GENETICS AND RANDOM KEYS FOR SEQUENCING AND OPTIMIZATION

James C. Bean
Department of Industrial & Operations Engineering
University of Michigan
Ann Arbor, MI 48109-2117

Technical Report 92-43

June 1992
Revised February 1993
Revised July 1993
Revised October 1993
Revised December 1993

Genetic Algorithms and Random Keys for Sequencing and Optimization

James C. Bean
Department of Industrial and Operations Engineering
University of Michigan
Ann Arbor, Michigan 48109-2117
(313) 763-1454
James_Bean@umich.edu

December 17, 1993

ABSTRACT

In this paper we present a general genetic algorithm to address a wide variety of sequencing and optimization problems including multiple machine scheduling, resource allocation and the quadratic assignment problem. When addressing such problems, genetic algorithms typically have difficulty maintaining feasibility from parent to offspring. This is overcome with a robust representation technique called *random keys*. Computational results are shown for multiple machine scheduling, resource allocation and quadratic assignment problems.

KEYWORDS: Computers-computer science : Artificial intelligence-genetic algorithms; Production-scheduling : Approximations-heuristic; Programming : Integer : Heuristic

Introduction

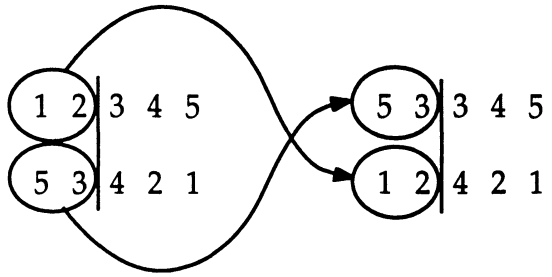
Genetic algorithms were developed in computer science in the mid 60's ([18]). They seek to breed good solutions to complex problems by a paradigm that mimics evolution. A population of solutions is constructed. Solutions in the population mate and bear offspring solutions in the next generation. These *reproduction* and *crossover* operations are programmed to replicate the paradigm of *survival-of-the-fittest.* Over many generations the solutions in the population improve until the best of the population is (hopefully) near optimal.

Adopting the basic terminology of genetics: a *chromosome* is an encoding of a solution and is a vector in $\Re^n$; a *gene* is an element of the chromosome (vector); an *allele* is a value taken by that element. For example, $x \in \Re^9$ might be a chromosome, x_4 one of its genes, and if $x_4 = 3.5$ then the fourth gene has allele 3.5.

Crossover is the process by which two parent chromosomes recombine to create a new offspring chromosome. The traditional operator is the one-point crossover. To illustrate, consider a simple genetic algorithm approach to the single machine sequencing problem. A candidate solution to a single machine sequencing problem is a permutation of the n jobs. Two such permutations for five jobs are $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ and $5 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 1$. The most direct chromosomal representation of such sequences are the permutations $x = (1,2,3,4,5)$ and $x' = (5,3,4,2,1)$. A one-point crossover operation would cleave each permutation at some point, say after the second job scheduled, and exchange leading segments. Executing that process on the example sequences gives $5 \rightarrow 3 \rightarrow 3 \rightarrow 4 \rightarrow 5$ and $1 \rightarrow 2 \rightarrow 4 \rightarrow 2 \rightarrow 1$ (see Figure 1). Neither is a valid tour. Genetic algorithms have been slow to gain acceptance for operations research problems since crossing over two feasible solutions does not, in many cases, result in a feasible solution as an offspring. See [12] for details on genetic algorithms.

Many authors have developed problem specific representations of solutions that overcome the offspring feasibility difficulty. Some of these include PMX crossover ([13]), the subsequence-swap operator ([15]), the subsequence-chunk operator ([16]), other subsequence operators ([7]), edge recombination ([33]), the ARGOT strategy ([28]) and forcing

FIGURE 1: One-point Crossover



([25]). Most of these applications use literal permutation encoding strategies, the notable exception being [28].

A continuing drawback has been the need for specialized representations for each problem variation. The major contribution of this paper is the concept of *random keys*, a method for representing solutions (chromosomal encoding) that produces feasible offspring for many sequencing and optimization problems. It guarantees feasibility of all offspring without creating additional overhead.

Section 1 presents this robust representation approach. Section 2 gives a detailed genetic algorithm that implements random keys. Section 3 presents computational results on three classes of problems. Summary and conclusions are in Section 4.

