Research question:
How does one go about developing a new crossover operator with an *a priori* expectation of its merit?

Kelly D. Crawford
The University of Tulsa

1 Introduction

Design of crossover operators is an important step in constructing a useful genetic algorithm (GA). Many off-the-shelf operators exist and can be easily gleaned from the literature. Not all existing operators, however, work well for all problems. Some problems cannot be solved with existing operators and require specially tailored operators. This paper provides guidelines for the construction of these operators.

In general, other than results from existing literature, there is no *a priori* way to know if a crossover operator is going to work well for a particular problem. However, a good crossover operator will typically have a set of characteristics that can be determined *a priori*. This paper describes one such particular set of characteristics that a GA researcher can use to design new crossover operators.

We begin with a brief overview of the GA and the Schema Theorem, followed by an extensive survey of existing crossover operators from the GA literature. Next, we will see how crossover operators have been classified to date and analyze some of the problems associated with crossover. Lastly, a set of guidelines will be offered together with some supporting theory.

2 Genetic Algorithms

The Genetic Algorithm (GA) is a search and optimization technique loosely based on principles of population genetics (e.g. reproduction, survival of the fittest, etc) [36, 40]. The basic data structure is the chromosome, or string, to which a solution to a problem is logically mapped. A chromosome consists of an array of genes, each of which can have a value called an allele. A gene in the chromosome is said to be at a locus, or index within the string.

As an example of a chromosome, consider the problem of satisfiability (SAT). Given a boolean expression of N variables, what combination of variables will make the expression true? This can be represented as a string of N bits. A 1 bit is true and a 0 bit is false. In this example, this bit string is the chromosome.

There are N genes in the chromosome, each of which can have the allele values of 0 or 1.

Representation concerns how the solution is represented by the chromosome. In the SAT example, the representation is bit-based. Other representations include order-based and real-valued. A good order-based example is the Traveling Salesman Problem (TSP). Given a list of cities numbered from 1 to N, and distances between each city (assuming a fully connected graph), what is the shortest tour through all of the cities? This can be represented by a permutation of N integers. A real-valued example could be a function of N variables. This can be represented by a list of N real numbers. The allele values are the set of real numbers, or some range of numbers.

Given a representation and a mapping from a solution to a chromosome, we now need some way to evaluate a solution. This evaluation function must decode the chromosome into the solution space, and evaluate the solution. Typically, this involves reducing the solution to some scalar value indicating the goodness of that solution. For example, for the TSP, the evaluation function is the sum of the length of the tour represented by the chromosome.

Lastly, we need some method for reproduction. There are two general types of mating schemes in GAs. They are crossover and mutation. In crossover, two parent chromosomes are recombined to create one or two child chromosomes. In other words, one or two new solutions to a problem. In mutation, a single chromosome is randomly altered. For example, a bit can be flipped in a bit-based problem, two alleles from different genes can be swapped in an order-based problem, etc. There are many crossover and mutation strategies in the literature.

Given all of these elements, the following algorithm describes a GA.

```
Randomly generate a population of p chromosomes
Evaluate all chromosomes in the population
Loop until stopping criteria is met
   Select two parents (with a tendency to select more fit chromosomes)
   Apply crossover to the parents to create two children
   Mutate children according to some (usually low) probability
   Evaluate children
   if (generational GA)
      Insert children into new population
   else if (steady-state GA)
      Insert children back into population if they are better than
         the least fit member of the population (displacing the one
         or two least fit members)
End Loop
```

Many variations exist. Selection bias (how likely to select more fit chromosomes), crossover rate (how likely to apply crossover), mutation rate (how likely

to mutate a gene), selection and insertion schemes, and the list goes on and on, can be changed. There are so many parameters, one paper actually proposed using a GA to set the GA's parameters [37].

The selection step, which is done according to the fitness values (from the evaluation function), is what actually drives the search. Crossover recombines these good solutions and tends to produce even better ones over time. Mutation is a quick way to explore a totally different area of the search space. The mechanics of these steps are described in the next section.

3 The Schema Theorem

The inner workings of GAs are described by the Schema Theorem [39, 40]. In short, it describes how similarities between strings are passed from generation to generation.

Consider chromosomes (strings) consisting of l bits, so all gene values are taken from the set of alleles {0,1}. By introducing a *don't care* symbol, *, which represents either 0 or 1, we can describe similarities between strings. These similarity templates, called schemata (plural of schema), can be thought of as pattern matching devices.

For example, consider the schema $H = 01*1*$. This schema matches the following set of strings: {01010,01011,01110,01111}. The schema $H = 01110$ only matches itself. The schema $H = *****$ matches any string of the 2^5 strings of length 5. For a string of length l, there are exactly 3^l schemata and 2^l bit strings that can be produced.

Two properties of schemata are order and defining length. The order of a schema, $o(H)$, is defined as the number of fixed positions in H. In the case of bits, this is the number of 1's and 0's. In fact, the number of *don't care symbols* is given by $l - o(H)$. Given this definition, note that a schemata matches exactly $2^{l-o(H)}$ strings.

Defining length, $\delta(H)$, is the distance (counting bit positions) between the first and last fixed positions in H. In the example $H = 01*1*$, the defining length is $\delta = 3$, since the first fixed position is the 0 (position 1) and the last fixed position is the 1 (position 4).

Given these definitions, what is the effect of reproduction on the expected number of schemata in the population? Suppose at time t there are $m(H,t)$ examples of a particular schema H in the population. A string will be selected for reproduction according to probability $p_i = \frac{f_i}{\sum_j f_j}$, where f_i is the fitness of the i'th string in the population, and $\sum_j f_j$ is the total fitness of all strings in the population. We also define $f(H)$ to be the average fitness of all strings matching the schema H.

We add a probability of crossover, p_c, and a probability of mutation, p_m, to come up with a general formula for schemata survival:

$$m(H,t+1) \geq m(H,t)\frac{f(H)}{\bar{f}}\left[1 - p_c\frac{\delta(H)}{l-1} - o(H)p_m\right]$$

This formula, the Schema Theorem, says that schemata are chosen for reproduction according to their relative fitness in the population. The part between the brackets, using the order and defining length of H, tells how likely it is that H will not be disrupted. Consider again $H = 01*1*$. If the results of crossover changes any of the 1 bits to 0, or vice versa, the schema does not survive (i.e. it is disrupted). The power of this theorem is that it tells us that the stronger schemata will survive while the weaker ones will die out.

The Building Block Hypothesis is a logical conclusion based on the Schema Theorem. It states that short, low-order, highly fit schemata, or building blocks, are sampled, recombined and resampled to form strings of potentially higher fitness. While both the Schema Theorem and the Building Block Hypothesis have been hotly debated, both are accepted to be generally true by the majority of the GA community.

This brings us to our first crossover design guidelines.

Guideline 1 *Crossover should adhere to the schema theorem.*

Guideline 2 *Crossover should adhere to the building block hypothesis.*

4 Recombination Survey

There are a multitude of crossover operators described in the literature, many of which are outlined in the next five sections. Each of these operators have different strengths and weaknesses. Many have even been hotly debated over the years (e.g. uniform crossover). One thing is clear, however. Little work remains to be done in the design of new operators. For bit-based and order-based problems in particular, new (useful) operators have not appeared for several years, indicating some level of maturity. Real-valued operators have not changed much either. However, subset operators and operators for hybrid methods are another matter. The subset operators are clearly in their infancy. The hybrid operators are usually different for each new problem, thus new or modified operators appear with almost every new paper.

Many research hours have gone into each of the existing operators in the form of design and testing. Thus, the following guidelines can save much wasted time and effort.

Guideline 3 *Use existing operators where applicable, especially with bit-based, order-based or real-valued problems.*

Guideline 4 *Modify existing operators when possible.*

4.1 Bit string recombination

Unless otherwise indicated, each of the crossover operators described below will deal with two parents, $p1$ and $p2$, both of length L. We will be generating one or two children, $c1$ and $c2$ (or just c). In general, bit string crossover can be described by a crossover mask [63]. A crossover mask, m, is a bit string of length L. For any string position i, a 0 bit in the crossover mask means child 1 gets the $i'th$ bit from parent 1, and child 2 gets the bit from parent 2. A 1 bit in the crossover mask inverts this order, with child 1 receiving the bit from parent 2, and child 2 receiving the bit from parent 1. The following diagram illustrates:

```
parent 1:        0 1 0 1 1 0 1 1 0 0 0 1
parent 2:        1 0 1 0 1 0 0 0 1 0 1 0
                 ----------------------
crossover mask:  1 0 0 0 1 0 1 1 1 0 1 0
                 ----------------------
child 1:         0 0 1 0 1 0 1 1 0 0 0 0
child 2:         1 1 0 1 1 0 0 0 1 0 1 1
```

For convenience, we will describe all bit string crossover operators in terms of a crossover mask.

4.1.1 1-point Crossover

[40]

Randomly generate a value, i, between 1 and $L-1$. Construct m with the first i bits being 0, and the remaining $L-i$ bits being 1. This results in masks of the form 0..01..1.

For example, for $L = 10$ and $i = 7$, the crossover mask would be $m = 0000000111$.

4.1.2 2-point Crossover

[27]

Randomly generate two different values, i and j, between 1 and $L-1$. Assuming $i < j$, construct m with the first i bits being 0, the next $j-i+1$ bits being 1, and the remaining $L-j$ bits being 0. This results in masks of the form 0..01..10..0.

For example, for $L = 10$, $i = 1$ and $j = 6$ the crossover mask would be $m = 0111110000$.

4.1.3 Uniform Crossover

[1, 63]

Randomly generate a crossover mask. This is essentially L-point crossover. Eschelman, et al [31] described a variant of uniform crossover where the crossover

mask is generated with a probability of x for a 0 and y for a 1. In standard uniform crossover, $x = y = .5$. In this probabilistic uniform crossover, they describe a test with $x = .25$ and $y = .75$. This produced no striking results.

4.1.4 Multi-point Crossover

[27]

In multi-point crossover, you first select the number of crossover points you want to generate ($1 \leq p \leq L$). Note that $p = 1$ is exactly 1-point crossover, $p = 2$ is exactly 2-point crossover, and $p = L$ is exactly uniform crossover. After selecting the sorted array of crossover points, P, the crossover mask is generated by alternating 0 and 1 according to the values in P.

For example, if $L = 10$, $p = 4$ and $P = 2, 3, 5, 8$, the following crossover mask would result: $m = 0010011100$, which is 2 $0's$, $3 - 2$ $1's$, $5 - 3$ $0's$, $8 - 5$ $1's$, and $L - 8$ $0's$.

4.1.5 Segmented Crossover

[31]

Segmented crossover is a variant of multi-point crossover that allows the number of crossover points to vary. Instead of choosing in advance a fixed number of unique crossover points (e.g. 2-point crossover), a segment switch rate is specified. The switch rate specifies the probability that a segment will end at any point in the string. For example, if the switch rate is 0.25, then once a segment is begun, there is a 0.25 probability of ending it at each bit and beginning a new segment by making a cut. The expected number of segments is thus L/s, where s is the switch rate.

