University of Huddersfield Repository

Bentley, Peter J. and Wakefield, Jonathan P.

Hierarchical Crossover in Genetic Algorithms

Original Citation

Bentley, Peter J. and Wakefield, Jonathan P. (1996) Hierarchical Crossover in Genetic Algorithms. In: Proceedings of the 1st On-line Workshop on Soft Computing (WSC1), 19-30 August 1996, Nagoya University, Japan.

This version is available at http://eprints.hud.ac.uk/3973/

http://eprints.hud.ac.uk/

Hierarchical Crossover in Genetic Algorithms

P. J. Bentley* & J. P. Wakefield

Abstract

This paper identifies the limitations of conventional crossover in genetic algorithms when operating on two chromosomes of differing lengths. To address these problems, the concept of a Semantic Hierarchy (i.e. tree of meaning) of a genotype within a genetic algorithm is introduced. With this in mind, a new form of crossover operator known as Hierarchical Crossover is presented, capable of performing crossover with genotypes of different sizes, while still being functionally equivalent to standard, single-point crossover (or uniform crossover). Various aspects and advantages of this method are discussed. Finally, an example of some results produced by an implementation is shown.

Key words: hierarchical crossover, semantic hierarchy, variable-length chromosome, genetic algorithm, evolutionary design.

1.   Introduction

The genetic algorithm (GA) is a highly efficient and robust search algorithm based on evolution in nature (Holland, 1975). Today, GAs are widely used to evolve good solutions to hundreds of different problems (Goldberg, 1989), and yet most GAs suffer from a significant drawback. Unlike nature, the standard genetic algorithm is incapable of allowing two individuals with different numbers of genes to breed together and produce offspring.

Normally, the standard GA uses single-point crossover to generate offspring. Given two individuals (candidate solutions to a problem) in a population, the GA generates a child solution by randomly taking some of the genes from one individual, and taking the remaining genes from the other. For example, if the two parents had chromosomes: 'ABCDEFG' and 'abcdefg', single-point crossover would pick a random *crossover point*, say 5 in this case, and the children: 'ABCDEfg' and 'abcdeFG' would be generated. These child solutions have inherited genes from both of their parents, and will take their parents' places in the population. Fit individuals (i.e. solutions that solve the problem better) have more offspring than unfit individuals, so, on average, good genes are passed on to offspring more than poor genes, resulting in evolution to good solutions to the problem.

Every gene in the chromosome of an individual typically represents a single, coded parameter of the problem. Evolution in the GA simultaneously finds good values for each of these coded parameters, to construct a good solution to the overall problem. However, for some problems the number of parameters is variable. For example, consider the problem of finding the optimal aerodynamic shape of a car. If 'building blocks' were used to define the shape of the car, not only would the positions and dimensions of these blocks need to be precisely determined, but also the *number* of blocks would need to be determined.

So why should variable numbers of parameters in a problem cause difficulties for GAs? Well, as stated above, every parameter corresponds to a gene in an individual. With variable numbers of parameters, there will exist in a population some individuals with more genes than others. If two such dissimilar individuals were operated upon by standard crossover, the offspring would become corrupted and unusable. To illustrate this, consider the two parents: 'ABCDEFG' and 'abcxyzdefg'. Note that the second parent has three extra genes 'xyz'. When these two are crossed, with a crossover point of, say 6, the following offspring are obtained: 'ABCDEFdefg' and 'abcxyzG'. The first child has three unwanted duplicated or overspecified genes: 'D' and 'd', 'E' and 'e', 'F' and 'f'. The second child has three required genes missing: 'e', 'f' and 'g'. The standard GA would not cope, resulting in a population of garbage (Harvey, 1992).

2.   Background

There are a number of existing attempts to solve the problem of applying crossover to variable-length chromosomes. Smith (1984) used a new 'alignment' stage within his classifier system LS-1, to initially align rules at boundaries before his version of crossover could operate on individuals consisting of rule sets. However, this idea is limited to crossing a variable number of fixed-length rules. Koza (1990) evolves programs defined as LISP expressions which are arranged in hierarchical tree-structures. Koza's crossover simply allows any

* Division of Computing and Control Systems, School of Engineering, University of Huddersfield, Huddersfield HD1 3DH, U.K.
E-mail: p.bentley@eng.hud.ac.uk

branches of the two parent trees to be interchanged. Although all genotypes can be defined hierarchically as will be shown in the next section, few would survive the simple methods of Koza and remain meaningful (i.e. a tree-like genotype with too many or too few branches could be unusable for many problems).

