

# Advances in Ordered Greed

*Peter G. Anderson<sup>1</sup> and Daniel Ashlock*  
*Laboratory for Applied Computing, RIT, Rochester, NY*  
*and Iowa State University, Ames IA*

## Abstract

Ordered Greed is a form of genetic algorithm that uses a population of permutations whose fitnesses depend on their use as the orders in which parts of a problem are solved. For example, a permutation may specify the order of the rows in which chess-board queens are placed to try to avoid attacks by other queens, or it may specify the order in which vertices of a graph are colored to avoid adjacent vertices getting the same color. The problems mentioned are surrogates for practical, difficult, real-life problems such as scheduling.

Ordered greed requires its own form of crossover. We have developed and investigate two new crossover operators. The first uses the standard combinatorial mathematics notion of permutations' signatures, which are a list of numbers that specify a permutation, but for which the usual bit-string notions of substring-swapping crossovers apply. The second form of crossover works with permutations directly, merging two parents, as in a riffle shuffle, and extracting two children from the merged list consisting of the lists of first or second instances of each value. We compare the new methods with each other and with the well-known PMX

The application for ordered greed that we have chosen for this investigation is that of coloring Hamiltonian planar graphs (i.e., map graphs), which are well-known to be four colorable.

## 1. Ordered Greed (OG)

Greedy algorithms are very simple, heuristic approaches to finding solutions to combinatorial optimization problems. *Greedy* generally indicates that the problem is solved incrementally, choosing an optimum assignment at each stage, with no look-ahead and no backtracking. The simplest example of a greedy algorithm that is always successful is change making, expressing a given amount of money in, say American or Euro currency; namely, iterative reduce the outstanding amount using the largest possible coin. Another typical example is the *sequential algorithm* for graph coloring [6]: select an ordering of the vertices of the graph to be colored,  $\{v_0, v_1, \dots, v_{N-1}\}$ , then color each vertex in order, using the first available color

for each vertex (that is, so no edge's two end points are assigned the same color). Notice that an ordered set of colors is required.

This algorithm is very unlikely to color a graph using anywhere near the minimum number of colors (the graph's chromatic number), but it does achieve a valid coloring rapidly. However, a coloring using the minimum possible number of colors is clearly the result of this algorithm, if a suitable permutation is used. In fact there are a large number of suitable permutations. Take a coloring of a graph that is a witness to the chromatic number (an optimal coloring). Order the colors used. Now present the vertices to OG so that all vertices with a lower color are before those with a higher color. The greedy algorithm will then exactly reproduce the optimal coloring. We then see that for a given optimal coloring there are as many permutations that reproduce that coloring as the product of the factorials of the sizes of the monochrome sets of vertices. The space of permutations is huge, with a large but sparse subset of good ones. This is the type of search problem modern metaheuristics are applied to.

OG uses a genetic algorithm with a population of permutations. It assigns a fitness to each permutation according to the quality of the answer the greedy algorithm yields using that permutation. For the problem of coloring the vertices of a planar graph, a permutation's fitness could be the number of vertices successfully colored using just the first four colors.

There is a wide repertoire of problems for which OG is applicable. The first successful real-world application was to schedule an invitational soccer tournament with a large number of teams, age groups, soccer fields and time slots [5, 2].

Another problem we use for testing is the classical  $N$ -Queens problem: place  $N$  chess Queens on an  $N \times N$  board so that no two attack each other (no two Queens may occupy the same row, column, or diagonal). This is a rather easy instance of the Maximum Independent Set problem (MIS), but, unlike MIS,  $N$ -Queens is not NP-complete (contrary to what we might read on the Internet); see, for instance [7]. The OG approach to solving the  $N$ -Queens problem is to use a given  $N$ -permutation to specify the *order of the rows* into which we place the Queens. If we have an ordering of the chess board rows,

<sup>1</sup>Send correspondence to pga@cs.rit.edu.

0					5		
1		2					
2						7	
3	0						
4			1				
5				3			
6							6
7				4			

Figure 1: How OG converts 3 4 1 5 7 0 6 2 to an 8-Queens solution. The row numbers are written vertically along the left of the chess board. The Queens are labeled in the order in which they were placed on the board, each one placed as far left as possible, avoiding attacks by previously placed Queens.

