**Technological evolution as self-fulfilling prophecy: From genetic algorithms to Darwinian engineering**

By Geoffrey Miller

Published as:
Miller, G. F. (2000). Technological evolution as self-fulfilling prophecy. In J. Ziman (Ed.), Technological innovation as an evolutionary process (pp. 203-215). Cambridge, UK: Cambridge U. Press.

## Introduction

Engineering and computer science are undergoing a Darwinian revolution. In the last ten years, computer scientists have hijacked the idea of "technological evolution", transforming it from a metaphorical model of historical change into literal methods for doing evolutionary engineering using explicit processes of random variation and selective replication inside computers. These methods, including genetic algorithms, genetic programming, and evolution strategies, have attracted exponentially increasing interest as powerful ways of finding good engineering solutions to hard, complex, real-world problems.

Surprisingly, these developments remain almost unknown to scholars interested in evolutionary models for technological progress and to evolutionary epistemologists interested in more general applications of Darwinian theory to human culture and knowledge. Conversely, computer scientists working on genetic algorithms or genetic programming know very little about studies of technological innovation that use explicitly evolutionary models. This mutual ignorance is unfortunate, because there is so much each field can learn from the other. Genetic algorithm research, for example, has developed powerful insights into the way that evolution works as a stochastic search method for exploring design spaces and finding good solutions, and these insights may hold even for technological evolution outside the computer. On the other hand, studies of evolutionary processes in invention, market competition, and historical change reveal a rich, diverse, multi-level interplay between design and selection that may hold valuable lessons for attempts to automate this process. This chapter is a first attempt at match-making between these two fertile fields.

## How computer science turned Darwinian

When computers were slow, expensive, and unreliable, they were no good for simulating evolution. But in the early 1960s, they became fast enough for evolutionary biologists (see Lewontin, 1974) to try solving some of the harder problems in theoretical population genetics using simulation rather than paper-and-pencil proof. Typically, this meant simulating how the allele frequencies of just one or two genes might change over evolutionary time in response to various selection pressures.

Biologists made no attempt to use such evolutionary simulation to actually design something useful in the computer. However, some researchers in the field of Artificial Intelligence realized that the same kinds of computer programs could be used as an engineering method rather than a tool for scientific simulation. In the mid-1960s, Fogel, Owens, and Walsh (1966) suggested that if real biological intelligence evolved through

real evolution, perhaps simulated intelligence could evolve through simulated evolution - that is, by evolutionary programming.

In the late 1960s and early 1970s, John Holland (1975) and his students at University of Michigan developed, a new type of computer program called a genetic algorithm. Populations of bit-strings - strings of zeros and ones - could be randomly generated to form an initial generation, and then each bit-string could be interpreted as a particular design according to some development scheme. Designs could be tested according to a fitness function that determined how good they were for solving a particular problem. The bit-strings that made good designs get to have many copies - "offspring" - in the next generation, while those that make bad designs are eliminated. And so on.

Holland (1975) proposed a general formalism for representing evolutionary processes, showing how they could be implemented inside a computer, and proving various theorems about how those processes will operate as search procedures for finding good solutions. The most important “schema theorem” showed that if bit-strings can recombine their sub-strings (“schemata”) by a method analogous to biological "crossover", then selection on individual bit-strings is sufficient to increase the frequency of good schemata and drive out the bad. This theorem suggested that genetic recombination rather than mutation generated the most important variations that selection acts on during evolution. Accordingly, Holland’s students made the study of recombination a major focus in genetic algorithms research.

Meanwhile, in Germany, engineers such as Rechenberg (1973) and Schwefel (1977) were independently developing a type of Darwinian engineering called “evolutionsstrategie”. For example, they constructed an aerofoil composed of many moveable sub-segments that could be adjusted and immediately tested in a wind tunnel. To optimize the aerodynamics of the evolving wing, they used a carefully controlled mutation and selection process to generate and test better wing shapes. They then generalized this method into a stochastic optimization technique centered around continuous mutation and testing of a single design.