Another variant on this is to use two switch rates, one for ending segments that are crossed over and one for ending segments that are not crossed over. Switch rates that total 1.0 create an operator equivalent to the modified uniform crossover with probabilities. Switch rates of .25 and .75, for example, would be equivalent to uniform crossover with an exchange rate of .25.

4.1.6 Shuffle Crossover

[31]

Shuffle crossover is not a crossover by itself, but is combined with some other (compatible) bit string crossover operator. The order of the bits in the string are shuffled, a crossover operator is performed on the shuffled bits (1-point, uniform, etc), and then the bits are unshuffled to generate the children.

4.1.7 Punctuated Crossover

[61]

Punctuated crossover is not a crossover operator itself, but rather a set of rules that govern where in the chromosome crossover is allowed to occur. A bit string of length L is supplemented with another bit string of length L containing the crossover punctuation. A 1 bit means that crossover may occur at that point in the string.

For example, under 1-point crossover, the following example illustrates the use of punctuation.

```
Parent 1:   a a a a a a a!b b b b b b b
Parent 2:   c c c c!d d d d d d!e e e e

Child 1:    a a a a d d d b b b e e e e
Child 2:    c c c c!a a a!d d d!b b b b
```

In these examples, the punctuation is represented by the symbol !, but can actually be represented as stated above with a secondary bit string as follows.

```
Parent 1:   a a a a a a a!b b b b b b b 0 0 0 0 0 0 1 0 0 0 0 0 0 0
Parent 2:   c c c c!d d d d d d!e e e e 0 0 0 1 0 0 0 0 0 1 0 0 0 0
```

4.1.8 Bonded Crossover

[14]

4.1.9 Gray Coding

Gray coding is a method of mapping contiguous values to bit strings with a Hamming distance of 1. Thus, an increase of one step in the parameter value corresponds to a change of a single bit in the code. Gray coding helps reduce the epistatic (nonlinear) relationships between subsequent bit positions.

In Gray coding, parents are decoded, crossed over, and then the resulting children are recoded (or vice versa). The conversion formula from binary coding to Gray coding is

$$\gamma_k = \begin{cases} \beta_1 & if k = 1 \\ \beta_{k-1} \oplus \beta_k & if k > 1 \end{cases}$$

where γ_k is the k^{th} Gray code bit, β_k is the k^{th} binary code bit, bits are numbered from 1 to n starting on the left, and $\oplus$ denotes addition mod 2.

The conversion from Gray coding to binary coding is

$$\beta_k = \sum_{i=1}^{k} \gamma_i$$

where the crossover is done mod 2. For example, the binary code 1101011 corresponds to the Gray code of 1011110 [68].

4.1.10 Modified 2-point Crossover

[8]

Modified 2-point crossover works on variable length strings. Cross points are determined according to corresponding markers in the two strings. The genetic material between the markers is allowed to vary.

4.1.11 H1X

[30]

Part of Eshelman's CHC strategy. H1X is 1-point crossover, except that the crossover point is always the mid-point of the string. This operator was not recommended for use, and was only used to show how HUX was developed (see HUX).

4.1.12 H2X

[30]

Part of Eshelman's CHC strategy. H2X is 2-point crossover, except that the first cross point is chosen at random, and the second is always chosen so as to be $L/2$ loci from the first point. This operator was not recommended for use, and was only used to show how HUX was developed (see HUX).

4.1.13 HUX

[30]

Part of Eshelman's CHC strategy. HUX is similar to uniform crossover. It cross over exactly half of the non-matching alleles (what Booker calls the reduced surrogate [8]), where the bits to be exchanged are chosen at random without replacement. HUX guarantees that the children are always the maximum Hamming distance from their two parents. It is disruptive, but it maximizes the chance of two good schemata, one from each parent, being combined in a child since half the material from each parent is chosen. Furthermore, all schemata of the same order have an equal chance of being disrupted or preserved.

4.1.14 Masked Crossover

[48]

In masked crossover, each parent has an associated mask that biases crossover. Once the masks are in place, the algorithm for crossing is simple:

```
copy Parent 1 to Child 1
copy Parent 2 to Child 2
for i from 1 to string-length
   if Mask2(i) = 1 and Mask1(i) = 0
```

```
      copy the i'th bit from Parent 2 to Child 1
   if Mask1(i) = 1 and Mask2(i) = 0
      copy the i'th bit from Parent 1 to Child 2
```

The children need masks in case they are ever chosen as parents in the future. In most cases, each child gets its mask from the dominant parent. In two cases, however, the rules are more complex.

Let P1 and P2 be the two parents, PM1 and PM2 their respective masks. Similarly, C1 and C2 are the two children, CM1 and CM2 their respective masks.

```
When both children are more fit than the best parent:
   a) CM1 and CM2: OR the masks of PM1 and PM2.  If there are any 0's
      left in CM1, toss a coin to decide their value
   b) PM1 and PM2: No changes except those produced by mutation

When both children are less (or equally) fit than worst parent:
   a) CM1: This mask should reflect the undesirability of the current
      search direction.  Contributions from PM2 were detrimental,
      therefore CM1 should search in the area of P2's contribution which
      is specified by the loci where PM1(i) is 0 and PM2(i) is 1.  Set
      these loci in CM1(i) to 0.  If P2 > P1 in fitness, toss a coin to
      set the bits of CM1 at those locations where both PM1(i) and
      PM2(i), are 1.
   b) CM2: A similar rule applies to CM2.
   c) PM1: P2's contribution to C1 led to a bad child.  The C1 positions
      copied from P2 need to be explored in P1.  These loci are those
      for which PM1(i) was 0 and PM2(i) was 1.  Therefore set these loci
      in PM1(i) by tossing a coin.  In addition, PM1 specified loci that
      were detrimental to C2.  Therefore when PM1(i) is 1 and PM2(i) is
      0, set these locations by tossing a coin.
   d) PM2: A similar rule applies to PM2.
```

4.1.15 Two-Dimensional Crossover

[11]

Two-dimensional crossover involves manipulation of matrices rather than vectors. Given two parent matrices (of bits, or whatever), some number of cross points are chosen. For example, one could choose two rows and two columns as crossover points. This would divide the matrices into nine sections, which could be alternated between parent 1 and 2 to the child, or could be controlled by a crossover mask of size nine, etc.

4.1.16 Bit-Based Simulated Crossover (BSC)

[65]

Consider a list of the values of a single bit position in a population (a bit column). It will contain some ratio of ones and zeroes. The owners of these bits, members of the population, have some probability of being selected, and these probabilities can be used to create a weighted average for the ones and zeroes in each bit column. This provides a probability for each bit as to whether it should be a zero or a one. These probabilities can be used to generate members that, on average, will have the same ratio of ones and zeroes for each bit position that crossover would have generated. This is bit-based simulated crossover.

4.1.17 Simplex Crossover

[6]

Given 3 bit strings, I_1, I_2 and I_3, and $I_1 > I_2 > I_3$, a new candidate I_4 will be generated as follows.

If the n-th bit of I_1 is equal to the n-th bit of I_2, then it will be the n-th bit of I_4. If the n-th bit of I_1 is different from the n-th bit of I_2 then it will be the n-th bit of I_3 with probability $1 - p$ and the contrary bit with probability p.

4.2 Order-based recombination

Many problems cannot be naturally represented with a string of bits. The traveling salesman problem, for example, requires solutions consisting of lists of cities (or numbered nodes on a graph). Such problems can be naturally represented with a permutation of n numbers.

Crossover on permutations, however, is not as simple as with bit strings. Crossing two order-based chromosomes using any of the standard bit-based operators will usually result in combinations of n numbers rather than the n unique numbers of a permutation. Usually these strings result in an invalid solution (as in the traveling salesman problem). The operators described in this section describe ways to get around these problems.

4.2.1 Inversion

[40]

The simplest of the order-based operators, inversion simply reverses a random-length subtour in a permutation. Two unique cross points are chosen at random, and the elements between those two points are inverted. An example follows.

```
Parent:  1 2 3 | 4 5 6 | 7 8 9

Child:   1 2 3 | 6 5 4 | 7 8 9
```

4.2.2 2-opt

[47]

The 2-opt operator selects two unique points in the chromosome and swaps the values at those positions. If the swap gives a worse fitness value, it is undone. An example follows.

```
Parent:  1 2 3 4 5 6 7 8 9
           ^       ^
Child:   1 2 7 4 5 6 3 8 9
```

This has also been referred to as asexual crossover (because of the single parent), but it is more properly called a mutation.

4.2.3 Order Crossover (OX)

[24, 55]

Choose two unique cross points at random. The section of the first parent between the first and second cut points is copied whole to the offspring. The remaining places are filled using elements not occurring in this crossover section. These places are determined by starting with parent 2 just past the second cross point, and working sequentially through the string (wrapping around when the end of the string is reached).

```
Parent 1:  h k c e f d | b l a | i g j
Parent 2:  a b c d e f | g h i | j k l

Child:     - - - - - -   b l a   - - -   Get material between cross
                                         points from parent 1.
Child:     - - - - - -   b l a   j k -   Start past 2nd cross point in
                                         parent 2.  Get values until a
                                         conflict occurs $l$.  Wrap
                                         to beginning of parent 2.  Two
                                         more conflicts occur $a$ $b$.
Child:     d e f g h i   b l a   j k c   Starting with $c$, copy
                                         rest of parent 2 into child.
```

4.2.4 Order Crossover #2 (OX2)

[64]

This operator differs from the other order crossover (see Order Crossover) in that several key positions are chosen randomly and the order in which these elements appear in one parent is imposed on the other parent to produce two offspring.

In the example shown below, positions 3, 4, 7 and 9 have been selected as the key positions. The ordering of the elements in these positions from Parent 2 will be imposed on Parent 1. The elements (in order) from Parent 2 are a, j, i and b. In Parent 1 these same elements are found in positions 1, 2, 9 and

10. In Parent 1, P1(1) = a, P1(2) = b, P1(9) = i and P1(10) = j, where P1 is Parent 1 and the position is used as an index. In the offspring the elements in these positions (i.e., 1, 2, 9 and 10) are reordered to match the order of the same elements found in Parent 2 (i.e., a, j, i and b). Therefore Off(i) = a, Off(2) = j, Off(9) = i and Off(10) = b, where Off is the offspring under construction. All other elements in the offspring are copied directly from Parent 1.

```
Parent 1:    a b c d e f g h i j
Parent 2:    c f a j h d i g b e
Cross Pts:      * *     *   *
Offspring:   a j c d e f g h i b
```

4.2.5 Position Based Crossover

[64]

In this operator, several random locations in the sequence are selected along with one parent; the elements in those positions are inherited from that parent. The remaining elements are inherited in the order in which they appear in the alternate parent, skipping over all elements which have already been included in the offspring. Thus, the operator appears to be similar to Davis's Order Crossover except that the elements copied from the selected parent come from random locations in the sequence and not from adjacent locations.