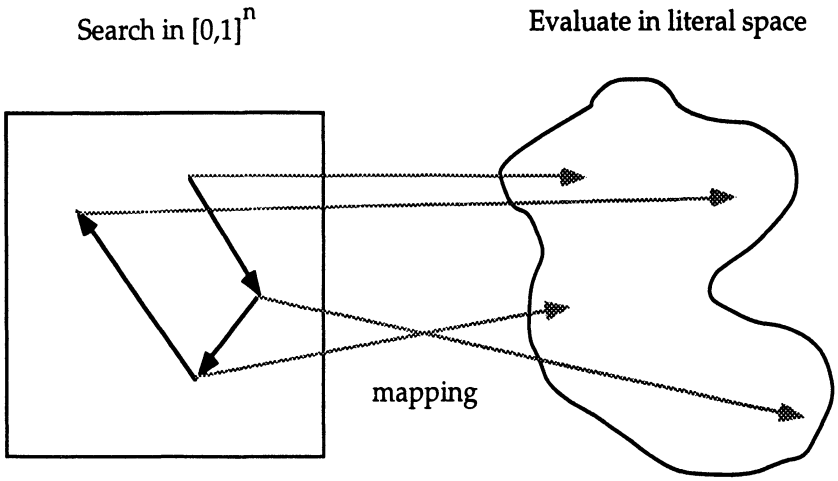
1. Random Keys

The random keys representation encodes a solution with *random* numbers. These values are used as sort *keys* to decode the solution. Random keys eliminates the offspring feasibility problem by using chromosomal encodings that represent solutions in a soft manner. These encodings are interpreted in the objective evaluation routine in a way that avoids the feasibility problem.

The primary difference between this encoding and those in the literature is the use of random numbers as tags to represent solutions. Random numbers are sampled from some space, typically $[0,1]^n$. The genetic algorithm searches that space as a surrogate for the literal space. Points in the random keys space are mapped to points in the literal space for evaluation. Figure 2 depicts this process. For this reason, the random keys approach is not similar to binary encodings. The generation of random numbers in the keys space

employs a sense of random search in conjunction with the genetic algorithm. No such random search occurs in binary encoding.

FIGURE 2: Random Keys Process



One advantage of this encoding is robustness to problem structure. The search over the keys space is similar in many of the problems discussed below. Variations in problem structure are captured in the mapping and the objective function value passed back. Mappings are problem specific, but generally involve sorting the random keys.

There are other encodings that use random variates (e.g. [1]) but not in the manner of random keys. Bagchi et al. ([2]) explore different scheduling problem encodings but none that are similar to random keys. The ARGOT strategy of Shaefer and Smith shares several characteristics with random keys. Both have chromosome spaces and literal spaces. Both take vectors in $\Re^n$ in the chromosome space and sort them to construct solutions. However, the ARGOT strategy uses problem specific, deterministic encodings that are translated from literal space to the chromosome space to form alleles. In random keys, alleles are generated randomly in the chromosome space. Further, random keys has no Lamarckian, backward translation as is central to the ARGOT strategy. In the ARGOT strategy the chromosome space shifts as a result of information passed back from the solution space. In random keys, the chromosome space is fixed. Random keys leads to a standard Darwinian genetic algorithm.

Following are several examples of problems for which random keys can be used to

design a genetic algorithm. For three of the problems we present substantive computational evaluations in Section 3. The others are presented to illuminate the robustness of the random keys representation. Computational tests on the remaining problems are topics of current research. No claim is made that random keys is the best approach for these problems.

1.1 Single Machine Scheduling Problem

For the single machine scheduling problem, create chromosomes where each gene corresponds to a job. To form an instantiation, generate a uniform $(0, 1)$ random deviate for each allele. The mapping to the literal space is accomplished by sorting the alleles and sequencing the jobs in ascending order of the sort. For a five job problem, the chromosome

$$(.46, .91, .33, .75, .51)$$

would represent the sequence

$$3 \rightarrow 1 \rightarrow 5 \rightarrow 4 \rightarrow 2.$$

This sequence can then be evaluated to compute total tardiness or any other regular measure. Note that many random key vectors would sort to the same sequence. While ties in the sort are unlikely, they are not problematic. Simply break them in a reasonable manner such as least index.