None of these methods made much progress until computers became much faster and cheaper in the 1980s. The problem in all evolutionary simulation using ordinary sequential computers is that the time it takes to produce a single “run” of evolution equals the time required to test each individual design, multiplied by the number of designs in the population, multiplied by the number of generations that the population evolves. With population sizes around 100 to 1000, and around 100 to 1000 generations per run, a typical genetic algorithm run requires somewhere between 10,000 and one million evaluations to produce a decent result. For a complex problem, testing each solution using 1970s computer technology might take several minutes, so that an entire evolutionary run would require about a year of computer time.

Genetic algorithms became viable as research and engineering tools only when computers became fast enough for each fitness evaluation to take only a few seconds. Also, the development of massively parallel computers (such as the revolutionary CM-2 Connection Machine, with over 64,000 processors) cuts out one of the loops, because an entire population can be evaluated at once, with each processor testing a different individual. Once it became possible in the 1980s for risk-averse computer science graduate students to complete hundreds of simulation runs during their Ph.D. program, the field of evolutionary computation flourished.

In 1985 was the first of a successful biannual series of International Conferences on Genetic Algorithms (see Grefenstette, 1985). The first genetic algorithm textbook appeared in the late 1980s (Goldberg, 1989) just in time for this author to take one of the first graduate courses on the subject at Stanford University. This course was given by one of Holland's students, John Koza, who shortly afterwards published a magnum opus (Koza, 1992), detailing how a modification of the genetic algorithm concept could be used to evolve Lisp programs to solve various computational tasks. Instead of representing designs using bit-strings, genetic programming applied selection and recombination directly to the Lisp programs, which can be represented as tree-like structures.

Since 1992, when the key journal, Evolutionary Computation, was founded, the whole field has taken off academically. The half dozen main journals complement the proceedings of nearly 30 periodic conferences, grouped in a number of regular series. The field has also produced three main textbooks (Goldberg 1989; Davis 1991; Mitchell 1996), which are used in the more than 30 graduate courses on evolutionary computation currently taught around the world.

In addition to research on genetic algorithm theory, research has flourished on real-world applications in a number of fields, including industrial design such as aerospace, automobiles, robotics, civil engineering, and factory layout, control systems engineering such as job shop scheduling, neural network design, and system identification, pharmaceutical engineering such as molecular design and protein folding, and financial optimization in spreadsheet programs. Several corporations have recently begun marketing genetic algorithm software packages for corporate and personal use, or consulting about genetic algorithm applications.

**How genetic algorithms work**

Classic genetic algorithms (e.g. Holland, 1975, 1992; Goldberg, 1989) have five key components: a genotype format that specifies how genetic information is represented in a data structure; a development scheme that maps that information into a phenotypic design; a fitness function that assigns a fitness value to each phenotype; a set of genetic operators that modify and replicate the genotypes from one generation to the next; and a set of evolutionary parameters such as population size and mutation rate that govern how evolution runs.

The genotype format specifies the type of data structure that will represent the genetic information. In place of the 4-letter nucleotide alphabet of DNA, genetic algorithms mostly use binary "bit-strings". Typically, these have a fixed length (e.g. 1000 bits), interpreted as a fixed number of "genes" (e.g. 100 genes) each composed of a fixed, equal number of bits (e.g. 10 bits per gene). In the last few years, however, researchers have explored a wider variety of genotype formats, including strings of real numbers, branching tree structures, matrices, directed graphs, and so forth. The initial generation of genotypes is usually produced randomly, for example by assigning a zero or one with equal probability at each point in a bit-string.

The development scheme maps a genotype into a phenotype according to some algorithm or recipe. Thus each gene might be interpreted as a binary number specifying some parameter of a possible engineering solution. For example, 20 genes might be

sufficient to specify (or “parameterise”) a design for a jet engine turbine blade, which could then be selected for its aerodynamic efficiency in a simulation. Alternatively, successive segments of a bit-string might be interpreted as successive rows in a matrix specifying possible connections between the processing units in a neural network (Miller, Todd, & Hegde, 1989). In more recent work on evolving dynamic neural networks capable of controlling robots that pursue or evade one another, we used a much more complex development scheme where some genes specify the spatial locations of neurons in a 2-D “brain”, whilst other genes specify their interconnections11.

