Introduction to Evolutionary Algorithms

Keith L. Downing
The Norwegian University of Science and Technology
Trondheim, Norway
keithd@idi.ntnu.no

September 14, 2006

1 Introduction

The field of Artificial Intelligence (AI) began in a blaze a glory. Within a decade or so of its inception, computers were solving geometry and physics problems at a college freshman level, playing chess like regional champions, diagnosing serious illnesses on par with expert physicians and designing complex VLSI circuits. No problem was too complex, but, as AI researchers discovered in the 1980's, many were too *simple*.

Indeed, the capabilities that humans take for granted, our basic sensorimotor skills such as walking, climbing, and grasping for objects, turned out to be orders of magnitude more difficult to program than backgammon, bridge and biochemical analysis. By the mid 1980's, AI researchers realized that a serious shortcoming in their systems was none other than commonsense. AI systems behaved like idiot savants, producing exceptional results on a wide range of situations, but floundering miserably on cases that demanded basic intuitions about the world, intuitions that most humans have acquired by their 2nd birthday.

Many attempts were made to force-feed this common sense into AI systems, in much the same manner and using similar knowledge-representation formats as had been successfully used to load expert rules-of-thumb into AI systems. In fact, the whole AI subfield of qualitative reasoning (QR) [8] was dedicated to this aim. Although QR produced many useful paradigms whose applications range from intelligent tutoring to plant monitoring to automobile and Mars-rover diagnosis, it did little to fortify AI systems with that broad base of general knowledge needed to transform idiot-savants into well respected (and trusted) gurus.

As AI was facing this disappointing truth in the 1980's, a related, but diametrically-opposed field began to take root: Artificial Life (ALife) [18]. Although ALife researchers primarily seek to understand the life process at a level far removed from that of neuroscience and psychology, the basic philosophy had immediate implications and inspiration for AI, although only a few AI researchers took note. The essential transferable concepts from ALife to AI are situatedness and embodiment. Although often trivial in their detail, the vast majority of ALife systems consist of simulated

organisms that reside in environments (situatedness) and have a body (embodiment) whose survival depends upon a fruitful interaction with those surroundings.

Classic AI systems, often called GOFAI (Good Old-Fashioned AI) systems, assume away all environmental and bodily factors to focus on cognition in a vacuum. This works well for chess but fails consistently in robotics. As GOFAI researchers found out, general abstract-reasoning systems do not plug-and-play with any set of sensors and motors. General commonsense exists not in a platform-independent piece of software, but in a behavioral repertoire that is finely tuned to the structure and dynamics of both body and environment. And although some aspects of this repertoire are easily explained in everyday terms and rules-of-thumb, many are the unique province of biology and engineering. Logics, so common to GOFAI, are of little utility, but many of ALife's kernel concepts: emergence, competition, cooperation, etc., form the backbone of this low road to understanding intelligence [5].

For that handful of AI researchers [3, 22] who saw ALife as more than Turing-equivalent cellular-automata gliders and evolving biomorphs, but as a more fundamentally sound approach to cognition, the motivating thesis can be approximated as:

> Complex intelligence is better understood and more successfully embodied in artifacts by working up from low-level sensory-motor agents than by working down from abstract cognitive mechanisms of rationality (e.g. logic, means-ends analysis, etc.).

Essentially, Situated and Embodied AI (SEAI) researchers believe that GOFAI's holy grail, common sense, comes only via the learned experiences of a body in a world. There are significant limits to how much knowledge one body (a teacher or an expert-system designer) can transfer to another (a student or an expert system), and with common sense, these limits are very stringent. Whereas "I think, therefore I am" might have been an appropriate slogan for GOFAI, it's converse more aptly summarizes SEAI. That is, by living, we acquire common sense, which then supports more complex reasoning.

Andy Clark [4] uses the term *cognitive incrementalism* to denote this general bootstrapping of intelligence:

> This is the idea that you do indeed get full-blown, human cognition by gradually adding bells and whistles to basic (embodied, embedded) strategies of relating to the present at hand.

The grand challenge to cognitive incrementalism may come from the work of Lakoff and Nunez [17], who explain mathematical reasoning, both simple and complex, as an extension of our sensorimotor understanding of the world. The neuroscientific grounding of their theory is weak, but the metaphorical ties between embedded and embodied action on the one hand and mathematical concepts on the other are striking. By linking everyday sensing and acting to one of man's most abstract cognitive endeavors, the author's implicitly motivate a Turing-type challenge for SEAI: build a sense-and-act robot that evolves the ability to do mathematics.

Ignoring the obvious difficulty of this challenge, the general SEAI philosophy and its bottom-up approach to knowledge acquisition hold some promise. After all, one can hardly deny the importance of first-hand experience in learning simple facts of life such as that wet things can be slippery, make you cold, etc. However, the cruel reality of engineering raises major obstacles, as all roboticists know.

Whereas GOFAI began with a divergent radiation of impressive applications that displayed many forms of (shallow) intelligence, SEAI seems to have converged on a menagerie of wall-following robots, all of which have very deep, functional (albeit implicit) understandings of their own body and domain: a barren floor surrounded by walls. To date, there are no biologically-inspired robots that display transferable common sense. That is, many systems behave intelligently and exhibit minimally cognitive behaviors, such as perceptual classification, attentional focusing, etc. [2], but the common sense is so tightly embedded in reactive routines (or controllers with simple notions of internal state) that it evades reuse for other tasks. So to date, sensing and acting has not produced common sense scaffolding for cognitive activities, such as the **planning** of motor sequences.

Regardless of the rather weak state of the art in SEAI vis-a-vis its ultimate goals, we are convinced that the bottom-up approach to machine intelligence is the correct one. After all, it is the path followed by nature. And since natural evolution took hundreds of millions of years to go from simple multicellular organisms to humans, we cannot expect a synthetic progression to happen overnight, no matter how many computers are crunching away on the task.

This conviction is rooted in our belief that a neural substrate, or some similar network of relatively simple, interconnected processors, is the most effective for the representations and information processing that underlie intelligence.

The commitment to a neural mechanism also seems to entail an evolutionary design process. Essentially, the basic structure of brains is beyond the design capabilities of standard engineering. Brains tend to have modules, but these are very tightly interconnected, with tens or hundreds of thousands of projections between one another. This violates most principles of engineering design, particularly software engineering, which prefer well-encapsulated modules with a limited number of signaling pathways. Although neuroscientists often characterize biological neural networks as box-and-arrow diagrams, they are well aware of a) the extreme complexity of connections both within and between the boxes, and b) the strong contribution of this spaghetti wiring to intelligence.

Although duplicating the complexity of an entire mammalian brain seems far fetched (at least today), the basic pattern of highly interconnected neural modules is feasible on a smaller scale, in artificial neural networks with, say, 10000 neurons, instead of 100 billion. So SEAI can set its sights on large, but not necessarily huge, neural networks and attempt to find useful topologies in this intermediate size class. For such networks, designing useful connection patterns by hand seems ominous. And even if the topology is hand-made, the weights between nodes are nearly impossible to hand code and must either be a) learned from experience using algorithms such as back propagation, or b) discovered by search techniques, where EAs are one of the most popular for the job.

Evolutionary Algorithms can function as either a replacement for or another level of adaptation on top of the ANN learning algorithms. They permit the exploration of a wide design space of

topologies. In addition, ANNs with recurrent connections (i.e., from downstream neurons back to upstream neurons) are so difficult to train with standard learning techniques that many researchers use EAs instead, to evolve proper weight vectors. Real brains exhibit extremely high recurrency, which many neuroscientists believe to be a critical foundation of cognitive processes such as attention and learning, so this is a topological trait that synthetic brains will need to incorporate.

In short, to find proper topologies and weight vectors for complex neural networks, evolutionary algorithms are extremely useful. Again, this may not be a coincidence, since natural evolution takes much of the honor for the amazing complexity of our own brains.

To understand SEAI and to assess its true prospects for achieving high-level intelligence, it is therefore important to have at least passing knowledge of ANNs and EAs. The first few sections of this book are intended to provide just that.

2 Evolutionary Biology: A Very Brief Overview

In 1789, Thomas Malthus wrote his *Essay on the Principle of Population*, in which he recognized that a) population growth rate is a function of population size, and therefore b) left unchecked, a population will grow exponentially. However, since environments have only finite resources, a growing population will eventually reach a point, the *Malthusian crunch*, at which organisms a) will have to compete for those resources, and b) will produce more young than the environment can support.

Crunch time was Darwin's ground-breaking entry point into the discussion:

> As many more individuals of each species are born than can possibly survive; and as, consequently, there is a frequently recurring Struggle for Existence, it follows that any being, if it vary however slightly in any manner profitable to itself, under the complex and sometimes varying conditions of life, will have a better chance of surviving, and thus be naturally selected (*On the Origin of Species by Means of Natural Selection* (1859), pg. 5)

In short, the combination of resource competition and heritable fitness variation leads to evolution by natural selection. When the Malthusian crunch comes, if a) there is any variation in the population that is significant in the sense that some individuals are better equipped for survival and reproduction than others, and b) those essential advantages can be passed on to offspring, then the population as a whole will gradually become better adapted to its environment as more individuals are born with the desirable traits. In short, the population will *evolve*. Of course, this assumes that populations can change much faster than geographic factors.

In addition to a) the pressure to favor certain traits over others (known as *selection pressure*) that the Malthusian crunch creates and b) the heritability of desired traits that enables the features of well-adapted individuals to spread throughout and eventually dominate the population, the concept

of variation is also essential to evolution, not merely as a precondition to population takeover by a dominant set of traits, but as a perpetual process insuring that the population never completely stagnates lest it falls out of step with an environment that, inevitably, does change.

Thus, the three essential ingredients for an evolutionary process are:

1. Selection - some environmental factors must favor certain traits over others.
2. Variation - individuals must consistently arise that are significantly (although not necessarily considerably) different from their ancestors.
3. Heritability - children must, on average, inherit a good many traits from their parents to insure that selected traits survive generational turnover.

These three factors are implicit in the basic evolutionary cycle depicted in Figure 1. Beginning in the lower left, a collection of genetic blueprints (more accurately, recipes for growth) known as genotypes are present in an environment. These might be (fertilized) fish eggs on the bottom of a pond, or the collection of all 1-day-old embryos in the wombs of a gazelle population. Moving upward in the diagram, each genotype directs a developmental process that produces a juvenile organism, a young *phenotype*. At the phenotypic level, traits (such as long legs, coloration patterns, etc.) encoded by genotypes become explicit in the organism.

In Figure 1, selection pressure is present in all processes surrounded by solid-lined pentagons. These represent the metaphorical *sieve of selection* through which populations (of genotypes and phenotypes) must pass. Already during development, this sieve may filter out genotypes that encode fatal growth plans, e.g., those that may lead to miscarriages in mammals. The sieve is relentless, however, so even genotypes that encode plans for healthy juveniles have little guarantee of proliferation; across the top of the diagram, selection pressure persists. Juveniles must survive the torrents and turmoils of life while growing to adulthood, only to enter a new arena of competition, at the highest level, for the right to produce offspring. Thus, by the time organisms reach the upper right-hand corner, the mating phenotypes, the genotypic pool has been narrowed considerably. In many species, individuals need not perish for lack of a mate; some can be recycled and *try again* next mating season, as depicted by the aging/death filter above the adult phenotype collection.

Heading down from the upper right corner of Figure 1, we return to the genotypic level via the production of gametes. This occurs within each organism via mitosis (copying) and meiosis (crossover recombination), resulting in many half-genotypes (in diploid organisms) that normally embody minor variations of the parent's genes. During mating, these half-genotypes pair up to produce a complete genotype, a new plan for development that is normally very similar to but slightly different from that of each parent. As the inverted pentagon indicates, there is normally an overproduction of gametes; only a chosen few become part of the new genotypes. Hence, even the reproduction box is a pentagon/sieve.

Although pinpointing the exact locations of variation and heritability are difficult, since genetic mutations can occur to germ cells (i.e., future gametes) at any time during life, it seems fair to say that the major sources of genetic variation are the (imperfect) copying and recombination processes during mitosis and meiosis, respectively. Inheritance is then located along the bottom path from

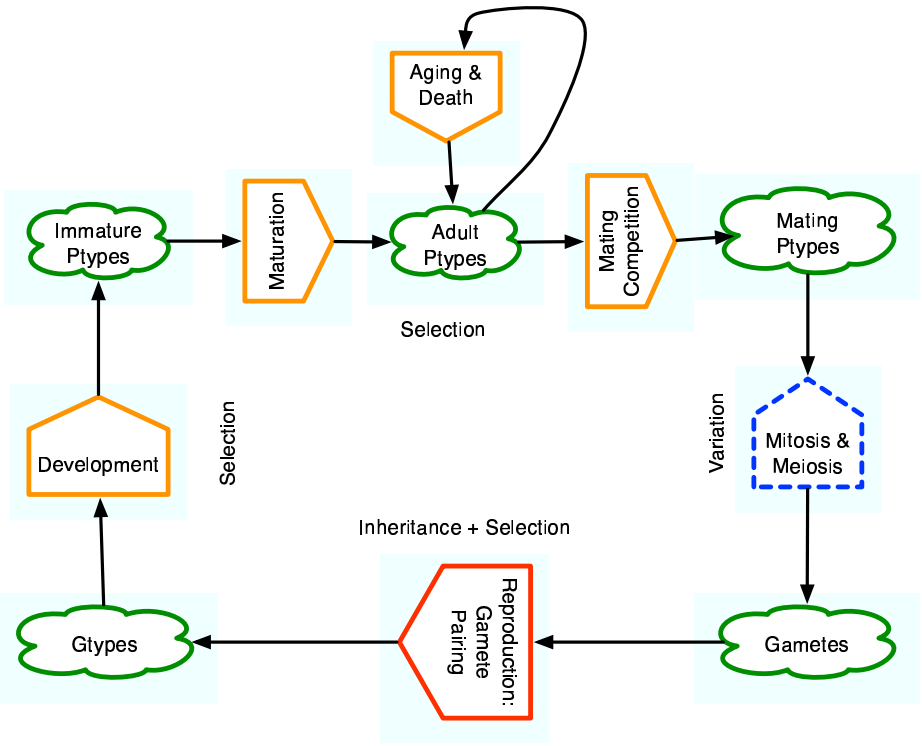

Figure 1: The basic cycle of evolution. Clouds represent populations of genotypes (gtypes) or phenotypes (ptypes), while polygons denote processes. Solid pentagons represent processes that filter the pool of individuals, while the dashed pentagon for mitosis and meiosis indicates an increase (in this case, of genetic material).

gametes to genotypes, since this is the point at which the parents DNA (and the traits it encodes) officially makes it into the next generation.

Via repeated cycling through this loop, a population gradually adapts to a (relatively static) environment. In engineering terms, we can say that the population evolves such that its individuals become better *designs* or better *solutions* to the challenges that the environment poses.

