Pergamon

0305-0548(93)E0024-N

Computers Ops Res. Vol. 22, No. 1, pp. 135–147, 1995

0305-0548/95 $7.00+0.00

GENETIC ALGORITHM CROSSOVER OPERATORS FOR ORDERING APPLICATIONS

P. W. POON[1]† and J. N. CARTER[2]‡

[1]Cambridge University Engineering Department, Trumpington Street, Cambridge CB2 1PZ, England
and [2]AEA Petroleum Services, Winfrith, Dorset DT2 8DH, England

Scope and Purpose—The Genetic Algorithm is widely recognized as a powerful and widely applicable optimization method. It has been very successful when applied to problems that can be coded naturally as binary strings. However, ordering problems are more naturally coded as ordered lists and there is no standard Genetic Algorithm for manipulating such representations. The main difficulty is the design of a suitable crossover operator. We suggest an operator that can be applied to several problems without modification.

Abstract—In this paper, we compare the performance of several crossover operators, including two new operators and a new faster formulation of a previously published operator. This new formulation performs better than the other operators we have tested while taking no more computation time. In addition, with practical applications in mind, we show how the use of problem specific information can improve the performance of the Genetic Algorithm and we describe a method for designing problem specific crossover incorporating a novel tie-breaking algorithm.

1. INTRODUCTION

Since Holland introduced the Genetic Algorithm (GA) in 1975 [12], many researchers have become interested in it both as a research topic in its own right and as a new method for solving large and/or difficult problems in search and optimization [6].

Much of the GAs attraction lies in the fact that it is a "weak" method (i.e. it is one that can be applied to many different problems) while also being a powerful method. For a method to have both these characteristics is often regarded as a contradiction—we expect to sacrifice power for increased generality and vice versa. The GA can overcome this by forcing every problem to be expressed as a binary string which is a coding it can exploit. Where a natural mapping to such a representation exists, the result is an algorithm that performs well [3].

However, there is a large class of problems which do not easily map to binary strings. These include ordering or permutation problems of which the Travelling Salesman Problem (TSP), a classic test for new optimization techniques, is the best known. As well as the academic interest in the TSP, there is considerable interest in solving real-life ordering problems such as bin-packing and job shop scheduling.

These have not been neglected by GA researchers (e.g. see [2, 17, 18]) but by limiting their attention to one problem only, many researchers have designed GAs that perform well on that problem but not on other problems [7] and so have sacrificed generality for power. While this is

†Pui Wah Poon graduated from Cambridge University with a B.A. degree in Mathematics, and has recently completed a Ph.D. degree in Engineering. Her Ph.D. research at Cambridge University, which involved using the Genetic Algorithm in an automated method for performing realistic nuclear fuel management tasks, won her the Institution of Nuclear Engineers' Graduate Prize. She now works for AEA Technology in the U.K. performing strategic and safety assessment work.

‡Jonathan Neil Carter graduated from Southampton University with a B.Sc. in Mathematics. His Ph.D. in Theoretical Physics was awarded by Warwick University in 1989. After lecturing mathematics at Coventry University, he joined the oil technology group of AEA Technology in the U.K. where he is currently involved in the optimization of production facilities for oil installations in the North Sea.

inevitable and valid where a solution to a particular problem is being sought, it remains an open research area to find a GA for ordering problems that is comparable in applicability and performance to the classical GA for binary-string representations. Such an algorithm would be the starting point for all GA ordering applications.

The main difficulty encountered when using non-standard representations is the design of a suitable crossover operator which must combine relevant characteristics of the parent solutions into a valid offspring solution. In this paper, we introduce two new crossover operators which use a novel tie-breaking algorithm and we give a faster new algorithm for performing a previously published operator.

The performance of the various operators is tested on two TSPs for which exact solutions are known and Oliver's 30-city TSP [13]. For the last, we investigate the effect of using the symmetry of the TSP to reduce the size of the search space. In addition, we study two more practical problems involving delivery scheduling and factory design.

From this study, we identify several points that will be of interest to those applying the GA to real-life ordering problems. These include the choice of crossover operator and the importance of using problem specific information, where it is available, to reduce the search space or design a problem specific crossover operator. A method for the latter is given with a practical example.

We commence the paper with a list of representations and associated crossover operators for ordering problems. We then describe our test problems, explaining how we use the symmetry of the TSP to reduce the search space. We present results from using the various crossover operators on the test problems. Finally, we discuss the relevance of this work to ordering applications.

2. CROSSOVER OPERATORS FOR ORDERING PROBLEMS

The role of the crossover operator is to recombine information from two good "parent" solutions into what we hope are even better "offspring" solutions. The problem is to design a crossover operator that combines characteristics of both "parents" while producing a valid solution to the ordering problem.

The form of the crossover operator depends on the way the problem is coded. In this section we describe some possible representations and list crossover operators that have appeared in the literature as well as introducing two new crossover operators and a new algorithm for performing a previously published operator.

2.1. Crossover in Permutation representation

The Permutation representation appears most often in the literature. Each trial solution is simply represented as the ordered list. For example, in a 6-city TSP, the string (b d c e a f) means that city b is visited first, then city d and so on.

The main attraction of this representation is that it is clear and natural (or obvious) and thus fulfills one of the basic rules in Genetic Algorithm Design that the coding of a problem should be a natural expression of it [6]. However, the genetic operators used in the classical GA do not give valid offspring tours so new operators must be designed.

Crossover operators which have been suggested in the literature include the Partially Mapped Crossover (PMX) [8], Order Crossover (OX) [13], Order Crossover #2 (OX2) [19], Position Based Crossover (PBX) [19] and Cycle Crossover (CX) [13].

