The SAGA Cross: The Mechanics of Recombination for Species with Variable-length Genotypes

Inman Harvey

CSRP 223, 1992

Cognitive Science Research Paper

Serial No. CSRP 223

The University of Sussex
School of Cognitive and Computing Sciences
Falmer
Brighton BN1 9QH
England, U.K.

This paper appears in:
R. Männer and B. Manderick, editors,
Parallel Problem Solving from Nature 2, pages 269–278.
North-Holland, 1992

The SAGA Cross: The Mechanics of Recombination for Species with Variable-length Genotypes

Inman Harvey [a]

[a]School of Cognitive and Computing Sciences, University of Sussex, Brighton U.K.
inmanh@cogs.susx.ac.uk

1. Introduction

Genetic Algorithms (GAs) have traditionally tended to use genotypes of a predetermined fixed length. The designer of a particular GA, for use as an optimisation technique within a given search space, decides which parameters are to be represented on the genotype, how they are to be coded, and hence the genotype length. For each parameter there is a given position or set of positions on the genotype which unambiguously code for it. This can be loosely translated as: the allele (parameter or feature value) for a particular gene (parameter or feature) is coded for at a particular locus (genotype position). This makes it simple for a recombination genetic operator, therefore, to take the same crossover point in each parent genotype, and exchange homologous segments.

When variable-length genotypes (VLGs) are used, absolute position of some symbols on the genotype can usually no longer be used to decide what feature those symbols relate to. Some examples of ways around this problem are given in the next section. A related problem is, how can one organise a recombination operator so that the resulting offspring genotypes are, firstly, sensibly interpretable, and secondly, have inherited meaningful 'building blocks' from both parents.

VLG GAs have been proposed in various domains where they seem to allow a natural genetic representation for the problem under consideration, and the variety of domains is reflected in the variety of representations suggested. In this paper the motivation for needing VLGs is that of wanting to extend GAs so as to allow for open-ended evolution. Although GAs have borrowed ideas from natural evolution to use in function optimisation, what they have ignored is perhaps the most impressive feature of natural evolution: how over aeons organisms have evolved from simple organisms to ever more complex ones, with associated increase in genotype lengths. It has been suggested elsewhere that this feature of evolution will need to be used in the only practical way of developing autonomous robots [6, 10], and more generally this is an obvious approach to incremental design by evolution of engineering systems. The SAGA framework was introduced in [7] to incorporate the necessary extensions to standard GAs, and the present paper looks at the consequences for a recombination operator.

It will be suggested that in this context the identification of the locus of a 'gene', or that section of a genotype which codes for some particular feature, will necessarily be by use of an identifying template. The problem for recombination becomes then, given a randomly selected crossover point in one parent genotype, how to identify an appropriate

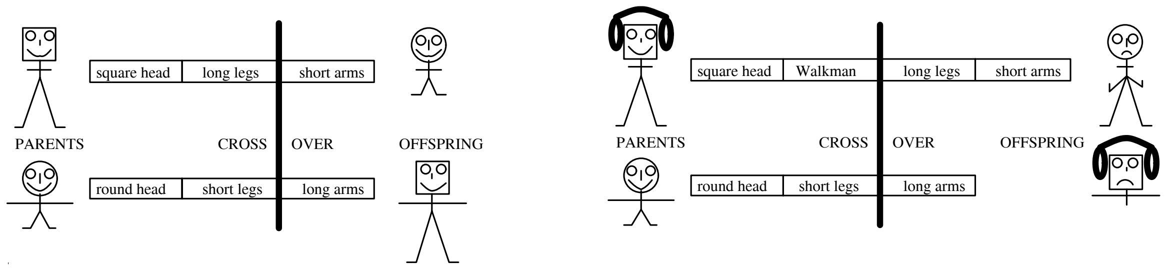


Figure 1. *A crossover operator which works well with fixed lengths may have sad consequences when unthinkingly applied to variable length genotypes.*

place to break the other parent genotype so as to exchange homologous sections as far as is possible. In this we will be aided by the fact that within the SAGA framework the genetic pool of a population will be largely converged to form a *species* or *quasi-species*; as shown in [7] and briefly summarised below.

