COMPUTER SIMULATION OF A PRIMITIVE, EVOLVING ECO-SYSTEM

Roger Weinberg
Larry K. Flanigan
Richard A. Laing

Department of Computer and Communication Sciences
The University of Michigan
Ann Arbor, Michigan

Department of Health, Education and Welfare
National Institutes of Health
Bethesda, Maryland
Grant No. GM-12236

National Science Foundation
Washington, D.C.
Grant No. GJ-519

Dr. Roger Weinberg
Department of Statistics and Computer Science
Kansas State University
Manhattan, Kansas 66502

SIMULATED, EVOLVING ECO-SYSTEM

FIGURE LEGENDS

Figure 1: Evolution loop: Flow chart.

Figure 2: Utility: FORTRAN calculation.

Figure 3: Individuals: FORTRAN description.

Figure 4: Cross-over: FORTRAN code for a cross-over between individuals IB1 and IB2. The cross-over products are loaded into individuals IW1 and IW2.

Figure 5: Inversion: FORTRAN code for inverting the chromosome segment between IR1 and IR2.

Figure 6: Mutation: FORTRAN code for incrementing locus IR1 by IR1 x T/100.

ABSTRACT

A computer simulation of a primitive, evolving eco-system has been written. The populations are bacteria, and the environments are three different kinds of bacterial media.

The evolutionary algorithm includes three genetic operators: mutation, crossing-over and inversion. Selection accomplishes evolutionary optimization. The program is effective in obtaining optimization of key parameters in the simulated bacteria. Crossing-over is shown to be significant in obtaining evolutionary optimization.

Forty different, non-interchangeable loci are used. The chromosome contains index numbers to identify the loci. This form of the evolutionary algorithm is quite general. It can be easily applied to problems in artificial intelligence.

COMPUTER SIMULATION OF A PRIMITIVE, EVOLVING ECO-SYSTEM

Introduction

A computer simulation of a living cell has been written (1-7). In this simulated living cell, the metabolic processes of a single bacterial cell were modeled. These metabolic processes could be affected in a realistic fashion by changes in the environmental media in which the cells were placed (1-7). That is, we simulated cells capable of "phenotypic" adaptation. The "genetic" rate constants which controlled the cell metabolic behavior were fixed at the beginning of each simulation experiment and did not undergo change. In this present paper we consider populations of cells, in which, first of all, the rate constants may differ from cell to cell. Thus some of these cells may be able easily to handle "environmental crises" while other cells may handle inputs from their environment so poorly as to be unable to survive. We also introduce the possibility of alteration of rate constants.

By introducing into our simulated living cell a technique for the modifying of the rate constant systems of the cell types, and by permitting the "fittest" cell types to survive (and the least fit cell types to die), we obtain a paradigm of some of the processes of genetic adaptation and evolution.

This paradigm for adaptation and evolution is not limited to studies of bacterial genetics. In fact, this system for simulation of adaptation and evolution can in a broader but still perfectly reasonable sense, be seen as a representative of a large class of biological systems of

interest (8-11). One excellent interpretation of this adaptive simulation is that of a simple eco-system (12).

An eco-system is a complex entity (3). It can, at a single time, contain interacting populations of animals, plants, and protists. These populations interact with their environment, changing the environment and being changed by it.

We would like to be able to study different kinds of ecological interaction whether the system of interest consists of bacteria on a petri dish, or the flora and fauna of a desert floor.

By gaining an overview of the rules which govern all such systems (whether customarily viewed as eco-systems or not) we hope to locate key control points in the system. Key control points are points which are sensitive to outside manipulation. By acting on these key points, we may powerfully affect the course of the interacting eco-systems. Once we pinpoint and understand these key points we may be able to guide a system toward goals (stable states) desirable to humans. This is, of course, an extremely difficult problem. Seemingly minor changes can lead to beneficial or to deleterious changes we did not foresee.

In order to attempt to analyze the behavior of interacting populations, and to come to understand the laws governing their interactions, we will want to begin with a very simple system. This simple system will consist of organisms interacting with each other, and with an environment. It will not contain much else. By employing an extremely simple initial eco-system, we can hope satisfactorily to study it, analyze it and to draw correct conclusions about it. We can then proceed to more complex situations.

We will represent the eco-system in a computer simulation, and will perform simulation experiments with the system. By means of computer simulation, we can vary various factors in the system. We can change the population size, the nature of the individuals in the populations, and the number of different populations.

We can also, in the course of the simulation experiment, manipulate various factors in the environment of our populations. In the course of our simulation experiments we may change these parameters: size, kind and nature of populations, environmental conditions. The changes we introduce in the system may in turn induce changes in others of our parameters. After a parameter change the resulting behavior of the simulated eco-system gives us information concerning the effect of that parameter change on our eco-system.

If a parameter change has a "strong" effect of eco-system, we call it a *key parameter*. Parameter changes which have no effect on the eco-system under study, are *minor parameters*. As is clear from many experimental studies, defining major and minor parameters is an important and demanding task (13). Computer simulation can be a valuable aid to the experimental biologist or ecologist in this respect.

Once we have established our major and minor parameters, we are in a position to control the course of our eco-system. The major parameters are likely candidates for the role of "control knobs" of the system. By manipulation of major parameters, we can guide the course of the eco-system with a minimum of effort.

We take as a concrete expression of our simple paradigm of an eco-system, an interacting population of *Escherichia coli* bacteria.

We chose *E. coli* because much is known of it, we have come to understand much of its internal behavior, and in our single cell simulations already carried out we have a sophisticated working computer simulation of individuals in such a population (1-7).