2.2. Crossover in Adjacency Listing representation

Many of the applications of the GA to ordering problems have concentrated on the TSP for which the relevant information is not the position of each element in the string but which elements are adjacent. This was the motivation for the Adjacency Listing representation first introduced by

Whitley *et al.* [20]. A parent string (or tour for the TSP) is translated into a list of cities with their associated neighbours. For example (b d c e a f) encodes as:

city	neighbours
a	e f
b	d f
c	d e
d	b c
e	a c
f	a b

Whitley *et al.* developed the Genetic Edge Recombination operator [20] for use with this representation. A superior version of this, the Enhanced Edge Recombination (EER) operator was later described by Starkweather *et al.* [18].

2.3. *Crossover in Position Listing representation*

Often, scheduling and ordering problems are more concerned with the actual position of each element in the string. In the Position Listing representation, each location in the string represents one of the elements in the ordered list and the character in that location is the position of that element in the ordered list. For example, to encode the list (b d c e a f), we start with a reference list of elements such as (a b c d e f). Then each character in the Position Listing corresponds to that element's position in the permutation, i.e.

Element	Position Listing
	(a, b, c, d, e, f)
b	(–, 1, –, –, –, –)
d	(–, 1, –, 2, –, –)
c	(–, 1, 3, 2, –, –)
e	(–, 1, 3, 2, 4, –)
a	(5, 1, 3, 2, 4, –)
f	(5, 1, 3, 2, 4, 6)

The advantage of this representation is that it explicitly records where each element appears in the ordered list. The difficulty is that traditional crossover does not produce valid solutions. For example, consider the following:

			a b c d e f
Parent 1:	(f b d e c a)	→	(6 2 5 3 4 1)
Parent 2:	(b d c e a f)	→	(5 1 3 2 4 6)
Crossover between points 3 and 5:			↓
Offspring 1:	??	←	(6 2 3 2 4 1)
Offspring 2:	??	←	(5 1 5 3 4 6)

The problem is that the resulting offspring solutions have ties for certain positions in the ordered list and no elements for other positions, e.g. Offspring 1 in the above example has b and d competing for the second position in the ordered list but no element marked to go in the fifth position. Valid offspring solutions could therefore be produced by breaking these ties in some way. This is exactly what is done by the two new operators Tie-Breaking Crossover #1 (TBX1) and Tie-Breaking Crossover #2 (TBX2) which are introduced in this paper and detailed in Figs 1 and 2 respectively.

2.4. *Crossover in Precedence Matrix representation*

The Precedence Matrix representation was suggested recently by Fox *et al.* [5]. The matrix stores all the elements that come before or after each element in the ordered list. If we have n elements