$\{r_0, r_1, \dots, r_{N-1}\}$ , we place the first Queen in the far left position of row  $r_0$ , and continue, for each  $i$ , place a Queen in row  $r_i$  as far left as possible (i.e., greedily) avoiding attacking any previously placed Queens.

The OG approach to finding Queens solutions makes the solutions much denser in the search space of permutations than the obvious alternative—namely working with the space of  $N$ -permutations, each of which solves the  $N$ -rooks problem, and assigning a fitness to be the number of unattacked Queens. For example, when we solve the classical 8-Queens problem with OG, we find a perfect solution as the third individual constructed for the initial population: 3 4 1 5 7 0 6 2. See Figure 1 to see how OG converts this permutation to a solution of the 8-Queens problem. Figure 2 shows the fitness histograms for 20,000 random permutations for the 64-Queens problem using the permutation as the Queens' placement (i.e., the  $k^{\text{th}}$  element of the permutation gives the column number for the Queen in row  $k$ ) and using the permutation to drive the OG placement.

In addition to graph coloring, sports scheduling, and  $N$ -Queens, OG may be applied to whole host of other problems, such as

- Bin packing. Each object is placed, in turn, into the first bin that can still hold it.
- Traveling salesman. From a permutation of the cities, create a triangle circuit using the first three cities.

Then add each city, in turn, into the circuit as inexpensively as possible.

- Polyominoes puzzles. An  $N$ -omino is a rook-wise connected collection of  $N$  unit squares. A commercially sold pentominoes puzzle consists of 12 plastic pieces to be placed into, say, a  $6 \times 10$  rectangle. Permute the shapes in all orientations and place each one (that has not yet been placed in another orientation) as close to the top left as possible.
- Satisfiability. Given a Boolean formula expressed as a product of sums of variables or their negations, work with a permutation of the variables, assigning each one, in turn, a truth value that can satisfy the first yet unsatisfied sum.
- Scheduling faculty teaching. Work with a permutation of the faculty, where each member is represented the number of times he or she is required to teach. Successively assign each faculty to the best course assignment consistent with the assignments already made.

This list can be extended quite a bit. We do, however, recommend that the following principle be observed:

**The OG Principle:** *Prove that at least one optimal solution does, in fact, follow from the greedy algorithm with at least one permutation.*

### 1.1. Our steady-state GA

For the experiments reported in the present paper, our GA takes a steady-state form. Namely, a population is initialized with random individuals. Their fitnesses are computed. Then, we iteratively select two parents using two 2-element tournaments. The tournament winners are crossed over to create two new children, the children are subject to mutation then fitness evaluation. The new children replace the two tournament losers in the population. The process continues until an individual with a desired fitness appears or the number of permitted fitness evaluations occurs.

A mutation rate of  $p$  means that every element of a given individual is subject to re-randomization (for a list of values, such as a signature) or a random transposition (for a permutation) with probability  $p$ .

## 2. Permutation Crossovers for OG

A genetic algorithm with a population of permutations (or representatives of permutations, as we will see below) requires a crossover technique that tends to preserve and bring together the desirable properties ("schema") of parents into their children. The specific desirable property of

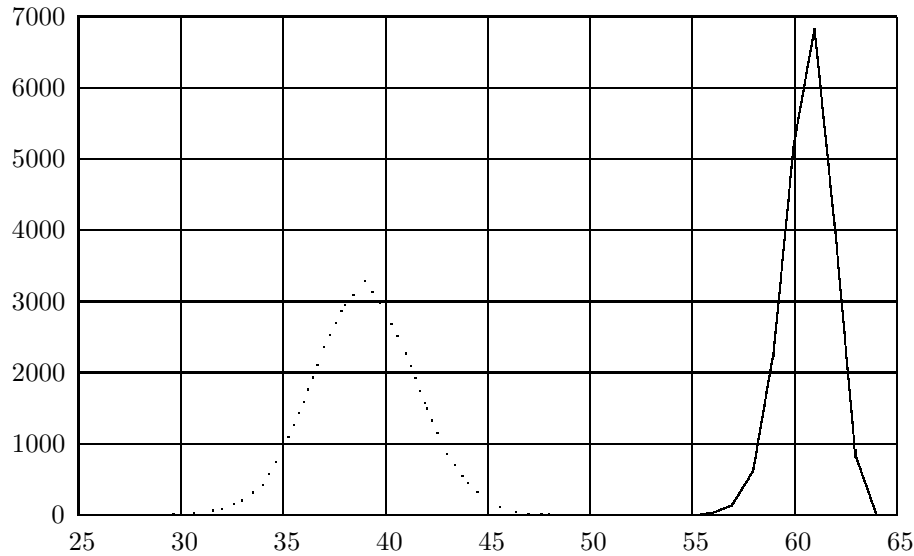


Figure 2: Histograms for the fitnesses of 20,000 permutations for the 64-Queens problem using permutation as placement (the dotted curve on the left) and using OG (the solid curve on the right).