Harp and Samad code neural networks as genotypes, using a fixed length block of genes to define a single layer in the network, with the number of such layers being variable. Their crossover ensures that if a block is split in one parent, another block is split in the same position in the other parent, ensuring that the resulting offspring is meaningful. Harvey (1992) also uses a GA to evolve neural networks, this time using a syntactic comparison technique to "minimise the difference between the swapped segments", and thus minimise the loss of meaning in offspring. Unfortunately, the method seems somewhat inefficient and slow.

Goldberg's Messy GAs (Deb & Goldberg, 1991) use a common technique of labelling every allele (value of a gene). Traditional GAs use the position or locus of an allele in its chromosome to define which gene it belongs to, e.g. the fifth allele might correspond to the gene for cost, the sixth to the gene for material. When every allele is labelled, a chromosome becomes more like a set, with values in any order, yet still 'knowing' which gene they correspond to. Messy GAs use very simple 'cut' and 'splice' operators in the place of crossover, allowing an offspring to inherit random alleles from each parent. This inevitably allows individuals to have duplicate alleles or missing alleles. An arbitrary rule determines which duplicated allele should be used as the actual value for an overspecified gene (e.g. always pick the first). If there is no allele for a gene (i.e. the gene is underspecified), a 'competitive template' (similar to a look-up table) is used to set the value.

Radcliffe also labels each allele, calling this an allelic representation (where every allele is a <gene, value> pair), then directly treats chromosomes as sets. He has created a number of different versions of crossover based on set operations, e.g. random assorting recombination (RAR), random transmitting recombination (RTR) and random respectful recombination (R^3) (Radcliffe & George, 1993).

While it is certainly advantageous to label every allele, as it overcomes problems of the locus of alleles being disrupted during crossover, the number of labels required can be wasteful. To return to the previous example, if the chromosome for a car design defined by a number of building blocks was arranged in a simple genetic hierarchy of two levels, e.g. multiple 'blocks' of multiple genes, then every allele requires one label to define which gene it belongs to, and another to define which block its gene belongs to. This is plainly nonsensical, for the very purpose of a hierarchical organisation is avoid such duplication. If a form of crossover existed which could take into account such hierarchies, then this unwanted repetition of labels would be avoided. Hierarchical crossover is such a beast, capable of performing crossover on chromosomes of variable length whether they are stored explicitly in hierarchies, or even just have a hierarchy of meaning - a semantic hierarchy.

2. Semantic Hierarchy of Genotypes

The *semantic hierarchy* of a genotype is simply a 'tree of meaning' (i.e. a compositional or 'part_of' hierarchy). This hierarchy defines the semantics of a genotype, not the syntax, i.e. the semantic hierarchy is independent of how chromosomes are actually stored in memory.

Upon consideration, it becomes clear that all chromosomes have a semantic hierarchy of some form. For example, the traditional problem with *m* genes and *n* bits per gene, forms a hierarchy with two levels of meaning, see fig. 1. The problem of *l* groups of *m* genes with *n* bits per gene forms a semantic hierarchy with three levels of meaning, see fig. 2. What is perhaps unusual as shown in these hierarchies is the fact that the separate bits are considered as alleles, rather than collections of *n* bits being considered as alleles. However, this is done for a specific reason: by explicitly identifying the separate bits, every part of the chromosome can be made variable. In other words, hierarchical crossover allows GAs to evolve not only the value of each bit, but also the number of bits per gene (and hence the precision), the number of genes in the problem, the number of blocks of genes (should the problem have a three or more level hierarchy), and so on. It should be noted that the hierarchy is of *meaning* only, i.e. the alleles shown in figures 1 & 2 are not part of the hierarchy, they are having their meaning defined by the hierarchy. Moreover, this meaning is independent of the order in which the alleles are actually stored in memory (as shown by the out-of-order nodes in the figures).