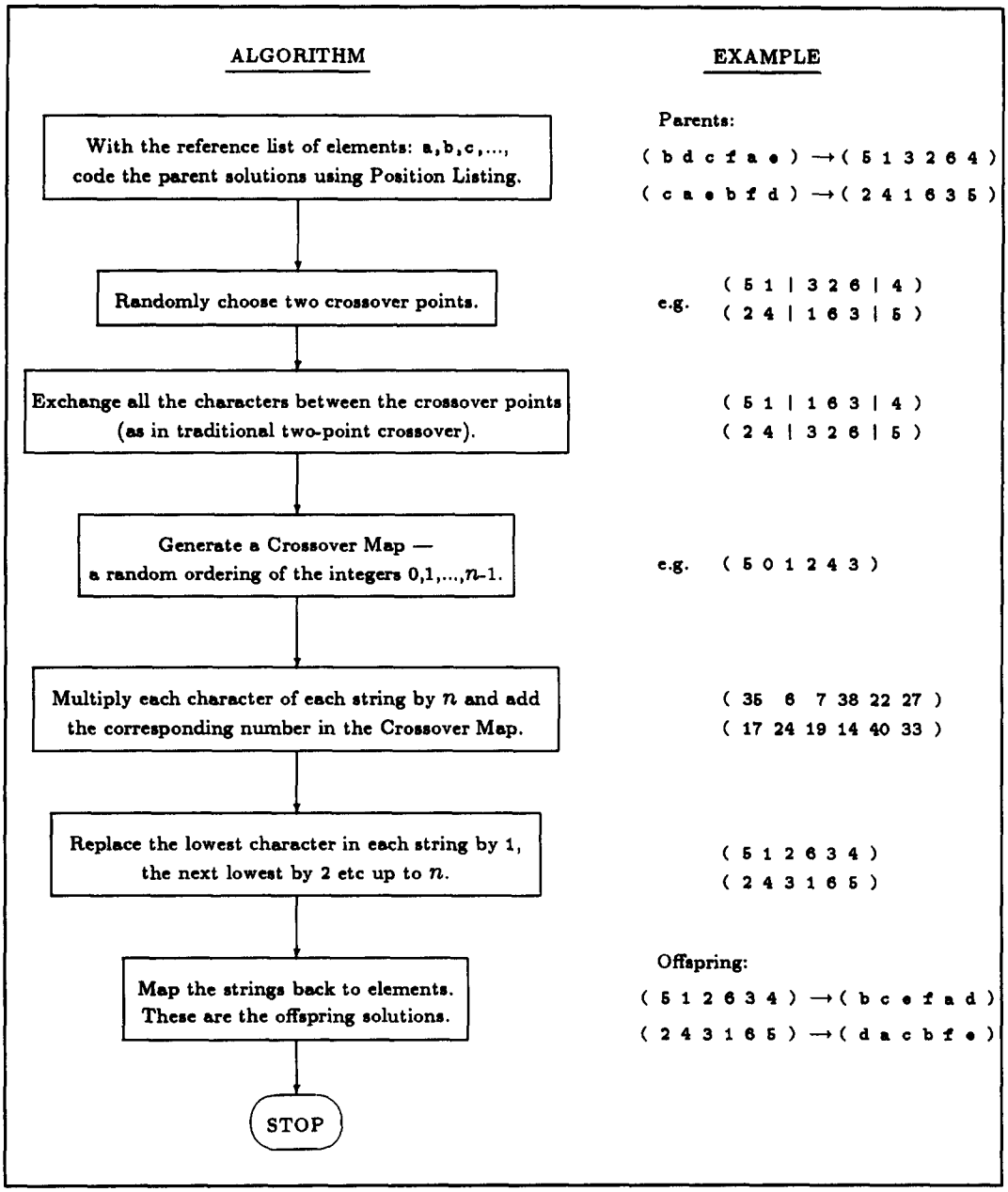

Fig. 1. The Tie-Breaking Crossover #1 (TBX1).

$x_1, \ldots, x_n$, the matrix is a $n \times n$ matrix **M** where

$$\mathbf{M}(x_i, x_j) = 1 \Leftrightarrow x_i \text{ precedes } x_j \text{ in the sequence}$$

$$\mathbf{M}(x_i, x_j) = 0 \Leftrightarrow \text{otherwise.}$$

For example, the sequence (b c d e a f) becomes

	a	b	c	d	e	f
a	0	0	0	0	0	1
b	1	0	1	1	1	1
c	1	0	0	1	1	1
d	1	0	0	0	1	1
e	1	0	0	0	0	1
f	0	0	0	0	0	0

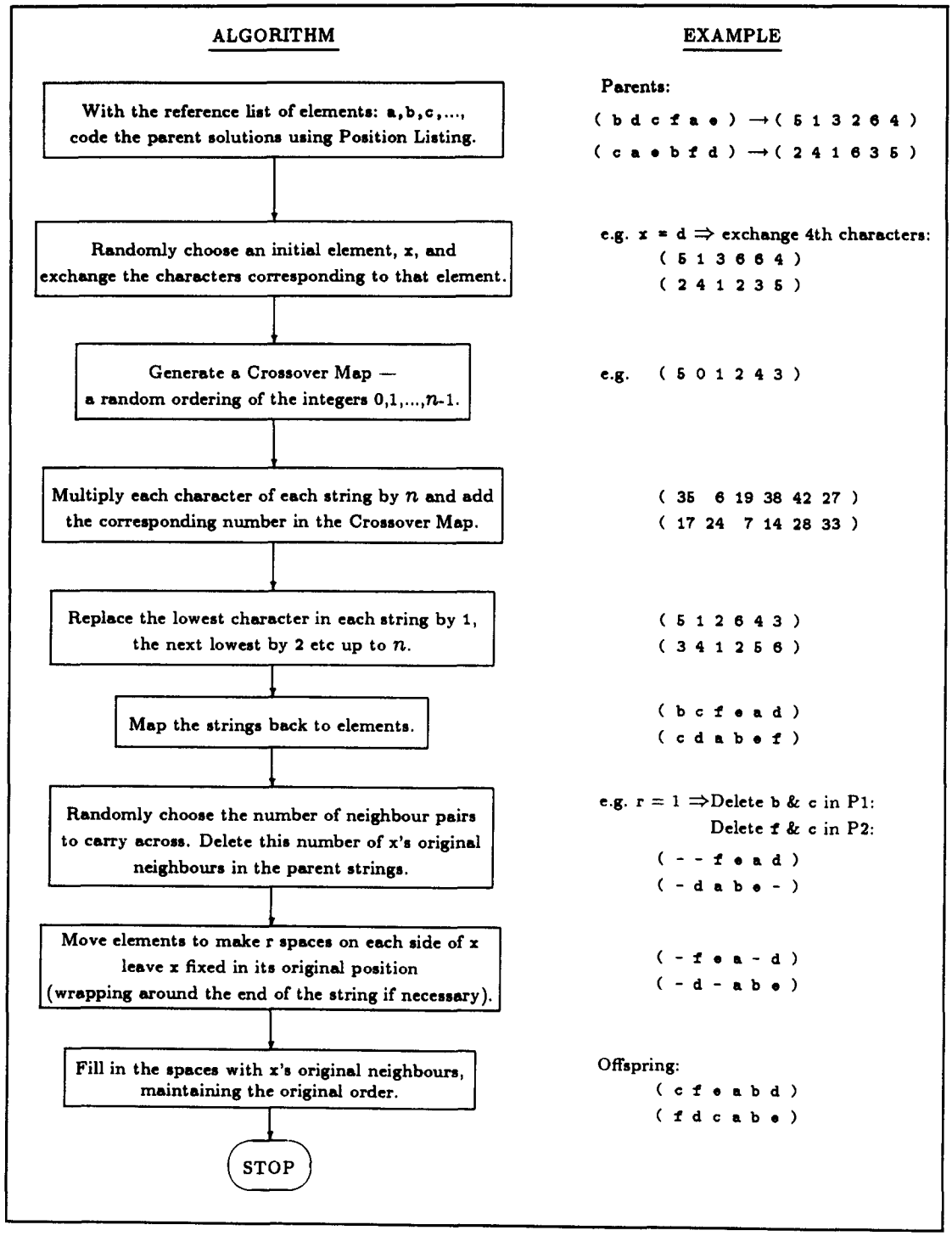

Fig. 2. The Tie-Breaking Crossover #2 (TBX2).