a permutation, from the point of view of OG, is that some given items precede others. Two of the three permutation crossover techniques given by Goldberg [4], partially-matched crossover (PMX) and cycle crossover (CX), preserve much of the absolute position of elements. The third method, ordered crossover (OX) preserves some orderings as well as absolute positions of elements.

We use PMX as one of the crossover techniques in the present investigation as well as two novel techniques. All three are described below.

### 2.1. Partially matched crossover (PMX)

Our implementation of PMX is the following. Initialize two children to be identical to their two parents. Denote these permutations by  $\{a_0, a_1, \dots, a_{N-1}\}$  and  $\{b_0, b_1, \dots, b_{N-1}\}$ . Then the following is repeated  $N/4$  times: select a random position  $k$ ,  $0 \leq k < N$ , and interchange the values of  $a_k$  and  $b_k$  at all places they occur in the two children. This has the effect of modifying up to half of the positions in the two new individuals.

### 2.2. Signature representations and crossover

A permutation's signature is a list of integers  $\{s_0, s_1, \dots, s_{N-1}\}$  satisfying

$$0 \leq s_k < N - k, \text{ for all } k$$

Clearly, there are  $N!$  signatures—they are equinumerous with permutations. A signature,  $s$ , may be converted into a permutation,  $p$ , by the following code snippet:

```
for ( k = 0; k < N; k++ )
    p[k] = k;
for ( k = 0; k < N; k++ )
    interchange p[k] and p[k+s[k]]
```

The above code represents a one-to-one onto transformation of the set of signatures to the set of permutations. (A less straightforward, yet still  $\mathcal{O}(N)$  procedure, can transform permutations into signatures. This is a student exercise.)

Signatures are particularly convenient for permutation-based genetic algorithms, because:

- Signatures are very easy to generate randomly.
- Signatures are easy to mutate.
- The bit-string (or small alphabet) crossover techniques, one-point, two-point, and uniform crossover can apply directly to signatures.

### 2.3. Merging crossover (MOX)

To form permutation children from parents using MOX, first randomly merge the two parents into a  $2N$ -element list,  $L$  (this operation is similar to a riffle shuffle of cards). The first instance of each value in  $L$  gives the ordering of elements for the first child, and the second instance gives the second child. For example, let  $p_1 = \{3\ 9\ 0\ 1\ 2\ 4\ 6\ 8\ 7\ 5\}$  and  $p_2 = \{2\ 6\ 7\ 1\ 4\ 8\ 0\ 3\ 5\ 9\}$  be the two parent permutations. One merging of  $p_1$  and  $p_2$  gives  $L = \{2\ \mathbf{3}\ 6\ 7\ 1\ 4\ \mathbf{9}\ 0\ \mathbf{1}\ 2\ 3\ \mathbf{4}\ \mathbf{6}\ 5\ \mathbf{8}\ 7\ 5\ 9\}$  (the elements from  $p_1$  are shown in bold). From  $L$ , we extract the children  $c_1 =$

$\{ \mathbf{2} \mathbf{3} \mathbf{6} \mathbf{7} \mathbf{1} \mathbf{4} \mathbf{9} \mathbf{0} \mathbf{8} \mathbf{5} \}$  and  $c_2 = \{ \mathbf{0} \mathbf{1} \mathbf{2} \mathbf{3} \mathbf{4} \mathbf{6} \mathbf{8} \mathbf{7} \mathbf{5} \mathbf{9} \}$  (the  $p_1$  contribution is still shown in bold).

The intermediate list,  $L$ , is not needed, except conceptually. All we need is a one-element buffer,  $X$ , that is filled from the initial elements of the two parents, treated as queues, with the parents chosen at random. Then  $X$  is appended to the first child, if  $X$  is not already present, otherwise it is appended to the second child.