3 Borrowing the Metaphor

The field of Evolutionary Computation (EC) capitalizes on the metaphor of evolution as a problem-solving and design process. By developing Evolutionary Algorithms (EAs) that incorporate bits and pieces of the kernel evolutionary process, EC practitioners solve complicated problems that are often beyond the reach of more traditional, more formal, optimization and design tools. This is not without its price, however, since evolution often finds rather strange solutions/designs that are hard to analyze and explain. But in many cases, it is precisely this *off the board* thinking that is required for creative, ground-breaking progress. EC has its share of such successes, in areas as diverse as antenna design, music, art, movie animation, circuit design, and proteomics.

Figure 2 illustrates the basic cycle of an evolutionary algorithm. As discussed below, different types of EAs include different subsets of the components shown, but all incorporate mechanisms for selection, inheritance and variation.

Beginning in the lower left corner of Figure 2, some EAs use distinct genotypic representations, such as bit strings, that must be translated into phenotypes, which normally represent explicit designs or problem solutions. In other cases, the genotype and phenotype are, for all intents and purposes, the same. The conversion from genotype to phenotype, termed *development* in the diagram, can be everything from a simple copy operation to a detailed growth process. The relationship between genotype and phenotype determines the form of the recombination and mutation operations, as elaborated below.

Once translated into phenotypes, the genotypes are evaluated using a fitness test. These vary with the problem domain, but in general, they quantify each phenotype's success on a particular problem. For example, an EA for solving a travelling-salesman problem would give higher fitness to shorter paths, while one for attempting circuit design might reward more energy efficient solutions with higher values. The most important aspect of the fitness function is that it should give *graded* evaluations such that good, but not necessarily optimal, solutions receive some partial credit. Without it, evolutionary search degrades to a hopeless, random, needle-in-a-haystack hunt.

Each genotype-phenotype pair retains its fitness score for use in all selection/filtering operations that follow. The first such sieve is *adult selection* in which the newly evaluated individuals compete for membership in the adult population. The nature of competition varies such that an individual may win acceptance merely by having a higher fitness value than $k > 0$ other randomly-selected individual (local criteria), or it may require that the individual be among the m most fit (global criteria). Also, many types of competition are preceded by a scaling of the fitness values to either increase or decrease fitness variance, corresponding to raising or lowering the selection pressure,

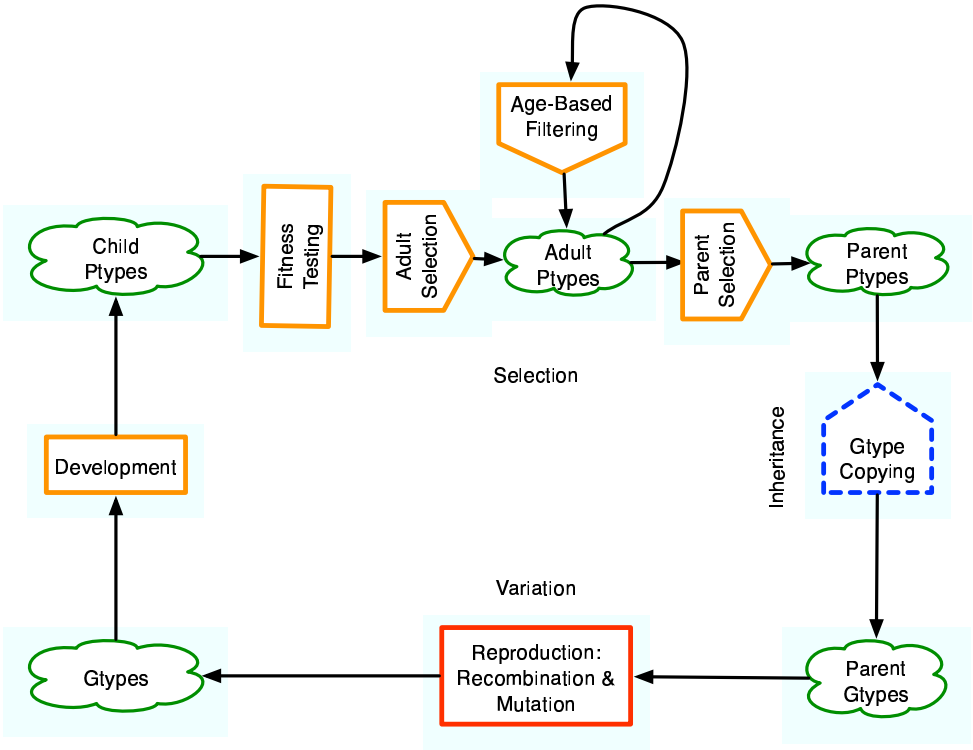

Figure 2: The basic cycle of an Evolutionary Algorithm. Clouds represent populations of genotypes (gtypes) or phenotypes (ptypes), while polygons denote processes. Solid pentagons represent processes that filter the pool of individuals, while the dashed pentagon for *gtype copying* indicates an increase (in this case, of genotypes). Rectangular processes produce no net change in the genotype and phenotype pools.

respectively.

Once the adult pool is updated, a second round of competition begins. Here, adults square off for the right to mate, or in the case of asexual EAs, to simply pass on one or more mutated copies of their genotype. As indicated by the constricting sieve of *Parent Selection* in Figure 2, there are often far fewer than m adults that receive a mating opportunity. Again, the actual protocol for these competitions can be local or global, and the fitness values may be scaled prior to the contest. The terms *parent selection* and *mate selection* will be used interchangeably in this document. The former is more technically correct, since asexual reproduction (which occurs often in EAs) involves no mating, but the latter term is more clearly distinguishable from *adult selection*.

Next, in contrast to biological organisms in which mutation and recombination normally occur prior to gamete union - essentially, children recombine their parents genes in preparation for producing their parents grandchildren - EAs typically select two parents, copy their genotypes, and then mutate and recombine those copies to produce one or two new genotypes. Hence, in EAs, children *receive* recombined genomes from their parents, except in certain EA types that prohibit recombination.

In EA terminology, mutation and crossover are *genetic operators*. They are performed on the genotypes, not the phenotypes, although constraints from the phenotypic level may be used to bias the genotypic manipulations. Mutation involves single genotypes, while crossover involves groups of 2 (or occasionally more). In mutation, a small component of an individual is randomly changed. If the genotype is a bit vector, then one of the bits is simply flipped. For more complex genotypes, such as a permutation, two integers of the permutation are swapped.

In general, crossover involves taking parts from 2 or more parent genotypes and combining them into one (or several) new child genotypes. With bit-vector genotypes, crossover swaps pairs of bit segments, as shown in Figure 3. With more complicated genotypes, the design of a crossover operator requires careful thought, since genotypic segments may be less modular and hence more difficult to mix and match under the standard constraint that the resulting child genotypes should be translatable into legal phenotypes.

The inverted pentagon from upper to lower right in Figure 2 indicates that some of the mating adults may produce several copies of their genotypes, portions of which will then appear in newly formed genotypes. The rectangular reproduction box indicates that, for the most part, all portions of the copied parent genotypes will appear in a new child genotype.

As Figure 2 shows, the three critical evolutionary components of selection, inheritance and variation are cleanly modularized in EAs. Adult and mate selection are the only points where the population is filtered, chosen parents are guaranteed to pass on their genotypes via simple copying, and mutation and recombination operators create the genotypic variations so critical to evolutionary change.

After many rounds through the EA cycle, the average fitness of phenotypes increases as better and better solutions/designs arise. Eventually, optimal solutions may be found, although there is no guarantee of optimality with EAs as there often is with exhaustive brute-force methods.

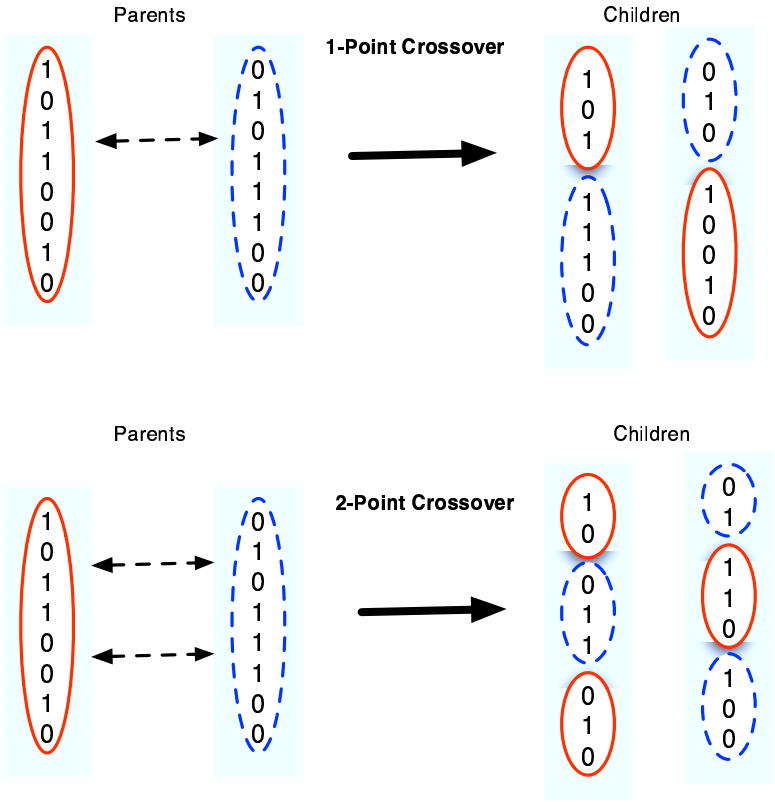

Figure 3: Single- and Double-point crossover operations on bit vectors.

3.1 Basic Data Flow in an Evolutionary Algorithm

Figure 4 illustrates the basic flow of data structures (typically implemented as objects) in an EA.

The evolutionary process begins with a population of genotypes as shown in the bottom left corner of the diagram. These are normally generated randomly, although some EA applications involve strong biases during initialization due to known or predicted structure in the solutions.

Phenotypes are then derived from the genotypes via the developmental phase. The genotypes themselves are retained for later use in reproduction, just as organisms retain their DNA throughout life.

Fitness testing of the phenotypes then assigns a fitness value to each individual, as depicted by the numbers in each cloud. This value reflects the ability of the *phenotype* to solve the problem at hand, but it determines the prospects for the individual to pass elements of its *genotype* into the next generation.

As described above, adult selection may weed out inferior (i.e., low fitness) offspring, allowing admission into the adult pool to only the better performers. Parent selection then chooses individuals whose genotypes (or portions thereof) will be passed on to the next generation. In theory, all adults remain in the adult pool until the next round of adult selection, when all or some of them may get bumped out to make room for promising new adults.

Finally, the genetic material of the chosen parents is recombined and mutated to form a new pool of genotypes, each of which is incorporated into a new object. These objects then begin the next round of development, testing and selection.

The cycle halts after either a) a pre-determined number of generations, or b) one of the individuals has a fitness value that exceeds a user-defined threshold. The latter is only practical when the user has concrete knowledge of the fitness value that corresponds to an optimal solution. For example, in a difficult travelling salesman problem, the value of an optimal tour may not be known ahead of time, but in a constraint-satisfaction problem such as a Sudoku, the optimal score is easily ascertained without detailed knowledge of the actual solution.

4 Essential Components of an Evolutionary Algorithm

To facilitate the process of Figure 2, an EA designer must include the following basic components:

1. Genotype representation
2. Phenotype representation
3. Translator of genotypes into phenotypes
4. Genetic operators for forming new genotypes from existing ones.

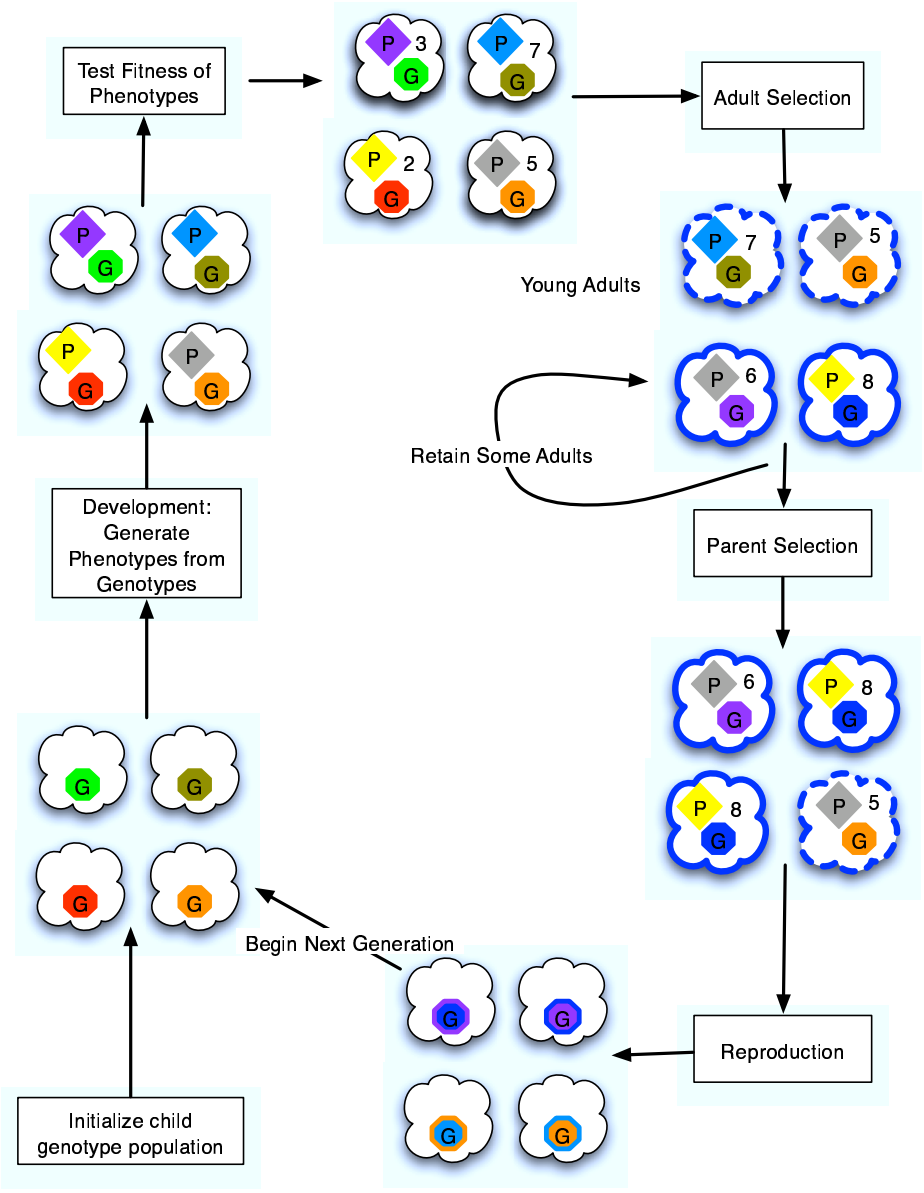

Figure 4: Flow of individual computing objects (clouds) through an evolutionary algorithm. Objects begin with only a genotype, then acquire a phenotype via development and a fitness value via performance testing. Fitness then biases selection of both adults and parents/mates.