In the example below, the elements b, c, f and i are inherited from Parent 1 in positions 2, 3, 6 and 9 respectively. The remaining elements are inherited from Parent 2 as follows (where Off means offspring): Off(1) = P2(3) since P2(1) and P2(2) have already been included in the offspring. Then going in order, Off(4) = P2(4), Off(5) = P2(5), Off(7) = P2(6), Off(8) = P2(8) and Off(10) = P2(10).

```
Parent 1:    a b c d e f g h i j
Cross Pts:     * *     *     *
Parent 2:    c f a j h d i g b e
Offspring:   a b c j h f d g i e
```

4.2.6 Modified Crossover

[23]

Modified crossover (which predates Order Crossover) is exactly the same as order crossover, except that the first cross point is always at the beginning of the string (see Order Crossover).

4.2.7 Partially Mapped Crossover (PMX)

[35]

In PMX, two unique cross points are selected. The elements between these two points define a matching section that is used to effect a cross through position-by-position exchange operations. Consider the following.

```
Parent 1:  9 8 4 | 5 6 7 | 1 3 2 0
Parent 2:  8 7 1 | 2 3 0 | 9 5 4 6
```

First, the sections between the crosspoints are exchanged. This is mapping parent 1 to parent 2.

```
Parent 1:  9 8 4 | 2 3 0 | 1 3 2 0
Parent 2:  8 7 1 | 5 6 7 | 9 5 4 6
```

We must map parent 2 back to parent 1 using the same pairs we just exchanged (5 and 2, 6 and 3, 7 and 0).

```
Parent 1:  9 8 4 | 2 3 0 | 1 6 5 7
Parent 2:  8 0 1 | 5 6 7 | 9 2 4 3
```

The idea is simply that for every exchange of element A from parent 1 and element B from parent 2, you must also exchange A from parent 2 with B from parent 1. This will preserve the permutation in both children.

4.2.8 Cycle Crossover (CX)

[55]

With cycle crossover, we wish to satisfy the following conditions: 1) Every position of the offspring must retain a value found in the corresponding position of a parent. 2) The offspring must be a permutation.

Consider these parents:

```
Parent 1:  h k c e f d b l a i g j
Parent 2:  a b c d e f g h i j k l
```

Consider position 1. Condition 1 says the offspring must contain either h or a. Suppose we choose h. Using condition 2, a cannot be chosen from parent 2 since that position is now occupied by h. So a must be chosen from parent 1.

Since a in parent 1 is above i in parent 2, we must also choose i from parent 1. Continuing the argument, having chosen h from parent 1 we must also choose a, i, j, and l.

The positions of these elements are said to form a cycle (thus, the name cycle crossover). Choosing any of the positions from parent 1 or parent 2 forces the choice of the rest if the above conditions are to hold.

The cycles are usually labeled by a number or are called unary and designated by the letter U:

```
Parent 1:     h k c e f d b l a i g j
Parent 2:     a b c d e f g h i j k l
Cycle Label:  1 2 U 3 3 3 2 1 1 1 2 1
```

In a standard crossover operator a random crossover point or cut section is chosen. In the cycle crossover a random parent is chosen for each cycle.

Choosing cycle 1 from parent 1 and the rest from parent 2 in the above example produces the following offspring.

```
Offspring:      h b c d e f g l a i k j
Parent number:  1 2 2 2 2 2 2 1 1 1 2 1
```

The absolute positions of on average half the elements of both the first and second parents are preserved.

4.2.9 Maximal Preservative Crossover 1 (MPX)

[49]

Consider the following parents.

```
Parent 1:  9 8 4 5 6 7 1 3 2 0
                 ^ ^ ^
Parent 2:  8 7 1 2 3 0 9 5 4 6
```

Parent 1 is called the donator and parent 2 is called the receiver. Before the crossover can take place, a swap makes the begin and end of the donator's matching section (marked with $\hat{}$) lay together in the receiver. Swapping can either bring the 5 nearby the 7 or the 7 nearby the 5.

```
Child 1:  5 7 1 2 3 0 9 8 6 4
            ^ ^ ^ ^ ^ ^
Child 2:  6 4 1 2 3 0 9 5 7 8
              ^ ^ ^ ^ ^ ^
```

Now the matching section from the donator may be inserted, between the begin and the end cities. The other cities in the matching section (in the example only city 6) will leave holes in the receiver which are filled, like Order Crossover, by a sliding motion.

```
Child 1:  5 6 7 1 2 3 0 9 8 4
Child 2:  4 1 2 3 0 9 5 6 7 8
```

The gain is that most edges are preserved, or if this is not possible, at least the order of the receiving string is preserved.

4.2.10 Maximal Preservative Crossover 2 (MPX)

[53]

MPX inserts part of chromosome A into the corresponding location at chromosome B, so that the resulting chromosome is most similar to A and B. A genetic repair operator then creates a valid tour. The pseudocode follows.

```
PROC crossover (receiver, donor, offspring)
1) Choose position 0 <= i < nodes and length b(low) <= k <= b(up)
   randomly
2) Extract the string of edges from position i to position
      j = (i+k) MOD nodes from the mate (donor).  This is the
      crossover string.
3) Copy the crossover string to the offspring.
4) Add successively further edges until the offspring represents a
   valid tour.  This is done in the following way:
      IF an edge from the receiver parent starting at the last city
         in the offspring is possible (does not violate a valid tour)
      THEN add this edge from the receiver
      ELSE IF an edge from the donor starting at the last city in
              the offspring is possible
           THEN add this edge from the donor
           ELSE add that city from the receiver which comes next in
                the string, this adds a new edge, which we will mark
                as an implicit mutation
```

4.2.11 2-quick

[49]

The 2-quick operator is simply a 2-opt without the check for increased fitness.

4.2.12 2-repair

[49]

The 2-repair operator uses the same swapping of elements used by 2-opt. It starts from the hypothesis that the ancestors are well adapted to their environment. Offspring should therefore not be too much different from their parents. So changes in the genotype at positions which carry new information, introduced by crossover and mutation, are checked for consistency. Without computing the phenotype, these new edges are checked and if it is possible to exchange them by better edges this step is performed. These better edges are again marked as new. This is done until all edges are checked.

4.2.13 Uniform Order Crossover (UOX)

[26]

This is analogous to bit string uniform crossover. 1) Generate a crossover mask (bits) the same length as the parent strings. 2) Fill in positions of child 1 by copying them from parent 1 wherever the crossover mask contains a 1. 3) Make a list of the elements from parent 1 associated with a 0 in the crossover mask. 4) Permute these elements so they appear in the same order they appear

in on Parent 2. 5) Fill these permuted elements in the gaps on Child 1 in the order generated by step 4. An example follows.

```
Parent 1:          1 2 3 4 5 6 7 8
Parent 2:          8 6 4 2 7 5 3 1
Crossover Mask:    0 1 1 0 1 1 0 0

                    After step 2
Child 1:           - 2 3 - 5 6 - -
Child 2:           8 - - 2 - - 3 1

                    After step 5
Child 1:           8 2 3 4 5 6 7 1
Child 2:           8 4 5 2 6 7 3 1
```

4.2.14 Relative Order Crossover (ROX)

[45]

ROX picks a certain number of alleles which define an order machine section. The relative ordering of this section is preserved in a child. The elements outside this section are copied from the second parent preserving the order at most.

To implement, begin by initializing a preserve list, P, by picking k genes from the first parent at random but maintaining their order. Also create a null waiting list, $W = \Theta$. Thereafter, scan the second parent from left end, one gene at a time, and execute one of the following steps until a full chromosome is created.

- If the allele is not a member of P, it is copied in the child.
- If the allele is a member of P but not the first member, it is appended in W.
- If the allele is the first member of P, it is copied in the child and is deleted from P. If the next entry in P is a member of W, it is also copied in the child and the entry is deleted from P. This process continues until the first entry in P is no more a member of W.

For example, two parents with masked alleles and two created children are shown below.

```
Parent 1:  1 2 3 4 5 6 7 8
             - -     -   -
Parent 2:  5 6 2 7 3 1 8 4
             - -     -   -

Child 1:   5 2 7 3 6 1 8 4
Child 2:   3 5 6 2 1 4 7 8
```

Initially, the elements of the preserve list are $P = 2, 3, 6, 8$. Since the first gene in the second parent has an allele, 5, which is not a member of P, it is copied in the first child. The allele for the second gene in the second parent is 6 and is a member of P but is not the first member. Thus, this allele is copied in W and nothing is copied in the child. At this point, there is only one entry in $W = [6]$. The third gene in the second parent has an allele, 2, which is the first member of P. Thus, allele 2 is copied in the child and is deleted from P. The alleles in P are $[3, 6, 8]$. The next entry in P is the allele 3 and is not a member of W. The fourth entry in the second parent is 7 and is not a member of P. Thus, allele 7 is copied in the child. The rest of the string is created by following the procedures described above. The second child is created using the same principle, but making the preserve list with the marked alleles from the second parent.

4.2.15 Scramble Sublist Mutation

[26]

Select a sublist of the items on the parent chromosome and permute them in the child. This is essentially selecting two unique points in the string and randomly scrambling the string elements between those points. For example, given a string of length 9, and two points 3 and 8, the following shows how scramble sublist mutation might work.

```
Parent 1 2 3 | 4 5 6 7 8 | 9
Child  1 2 3 | 6 4 8 7 5 | 9
```

4.2.16 Alternating Edges

[38]

Alternating edges uses the adjacency representation for the traveling salesman problem. A tour is described by a list of cities. There is an edge in the tour from city i to city j iff the allele in position i is j. For example, the path tour (13542) corresponds to the adjacency tour (31524).

Using the alternating edges operator, an offspring is constructed from two parent tours as follows: First choose an edge at random from one parent. Then extend the partial tour by choosing the appropriate edge from the other parent. Continue extending the tour by choosing edges from alternating parents. If the parent's edge would introduce a cycle into a partial tour, then extend the partial tour by a random edge which does not introduce a cycle. Continue until a complete tour is constructed. The following example illustrates.

```
Parent 1:  2 3 4 5 6 1
Parent 2:  2 5 1 6 4 3

Child:     2 - - - - -   Select edge (1 2) from parent 1.
```

```
Child:     2 5 - - - -   Select edge (2 5) from parent 2.
Child:     2 5 4 - - -   Select edge (3 4) from parent 1.
Child:     2 5 4 6 - -   Select edge (4 6) from parent 2.
Child:     2 5 4 6 1 -   Edge (5 6) from parent 1 introduces a cycle.
                         Generate random edge (5 1).
Child:     2 5 4 6 1 3   Select edge (6 3) from parent 2.
```

Here, the only random edge introduced into the offspring is (51). All other edges were inherited by alternately choosing edges from parents, starting with the edge (12) from parent 2.

4.2.17 Subtour Chunking Operator

[38]

Subtour chunking uses the adjacency representation for the traveling salesman problem (see Alternating Edges).