Crossovers are executed on the chromosomes, the random keys, not on the sequences. Consider two individuals:

$$(.46, .91, .33, .75, .51) \equiv 3 \rightarrow 1 \rightarrow 5 \rightarrow 4 \rightarrow 2$$

and

$$(.84, .32, .64, .04, .48) \equiv 4 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 1.$$

Using a traditional one-point crossover (as in Figure 1), assume that the crossover point is after the second gene. Then the two offspring are:

$$(.84, .32, .33, .75, .51) \equiv 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 1$$

and

$$(.46, .91, .64, .04, .48) \equiv 4 \rightarrow 1 \rightarrow 5 \rightarrow 3 \rightarrow 2.$$

Since any sequence of numbers can be interpreted as a sequence, all offspring are feasible solutions.

Through the dynamic of the genetic algorithm, jobs that should be early in the sequence evolve low numbers and jobs late in the sequence evolve large numbers. The random keys simply serve as tags which the crossover operator uses to rearrange jobs.

1.2 Multiple Machine Scheduling Problems

Genetic algorithms have been applied to more complicated scheduling problems in [6], [19], [32], [9], [34], [8], [17], [31], [21], and others. We present an alternative formulation based on random keys. See [3] or [10] for discussions of scheduling problems and terminology.

The approach in Section 1.1 can be extended to multiple machine problems with regular measures. Consider the m identical machine n job problem to minimize total tardiness. For each job generate an integer randomly in $\{1, \ldots, m\}$ and add a uniform $(0, 1)$ deviate. In the mapping, the integer part of any random key is interpreted as the machine assignment and the fractional parts sorted to sequence on each machine. A single sort gives the jobs assigned to machine 1 in processing order, followed by the jobs on machine 2, etc. Assuming that jobs are processed at their earliest possible time, a schedule can be constructed and evaluated for total tardiness. Section 3.1 contains some successful computational tests for these problems.

We have successfully generalized this approach to the job shop with precedence, release times, sequence dependent setups and nonregular measures such as a sum of weighted earliness and tardiness ([5]).

1.3 Vehicle Routing

Adapting the technique from multiple machine scheduling, we can handle the uncapacitated vehicle routing problem ([14]). Here we create a gene for each load. Randomly generate an integer for the vehicle and add a uniform $(0, 1)$ deviate to sequence stops. By

sorting the random keys we get the loads assigned to vehicle 1, in the order they are to be delivered, followed by the loads on vehicle 2, etc.

1.4 Generalized Traveling Salesman Problem

The generalized traveling salesman problem combines location and sequencing. Cities to be served are partitioned. We must choose one city from each partitioning set to visit and then sequence stops at the cities chosen. There is clear interdependence between the location and sequencing portions of the decision. For applications and more details see [26] (note that this definition of the generalized TSP is common in the operations research literature, but differs substantially from the use of that term in the computer science literature).

To represent this problem with random keys, designate one gene for each partitioning set. For set i, randomly choose an integer in $\{1, 2, \ldots, n_i\}$ where n_i is the number of cities in that set. Add a uniform $(0, 1)$ deviate. To construct a solution from a chromosome assume that, in each set, you visit the city indexed by the integer part of the allele. Then sequence the cities in ascending order of the fractional parts of the alleles. The dynamic of the genetic algorithm will simultaneously choose cities in each set and route them.

1.5 Resource Allocation Problems

Consider the linear binary program

$$\begin{aligned} &\max cx \\ \text{subject to}: \quad &Ax \leq b \\ &x \in \{0,1\} \end{aligned}$$

with the restriction that all elements of A and b are nonnegative. This is commonly referred to as a resource allocation problem since b can be viewed as the available resources, A as the consumption of resources by each option, x_j, and c as the profit vector. If A has only one row we have the $0-1$ knapsack problem. If A is a $0-1$ matrix we have the set partitioning problem. The related set covering problem is addressed by genetic algorithms in [22]. Various genetic approaches to constrained problems are analyzed in [27] and [24].

Random keys can be used to solve this common class of integer programs. Chromosomes have one gene for each variable, x_j. Generate a uniform $(0,1)$ for each variable. To map to the literal space, fix variables to 1 in ascending order of the keys, so long as all constraints remain feasible. As soon as any constraint becomes infeasible, set to 0 the last variable considered and all remaining variables. Objective evaluation is accomplished by multiplying this x vector by c. Through the dynamic of the genetic algorithm, variables that should be 1 will evolve low key values. Section 3.2 contains some successful computational tests on these problems.