5. Fitness assessment procedure
6. Selection strategy for child phenotypes
7. Selection strategy for adult phenotypes

The first four of these components are often discussed together, since a) the genotypic level is, by definition, that for which we define genetic operators, and b) the choices of genotypes and phenotypes are strongly influenced by the potential ease/difficulty of both the translation mechanisms and genetic operations. Therefore, the upcoming section on genotypes and phenotypes is interspersed with discussions of points 3 and 4 as well.

Several of these 7 components are general and hence reusable across many different problem domains. Typically, the selection strategies are extremely modular and highly problem independent. A good EA has a library of a half dozen or so (see below) strategies which can be tested to find the best fit for a given situation.

At the other extreme, the fitness assessment procedure is always very problem dependent, since it embodies the key knowledge as to *what* a solution to a particular problem should do.

Phenotypes also have a tendency to be problem dependent, since they embody much of the semantics of the domain. However, there is still possibility for code reuse at the phenotypic level. For example, a general phenotypic type such as *permutation of integers 1..N* could be used to represent solutions to both an N-city travelling salesman problem (TSP) and an N-task ordering problem. In this case, the only difference would be in the fitness assessment procedure.

Genotypes are often amendable to reuse, particularly the very low level variants such as bit vectors. For these genotypes, general genetic operators are easily written. As genotypes move to higher levels, i.e., move closer to the phenotypic representations, their generality declines and the need for special purpose genetic operators often arises.

In general, a good EA provides a solid backbone of code for running the basic evolutionary loop along with a library of reusable phenotypes, genotypes, genotype-to-phenotype translators, and selection strategies. It also includes simple hooks for adding in new representations and translators, preferably as sub-classes to existing ones.

Despite this generality, even a good EA will require that the user understand the basic evolutionary procedure in order to tune parameters such as the population size, selection strategy, mutation and crossover rate, etc. Furthermore, many complex problems will require a specialized phenotypic (and possibly genotypic) representation not included in the off-the-shelf EA. In short, it helps to enjoy programming!

5 Genotype and Phenotypes

In most cases, the problem domain and target computational machinery for an EA will influence the choice of phenotype, which should *mean something* in that domain. For instance, if the goal is to design a finite state autonoma for controlling a chemical process, then the phenotype will probably include states and the transitions between them. The choice of genotypic syntax will depend upon the phenotypes and on the EA designer's preferences. Genotypes may be close (or identical) to phenotypes, and thus very problem dependent, or they may be very generic. For example, bit vectors are very generic genotypes, while real numbers arranged into rows and columns are more specific and probably map directly to real-array phenotypes.

A complete, exception-free classification of EA genotypes and/or phenotypes is difficult to formulate, since many thousands of EA applications exist, most (but not all) of which are basic variations on a few common themes. De Jong [13] uses 4 representational categories for genotypes:

1. *fixed-length linear objects* - simple, vectors of values, with each value denoting a gene and each vector having the same length and retaining that length throughout the EA run.
2. *fixed-size nonlinear objects* - potentially complex data structures, some of which are easy to linearize, such as n-dimensional arrays and trees, and others which are take more work, such as connected graphs with loops. These do have a fixed size, so crossover is generally straightforward to implement. However, the sections that get swapped during crossover are not necessarily effective, modular, or *meaningful*, building blocks.
3. *variable-length linear objects* - vectors of values whose lengths are not all equal and may vary during the course of the run. Here, mutation is easy but crossover can pose a challenge.
4. *variable-length nonlinear objects* - These are the most difficult, since the complex objects can also vary in size, making crossover potentially difficult both syntactically and semantically.

Our classification of genotypes is orthogonal to DeJong's and focuses on the semantics of the information that the genotype contributes to the phenotype, rather than the syntax. We differentiate two main classes of genotypic representations:

1. *data oriented* - these encode several data values whose usage in the phenotype may vary but does not include actual data manipulation or program control.
2. *program oriented* - these encode explicit data processing and control information - supplementary data may also be encoded - to form the kernel of an executable program at the phenotypic level.

This separation closely mirrors two distinct representational semantics found in the EA community:

1. - collections of parameters - often variables in optimization problems - used by genetic algorithm (GA), evolutionary strategy (ES) and evolutionary programming (EP) researchers.

2. computer programs - used in the field of genetic programming (GP).

Later sections discuss the key differences between GA, ES, EP and GP, the 4 primary types of EAs.

5.1 Data-Oriented Genotypes

The classic example of a data-oriented genotype is a list of parameters, encoded either as a bit string or as an array of integers or reals. These parameters may represent anything from dial settings for a factory controller, to variable values in a function optimization problem, to weights for a neural network, to room numbers for an exam scheduler. The examples are endless, and in many cases, an EA is the perfect tool for the job.

At a lower level, there are a host of relatively standard representations for genotypes and their accompanying mappings to phenotypes. Six of these appear in Figure 5.

Beginning in the center of Figure 5, the classic bit vector (common to genetic algorithms) is a simple list of bits, subsequences of which are translated into phenotypic traits such as integers or real numbers. Mutation is a simple bit flip.

One potential weakness with the classic bit vector representation is the mediocre correlation between genospace and phenospace. There is no guarantee that a small (large) change in a genotype will result in a correspondingly small (large) change in the resulting phenotype. Consider the genotype 011 for phenotype 3. By flipping EVERY bit, i.e., by making the maximal change to the genotype, we form the neighboring phenotype, 4. So a large change in genotype space produces a small change in phenospace. Conversely, by mutating only one bit, the most significant, in 111, the phenotype jumps dramatically from 7 to 001 = 3.

In short, the classic bit vector representation has only a mediocre correlation between genospace and phenospace, and there is a modest amount of effort required to convert genotypes into phenotypes, i.e., to convert bit strings to integers.

To combat the correlation problem, many EA researchers use Gray coding. Gray codes are binary encodings for integers that are designed to have a high correlation. Hence, given any gray-coded bit vector, V, that maps to integer M, any single-bit mutation to V will produce an integer close to M. The conversion of gray-coded bit vectors to integers is nearly as easy as the decoding of normal bit strings, so classic and gray-coded bit vectors appear at similar locations along the x axis of Figure 5, but the gray-codes are much higher on the y axis.

Many EA practitioners use representations that have both high correlation and a simple conversion process. Real vector genomes are typical of this type. Here, the genome is simply a list of real numbers, so the genotype and phenotype are essentially the same. Mutation is performed directly on the reals, not on their binary representations. The standard form of mutation to perturb the original value by a small amount d, which is chosen from a normal distribution with a mean of 0 and a problem-dependent standard deviation.

Many applications, such as the Travelling Salesman Problem (TSP), involve solutions that are permutations of a set of integers. For these, a direct permutation representation is often appropriate. These also employ a list of numbers (in this case, integers), where genotype and phenotype are identical. Mutation involves the simple swapping of two integers in the list. Crossover of permutations is a bit more complicated and discussed in a later chapter.

Indirect representations of permutations are also possible such that the genotype is a bit string that translates into a valid permutation. As shown in Figure 5, indirect permutations require some conversion effort between genotypes and phenotypes, and the correlation between the two spaces is very weak. Hence, this is a representation that must be used with caution.

Finally, some genotype-phenotype mappings are very complicated such that the developmental process is very involved and the correlation between the two spaces is low. For example, bit strings may be converted into circuit layouts or neural network topologies (as depicted in the bottom right of Figure 5). These are much more difficult representations to handle, but they enable evolution to solve intricate design problems. Several examples of these are given throughout the book.

5.1.1 A Robot Example

To illustrate the wide variety of data-oriented representations for any given problem, consider a situation where the link between genotype and phenotype is less obvious. In this example, a robot must be evolved to enter a potentially dangerous area and outline the periphery of all suspicious objects with warning markers. The first prototype might be a simple office robot that must differentiate between red (dangerous) and blue (harmless) objects and learn to pick up and place blue objects around red ones to mark the hazard, as depicted in Figure 6.

Assume that the robot has 8 color sensors, 4 for blue and 4 for red, with a sensor of each type for each of the four directions: front, back, left and right. The red sensors are labeled RF, RB, RL, RR, while the blue are BF, BB, BL, BR. Sensors give simple binary readings, so, for example BB = T means that a blue object is detected directly in back of the robot. In addition, two sensors detect whether the robot is carrying a red or a blue object. These are denoted RC and BC and are also binary.

The robot has 6 possible actions:

1. MF - move forward
2. MB - move back
3. TL - turn left 90°
4. TR - turn right 90°
5. PICK - pick up the object in front of the robot.
6. DROP - put down object being carried in front of the robot.

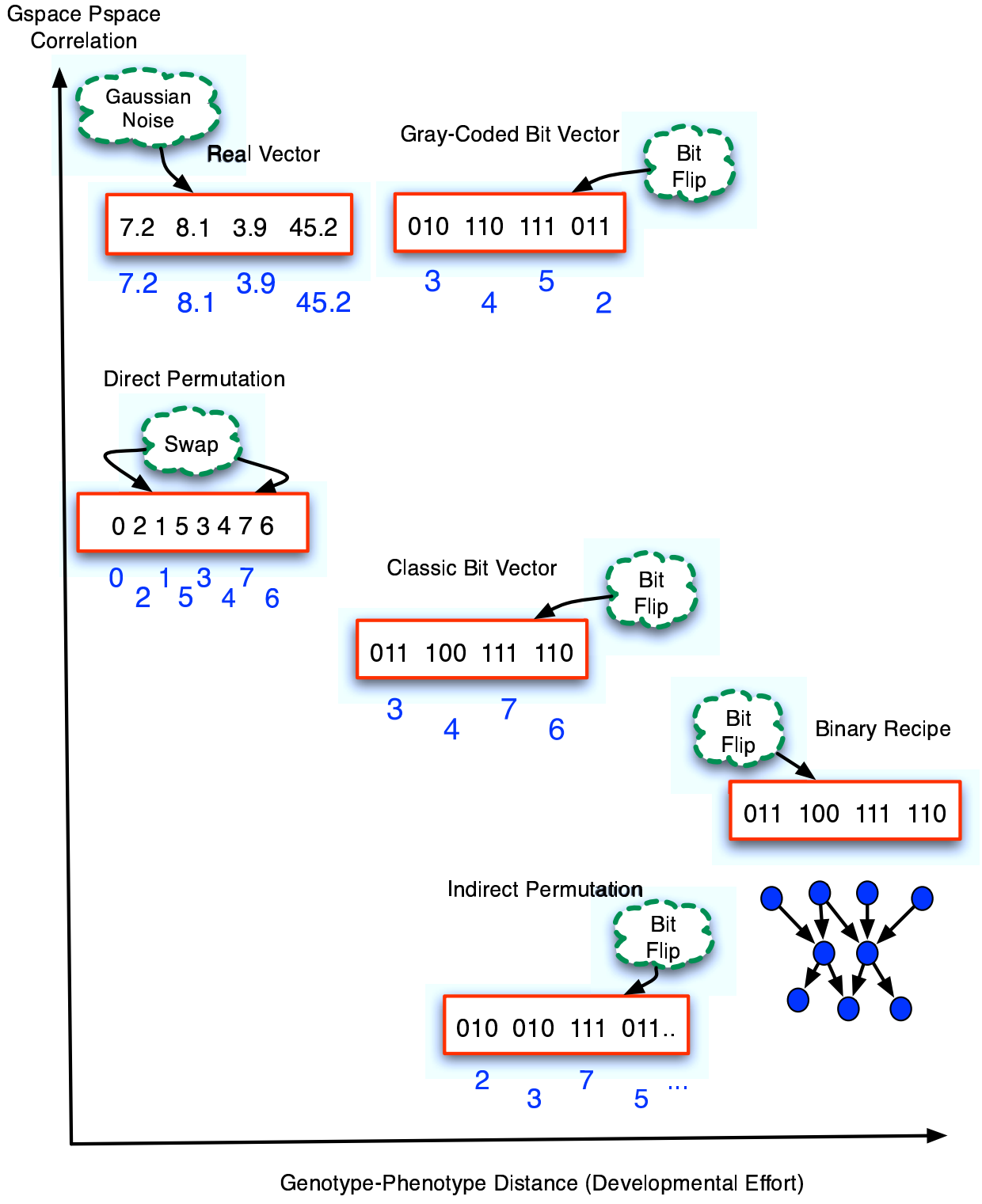

Figure 5: Six of the many data-oriented representational approaches in evolutionary computation. These vary with respect to the distance between genotypes and phenotypes, i.e., the developmental effort (x axis) and the correlation between genospace and phenospace (y axis). For each approach, the genotype is a red box, while the phenotype is underneath in blue; the mutation operator is in the dashed cloud. Representations that traditionally require the least computational overhead are in the upper left, while those involving more resources appear in the lower right.