Using the subtour chunking operator, an offspring is constructed from two parent tours as follows: First choose a subtour of random length from one parent. Then extend the partial tour by choosing a subtour of random length from the other parent. Continue extending the tour by choosing subtours from alternating parents. During the selection of a subtour from a parent, if the parent's edge would introduce a cycle into a partial tour, then extend the partial tour by a random edge which does not introduce a cycle. Continue until a complete tour is constructed. The following example illustrates.

```
Parent 1:  2 3 4 5 6 1
Parent 2:  2 5 1 6 4 3

Child:     2 3 4 - - -   Select subtour (2 3 4) from parent 1.
Child:     2 3 4 6 1 -   Select subtour (6 4) from parent 2.  Edge
                         (5 4) introduces a cycle.  Randomly generate
                         edge (5 1).
Child:     2 3 4 6 1 5   Select subtour (1) from parent 1 which creates
                         a cycle.  Randomly generate edge (6 5).
```

4.2.18 Heuristic Crossover

[38]

Heuristic crossover uses the adjacency representation for the traveling salesman problem (see Alternating Edges).

Heuristic crossover attempts to perform a secondary apportionment of credit at the allele level. This operator constructs an offspring from two parent tours as follows: Pick a random city as the starting point for the child's tour. Compare the two edges leaving the starting city in the parents and choose the shorter edge. Continue to extend the partial tour by choosing the shorter of the two

edges in the parents which extend the tour. If the shorter parental edge would introduce a cycle into the partial tour, then extend the tour by a random edge. Continue until a complete tour is generated.

4.2.19 Greedy Crossover

[46]

A greedy algorithm always selects the best option available and never backtracks. Grefenstette introduced a greedy crossover strategy for the traveling salesman problem which he called Heuristic crossover (see Heuristic crossover). Liepins, et al, modified this operator to create what they called a greedy crossover. They implemented three different versions of the operator tailored specifically for the Set Covering Problem (SCP), Job Shop Scheduling (JSS) and the Traveling Salesman Problem (TSP). In each case, note how the problem-dependent definition of greedy affects the construction of the operator.

SCP implementation:

For every pair of parent genes, 1) Initialize the set S to be empty and the matrix A to be the original set covering matrix with columns c_i and associated costs w_i. 2) For the unused columns and uncovered rows calculate the cost-ratio

$$\frac{cost}{numberofrowscovered} = \frac{w_i}{numberofrowscovered}$$

. 3) Append to S the column (say, column c) with the least cost that is included in one of the parents. 4) Strike column c and the rows covered by c from A and let this new matrix be A'. 5) If S is a cover or of no other columns are represented in the parents, stop. Otherwise, set A to A' and go to step 2.

JSS implementation:

1) For each parent pair, start at the first job ($i = 1$). 2) Compare the two jobs at the $i'th$ position of the two parents and place the better (according to the heuristic being used) of the two in the child's $i'th$ position. If one of the jobs has already been placed in the child, then automatically pick the other. If both of the jobs have already been placed, then pick a job randomly from the yet unplaced jobs. 3) Repeat step 2, incrementing i, until all positions in the child string are defined.

TSP implementation:

1) For a pair of parents, start at the first position (always the same city). Choose the shortest edge leading from the current city (that is represented in the parents). If this edge leads to a cycle, choose the other edge. If this leads to a cycle, choose a random city that continues the tour. 3) If the tour is complete, stop; else go to 2.

4.2.20 Subtour Swap Operator

[12]

The subtour-swap operator works by alternately selecting segments from the two parent solutions and incorporating those into the children solutions. In the example below, the child solution is constructed as follows:

1) Select a segment from parent 1. In this case it is (326). Place this segment at the front of the child solution. 2) Select a new segment from parent 2. This time it is (704). Concatenate it to the existing child solution under construction. 3) Select another segment from parent 1. This time it is (71). The 7 causes a conflict. Replace the 7 by random selection to form (81). Add this new segment to the forming solution. 4) Select another segment from parent 2. (50) is selected. Treat the conflict (the 0) as before and finish the child solution.

```
Parent 1:   9 4 3 2 6 8 7 1 5 0
Parent 2:   3 5 8 7 0 4 6 9 2 1

Child:      3 2 6 7 0 4 8 1 5 9
```

4.2.21 Subtour Replace Operator

[12]

The subtour replace operator examines the two solutions to be crossed for similarly located subsequences that have the same elements (and length at least two). When two such subsequences are found in the respective solutions, they replace each other in the two solutions. The following example demonstrates.

1) Choose a subtour from parent 1. In this example, let it be (1320). 2) Look for a subtour in parent 2 that is a subset of the one selected from parent 2. In this example, we find the subtour (1230). Note also that a larger subtour could have been selected from parent 1 but, by chance, was not. 3) Swap the two subtours to construct the two offspring solutions.

```
Parent 1:  9 8 4 5 6 7 1 3 2 0
Parent 2:  8 7 1 2 3 0 9 5 4 6

Child 1:   9 8 4 5 6 7 1 2 3 0
Child 2:   8 7 1 3 2 0 9 5 4 6
```

4.2.22 Weighted Chunking

[12]

The weighted chunking operator can be considered a heuristic operator since it uses domain knowledge to construct better subtours. This operator is exactly like the subtour chunking operator, but it also incorporates a bit of domain knowledge - that of due dates. When two chunks collide in the newly built solution, one of the chunks is moved around the other and this decision is made according to the due dates of the boards. Under many circumstances, this operator behaves exactly as the subtour chunking operator but the expectation is

that the disturbance to exchanged subsequences. This use of the term disturbance refers not to the number of positions affected but to the magnitude of effect for each position. An example follows.

1) Choose a chunk from parent 1. Suppose it is (6871). Place it into the child in the same position it came from. 2) Choose a chunk from parent 2. Suppose it is (046921). Trim the chunk down to (04) to avoid conflicts. The chunk may now be placed in either the second and third positions or in the ninth and tenth positions. The new chunk is placed to the right of the obstacle because the due dates for jobs #10 and #4 are relatively late compared to the due dates of jobs #6, #8, #7 and #1. 3) Choose a chunk from parent 1. Suppose it is (943). Trim the chunk down to (9) and place it in the first position. 4) Choose a chunk from parent 2. Suppose it is (358). Trim it to (35) and place it in the second position to avoid the obstacle in the first position. 5) Choose a chunk from parent 1. Suppose it is (50). Both the 5 and and 0 have already been placed in the offspring solution so no action is taken. 6) Choose a chunk from parent 2. It must be (7). This chunk is trimmed to nothing and no further action is taken. 7) Choose a chunk from parent 1. It must be (2). Place this chunk in the third position.

```
Parent 1:  9 4 3 2 6 8 7 1 5 0
Parent 2:  3 5 8 7 0 4 6 9 2 1
Child:     9 3 5 2 6 8 7 1 0 4
```

4.2.23 Analogous Crossover

[19]

Analogous crossover is a modified crossover operator designed to work with order dependent production programs. Cross points are determined according to the phenotypic function of parameters, and crossover is performed on variable length strings. Thus, the locus of the cross point of the first parent has no part in determining the locus of the corresponding cross point in the second parent. The second parent's cross point is determined according to the proximity the parameters have in the phenotype space.

As an illustration, consider a road system. Driving between two arbitrary points A and B, one can devise several alternative routes, while some routes may even have shared journeys. The logical and economical way of combining different routes is to switch over at either shared stretch of road, or at intersections of routes. Switching routes is more likely to occur at route positions where the two routes are close to each other. Analogous crossover is problem dependent, and the exact implementation of this operator will depend on the interpretation of the phenotype space.

4.2.24 Segregation Crossover

[21]

When using analogous crossover with multi-bit parameters (as opposed to single-bit parameters), the crossover operation which crosses between genes is called segregation crossover. Davidor's use of segregation crossover was used to optimize a robot arm. The arm had two elbows and an end (called an end-effector). The problem was to have the end-effector construct a straight line (called a trajectory) given a set of instructions that manipulated the arm.

The actual position of the arm (i.e. the phenotypic information as decoded from a chromosome) determined the crossover points. The algorithm follows:

```
1. Choose two parent trajectories
2. Randomly choose a cross point in one parent trajectory (respecting
   segregation boundaries
3. Slide the second parent along the first end-effector position at the
   cross point of the first parent in order to find the most analogous
   end-effector position
4. Repeat steps 1 and 2 for each of the desired number of cross points
5. Copy the counterpart sub-trajectories from the two parents to obtain
   a new offspring trajectory
```

4.2.25 Edge Recombination

[67]

Edge recombination is not really an order-based crossover method, but since it is used to handle problems with order-based representations, it it appropriate to mention it here.

Consider the traveling salesman problem with a tour of 6 cities. We have selected the following two tours: ($ABCDEF$) and ($BDCAEF$). If we represent the "link" information in the tour we have "ab bc cd de ef af" and "bd dc ca ae ef bf" as parent tours (we assume that the tours are circular). Note that edge "ab" has exactly the same value as "ba". An edge operator constructs an offspring tour by exclusively using links present in the two parent structures.

To do this, use an "edge map" that stores all connections from the two parents that lead into and out of a city. Since the distance is the same between two cities whether one is coming or going, each city will have at least two and at most four edge associations – two from each parent. The edge map for the above tours is as follows:

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

The edge map is used to make it possible to choose the next city in the tour such that thos cities which have the fewest unused edges (and thus appear to have the hightes probability of being isolated) have priority of being chosen next. The candidate with the fewest edge connections is chosen. In the case of a tie, the algorithm chooses randomly from the candidate edges.

To start, the child tour is initialized with one of the two initial cities from its parents. In the above case, A and B both have four edges, so the algorithm will make a random choice. Assume B is chosen. The candidates for the next city are A C D and F. C, D and F all have two edges (the initial three minus B – A now has three edges and thus is not considered) so randomly choose between C, D and F. Assume C is chosen. C now has edges to A and D. D is chosen next, since it has fewer edges. D only has an edge to E, so E is chosen next. E has edges to A and F, both of which have one edge left. Randomly choose A. A must now go to F. The resulting tour is $(BCDEAF)$ and is composed entirely of edges taken from the two parents.

4.2.26 Edge-2 Recombination

[62]

Edge-2 recombination is an improved version of edge recombination described above (see Edge Recombination). When originally implemented, it was noticed that there was no active mechanism to preserve "common subsequences" between the two parents. As a solution, when the edge table is constructed, if an item is already in the edge table and we are trying to insert it again, that element of the sequence must be a common edge. The elements of a sequence are stored in the edge table as integers, so if an element is already present, the value is inverted: if A is already in the table, change the integer to -A. The sign acts as a flag. Consider the following sequences and edge table.

```
Parent 1: a b c d e f
Parent 2: c d e b f a

a: b, -f, c
b: a, c, e, f
c: b, -d, a
d: -c, -e
e: -d, f, b
f: e, -a, b
```

The new edge table is the same as before except for the flagged elements. One of three cases holds for an edge table entry. 1) If four elements are entered in the table as connections to a given table entry, that entry is not part of a common subsequence. 2) If three elements are entered as connections to a given table entry, then one of the first two elements will be negative and represents the beginning of a common subtour. 3) If only two elements are

entered for a given table entry, both must be negative and that entry is an internal element in a common subsequence. Giving priority to negative entries when constructing offspring affects edge recombination for case 2 only. In case 1, no connecting elements have negative values, and in case 3 both connecting elements are negative, so edge recombination behaves just as before. In case 2, the negative element which represents the start of a common subtour is given first priority for being chosen. Once this common subsequence is started, each internal element (case 3) of the sequence has only one edge in and one edge out, so it is guaranteed that the common sections of the sequence will be preserved.