As a matter of practical concern, therefore, an algorithm needs to be developed which can determine on 'syntactic' grounds rather than 'semantic' ones how to exchange homologous segments. This can be quantified as maximising the similarity (under some appropriate measure) of the segments exchanged. This is of course a problem which nature, at the level of molecular biology, has found its own method of tackling, so an investigation of the relevant literature is suggested. It turns out that molecular biologists have developed algorithms for their own rather different, but related purposes. They are interested in quantifying on 'syntactic' grounds the similarities between two given nucleotide or amino-acid sequences, and doing so with computational efficiency, and it turns out that their algorithms can be adapted and extended for our present purposes. The method of doing so will be here presented; C code for implementing this is available from the author.

2. Examples of Variable-length systems

VLGs have been proposed for a number of purposes, e.g. Smith's LS-1 classifiers [16], Koza's Genetic Programming [12], Goldberg's Messy GAs [2], Harp and Samad's genetic synthesis of neural network architectures [4]. Care needs to be taken that a crossover operation exchanges meaningful building blocks. In the case of LS-1 this is relatively simple, as a genotype is effectively a list of rules each coded as a fixed-length string. The number of such rules is not fixed, and the ordering of them on the genotype has no significance; hence provided that a crossover exchanges homologous sections of an individual rule, the resulting offspring genotype can still be interpreted sensibly. If specific rules needed to be individuated, however, this method would not work.

In Koza's work, the genotype is interpreted as LISP S-expressions, which can be depicted as rooted point-labelled trees with ordered branches. This allows a recombination operator to swap complete sub-trees between parents. The result is syntactically sensible, and preserves and transmits the building blocks that the sub-trees effectively constitute.

This solution relies on the hierarchical tree-decomposition of the genotype, and would not extend to genotype representations where the interactions between 'building blocks' cannot be so decomposed.

In Goldberg's Messy GAs, each locus on the genotype in effect carries its identification tag around with it. Instead of a crossover operator, *cut* and *splice* operators are used, which allow genotypes of any length to develop over time. But the number of loci is fixed at the start, and everything is in effect based on an underlying fixed-length representation, which may be underspecified or over-specified. In the former case, where a genotype does not contain an allele for every locus, the deficiencies are made up by a 'competitive template' scheme. In the latter case, conflicts can arise where the identity tag for a specific locus occurs more than once with different associated values; in this case an arbitrary rule is used, such as choosing the one nearest a specified end of the genotype. It should be noted that the solution to the under-specification problem relies on there being a predetermined number of loci, and cannot be extended to arbitrary numbers of loci.

Harp and Samad use a linear genotype to code for a neural network by having building blocks of a fixed length on the genotype code for the specification of an individual layer in the network, including the connectivity from that layer to other layers. The format for each block is the same, but the number of them is not fixed. A crossover operator can therefore be used which, when a crossover point in one parent genotype occurs inside a block, ensures that the crossover in the other parent genotype occurs in the same position within a block, thus exchanging homologous segments. But unlike Koza's system, the interactions between layers in the network, represented by blocks on the genotype, are not restricted to a hierarchical organisation. So a method must be found for coding within each block so as to identify the other blocks representing network layers projected onto. The solution is adopted of giving each block an ID code, and having two forms of addressing. The first is by absolute address, where the ID (for the block being projected to) is explicitly listed on the genotype; the alternative is by relative addressing, where a relative address is indicated which specifies a block by its position relative to the block being projected from.

The explicit purpose of these alternative forms of addressing is to allow relationships between blocks (and hence projections between layers in the neural network) to develop and be sustained and generalised across generations. Absolute addressing allows a target block to be identified no matter where it finishes up in a genotype in later generations; relative addressing allows groups of blocks close together on the genotype to maintain their mutual interactions.