The trick in genetic algorithms is to find schemes that do this mapping from a binary bit-string to an engineering design efficiently and elegantly, rather than by brute-force. Good development schemes map from small genotypes into complex, promising phenotypes that already obey fundamental design constraints. Bad schemes require large genotypes and usually produce phenotypic monstrosities. Of course, the smaller the genotype a development scheme can use to specify a set of phenotypes to be searched, the faster evolution can proceed.

The fitness function maps from phenotypes into real numbers that specify their "fitness", and hence the probable number of copies the underlying genotype will be awarded in the next generation. The fitness function is the heart of the genetic algorithm: it is at once the environment to which all designs must adapt, and the grim reaper (or “selective pressure”) that eliminates poorly adapted designs. As with development schemes, fitness functions can range from the trivial to the astoundingly complex. Early exploratory research on genetic algorithms often used literal mathematical functions, such as $y = x^2 - \cos x$ , to map from a real-number phenotype (x) into a fitness score (y). For real applications, however, fitness functions are usually computer simulations of how a phenotype design would perform at some task. In our research on evolving pursuit and evasion strategies (Cliff & Miller, 1996), for example, each neural network pursuer was tested in about a dozen simulated chases around a virtual arena and awarded fitness points for catching different randomly selected opponents as fast as possible. In a civil engineering application, a fitness function might assign points to bridge designs based on their structural integrity, estimated cost, traffic capacity, and resistance to wind-induced oscillations.

If the fitness function does not realistically reflect the real-world constraints and demands that the phenotypic designs will face, the genetic algorithm may deliver a good solution to the wrong problem. But if each fitness evaluation takes too long, a genetic algorithm that relies on millions of evaluations to make evolutionary progress will not be practical. Most difficult in practice is the “multi-objective optimization” problem: giving just the right weight to each design criterion in the fitness function so the evolved designs reflect intelligent trade-offs rather than degenerate maximization of one criterion over all others. For example, giving too much weight to the traffic capacity criterion in a bridge-evaluation program might result in 1000-lane bridges with no structural integrity and exorbitant cost.

In effect, the fitness function must embody not only the engineer’s conscious goals, but also her common sense. This common sense is largely intuitive and unconscious, so is hard to formalize into an explicit fitness function. Since genetic algorithm solutions are only as good as the fitness functions used to evolve them, careful development of appropriate fitness functions embodying all relevant design constraints, trade-offs, and criteria is a key step in evolutionary engineering.

The genetic operators copy and modify the genotypes from one generation to the next. Classic genetic algorithms used just three operators: fitness-proportionate reproduction - genotypes are copied in proportion to the fitness scores that their phenotypes received; point mutation - each bit in a bit-string is flipped from a 1 to a 0 or vice-versa, with some very low probability per generation; and crossover - "offspring" are formed by swapping random genotype segments between two randomly matched "parents". Mutation and crossover thus generate "blind variation", and fitness-proportionate reproduction provides "selective retention" (see Campbell, 1960).

Much genetic algorithm research has focused on making these basic genetic operators work well together, and trying new, quasi-biological genetic operators such as "gene inversion", "duplication", "deletion", and "translocation". Getting the right balance between mutation and selection is especially important. If selection pressures are too strong relative to mutation, genetic algorithms suffer from "premature convergence" on to a genotype that was better than any other in the initial, random generation, but which is far from optimal.

The typical evolutionary problem of getting stuck on a "local fitness peak" can be especially acute with genetic algorithms, where crossover between nearly identical parents does not introduce significant genetic variation, and the vast majority of mutations tend to make even suboptimal designs worse, so get "selected out" almost immediately. Significant genetic diversity can be preserved by speading the population across a simulated geographic area, allowing sub-populations to evolve different solutions and then exchanging innovations via migration and crossover (Cliff & Miller, 1996). Alternatively, if "assortative mating" is favoured, so crossover is programmed to occur more frequently between similar "parents", then the population tends to split apart into divergent "sub-species" with different adaptations (Todd & Miller, 1997).

Finally, the evolutionary parameters determine the general context for evolution and the quantitative details of how the genetic operators work. Classic genetic algorithm parameters include the population size (usually between 30 and 1000 individuals), the number of generations for the evolution to run (usually 100 to 10,000 generations), the mutation rate (usually set to yield around one mutation per genome per generation), the crossover rate (usually set around 0.6, so three-fifths of genotypes are recombined, and two-fifths are replicated intact), and the method of “fitness scaling” (e.g. how differences in fitness scores map onto differences in offspring number).