This can be translated back to the usual Permutation representation by adding up the column totals. Fox *et al.* [5] suggest two crossover operators to use with this representation. The first, called Intersection (IX), preserves all precedence relationships which are common to both parents. The second, called Union (UX), combines some precedence relationships taken from each parent. Fox *et al.* report that these operators perform well when compared with other crossover operators on the basis of the number of objective function evaluations required to find equally good solutions.

However, the computation time required by these operators is high compared to the others we

have described. For example, Fox *et al.* found that the GA using UX took about 30 times as long as that using PMX to run the same number of generations. In most real-life applications, the reason for using the GA is to find near-optimal solutions without excessive computing time so this time factor may not be acceptable. In fact, UX need not be performed in the way that Fox *et al.* describe. In Section 2.6, we introduce UX2, a new operator that does exactly the same operations as UX but in significantly less computing time.

2.5. Crossover in other representations

The above list of representations and crossover operators is not intended to be exhaustive. Other representations include the Ordinal and Adjacency representations [11]. However, the crossover operators designed for these representations are unsatisfactory and they are not included in this study.

2.6. The Union crossover in Permutation representation

The most time-consuming part of the UX algorithm is filling in the precedence matrix. UX2 produces exactly the same offspring solutions as UX but without going into the Precedence Matrix representation. Instead, we note that what the UX operator actually does is take a mutually exclusive substring from each parent string, preserve the precedence relationships of those substrings and then randomly allocate the remaining precedence relationships. This can be done more efficiently in Permutation representation by selecting the substrings and then writing the elements directly to the offspring string ensuring the precedence relations are preserved. Figure 3 describes this in detail with an example. The resulting algorithm, which we call UX2, has exactly the same effect as UX.

3. A COMPARISON OF CROSSOVER OPERATORS

3.1. The problem set

We have used the following six problems to test the ten crossover operators of the previous section.

Problem 1: a 20-city Hamiltonian Path TSP. In this Hamiltonian Path TSP, the salesman can start at any city and finish at any other city visiting every other city exactly once. The twenty cities lie along the real line at the integers 1 . . . 20. The two equivalent but opposite optimum routes are $(a_1, a_2, a_3 \ldots, a_{19}, a_{20})$ and $(a_{20}, a_{19}, \ldots, a_3, a_2, a_1)$, each with a journey length of 19 units.

Problem 2: a 20-city Hamiltonian Cycle TSP. In the Hamiltonian Cycle TSP, the salesman must start and finish at the same city. The 20 cities lie equally spaced on the edge of a circle and journeys are measured around the circle. There are 40 equivalent optima, since we may start at any of the 20 cities and travel either clockwise or anti-clockwise, e.g. $(a_7, a_8, a_9, \ldots, a_{19}, a_{20}, a_1, a_2, \ldots, a_6, a_7)$, each with a journey of 20 units.

Problem 3: Oliver's 30-city Hamiltonian Cycle TSP. This Hamiltonian Cycle TSP used by Oliver *et al.* [13] is a classic test problem for GAs. The 30 cities whose coordinates are given in appendix A are located on a 2-D plane. There are 60 equivalent optimum solutions with length 424 units.

Problem 4: Oliver's 30-city problem with reduced search space. This is identical to problem 3 except that the symmetry is exploited to reduce the size of the search space. Since tours are cyclic and can be traversed in two direction, each offspring tour is cycled and/or reversed as necessary so that a_1 is the first city in the list and a certain randomly chosen city appears in the second half of the string. This allows the GA to recognise equivalent tours as such. For a string of length n this reduces the search space from $n!$ to $\frac{1}{2}(n-1)!$.

In many real-life problems, there is similar information that can be used to reduce the search space. For ordering problems, the size of the search space grows exponentially with the length of the list to be ordered and so any methods available to reduce the space should be exploited. GAs should be flexible enough to perform well when such problem specific information is used. Studying this problem allows us to investigate the performance of the various crossover operators when used with such information.

Problem 5: delivery scheduling. This is a typical delivery scheduling problem were goods must be delivered from a central warehouse to a number of different destinations, each with a different requirement. The problem we use is an extension of the problem described by Foulds [4]. There are 14 towns to which deliveries have to be made from a central warehouse in town 15. The delivery