The environments we will simulate will be environments such as those in which *E. coli* lives in the real world: beef broth, sugar water, enriched sugar water. The interaction between organisms will be represented by following the evolution of our population of organisms over time.

We will permit competition to exist among the bacterial organisms. As the organisms grow in an environment, fit organisms will replace unfit organisms.

We will describe the set of genetic subroutines which has been written. These genetic subroutines supervise the evolution of our populations of bacteria over time. We will show that genetic recombination speeds up evolution in our system. We will also use the evolutionary algorithm in order to obtain a 10^{10} fold increase in the metabolic stability of our simulated living-cell.

<u>Simulation of Evolutionary Processes</u>

DNA: In order to produce genetic adaptation, we will simulate evolution. We will have populations of simulated cells evolving within the computer. A large part of the evolution will occur through operations on simulated DNA.

We will represent DNA as two arrays (tables of entries) in the computer: 1) the ISTR array, 2) the VSTR array (Table 1). We will store indices of various rate constants in the ISTR array. The values for these rate-constants will be stored in the VSTR array.

TABLE 1

ISTR and VSTR arrays for Cell-type Number I

ISTR(I,J) contains the index of the rate constant which is stored in position J, for the Ith subpopulation. Let ISTR(I,8) = 4 for our example. This means that VSTR(L,8), for the Ith sub-population, contains information referring to rate constant K(4)

Let VSTR(L,8) = 93 in our example, and assume that the Lth individual is in the Ith population. This means that K(4) = 93 for the Lth individual.

Sub-populations 1, 2, 3, and 4 contain, respectively, individuals 1-10, 11-20, 21-30, and 31-40.

The rate-constants about which information is stored are rate-constants for separate metabolic processes (Tables 2,3). These metabolic processes appear in the simulated living-cell. They are part of the equations by which the simulated living cell changes its metabolic state over time.

Rate-Constants and DNA: We represent each rate-constant as two numbers. These two numbers are stored in our arrays. The numbers are 1) an index number in the ISTR array and 2) a value in the VSTR array. The index number tells us which of the metabolic rate-constants we are referring to, and the value tells us the value of the rate-constant which is indexed (Table 1).

The genetic map for storing indices and values of rate-constants is separate from the operating computer program which realizes the expression of this information. The index numbers and rate-constants stored in the array contain information for setting parameters in the simulated living-cell. The simulated cell can then function by metabolizing in various environments. In effect, the rate-constant arrays are the genotype, while our simulated living cell is the phenotype.

It turns out that this separation of genotype from phenotype is vital in order to utilize nonlinear interactions among different genes. This nonlinear interaction is utilized in the evolutionary scheme.

The ISTR and VSTR Arrays: We said that we represent genotypic information in two arrays in the simulation; the ISTR and the VSTR arrays. Each array has two dimensions: ISTR(I,J) and VSTR(K,L).

In order to access (for computer manipulation) the information in

TABLE 2

Evolving Parameters

The parameters which evolve are rate constants. One enzyme has 4 rate constants; each rate constant is the activity of the enzyme with a certain number of product attached to that enzyme

Number of Molecules of Product Attached to Jth Enzyme, with Associated Rate Constant			
0 KK(J)	1 KB(J)	2 KBB(J)	3 KBBB(J)

TABLE 3

Products Associated with Enzyme Rate-constants

Each rate-constant is indexed. The indices run from 1 to 10. From the index-number, you can tell which product is produced by the enzyme.

Index Number	Product produced by enzyme
1	nucleosides and nucleotides
2	amino acids
3	ATP
4	cell wall
5	ADP
6	DNA
7	proteins
8	messenger RNA
9	ribosomes
10	transfer RNA

these arrays, specific number values must be given to I, J, K, and L.

1) The K entry (which runs from 1 to 40) determines which of the 40 individual cell-types in the population we wish to refer to.

2) The I entry may be 1, 2, 3, or 4. The I entry tells us which sub-population we are referring to. Sub-populations 1, 2, 3, and 4 refer, respectively, to individuals 1-10, 11-20, 21-30, and 31-40. Since there is one, characteristic linkage map per sub-population, knowing the linkage maps of each of the 4 sub-populations tells us the linkage map of each individual in the simulation.

3) The J entry (which runs from 1 to 40) gives the position on the simulated chromosome of the information for the L entry in VSTR. (This positional information on rate-constant characteristics is required in order to carry out the genetic spatial rearrangement processes of inversion and crossover.)

Thus, ISTR(1,8) = 4 means that K(4) is the *index name* of the rate-constant that individuals in sub-population 1 have in their chromosome position 8, while VSTR(6,8) = 93 means that 93 is the *value* of the rate-constant that individual cell-type number 6 has in its chromosome position 8. Since individual 6 is in sub-population 1, K(4) = 93 for individual 6.

Effective Evolution: The genetic operators we chose can use nonlinear interactions to permit the population to evolve toward a desired end. E.g., we will be selecting for individuals in which a set of three rate-constants, set at certain values, produce a metabolically stable cell. We have a good chance for selecting for this three-rate-constant combination even if each rate-constant alone does

not confer a selective advantage.

The genetic operators we will employ are those which have evolved over millenia of time in real populations. Since selection is intense in the real world, and evolution of systems of characteristics does occur, we have experimental evidence that we can select for systems of desirable characteristics. Furthermore, theoretical analyses of evolution assure us that simple genetic operators are powerful devices for nonlinear optimization. Our scheme is highly specific, and has a strong chance of success from the beginning.