1.6 Quadratic Assignment Problem

The quadratic assignment problem seeks to assign m agents to m locations, one to each location, to minimize a quadratic objective function. It has many applications such as facility layout ([20]) and has been attacked successfully by other heuristic search techniques such as simulated annealing and tabu search ([29]). The quadratic assignment problem can be stated mathematically:

$$\begin{aligned} & \min \quad cx + x^t Q x \\ \text{subject to}: \quad & \sum_{j=1}^{m} x_{ij} = 1, \quad \text{for all } i \\ & \sum_{i=1}^{m} x_{ij} = 1, \quad \text{for all } j \\ & x_{ij} \in \{0,1\}. \end{aligned}$$

Commonly, c is identically zero and an element of the matrix Q, q_{ijkl}, equals $f_{ik}d_{jl} + f_{ki}d_{lj}$ where f_{ik} is the material or communication flow from agent i to agent k and d_{jl} is the distance from location j to location l.

Random keys can be used to design a genetic algorithm for this problem as well. Chromosomes have one gene per agent. Generate a uniform $(0,1)$ deviate for each agent. To map to the literal space, sort and assign agents to locations according to the sort. For example, if $m = 4$ and sorting the indices $1,2,3,4$ by their random keys results in the sequence $3,1,4,2$, then assign agent 1 to location 3, agent 2 to location 1, etc. Evaluate

the assignment in the quadratic objective function. As above, all chromosomes resulting from crossovers represent feasible assignments. Section 3.3 gives computational experience for three quadratic assignment problems.

1.7 Discussion

The general structure of the random keys concept is:

1. Form each chromosome by generating random numbers for each decision.
2. From a given chromosome, derive a solution by sorting the random keys and taking the priorities from the sort.
3. All crossovers are done on the random keys, not the derived solutions.

Note that in step 1 deciding what entails a "decision" is a modeling issue of some importance.

The important feature of random keys is that all offspring formed by crossover are feasible solutions. This is accomplished by moving much of the feasibility issue into the objective evaluation procedure. If any random key vector can be interpreted as a feasible solution, then any crossover is feasible. Through the dynamic of the genetic algorithm, the system learns the relationship between random key vectors and solutions with good objective values.

2. Genetic Algorithm

The population of random key vectors must be operated upon by a genetic algorithm to breed good solutions. There are many variations of genetic algorithms formed by altering the reproduction, crossover and mutation operators. The reproduction and crossover operators determine which parents will have offspring, and how genetic material is exchanged between the parents to create those offspring. Mutation allows for random alteration of genetic material. Reproduction and crossover operators tend to increase the quality of the populations and force convergence. Mutation opposes convergence and replaces genetic material lost during reproduction and crossover. For details see [12].

The random keys representation scheme is not limited to implementation within the

genetic algorithm below. Many variations are yet to be investigated. However, the following algorithm has proved very robust. It is used in each computational test in Section 3.

Reproduction is accomplished by copying the best individuals from one generation to the next, called an elitist strategy ([12]). The advantage of an elitist strategy over traditional probabilistic reproduction is that the best solution is monotonically improving from one generation to the next. The potential downside is population convergence. This is overcome by high mutation rates described below.

Parametrized uniform *crossovers* ([30]) are employed in place of the traditional one-point or two-point crossover. After two parents are chosen randomly from the full, old population (including chromosomes copied to the next generation in the elitist pass), at each gene toss a biased coin to select which parent will contribute the allele. Returning to the single machine sequencing problem example above, assume that a coin toss of head selects the allele from the first parent, a tail chooses the allele from the second parent, and that the probability of tossing a head is 0.7 (this value was selected empirically). Below is one potential crossover outcome:

coin toss	*H*	*H*	*T*	*H*	*T*
parent 1	*.46*	*.91*	.33	*.75*	.51
parent 2	.84	.32	*.64*	.04	*.48*
offspring	*.46*	*.91*	*.64*	*.75*	*.48*