4.2.27 Edge-3 Recombination

[50]

Edge-3 recombination is essentially edge-2 recombination with an additional failure guarding mechanism (see Edge and Edge-2 Recombination). When a potential failure occurs during Edge-3 recombination, an attempt is made to continue construction at a previously unexplored terminal point in the tour. Consider the following parents.

```
Parent 1: g d m h b j f i a k e c
Parent 2: c e k a g b h i j f m d
```

A terminal is a city (in the Traveling Salesman Problem, for example) which occurs at either end of a partial tour, where all edges in the partial tour are inherited from the parents. The terminal is said to be live if that city still has entries in its edge list; otherwise it is said to be a dead terminal. Choose city a (randomly) to start the tour. This will serve as a new terminal in the event of a failure. Conceptually this is the same as inverting the partial tour to build from the other end. This situation is best described by the following sequence of events.

```
1) Start Tour            [L1]
2) Build Tour            [L1 ... L*]
3) First Failure Occurs  [L1 ... D1]
4) Reverse Partial Tour  [D1 ... L1]
5) Continue Partial Tour [D1 ... L*]
6) Second Failure Occurs [D1 ... D2]
```

In his representation scheme, $L1$ represents the city chosen to randomly start the tour. It is a live edge, but after step 1, it is not currently active. $L*$ is used to indicate a live terminal that is currently being extended. $D1$ represents the first dead terminal and $D2$ represents the second dead terminal.

After both terminals on the partial tour are dead, we now have no choice but to start a new subtour. A new city, $L3$, is randomly chosen as the new live terminal. The construction of the next partial tour is illustrated as follows.

```
 7) Start Subtour    [D1 ... D2] [L3]
 8) Build Tour       [D1 ... D2] [L3 ... L*]
 9) Third Failure    [D1 ... D2] [L3 ... D3]
10) Reverse Subtour  [D1 ... D2] [D3 ... L3]
11) Continue Subtour [D1 ... D2] [D3 ... L*]
12) Fourth Failure   [D1 ... D2] [D3 ... D4]
```

When a failure occurs, there is at most one live terminal in reserve at the opposite end of the current partial tour. In this example, the unused terminal at the opposite end of the partial tour was assumed live. In fact, it is not guaranteed to be live, since the construction of the partial tour could isolate this terminal city. Once both terminals of the current partial tour are found dead, a new partial tour must be initiated. Note that no local information is employed.

The offspring is build using city a as the first in the tour. City a is saved as the reserve terminal. The tour is built exactly the same way as with Edge-2 recombination until the failure occurs at city g where the tour is $[akecdmhbg]$. Since there is a reserve terminal, city a, the edge list of city a is examined for a city that has not been previously used in the tour. In this case, the only city remaining in the edge list of city a is city i. If there were others, then the next city would be chosen according to the previously described priority scheme.

The partial tour inclusive of the live and dead terminals is reversed to produce $[gbhmdceka]$. Then city i is added to the tour after city a. The tour is then constructed in the normal fashion. In this case, there are no further failures. The final offspring tour is $[gbhmdcekaifj]$. The offspring produced has a single foreign edge, $j - g$.

4.2.28 Modified PMX (CrossOver)

[9]

Modified PMX works as follows: 1) Partition each of the parent structures and the initially empty offspring structure into three sections by randomly drawing two integers to serve as crossover points. 2) The second section of the child's structure is obtained by copying terms directly from the corresponding positions in parent 1's structure. 3) The first and third sections of the child's structure are obtained by copying terms from corresponding positions in parent 2's structure which have not already been assigned in step 2. 4) The remaining empty positions in the first and third sections of the child are then obtained from a permutation of the terms not yet allocated in steps 2 and 3. The following example illustrates.

```
Parent 1:  1 2   3 4 5   6 7 8
Parent 2:  6 4   8 3 7   2 1 5
```

```
Step 1:    - - | - - - | - - -
Step 2:    - - | 3 4 5 | - - -
Step 3:    6 - | 3 4 5 | 2 1 -
Step 4:    6 8 | 3 4 5 | 2 1 7
```

4.2.29 Voting Recombination

[2, 52]

In voting recombination each parent gets to vote on the desired allele value at each locus in the chromosome. In Mühlenbein's p-sexual voting recombination, p parents vote on the allele values to place in the child. In addition, a threshhold value tells how many parents must be in agreement before the vote is good (think of it as the definition of a majority).

In the following example, four parents are voting on the allele values to go into the child. The threshhold is three, meaning that three parents must agree before the child gets the allele. Any positions not agreed upon by the parents are then determined by mutation.

```
Parent 1: 1 4 3 5 2 6
Parent 2: 1 2 4 3 5 6
Parent 3: 3 2 1 5 4 6
Parent 4: 1 2 3 4 5 6

Child:    1 2 - - - 6  (after voting)
Child:    1 2 4 5 3 6  (after mutation)
```

Note that this method would work for most any representation.

4.2.30 Assignment Crossover

[13]

Assignment crossover is a modification of Syswerda's position-based crossover. Assignment crossover ensures that the absolute allele assignments of the parents are maintained in the children as much as possible (not just relative positions). Three steps are involved. 1) Exchange allele information at selected positions; 2) Copy the alleles of a parent into the empty positions of the corresponding child (e.g., from parent 1 to child 1), unless the child just inherited the value from the other parent, and 3) place the crossed over values of a parent (e.g., alleles from parent 1 to child 2) into the remaining empty slots of its own child, if possible. Four positions are randomly selected for application of the operator. An example follows.

```
Parent 1:            0 1 3 7 9 2 5 6 8 4
Parent 2:            7 3 5 9 8 4 0 2 6 1
Selected Positions:  *   *       * *
```

```
Child 1:            7 1 5 3 9 6 0 2 8 4
Child 2:            0 7 3 9 8 4 5 6 2 1
```

As a result of the assignment crossover above, parents and their corresponding children have the values 1, 9, 8 and 4 in the same position. this crossover operator minimizes the disruption in the assignment of the digits.

4.2.31 Precedence Matrix Representation

[33]

Assume the tour $(BACED)$. This can be represented by a precedence matrix. In a precedence matrix, a row tells which elements follow an element of the tour, and a column tells which elements precede an element of the tour. The following matrix describes our tour.

```
   A  B  C  D  E
A  0  0  1  1  1  3
B  1  0  1  1  1  4
C  0  0  0  1  1  2
D  0  0  0  0  0  0
E  0  0  0  1  0  1
   1  0  2  4  3
```

Row A says that elements C, D and E are to the right in the tour. Column A says that element B is to the left. Note the sum of the rows. A has 3 elements to the right. Note the sum of the columns. A has 1 element to the left. Note that A is in position 2, and this is described by the precedence matrix. The other elements can be placed into position similarly. Note that you only need either the row or column sums to determine the positions. The information is redundant.

4.2.32 Intersection

[33]

Intersection uses the precedence matrix representation described above (see Precedence Matrix). First, two matrices are intersected (using the AND function where 1 AND 1 = 1, anything else = 0). Second, add into that matrix a subset of the ones that are unique to one of the parents. Finally, convert this matrix to a sequence through an analysis of the row/column sums. This is analogous to completing the matrix without th necessity of actually performing the matrix manipulations.

4.2.33 Union

[33]

Union uses the precedence matrix representation described above (see Precedence Matrix). First, partition the set of symbols (cities) into two disjoint sets. Second, construct the matrix which contains the bits from the first parent that define the relationships within the first subset of cities and construct the matrix which contains the bits from the second parent that define the relationships within the second subset of cities. Third, perform the union (logical OR where 0 OR 0 = 0, anything else = 1) of these two matrices resulting in a matrix that contains unique attributes from both parents but with some attributes still undefined. Finally, convert this underconstrained matrix to a sequence through an analysis of the row/column sums.

4.2.34 Epistasis Based and Extra Knowledge Operators

[22]

Epistasis based reproduction assigns a pseudo-fitness value to each of the symbols in each of the parameters in the string. This first order statistic [20] is based on the parent sample from which offspring are produced. Thereafter, offspring are created with the "best value" symbols. First offspring with the highest values combination, second offspring with the second best values combination, and so on. There are eight epistasis based operators:

```
- from 2 parents make 1 offspring,
     use the best pseudo-fitness parameters
- from 2 parents make 2 offspring,
     use the best and second best pseudo-fitness
- from 2 parents make 1 offspring,
     assign bits probabilistically to pseudo-fitness
- from 2 parents make 2 offspring,
     assign bits probabilistically to pseudo-fitness
- from 4 parents make 2 offspring,
     use the best and second best pseudo-fitness
- from 4 parents make 4 offspring,
     use the best and second best pseudo-fitness
- from 4 parents make 2 offspring,
     assign bits probabilistically to pseudo-fitness
- from 4 parents make 4 offspring,
     assign bits probabilistically to pseudo-fitness
```

Extra knowledge operators are the same as epistasis based, but extract the pseudo-fitnesses from information given to the system from the outside (by us). Four operators are included in this category.

```
- from 2 parents make 1 offspring, assign bits probabilistically
- from 2 parents make 2 offspring, assign bits probabilistically
- from 2 parents make 1 offspring, assign best pseudo-fitness
```

- from 2 parents make 2 offspring, assign best and worst pseudo-fitness

4.3 Real-valued recombination

Real-valued operators work on strings (arrays) of real numbers. When the values are constrained, a simple mapping can transform the real numbers into bit strings of arbitrary precision. From here, the standard bit operators can be used. To further assist in the mapping, gray coding can be used to ensure that adjacent values have a Hamming distance of 1 from their two neighbors. According to the schema theorem, this reduces epistasis (nonlinear relationships between sets of bits in the string).

However, much work has been done with actual real-coded operators. The following is a summary from the literature.

Davis [25] proposed a set of "guaranteed" operators for real-valued strings. The "guaranteed" feature is a local check that an operator is not producing a child that is a copy of its parents. If, after any of these operations, the child is identical to one of the parents, the child is discarded.

The guaranteed operators include uniform, average, mutation, big creep and little creep. They are summarized below. Note that in the following discussions, the term field refers to a gene or string element, while value refers to the allele value at that gene.

4.3.1 Guaranteed Uniform

[25]

Create two children from two parents by deciding randomly, field by field, which child receives the corresponding value from which parent.

4.3.2 Guaranteed Average

[25]

Create one child from two parents by averaging their fields. Choose randomly which integer to use if the average lies between two integer values.

4.3.3 Guaranteed Mutation

[25]

Create a child from one parent by passing down the parent's fields and, with 10and the maximum allowed value.