It can be seen that Harp and Samad's approach avoids the restrictions inherent in Smith's, Koza's and Messy GAs. Nevertheless there remains one restriction which prevents it being satisfactorily used as it is for genotypes of completely unlimited length. The number of bits on the genotype that code for the IDs, and for either absolute or relative addresses being referred to, must be pre-specified. Whereas for instance 4 bits might seem adequate (and 8 bits more than adequate) for genotypes coding for networks with less than 10 layers, for eventually 500 layers or more it would become inadequate. This will of course seem a practically irrelevant restriction to those who know the computational requirements of a network with many layers.

Nevertheless, both from a purist perspective, and from a practical perspective when the building blocks are not layers in a network but some smaller design primitives for a system being constructed, there are reasons for wanting to solve this addressing problem satisfactorily for arbitrary numbers of addresses. In particular this is so in the SAGA framework outlined in a later section.

3. Template Addressing

Harp and Samad's addressing system was limited to addresses of fixed size. The obvious extension is to allow addresses of unlimited length. Assume that any building block on the genotype has an ID string at one end to identify it; and where within another block reference needs to be made to the first ID string (so as to identify a projection or interaction between the two features or modules coded for by the two blocks) this reference is coded for by a reference string. Both ID string and reference string can be of any length, and it no longer is necessary to think of them as ID *numbers*, but instead as strings that need to be somehow matched. In molecular biology such matching of nucleotide strings can be done through the physical interactions between them.

An initial solution would be to use the identical string for both the address marker and the reference marker:

```
ID1 code1 ID2 code2 ref-ID1 ID3 code3 ref-ID1
```

But here the string for ID1 appears three times, and when a call is made to ID1 a simple string search for the string will not know which of the three to find; unless some additional code, or some transformation of the ID, is used to distinguish the two uses. In binary code a simple transformation would be to use the inverse as a template. Ray was probably the first to propose using template matching as a system of addressing, based on molecular biology, in the context of a synthetic evolutionary system [14]. Other equivalent methods can be devised to tackle the problem.

Thus we could implement absolute addressing by means of ID strings, or templates, of any size; indeed some system equivalent to this is necessary. Relative addressing for relative jumps coded by a string of arbitrary length is also easy to implement, but is no longer necessary. A form of template addressing is both necessary and sufficient to have the addressing power of Harp and Samad's system extended to genotypes of arbitrary length.

4. Outline of SAGA

In this section we digress briefly to give the context for desiring open-ended evolution, and hence VLGs.

Some hints from natural evolution have been used by the GA community to produce effective search techniques for complex multi-dimensional search spaces. But this use of GAs for function optimisation is problem-solving in what is, although enormous, a pre-defined space of possibilities of known size — this size being a maximum of a^l when genotypes are of length l with a possible alleles at each position. But the most impressive feature of natural evolution is how over aeons organisms have evolved from simple organisms to ever more complex ones, with associated increase in genotype lengths. This

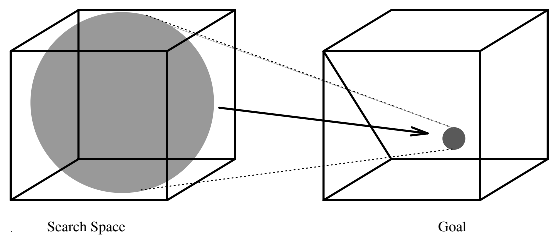


Figure 2. The evolution of a standard GA in a fixed-dimensional search space

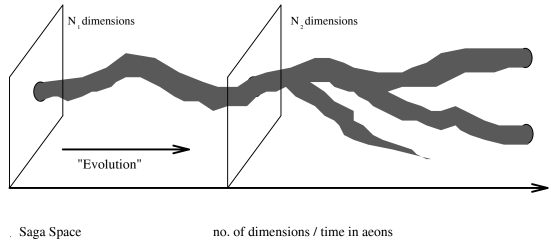


Figure 3. The progress of the always compact course of a species in a SAGA space.

aspect of evolution has been completely ignored in the standard GA literature. GAs have been adapted to problem-solving, and the problem-solving metaphor or frame of mind is, I believe, much of the time inappropriate for considering both natural evolution and potential evolution of control systems for ill-defined domains; such as autonomous robots.