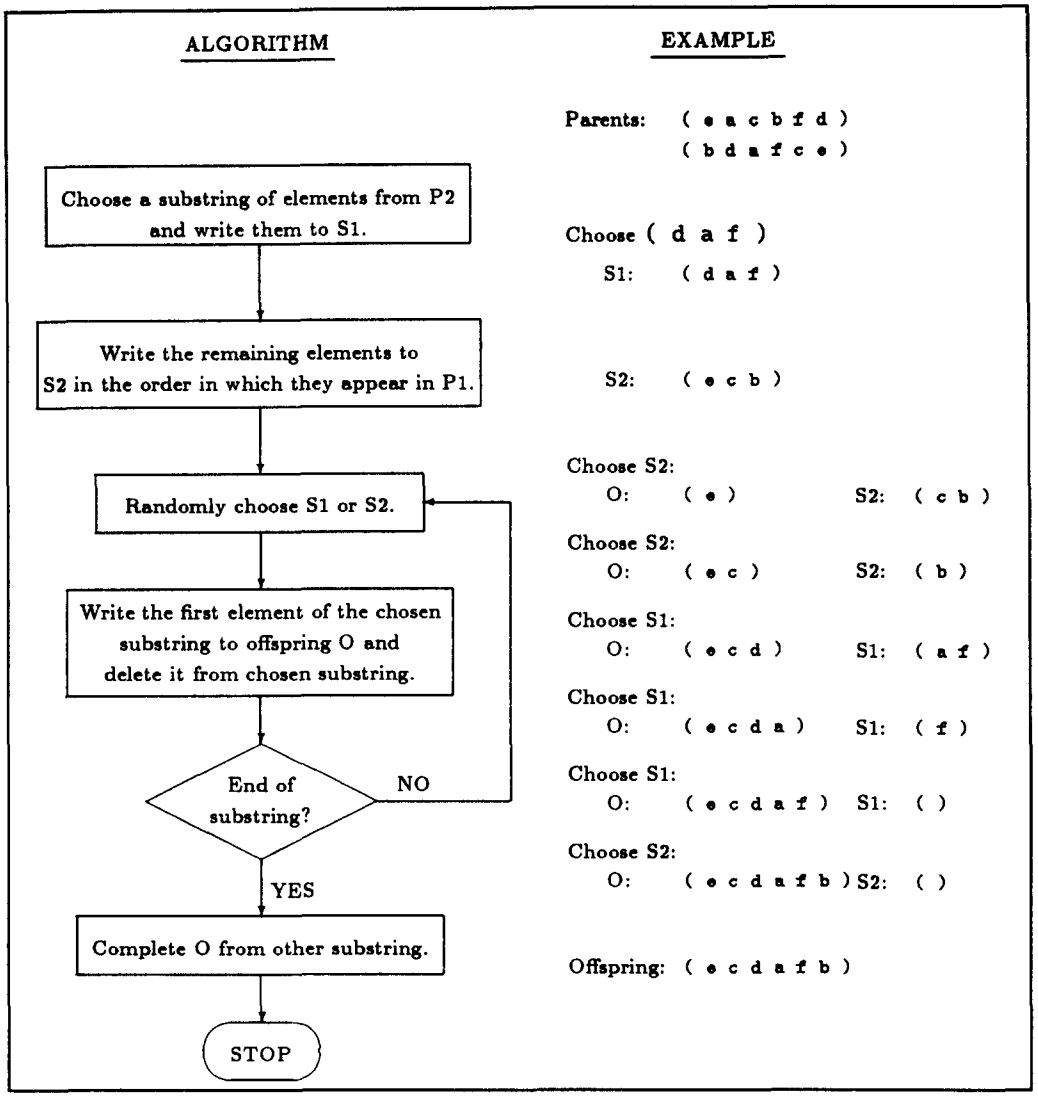

Fig. 3. The Union Crossover #2 (UX2)—a faster way to perform Union Crossover.

van can carry a maximum of 40 units, and must take complete deliveries to each town. The details of the distances and the required deliveries are given in Appendix A. The aim of the problem if to minimize the total distance travelled by the van. A minimum of 4 separate journeys are needed to complete the deliveries, however, more than this might be needed to minimize the total distance travelled. The worst case scenario is that the van should return to the warehouse after each delivery. The problem is then to find the order of the towns $a_1 \ldots a_{14}$ and 13 copies of a_{15} (a total of 27 elements) that minimize the total journey subject to the delivery constraint. The best solution discovered during our calculations is given in Appendix B.

Problem 6: factory design. This is another extension of a problem given by Foulds [4]. A new factory is to be composed of a number of separate work areas, 20 in total. The relative frequency of journeys between various work areas is known, and given in Appendix A. The problem is to maximize the number of trips to adjacent work areas. This is achieved by using the order of the areas to construct a maximally planar graph, where the edges have the values for the number of journeys undertaken. Starting with a graph defined by the first four elements, each new area is added to the previous graph so as to maximize the edge sum of the new graph. One difficulty with the algorithm is that if the number of work areas is greater than 17 then the number of graphs that can be formed is greater than the number of different orderings of the areas. However, in most cases the optimum graph can be constructed from one of the possible orderings. The best solution discovered during our calculations is given in Appendix B.

3.2. Implementation details

Starting with a randomly generated initial population, our GA program produces generations of offspring populations using a given crossover operator and Swap mutation [13]. We use an "Elitist" strategy, i.e. the best member in each generation is guaranteed to pass into the next generation unaltered, and fitnesses are scaled using Ranking [1]. This GA is repeated 25 times, always starting with the same initial population but with a different random number seed.

We use a population size of 21 for all the problems. Problems 2–6 are run for 300 generations but results are displayed for problem 1 after only 100 generations reflecting the faster convergence for this problem. The probability ratio for performing mutation/crossover is 0.2/0.8.

These parameter settings are not intended to be optimal. In particular, we use a much smaller population size than is usual. (It is likely that a larger population size is optimal—see Section 3.3.) However, these conditions allow us to investigate the effect of repeatedly applying the genetic operators.

3.3. Results

The results from running the GA with each of the crossover operators on each of the test problems are displayed in Tables 1 and 2 as the mean and standard deviation of the best solutions found by the 25 GA runs after the given number of generations. In this section, we highlight some of the interesting features of these results.