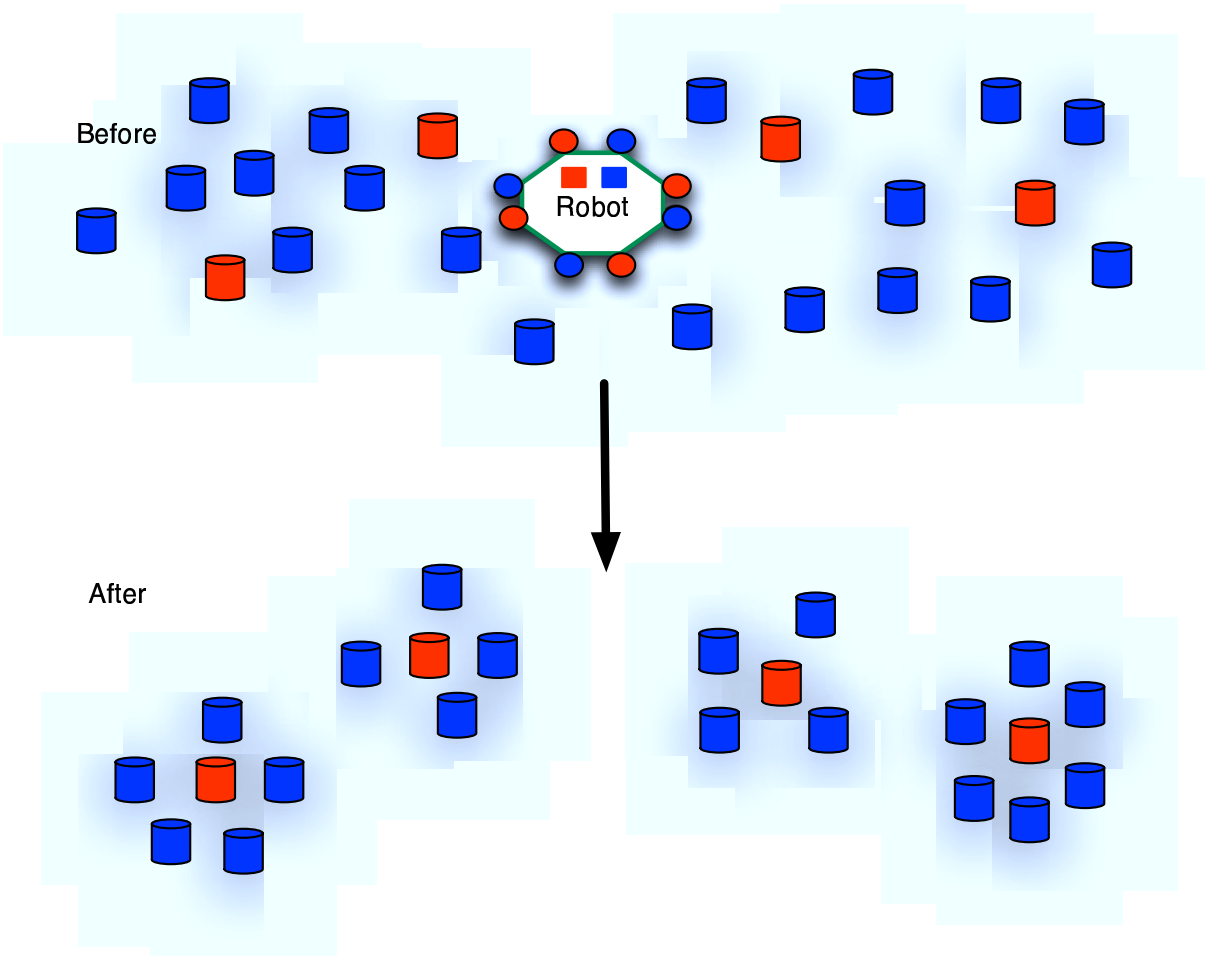

Figure 6: A robotic task of placing (blue) warning markers around dangerous (red) objects, where the robot has 8 color sensors, 4 each for red and blue (drawn as small colored circles on the robots exterior) and two carry sensors (drawn as colored squares on the robot's body).

If the robot were designed by hand, then some of the rules might be:

$$BC \wedge RR \Longrightarrow TR \tag{1}$$

When carrying a blue object and detecting a red object on the right, turn to the right.

$$BC \wedge \neg BB \wedge \neg RB \wedge RF \Longrightarrow MB \wedge DROP \tag{2}$$

When carrying a blue object and detecting a red object in front but no objects in back, back up and drop the object.

$$RL \wedge \neg RR \wedge \neg BR \wedge \neg BC \wedge \neg RC \Longrightarrow TR \wedge MF \tag{3}$$

When detecting a red object on the left and no objects on the right, and not carrying anything, turn right and move forward.

The EA must use evolution to design a set of rules that enable the robot to find red objects and surround them with blue objects. We can bias the EA such that all rules have sensory conditions on the left and action combinations on the right, but beyond that, we should probably give evolution free reign to discover useful sense-and-act heuristics. As a aside, note that this representation does not qualify as program-oriented, since the explicit control commands such as the *if* and *then*, along with kernel control code for the underlying rule-based system is not specified by the genome, but assumed during the running of the phenotype.

The straightforward phenotypic representation would be a list of rules, using the same sensing and acting primitives and similar in format to those above. However, the choice of genotype is a bit more difficult. One option, shown in Figure 7, encodes rules as bit strings in which the first 3 bits of each rule determine the number of sensory input variables, K. The next 5K bits are then parsed as input variables, with the decoding shown at the top of the figure. After that, the final 6 bits determine how many of the 6 possible actions will be performed, with opposing actions such as TL and TR simply cancelling on another out if both are true.

This representation is reasonably compact and flexible, since few bits are wasted, and rules can have varying lengths. However, it is highly susceptible to lethal mutations, particularly those of the length-3 count bits. If one bit of one count changes, this alters the translation of the rest of the genome. Hence, a good rule set could, with one bit flip, become a terrible one. In other words, variation would be high, but at the cost of low inheritance. Of course, mutations to other bits would provide more local changes and strike a friendlier balance between variation and inheritance.

One way to provide some insurance against lethal variations is to enable rule codes to spread themselves around the genome, without necessarily bordering one another. The rough sketch of the genome on the top of Figure 8 illustrates the basic idea. Here, as in real biological genomes, various tags indicate the beginning of a coding segment. For simplicity, assume the tag 11111 denotes the start of a new rule segment. The genotype-to-phenotype parser simply searches the genome from left to right until it hits a 11111 tag. It then parses the bits that immediately follow the tag as a rule (in the same manner as in Figure 7). From the end of this rule, it continues to the

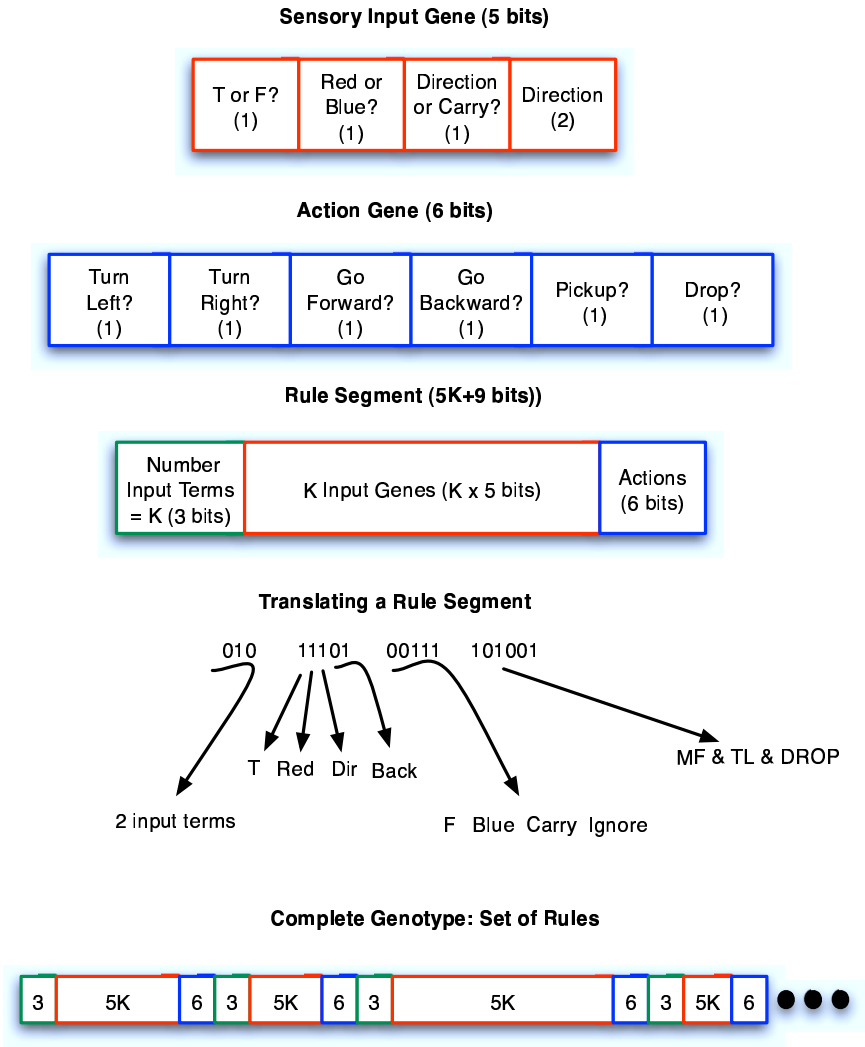

Figure 7: A flexible genotype representation for the robot controller. Rules consist of K input terms, where $0 \leq K \leq 7$ is encoded by the first 3 bits (the count bits) of the rule, and a 6-bit action vector. The dynamic coding of K implies that rules differ in bit length, and a small change to the count bits of one rule can alter the interpretation of all succeeding rules. The translated sample rule is $RB \wedge \neg BC \Longrightarrow MF \wedge TL \wedge DROP$.

right in search of another tag. Between the end of a rule segment and the next tag, many unused bits may lie. These perform a similar function to introns in genetics, since they help isolate gene segments and thus reduce their interaction with other segments. For instance, if the count of one rule increases, the unused filler bits can be used to extend the rule segment, without interfering with the downstream rules. They also increase the probability that randomly-chosen crossover points will occur between such segments rather than within them, thus preserving linkage inside the segment; i.e., the segments tend to be inherited as units.

Even with tags and filler bits, a potentially unwanted source of variation still exists within the rule segments. If the count bits mutate to a higher number, then the new inputs will be taken from the old action segment, and the new actions will be bound in the previous filler bits. Similar problems occur if the count decreases. This annihilates potentially useful action combinations (that evolution may *work so hard* to produce). It makes more sense to preserve the actions and existing sensory-input conditions of a rule and to use the filler bits as the source of the new input condition. A simple reordering of the rule bits facilitates this straightforward reduction in variation: now the action bits come directly after the count bits, and before the input bits, as shown at the bottom of Figure 8.

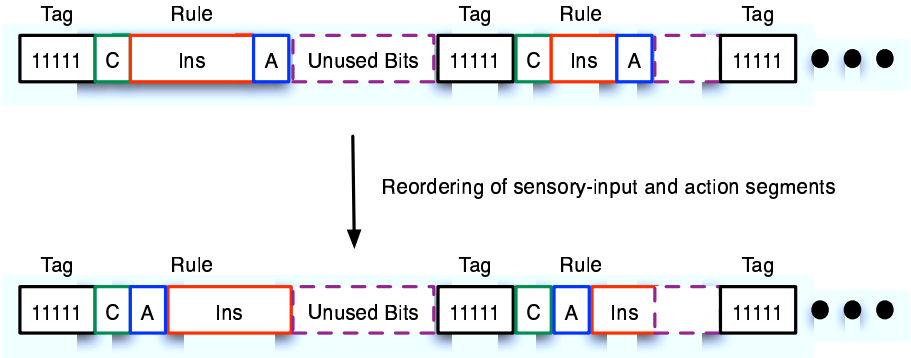

Figure 8: A genotype representation for the robot controller that helps prevent lethal mutations by allowing rule segments to be separated by free space (i.e., unused bits). The tag 11111 indicates the start of a rule segment, while C, Ins and A denote the count, sensory-input and action bits, respectively. The lower genome illustrates an alternate encoding wherein action bits occur directly after the count bits and become protected from the side-effects of count changes.

Yet another bit-vector genotype for the robot problem is possible; this one removes all potential inter-rule interactions stemming from the side-effects of genetic operations. Consider that each of the 10 possible sensory inputs can take on one of 3 values in a rule: true, false, unimportant. This gives a total of $3^{10} = 59049$ possible preconditions for a rule, hardly a difficult size for today's computers. To represent any number between 0 and 59048 (or 1 and 59049) requires $\lceil log_2 59049 \rceil = 16$ bits. Thus, as shown on top of Figure 9, a rule could be represented by a 16-bit precondition selector, which chooses among the 59049 unique preconditions, followed by a 6-bit action vector identical to those used in the previous 3 representations.

Finally, if a complete lookup table for all possible sensory input scenarios is desired, then we can ignore the *don't care*/unimportant option for each variable. This yields a much smaller total of

$2^{10} = 1024$ scenarios. Associating 6 action bits with each scenario gives a complete action strategy and uses only $6 \times 1024 = 6144$ bits. A rough sketch of such a strategy appears at the bottom of Figure 9.

For this last strategy, the precondition bits are not needed in the genome, since they are implicitly represented by the indices of each 6-bit action vector. For example, the precondition for the 14th action vector (using 0-based indexing) is 0000001110 (i.e., 14 in binary), which denotes the conjunction:

$$\neg RF \wedge \neg RB \wedge \neg RL \wedge \neg RR \wedge \neg BF \wedge \neg BB \wedge BL \wedge BR \wedge RC \wedge \neg BC \tag{4}$$

under the assumption that, for genotype-to-phenotype translation, the sensor variables are ordered as follows: RF, RB,RL, RR,BF, BB, BL, BR, RC, BC.

Clearly, this representation explicitly encodes an action strategy for each of the 1024 specific sensory scenarios. Hence, it can be described as an *extensional* strategy, since the extension of a concept is all its individual instances. Conversely, the earlier versions are *intensional* in the sense that single rules are normally general (unless they explicitly mention all 10 input sensors on their left hand sides) such that their extensions must be computed.

For example, the rule:

$$RB \wedge \neg BC \Longrightarrow MF \wedge TL \wedge DROP \tag{5}$$

is intensional, with an extension consisting of $2^8 = 256$ detailed rules to account for all possible values of the 8 sensory variables that are **not** explicitly mentioned on the left-hand side of 5. One such rule is (with the core intensional component in boldface):

$$\mathbf{RB} \wedge \neg \mathbf{BC} \wedge RF \wedge RL \wedge RR \wedge BF \wedge \neg BB \wedge \neg BL \wedge \neg BR \wedge \neg RC \Longrightarrow \mathbf{MF} \wedge \mathbf{TL} \wedge \mathbf{DROP} \tag{6}$$

and another is:

$$\mathbf{RB} \wedge \neg \mathbf{BC} \wedge \neg RF \wedge \neg RL \wedge \neg RR \wedge \neg BF \wedge BB \wedge BL \wedge BR \wedge RC \wedge \Longrightarrow \mathbf{MF} \wedge \mathbf{TL} \wedge \mathbf{DROP} \tag{7}$$

Although, in theory, these two representations, intensional and extensional, provide the same *expressibility* (i.e., they can represent the same behavior rules), it is clear that a general rule such as 5 would have a very slim chance of arising in the lookup table, since its complete extension (i.e., all 256 cases) would need to appear. In other words, the table would need to have 256 identical action vector entries (of $MF \wedge TL \wedge DROP$) to account for all scenarios in which RB is true, BC is false, and the other 8 sensory variables take on any possible truth-value combination.