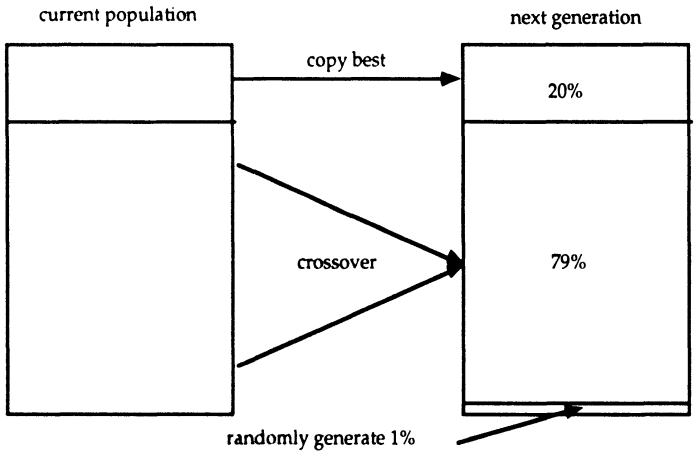
Rather than the traditional gene by gene *mutation* with very small probability, we employ the concept of *immigration*. That is, at each generation one or more new members of the population are randomly generated from the same distribution as the original population. This process prevents premature convergence of the population, like a mutation operator, and leads to a simple statement of convergence.

Figure 3 depicts the entire generational transition. A pseudocode statement of the algorithm in Exhibit I.

EXHIBIT I: Genetic Algorithm Pseudo Code

```
for each member of the population BEGIN [setup]
    generate random keys
    evaluate objective values
    END [setup]
while (not stop) BEGIN [transition]
    sort population by objective value
    copy top 20% of population [reproduction]
    for remainder of population BEGIN [crossover]
        randomly choose two parents from old population
        for each gene choose allele from parent 1 with prob.  0.7
            (from parent 2 with prob.  0.3)
        END [crossover]
    randomly generate 1% of population [mutation]
    evaluate objective values for new members
    stop if hit value target or generation count limit
    END [transition]
```

FIGURE 3: Generational Transition



For completeness we note that the algorithm converges in probability. The immigration operator samples a finite space. If a particular random keys implementation generates an optimal solution with a finitely positive probability, then the algorithm converges due simply to this sampling. However, the question of interest is the rate of convergence. If

genetic algorithms have true value for optimization, it must come from their ability to greatly speed up the rate of convergence over random sampling of the feasible region. A probabilistic derivation of the rate of convergence is a topic of current research. To date, justification for the algorithm lies in its empirically observed robustness. That is, this same framework has worked well for several problems tested. Below we present the output of some of that computational work.

3. Computational Results

To demonstrate the effectiveness and robustness of the random keys genetic algorithm we present computational results for three classes of problems: the multiple machine scheduling problem to minimize total tardiness, the resource allocation problem and the quadratic assignment problem. All computation below is reported in seconds on an IBM RS/6000-320H. Population sizes were fixed by running small pilot studies and held constant for all tests on a given problem set. The results presented attempt to find a solution within a few percent of the optimal or best known solutions. In real problem solving, the data are rarely known this accurately and such results are satisfactory. The algorithm takes substantially longer than the times presented to fine-tune the solution beyond this level.

3.1 Multiple Machine, Tardiness Scheduling Problems

Table 1 presents the results from a total of 20 runs of the genetic algorithm on two different problems.

Problem `fshx4` is a four identical machine, 200 job, total tardiness problem formed by merging the data for four single machine, 50 job problems reported in [11]. The optimal solution for this problem is unknown. A modified due date heuristic as in [4] found a value of 7924. The best known value for this problem is 7753. The computational test in Table 1, line `fshx4`, reports the results of 10 runs of the genetic algorithm, each with a different random seed, terminated when a solution with value within 2.5% of the best known value (7753) was discovered. A population size of 200 was used in all runs.

Problem `fshx8` is an eight identical machine, 400 job, total tardiness problem formed

TABLE 1: Computational results for scheduling
Computation required to get within 2.5% of the best known solution
over 10 random seeds

	#jobs	#m/c	generations			seconds		
			min	median	max	min	median	max
`fshx4`	200	4	120	129	136	270.1	290.6	305.8
`fshx8`	400	8	250	302	397	1213.6	1509.0	1923.5

by duplicating each job and each machine in `fshx4`. An upper bound on the optimal value of this problem is 15506, double the best value for `fshx4`, since a feasible solution can be found by replicating the solution from `fshx4` on the second set of machines. The modified due date heuristic found a solution with value 15825. Table 1, line `fshx8`, reports the results for 10 runs with different random seeds, terminated when the algorithm found a solution with value within 2.5% of the best known value (15506). A population size of 400 was used in all runs.