Genetic Operators: We will use the genetic operators found in nature. The physical operation, carried out on a real chromosome by each real genetic operator, will be simulated by a programming operation employing simulated genetic operators acting upon the simulated chromosome. The three genetic operators we will simulate are 1) crossing-over, 2) inversion, and 3) mutation.

1) *Crossing-over* in the real cell is the interchange of segments between two chromosomes. Crossing-over in the simulated cell is the interchange of numbers contained in segments of two simulated chromosomes. Since a simulated chromosome is an array, simulated crossing-over is the interchange of numbers in segments of two arrays.

Crossing-over permits preferential increase in the numbers of groups of subroutines which interact well. This is true for real and simulated populations (12,14).

Cross-over sets up the concept of genetic distance. Without genetic distance, there could not be close-linked genes versus weakly-linked genes. Crossing-over is basic for the system an evolving population

uses, since it permits it to take advantage of nonlinear interactions among systems of genes.

2) *Inversion* in a real cell is a reversal in sequence of part of the real chromosome. Inversion in a simulated cell is a reversal in the contents of part of an array. The array used for the simulated inversion is the array which represents DNA. Inversion is necessary in order to rearrange the genetic locations of different subroutines. Functionally-related genes may work well together. These genes should remain together during evolution. In order for genes to remain together, they should be closely-linked. Inversion allows genes to become closely-linked.

Inversion changes the locations of genes at random. By changing the locations of genes, inversion permits genes to become closely linked. Inversion also separates closely-linked genes from each other. If a close-linked system of favorably-interacting genes arise, the system will tend to evolve as a group. Evolution of groups of genes which function as a single system is quite common in real populations. The process is called co-adaptation (8,9,10,15,16,17). In order to implement co-adaptation, the genes of the interacting system often become closely-linked on a linkage map. Genes which are closely-linked tend, through the course of evolution, to remain together. If such closely-linked genes work well together, then this confers a strong selective advantage upon the organism possessing the felicitous combination.

3) *Mutation* is the process of changing the value of a gene. In our simulation, a gene may start with a value of 10. We may mutate the gene by adding 5 to its value. After the mutation, the gene has a

value of 15. It has mutated from a value of 10 to a value of 15.

Haploidy in Nature: We simulate haploid populations. There is evidence that such populations can be quite effective in the struggle for survival. Many haploid populations of bacteria exist in nature. Furthermore, haploid bacteria often crowd out their diploid protozoan competitors. Bacteria crowd out protozoa in beef stew left out in the open during warm weather. Indeed, haploid bacteria and algae replace diploid fish in a stream, if food for bacterial and algae growth becomes available in the stream. In rapidly changing environments, primitive, haploid organisms often replace diploid competitors.

Haploid bacteria do not use dominance, and yet they do well in the struggle for survival. We reason that our simulated population, which represents a bacterial population, will also be able to evolve without the use of dominance.

Populations of Strings: Efficiency of programming also motivated our representation of a population of strings. There is only one representation of any particular string in the program. The utility of a string is increased if it is supposed to represent a large number of individuals. That is, instead of storing the same information in a large number of separate strings, to represent each of the separate copies of a string, we merely record the utility of each string-type. From the utility value of a particular string we can calculate the "size" of the population which would possess that string. By allowing each array to represent a different string, we are preserving maximum variability with minimum computation and storage. Since the rate-in-change

of fitness-of-the-population is approximated by the variability, we would like to preserve maximum variability in the population. We should note that linkage increases the rate-of-change of fitness. The contribution of linkage to rate-of-change of fitness is added to the contribution of variance (16).

We have said that our population consists of strings which are all different from each other. We did this to preserve variance, and thereby speed up the increase-in-fitness of our population. Although high variability speeds up the increase-in-fitness, it has a disadvantage. Fit individuals may not become fixed. A population which does very well may not be maintained. We do remedy the lack-of-fixation of fit individuals. Old strings are preserved each generation as members of the next generation (40% of the old population will be saved intact in an example we will use later). Therefore, the population remembers good genes by saving good individuals.

<u>Carrying Out the Simulation</u>

In our simulation of evolving DNA, we will make all nonevolving traits part of the environment. Only the evolving traits will be subject to genetic manipulation. In effect, the simulated cell, and the simulated environment are largely fixed. Only a few of the parameters in the simulated cell evolve. These parameters are present in our evolving strings. They are used to set values in a simulated cell; the simulated cell then grows in three simulated environments. The success of the simulated cell measures the success of the parameter settings. The string is judged by the success of the parameter settings.

The simulation is written in FORTRAN. It takes 30 minutes of computer time per generation. One generation involves the reproduction of all 40 individuals in our population. Most of this computer time is spent in growing the simulated-cell. The genetic sub-routines take on the order of one minute per generation.

Selection: Survival of the fittest strings will govern the evolution of our populations of strings (Figure 1).

Criteria for Selection: We can easily select the best strings in a population. The property of the simulated cells which indicate phenotypic limitations for the cell, and is particularly easy to observe, is a wide disparity between simulated chemical concentrations of cell metabolites and the concentrations necessary for balanced growth. This property is correlated to a program variable, and the program variable is used to calculate the performance of a string. We can see the inability to maintain biochemical equilibria necessary for life in a disparity between the concentrations of biochemicals in the running simulation, and optimal concentrations.