Any chromosome, no matter how it is stored in memory, has a semantic hierarchy and hence can be handled by a GA using hierarchical crossover. Although it is not necessary to store genotypes hierarchically, by doing so, substantial memory will be saved (especially with every bit requiring an identifier, in addition to every block of

bits and every block of block of bits, and so on). Moreover, hierarchical crossover is designed to take advantage of hierarchically stored chromosomes, so the speed of the operation becomes very much more efficient (i.e. as fast or faster than standard crossover).
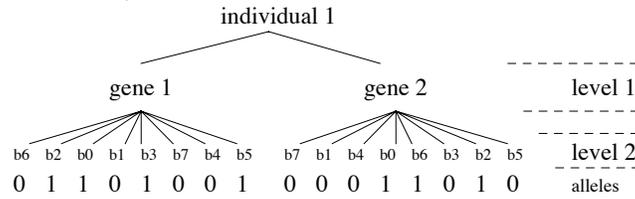
individual 1

gene 1   gene 2   - - - - - - - - -   level 1
                                                                          - - - - - - - - -
b6 b2 b0 b1 b3 b7 b4 b5   b7 b1 b4 b0 b6 b3 b2 b5   - - - - - - - - -   level 2
0 1 1 0 1 0 0 1   0 0 0 1 1 0 1 0   alleles

Fig.1 Semantic hierarchy of a 16 bit, 2 gene chromosome

individual 1

group 1   group 2   - - - - - - - - - - -   level 1
                                                                                        - - - - - - - - - - -
gene 1   gene 2   gene 2   gene 1   - - - - - - - - - -   level 2
                                                                                   - - - - - - - -
b6 b2 b0 b1 b3 b7 b4 b5 b5 b0 b4 b7 b6 b1 b2 b3 b6 b5 b4 b3 b0 b2 b1 b7 b1 b7 b0 b2 b4 b3 b5 b6   - - - - - - - - level 3
0 1 1 0 1 0 0 1   0 0 0 1 1 0 1 0   1 1 0 1 0 0 0 1   0 0 0 1 1 0 1 0   alleles
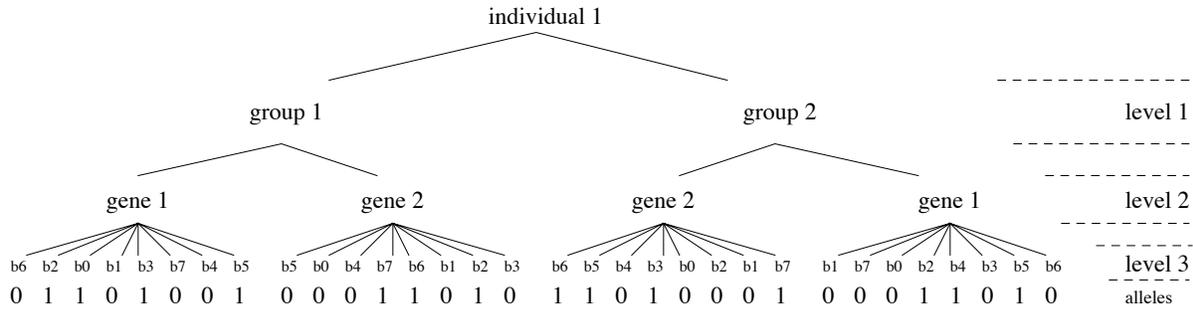
Fig. 2 Semantic hierarchy of a 32 bit, 4 gene, 2 group chromosome

Finally, one more point of interest: the previous explanation has described a problem with m genes and n bits per gene as having a semantic hierarchy with two levels of meaning. There are, in reality, two levels of meaning being ignored by such a statement: the level at which individual solutions in a population are situated, and, as is the case for many GAs, the population level. In other words, a GA could be said to operate on a semantic hierarchy of j populations of k solutions, each solution having, say l groups of m genes defined by n bits. However, since hierarchical (and all other types of) crossover always operates at the level of individual solutions, these higher levels of meaning can safely be ignored in this paper.

3.   Hierarchical Crossover

Hierarchical crossover is based upon the same principles as the normal single-point crossover outlined in the introduction to this paper. It consists of a two-stage process: first, find a suitable crossover point within the two parents, and second, perform the crossover to generate two children. Of course, for the first stage, standard crossover simply picks a random position. However, when dealing with two chromosomes with potentially different numbers of bits in genes, or genes in a block or even different numbers of blocks of genes, finding a suitable crossover point is more of a challenge. This problem is overcome by using the concept of a semantic hierarchy to define levels of meaning for the chromosome, allowing hierarchical crossover very quickly to traverse both chromosomes in order to locate points of similarity.