Deciding the best values for these parameters in a given application remains a black art, driven more by blind intuition and communal tradition than by sound engineering principles. For example, there is a trade-off between population size and generation number: the larger your population, the fewer generations you can run for a given amount of computer time. The genetic algorithm community has no consensus yet about how best to allocate these computer cycles.

**Some strengths and weaknesses of genetic algorithms**

Conjointly, the five components outlined above determine a "design space" (chap ?). Genetic algorithms search these spaces using a massively parallel, stochastic, incremental strategy called "evolution". They are not an engineering panacea. Their performance is only as good as their ability to search a particular design space efficiently

and inventively. This in turn depends critically on a host of subtle interactions between genotype formats, development schemes, genetic operators, fitness functions, and evolutionary parameters. Genetic mutations should tend to produce slight but detectable alterations in phenotypic structure that open the way for cumulative improvement. Genetic crossover should tend to swap functionally integrated parts of phenotypes to yield new emergent properties and behaviors. And so on.

For very simple problems, one can be a bit sloppy about bringing all five components into alignment, because genetic algorithms are rather robust search methods for small design spaces. But for hard problems and very large design spaces, designing a good genetic algorithm is very, very difficult. All the expertise that human engineers would use in confronting a design problem -- their knowledge base, engineering principles, analysis tools, invention heuristics, and common sense -- must be built into the genetic algorithm. Just as there is no general-purpose engineer, there is no general-purpose genetic algorithm.

Most obviously, there is no general-purpose development scheme because different applications require completely incommensurate types of designs. As we saw in chapter X, the design spaces of possible bridges, neural networks, proteins, factory layouts, jet turbines, computer circuits, and corporation financial strategies cannot be translated into a common language, and even if they could be, searching that generic design space would be vastly less efficient that searching a more focused subset.

Genetic algorithms tend to work best when the design space they are searching has already been rather well-characterized or, ideally, fully formalized into a kind of design grammar. For example, genetic programming (Koza, 1992) seems to work well because the design space of computer programs in a particular programming language is clearly structured by that language's formal grammar. Genetic programmers favour languages like Lisp because the "S-expressions" that constitute Lisp programs are branching tree structures that remain interpretable when their end-nodes or sub-trees are mutated or crossed over.

By contrast, there is no design grammar yet for fully re-useable ground-to-orbit spacecraft -- indeed, there remain wildly disparate strategies for solving this difficult problem, each of which require some components that go beyond current technology. In such a case, using a genetic algorithm to generate promising new design solutions would be vastly more difficult than in genetic programming. Still, it might be useful, because by forcing engineers to think about characterizing the design space as a whole rather than perfecting one particular solution, the discipline of setting up the genetic algorithm may yield new insights and ideas.

One of the most disturbing features of genetic algorithms is that they often produce solutions that work, but one cannot quite understand how or why they work. Whereas traditional engineers are constrained to working on designs that they more or less understand, genetic algorithms select only for performance, not for clarity, modularity, or comprehensibility. Like insect nervous systems that have been under intense selection for millions of generations to get the most adaptive behavior out of the smallest, fastest circuits, artificial neural networks evolved for particular tasks (see Miller, Todd, & Hegde, 1989; Husbands, Harvey, Cliff, & Miller, 1994) almost never do so in a way that makes any sense to human minds that expect modular decomposition of function.

Is this a problem? It depends more on the social, cultural, and legal context of engineering in particular domains. The patent office may be sceptical of a design delivered by a genetic algorithm if you cannot explain why it works. A client corporation may reject an optimal marketing strategy designed by a management consultancy using a genetic algorithm if you cannot explain why this strategy makes sense. The automaticity of the genetic algorithm and the opacity of its solutions may create problems of accountability, liability, safety, and public confidence. Also, well-understood solutions can be easily modified and generalized to other problems and other contexts, whereas specifically evolved solutions may not.