The theoretical underpinning for GAs, Holland's Schema Theorem [9, 3] is no longer valid when the genotypes within a population vary in length. Where an analysis for VLGs has been offered, as in Smith's LS-1 classifiers [16] and Koza's genetic programming [12], the analyses offered have not satisfactorily extended the notion of a schema such that schemata are preserved by the genetic operators [5, 7].

The conceptual framework of SAGA was introduced in 1991 in order to try to understand the dynamics of a GA when genotype lengths are allowed to increase [7]. Working with a finite population, a standard GA often starts with a random distribution that spans the whole search space; the genetic operators, particularly recombination, shift the population over successive generations until hopefully it converges around some optimum (see figure 2). If genotype lengths are going to be allowed to increase indefinitely, then there is no finite search space of pre-determined size, and this picture can no longer be valid. In [7] it is shown, using concepts of epistasis and fitness landscapes drawn from theoretical biology [11], that progress through such a genotype space will only be feasible through relatively gradual increases in genotype length. A general trend towards increase in length turns out to be associated with the evolution of a *species* rather than global search. The word *species* I use to refer to a fit population of relative genotypic homogeneity.[1]

In contrast to the goal-seeking metaphor of figure 2, a journey through SAGA space can be characterised in the form of figure 4. The conclusion of [7], that only gradual increases in genotype length are likely to be viable, means that the finite resources of the population in searching around its current focus should be concentrated on just such gradual increases. The analysis given was supplemented by experimentation using an NK model [11].

In general, the 'problem-solving', or 'goal-seeking' metaphor for evolution is misleading. Within a SAGA space, however, it can still be useful to use this metaphor in the restricted

[1]This is only indirectly related to a biological definition of the word. However it follows from my definition that crosses between members of the same species have a good chance of being another fit member of the same species; whereas crosses between different species will almost certainly be unfit.

sense of searching around the current focus of a species for neighbouring regions which are fitter, or in the case of neutral drift, not less fit. Such a search takes place through application of genetic operators such as crossover, mutation or change-length. The latter two operators are discussed in [6]. In this paper we concentrate on crossover.

5. Where to cross

When evolving systems of arbitrary increasing complexity within the SAGA framework, it will be assumed that there are building blocks coded for along a linear genotype, and that interactions between such building blocks are mediated by some addressing system, as discussed earlier. For recombination it will be relevant that the population will be largely converged; any two parent genotypes will be broadly similar.

The SAGA cross has therefore the requirements that, given any chosen crossover point in one parent genotype, a crossover point in the other parent genotype needs to be chosen so as to minimise the differences between the swapped segments. This can be rephrased as: we should maximise the similarities between the two left segments that are swapped, and between the two right segments that are swapped. Please note that the VLG crossover problem that the SAGA cross handles *only* refers to the choice of the second complementary crossover.

The similarity has to be based on 'syntactic' measures rather than 'semantic', so can only be based on equality of symbols, where the genotype is considered as a string of symbols. The ordering of symbols is also relevant. This leads to the use as a measure of similarity of the longest common subsequence (LCSS).

Algorithms for efficiently computing this have been developed for quantifying similarities between two given nucleotide sequences, starting with the Needleman and Wunsch algorithm [13, 15]; and for the problem of how many editing operations are needed to change one string to another [17]. In the present paper a method will be presented of using the algorithm to solve the VLG crossover problem. The starting place will be Hirschberg's exposition [8].

Hirschberg defines an 'Algorithm B' which accepts as input strings A_{1m} and B_{1n} of lengths m and n; and produces as output vector L_{0n} of length $(n+1)$. L_j will contain the length of the LCSS of string A_{1m} and substring B_{1j}. An array of size $2(n+1)$ is used for intermediate calculations, $K_{01,0n}$.

```
     ALGB(m,n,A,B,L)
1.   Initialisation: $K(1,j) \leftarrow 0$ [$j = 0 \cdots n$];
2.   for $i \leftarrow 1$ to $m$ do
     begin
3.       $K(0,j) \leftarrow K(1,j)$ [$j = 0 \cdots n$];
4.       for $j \leftarrow 1$ to $n$ do
             if $A(i) = B(j)$ then
                 $K(1,j) \leftarrow K(0,j-1) + 1$
             else
                 $K(1,j) \leftarrow \max\{K(1,j-1), K(0,j)\}$;
     end
5.   $L(j) \leftarrow K(1,j)$ [$j = 0 \cdots n$]
```