individual 1   individual 2

gene 1   **gene 2**   **gene 2**   gene 1

b6 b2 b0 b1 b3 b7 b4 b5 b7 b1 b4 b0 b6 b3 **b2** b5 b0 b1 **b2** b7 b6 b5 b3 b4 b3 b0 b5 b1 b2 b4 b7 b6
0 1 1 0 1 0 0 1   0 1 0 0 1 0 1 0   1 0 0 1 1 0 1 1   1 0 0 0 1 0 0 1
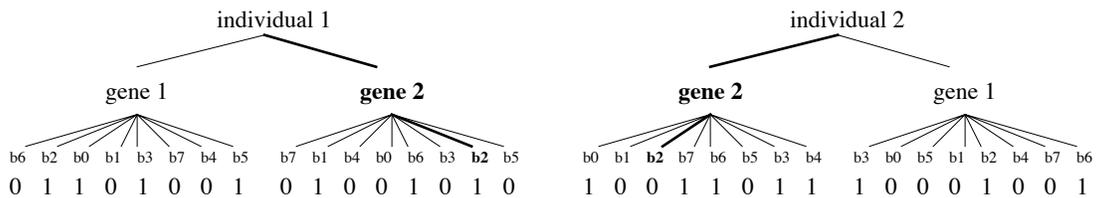
Fig. 3 Locating points of similarity (POS) between two parents (shown in bold).

Briefly, starting at the top level of the semantic hierarchy for each chromosome, a random node is picked from individual 1, and a corresponding node is selected from individual 2. The algorithm then traverses down these nodes, and two more nodes that correspond with each other are picked in the same way, and so on, until the leaves of the trees are reached. If at any stage, there is no corresponding node in individual 2, an alternative is randomly picked in individual 1 and a new corresponding node is searched for in individual 2. If there is still no success when all alternatives at that level have been considered, the algorithm backtracks up a level, picking

a new alternative node in individual 1 at the higher level. If the algorithm backtracks right to the root, then there are no points of similarity between the two chromosomes and the crossover operation is aborted. Figure 3 shows the results of this process, with the points of similarity (POS) between the two individuals being randomly selected as: <gene 2, bit 2>.

Once the POS (or hierarchical crossover point) has been established, the actual crossover process can begin. Again, the algorithm starts at the root of each individual. The top-level node of meaning is then picked from individual 1, either in a pre-defined order, or randomly (see the next section). The corresponding node is found in individual 2 (if it exists). If these nodes are *not* the same as any point of similarity found previously, they are copied (or simply moved) from both parents to the offspring. Exactly which offspring should get which parent's node will be discussed later. Note that these 'nodes' are really abstract meanings encompassing anything from one bit to blocks of genes of many bits. Hence, when a node is copied from a parent to a child, in reality, all alleles that are defined by the node are copied. In other words, if a block node was being copied from a parent to a child, then all genes in that block and all bits in those genes would be copied together.

To digress briefly, this is the feature of hierarchical crossover that speeds up the process, for if chromosomes are internally stored in the same hierarchy as their semantic hierarchy, a parent's node can be literally *moved* to the child. In this way the majority of the crossover process can be achieved by simply changing one or two pointer values, and rather attractively, children become literally composed of their parents genes. This in itself is a highly useful property, as it means that memory need not be continuously allocated and destroyed - the two children completely re-use the memory taken up by their parents in the computer.

Returning to the algorithm again, the copying process continues until all the parents' nodes at the current level have been copied or moved, except the node listed as a point of similarity. Figure 4 shows the slightly denuded parents from fig. 3 and the partially formed children at this point, with gene 2 being the current POS (assuming that nodes are being moved rather than copied, from parents to children).
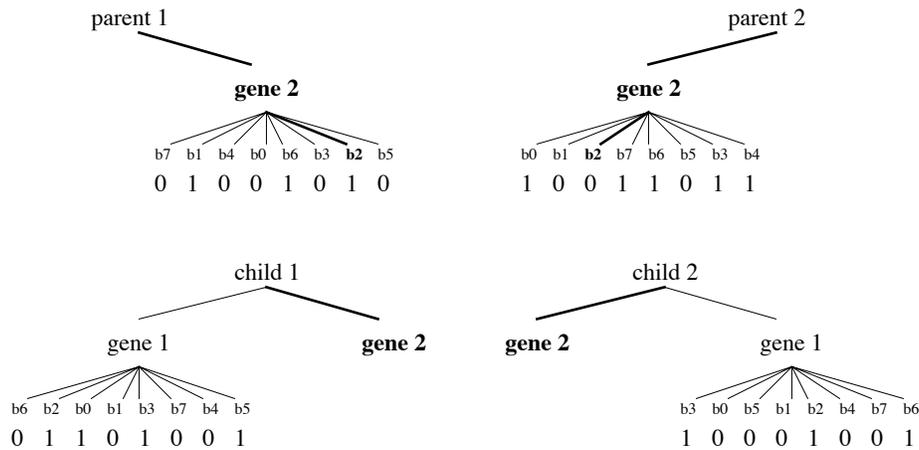
Fig. 4 Partially formed children, halfway through hierarchical crossover.