Since data are rarely known within 2.5%, the genetic algorithm essentially solves these difficult scheduling problems within a few minutes on a desktop machine. Two points are particularly notable. First, there is little variance across the random seeds. The algorithm appears very robust to this parameter. Second, the algorithm appears very scalable. That is, computation increases in a reasonable fashion as problem size is increased. However, no conclusions should be drawn from two problems.

3.2 Resource Allocation Problems

Table 2 presents the results from a total of 40 runs of the random keys genetic algorithm on four different resource allocation problems.

Problems `ra1` and `ra2` have 100 variables and five constraints and were randomly generated as in the generalized assignment problems type "D" in [23]. The multiple-choice constraints were not imposed. Hence, the problems could actually be solved as five separate 20 variable knapsack problems. This characteristic was not exploited. Lines `ra1` and `ra2` each report the outcomes of 10 runs with different random seeds. Each run was terminated when a solution with value within 2.5% of the optimal value was discovered. A population size of 400 was used on these runs. Each line also reports the time required for IBM's OSL

TABLE 2: Computational results for resource allocation
Computation required to get within 2.5% of the optimal solution
over 10 random seeds

	#var	generations			seconds			seconds
		min	median	max	min	median	max	OSL
ra1	100	81	95	260	93.2	109.2	298.0	26.5
ra2	100	67	84	92	77.2	96.5	105.8	51.4
ra1d	200	156	177	201	471.6	532.9	606.3	6485.2
ra2d	200	139	163	201	421.0	497.3	606.3	$> 50000^{\dagger}$

† Shut down after nearly fourteen hours without finishing the branch-and-bound. It had not found the optimal solution.

package to solve the problems to optimality on the same machine.

Problems ra1d and ra2d have 200 variables and five constraints and were formed by duplicating every variable and doubling each element of b in problems ra1 and ra2, respectively. This process has two effects: the problems are twice as large, and they become massively dual degenerate. That is, they have many optimal solutions. This characteristic is known to be very difficult for branch-and-bound codes, such as OSL, and is common in real data sets. The problems were designed to show the robustness of the genetic algorithm relative to branch-and-bound. Each run of the genetic algorithm was terminated when a solution with value within 2.5% of the optimal value was discovered. A population size of 600 was used on these runs. Each line also reports the time required for OSL to solve the problems to optimality.

Again, the genetic algorithm essentially solved these integer programs within several minutes on a desktop machine. Of particular note are the computation times relative to OSL on the larger, dual degenerate problems. The scalability and robustness across different data sets are apparent here.

3.3 Quadratic Assignment Problem

Not all problems that can be represented by random keys are solved by the genetic algorithm as easily as those discussed in Sections 3.1 and 3.2. The quadratic assignment problem was more difficult for the algorithm and points out some of its limitations.

Table 3 presents the results for a total of 30 runs on three different quadratic assign-

TABLE 3: Computational results for QAP
Computation required to get within 5% of the best known solution
over 10 random seeds

	#agents	generations			seconds		
		min	median	max	min	median	max
n15	15	25	232	5653	0.8	7.0	175.0
n20	20	34	183	1409	2.3	11.9	91.1
n30	30	520	7462	†	97.2	1409.3	†

† Two problems of the ten did not find a solution with value within 5% in the allotted 30000 generations. However, they did find a solution with value within 5.7% and 5.1% in 18350 and 2664 generations (3414.7 seconds and 493.9 seconds).

ment problems; the three Nugent problems discussed in detail in [29]. Each line reports the outcome of 10 runs with different random seeds. Each run was terminated when a solution with value within 5% of the best known value was discovered. A population size of four times the number of agents was used on each run. A relaxed target of 5% error was used here since the algorithm, for many seeds, did not find a solution within 2.5% in the allotted 30000 generations.