The rationale behind this algorithm is as follows: The length of the longest common subsequence of two strings A_{1i} and B_{1j} is to be written into $L(i,j)$. If $L(i-1,j-1)$, $L(i,j-1)$ and $L(i-1,j)$ are known, then $L(i,j)$ can be derived from them, the value depending also on whether or not the i^{th} symbol of A and the j^{th} symbol of B match.

$L(i,j)$ must be at least equal to the best of $L(i-1,j)$ and $L(i,j-1)$; and if the symbols do match, then $L(i,j)$ will be one better than $L(i-1,j-1)$. Algorithm B keeps track of the necessary amounts, and updates them within the j and i loops.

In Hirschberg's development, a further algorithm C is used to recursively use Algorithm B, by dividing a given problem into two smaller problems, bottoming out of the recursion when there are trivial subproblems. This is used to output the sequence which is the LCSS of A and B. The purposes of the present paper are rather different, and I have developed an algorithm D to solve the VLG crossover problem.

The initial step is to add a feature to algorithm B so that it will work with substrings, and equally well when comparing two strings enumerated from one end or from the other end. For this it is necessary to explicitly pass as inputs the initial and final indices for the substrings of A and B.

Algorithm D accepts input strings A_{1m} and B_{1n} of lengths m and n, and an integer c which represents the crossover point in A. As output it returns a vector M which keeps track of the current best-so-far candidates for a crossover point in B (which may be one point or a sequence of them). For intermediate calculations two vectors $L1(n+1)$, $L2(n+1)$ are used, which contain the outputs from two separate calls to algorithm B. Internal integer variables r, s and t are used respectively as the current best score, the number currently equal to the best-so-far, and a temporary store.

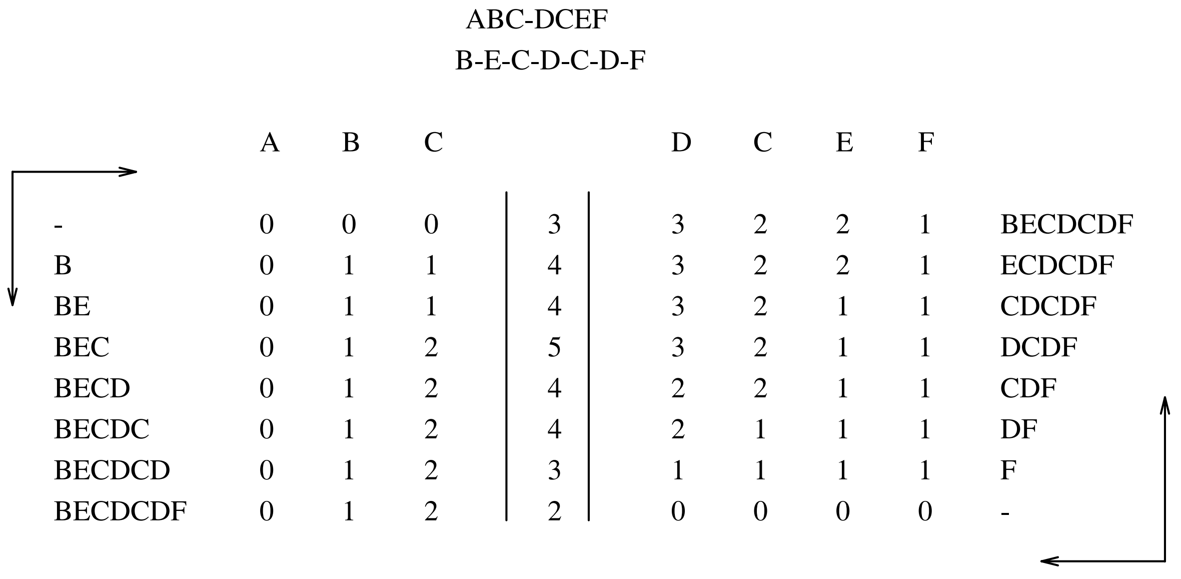
```
        **ALGD(m,n,A,B,c,M)**
    1.  $ALGB(1,c,1,n,A,B,L1)$
    2.  $ALGB(m,c-1,n,1,A,B,L2)$
    3.  $r \leftarrow 0; s \leftarrow 0;$
    4.  **for** $i \leftarrow 1$ **to** $n+1$ **do**
        **begin**
    5.      $t \leftarrow L1(i) + L2(n-i)$
    6.      **if** $t > r$ **then**
                $s \leftarrow 0; r \leftarrow t;$
    7.      **if** $t = r$ **then**
                $M(s) \leftarrow i; s \leftarrow s+1;$
    8.  **end**
```