The algorithm then traverses down both points of similarity in both parents, and repeats the same copying/moving process with the nodes at this level, again omitting the POS (bit 2 in figure 4). The algorithm traverses down again, or if it has reached a leaf, as in the case in Fig. 4, the last remaining allele of each parent is randomly copied to a child. Figure 5 shows how the two children look after the completion of hierarchical crossover.

Fig. 5 Children produced by hierarchical crossover (from parents shown in fig. 3)

Upon careful examination of figure 5 (taking account of the fact that the bits in each gene are stored out of order in this example), it should be clear that hierarchical crossover has behaved exactly as normal crossover would (with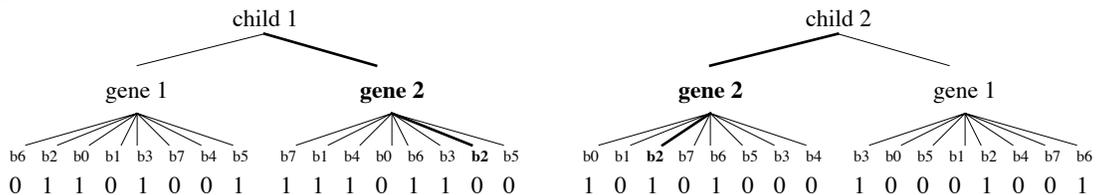 a crossover point of 11) in this case. However, unlike standard crossover, hierarchical crossover can also deal with variable numbers of anything, in each chromosome. The complete algorithm is given in the appendix.

4.   Discussion

As shown above, hierarchical crossover is capable of acting in an identical manner to single-point crossover. This is achieved by maintaining an arbitrary order for each node in the semantic hierarchy, so that when nodes are being copied from parents to children, if the current node is less than the current POS, parent 1's node is copied to child 1 and parent 2's node to child 2. If the node is greater than the current POS, parent 1's node is copied to child 2 and parent 2's node to child 1. Such an ordering is normally implicit in most chromosomes anyway, e.g. gene 1 is less than gene 2 is less than gene 3, bit 0 is less than bit 1, and so on. However, should such an ordering prove difficult, or should single-point crossover be unsuitable, hierarchical crossover will behave as uniform crossover by simply using random choice to determine which child receives a node from which parent.

So far, only an example of crossover between parents with chromosomes of identical lengths has been given. Chromosomes of different numbers of nodes (e.g. genes, bits, etc.) are detected by hierarchical crossover when corresponding nodes are being searched for in the second individual. For example, if individual 1 has three genes: A, B, C and individual 2 has only two: A, C, the algorithm will find no corresponding gene 'B' in individual 2, hence one child will receive gene 'B' of individual 1, and the other child will not receive a gene. Equally, the algorithm checks for the reverse: if individual 1 has two genes: A, B and individual 2 has an extra gene: A, B, X, then one child receives gene 'X', and the other child receives no gene. To illustrate further, consider two parents, one with a two group, four gene semantic hierarchy as shown in fig. 2, the other identical except that it is missing gene 2 in group 2 (the semantics are shown by brackets).