On the other hand, animal and plant breeders have been content for thousands of years to use artificially selected products without knowing exactly how they work. Likewise, many traditional technologies such as Japanese sword-making (see Martin, this volume) evolve by trial and error experimentation and cultural imitation, without any theoretical understanding of why the production techniques work. Computer scientist Danny Hillis has commented that he would rather fly on an airplane with an autopilot evolved through a genetic algorithm than one with a human-engineered autopilot. The reason: the evolved autopilot's very existence was at stake every time it confronted the simulated emergencies used to test it and breed it, whereas the human designer's existence was never at stake. For better or worse, genetic algorithms break the link between innovation and analysis that has been considered a fundamental principle of modern engineering.

**Fitness evaluation in Darwinian engineering**

As I noted earlier, the crucial practical issue for real "Darwinian engineering" is whether the process of evaluating candidate designs can be automated. Fitness evaluation is the computational bottleneck in simulated evolution. For example, in our project on pursuit and evasion, it took 95-99% of the computer time (Cliff & Miller, 1996). Applying the genetic operators to breed one generation from the previous generation is usually computationally trivial by comparison. This is because physical reality has a lot of detail that needs simulating, and most serious applications require evaluation in many tests under many different conditions. Consider the computational requirements for evaluating the aerodynamic efficiency and stability of a single jet fighter in simulation under a reasonable sample of different altitudes, speeds, weather conditions, and combat scenarios. Now multiply by the million or so evaluations needed to evolve a decent jet fighter design. Serious Darwinian engineering seems to require prohibitive amounts of computer power.

Is there any alternative to doing all fitness evaluation in simulation? Engineers didn't always test things in computers. Technological progress used to depend on visualizing or sketching design solutions, mentally imaging how they would fare in various tests, and hand-building prototypes for testing in the real world. Before the 20th century, much of engineering and architecture also depended on building scale models and testing them in various experiments. Prototyping and model-making are very efficient ways to let the physics of the real world do much of the evaluation work for you. But how could this sort of real-world testing be incorporated into an automated fitness evaluation method for Darwinian engineering?

A possible future strategy for Darwinian engineering is suggested by a project at the University of Sussex (Husbands, Harvey, Cliff, & Miller, 1994) to use genetic algorithms

to evolve computer vision systems for guiding mobile robots. To evaluate how well each vision system would work, they originally had the robots moving around in a virtual environment, using very time-consuming computer graphic ray-tracing methods to determine what visual input the robot would get at each position in its little world. They decided to let the real world take care of the ray-tracing for them, and put a video camera on to a gantry that could move around in a tabletop model of the test environment. Each robot vision system to be tested was downloaded to a small computer that could translate the robot's simulated movements into gantry movements, with the digital video input then being used directly as the input to the simulated robot vision system. Apart from graduate students needing to untangle the video input cable once in a while, this hybrid between simulated evolution and real-world testing could automatically evaluate a few dozen robot vision systems per day, and led within a few weeks to the evolution of a system capable of navigating towards triangles in preference to circles.

Carried to its logical extreme, this strategy interfaces with combinatorial chemistry, where innumerable variant molecular entities are synthesized automatically by the random combination of segments drawn from different populations, and then screened automatically for the part they play in a particular chemical or biological process. Effective new catalysts, enzymes, therapeutic drugs, electronic materials, etc. can thus be discovered and systematically improved by an evolutionary process that is closely akin to a genetic algorithm.

New methods of computer-controlled manufacturing, robotic assembly, rapid prototyping, and automated lab testing may allow candidate designs to be incarnated and evaluated without human intervention. Automating certain aspects of research and development in this way will be much more challenging than automating the manufacture of standardized products, because candidate designs are worth evaluating only if they are unique. Modifying factories to do automated prototyping, testing, and fitness evaluation will be a major challenge for Darwinian engineering.

Another alternative is to put human judgment into the evaluation loop. Interactive artificial selection can be used to guide evolutionary search through a design space. Thus, computer graphics artists (e.g. Sims, 1991; Todd & Latham, 1992) have applied human aesthetic judgement to evolve fantastic images based on compact "genotypes" combining mathematical formulae and computer-graphic primitives, with the advantage that favoured "phenotypes" can be easily copied and recreated without having to store an entire multi-megabyte image.