In this, and many other, situations, the choice between an intensional and an extensional representation must be considered. Extensional representations can often by molded into fixed-size

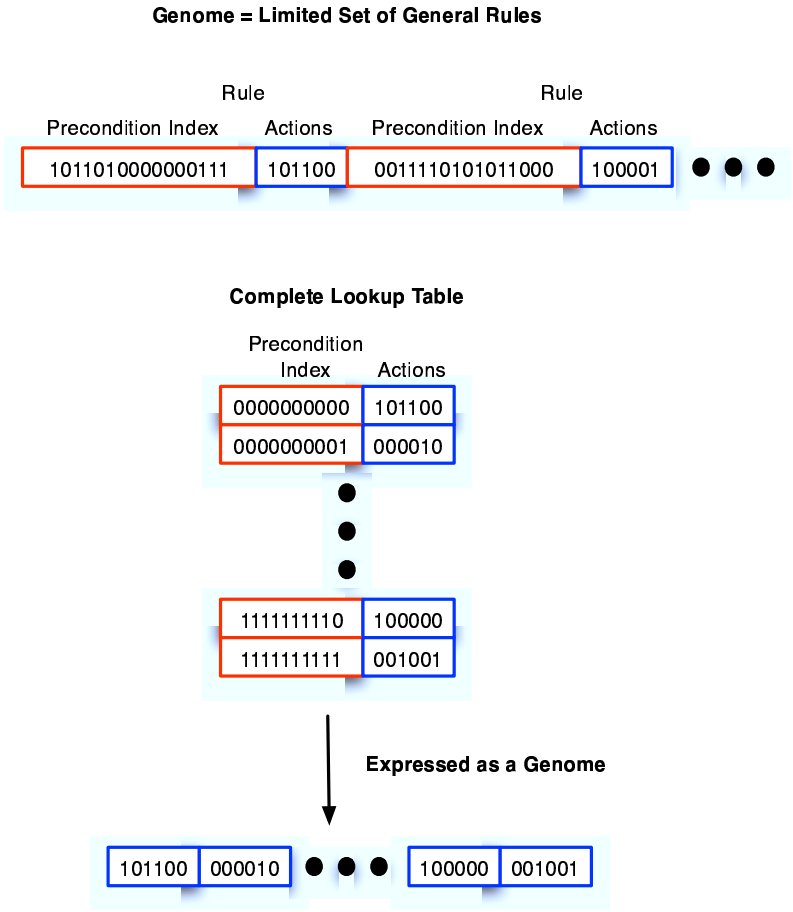

Figure 9: Additional genotypes for the robot problem that use indices into the collection of all possible preconditions. (Above) The genome handles only some of the sensory-input situations, so the index of each scenario is required. (Below) All 1024 scenarios are accounted for, so scenario indices are implicit in the genome.

genomes with none of the inter-gene dependencies (termed *epistasis* in biology) that plague many of the more-flexible intensional representations. With low epistasis, the danger of one mutation completely reorganizing the phenotype decreases significantly. Hence, heritability, variation and selection can cooperate to evolve useful problem solutions. However, the extensional approaches hinder the emergence of useful general rules. They can also produce very large genomes, which require more evolutionary time to improve.

Regardless of these differences, a key feature of all the above representations is that they allow random bit-flip mutations and recombinations - simple examples of bit-vector crossover appear in Figure 3 - to produce genotypes that are guaranteed to be translatable into legal phenotypes. The balance between variation and heritability of these genotypes with respect to the parent genotypes will vary, but they will develop into understandable phenotypes, regardless of the extent of mutation and crossover.

This feature insures that the basic bit-vector genetic operators of mutation and crossover can be programmed once and used for all of the genotypic representations above, and many more. For each new bit-vector representation, only a new translation/development module must be written to convert the genotypes to legal phenotypes. Thus, at least at the genotype level, an EA appears quite representation independent: the same generators of variation (i.e., new solution hypotheses) can work on genotypes that encode rule sets, neural networks, arrays of control variables, and many other phenotypes.

In effect, the genetic operators embody the hypothesis-generating intelligence of an EA. If these are designed to work on any bit string, regardless of the phenotype it encodes, then their extreme generality and domain independence trades off against a complete lack of understanding concerning strategic changes most likely to improve solutions, i.e., a complete lack of semantic knowledge.

Although all genotypes are represented by bits at some layer of the computer, the true level of the genotype for evolutionary computation is defined by the genetic operators and the knowledge that they use to mutate and recombine genotypes. The genotype becomes equivalent to the phenotype in those cases where the genetic operators incorporate the high-level semantics of the phenotype to manipulate genotypes.

In the above robot example, we could elevate the genetic operators closer to the phenotypic level by devising mutation rules such as:

1. Switch any blue sensor reading, such as BF, to the corresponding red one, RF, or vice versa, in the precondition of a rule.
2. Switch any movement to the opposing movement, i.e., MF to MB, and TR to TL.
3. If the rule includes a positive red sensory reading and no mention of a blue object being carried, then supplement the actions with a move away from the red object and remove any move toward it.

Crossover rules might include conditions such as:

1. When combining partial precondition lists of two rules, resolve any contradictions such as $RL \wedge \neg RL$ by randomly choosing one of the two terms.

2. When combining partial action lists of two rules, resolve pairs of opposing actions by randomly keeping one and deleting the other.

In this case, it would be safe to say that the genetic operators are working directly on the phenotype, although we will always differentiate the genotype and phenotype, since internally in the EA, there are very often two distinct representations, even though the *developmental* process may be trivial. Although the use of high-level genetic operators can lend advantageous direction to search, it may also bias evolution too strongly, precluding the attainment of optimal phenotypes whose design might require the creative forces of synthetic evolution over the rigid guidance of fundamental engineering principles.

Unfortunately, there is no standard answer for the proper level of abstraction for genetic operators. Problem-independent operators have obvious advantages, including both code reuse and providing evolution with maximum search flexibility. However, extremely complicated search problems sometimes require the bias of higher-level operators. As with much of evolutionary computation, the crafting of representations and genetic operators is more art than science. And as such, it is one of the more interesting, creative, and challenging aspects of evolutionary problem solving.

5.2 Program-Oriented Genotypes

In the robot example above, the genotype encoded a set of if-then behavioral rules. Thus, the genotype formed the *basis* for a rule-based controller. However, the genotype did not contain the if-then structure; that decision was taken by the EA designer, who interprets bit segments as if-then rules to be run by a rule-based system kernel.

In program-oriented genotypes, a significant amount of the computational control knowledge resides in the genotype itself, and is thus open to manipulation by genetic operators. Hence, an if-then may mutate into an if-then-else or a while loop or a case statement.

The term *genetic programming* (GP) was coined for this type of EA. Several comprehensive books [14, 15, 16] by Koza are chock full of both toy and real-world GP applications along with explanations of the essential components of a GP system. Banzhaf et. al. [1] give an excellent (and relatively concise in comparison to Koza) overview of GP and clearly show its relationship to other EA approaches.

In most GP applications, the genotype and phenotype are nearly identical. Hence, the genotype resembles a piece of computer code, although it must often be slightly re-packaged to actually run it. This means that the genetic operators must be designed to manipulate code, not just bit strings, such that new genotypes are syntactically valid code segments, not random gibberish. For many programming languages, designing such genetic operators would be an arduous task indeed. Imagine randomly swapping the lines of 2 C or JAVA programs. Would the child programs run? In most cases, they surely would not.

Fortunately, the LISP programming language provided the perfect substrate for Koza's groundbreaking forays into GP. The prefix format of LISP commands, combined with the recursive structure of LISP programs, facilitates a simple tree-based representation of LISP code. Trees are then easily mutated and recombined such that, with the enforcement of a few minor constraints, genotypes can be randomly combined to produce viable new code trees.

As a simple example, consider a curve-fitting problem of finding a mathematical function (of a single variable X) that best matches a set of data points: $(x_1, y_1), (x_2, y_2), ...(x_n, y_n)$. This function may be complex, but it can probably be decomposed into the recursive application of many simple operators, such as addition, subtraction, multiplication and division. Hence, these will be our 4 primitive operators: +, -,* and /. In GP terminology, these are the *function set*.

Computer programs (and mathematical functions) also need variables and constants; these are called *terminals* in GP. For a single-variable curve-fitting problem, the terminal set must contain X along with a few numerical constants, such as (-3, -2, -1, 0, 1, 2, 3).

Figure 10 shows a typical GP program in both tree form and as a linear list. The list is easily interpreted when a) the *arity*, i.e. number of arguments, of each function are known, and b) the code is known to be written using prefix notation (i.e., operators come before their operands).

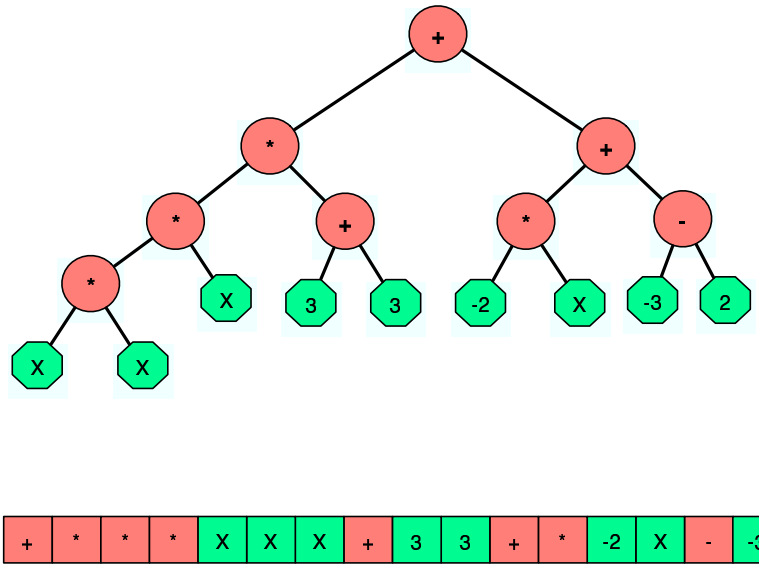

Figure 10: (Top) A genetic program tree representing the function $6X^3 - 2X - 5$. Functions are denoted by circles, terminals by octagons. (Middle) A linearization of the same function using prefix notation. (Bottom) LISP code for the same function. Note the lambda wrapping needed to make the expression an executable program.

A few basic design constraints insure that any randomly-generated GP tree that combines the

above functions and terminals will compile (when preceded by the *(lambda (X)..* function-creating code) and run:

1. When building trees, all functions must have all of their argument slots (i.e., child nodes) filled by either terminals or subtrees with functions as their roots.
2. *The closure property* - all functions must be defined so that all of their argument slots accept any of the terminals and any of the possible outputs from any of the functions.
3. Functions such as division that would normally crash on certain values, such as a 0 denominator, must be rewritten to handle those cases and output a value that is an acceptable input to all functions.

Normally, to satisfy the closure property, all GP functions for a given problem domain are designed to work with the same type of data, such as real numbers or booleans. If arithmetic and logical functions are combined in a program, then its common to rewrite the booleans so that they a) interpret positive input arguments as true, and non-positive values as false, and b) output a 1 for true and a -1 for false.

Functions such as division are rewritten to output a 0 if division by zero is attempted. Similarly, a log function may output a 0 if given a negative input.

Together, these constraints insure that random combinations of functions and terminals will run without error. Of course, it says nothing about the actual fitness of the programs.

The standard genetic operators for GP are mutation and crossover. Mutation involves the replacement of a random subtree with a newly-generated subtree (not necessarily of the same size). Standard GP crossover, depicted in Figure 11, merely swaps random subtrees of two GP trees. The closure property guarantees that the results of mutation and crossover are valid programs.

When working with a linear representation of a prefix-coded tree, mutation and crossover work similarly, in that subtrees are replaced and swapped, respectively. However, subtrees are not inherent in the linear data structure and must be found using a simple trick:

- count = 0; subtree S = $\emptyset$
- i = random index into the code vector, V.
- while count $\neq$ -1 do
  - count = count - 1 + arity(V(i))
  - append V(i) onto end of S
  - i = i + 1

  return S

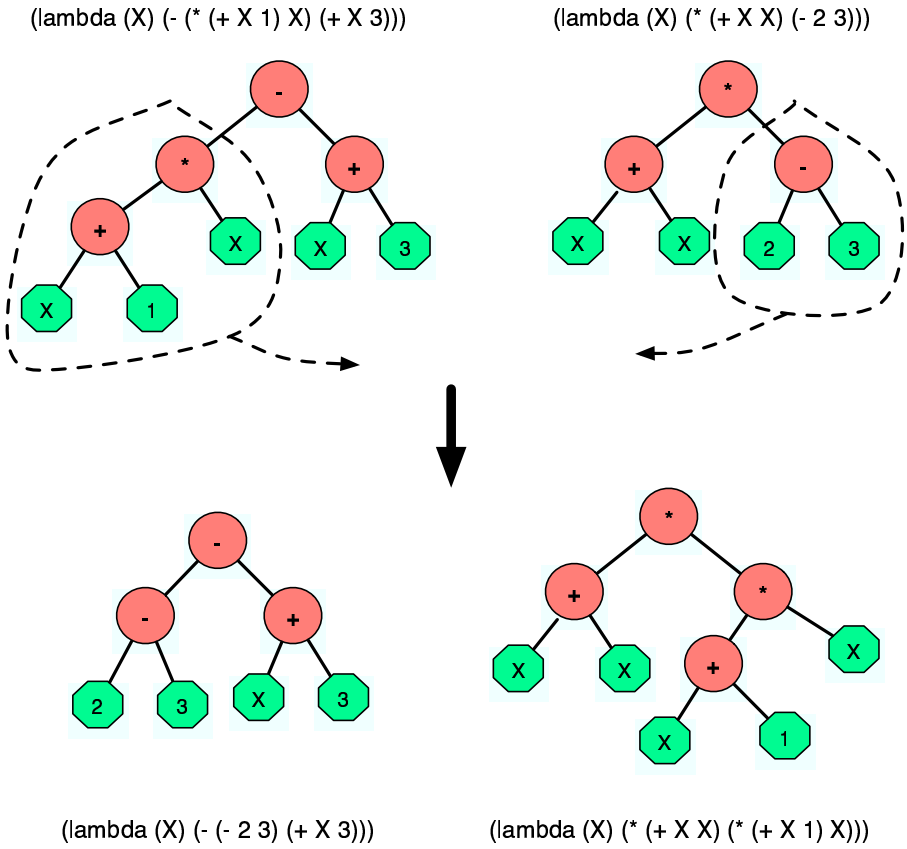

Figure 11: Crossover via subtree swapping in genetic programming. (Above) The subtrees (outlined by dotted lines) of two parents are exchanged. (Below) The child genotypes resulting from the subtree swap.

Here, the arity of a function is the number of arguments that it takes, while the arity of a terminal is 0. Figure 12 shows a simple example of a subtree hunting in the program from Figure 10. Once subtrees are found, they are swapped by removal and insertion into their code arrays. Since the two swapped subtrees may vary in size, code vectors for linear GP must have flexible dimensions.

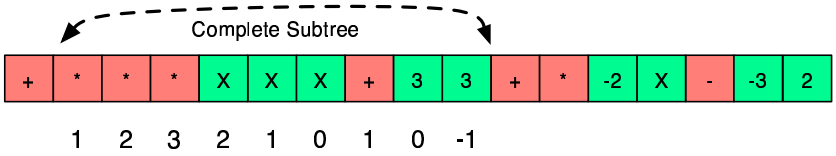

Figure 12: Using the counting procedure to find a random, but complete, subtree. Numbers below each code element denote values of the *count* variable after that particular item has been processed.

