likely to be hard problems in the first place. If a problem is sufficiently well structured to have multiple solutions that are essentially variations on the same scheduling strategy, then these solutions are likely to be obvious. Thus, in general, we do not believe that scheduling problems in which the objects or events to be scheduled are not unique pose a real difficulty.

Certainly, more work needs to be done. There is a critical need for comparative studies on larger problems that are known to be difficult and for which other scheduling strategies have been developed. However, the work conducted so far has produced positive results that suggest a very exciting potential.

ACKNOWLEDGMENTS

This research was supported in part by a grant from the Colorado Institute of Artificial Intelligence (CIAI). CIAI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the state of Colorado. CATI promotes advanced technology education and research at universities in Colorado for the purpose of economic development.

1.6 References

Baker, J. and J. Grefenstette. (1989). How genetic algorithms work: a critical look at implicit parallelism. *Proceedings of the Third International Conference on Genetic Algorithms*, Palo Alto, Ca.: Morgan Kaufmann. pp. 20-27.

Goldberg, David. (1989). *Genetic algorithms in search, optimization, and machine learning.* Reading, Mass.: Addison-Wesley.

Grefenstette, John. (1987). Incorporating problem specific knowledge in genetic algorithms. In Lawrence Davis (ed.), *Genetic algorithms and simulated annealing.* Palo Alto, Ca.: Morgan Kaufmann. pp. 42-60.

Holland, John. (1975). *Adaptation in natural and artificial systems.* Ann Arbor: University of Michigan Press

Oliver, I.M. and D.J. Smith and J.R.C. Holland. (1987). A study of permutation crossover operators on the traveling salesman problem. In John Grefenstette (ed.), *Genetic algorithms and their applications, Proceedings of the Second International Conference.* Hillsdale, N.J.: Erlbaum Associates. pp. 224-230

Schaffer, David J. (1987). Some effects of selection procedures on hyperplane sampling by genetic algorithms. In Lawrence Davis (ed.), *Genetic algorithms and simulated annealing.* Palo Alto, Ca.: Morgan Kaufmann. pp. 89-103.

Whitley, Darrell. (1989). The GENITOR algorithm and selective pressure: why rank-based allocation of reproductive trials is best. *Proceedings of the Third International Conference on Genetic Algorithms*, Palo Alto, Ca.: Morgan Kaufmann. pp. 116-121.

Whitley, D. and J. Kauth. (1988). Genitor: a different genetic algorithm. *Proceedings of the Rocky Mountain Conference on Artificial Intelligence.* Denver, Colo. pp. 118-130.

Whitley, Darrell and Tim Starkweather. (1990). GENITOR II: A distributed genetic algorithm. *Journal of Theoretical and Experimental Artificial Intelligence*, (in press).

Whitley, Darrell and Tim Starkweather and Chris Bogart. (1990). Genetic algorithms and neural networks: optimizing connections and connectivity. *Parallel Computing*, (in press).

Whitley, Darrell, Tim Starkweather and D'Ann Fuquay. (1989). Scheduling problems and traveling salesman: the genetic edge recombination operator. *Proceedings of the Third International Conference on Genetic Algorithms*, Palo Alto, Ca.: Morgan Kaufmann. pp. 133-140.