The rationale behind this algorithm is:

In line 1, algorithm B is applied to the 'left-hand' substring of A, from the start up to the crossover point, and to the whole of string B. The result is output in $L1$.

In line 2, algorithm B is applied to the right-hand substring of A, and to the whole of B, but treating each string in reverse order, starting from the right-hand ends. The result is output in $L2$. Since increasing the length of one of a pair of strings can only either retain or increase the length of the LCSS, both $L1$ and $L2$ have this property.

The loop started in line 4 places, for each possible cross point i in B, the sum of LCSSs for left and right segments into a variable t. As i increases the value of t will each time either increase or remain steady, until it reaches a peak value or a plateau; thereafter t



ABC-DCEF
B-E-C-D-C-D-F

	A	B	C		D	C	E	F	
-	0	0	0	3	3	2	2	1	BECDCDF
B	0	1	1	4	3	2	2	1	ECDCDF
BE	0	1	1	4	3	2	1	1	CDCDF
BEC	0	1	2	5	3	2	1	1	DCDF
BECD	0	1	2	4	2	2	1	1	CDF
BECDC	0	1	2	4	2	1	1	1	DF
BECDCD	0	1	2	3	1	1	1	1	F
BECDCDF	0	1	2	2	0	0	0	0	-

Figure 4. Algorithm D on ABC-DCEF (cross between C and D) and BECDCDF (best cross to be determined). On left, algorithm B gives in column C best scores matching substrings against ABC. On right, working backwards, best scores in column D. Central column shows best total (5 matches) given by splitting BEC-DCDF.

will decrease with occasional level stretches.

The purpose of lines 6 and 7 is to monitor this, and to store in M the values of i for the current best, or several best-equal, values of t. Hence when the loop finishes, the first s values in M contain the proposed crossover positions for maximising t, the sum of left and right LCSSs.

It is then possible to select at random one of the optimal positions, and return this as the proposed crossover point. The C code, available from the author, also economises on memory; rather than using a separate array M, there is enough space to keep track of the optima as we go in array $L1$, since we will never overtake what we are reading from $L1$ with what we are writing into it.

6. Two Point Crossover

Only one-point crossover has been considered here. This has the feature that building blocks near each end of one genotype are much more likely to get separated than ones nearer the middle. For some purposes it may be better to use a two point crossover which avoids this bias. Two crossover points are chosen at random in one parent genotype, and two complementary points need to be selected in the other parent genotype; the offspring are made by swapping the middle sections of each parent.

The present algorithm can be extended to handle this, or indeed multiple-point crossovers, by adding further outer loops.

7. Computational requirements

This algorithm requires for one-point crossover memory space of order $(m + n)$ where m and n are the lengths of the two genotypes. The main loops are in algorithm B, and the time requirements are of order (mn). The time taken is independent of the similarity or otherwise of the two genotypes. The only test on symbols on the genotype is for equality, so whether the genotype uses a binary alphabet or any larger one makes no difference. On a reasonably loaded Sun4 using two genotypes each of length 1000 characters, approximately one second is needed.

In GAs the computational requirements for fitness-evaluation generally far outweigh those for genetic operations, and this could also be expected for any system which needed genotypes of this length.