OG operates by seeking good precedence orders among permutation elements. Let “ $a \prec b$ ” denote that “ $a$  precedes  $b$ ” in a given permutation. We believe that MOX is particularly suitable for OG because of the following property: if  $a \prec b$  in both parents, then  $a \prec b$  in both children produced by MOX. This is illustrated in the MOX example above:  $1 \prec 4$  in both parents and in both children. In case  $a \prec b$  in one parent and  $b \prec a$  in the other, then both children can have  $a \prec b$ , or both can have  $b \prec a$ , or the two children can be mixed.

It is easy to come up with examples for all of the other mentioned permutation crossover operations (plus Ordered Crossover (OX) and Cycle Crossover (CX) [4]) in which the property of  $a \prec b$  in both parents is not preserved for both children.

In the following section, we describe a simple experiment that illustrates some of the power of MOX.

### 3. Evens precede odds

Following Goldberg’s simple illustration problem [4] of a genetic algorithm search for a bit string with as many 1’s as possible, we propose a permutation based genetic algorithm to search for a permutation in which all the even entries precede all the odd entries. As with Goldberg’s problem, this is not to be taken seriously, but as representative of some aspects of the heuristic technique in question, and as a place-holder for the serious problems to follow.

An experiment in which the fitness counted the number of correct parity values in the desired half of the string showed us clearly that another fitness measure was called for—one that was able to assign finer partial credit and allow desired solutions to exert a pull on the search procedure. The fitness measure we found that does this works as follows: starting at the left end of the string, with subscript positions  $k < \frac{N}{2}$  (where we want the even values to migrate), if a value has the correct parity, it contributes  $\frac{N}{2} - k$  to the fitness. Positions with subscript  $k > \frac{N}{2}$  with the desired (odd) parity contribute  $1 + k - \frac{N}{2}$  to the fitness. The fitness of a perfect individual is therefore  $\frac{N}{2}(\frac{N}{2} + 1)$ .

We tested several crossover techniques for this problem, running each choice 100 times with different random seeds (provided by the Linux `$RANDOM`), with the results shown in Table 3. For these runs we used permutations of length 100, a population size of 100, and a mutation rate of

xover	min	Q1	median	Q3	max
MOX	3,769	5,647	6,392	7,347	11,028
PMX	5,239	17,069	30,012	59,263	–
1 Pt.	15,824	31,325	44,534	70,175	–
2 Pt.	12,514	29,298	45,928	68,969	–
Unif.	8,648	23,764	38,847	61,547	–

Table 1: The number of fitness evaluations needed to solve the “evens precede odds” for five crossover techniques. 100 attempts were made for each crossover. We report the minimum first quartile, median, third quartile, and maximum number of fitness evaluations. The entry “–” indicates the maximum value is over 100,000, at which point the process was stopped.

xover	min	Q1	median	Q3	max
MOX	162	974	1,444	2,049	5,005
PMX	184	988	1,722	2,573	7,862
1 Pt.	128	766	1,144	1,627	29,706
2 Pt.	30	763	1,142	1,726	10,885
Unif.	204	1,079	1,551	2,539	25,423

Table 2: The number of fitness evaluations needed to solve the 500-Queens problem for five crossover techniques.

0.001. The permutation crossover MOX is a clear winner.

### 4. Testing with 500 Queens

We performed a similar experiment on the 500-Queens problem using the same GA parameters as those for “evens precedes odds” with results shown in Table 4. Here, MOX did well, but the signature representation did well also. We consider MOX the winner based on the maximum values.

## 5. Generating & Coloring Planar Graphs

### 5.1. Two Types of Planar Graphs

For the present study, we considered two types of Hamiltonian planar graphs: (1) triangulations of a disk with all vertices on the boundary and (2) the union of two graphs of type (1), with the vertices identified. Both types are planar, hence four-colorable: type (1) graphs are a triangulation of a disk, hence planar; type (2) graphs are triangulations of two disks with their boundaries identified, hence they are graphs on a sphere, hence planar. The type (1) graphs are also three-colorable, which is easy to establish inductively. Such a graph,  $G$ , with  $N$  vertices must contain a triangle consisting of three consecutive vertices on the boundary,  $v_1, v_2, v_3$ . Remove  $v_2$  and its two incident edges from the graph  $G$  to form  $G'$ .  $G'$  has  $N - 1$  vertices, which by an inductive hypothesis is three-colorable. Consequently  $G$

is three-colorable. (A previous study [1] applied ordered greed to the coloring of large, random trivalent graphs.)