Darwinian engineering could extend this sort of artificial selection in two main ways. First, human engineers could use their common sense and expertise to rate candidate designs for their overall plausibility as they are generated by a genetic algorithm on the computer screen in front of them, with computer simulation to test design details. Moreover, the computer could keep track of the human responses to candidate designs, learning how to make its own ratings - for example by training a neural network to emulate human judgment. The outcome might then be an automated evaluation function that combined common sense assessments with sound engineering principles.

A more revolutionary way to put humans in the evaluation loop is to use consumer judgments directly to evolve customized products through interactive, online Darwinian engineering. Modern businesses usually try to make money by second-guessing

average consumer taste, manufacturing a limited range of products to span that taste, and trying to attract mass sales. Genetic algorithms with interactive evaluation may permit a radically different strategy that integrates design, manufacturing, marketing, and sales in a single system.

Suppose individual consumers could log on to a company's "interactive catalogue" directly. Each product line would be a genetic algorithm for evolving a customized product design. It would start with an initial population of candidate designs that could be rated by the consumer, and then mutated and recombined to yield successive generations of new, improved designs. Allow each consumer to evolve their preferred designs, subject only to certain safety, functional, and legal requirements, and which are capable of being manufactured profitably by the automated production system. A few minutes of interactive evolution should lead to a most-preferred design, needing only to be priced and debited to a credit card number before being manufactured and shipped to the consumer. Up-to-data consumer-preference data could be fed into the interactive catalog so as to bias the interactive evolution for each consumer towards areas of design space that have recently proven popular with other consumers with the same demographics and tastes.

This sort of interactive evolutionary consumerism would be most appropriate for fairly low-tech, easily modularized products such as wallpaper, textiles, clothes, jewelry, furniture, toys, holiday packages, and standard financial services. Yet even for relatively high-tech, complex products that must function safely and reliably, like automobiles, cardiac pacemakers, nuclear-powered aircraft carriers, and automated stock-trading systems, where companies normally market only a few carefully-optimized, thoroughly-tested designs, interactive evolution might allow consumers to explore the design space around these designs and observe the fitness trade-offs and constraints for themselves.

In any case, by bringing consumers directly into the design loop as agents of interactive evolutionary selection, the diversity, originality, and richness of human material culture might be substantially increased. One benefit is that it would introduce an analogue of sexual selection. Consumers select products for more than their functional fitness; they also impose various aesthetic and symbolic criteria. These two modes of selection produce quite different evolutionary dynamics and can powerfully complement each other as search and optimization processes (Miller & Todd, 1995). For example, selection for apparently maladaptive "sexual" traits is a very efficient way for populations to escape from local optima in which purely functional selection would otherwise leave them trapped. By allowing consumers to bring their apparently frivolous aesthetic judgments to bear on product development, they may stumble upon promising new areas of design space that more utility-minded engineers may have overlooked.

**The future of technological evolution**

Most historians, psychologists, and engineers who have studied technological evolution in the past agree that the evolutionary process has always been "automated" to some extent, both in the unconscious mental processes of inventors searching design spaces for innovative solutions (see chapter X, Carlson, Perkins), and in the competitive market processes the sift good product designs from bad (Nelson, Stankiewicz, Vincenti). The rise of genetic algorithms as engineering methods adds a third, more explicit, type of automation: the evolution of designs inside computers. Clearly, the utility of genetic algorithms will depend heavily on their ability to complement these other two types of

highly efficient, massively parallel search processes – human creativity and economic markets.

There are good reasons for expecting this complementarity to prosper. Minds and markets are excellent at combining vast amounts of diverse, distributed information under multiple constraints into workable solutions (e.g. ideas that solve the problem or prices that clear the market). Genetic algorithms do something similar, combining vast amounts of information about the fitnesses of different design components (i.e. genes and their phenotypic effects in the simulation) into good designs that are at least locally optimal.

But genetic algorithms work by principles rather different from human creativity and market competition. Human minds are not as good as computer simulations at detailed, quantitatively accurate fitness evaluation. And markets are not nearly as fast, or as free from social, cultural, political, and legal biases. There is a niche then, between minds and markets, for genetic algorithms to contribute to technological evolution. This will probably lead to a division of labour in technological evolution, where problem-solving is often automated using genetic algorithms, but problem-framing still depends on individual and group creativity, and solution-verification still depends on market processes and social history. Engineers will have to think more like selective breeders who design fitness functions, set population parameters, and oversee evolution inside computers and automated design-testing facilities.