8. Conclusions

Both biological evolutionary theory, and GA theory, rely on the notion of genes or building blocks being expressed in compact sections on the genotype. For inter-relationships between such building blocks to be incorporated, a method of identification is needed, and this issue comes particularly to the fore when crossover is considered. For fixed length GAs the issue can be solved by identification being implicit in position on the genotype, but not in general when genotype lengths are variable.

Addressing methods for several variable length genotype GAs have been surveyed, and their restrictions noted. Harp and Samad's approach avoids many of these restrictions, but nevertheless does not extend immediately to genotypes of completely arbitrary length. It has been suggested that for present purposes some form of template addressing will be both necessary and sufficient.

A recombination operator needs to be designed that, given any crossover point on one parent genotype, can choose a complementary crossover on the other parent, when the genotypes are of arbitrary, differing, length. The choice must be made on purely 'syntactic' grounds; i.e. through operations solely on the symbols of the genotype, not on their interpretation. Nevertheless the crossover must exchange homologous segments as far as is possible. The fact that in a SAGA system, of gradually increasing complexity and gradually increasing genotype lengths, the population will be largely converged, means that there will be a high degree of similarity between two parent genotypes.

It should be emphasised here that this crossover operator is completely impartial as to where the *initial* choice of a cross on the *first* parent genotype is; it merely then selects where to cross on the *second* parent genotype. The first cross may, for instance, be deliberately chosen for different reasons to be more or less disruptive of schemata [1]. The SAGA cross is only concerned with the complementary crossover; this is a problem which conventional GA practice never has to face, as with fixed-length genotypes the complementary position is trivially obvious.

Building on algorithms developed for the Longest Common Subsequence problem, a novel algorithm has been presented which allows a random crossover point in one parent genotype to be optimally matched by a specified crossover point (if relevant, a restricted range of possible points) in the other parent genotype. The criterion for optimality is well-defined in syntactic terms, being that of maximising the sum of the length of the LCSS in the left-hand segments and the length of the LCSS in the right-hand segments. The algorithm is computationally efficient.

Acknowledgment

This work was supported by a grant from the Science and Engineering Research Council.

References

1 K.A. De Jong and W.M. Spears. An Analysis of the Interacting Roles of Population Size and Crossover in Genetic Algorithms. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature*, pages 38–47. Springer-Verlag, 1991.

2 David E. Goldberg, K. Deb, and B. Korb. An investigation of messy genetic algorithms. Technical Report TCGA-90005, TCGA, The University of Alabama, 1990.

3 David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, Massachusetts, USA, 1989.

4 S.A. Harp and T. Samad. Genetic synthesis of neural network architecture. In L. Davis, editor, *Handbook of Genetic Algorithms*, pages 202–221. Van Nostrand Reinhold, 1992.

5 Inman Harvey. The artificial evolution of behaviour. In J.-A. Meyer and S.W. Wilson, editors, *From Animals to Animats: Proceedings of The First International Conference on Simulation of Adaptive Behavior*, pages 400–408. MIT Press/Bradford Books, Cambridge, MA, 1991.

6 Inman Harvey. Evolutionary robotics and SAGA: the case for hill crawling and tournament selection. Technical Report CSRP 221, COGS, University of Sussex, 1992. Also submitted to Artificial Life III, 1992.

7 Inman Harvey. Species Adaptation Genetic Algorithms: The basis for a continuing SAGA. In *Proceedings of the First European Conference on Artificial Life.* MIT Press/Bradford Books, Cambridge, MA, 1992.

8 D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commmunications of the A.C.M.*, 18(6):341–343, 1975.

9 John Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, USA, 1975.

10 P. Husbands and I. Harvey. Evolution versus design: Controlling autonomous robots. In *Integrating Perception, Planning and Action, Proceedings of 3rd Annual Conference on Artificial Intelligence, Simulation and Planning.* IEEE Press, forthcoming.

11 Stuart Kauffman. Adaptation on rugged fitness landscapes. In Daniel L. Stein, editor, *Lectures in the Sciences of Complexity*, pages 527–618. Addison Wesley: Santa Fe Institute Studies in the Sciences of Complexity, 1989.

12 John R. Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Department of Computer Science, Stanford University, 1990.