Biochemical Pools and Fitness: The ratios of simulated biochemical pools to optimal biochemical pools should be 1 if metabolic equilibrium is perfectly maintained (5). Departures of the ratios from 1 indicate metabolic instability. The further the ratio departs from 1, the lower should be the value of the utility function. The utility function decreases as the ratio departs from 1 because we have included the term (ratio + 1/ratio) in the denominator of the utility function. Recall that the lower is the utility function, the less is the rate of reproduction of the individual under consideration. Therefore by the previous calculation, we select against metabolic instability.

Programming the utility is done as follows: 1) Utility is set to one at the beginning of each cell's growth. 2) Each time step, utility is decremented according to the formula in Figure 2. As we said, the larger the deviation of the chemical rations of products 1-10 is from 1, the smaller is the utility. The cell is run for 15 minutes in each environment, with a simulated time step of 1 minute. Thus, the utility loop is iterated 45 times in obtaining a utility for an individual.

Once we obtain utilities for all 40 individuals, we select the best 40% of the population as parents. The offspring replace the worst 40% of the population. The middle 20% of the population remain as they were, except for mutational modifications.

We destroyed 40% of the population each generation. Death rates are not this regular in real populations. However, even these primitive genetic procedures do induce a complex quantity, average excess, on each string in the population. To obtain this average excess, we would have to calculate it over the run of the program. It is important to perceive that although this average excess exists, it does not appear as a number in the running genetic-program which effects the evolution of programs in the computer. If one were simulating the theory of evolution rather than attempting the evolution of an effective (here effective means metabolically stable) program, one might want to calculate the average excess of each program, rather than to define it implicitly. Our implicit definition of average excess consists of the genetic procedure we use when we produce new programs from old ones.

Let us briefly consider the probability of replacement of a program in the population. We will see that the amount of utility awarded to a

program influences the probability that the program will be retained by its genetic supervisor. In our example, forty percent of the programs disappear after a run. In order to be a member of the survival population, a newly generated program has to be in the best 60 percent (6 top out of a population of 10). The higher the value of one of the old surviving programs is, the less likely it is that the old-timer will be supplanted by a newcomer in the next reproductive cycle.

In exploring a genetic space with evolutionary procedures, we can never expect to explore the entirety of the genetic space. Indeed, part of the value of the evolutionary search is the exploration of "interesting" regions of the genetic space. The type of simulated cell we start out with in our simulated evolution makes our search space even more interesting. We have used sophisticated molecular interactions in our beginning simulated cell, and we use this as a take-off point for further evolution. We have included negative feedback of metabolic processes at both the DNA and cytoplasmic levels. We have also used positive controls of DNA and cell division. All of these mechanisms enable real cells to survive, and imply that further evolution will take place in a particularly productive subset of possible physiological states. Furthermore, the molecular mechanisms underlying many of these sophisticated relationships are often simple and direct, and easy to program. For example, DNA replication involves only a few basic concepts (1). We have an initiation site at which DNA replication begins. Once DNA starts to replicate, it continues to replicate until the whole DNA molecule has been duplicated. A few simple rules of this type insure a stable transfer of genetic information from one generation to the next. The ease with which one can program realistic life-phenomena makes it easy to test fairly large populations of cells as to fitness in

a particular environment.

We have considered methods for altering the genetic makeup of populations of simulated cells. These alterations will allow us to explore a space consisting of alternate genotypes. The point at which we begin our exploration, as well as the region we will investigate, are very small compared to the complete space of all possible genotypes theoretically available to the simulated cell. The complexity of our simulated cell at the beginning of an evolutionary run implies that we begin by storing information in a complex genotype. In effect, we begin our evolution from an interesting point in our genetic space. We are limited as to how profoundly we may modify this genetic point.

Genetic inertia exists in that genetic changes from a given point in evolution are small compared to the total changes which can occur in a very long period of evolutionary time. A mutation only effects a relatively small change in total genotype. The same may be said for other genetic modifications which occur. Which genetic changes survive is very closely dependent on the structural and functional relationships existing in the cell. Sophisticated evolutionary schemes may well embody this information. We hope that by allowing our genetic programs to evolve, we can obtain a good evolutionary scheme for our simulation. In simulating our evolutionary scheme, we must specify the genetic loci we wish to use, as well as procedures for modifying the values at each locus.

The Operation of the Computer Program

Genetic Loci: We will pick as unfixed variables from which to generate our population of strings 40 control parameters which the cell uses for phenotypic adaptation to changing environments (Tables 2,3). These 40

parameters are control constants which function to calculate allosteric modification of enzyme activity after the enzyme has already been formed. They fall into 4 natural groups: 1) KK(J) = rate constant for the Jth enzyme when that enzyme has no molecules of product attached to it, 2) KB(J) = rate constant for the Jth enzyme when that enzyme has 1 molecule of product attached to it, 3) KBB(J) = rate constant for the Jth enzyme when that enzyme has 2 molecules of product attached to it and 4) KBBB(J) = rate constant for the Jth enzyme when that enzyme has 3 molecules of product attached to it.

The product associated with the Jth enzyme is listed in Table 3. For indices 1-10 respectively, the associated products are 1) nucleosides and nucleotides, 2) amino acids, 3) ATP, 4) cell wall, 5) ADP, 6) DNA, 7) proteins, 8) messenger RNA, 9) ribosomes, and 10) transfer RNA.

The computer details of the total evolutionary system are straightforward. We will keep the simulated-cell program, and all variables, in program common (the fixed storage area shared by many sub-programs.) We will instruct the computer to bring in simulated-environments, a genetic-program, and the 40 string population supervised by the genetic-program.