Type (1) graphs are easily colored by an algorithm similar in spirit to the above inductive proof. First, locate any triangle and color its vertices. Then locate triangles with two of the three vertices already colored and color the third vertex. This illustrates a successful application of the sequential algorithm. However, if the sequential algorithm is given an unfortunate permutation of vertices, many more colors will be needed. We have an example of a type (1) graph with 25 vertices for which one permutation yielded a coloring with five colors.

A corollary of the proof of three-colorability for type (1) graphs is that these graphs admit only a single valid three coloring, modulo a permutation of the colors. This suggests a reason why they are so resistant to coloring by OG.

## 5.2. The Experiments

We generated 190 random graphs of each type with  $N$  vertices,  $11 \leq N \leq 200$ , and attempted to color them minimally, with three or four colors, with 21 attempts for each graph. The genetic algorithm used permutation representation for individuals with MOX crossover, a population size of 20, and a mutation rate of 0.01. We allowed a maximum of 100,000 fitness evaluations.

## 5.3. The Results

Surprisingly, type (1) graphs proved harder to color than type (2). With  $N$  (approximately) in the ranges indicated, the following results occurred:

$11 \leq N \leq 30$	all trials succeeded
$31 \leq N \leq 64$	most trials succeeded
$65 \leq N \leq 110$	fewer than half succeeded
$111 \leq N \leq 200$	all trials failed

Type (2) graphs fared much better, with every graph being four-colored at least once:

$11 \leq N \leq 50$	all trials succeeded
$59 \leq N \leq 171$	most trials succeeded
$172 \leq N \leq 200$	fewer than half succeeded

The fact that the sparser graphs are more difficult to color may be for the following reason. When a graph has more edges, more vertex colorings are forced. A sparse graph requires fewer colors, on average, and in fact a well designed greedy algorithm can correctly color a bipartite graph every time. In a bipartite graph, however, a greedy algorithm that does not color vertices adjacent to vertices that have already been colored has an excellent chance of forcing a sub-optimal coloring. For example, if we are coloring an  $N$ -path,  $(v_0, v_1, \dots, v_{N-1})$  with OG then any

order placing an even subscripted and a non-adjacent odd subscripted vertex at the beginning of the coloring forces a sub-optimal three-coloring of the path. The lower chromatic number and sparsity of the graph create more room for OG to make a suboptimal assignment of a color that does not manifest itself until later in the coloring process.

## 6. Conclusion

In this paper we have introduced the permutation crossover MOX and discussed our other permutation representation, signatures. MOX appears to be quite suitable for OG applications, although, clearly, more work needs to be done in the area of graph coloring.

Future investigation will include a method inspired by Warnsdorff's heuristic for the knight's tour, which specifies that the knight should move to a cell with the fewest continuations [3]. Specifically, OG would color vertices with the fewest legal colors (or most varied colored neighbors) at each stage, using the permutation to break ties. This will color type (1) graphs with the first attempt, and it should improve performance on type (2) graphs.

## References

- [1] Peter G. Anderson. Ordered greed, II: Graph coloring. In *Proceedings of the International NAISO Congress on Information Science Innovations, Dubai, U. A. E.* Natural and Artificial Intelligent Systems Organization, 2001.
- [2] Peter G. Anderson and William T. Gustafson. Ordered greed. In *Proceedings of the ICSC Conference on Soft Computing, Genoa, Italy (SOCO'99)*. International Computer Science Conventions, 1999.
- [3] W. W. Rouse Ball and H. S. M. Coxeter. *Mathematical Recreations & Essays*. University of Toronto Press, 1974.
- [4] David Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.
- [5] William Gustafson. Building a soccer tournament schedule using a genetic algorithm. Master's project, Rochester Institute of Technology, 1998.
- [6] Marek Kubale. *Introduction to computational complexity and algorithmic graph theory*. Gdańskie Towarzystwo Naukowe, Gdańsk, 1998.
- [7] R. Sosic and J. Gu. A polynomial time algorithm for the N-queens problem. *SIGART Bulletin*, 1(3):7-11, October 1990.