Table 1. Results for the test problems with a population size of 21

	Problem 1 Generation 100		Problem 2 Generation 300		Problem 3 Generation 300	
	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
TBX1	58.60	11.03	44.44	6.70	829.00	72.69
CX	67.64	11.65	48.40	7.75	881.84	52.48
PMX	54.96	9.87	45.28	6.35	812.56	73.47
TBX2	61.00	12.44	48.88	6.86	802.96	61.51
EER	50.32	7.40	41.12	8.87	804.00	55.76
OX	50.52	13.11	32.16	8.33	636.32	55.33
OX2	59.40	14.87	46.24	8.01	791.32	70.53
PBX	86.54	8.58	51.60	6.83	923.88	35.78
IX	68.68	14.07	52.40	5.74	866.64	52.51
UX2	40.76	5.66	29.12	6.83	644.64	48.43
	Problem 4 Generation 300		Problem 5 Generation 300		Problem 6 Generation 300	
	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
TBX1	803.24	61.56	210.00	11.33	1422.64	19.49
CX	800.92	77.48	217.20	14.91	1414.32	15.42
PMX	648.12	82.47	203.48	12.57	1430.00	17.21
TBX2	701.44	86.54	213.48	14.63	1422.80	20.90
EER	797.24	71.02	208.16	10.55	1420.04	25.81
OX	621.48	55.18	196.80	11.60	1427.36	15.00
OX2	786.24	63.15	209.32	11.91	1439.04	17.44
PBX	926.20	33.06	234.20	11.49	1396.44	10.28
IX	844.36	62.68	221.76	11.95	1394.88	16.33
UX2	624.92	30.37	188.92	8.76	1452.92	4.43

Table 2. Results for problem 3 with a population size of 31

	Problem 3 Generation 200	
	$\bar{x}$	σ
TBX1	787.40	58.03
CX	832.24	56.42
PMX	759.60	70.28
TBX2	772.92	70.49
EER	721.72	68.06
OX	629.00	52.43
OX2	751.72	65.55
PBX	947.88	33.54
IX	863.92	69.83
UX2	667.80	52.50

Good operators and bad operators. Because we have studied a range of problems, we might expect that the operators would perform well on some problems and badly on others. In fact, PBX and IX are the worst performers on all the problems closely followed by CX. This is because none of these are good at preserving substrings of neighbouring elements or "building-blocks". OX performs quite well on all the problems but the best overall performance is from UX2 crossover which is either the best or second best operator for every problem. Although UX2 does not always preserve a long substring like OX and PMX, it does preserve some, often shorter, substrings and precedence relationships.

Using symmetry to reduce the search space. Comparing the results for problems 3 and 4 show that reducing the search space for Oliver's TSP improved the performance of the GA for most of the operators. The most noticeable improvements are for CX and TBX2. These operators are good at preserving the actual locations of elements in a string (rather than the locations of elements relative to each other) which become more significant when the search space is reduced in this way.

The effect of local minima. The interesting feature of problems 1 and 2 is the existence of strong local minima. These have a tour-length of 37 and 38 units respectively and correspond to tours that traverse the line or circle once in each direction, e.g. for problem 2,

$$(a_7, a_9, a_{12}, a_{14}, a_{15}, a_{17}, a_1, a_3, a_6, a_5, a_4, a_2, a_{20}, a_{19}, a_{18}, a_{16}, a_{13}, a_{11}, a_{10}, a_8)$$

(in this example, we start at a_7, travel around the circle to a_6 and then change direction travelling round the circle the other way to reach a_7 again).

All the GAs we have run have some difficulty with this which accounts for the high average results of over 37 and 38 in most cases. Only UX2 and OX manage to escape the local optima in a reasonable number of runs.

Population size. The population size is an important parameter. Because this study is primarily about crossover operators, we have used the same population size for all the problems and operators to aid the comparison. However, results for problem 3 using a population size of 31 for 100 generations (so that the number of objective function evaluations is the same) show that the GA performance can be improved by careful choice of population size (see Table 2).

4. APPLYING THE GA TO REAL-LIFE ORDERING PROBLEMS

In this section, we discuss how the work presented in this paper can be applied to real-life optimization problems involving ordering. We start with a discussion of the new TBX operators introduced and explain how they can be used as "intelligent" crossover for ordering problems. We then discuss how the computation required to execute the crossover operators can affect the choice of crossover operator.

4.1. Using tie-breaking for intelligent crossover

In this section we describe the action of the new tie-breaking crossover operators and explain how they differ from other crossover operators in the literature. We also discuss how they might be used in practical GA applications with an example taken from nuclear reactor fuel management.

Recall from Section 2.3 that the TBX operators were designed to break ties between two string locations competing for the same location. In the example given, classical crossover in Position Listing representation resulted in the following:

a	b	c	d	e	f
(6	2	3	2	4	1)

with b and d competing for the second location. This tie is broken by multiplying by the string-length (6) and adding a random crossover map, e.g. (*5 2 0 1 3 4*), then ranking to obtain the offspring solution:

a	b	c	d	e	f	→	a	b	c	d	e	f
(41	14	18	13	27	10)		(6	3	4	2	5	1)

The "tie" has been broken randomly, by the crossover map, with "d" winning the second location. The losing element, "b", is awarded a neighbouring location. The other elements are shifted along by one location until the unallocated location, 5, is filled. Thus in the offspring solution, each element is in or close to a location inherited from one parent solution.

The tie-breaking algorithm (i.e. multiplying by the string-length, adding a random crossover map and ranking) can be used to good effect, without using Position Listing representation, on ordering problems where elements can be ranked by some characteristic.

The example we give is optimizing fuel placement in a nuclear ractor, which is a large non-linear problem with computationally intensive objective function and constraints [15]. The 48 fuel elements are ranked by their reactivity, which is known, and the rank is used to identify each fuel element in the GA representation:

Fuel inventory:						
Fuel element	A	B	C	D	E	F
Reactivity	1.1	1.2	0.9	1.0	1.0	1.3
Rank	4	5	1	2	3	6
An example of a trial solution:						
Loading pattern:	B	D	F	E	A	C
GA representation:	5	2	6	3	4	1

In this representation, classical crossover can be performed and the tie-breaking algorithm used to produce valid ordered lists which can be mapped back to fuel elements. The resulting problem specific crossover, HTBX, easily out-performs other crossover operators including TBX2 and UX2 [14].