The program for the simulated cell receives the values of the variables according to the information in a stored string. Each string represents one individual cell. The simulated-cell program will be run, and a utility will be calculated for it according to how well it did at maintaining metabolic stability in fluctuating simulated-environments.

The supervising genetic-program will continue to allow cells in its population to grow for a short time, until it has awarded a utility to each of the 40 strings in its population. The genetic-program will then operate on its 40 string population to form a new 40 string population.

The genetic-program will direct the evolution or its 40 string pop-

ulation (made up of 4 10-string sub-populations) according to the utilities awarded to the members of its population. The genetic-program will use these utilities to select the best members of each sub-population. The genetic-program will utilize its own genetic information to operate on the strings in its population.

The genetic-program stores information concerning genetic manipulation of the evolving strings. This information determines crossing-over, inversion, mutation, and selection.

The actual mechanics of programming are straightforward (Figure 3). The description of the INDIVth string is easily obtained by reading the attributes of the entities into the performing program. This is done for VSTR(INDIV,1) through VSTR(INDIV,40). The program for the simulated cell then has the proper values for its variables. It can, therefore, make a simulated run and receive a utility rating.

Each of the 40 individuals, in turn, grows accqrding to the values read into that individuals VSTR segment. After this time, the generation counter, GCNT, is updated by adding 1 to it. The genetic sub-routines then carry our crossing-over, inversion, mutation, and selection.

Simulated Crossing-Over: Crossing over is very easy to program since all individuals which form crossover pairs have the same linkage map. Inversions do occur, but when an inversion is produced, it is used to generate a population which evolves as a group. The unequal probabilities of crossing-over for different regions of the linkage map are realized by associating a cumulative frequency-distribution with the crossover operator indexed in the genetic program (adaptive or nonadaptive). This probability is an attempt to simulate the inequalities in probability of crossing over for different regions of real chromosomes. Such inequalities are induced by the presence of inversion heterozygotes during crossing

over in real organisms. Such inversion heterozygotes are not simulated because of the complications introduced into the programming procedure for crossing over. Inversions are simulated, however, since they are a powerful permutation operator allowing the evolving populations to experiment with various linkage maps.

The program for exhibiting crossing-over will be discussed. The two individuals with the highest and next highest utilities will be used in our example.

In order to program a cross-over between individuals VSTR(IB1,J) and VSTR(IB2,J), the following sequence of instructions can be used (Figure 4). IB1 and IB2 are variable, and denote individual strings. The cross-over takes place between position IR1 and IR2. The cross-over products are loaded into strings VSTR(IW1,J) and VSTRIW2,J). IW1 and IW2 denote the worst two individuals in the sub-population.

Since the best 4 strings in a sub-population are saved after a round of phenotypic adaptation and evaluation, the foregoing procedure will be executed twice. The first execution obtains the cross-over products of the two best individuals. It loads them into the space occupied by the two worst individuals. The second execution obtains the cross-over products of the third and fourth best individuals. It loads them into the space occupied by the third and fourth from worst individuals.

We will pick the column for crossover by Monte Carlo techniques. We used a random-number interval from 1 to 40. Two calls to IRAND gave us the two points between which a crossover occurred.

Biological crossover-frequency is influenced by many factors. We have already mentioned inversion heterozygotes. Chemical differences

on different parts of real chromosomes also alter crossover-frequency of each chromosome region. There are also complex interactions between different parts along the length of chromosomes undergoing crossing-over. However, the assumption of constant crossover-frequency per unit string-length for simulated strings is a close approximation to the relation between percent recombination per unit physical-length for real chromosomes. The linkage map of the real chromosome approximates a linear-function of percent-recombination for map-distances of less than forty percent. Constant probabilities of crossover per unit-length is therefore a reasonable first approximation.

Simulated Inversion: Inversions are somewhat artificially simulated (Figure 5). For programming simplicity, there is never any sub-population in which different members of the same sub-population have different inversions. This is accomplished as follows.

The best sub-population, as judged by the genetic-program, is selected. An inversion is introduced into a copy of this best sub-population. The inverted copy then replaces the worst sub-population. The process is then repeated, with the inverted version of the second best sub-population replacing the next to worst sub-population. The goodness of a sub-population is judged from its best individual.

To select the actual inversion site, the genetic-program uses a random number interval from 1 to 40. It calls IRAND to obtain the two inversion sites. The genetic-program inverts the chromosome segment between IR1 and IR2 (Figure 5).

After two execution of the inversion sub-routine each of the two worst sub-populations will be replaced with inverted versions of the two

best sub-populations. The point of this complex interaction is the improvement of linkage maps for the purpose of putting genes in favorable positions on the string (recall our discussion of co-adaptation).

Simulated Mutation: The locus to undergo mutation is chosen by the genetic program (Figure 6). The mutational increment is then obtained by using the random-number interval (-200, +200).

Mutation is a string is effected by the genetic-program. An example of the kinds of values used in mutation follows.

The entity to mutate is K(5), which has a current value of 5. The mutational increment is set at +T/100 by calling on the random-number generator, IRAND. IRAND generates a number between -200 and +200. The mutational increment thus may range between -2 x K(5) and +2 x K(5). Since K(5) has a current value of 5, the mutational increment ranges from -200 x 5 to +200 x 5. The next value of K(5) will be in the range 5 + (-2 x 5) to 5 + (2 x 5). This value of K(5) will be obtained by adding the value of the mutational increment (-2 x 5 to +2 x 5) to the current value for K(5) which is 5.