Parent 1:   ((abcdefgh) (ijklmnop)) ((qrstuvwx) (yzabcdef))
Parent 2:   ((ABCDEFGH) (IJKLMNOP)) ((YZABCDEF))

For this example, a random point of similarity (POS) was chosen as: < group 2, parameter 2, bit 1 > (i.e., the allele 'z' in parent 1 and 'Z' in parent 2). If hierarchical crossover is used to emulate standard single-point crossover, the two children would be:

Child 1:   ((abcdefgh) (ijklmnop)) ((qrstuvwx)(yzABCDEF))
Child 2:   ((ABCDEFGH) (IJKLMNOP)) ((YZabcdef))

However, if a random choice was made to determine which node is copied to each child, the uniform crossover would produce children similar to:

Child 1:   ((ABCDEFGH) (IJKLMNOP)) ((qrstuvwx) (YzABcDef))
Child 2:   ((abcdefgh) (ijklmnop)) ((yZabCdEF))

It should be clear that, unlike conventional crossover, this crossover operator has not created any overspecification or underspecification of genes within the children. In other words, as long as the parents are meaningful, hierarchical crossover will always ensure that the children are meaningful. It is also clear that using hierarchical crossover to emulate uniform crossover mixes the parents' genes more effectively.

Finally, a few more general points of note. Many existing crossover methods for variable-length chromosomes have time complexities of $O(nm)$ where n is the length of chromosome 1 and m is the length of chromosome 2 (the two chromosomes being crossed). The algorithm outlined in this paper has a reduced time complexity of:

$$O\left(\sum_{i=1}^{depth} n[i] \times m[i]\right)$$

where 'depth' is the maximum depth of both the semantic hierarchies of the two chromosomes, and $n[i]$ and $m[i]$ are the number of nodes in the hierarchies at depth i for chromosome 1 and chromosome 2, respectively.

Another aspect of hierarchical crossover is that it permits genes to be deliberately underspecified (although care must be taken when allowing the number of bits per gene to be variable - should mutation knock 'holes' in the middle of a number (remember every bit knows its place), this would lead to under-specification of an awkward sort). Moreover, deliberate overspecification of genes can be handled in a similar way to messy GAs, since the algorithm preserves order, letting offspring inherit the relative ordering of duplicate nodes from their parents. Indeed, if duplicate nodes appearing first in a hierarchically organised chromosome were treated as the 'actual' nodes (with mutation having the ability to change the ordering of such nodes), a new type of structured GA is possible, without the need for control genes to switch 'on' or 'off' segments of the chromosome.

5.   Results

To demonstrate the use of hierarchical crossover, it was implemented to allow variable numbers of genes within a generic design system (Bentley & Wakefield, 1996). The system was applied to the problem of evolving the aerodynamic shape of a car (ensuring minimum wind-resistance, whilst providing specified down-forces over the wheels, and satisfying constraints of external and internal dimensions). The system evolves designs from scratch (i.e. from purely random beginnings), using a collection of 'blocks' of variable position and dimension to represent the designs. Each block can also be intersected by a plane to allow the approximation of curves.

Fig. 6 Car design evolved using a GA with hierarchical crossover.

The system uses a GA to evolve the dimensions, positions and numbers of the blocks, using a mutation operator to delete blocks (i.e. delete groups of genes) and another to split existing blocks into two (i.e. add groups of genes). Fig. 6 shows an example of a design which was successfully evolved by the system, using hierarchical crossover to generate all offspring within the GA. Typical results using this new crossover operator were very successful, with the quality of evolved designs often being improved compared to the use of a standard crossover operator.
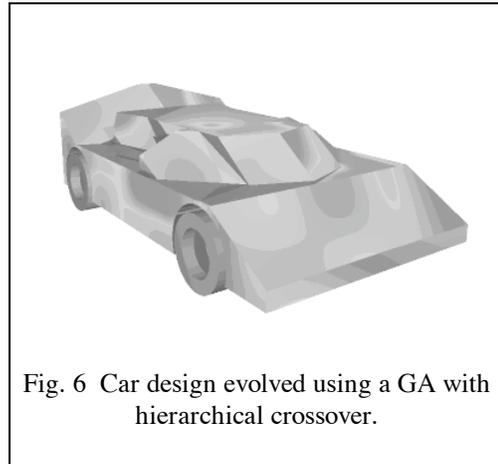
6.   Conclusions

This paper identified the need for a general crossover operator within genetic algorithms that is capable of handling chromosomes of different lengths. The concept of a semantic hierarchy of chromosomes was introduced and explained. This 'tree of meaning' allows the quick identification of points of similarity (POS) within two chromosomes. Once a random POS has been identified, the new crossover operator known as hierarchical crossover (introduced in this paper) can be used to generate meaningful offspring from parent solutions within a GA. This crossover operator is a fast, efficient way of generating offspring, capable of dealing with chromosomes of different sizes (i.e. variable numbers of bits in genes, of genes, blocks of genes, and so on). Hierarchical crossover can cope with underspecification and overspecification of genes, and can behave as a standard single-point or uniform crossover operator. Finally, some improved results obtained from using an implementation of hierarchical crossover within a generic design system were shown.