In many other practical problems, there will be information which can be used to rank the elements which must be ordered. For example, in scheduling problems, the jobs to be scheduled can be ranked by priority or the time required to process each job if these are known. The resulting GA is more "intelligent" because it uses more information and the tie-breaking algorithm described in this paper provides a suitable crossover operator.

4.2. Computation time requirement

In our results we have compared the crossover operators on the basis of improvement in function value against generation (equivalently, number of trial solutions evaluated). However, for practical purposes, it is improvement in function value against computation time that is important. For our 20-city Hamiltonian Path problem (problem 1), we have observed the following relative timings.

Time	TBX1	CS	PMX	TBX2	EER	OX	OX2	PBX	IX	UX	UX2
Crossover	16	7	7	17	37	7	8	11	87	257	9
Function evaluation	3	3	3	3	3	3	3	3	3	3	3
Total per generation	19	10	10	20	40	10	11	14	90	260	12

UX2, our new formulation of UX which produces the same offspring, is much faster than UX and is comparable in speed to the other operators. UX2 is a much simpler algorithm and should be used in preference to UX.

Of the other operators, whether it is acceptable to use the more time-consuming operators such as IX, EER or TBX2 will depend on the time required to evaluate the objective function as well as how well they perform. For this problem, using EER is slower than, say, PMX even if it converges onto the same near-optimal solutions in half the number of generations. However, for many practical problems, such as the nuclear fuel management problem described in Section 4.1, the evaluation of the objective function can be hundreds of times slower. In these cases, a factor of 3 or 4 difference in the time taken to execute the crossover is negligible.

5. SUMMARY

In this study, we hoped to identify a crossover operator for ordering problems that could be

comparable in applicability and performance to the classical crossover used in binary-string GAs. UX appears to be such an operator since it performs well on all the test problems we have studied.

However, UX takes a long time to run compared to the other crossover operators in this study. In this paper, we have introduced a new faster algorithm for performing UX, UX2, which is comparable in speed to the fastest crossover operators we have used. We therefore recommend UX2 as a good starting point when applying the GA to ordering problems.

We have also introduced a novel tie-breaking algorithm which can be used for crossover in the Position Listing representation or which can be used to design problem specific crossover operators when there is information available to rank the elements to be ordered. We have given an example of where the latter has been extremely effective in solving a difficult real-life ordering problem.

Another way to exploit problem specific information is to use any symmetry in the problem to reduce the size of the search space. We have shown that this can improve the performance of the GA.

6. CONCLUSIONS

The GA can be a useful tool for solving practical ordering problems. Its performance can be improved by exploiting any information that is available additional to the objective function values. Such information could be used to reduce the size of the search space or incorporated into the crossover operator. Where the latter is not possible, the new formulation of UX is both fast and effective.

Acknowledgements—The first author gratefully acknowledges the financial support of the Science and Engineering Research Council (SERC) and of Nuclear Electric PLC.

REFERENCES

1. J. E. Baker, Adaptive selection methods for genetic algorithms. *Proc. First Int. Conf. Genetic Algorithms and their Applications*, pp. 101–111. Erlbaum (1985).
2. L. Davis, Job shop scheduling with genetic algorithms. *Proc. First Int. Conf. Genetic Algorithms and their Applications*, pp. 136–140. Erlbaum (1985).
3. K. A. De Jong and W. M. Spears, Using genetic algorithms to solve NP-complete problems. *Proc. Third Int. Conf. Genetic Algorithms and their Applications*, pp. 124–132. Kaufmann (1989).
4. L. R. Foulds, *Combinatorial Optimization for Undergraduates*. Springer, New York (1984).
5. B. R. Fox and M. B. McMahon, Genetic operators for sequencing problems. In *Foundations of Genetic Algorithms* (Edited by G. J. E. Rawlins), pp. 284–300. Kaufmann (1991).
6. D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison–Wesley, Reading, Mass. (1989).
7. D. E. Goldberg, Zen and the art of genetic algorithms. *Proc. Third Int. Conf. Genetic Algorithms and their Applications*, pp. 80–85. Kaufmann (1989).
8. D. E. Goldberg and R. L. Lingle, Alleles, loci and the traveling salesman problem. *Proc. First Int. Conf. Genetic Algorithms and their Applications*, pp. 154–159. Erlbaum (1985).
9. J. J. Grefenstette (Ed.), *Proc. First Int. Conf. Genetic Algorithms and their Applications*, Erlbaum (1985).
10. J. J. Grefenstette (Ed.), *Proc. Second Int. Conf. Genetic Algorithms and their Applications*, Erlbaum (1987).
11. J. Grefenstette, R. Gopal, B. Rosmaita and D. Van Gucht, Genetic algorithms for the traveling salesman problem. *Proc. First Int. Conf. Genetic Algorithms and their Applications*, pp. 160–168. Erlbaum (1985).
12. J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Mich. (1975).
13. I. M. Oliver, D. J. Smith and J. R. C. Holland, A study of permutation crossover operators on the traveling salesman problem. *Proc. Second Int. Conf. Genetic Algorithms and their Applications*, pp. 224–230. Erlbaum (1987).
14. P. W. Poon, The genetic algorithm applied to PWR reload core design. Ph.D. thesis, Cambridge University. In preparation.
15. P. W. Poon and G. T. Parks, Optimising PWR reload core designs. *Parallel Problem Solving from Nature 2* (Edited by R. Männer and B. Manderick), pp. 371–380, North-Holland, Amsterdam (1992).
16. J. D. Schaffer (Ed.), *Proc. Third Int. Conf. Genetic Algorithms and their Applications*, Kaufmann (1989).
17. D. Smith, Bin packing with adaptive search. *Proc. First Int. Conf. Genetic Algorithms and their Applications*, pp. 202–226. Erlbaum (1985).
18. T. Starkweather, S. McDaniel, K. Mathias and D. Whitley, A comparison of genetic sequencing operators. *Proc. Fourth Int. Conf. Genetic Algorithms and their Applications* (Edited by R. K. Belew), pp. 69–76. Kaufmann (1991).
19. G. Syswerda, Schedule optimization using genetic algorithms. In *A Handbook of Genetic Algorithms* (Edited by L. Davis), pp. 332–349. Van Nostrand Reinhold, Amsterdam (1991).
20. D. Whitley, T. Starkweather and D. Fuquay, Scheduling problems and traveling salesman: the genetic edge recombination and operator. *Proc. Third Int. Conf. Genetic Algorithms and their Applications*, pp. 133–140. Kaufmann (1989).