The previous example illustrates the representational flexibility of GP: given a few primitive functions and terminals, a wide variety of complex functions can be crafted by evolution. These open-ended design possibilities are the trademark of GP as compared to both other EAs and automated search and design algorithms in general.

To see how GP can evolve computer programs other than mathematical functions, consider our familiar robot example. A relatively simple GP permits the evolution of quite complex control code. Assume a terminal set consisting of the 10 sensory inputs and 6 motor outputs. These can also be viewed as 0-argument functions. In this case, the most obvious primitive functions are logical, not arithmetic, so AND, OR and NOT are appropriate choices, with the former two taking two arguments and the latter taking one.

In addition, a conditional expression such as IF is helpful, so we define two versions:

1. IF2(condition, action)
2. IF3(condition, action, alternate action) - corresponding to an if-then-else

Finally, a block-building construct enables the sequential execution of large code segments. In LISP, PROGN(code sequence, code sequence) serves this purpose. To create longer sequences of code, simply nest the PROGNs; for example, (PROGN TL (PROGN MF PICK)) performs a left turn, forward move and pickup operation in sequence.

So the complete function set is: AND, OR, NOT, IF2, IF3, PROGN, and the terminal set is: RF, RB,RL, RR,BF, BB, BL, BR, RC, BC, MF, MB, TL,TR, PICK, DROP.

To satisfy the closure property, each logical operator returns T (true) or F (false), while all 6 motor commands output T. IF2, IF3 and PROGN are defined (as in standard LISP) to return the output value of the final argument that gets evaluated So, for example, IF3 returns the output of its alternate action in cases where the condition is false, and PROGN returns the return value of its second code sequence.

Evolution will invariably produce programs that are inefficient and/or, at least on the surface, make little sense; but they do run! For example, (IF3 ML RF BB), uses a movement command in the condition spot. Since ML always returns true, the net effect is to move left and then read the front red sensor but do nothing contingent on its value. Consequently, GP programs are often very hard to interpret with the naked eye.

Figure 13 shows a GP program of similar functionality to 4 sense-act rules:

1. $(RR \vee RL) \wedge BC \Longrightarrow DROP$
2. $RF \Longrightarrow MB$
3. $\neg RF \Longrightarrow MF$
4. $BF \Longrightarrow PICK$

This code would run on each timestep of the fitness-assessment simulation.

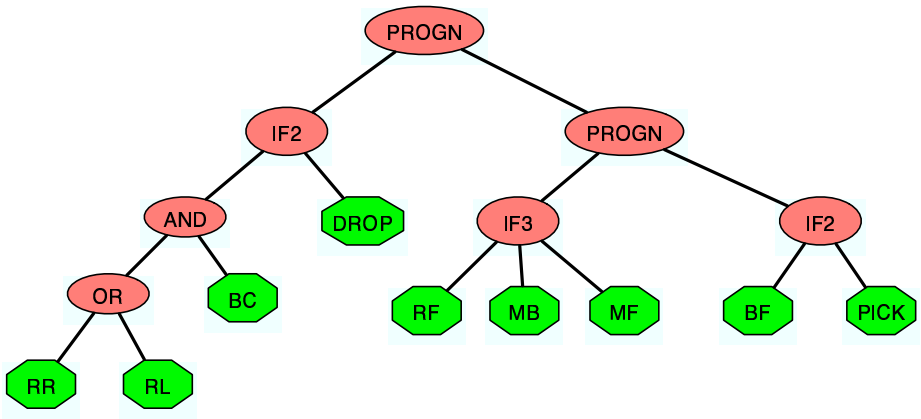

Figure 13: A sample GP program for the robot problem.

Although a more complete comparison of GP and the other EAs appears in a later section, the essential difference is representational: the GP evolves complete algorithms with the great majority of data processing and control decisions determined by evolution, not the user. In theory, nothing else separates GP from the rest of EA. All general discussions of fitness landscapes, fitness assessment, selection strategies, etc. need not specify the type of EA, although each community (GA,ES,EP,GP) has different consensus preferences.

6 Fitness Assessment

In population biology, *fitness* generally refers to an individuals ability to produce offspring [20]. In evolutionary computation, such a definition would appear circular, since EAs use fitness values to

determine reproductive success. EAs, unlike biologists, cannot watch a population reproduce and then afterwards assign fitness to the productive individuals. EAs must be more proactive and *play God* by stepping in and restricting access to the next generation. The fitness evaluation is the first (and main) step in that process.

As discussed earlier, an EA generally has little knowledge about *how* a good solution/hypothesis should be designed, but solid information about *what* a good solution should be. Most of that information is implicit in the fitness function. Whereas the phenotypic representation and the genotype-to-phenotype mapping embody knowledge of how to generate a *legal* individual, it is the fitness function that assesses the individual's *goodness* with respect to the problem. So the extent of *how-to-construct* knowledge in the EA is normally restricted to legal phenotypes, not superior ones.

Fitness assessment involves two steps: performance testing and fitness assignment. In the former, the phenotype is applied to the problem to be solved by the EA. The results of this are then converted into a quantitative fitness value, which then follows the individual around like a college grade-point average to open (and close) doors to the future. That is, the selection mechanisms use the fitness value to prioritize and filter hypotheses.

In the curve-fitting GP example, a typical performance test goes through each of the n x-y pairs, (x_i, y_i). The value x_i is used as input to F_j, the complete function defined by phenotype j, producing output $\tilde{y}_i$, which is then compared to y_i to compute a contribution to the total error, E_j. The sum-of-squares is a typical error function for this purpose:

$$E_j = \sum_{i=1}^{n} (y_i - \tilde{y}_i)^2 \tag{8}$$

The fitness of phenotype j could then be as simple as:

$$Fitness(F_j) = \frac{1}{1 + E_j} \tag{9}$$

In the robot example above, the performance test would involve a simulated 2d block world with red and blue objects. Each phenotype (i.e., set of behavioral rules) would be downloaded into a simulated robot that would then move around this world under the control of these rules. Initially, the red and blue objects would be randomly spread about the plane. The robot would then run for a number of timesteps, say 1000. Upon completion, the state of the world would be analyzed for certain factors of relevance to a solution. These factors are, ideally, necessary and sufficient conditions for problem success, but such conditions are not always easy to define. But in this case, our definition of the task is fairly straightforward, so a good final world state should have a) all red objects surrounded by many blue objects b) few blue objects standing alone in the plane.

To quantify this, for each red object, count the number of blue objects that are within a short distance d of its center, a.k.a., *warning blues*. The initialization procedure might insure that no blue objects are within d of any red object. By itself, this warning-blue count could be a fairly

effective fitness value. In addition, the EA could take into account the isolated blue objects and, say, subtract their count from that of the warning-blues. In most cases, it is useful to avoid negative fitness values, so the isolated-blues count might be divided by a scaling constant, or all negative results could simply be mapped to zero fitness.

For a travelling-salesman problem (TSP), the phenotype would probably encode the proper sequence of cities to visit: first Dallas, then Denver, then Portland, etc. The performance test would then *walk through* that route and tally up the total distance. A standard fitness value would then be the inverse of this distance.

If an EA were used to design a good classifier for a machine-learning system, then the performance test would involve testing each phenotype classifier on a training set of (input, desired-output) pairs. The classification error, E, of a phenotype - i.e., number of the classifiers outputs that differed from the desired outputs - could then be the basis of its fitness, F. For example, $F = \frac{1}{1+E}$.

Error is also a natural fitness metric for scheduling problems. Assuming that the EA must design a good assignment of college exams to discrete time slots and classrooms, then the phenotype would consist of times and rooms for each exam. Then, given a list of all students, professors and the courses that each takes/gives, the performance test would measure error as the number of violations of hard (e.g., no student or professor should have two exams at the same time) and soft (e.g., no student should have two exams on the same day) constraints.

In general, the exact fitness value matters very little, but the *relative* fitnesses of hypotheses must reflect the relative utility of their solutions. If one individual receives a fitness of 10, while another receives 2, then in the eyes of the system designer, the former should be approximately 5 times *better*. Failure to achieve appropriate fitness spacing among individuals can cause evolutionary search to sputter. However, as discussed in the section on selection mechanisms, there are certain phases of the search when it is desirable to artificially widen or narrow the effective fitness gaps.

7 Selection Strategies

Whereas fitness assessment should provide a very objective, unbiased measure of an individual's quality, selection strategies often introduce stochasticity into the processes of survival and mating. Just as in nature, where the strongest and fastest may accidentally fall off a cliff, meet an oncoming train, or simply have a couple *off nights* during mating season, there is often no guarantee with EAs that either a) the most fit individuals will pass on the most genes, or b) the worst fit individuals will not reproduce at all. This is generally an advantage, since high-fitness individuals may have hidden weaknesses or represent merely local optima, while low-fitness genotypes may encode useful traits that have not yet been complemented with enough other traits to show their true utility. In short, an EA normally profits by keeping some options open: by maintaining a good balance between exploration and exploitation.

7.1 Fitness Variance

In theoretical evolutionary biology, classic results by Sewall Wright and Ronald Fisher led to the *Fundamental Theorem of Natural Selection* [20, 9], which states that the rate of evolution (measured as the rate of change of the composition of the gene pool) is directly proportional to the **variance** of individual fitness. So evolution requires fitness variation, and the greater the variation, the faster the population evolves (and adapts to its environment).

Noting that fitness in biology refers to reproductive efficacy, the fundamental theorem implies that evolution can only occur when individuals have different reproductive rates. If each individual produces one offspring (or each mating pair produces 2), then the gene pool will stagnate.

The same principle applies to evolutionary algorithms: If all individuals have the same fitness, then they will all have approximately the same likelihood of passing on their genes, and the population will not change much from one generation to the next. Hence, for evolution to proceed, a sizable fitness variance should be maintained.

Unfortunately, EAs often suffer from a fitness variance that is either too high or too low. In the early generations of an EA run, most of the individuals tend to have (very) low fitness, while a few will have more respectable values simply by virtue of doing a few things correctly (often by accident). For example, a robot that always picks up blue blocks and then deposits them anytime it sees other blocks, whether blue or red, will score better than one that never picks up anything. This results in a very high fitness variance, since most of the individuals will have zero or near-zero fitness, while a select few will have values reflecting *decent* performance, but nothing exceptional.

The problem is that the low fitness individuals will normally have almost no chance of reproducing, while the respectable genotypes will quickly dominate the population. Hence, the EA will *prematurely converge* to a homogeneous population of mediocre individuals, a local optima. This homogeneity implies very low fitness variance, so evolution will grind to a halt and the EA's best-found solution will be far from optimal. To remedy this situation, the fitness variance should be artificially reduced during the early stages of an EA run.

Conversely, near the end of a run, the population will often converge on an area of the fitness landscape that, indeed, does hold an optimum or near-optimum solution, but since all individuals are quite good, there will be little fitness variance and thus no driving force for continued improvement. The population will essentially run out of useful hints regarding the goal. In these cases of *late stagnation*, the fitness variance should be artificially increased so that *the cream of the cream* gets a slight, but significant, reproductive edge.

These artificial modifications to fitness variance are achieved by selection mechanisms, which often scale fitness values before picking and pruning parents. In effect, this alters the fitness landscape, as shown in Figure 14.

7.2 Selection Pressure

The concept of *selection pressure* ties directly to the scenario of Figure 14. In effect, it is the degree to which the real fitness variance translates into differences in reproductive success. A high selection pressure will give a strong reproductive advantage to individuals that are only slightly better than their peers, while a low selection pressure will treat individuals more evenly, despite fitness differences. So the example of Figure 14 reflects an initially low selection pressure, with a high selection pressure toward the end. This strategy of selection-pressure change is often ideal for an EA.

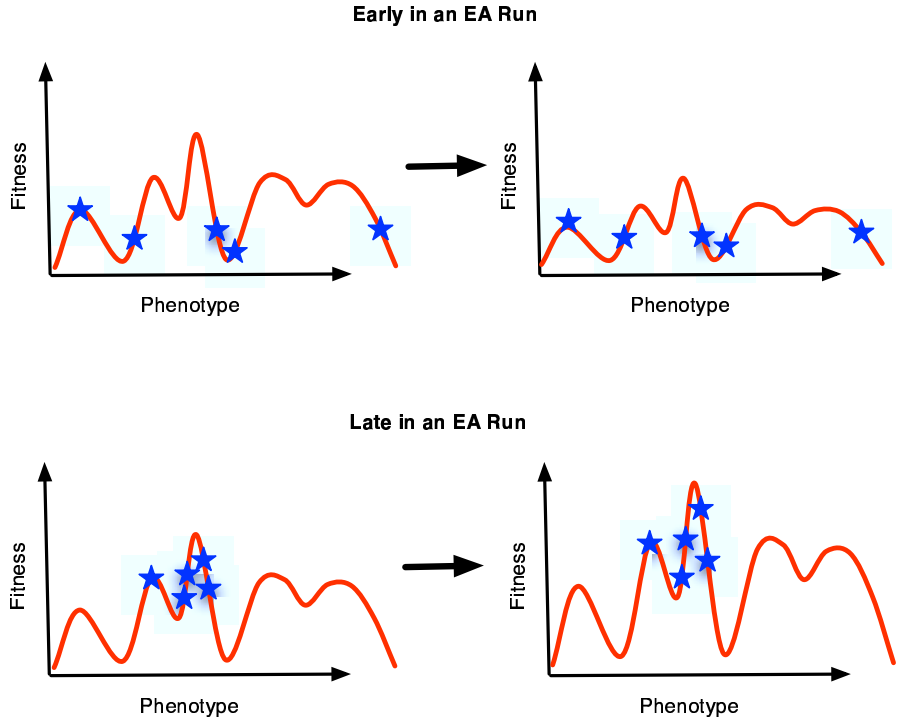

Figure 14: Virtual modifications to the fitness landscape performed by typical selection mechanisms. Stars represent individuals. In early stages of evolution, fitness variance is artificially decreased, which implicitly compresses the fitness landscape. In late stages, variance is artificially increased, leading to an implicit stretch of the landscape.

7.3 Selection Strategy for an EA

In the classic EA reference literature [13, 1, 19], the terminology of selection varies a bit. In the following, we introduce a slightly new framework but stick as close as possible to the consensus terms.

A *selection strategy* consists of a *selection protocol* and a *selection mechanism*. The protocol determines which individuals will participate in selection; it bases this decision upon the type of

individual (e.g., parent or child) not upon the fitness value. Conversely, the selection mechanism filters participants based on their fitness values.