The mutational increment is accomplished by the code VSTR(I,IR1) = VSTR(I,IR1) + (1. + T/100.) (Figure 6).

Generality of the Reproductive Scheme

Since the parameters indexed by the chromosome arrays are not limited to biochemical rate-constants, the computer simulation of an evolving eco-system may be used to explore many different genetic spaces. This generality may be used to realize various heuristic programs. An example is selection of sets of sub-routines useful in pattern recognition (18). The pattern recognition task has been written as a population of computer

TABLE 4

UTILITY AFTER 8 GENERATIONS OF SELECTION

Crossing-over	no Crossing-over
$.732 \cdot 10^{-2}$	$.728 \cdot 10^{-2}$
$.236 \cdot 10^{-1}$	$.202 \cdot 10^{-1}$
$.618 \cdot 10^{-4}$	$.539 \cdot 10^{-4}$
$.719 \cdot 10^{-10}$	$.765 \cdot 10^{-10}$
$.265 \cdot 10^{-2}$	$.236 \cdot 10^{-2}$
$.230 \cdot 10^{-2}$	$.184 \cdot 10^{-2}$
$.346 \cdot 10^{-2}$	$.908 \cdot 10^{-3}$
$.566 \cdot 10^{-10}$	$.565 \cdot 10^{-10}$
$.241 \cdot 10^{-2}$	$.225 \cdot 10^{-2}$
$.595 \cdot 10^{-2}$	$.496 \cdot 10^{-2}$
$.224 \cdot 10^{-1}$	$.181 \cdot 10^{-1}$
$.882 \cdot 10^{-2}$	$.425 \cdot 10^{-2}$

$F = 774$

The probability that the variance due to crossing-over is significant is greater than 99.995%.

programs which evolve over time. Their evolution is a function of how well the programs do the specific task, pattern recognition, assigned to them.

Heuristic programming may have interesting applications. One can obtain programs by heuristic programming which carry out ill-defined algorithms. Tasks with a well-defined goal-and-reward scheme are particularly suitable for evolutionary programs (11,12,15,16).

Results and Discussion

We obtained two basic results: 1) crossing-over improves our evolutionary algorithm and 2) a 10^{10} fold increase in utility is obtainable by using the evolutionary algorithm. These results will be presented. We will then discuss their significance.

Crossing over improves our evolutionary algorithm: We did the following experiment in order to test the effect of crossing-over on our system. We generated 4 sub-populations of strings. We allowed each sub-population to evolve for 9 generations. During this time, crossover, mutation, and selection were all used by the genetic program.

At the end of 9 generations of evolution, we recorded the maximum-utility individual in each of the 4 sub-populations.

We then repeated the previous experiment, but did not use crossing-over during evolution. Thus we obtained the result of evolution without crossing-over.

We repeated our experiments to obtain 12 populations which had evolved with crossing-over, and 12 populations which had evolved without crossing-over (Table 4). Crossing-over improved the final utility obtained

at the 99.995% significance level. We used the F test for two-way classification (19). We can conclude that crossing-over is an essential part of the genetic algorithm. There are situations, such as ours, where crossing-over significantly improves the evolutionary algorithm.

Theoretical work (16,17) indicates that cross-over should lead to more effective evolutionary optimization. Our results support this prediction. Our simulation of a living-cell realistically embodies existing biochemical pathways. Our simulated evolution represents the evolution of genes concerned with these biochemical pathways. Our result is that cross-over speeds up evolution for our system. Our genes represent 40 different control constants interacting in a complex, and realistic way. Therefore, our results are a valid instantiation of the theoretical prediction that cross-over is a necessary and effective part of the evolutionary scheme.

A 10^{10} fold increase in utility is obtainable: The simulated population evolved for 120 generations. The genetic program used all of its genetic operators during this period of evolution. The operators mutation and crossing-over operated during each generation. Inversion operated every 10 generations. Selection operated every generation, as described previously.

The population started out its evolutionary run with a very low utility. The maximum utility of any individual was $.202 \times 10^{-10}$ in the starting population. The final population, after 120 generations of evolution, had a maximum utility of $.932 \times 10^{-1}$. This 10^{10} fold improvement is impressive.

We had the population traverse this long path of improvement in order to test the kind of improvement accessible to the evolutionary algorithm. We were interested in seeing whether the evolutionary algorithm would become stranded on any local optimization peaks. Since our evolving parameters interact in complex, and often unknown ways, the existence of local peaks is strongly probable. The very improvement of evolution by crossing-over argues for nonlinear interaction of the different, evolving parameters (14). The success of the evolutionary-algorithm in producing a high final maximum from an extremely strong low initial value is an argument that the evolutionary-algorithm is often able to avoid being stranded on local optimization peaks.

Discussion of Results: Many computer programs have been written analyzing genetic interaction of a small number of loci (15,16). Genetic simulations of quantitative inheritance have also been written (12,14). Our simulation, on the other hand, studies the complex interaction of 40, quite different parameters. Our use of an index set also allows us to vary genetic order as we run.

In addition to analyzing genetic events, we are using genetics in an optimization technique. We feel that it is important to emphasize the importance of crossing-over in the genetic algorithm, since many evolutionary optimization programs do not use it (11).

The generality with which we wrote our genetic algorithm makes it applicable to problems in pattern recognition, and artificial intelligence in general.