4.3.4 Guaranteed Big Creep

[25]

Create one child from one parent by passing down the parent's fields and, with 20that is 1, 2, or 3 units above or below the field value. Select the amount and the direction of the creep randomly.

4.3.5 Guaranteed Little Creep

[25]

Create one child from one parent by passing down the parent's fields and, with 10that is 1 unit above or below the field value. Select the direction of creep randomly. Do not go below 0 or the maximum allowable value in creeping values.

4.3.6 Simple Crossover

Any of the bit-based crossover methods easily extend to real valued alleles. A string of length L is a list of L real numbers. Given two parents, 1-point crossover (or any of the others) can be performed exactly as with bit strings. Instead of taking a bit-valued allele from a locus in a parent, we take a real-valued allele. A simple mutation is just some mathematical operator performed on a single allele value. These are similar to bit-encoded real problems where crossover points are restricted to the L boundaries (i.e. you cannot split the bit-encoded real value across parents), but mutation can change any bit.

4.3.7 Arithmetical Crossover

There are any number of varieties of arithmetical crossover. Given two allele values from two parents (at the same locus), devise a mathematical function to determine a new allele value.

4.3.8 Linear Crossover

[68]

Consider the list of L real values as a single point in L-space. Linear crossover takes two parents, p_1 and p_2 and generates three new children with the formulas $\frac{1}{2}p_1 + \frac{1}{2}p_2$, $\frac{3}{2}p_1 - \frac{1}{2}p_2$ and $-\frac{1}{2}p_1 + \frac{3}{2}p_2$. The first is the midpoint while the other two line on the line determined by p_1 and p_2.

4.3.9 Flat Crossover

[56]

Choose parameters for an offspring by uniformly picking parameter values between (inclusively) the two parents parameter values.

4.3.10 Blend Crossover (BLX-α)

[32]

Blend crosover is a generalization of flat crossover (see Flat Crossover). If uniformly picks values that lie between the two points that contain the two parents, but may extend equally on either side determined by a user specified GA parameter α. For example, BLX-0.5 picks parameter values from points that lie on an interval that extends $0.5I$ on either side of the interval I between the parents. BLX-0.0 is equivalent to flat crossover.

4.4 Subset-based recombination

Very little has been done with subset-based problems in the GA literature. Only a small handful of operators surfaced.

4.4.1 Subset Crossover

[10]

Lucasius defines four types of subset crossover: Identity preserving, parental cross-section preserving, position preserving and order preserving. Consider the following parent strings in a problem where the full set is size $N = 9$ and the subset size $n = 5$.

```
Parent 1: 1 8 3 4 7
Parent 2: 6 2 4 9 1
```

Subset crossover, in this case somewhat similar to uniform order crossover, is described as follows.

```
1. Create a random binary mask of a size identical to the length of a
   parent string (size $n$).

   Parent 1: 1 8 3 4 7
   Mask:     0 1 0 1 1
   Parent 2: 6 2 4 9 1

2. Copy the 1-masked elements in parent 1 to the same respective
   positions in the child (which thereby inherits identidy, position
   and order information).

   Child:    - 8 - 4 7

   The -'s denote undefined elements.  Let nC be the number of defined
   elements in the child (nC <= n, and in this example, nC = 3).  Let
   dC = n - nC.
```

3. Mark in parent 2 all elements that are now defined in the child as undefined elements (in this example, there are 4).

   Parent 2: 6 2 - 9 1

   Let nP2 be the number of defined elements in parent 2 (nP2 <= n, and in this example, nP2 = 4). Note that the inequality nC <= nP2 must apply at this stage.

4. If dC = nP2

   Copy each defined element in parent 2 to an undefined element in the child as follows:

   a) In identity preserving mode
      Copy without any preference per se for position or order (randomly)
   b) In position preserving mode
      Wherever possible, copy a defined element in parent 2 to an undefined element in the child at the same position. Remaining elements are copied without any preference per se for order (randomly)
   c) In order preserving mode
      Copy the defined elements in parent 2 to the undefined elements in the child, retaining the order they have in parent 2

5. Otherwise (dC < nP2)

   Copy each element in a fraction of dC elements among the nP2 defined elements in parent 2 to an undefined element in the child as follows.

   a) In identity preserving mode
      Select the fraction randomly, then proceed as in 4a. In this example, if the dC (= 2) selected elements in parent 2 are for instance 9 and 6, then child becomes (9 8 6 4 7).
   b) In position preserving mode
      Select the fraction semi-randomly, i.e. a number of elements in the fraction is selected deterministically in such amount as to maximize preservation of position in the next step, which is to proceed as in 4b. In the example, this means that in parent 2, 6 must be selected and the other dC - 1

```
        (= 1) elements randomly, for instance 2 in the example.
        The child then becomes (6 8 2 4 7).
    c) In order preserving mode
        Select the fraction randomly, then proceed as in 4c.  In the
        example, if the dC (= 2) selected elements in parent 2 are
        for instance 2 and 9, then the child becomes (2 8 9 4 7).
```

To obtain the second child string, repeat the procedure with the roles of the parent strings interchanged.

A modification of the identity preserving mode is to preserve the parental cross-section. This constrains the creation of the binary template in Step 1 of the description of subset crossover. For instance, in the above example, since the elements 1 and 4 comprise the parental cross-section, the binary template for parent 1 must be of the form (1??1?), where the wildcard symbol, ?, denotes unconstrained bit values that can be chosen randomly. Similarly, a modification of the position and order preserving modes is to preserve either one or two segments of one of he parent strings in a fashon similar to segmented order crossover or partially mapped crossover. Again, this constrains the creation of the binary template in Step 1 of the description of subset crossover, namely the template must be comprised of either two or three uniform segments which adjacently alternate by bit value, e.g.: 00111 (two uniform segments: 00 and 111), 11001 (three uniform segments: 11, 00 and 1), 01100 (three uniform segments: 0, 11 and 00), etc. These constraints imply that there are either one or two 1-segments that mask the selected parent string, as stated before.

4.4.2 Trade Mutation

[10]

Given a set of size N and a fixed-length subset of size n, trade mutation exchanges a member of the subset with a member of the subset's complement. For example, with $N = 10, n = 6$, a parent string (260349) and a complementary string (1578), randomly select a member from the subset and the complement (say number 2 in the subset and number 3 in the complement) and exchange them to create the new string (270349).

This was called asexual crossover in [17, 15], but is more properly termed a mutation or a constrained 2-opt on a permutation.

4.4.3 Uniform Subset Crossover

[18]

Given a set size of N and a fixed-length subset of size n, uniform subset crossover takes two subsets and performs a standard uniform crossover operation. A cleanup phase replaces any duplicate alleles with members outside of the set. Alternatively, a child can be constructed from a union of the two parents,

randomly selecting n values. Both of these crossover operators suffer heavily from allele loss, and tend to prematurely converge without additional selection pressure, removal of duplicates from the population, etc.

4.5 Hybrid Methods

4.5.1 Problem-Specific Recombination

Representation in GAs refers to how a potential solution to a problem is mapped onto some data structure. The above examples dealt with solutions mapped onto bit strings, integer permutation, integer subsets and real numbers.

There are those solutions that do not (easily or usefully) map onto any of the above structures. [1]

Many problem-specific crossover operators exist in the literature: Job Shop Scheduling [4, 69], k-way Graph Partitioning [66], constrained problems [51], G/SPLINES [60], Database Query Optimization [5], Clustering [7], Route and Schedule Learning [34], Partitioning Problems [42], Primitive Extraction [54],

4.5.2 Problem-Independent Recombination

[57]

Radcliffe has illustrated some general concepts of genetic algorithms using analysis of formae. In this model, general operators that manipulate formae are introduced and discussed. These are beyond the scope of this survey but are mentioned here for reference.

A recombination operator which makes a uniform random choice of children from the similarity set of the parents is called random, respectful, recombination (also called R^3). Random assorting recombination (RAR) and random transmitting recombination (RTR) are two other methods explored by Radcliffe in [58]. In [59] a generalized N-Point Crossover (GNX) and Binomial Minimal Mutation (BMM) are described.

5 Classifying Crossover

Crossover has been classified in numerous ways. The most obvious divisions deal with representation, leading to bit-based, order-based and real-valued groups. These divisions are standard and widespread in the literature. Hybrid methods are often mentioned, but as the name suggests, the point is that they deviate from the standard groups, and are thus lumped together as a large disjoint

[1] Of course, the solution to any computer-solvable problem can be mapped into a bit string, permutation, etc. But these simple mappings can be either too complex, too unwieldy, result in huge bit strings, be completely arbitrary (which artificially introduces epistatic relationships between genes), or some combination of the above. Often the best solution is to abandon traditional representations and hand-tailor a solution.

collection of strategies. To a much lesser extent, fixed-length subset problems have formed yet another group. Note also that we are only dealing with genetic algorithms as opposed to genetic programming, evolution strategies, etc.

In the next few sections we examine much of the work done to date in the classification and analysis of crossover.

5.1 Crossover Bias

Eshelman, Caruana and Schaffer [31] developed the concepts of bias in crossover. Their notions of positional and distributional bias describe the effect of an operator on schemata. In short, an operator has positional bias to the extent that the creation of any new schema by recombining existing schemata is dependent upon the position of the location of the bits in the chromosome. An operator has distributional bias to the extent that the amount of material that is expected to be exchanged is distributed around some value or values as opposed to uniformly distributed across the chromosome.

For example, one-point crossover has a very strong positional bias, but no distributional bias. In one-point crossover, a position in the string is selected, and then the bits to the right of that position are exchanged to produce the children. Bits that are close together on the string (i.e. low-order, short defining length) are less likely to be disrupted than those that are farther apart, leading to the positional bias. However, the amount of material that will be exchanged varies uniformly between 1 and L-1 bits, and thus there is no distributional bias. Uniform crossover provides the opposite case. The position of a bit on the string has no bearing on its likelihood of being disrupted from any other bit. Its positional bias is low. However, on the average, $\frac{L}{2}$ bits will be exchanged between the parents, so its distributional bias is high.

This crossover classification was used to demonstrate that one-point crossover was not as advantageous as previously thought. Uniform crossover, although it suffers from disruption, wins out in many cases. This was not generally regarded to be true when this work was done.

5.2 Landscapes

Few terms have drawn as much fire in the GA community as the term landscape. Arguments aside, landscape simply refers to the entire set of chromosomes (solutions) and associated fitnesses. In other words, the landscape is the search space. From this concept, such terms as peaks or hills, valleys or troughs, and so on, have emerged.

There are two major problems with these concepts. First, most GA spaces are discrete (with the exception of real function optimizers, termed GAFOs by DeJong [29], and probably a small handful of others). Thus, a hill is described by neighboring chromosomes with differing fitness. No derivative exists, however, so hills can only be sampled probabilistically.

Which leads to the second problem. What defines neighboring chromosomes? For many operators, every chromosome can be reached from every other in a single reproduction step (bit-based uniform crossover, for example), although not with equal likelihood. Thus, the search space is a fully connected graph with probabilities that define the likelihood of traversal [43, 44]. It can be said, then, that neighboring chromosomes are chromosome sets with a high likelihood of traversal between them via crossover. The more likely we are to get from chromosome A to chromosome B with crossover, the more "neighboring" these chromosomes are said to be.