13 S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

14 Thomas S. Ray. An approach to the synthesis of life. In J.D. Farmer, C.G. Langton, S. Rasmussen, and C. Taylor, editors, *Artificial Life II.* Addison-Wesley, 1992.

15 David Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Science, USA*, 69(1):4–6, 1972.

16 Stephen F. Smith. *A Learning System based on Genetic Adaptive Algorithms.* PhD thesis, Department of Computer Science, University of Pittsburgh, USA, 1980.

17 R.A. Wagner and M.J. Fischer. The string-to-string correction problem. *Journal of the A.C.M.*, 21(1):168–173, 1974.

Appendix

```
#include <stdio.h>
#include <string.h>
#define MAXLEN 1000  /* Max length of genes */
#define max(a,b) ((a)>(b) ? a : b)

/************************************
A and B contain genes as char strings
as, ae are start and end of substring of A
lenb is the length of B
L is the output vector mentioned in text
************************************/
algb(A,as,ae,B,lenb,L)
char *A,*B;
int *L;
int as,ae,lenb;
{
  int this,last=0;
    /* this/last alternate between 0 and 1 to
       identify different rows in K[][]     */
  int fwd;
    /* flag to identify whether we are running
       forwards or backwards along string A */
  int i,j;
  int K[2][MAXLEN+1];
    /* array described in text              */

  fwd=(ae>as ? 1 : -1);
    /* set fwd flag                         */

  for (j=0;j<lenb+1;j++)
    K[1][j]=0;           /* clear row of K  */

  for (i=as;i*fwd<ae*fwd;i+=fwd)
    /* runs backwards if nec */
    {
      last=1-(this=last); /* flip this/last   */
      for (j=0;j<lenb;j++)
        K[this][j+1]=
          (A[i]==B[(fwd==1 ? j : lenb-j-1)] ?
                  /* are the chars matching ? */
            K[last][j]+1 :          /* yes or */
            max(K[this][j],K[last][j+1]));
                                        /* no */
    }

      /* internal calculations finished;
         copy into output                   */
    for (j=0;j<lenb+1;j++)
      L[j]=K[this][j];
  }
```

```
/**********************************
A and B are char strings for the genes,
of lengths lena and lenb.
cross1 is the selected crossover point in A.
algd will return the proposed position for
crossover point in B
**********************************/
int algd(A,B,lena,lenb,cross1)
char *A,*B;
int lena,lenb,cross1;
{
  int L1[MAXLEN+1],L2[MAXLEN+1];
    /* used by algb                         */
  int best=0;         /* keep track of best */
  int numbest=0; /* and how many equal-best */
  int temp,i;

  algb(A,0,cross1,B,lenb,L1);
    /* left part of A, and all B            */
  algb(A,lena-1,cross1-1,B,lenb,L2);
    /* right part of A, all B, BOTH BKWDS   */

    /* Now go through keeping track of max of
       L1[i]+L2[lenb-i].                    */
  for (i=0;i<=lenb;i++)
  {
    temp=L1[i]+L2[lenb-i];
    if (temp>best)
      {numbest=0; best=temp;}
    if (temp==best)
      L1[numbest++]=i;
  }

    /* Now choose at random from the best   */
  return L1[random() % numbest];
}
```

```
/***********************************
Test program to read in file containing 2
gene strings, choose a random crossover point
in first, and select appropriate crossover
point in second.
**********************************/
main()
{
  FILE *fp;
  char gene1[MAXLEN]; /* strings for genes */
  char gene2[MAXLEN];
  int len1,len2;      /* lengths of genes  */
  int i,j,k,displace1,displace2;
  int cross1,cross2;  /* crossover points  */

  fp=fopen("genefile","r");
  fscanf(fp,"%s",gene1);
  fscanf(fp,"%s",gene2);
  fclose(fp);

  len1=strlen(gene1);
  len2=strlen(gene2);

    /* make sure cross1 is within gene1    */
  cross1=1+(random()%(len1-1));
  cross2=algd(gene1,gene2,len1,len2,cross1);

  printf("\n%d   %d\n",cross1,cross2);
}
```