Such a development sounds strange, but would simply mark a return to the earliest, most important technological revolution in human history: the domestication and selective breeding of animals by Neolithic pastoralists, followed by the domestication and selective breeding of plants by farmers around five thousand years ago. The early pastoralists and farmers did not know how pre-existing biosystems - wolves, wild cattle, seed-headed grasses, etc. - worked or where they came from; they simply substituted their own selection pressures for those of nature, and reaped the benefits. As the complexity of manufactured technologies begins to approach that of natural biosystems, we may be forced to revert to this humbler form of engineering qua selective breeding. This exceptional millennium, in which humans were able to comprehend their own technologies sufficiently to design them through engineering principles, may be drawing to a close. In future, technological evolution may rejoin the main stream of biological evolution, with humans breeding designs whose operational details lie far beyond their comprehension.

**References**

Campbell, D. T. (1960). Blind variation and selective retention in creative thought as in other knowledge processes. *Psychological Review, 67*, 380-400.

Cliff, D., Husbands, P., Meyer, J.-A., & Wilson, S. W. (Eds.). (1994). From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior. Cambridge, MA: MIT Press.

Cliff, D., & Miller, G.F. (1996.) Co-evolution of pursuit and evasion II: Simulation methods and results. In P. Maes et al. (Eds.), *From animals to animats 4* (SAB96), pp. 608-617. Cambridge, MA: MIT Press.

Davis, L. (1991). *Handbook of genetic algorithms*. Van Nostrand Reinhold.

Fogel, L.J., Owens, A.J., & Walsh, M.J. 1966. *Artificial intelligence through simulated evolution*. New York: John Wiley.

Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, MA: Addison-Wesley.

Grefenstette, J. J.(Ed.). (1985). *Proceedings of an International Conference on Genetic Algorithms and Their Applications*. Hillsdale, NJ: Lawrence Erlbaum.

Holland, J. H. (1975). Adaptation in Natural and Artificial Systems, Ann Arbor, MI: University of Michigan Press. (MIT Press 2nd Ed.: 1992).

Holland, J. H. (1992). Genetic algorithms. Scientific American, 267(1):44-50.

Husbands. P., Harvey, I., Cliff, D., & Miller, G. F. (1994). The use of genetic algorithms for the development of sensorimotor control systems. In P. Gaussier & J. D. Nicoud (Eds.), *Proceedings of the International Workshop from Perception to Action* (PerAc94), pp. 100-121. Los Alamitos, CA: IEEE Computer Society Press.

Koza, J. R. (1992). *Genetic programming: On programming computers by means of natural selection*. Cambridge, MA: MIT Press.

Lewontin, R.C. (1974). *The genetic basis of evolutionary change*. New York: Columbia University Press.

Miller, G. F., & Todd, P. M. (1995). The role of mate choice in biocomputation: Sexual selection as a process of search, optimization, and diversification. In W. Banzaf & F. H. Eeckman (Eds.), *Evolution and biocomputation: Computational models of evolution. Lecture notes in computer science 899*, pp. 169-204. Springer-Verlag.

Miller, G. F., Todd, P. M., & Hegde, S. U. (1989). Designing neural networks using genetic algorithms. In J. D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 379-384. Morgan Kaufmann.

Mitchell, M. (1996). *An introduction to genetic algorithms*. Cambridge, MA: MIT Press.

Rechenberg, I. (1973). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart: Frommann-Holzboog.

Schwefel, H. P. (1977). *Numerische Optimierung von Computermodellen mittels der Evolutionsstrategie.* Basel: Birkhauser.

Sims, K. (1991). Artificial evolution for computer graphics. Computer Graphics, 25(4):319-328, July 1991

Todd, S., & Latham, W. (1992). *Evolutionary art and computers*. San Diego, CA: Academic Press.

Todd, P.M., and Miller, G.F. (1997). Biodiversity through sexual selection. In C.G. Langton and K. Shimohara (Eds.), *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems,* pp. 289-299. MIT Press/Bradford Books.