To demonstrate the effectiveness of the full, evolutionary algorithm

we allowed 125 generations of growth, using crossing-over, inversion, mutation and selection. We started our run with a population whose maximum utility was $.202 \times 10^{-10}$. The final population had individuals with a utility of $.932 \times 10^{-1}$. This 10^{10} fold increase in utility is an impressive example of the kind of improvement obtainable from the evolutionary algorithm.

ACKNOWLEDGEMENTS

Professor B. P. Zeigler stimulated us intellectually, supported us emotionally, and in addition helped us with the writing. Professor J. H. Holland gave freely of his own ideas, and of the computer facilities at the Logic of Computers Group at The University of Michigan.

BIBLIOGRAPHY

1. Goodman, E.D.; Weinberg, R.; and Laing, R.A. A Cell Space Embedding of Simulated Living Cells. *1970 Summer Computer Simulation Conference, Sponsored by ACM, IEEE, SHARE, and SCI,* Denver, Colorado, (1970).

2. Weinberg, R. Analytic and Logical Equations in a Computer Simulation of Cell Metabolism and Replication. *Sixth Annual Symposium on Biomathematics and Computer Science in the Life Sciences,* The University of Texas, 102, (1968a).

3. Weinberg, R. Computer Simulation of a Living Cell. *Bacteriological Proceedings,* G114, (1968b).

4. Weinberg, R. and Zeigler, B. P. Computer Simulation of a Living Cell: Multilevel Control Systems. *J. of the Amer. Soc. for Cyber.* (to be published) (1970).

5. Weinberg, R. and Berkus, M. Computer Simulation of Evolving DNA. *Biometrics,* **25**, 447, (1969a).

6. Weinberg, R. and Berkus, M. Computer Simulation of a Living Cell. Technical Report 01252-2-T, Ann Arbor, Michigan: The University of Michigan, (1969b).

7. Zeigler, B. P. and Weinberg, R. System Theoretic Analysis of Models: Computer Simulation of a Living Cell. *J. of Theor. Biol.* (to be published) (1970).

8. Fraser, A. S. The Evolution of Purposive Behavior, in *Purposive Systems.* von Foerster, H. (Ed.) New York: Spartan, 1968.

9. Holland, J. H. Hierarchical Descriptions, Universal Spaces and Adaptive Systems, in *Essays on Cellular Automata.* Burks, A. W. (Ed.) Urbana, Illinois: University of Illinois Press, 1970.

10. Holland, J. H. A New Kind of Turnpike Theorem, *Bulletin of the American Mathematical Society,* *75*, 1311, (1969).

11. Fogel, L. J.; Owen, A. J.; and Walsh, M.F. *Artificial Intelligence Through Simulated Evolution,* New York: Wiley, 1966.

12. Jain, S. K. Simulation of Population Biology Models in the Theory of Evolution. *1970 Summer Computer Simulation Conference, Sponsored by ACM, IEEE, SHARE, and SCI,* Denver, Colorado, (1970).

13. Watt, K. E. F. *Ecology and Resource Management,* New York: McGraw-Hill, 1968.

14. Hedrick, P. W. Selection in Finite Populations. II. The Selection Limit and Rate of Response for a Monte Carlo Simulation Model, *Genetics, 65,* 175, (1970).

15. Lewontin, R. C. Population Genetics, in *Annual Review of Genetics, Volume 1,* Palo Alto: Annual Reviews, Inc., (1967).

16. Crow, J. E. and Kimura, M. *An Introduction to Population Genetics Theory,* New York: Harper and Row, 1970.

17. Wallace, B. *Topics in Population Genetics,* New York: W.W. Norton, 1968.

18. Cavicchio, D. J., Jr. Adaptive Search Using Simulated Evolution. Ph.D. Dissertation, The University of Michigan: Ann Arbor, Michigan, 1970.