Referring back to Figure 2, a selection strategy must be defined for both *adult selection* and *mate selection*. As shown in Figure 15, there are three main protocols for adult selection:

1. *A-I: Full Generational Replacement* - All adults from the previous generation are removed (i.e., *die*), and all children gain free entrance to the adult pool. Thus, selection pressure on juveniles is completely absent.
2. *A-II: Over-production* - All previous adults die, but m (the maximum size of the adult pool) is smaller than n (the number of children). Hence, the children must compete among themselves for the m adult spots, so selection pressure is significant. This is also known as (μ, λ) selection, where μ and λ are sizes of the adult and child pools, respectively.
3. *A-III: Generational Mixing* - The m adults from the previous generation do not die, so they and the n children compete in a free-for-all for the m adult spots in the next generation. Here, selection pressure on juveniles is extremely high, since they are competing with some of the best individuals that have evolved so far, regardless of their age. This is also known as $(\mu + \lambda)$ selection, where the plus indicates the mixing of adults and children during competition.

For mate selection, the protocol is normally trivial: all adults participate in the mating competition.

For both adult and mate selection, a host of selection mechanisms are available. These vary in the degree to which individuals compete locally or globally and by the type of fitness scaling that may occur prior to filtering.

With *local* selection mechanisms, such as tournament selection, individuals participate in a competition with only a small subset of the population, with the winner moving immediately to the mating pool. With *global* selection mechanisms, an individual implicitly competes with every member of the adult population.

7.4 Global Selection Mechanisms

Many of the global selection mechanisms involve a *roulette wheel* on which each adult is alloted a sector whose size is proportional to the adult's fitness. In asexual reproductive modes, if n children are to be produced, the wheel is spun n times, with the winning parent on each spin sending a (possibly mutated) copy of its genotype into the next generation. In sexual reproductive schemes, pairs of wheel spins are employed, with the two winner parents passing on their genotypes, which may be recombined and mutated, normally yielding 2 children. Repeating this process n/2 times yields the complete child population.

Naturally, fitness values must be normalized in order to divy up the area of the roulette wheel. In addition, almost all global selection mechanisms scale the fitness values prior to normalization. These scaling techniques are typically the defining feature of the selection mechanism.

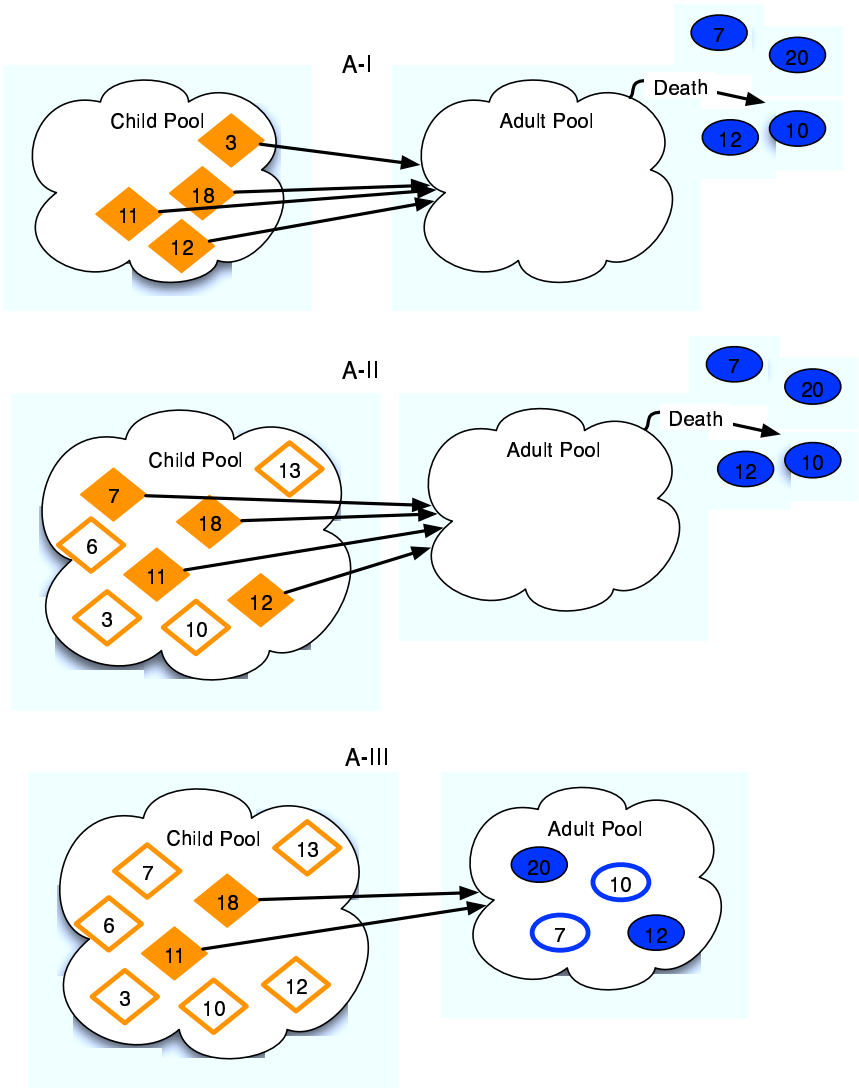

Figure 15: Three basic protocols for adult selection. Top: Complete turnover of adults with m = n. Middle: Complete turnover of adults with n > m. Bottom: Adults and children compete equally for the m adult spots. In each diagram, numbers denote fitness, while unfilled diamonds (children) and circles (adults) indicate individuals who lose out in the competition. Arrows emanating from diamonds indicate children who win acceptance into the adult pool.

Our overview of basic selection mechanisms uses Holland's [12] concept of *expected value*: the expected number of times that the parent will reproduce. The original fitness values are scaled into these expected values prior to roulette-wheel normalization. We also closely follow the description of selection mechanisms given by Mitchell [19].

The roulette-wheel metaphor is best explained by a simple example. Assume a population of 4 individuals with the following fitness values: 1) 4 2) 3 3) 2 4)1. To convert these to space on the roulette wheel, simply divide each by the sum total of fitness, 10. By *stacking* the resulting fractions, the individuals each get a portion of the [0, 1) number line. The sub-ranges for each are: 1) [0 0.4), 2) [0.4, 0.7), 3) [0.7, 0.9) 4) [0.9, 1.0). These are equivalent to sectors on a roulette wheel. Selecting a parent becomes a simple weighted stochastic process wherein a random fraction, F, in the range [0, 1) is generated. The sub-range within which F falls determines the chosen parent. Clearly, individuals with higher fitness have a greater chance of selection.

The classic selection mechanism is *fitness proportionate*, in which fitness values are scaled by the average population fitness. Of course, dividing m fitness values by their average and then normalizing is equivalent to simply normalizing the original m values. Hence, this mechanism merely scales fitnesses so that they sum to 1 and thus properly fill up the roulette wheel; it does not modify their relationships to one another. In other words, it does not alter the selective advantages/disadvantages inherent in the original values and thus does not implicitly change the fitness landscape.

Mitchell and others often use the roulette-wheel metaphor as a unique feature of fitness proportionate selection, but it applies equally well to many global selection mechanisms, since most work by normalizing all expected values and then using randomly-generated fractions to simulate the *spinning* of the wheel and choice of a parent.

Sigma scaling selection successfully modifies the selection pressure inherent in the raw fitness values by using the population's fitness variance as a scaling factor. Hence, unless this variance is 0 (in which case all fitnesses scale to expected values of 1.0), the conversion is:

$$ExpVal(i,g) = 1 + \frac{f(i) - \bar{f(g)}}{2\sigma(g)} \tag{10}$$

where g is the generation number of the EA, f(i) is the fitness of individual i, $\bar{f(g)}$ is the population's fitness average in generation g, and $\sigma(g)$ is the standard deviation of population fitness.

This has the dual effects of a) damping the selection pressure when a few individuals are much better (or worse) than the rest of the population, since such cases have a high $\sigma(g)$, and b) increasing the selection pressure when the population has homogeneous fitness (thus low $\sigma(g)$). This helps avoid the problems of early and late stages of EA runs, as discussed above. Figures 17 and 18 illustrate these effects, showing the clear advantage of sigma-scaling over fitness-proportionate scaling in effectively modifying the selection pressure inherent in the original fitness values.

Boltzmann selection is based on simulated annealing, in which the degree of randomness in decision making is directly proportional to a temperature variable. This is based on the physical property

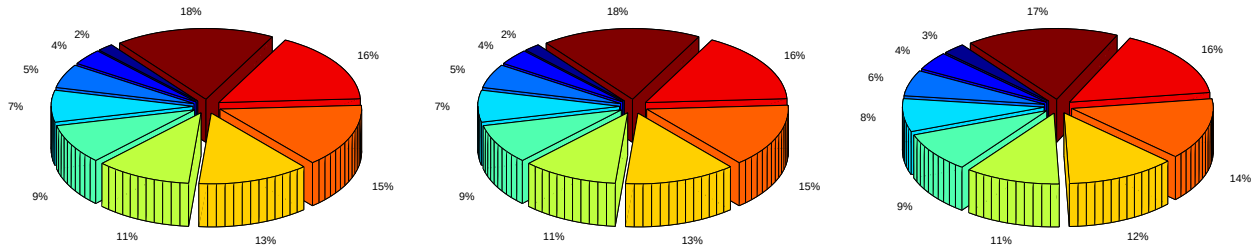

Figure 16: Comparison of selection wheels using (left) unscaled, only normalized, fitness values, (middle) fitness-proportionate scaling and normalization, and (right) sigma-scaling and normalization. The original fitness values in all 3 cases are 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

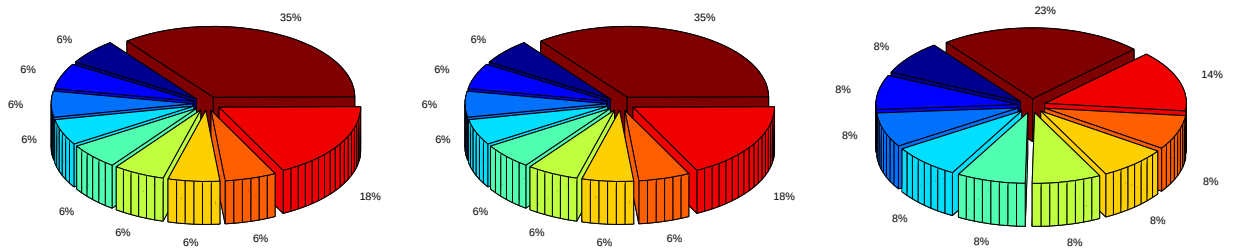

Figure 17: Comparison of selection wheels using (left) unscaled, only normalized, fitness values, (middle) fitness-proportionate scaling and normalization, and (right) sigma-scaling and normalization. The original fitness values in all 3 cases are 1, 1, 1, 1, 1, 1, 1, 1, 3, 6, which are typical of an early generation of an EA run. Note the more even distribution (i.e., lower selection pressure) for sigma scaling.

that molecules in a heated mixture exhibit more random movement (especially when it transitions from solid to liquid or liquid to gas) than under cooler conditions. With Boltzmann selection, higher (lower) heat entails a more (less) random choice of the next parent and hence less (more) selection pressure, since superior individuals have less (more) of a guarantee of passing on their genes.

The scaling equation for the Boltzmann selector is:

$$ExpVal(i,g) = \frac{e^{f(i)/T}}{\langle e^{f(i)/T} \rangle_g} \tag{11}$$

where g is the generation, T is temperature, f(i) is the original fitness of individual i, and $\langle e^{f(i)/T} \rangle_g$ is the population average of the fitness exponential during g.

As shown in Figure 19, as temperature falls, the odds of choosing the best-fit individual increase dramatically as it garners more and more area on the roulette wheel. Ideally, a Boltzmann selector uses a temperature that gradually decreases throughout the EA run, such that selection pressure gradually increases. Again, this ameliorates premature convergence and late stagnation.

Rank selection ignores the absolute differences between fitness values and scales them according to

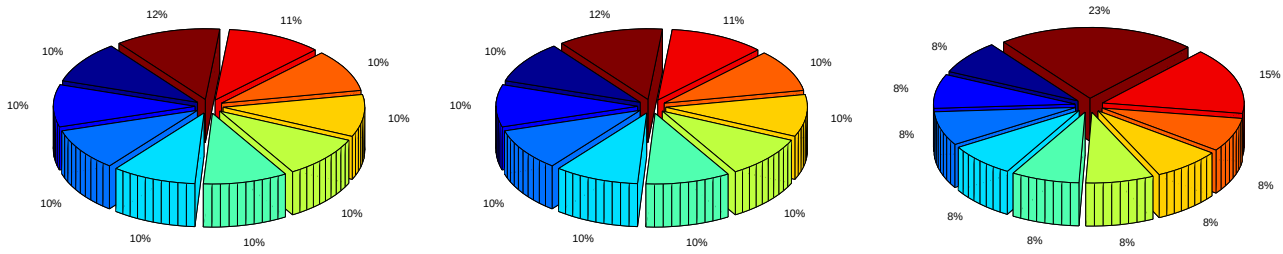

Figure 18: Comparison of selection wheels using (left) unscaled, only normalized, fitness values, (middle) fitness-proportionate scaling and normalization, and (right) sigma-scaling and normalization. The original fitness values in all 3 cases are 8, 8, 8, 8, 8, 8, 8, 8, 9, 10, which are typical of the later stages of an EA run. Note the more skewed distribution (i.e., higher selection pressure) for sigma scaling.

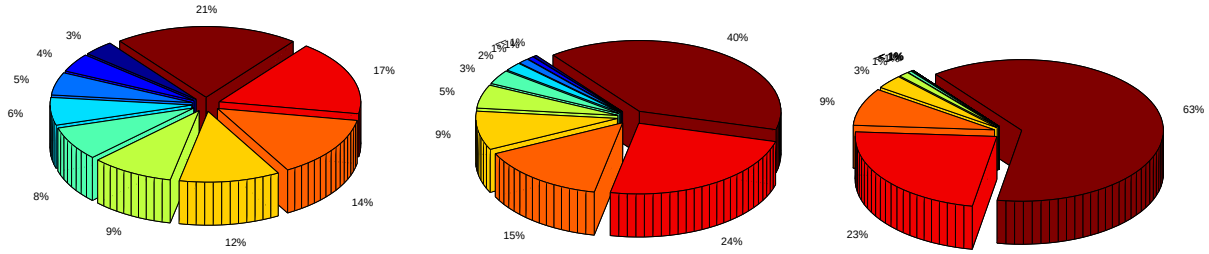

Figure 19: Increasing selection pressure by decreasing temperature in the Boltzmann selector from (left) 5° to (middle) 2° to (right) 1°. The original 10 fitness values are 1, 2, ... 10.

their relative ordering. Hence, if the best individual in the population has a fitness of 10, while second place is a 3, then the 10 will achieve no more roulette-wheel area than would a 3.1. This type of scaling also helps adjust selection pressure. It decreases the advantage of *lucky starters* during early stages of a run, and it tends to add some spacing between individuals when population fitness becomes very homogeneous. This helps combat the problems of premature convergence and late stagnation discussed above.