The tests on the quadratic assignment problem are not as impressive as those in Sections 3.1 and 3.2. The error is larger and the computation times less predictable for problems with smaller chromosomal representations. Neither was the test a complete failure. Thirty agent quadratic assignment problems are not easy problems. In most cases, the algorithm found reasonable solutions in reasonable times. These results are not as good as those reported in [29] for simulated annealing or tabu search. On the other hand, this code took no advantage of the structure of the quadratic assignment problem and, hence, is more robust to problem structure.

4. Summary and Conclusions

We present a robust genetic algorithm that can effectively address a wide range of sequencing and optimization problems such as multiple machine scheduling problems, resource allocation problems and quadratic assignment problems. There are undoubtedly many others that it can address.

Sections 3.1 and 3.2 show excellent computational results for scheduling and resource allocation problems. However, the algorithm does not perform equally well on all problems that can be formulated within its framework. Section 3.3 shows only moderate success for quadratic assignment problems.

We make no claim that this algorithm is the most efficient for any of the problems presented. The contribution here is that one code, with minor variations in the objective evaluation routine, can effectively address so many important problems. We do not compare this approach with other heuristics because a fair comparison would require that scheduling codes also solve quadratic assignment problems and integer programming problems; and that integer programming codes also solve scheduling and quadratic assignment problems. There are no such codes to the best of our knowledge. While OSL can be used on scheduling problems or quadratic assignment problems, note that the traditional formulation of problem `fshx8` would have on the order of 80,000 integer variables and disjunctive constraints that lead to very loose linear relaxation bounds.

To editorialize, one of the major barriers to use of operations research in industry is the massive library of special purpose codes necessary to address the range of problems faced by the firm. Implementation could be expanded greatly by the existence of generally applicable approaches that deliver good solutions in predictable and reasonable computation times. While much work is needed to establish that the random keys genetic algorithm is such an approach (in particular a probabilistic analysis of rate of convergence) these early computational tests are encouraging.

Acknowledgments

I would like to thank the participants of the Symposium on Operations Research and Complex Adaptive Systems at the Santa Fe Institute, May 24-26, 1992, for many insights that contributed greatly to this paper. I thank Atidel Hadj-Alouane for help with OSL and Bryan Norman for assistance with the literature search. Darrell Whitley provided many helpful comments and references. I would also like to thank Professor Jadranka Skorin-Kapov for the quadratic assignment problem data, Professor Nejat Karabakal for the resource allocation data and Professor Marshall Fisher for the scheduling data.

This research was supported in part by National Science Foundation grants DDM-9018515 and DDM-9202849 to the University of Michigan.

REFERENCES

1. T. BÄCK, F. HOFFMEISTER, AND H. SCHWEFEL, 1991. A Survey of Evolution Strategies, *Proc. of the Fourth International Conference on Genetic Algorithms*, 2-9.
2. S. BAGCHI, S. UCKUN, Y. MIYABE, AND K. KAWAMURA, 1991. Exploring Problem-Specific Recombination Operators for Job Shop Scheduling, *Proc. of the Fourth International Conference on Genetic Algorithms*, 10-17.
3. K. BAKER, 1974. *Introduction to Sequencing and Scheduling*, Wiley.
4. K. BAKER, J. KANET, 1983. Job Shop Scheduling with Modified Due Dates, *Journal of Operations Management 4*, 11-22.
5. J. BEAN, 1993. Methods for Rescheduling in the Matchup Paradigm, *Proc. of the 1993 NSF Design and Manufacturing Systems Conference 1*, Charlotte, NC, 777-782.
6. J. BIEGAL AND J. DAVERN, 1990. Genetic Algorithms and Job Shop Scheduling, *Computers and Industrial Engineering 19*, 81-91.
7. G.A. CLEVELAND AND S. F. SMITH, 1989. Using Genetic Algorithms to Schedule Flow Shop Releases, *Proc. of the Third International Conference on Genetic Algorithms*, 160-169.
8. U. DORNDORF AND E. PESCH, 1992. Evolution Based Learning in a Job Shop Environment, Working Paper. INFORM - Institut für Operations Research und Management GmbH, Pascalstraße 23, D-5100 Aachen, F.R.G.
9. E. FAULKENAUER AND S. BOUFFOIX, 1991. A Genetic Algorithm for Job Shop, *Proc. of the 1991 IEEE International Conference on Robotics and Automation*, 824-829.
10. S. FRENCH, 1982. *Sequencing and Scheduling*, Halsted Press.
11. M. FISHER, 1976. A Dual Algorithm for the One-Machine Scheduling Problem, *Mathematical Programming 11*, 229-251.
12. D. E. GOLDBERG, 1989. *Genetic Algorithms in Search Optimization and Machine Learning*, Addison Wesley.
13. D. E. GOLDBERG AND R. LINGLE, JR., 1985. Alleles, Loci, and the Traveling