19. Mood, A. M. and Graybill, F. A. *Introduction to the Theory of Statistics,* New York: McGraw-Hill, 1963.

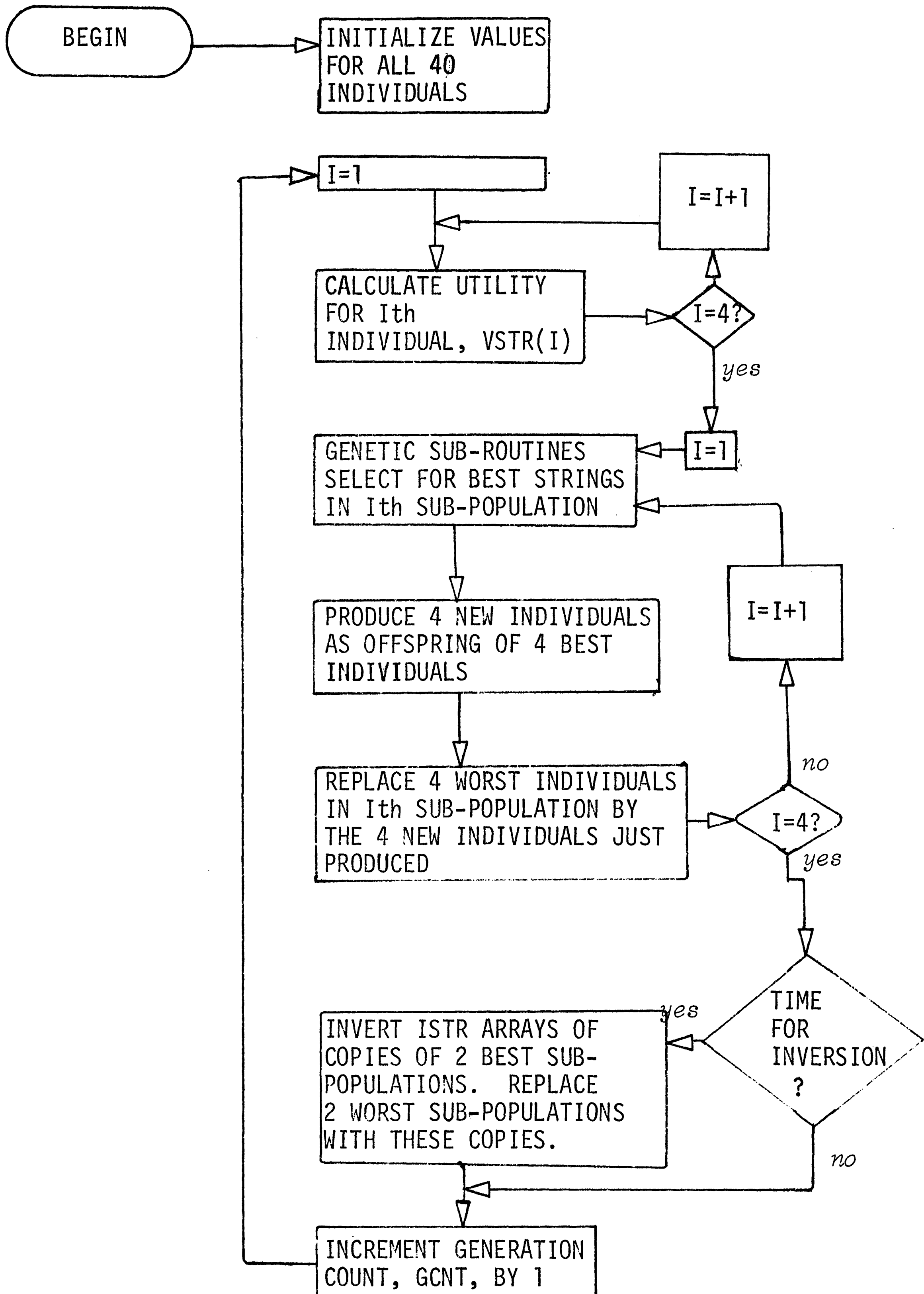

Fig. 1. Evolution loop: Flow chart.

```
      DO 5 I = 1, 10
5     UTIL(INDIV) =(2*UTIL(INDIV) + 1.E-12)/
     1(RATIO(I) + (1/(RATIO(I) + 1.E-12)))
```

Fig. 2. Utility: FORTRAN calculation.

```
      IFILE=1
3     I=IFILE
      READ (1'IFILE) (VSTR(I,J), J=1,40)
      IF (IFILE-41) 3,5,5
5     GCNT=GCNT+1
```

Fig. 3. Individuals: FORTRAN description.

```
C      CROSSOVER CODE
C
       DO 50 K=10,40,10
       DO 50 J=1,2
       I=K+J+J-4
       IB1=IUTIL(I-7)
       IB2=IUTIL(I-6)
       IW1=IUTIL(I-1)
       IW2=IUTIL(I)
       IF (INCNT) 30,30,26
26     CALL IRAND(1,40,IR1,T)
       CALL IRAND(1,40,IR2,T)
       IF (IR1-IR2) 30,50,25
25     I=IR1
       IR1=IR2
       IR2=I
30     DO 35 I=1,40
       VSTR(IW1,I)=VSTR(IB1,I)
35     VSTR(IW2,I)=VSTR(IB2,I)
       IF (INCNT) 50,50,36
36     DO 40 I = IR1,IR2
       T=VSTR(IW1,I)
       VSTR(IW1,I)=VSTR(IW2,I)
40     VSTR(IW2,I)=T
50     CONTINUE
```

Fig. 4. Cross-over: FORTRAN code for a cross-over between individuals IB1 and IB2. The cross-over products are loaded into individuals IW1 and IW2.

```
C      INVERSION CODE
C
110    CALL IRAND(1,40,IR1,T)
       CALL IRAND(1,40,IR2,T)
       IF (IR1-IR2) 120,110,115
115    K=IR1
       IR1=IR2
       IR2=K
120    IF (IR1-1) 125,135,125
125    IC=IR1-1
       DO 130 J=1,IC
       ISTR(I2M,J)=ISTR(I1M,J)
       DO 130 K=1,10
       I1I=I1+K
       I2I=I2+K
130    VSTR(I2I,J)=VSTR(I1I,J)
135    IF (IR2-40) 140,150,140
140    IC=IR2+1
       DO 145 J=IC,40
       ISTR(I2M,J)=ISTR(I1M,J)
       DO 145 K=1,10
       I1I=I1+K
       I2I=I2+K
145    VSTR(I2I,J)=VSTR(I1I,J)
150    IC=IR2
       DO 160 J=IR1,IR2
       ISTR(I2M,J)=ISTR(I1M,IC)
       DO 155 K=1,10
       I1I=I1+K
       I2I=I2+K
155    VSTR(I2I,J)=VSTR(I1I,IC)
160    IC=IC-1
```

Fig. 5. Inversion: FORTRAN code for inverting the chromosome segment between IR1 and IR2.

```
C      MUTATION CODE
C
       DO 60 I=1,40
       CALL IRAND(1,40,IR1,T)
       CALL IRAND(-200,200,IR2,T)
       VSTR(I,IR1)=VSTR(I,IR1)*(1.0+T/100.0)
60     CONTINUE
```

Fig. 6. Mutation: FORTRAN code for incrementing locus IR1 by IR1 x T/100.