Landscapes are important because crossover is the primary mechanism for moving around in the search space (mutation would be the secondary mechanism). It also helps us understand the concepts described in the following section.

5.3 Exploration and Exploitation

A good GA balances exploration and exploitation. These terms have their basis in intuition. Consider a landscape with peaks, valleys and flat spaces. Moving about freely in the landscape, even great distances in a small number of steps, is considered exploring. Looking around in a localized area such as a peak, is considered exploiting.

Exploitation is performed by selection. Crossover knows nothing about fitness. It simply explores (in the canonical GA, that is – some hybrid methods include fitness knowledge in the operators).

Guideline 5 *Crossover should explore, not exploit (selection exploits).*

6 Crossover Design Guidelines

We now examine some of the most important considerations for crossover operator design.

6.1 Representation

Clearly representation is an issue long before crossover. Genotype and phenotype are two important concepts in GAs. The genotype is the actual chromosomal representation of a solution to a problem. If you are using bit strings, then that bit string is called the genotype. The genotype space, then, is the hypercube or graph defined by the bit strings and the operations on them such as crossover and mutation.

The phenotype is what the genotype represents in the real world. For instance, with the traveling salesman problem, the genotype is the ordered list of unique integers (using an order-based representation), while the phenotype is the actual tour through the cities.

Now we can make some observations about representation, or the genotype to phenotype mapping. If the genotype doesn't closely match the phenotype, the GA will likely not work. Given two similar schemata (110##10 and 111##10, for example), we expect them to map to fairly similar phenotypes. If the phenotypes are radically different (from a single bit being different in the schemata), and if this is generally true for most of the schemata, then the GA will never be able to effectively search the space since the search is driven by selection, and stochastically sampled by crossover.

6.2 Gray Coding

Researchers have argued for the past two decades about the relative merits of bit-based versus real-valued encoding schemes [3, 28, 40, 41, 68]. Inconclusive debates aside, real-valued problems can be easily encoded into bit strings. But what happens when this is done?

Consider the integers 0 to 7 encoded into 3 bits. The following table illustrates. Note the integers and their bit representations, plus one other column denoting the Hamming distance to the next integer (+1). The Hamming distance is the number of bits that are different between two bit strings.

integer	bits	Hamming distance
0	000	1
1	001	2
2	010	1
3	011	3
4	100	1
5	101	2
6	110	1
7	111	3 (wrap back to 0)

Note how the 3 and 4 are adjacent in real space, but are maximally distant in bit space. What happens is that the bit space encoding bears little resemblance to the real space. Thus, schemata are not usefully processed, and the GA struggles in its search.

To help alleviate these difficulties, the bit mapping can be Gray coded. Gray coding sets up the mapping such that a single move in real space (in this case, + or - 1) corresponds to a single bit flip in bit space. The following illustrates.

integer	bits	Gray code	Gray Coded Hamming distance
0	000	000	1
1	001	001	1
2	010	011	1
3	011	010	1

```
4        100       110               1
5        101       111               1
6        110       101               1
7        111       100               1 (wrap back to 0)
```

Guideline 6 *Gray coding can minimize Hamming distances between neighboring chromosomes.*

6.3 Cross-Representational Encoding

Since real-valued problems can be successfully mapped into bits, why not order-based problems? Again, we have the same problem, but worse. Consider the following example.

permutation	bits	Gray code
1 2 3	000	000
1 3 2	001	001
2 1 3	010	011
2 3 1	011	010
3 1 2	100	110
3 2 1	101	111

Here we have multiple problems. First, the entire bit string isn't used. Thus, we will have to incorporate constraints on crossover, a penalty function for straying out of the space, etc. Second, even with Gray coding we have cases where small changes in the genotype correspond to large changes in the phenotype. Similar schemata bear little resemblance to each other in the phenotype (note here that we are speaking of an intermediate phenotype, which is just the permutation - we assume that the permutation maps nicely into the phenotype, so we just use the term in this case to mean the permutation or the actual real-world meaning of the permutation).

The most important lesson to glean in this analysis is that manipulating the genotype via crossover must make sense in the phenotype. Schemata represent a set of characteristics in the phenotype. If very similar schemata represent totally different characteristics in the phenotype, then crossover will be unsuccessful and your representation and operators must be changed.

Guideline 7 *Schemata in the genotype (representation) must closely parallel the phenotype (real world).*

Guideline 8 *In general, small (large) changes in the genotype should produce small (large) changes in the phenotype.*

7 Crossover Theorems

Balance, covering and schema preservation are fundamental components of a good crossover operator. The following section provides a formal definition of these concepts.

This work will be submitted to the 1996 Foundations of Genetic Algorithms conference (FOGA) [16].

7.1 Bit-based Theorems

To examine the effect of crossover and mutation we will make some simplifying assumptions. We assume parents are chosen in a uniform random fashion. This will allow us to strictly measure the effect of a crossover operator. We use uniform random selection, which ignores fitness, and we assume a uniform randomly generated infinite population.

A crossover mask is a bit string of length L which dictates how the bits of two parent strings are distributed to a child string. Bit i of the child is taken from bit i of parent 1 if bit i of the crossover mask is 1, and from bit i of parent 2 if the bit i of the crossover mask is 0 ($i = 0..L-1$). This is a generalized notion for N-point crossover (including 1-point, 2-point and uniform crossover). Define S to be the set of all binary strings of length L.

$F: S \rightarrow \Re, (\forall a \epsilon S)$
$F(a)$ is the fitness value of the string.

$\Delta_a: S \rightarrow \Re$
Δ is a parametrized family of functions.

Given any $a \epsilon S, \Delta_a$ is defined to be $\Delta_a(b) = F(b) - F(a); \forall b \epsilon S$.

$p_a: S \rightarrow [0,1]$, given any $a \epsilon S, p_a$ is the transition probability distribution function (pdf) for string a, e.g. $p_a(b)$ is the probability that string a will be transformed into string b by a specified genetic operator, say $\mathcal{R}$. The function p_a is essentially the i^{th} column of the transition matrix from a Markov model for Genetic Algorithms (GA), where i is the result of the one-to-one mapping from strings in S to the integers, and $0 \leq i < 2^L$.

Define "$\mathcal{R}$ covers S" to mean the operation $\mathcal{R}$ will, given randomly chosen strings, eventually generate every member of set S.
Define "$\mathcal{R}$ preserves input schemata" to mean the output from operation $\mathcal{R}$ will have 2^i schemata from parent 1 and 2^j schemata from parent 2, where $0 \leq i, j \leq 2^L$, and $i + j = L$.

The following three rules are formalizations of the concepts discussed earlier:

Balanced Recombination: Define $\Omega = \{\mathcal{R}: \sum_{\forall a \epsilon S} E(\Delta_a(b)) = 0$ under the operation of $\mathcal{R}\}$
Covering Recombination: Define $\Xi = \{\mathcal{R}: \mathcal{R}$ covers $S\}$
Schema Preserving Recombination: Define $\Phi = \{\mathcal{R}: \mathcal{R}$ preserves input schemata $\}$

Note that crossover should always be balanced, covering and schema preserving, while mutation will only be balanced and covering.

Theorem 1 *If $p_a(b) = p_b(a)$ for any recombination operator $\mathcal{R}$ and any $a,b \in S$, then $\mathcal{R} \in \Omega$.*

Proof: Given any two strings a and b, the change in fitness after recombination will be $F(b) - F(a)$ (where string a is crossed with some other string c, also in S, or where string a is permuted into string b directly). The likelihood of this is $p_a(b)$, so the expected change in fitness is given by $p_a(b)(F(b) - F(a))$. Conversely, given any b and a, the change in fitness after crossover will be $F(a) - F(b)$, and the expectation will be $p_b(a)(F(a) - F(b))$. Assuming strings a and b are equally likely to be drawn as parent or child (i.e. for every unique a,b there is an equally likely b,a), if $p_a(b) = p_b(a)$, and remembering that the expectation is a sum of the fitness changes times the probability of those changes, then

$= p_a(b)(F(b) - F(a)) + p_b(a)(F(a) - F(b))$
$= p_a(b)(F(b) - F(a)) + p_a(a)(F(a) - F(b))$
$= p_a(b)(F(b) - F(a) + F(a) - F(b))$
$= 0$. □

Theorem 1 establishes the fact that if there is symmetry in recombination probabilities (in fact, the transition matrix of the Markov model is symmetric), then we have the level playing field discussed earlier (this is not the only way this can be accomplished, but it is the most likely). We now prove that N-point crossover (and thus, 1-point, 2-point and uniform crossover) has this property, and thus establishes this level playing field.

Theorem 2 *N-point crossover is $\in \Omega$.*

Proof: We will prove that $p_a(b) = p_b(a)$ for any a and b of length L in S.

Case 1: If there is no possible way to generate b from a, there is also no possible way to generate a from b. Both probabilities are 0. □

Case 2: Consider a crossover mask of size L defined by N-point crossover. Values of 1 indicate that the child gets the corresponding bit from parent 1 (which is a in $p_a(b)$, and b in $p_b(a)$). Values of 0 indicate that the child gets

the corresponding bit from parent 2. For any crossover mask we will have i $1's$, which means there are exactly 2^i possible strings (randomly chosen from our infinite population) that we can cross with parent 1 to get the child. This holds for crossing a to get b and vice versa, thus, the probabilities are equal. □

Theorem 3 *Probabilistic bit flipping (mutation) is* $\in \Omega$.

Proof: The probability of generating string b from string a using probabilistic bit flipping is $\mu^h(1-\mu)^{L-h}$, where h is the Hamming distance between a and b and μ is the mutation rate ($0 < \mu < 1$). The probability of generating string a from b is the same, thus $p_a(b) = p_b(a)$. □

We have now demonstrated a level playing field for these recombination operators. But we still need a way to show that our operator will be able to cover the entire set. We want to make sure we can generate all possible strings in the set. The following theorems do this.

Theorem 4 *N-point crossover is* $\in \Xi$.

Proof: Assume we are attempting to generate a string $c \in S$ of length L under N-point crossover with a crossover mask m. If i is the number of $1's$ in the crossover mask, there are exactly 2^i possible strings that would qualify as parent 1 and 2^{L-i} possible strings that would qualify as parent 2, where $0 \leq i \leq L$. Thus, for any string c and any crossover mask m, there will always be 2 parents (random selection on an infinite population) that will produce c. □

Theorem 5 *Probabilistic bit flipping (mutation) is* $\in \Xi$.

Proof: Consider a string a of length L in S and a mutation rate $0 < \mu < 1$. Let h be the Hamming distance between a some other string in S. The likelihood of producing any string in S from a is $\mu^h(1-\mu)^{L-h}$. This value is nonzero and positive for all $h = 0..L$. □

Lastly, we want to make certain that we preserve the parent's schemata after crossover. Note that for mutation (and relatively low mutation rates), we usually preserve much, but not all, of the parental schemata.