Salesman Problem, *Proc. of the First International Conference on Genetic Algorithms.*

14. B. L. GOLDEN AND A. A. ASSAD, 1988. *Vehicle Routing: Methods and Studies*, North-Holland.

15. J. J. GREFENSTETTE, R. GOPAL, B. ROSMAITA, AND D. VAN GUCHT, 1985. Genetic Algorithms for the Traveling Salesman Problem, *Proc. of the First International Conference on Genetic Algorithms.*

16. J. J. GREFENSTETTE, 1987. Incorporating Problem Specific Knowledge into Genetic Algorithms, *Genetic Algorithms and Simulated Annealing*, (ed. L. Davis) Morgan Kaufman Publishers.

17. J. HERRMANN AND C-Y LEE, 1992. Solving a Class Scheduling Problem with a Genetic Algorithm, Technical Report, Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL, 32611.

18. J. H. HOLLAND, 1975. *Adaptation in Natural and Artificial Systems*, University of Michigan Press.

19. J. J. KANET AND V. SRIDHARAN, 1991. PROGENITOR: A genetic algorithm for production scheduling, *Wirtschafts Informatik 33*, 332-336.

20. A. KUSIAK AND S. S. HERAGU, 1987. The Facility Layout Problem, *European Journal of Operational Research 29*, 229-251.

21. C. LEE AND J. HERRMANN, 1993. Decision Support Systems for Dynamic Job Shop Scheduling, *Proc. of the 1993 NSF Design and Manufacturing Systems Conference 2*, Charlotte, NC, 1119-1123.

22. G. E. LIEPINS AND W. D. POTTER, 1990. A Genetic Algorithm Approach to Multiple-Fault Diagnosis, *Handbook of Genetic Algorithms*, 237-250.

23. S. MARTELLO AND P. TOTH, 1990. *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley and Sons.

24. Z. MICHALEWICZ AND C. JANIKOW, 1991. Handling Constraints in Genetic Algorithms, *Proc. of the Fourth International Conference on Genetic Algorithms*,

151-157.

25. R. NAKANO, 1991. Conventional Genetic Algorithm for Job Shop Problems, *Proc. of the Fourth International Conference on Genetic Algorithms*, 474-479.

26. C. NOON AND J. BEAN, 1991. A Lagrangian Based Approach to the Asymmetric Generalized Traveling Salesman Problem, *Operations Research 39*, pp. 623-632.

27. J. RICHARDSON, M. PALMER, G. LIEPENS AND M. HILLIARD, 1989. Some Guidelines for Genetic Algorithms with Penalty Functions, *Proc. of the Third International Conference on Genetic Algorithms*, 191-197.

28. C. SHAEFER AND S. SMITH, 1990. The ARGOT Strategy II: Combinatorial Optimization, Thinking Machinesnical Report RL90-1.

29. J. SKORIN-KAPOV, 1990. Tabu Search Applied to the Quadratic Assignment Problem, *ORSA Journal on Computing 2*, 33-45.

30. W. M. SPEARS AND K. A. DE JONG, 1991. On the Virtues of Parameterized Uniform Crossover, *Proc. of the Fourth International Conference on Genetic Algorithms*, 230-236.

31. R. H. STORER, S. D. WU, AND R. VACCARI, 1992. New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling, *Management Science 38*, 1495-1509.

32. G. SYSWERDA, 1991. Schedule Optimization Using Genetic Algorithms, in *Handbook of Genetic Algorithms*, L. Davis (ed), Van Nostrand, 332-349.

33. D. WHITLEY, T. STARKWEATHER AND D. FUQUAY, 1989. Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator, *Proc. of the Third International Conference on Genetic Algorithms*, 133-140.

34. D. WHITLEY, T. STARKWEATHER AND D. SHANER, 1991. The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination, *Handbook of Genetic Algorithms*, L. Davis (ed), Van Nostrand, 350-372.