APPENDIX A

Data Used to Define the Problems

A.1. Oliver's 30 cities

These coordinates for the 30 cities used in the problems 3 and 4 are taken from Oliver *et al.* [13].

City	Coord.	City	Coord.	City	Coord.
1	(87, 7)	11	(58, 69)	21	(4, 50)
2	(91, 38)	12	(54, 62)	22	(13, 40)
3	(83, 46)	13	(51, 67)	23	(18, 40)
4	(71, 44)	14	(37, 84)	24	(24, 42)
5	(64, 60)	15	(41, 94)	25	(25, 38)
6	(68, 58)	16	(2, 99)	26	(41, 26)
7	(83, 69)	17	(7, 64)	27	(45, 21)
8	(87, 76)	18	(22, 60)	28	(44, 35)
9	(74, 78)	19	(25, 62)	29	(58, 35)
10	(71, 71)	20	(18, 54)	30	(62, 32)

A.2. Delivery problem

The following table contains the distances between the towns and the required deliveries for problem 5.

Towns	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	8													
3	5	9												
4	9	15	7											
5	12	17	9	3										
6	14	8	11	17	8									
7	16	18	12	7	6	14								
8	17	14	12	15	15	8	11							
9	22	22	17	18	15	16	11	10						
10	12	11	7	10	10	9	8	6	11					
11	25	29	20	13	11	24	8	16	7	14				
12	19	25	15	7	5	25	5	18	16	14	9			
13	12	21	12	6	8	26	13	25	26	17	21	11		
14	21	30	19	11	10	34	14	30	26	22	18	8	8	
15	11	8	7	13	14	5	13	7	16	4	20	19	20	27
Deliveries	10	15	18	17	3	5	9	4	6	6	7	12	17	19

A.3. Factory design problem

In the table below are the journey frequencies between various work areas in the factory used for problem 6.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1																				
2	8																			
3	5	25																		
4	4	11	10																	
5	6	14	11	3																
6	7	9	12	4	4															
7	2	30	44	5	6	5														
8	2	3	0	2	2	2	0													
9	2	3	75	2	2	2	39	2												
10	0	2	14	0	0	0	15	0	2											
11	82	2	22	2	2	2	20	0	31	64										
12	42	5	8	48	1	9	24	1	20	0	16									
13	12	71	5	3	3	14	29	2	22	3	17	26								
14	38	14	7	2	6	19	30	0	20	4	32	18	1							
15	7	8	5	12	4	56	12	0	1	10	1	3	19	20						
16	7	7	9	4	8	1	17	1	3	19	3	66	2	14	2					
17	42	8	4	0	9	2	0	3	2	10	33	12	22	2	3	8				
18	2	7	0	1	1	0	64	1	2	2	9	14	4	8	10	01	1			
19	9	8	0	1	1	3	2	0	2	8	9	1	0	30	40	6	4	2		
20	6	15	3	3	3	4	3	0	1	8	2	3	66	7	5	10	10	18	26	

APPENDIX B

Best Solutions

B.1. Problem 5: delivery scheduling

The shortest delivery schedule that any of the algorithms found was 171 units in length and consisted of the following journeys:

(15, 6, 2, 4, 15) (15, 1, 3, 10, 15) (15, 8, 9, 11, 12, 7, 15) (15, 13, 14, 5, 15)

B.2. Problem 6: factory design

The best solution that any of the algorithms found for the factory design problem had a edge sum of 1463. The nodes of the triangles that form the maximally planar graph are below.

(4, 13, 7)	(20, 13, 2)	(5, 7, 2)
(7, 13, 2)	(14, 13, 12)	(18, 12, 7)
(15, 13, 20)	(14, 20, 2)	(17, 12, 11)
(10, 11, 7)	(14, 12, 1)	(14, 1, 11)
(6, 13, 15)	(19, 15, 20)	(14, 11, 9)
(14, 9, 7)	(9, 11, 3)	(9, 3, 7)
(8, 12, 17)	(1, 17, 11)	(11, 12, 18)
(11, 18, 7)	(14, 7, 5)	(14, 5, 2)
(12, 13, 4)	(16, 4, 7)	(3, 11, 10)
(3, 10, 7)	(14, 15, 19)	(14, 19, 20)
(12, 4, 16)	(12, 16, 7)	(14, 13, 6)
(14, 6, 15)	(1, 12, 8)	(1, 8, 17)