Theorem 6 *N-point crossover is* $\in \Phi$.

Proof: The crossover mask defined by N-point crossover consists of i $1's$ and j $0's$ ($0 \leq i, j \leq L$ and $i + j = L$). This results in 2^i schemata being inherited from parent 1, and 2^j schemata being inherited from parent 2. Combining these into a new child results in $2^i 2^j = 2^L$ total schemata, with all schemata coming from parent 1 and/or parent 2. □

Theorem 7 *Probabilistic bit flipping (mutation) is* $\notin \Phi$.

Proof: Since there is only one parent, any change to the string will result in a loss of schemata. □

Guideline 9 *Crossover must be balanced, covering and schema preserving.*

8 Summary

The following list summarizes our crossover design guidelines.

- Crossover should adhere to the schema theorem.
- Crossover should adhere to the building block hypothesis.
- Use existing operators where applicable, especially with bit-based, order-based or real-valued problems.
- Modify existing operators when possible.
- Crossover should explore, not exploit (selection exploits).
- Gray coding can minimize Hamming distances between neighboring chromosomes.
- Schemata in the genotype (representation) must closely parallel the phenotype (real world).
- In general, small (large) changes in the genotype should produce small (large) changes in the phenotype.
- Crossover must be balanced, covering and schema preserving.

References

[1] D. H. Ackley. *Stochastic iterated genetic hillclimbing.* PhD thesis, Carnegie Mellon University, 1987.

[2] David H. Ackley. An empirical study of bit vector function optimization. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing.* Pittman Publishers, London, 1987.

[3] Jim Antonisse. A new interpretation of schema notation that overturns the binary encoding constraint. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms.* Morgan Kaufmann Publishers, 1989.

[4] Sugato Bagchi, Serdar Uckun, Yutaka Miyabe, and Kazuhiko Kawamura. Exploring problem-specific recombination operators for job shop scheduling. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[5] Kristin Bennett, Michael C. Ferris, and Yannis E. Ioannidis. A genetic algorithm for database query optimization. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[6] Hugues Bersini and Gregory Seront. In search of a good evolution-optimization crossover. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature, 2*, Amsterdam, 1992. Elsevier Science Publishers, B. V.

[7] Jay N. Bhuyan, Vijay V. Raghavan, and Venkatesh K. Elayavalli. Genetic algorithm for clustering with ordered representation. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[8] Lashon B. Booker. Improving search in genetic algorithms. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*. Pittman Publishers, London, 1987.

[9] Donald E. Brown, Christopher L. Huntley, and Andrew R. Spillane. A parallel genetic heuristic for the quadratic assignment problem. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1989.

[10] C. B. Lucacius and G. Kateman. Towards solving subset selection problems with the aid of the genetic algorithm. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature, 2*, Amsterdam, 1992. Elsevier Science Publishers, B. V.

[11] David L. Calloway. Using a genetic algorithm to design binary phase-only filters for pattern recognition. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[12] Gary A. Cleveland and Stephen F. Smith. Using genetic algorithms to schedule flow shop releases. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1989.

[13] Helen G. Cobb. Is the genetic algorithm a cooperative learner? In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, San Mateo, CA, 1993. Morgan Kaufmann Publishers.

[14] Arthur L. Corcoran. *Techniques for Reducing Disruption of Superior Building Blocks in Genetic Algorithms*. PhD thesis, The University of Tulsa, 1994.

[15] Kelly D. Crawford. Solving the n-queens problem using genetic algorithms. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, pages 1039–1047, 1992.

[16] Kelly D. Crawford, Cory J. Hoelting, Roger L. Wainwright, and Dale A. Schoenefeld. Selection vs recombination: Who's driving here? In *Submitted to 1996 FOGA*, 1996.

[17] Kelly D. Crawford, Roger L. Wainwright, and Daniel J. Vasicek. Detecting multiple outliers in multiple dimensions using genetic algorithms. Technical Report UTULSA-MCS-92-4, The University of Tulsa, July 1992.

[18] Kelly D. Crawford, Roger L. Wainwright, and Daniel J. Vasicek. Detecting multiple outliers in regression data using genetic algorithms. In *Proceedings of the 1995 ACM/SIGAPP Symposium on Applied Computing*, 1995.

[19] Yuval Davidor. Analogous crossover. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1989.

[20] Yuval Davidor. Epistasis variance: Suitability of a representation to genetic algorithms. *Complex Systems*, 4(4), 1990.

[21] Yuval Davidor. Improving search in genetic algorithms. In Lawrence Davis, editor, *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, NY, 1991.

[22] Yuval Davidor and Oren Ben-Kiki. The interplay among the genetic algorithm operators: Information theory tools used in a holistic way. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature, 2*, Amsterdam, 1992. Elsevier Science Publishers, B. V.

[23] Lawrence Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1985.

[24] Lawrence Davis. Job shop scheduling with genetic algorithms. In John J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. Lawrence Erlbaum Associates, Publishers, 1985.

[25] Lawrence Davis. Adapting operator probabilities in genetic algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1989.

[26] Lawrence Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, NY, 1991.

[27] K. A. DeJong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, University of Michigan, Ann Arbor, MI, 1975. Dissertation Abstracts International 36(10), 5140B, University Microfilms Number 76-9381.

[28] Kenneth A. DeJong. Genetic algorithms: a 10 year perspective. In John J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications.* Lawrence Erlbaum Associates, Publishers, 1985.

[29] Kenneth A. DeJong. Genetic algorithms are not function optimizers. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, San Mateo, CA, 1993. Morgan Kaufmann Publishers.

[30] Larry J. Eshelman. The chc adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms.* Morgan Kaufmann Publishers, 1991.

[31] Larry J. Eshelman, Richard A. Caruana, and J. David Schaffer. Biases in the crossover landscape. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms.* Morgan Kaufmann Publishers, 1989.

[32] Larry J. Eshelman and J. David Schaffer. Real-coded genetic algorithms and interval schemata. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, San Mateo, CA, 1993. Morgan Kaufmann Publishers.

[33] B. R. Fox and M. B. McMahon. Genetic operators for sequencing problems. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms.* Morgan Kaufmann Publishers, 1991.

[34] Paula S. Gabbert, Donald E. Brown, Christopher L. Huntley, Bernard P. Markowicz, and David E. Sappington. A system for learning routes and schedules with genetic algorithms. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[35] David E. Goldberg and Robert Lingle, Jr. Alleles, loci, and the traveling salesman problem. In John J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications.* Lawrence Erlbaum Associates, Publishers, 1985.

[36] David E. Goldberg and Philip Segrest. Finite markov chain analysis of genetic algorithms. In John J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms.* Lawrence Erlbaum Associates, Publishers, 1987.

[37] John J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man & Cybernetics*, SMC-16(1):122–128, January-February 1986.

[38] John J. Grefenstette, Rajeev Gopal, Brian J. Rosmaita, and Dirk Van Gucht. Genetic algorithms for the traveling salesman problem. In John J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications.* Lawrence Erlbaum Associates, Publishers, 1985.

[39] John H. Holland. Hierarchical descriptions of universal spaces and adaptive systems. Technical Report ORA Projects 01252 and 08226, University of Michigan, Ann Arbor, 1968.

[40] John H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, MI, 1975.

[41] Cezary Z. Janikow and Zbigniew Michalewicz. An experimental comparison of binary and floating point representations in genetic algorithms. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[42] Donald R. Jones and Mark A. Beltramo. Solving partitioning problems with genetic algorithms. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[43] T. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search.* PhD thesis, University of New Mexico, Albuquerque, NM, 1995.

[44] Terry Jones and Stephanie Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In Larry J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, San Mateo, CA, 1995. Morgan Kaufmann Publishers.

[45] Hillol Kargupta, Kalyanmoy Deb, and David Goldberg. Ordering genetic algorithms and deception. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature, 2*, Amsterdam, 1992. Elsevier Science Publishers, B. V.

[46] G. E. Liepins, M. R. Hilliard, Mark Palmer, and Michael Morrow. Greedy genetics. In John J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms.* Lawrence Erlbaum Associates, Publishers, 1987.

[47] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, pages 498–516, 1972.

[48] Sushil J. Louis and Gregory J. E. Rawlins. Designer genetic algorithms: Genetic algorithms in structure design. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[49] Martina Gorges-Schleuter. Asparagos: An asynchronous parallel genetic optimization strategy. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1989.

[50] Keith Mathias and Darrell Whitley. Genetic operators, the fitness landscape, and the traveling salesman problem. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature, 2*, Amsterdam, 1992. Elsevier Science Publishers, B. V.

[51] Zbigniew Michalewicz and Cezary Z. Janikow. Handling constraints in genetic algorithms. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[52] Heinz Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, Inc, 1989.

[53] Heinz Mühlenbein. Evolution in time and space – the parallel genetic algorithm. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers, 1991.

[54] Jean-Pierre Nordvik and Jean-Michel Renders. Genetic algorithms and their potential for use in process control: A case study. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[55] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the traveling salesman problem. In John J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*. Lawrence Erlbaum Associates, Publishers, 1987.

[56] N. J. Radcliffe. *Genetic Neural Networks on MIMD Machines*. PhD thesis, Edinburgh University, 1990.

[57] Nicholas J. Radcliffe. Forma analysis and random respectful recombination. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[58] Nicholas J. Radcliffe. Genetic set recombination. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, San Mateo, CA, 1993. Morgan Kaufmann Publishers.

[59] Nicholas J. Radcliffe and Patrick D. Surry. Fitness variance of formae and performance prediction. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, San Mateo, CA, 1995. Morgan Kaufmann Publishers.

[60] David Rogers. G/splines: A hybrid of friedman's multivariate adaptive regression splines (mars) algorithm with holland's genetic algorithm. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[61] J. David Schaffer and Amy Morishima. An adaptive crossover distribution mechanism for genetic algorithms. In John J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*. Lawrence Erlbaum Associates, Publishers, 1987.

[62] T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In H. Schwefel and R. Maenner, editors, *Parallel Problem Solving from Nature*, Berlin, Germany, 1991. Springer Verlag.

[63] Gilbert Syswerda. Uniform crossover in genetic algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1989.

[64] Gilbert Syswerda. Schedule optimization using genetic algorithms. In Lawrence Davis, editor, *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, NY, 1991.

[65] Gilbert Syswerda and Jeff Palmucci. The application of genetic algorithms to resource scheduling. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[66] Gregor von Laszewski. Intelligent structural operators for the k-way graph partitioning problem. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[67] Darrell Whitley, Timothy Starkweather, and D'Ann Fuquay. Scheduling problems and traveling salesman: The genetic edge recombination operator. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1989.

[68] A. Wright. Genetic algorithms for real parameter optimization. In G. Rawlins, editor, *Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers, 1991.

[69] Takeshi Yamada and Ryohei Nakano. A genetic algorithm applicable to large-scale job-shop problems. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature, 2*, Amsterdam, 1992. Elsevier Science Publishers, B. V.