7.   References

1. Bentley, P. J. & Wakefield, J. P. (1996). Overview of a Generic Evolutionary Design System. In *Proceedings of the 2nd On-line Workshop on Evolutionary Computation* (WEC2), Nagoya University, Japan, (pp. 53-56).
2. Deb, K. & Goldberg, D. E., (1991). mGA in C: A Messy Genetic Algorithm in C. *Illinois Genetic Algorithms Laboratory* (IlliGAL), report no. 91008.
3. Goldberg, D. E., (1989). *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley.
4. Harvey, I., (1992). The SAGE Cross: The Mechanics of Recombination for Species with Variable-length Genotypes. *Parallel Problem Solving from Nature 2*, North-Holland, (pp. 269-278).
5. Holland, J. H., (1975). *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor.
6. Koza, J. R., (1990). Genetic Programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Stanford University.
7. Radcliffe, N. J. & George, F. A. W. (1993). A Study in Set Recombination. In *Proceedings of the fifth international conference on Genetic Algorithms*, Urbana-Champaign, USA. Morgan Kaufmann Pub. (pp. 23-30).
8. Smith, S. F., (1984). Adaptive Learning Systems. In Forsyth, R. (ed.), *Expert Systems: Principles and case studies*. New York: Chapman and Hall, 169-189.

Appendix

HIERARCHICAL CROSSOVER ALGORITHM

STAGE 1
Establish point(s) of similarity (POS) between parent 1 and parent 2:
(i.e. pick a viable hierarchical crossover point for both parents)

start at roots of both parents

```
loop until finished
{       pick at random a child node common to both parents (not previously picked)
        if successful then
        {       store node as a POS
                traverse down to node in both parents
                if node is a leaf of both parents then
                {       finished
                }
                if node is a leaf of only one parent, but not the other then
                {       reject node as a POS
                        traverse back up to parent of node
                }
        }
        else (no nodes in common or all previously picked)
        {       if at root then
                {       parents have no point of similarity and cannot reproduce
                        give up (finished)
                }
                else
                {       reject node as a POS
                        traverse back up to parent of node
                }
        }
}
```

STAGE 2
Perform hierarchical crossover at POS to generate new offspring:
(*Replacing with (node < POS) here will make hierarchical crossover equivalent to uniform single-point crossover, but a purely random choice will mix parents' genes more thoroughly.)
(**Using normal crossover here instead will mix any fixed length genes or blocks of genes.)

start at roots of both parents
find/create new roots for children

```
loop until finished
{       pick from parent 1 the first child node ≠ POS[current depth] not picked before
        if successful then
        {       pick from parent 2 the first corresponding child node not picked before
                if toss_a_coin equals heads* then
                {       attach child node (with all connected sub-nodes) from parent1 to
                          current node of child1
                        attach child node (with all connected sub-nodes) from parent2
                          (if it exists) to current node of child2
                }
                else (if it equals tails)
                {       attach child node (with all connected sub-nodes) from parent1 to
                          current node of child2
                        attach child node (with all connected sub-nodes) from parent2
                          (if it exists) to current node of child1
                }
```

```
            }
    else (parent 1 only has the POS node left)
    {        pick from parent 2 the first child node ≠ POS[current depth] not picked before
             if successful (parent 2 has an extra node) then
             {        if toss_a_coin equals heads* then
                      {        attach child node (with all connected sub-nodes) to current
                                 node of child2
                      }
                      else (if it equals tails)
                      {        attach child node (with all connected sub-nodes) to current
                                 node of child1
                      }
             }
             else (parent 2 also only has the POS node left) then
             {        traverse down to child node in parent1 and parent2
                      if the remaining two nodes are leaves then**
                      {        if toss_a_coin equals heads
                               {        attach leaf from parent1 to current node of child1
                                        attach leaf from parent2 to current node of child2
                               }
                               else (if it equals tails)
                               {        attach leaf from parent1 to current node of child2
                                        attach leaf from parent2 to current node of child1
                               }
                               finished
                      }
                      else
                      {        empty and extract both nodes from parents
                               attach one new empty node to each child
                               traverse down to empty node in child1 and child2
                      }
             }
    }
}
```