The basic scaling equation for rank selection is:

$$ExpVal(i,g) = Min + (Max - Min)\frac{rank(i,g) - 1}{N - 1} \tag{12}$$

where N is the popuation size, g is the generation, and Min and Max are the expected values of the least and best fit individuals, respectively. rank(i,g) is the rank of the ith individual during generation g, with the least-fit individual having rank 1 and the best having rank N.

Mitchell [19] points out the basic constraints that $Max \geq 0$ and $\sum_i ExpVal(i,g) = N$, from which it is straightforward to prove that $1 \leq Max \leq 2$ and $Min = 2 - Max$:

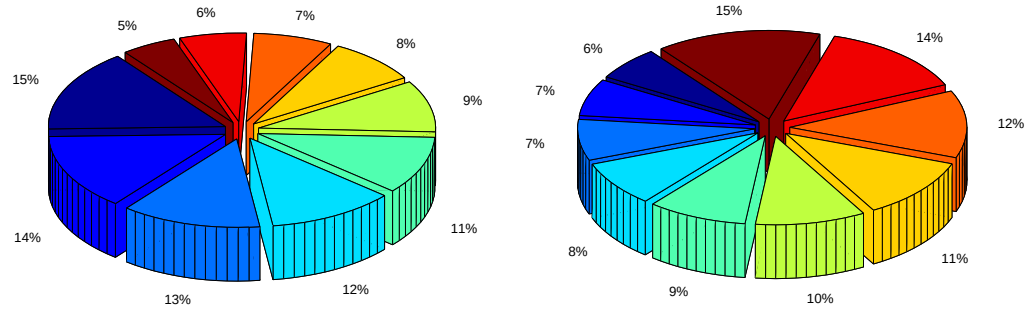

Figure 20: (Left) Normalized, rank-scaled fitness values for 10 phenotypes with original fitnesses 1,2...10. The scaling range is [0.5, 1.5] . (Right) The same fitnesses scaled by a Boltzmann selector using a temperature of 10. Notice that both give a fairly even partitioning of the roulette wheel, despite the high fitness variance. Hence, both exhibit low selection pressure for this wide fitness distribution.

$$\sum_i ExpVal(i,g) = \sum_{i=1}^{N} Min + (Max - Min)\frac{rank(i,g)-1}{N-1} = N \cdot Min + (Max - Min)\sum_{i=1}^{N}\frac{i}{N-1} \tag{13}$$

Then,

$$\sum_{i=1}^{N}\frac{i}{N-1} = \frac{1}{N-1}\sum_{i=1}^{N} i = \frac{1}{N-1}\cdot\frac{(N-1)N}{2} = \frac{N}{2} \tag{14}$$

Hence:

$$N \cdot Min + (Max - Min)\cdot\frac{N}{2} = N \tag{15}$$

Solving the above equation for Max, yields $Max = 2 - Min$. Since Min is the expected value of the worst individual, it should lie within the range [0, 1]: at best, the worst individual should get one copy of its genome in the next generation. Thus, $1 \leq Max \leq 2$.

As a typical setting, Min = 0.5 and Max = 1.5, or, for the maximum selection pressure possible with rank selection: Min = 0 and Max = 2.0.

Figure 20 shows that a rank selector with scaling range [0.5, 1.5] behaves similarly to a Boltzmann selector with temperature = 10. Although the color schemes are reversed in the two roulette wheels, the corresponding proportions match up well, and both exhibit a clear reduction of selection pressure (i.e. smoothing of selective advantages) for the heterogeneous fitness scenario.

For any roulette-wheel selectors, a *universal* variant is possible, wherein N pointers are evenly spaced about the wheel. Then, with just one spin, the N parents for $\frac{N}{2}$ pairings are chosen. This approach removes the random possibility of good individuals dominating the mating to a much

greater degree than sanctioned by the scaled fitness distribution. The next generation thus *holds true* to that distribution.

Finally, the term *uniform selection* refers to situations where, in effect, there is no selection pressure: all individuals have exactly the same chance of being chosen. In *deterministic uniform selection*, each parent gets to produce the same number of children, while *stochastic uniform selection* involves a roulette wheel on which all adults have equal area. EAs that use uniform selection at one level, such as during mate/parent selection, must typically use some form of fitness-based selection elsewhere.

7.5 Local Selection Mechanisms

The classic local mechanism is *tournament selection*, wherein random groups of K adults are chosen and their fitnesses compared. With a probability of $1 - \epsilon$, the best fit of the K is chosen for the next mating, while the choice is random with a probability of ϵ. The parameter ϵ is a user-defined value. N tournaments are thus required to determine all mating pairs.

Notice that this procedure allows poor genotypes to slip through to the next generation if either a) they get grouped with even worse individuals in a small tournament, or b) they lose a tournament but sneak through via the ϵ-ruling. In general, selection holds truer to the original (global) fitness distribution when both K is large and ϵ is small. In other words, selection pressure is a function of K and ϵ, with low K and high ϵ exhibiting low pressure, and high K and low ϵ giving much stricter selection.

8 Supplements to Selection

Although traditionally defined as a selection mechanism in its own right, *truncation* involves the immediate removal from mating consideration of an often sizeable fraction, F, of the adult population. In the classic case, the remaining adults simply produce an equal number of offspring. However, nothing prevents the subsequent application of any global (or local) selection mechanism to choose among these adults. Truncation is quite useful in situations where the genetic operators have a hard time maintaining substantial heritability, and thus, many children from good parents are, nonetheless, very poor performers and should be culled from the population.

Elitism is simply the retainment of the best E individuals in a generation. These are simply copied directly, without mutation or recombination, to the next generation. This prevents EAs from *losing* a superior performer before a better one comes along. This can be very important when population diversity is high, and thus the odds of recreating a similar good individual on each round are lower than when the population has converged upon a good region of the search space. Again, elitism can easily be added to any local or global selection mechanism. A typical value for E is 1 or 2 individuals, or a small fraction (1-5%) of the total population. Higher fractions can often lead to premature convergence.

9 Setting EA Parameters

The focus of this chapter has been on representations. The astute choice of genotype, phenotype, genetic operators and fitness function are normally the difference between problem-solving success and sending the EA off into a hopeless search space with nothing but a broken compass! Thus, at a qualitative level, these decisions are pivotal.

At a quantitative level, other choices can affect the efficiency of an EA; if not considered seriously, they can also ruin the EA's search. These parameters include the population size, the stopping criteria, and the mutation and crossover rates.

Although EA theoreticians have tried for years to produce useful guidelines for selecting these parameter values, the *No Free Lunch Theorem* [23] has been the most influential, and it effectively says that you **cannot** make general statements about proper parameter settings for search algorithms: they are strongly problem dependent. So, although an EC guru cannot give you the magic value of population size, mutation or crossover rate for all problems, a domain expert who has applied EAs to specific problems can tell you good parameter settings for EAs in that domain.

For example, if the representational choices have been so difficult that no recombination operator can guarantee high heritability, then a low crossover rate, of say 0.2, might be wise. This indicates that when two parents are chosen for crossover, only 20% of the time will they actually be recombined; in the remaining 80%, they will simply be copied (with possible mutations) into the next generation.

Mutation rates typically come in at least two varieties: per genome and per genome component (e.g., per bit in a bit-vector genome). Depending upon the problem, these may vary from as high as .05 per component (e.g. 5% of all genome components are modified) to .01 per individual (e.g. 1 % of all individuals are mutated in just ONE of their components).

Goldberg [10] and De Jong[13] provide some useful, general, tips for choosing EA parameters. One of the most critical, and most general, involves the well-known balance between exploration and exploitation [12]. To wit, De Jong emphasizes a balance of strength between the explorative forces of reproduction and the exploitative powers of selection.

Reproduction explores the search space by creating unique new genomes; as mutation and crossover rates rise, the extent of this exploration increases. Selection mechanisms control exploitation via the degree to which they favor high-fitness over less-impressive genotypes. De Jong's key message is that if reproductive exploration is high (e.g., the mutation rate is high), then selection should be highly exploitative as well. Conversely, low reproductive exploration should be complemented with weak selection.

This makes intuitive sense. If the EA is generating many unique genotypes in each generation, then in any reasonably-difficult search space, most of those genotypes will have lower fitness than the parents. Hence, it makes sense to filter them somewhat ruthlessly with a strong selection mechanism, such as tournament selection with low ϵ and high K. But if reproduction produces only a few unique individuals, then weak selection is required to give those new children a fighting

chance; otherwise, evolution will stagnate. Note that in both cases, the end result of a reproductive step followed by a selective step should be approximately the same amount of *innovation* in the next generation. The bottom line is that the *absolute* mutation and crossover rates are less important than their exploratory strength *relative* to the exploitative degree of selection.

Also, the population size of an EA deserves careful consideration. Normally, a large population size of 100, 1000 or even 500,000 is desirable for hard problems. Unfortunately, computational resources generally restrict practical population sizes to the 1000-10000 range, although, again, this is highly problem dependent. Some phenotypes, such as solutions to 20-city travelling-salesman problems or 20-node map-coloring problems, are easily checked for fitness, and thus, large populations can be simulated over many generations in just a few seconds of run time. Other phenotypes, such as electronic circuits or robot controllers, require extensive simulations (if not live runs of real robots) to test fitness. This can greatly reduce the practical population size to values as low as 10 or 20. Higher values may take weeks to run 50 or 100 generations!

The different branches of Evolutionary Computation (discussed below) have varying philosophies on population size. In general, Evolutionary Strategies (ES) and Evolutionary Programming (EP) researchers tend to use very small populations of 20 or less; some use a single individual! Genetic Algorithm (GA) and Genetic Programming (GP) aficianados frequently prefer large populations of hundreds or thousands. In general, to achieve the full power of parallel stochastic search in tough solution spaces, large populations are necessary. But to find a satisfactory combination of 30 parameters, for example, a small population of only 10 or 20 may be sufficient.

Finally, the stopping criteria for an EA must be determined. One can either a) set in a known (or estimated) maximum fitness as a threshold and stop simulation when a genotype achieves it, or b) simply set a pre-determined generation limit, G, and run until then. Both are trivial aspects of the EA, and the optimal choice is easily determined via experience in the problem domain. In general, if multiple runs of the same EA (on the same problem) are being performed in order to accurately assess the EA's problem-solving efficiency, then considerable run-time can be saved by cutting a run when fitness reaches a threshold value.

The tuning of EA parameters can often use up a significant fraction of your total project time. It is not unusual to hack together an EA in a few days (or hours) but to then spend an order of magnitude more time to actually get it to find good solutions. Start early! Be patient!

10 Four Evolutionary Algorithm Types

Unlike religion, evolutionary computation revolves around the belief that random variations and selection combine to produce incredible complexity of structure and behavior. However, like religion, the field of evolutionary computation is divided into several camps, all of which share similar fundamental views but insist on harping upon the minor (at least to the outsider) differences. And, as with religion, this has led to its share of animosity.

The four main EA types are Evolutionary Strategies (ES), Evolutionary Programming (EP), Genetic Algorithms (GA) and Genetic Programming (GP). Each shares the basic EA design philosophy

of (relatively) domain-independent hypothesis generation followed by domain-specific selection, and thus, each follows the basic scheme of Figure 2. However, they differ in primary representations, the relative importance of mutation versus crossover, and preferred selection strategies.

11 Evolutionary Strategies

In the 1960's, Rechenberg and Schwefel [21] invented Evolutionary Strategies in an attempt to solve search problems in hydrodynamics by randomly mutating existing solutions, assessing fitness, and applying selection. Their genotypes were vectors of real values that were mutated by making small changes to the reals, with the magnitude of change drawn from a Gaussian distribution centered at 0. Hence, small mutations were much more common than large, and evolution in ES was normally very gradual - especially given the fact that, in the earliest attempts, populations were of size 1!

Although ES initially ignored recombination, later work introduced several variations including both simple vector-segment swapping and the combining of corresponding vector values (e.g., via averaging) from the two parents. For example, if parents A and B had values 6 and 10, respectively, for the same gene, then the child would receive an 8 for that gene.

ES traditionally uses the over-production selection protocol A-II in Figure 15, although protocol A-III, Generational Mixing, is also employed. The latter provides a measure of elitism, since good parents can survive many generations by out-competing their offspring.

ES researchers also introduced the concept of genetically determined variation parameters, wherein factors such as mutation and crossover rates, Gaussian variances, etc., are encoded in the genome and thus open to evolutionary control. For example, in many simulations, an optimal evolutionary search employs high mutation rates early on (for greater exploration) and lower rates (for more exploitation) as the population converges on an optimal region.

ES is a strongly engineering-driven approach to evolutionary computation. Thus, those who use it are most interested in finding optimal solutions to technical problems, not in testing theories of evolution or intelligence.

12 Evolutionary Programming

Lawrence Fogel [7] had both engineering and artificial intelligence in mind when he invented Evolutionary Programming (EP) in the 1960's. He viewed simulated evolution as a potent tool for solving engineering problems and achieving human-level machine intelligence, even consciousness.

The philosophical difference between EP and the other EAs lies in the view of each phenotype as representing an entire species. Since, by definition, members of one species cannot mate with members of another, the concept of genotype recombination has no place in EP. Hence, it is the only EA in which crossover is forbidden.

This philosophy also implies that each parent/species should produce some offspring, so early EP systems used *uniform mate/parent selection* in which each parent produced the same number of offspring (originally just 1). Parent selection was then of type A-III, generational mixing.

EP was somewhat of a precursor to Genetic Programming (GP) in that the original applications were the design of finite state automata (FSAs) for sequential prediction tasks. FSAs consist of states, rules for inter-state transitions, and criteria for producing output. In theory, FSA's can be *souped up* to be Turing equivalent, but the basic versions are not. Regardless, FSAs provide only rather cumbersome solutions to problems outside their primary domain of sequence generation and recognition, so their generality as a programming substrate is far inferior to even the simplified LISP used in GP.

Typical EP genomes are integer vectors that encode the states, output conditions and transitions for an FSA. Gaussian mutations similar to ES are frequently employed; and like ES, EP has a form of self-adaptation. In this case, Gaussian variance is directly proportional to the distance from the current population to the known optimal value. So once again, as the population approaches an optimal solution, mutation decreases (in magnitude although not necessarily in frequency).

One of EP's most celebrated achievements comes from David Fogel [6], son of Lawrence Fogel. He and Kumar Chellapilla used the basic EP framework to evolve checkers-playing neural networks. As detailed in the book, their networks improved by playing against one another and against humans on zone.com, a popular checkers web site. Their virtual player, Blondie24, climbed the site's rankings to the elite 99.5 percentile; on one occasion it even beat Chinook, the world checkers champion (a human-written computer program). This served as at least partial confirmation that evolution could indeed be used to achieve impressive artificial intelligence.

13 Genetic Algorithms

Also in the 1960's, John Holland [12] recognized the potential of evolutionary mechanisms for artificial adaptive systems. His classifier systems combined learning and evolution using the same bit-string representation. If there is one name that most often comes to mind in connection with EAs, it is John Holland. Many, if not an overwhelming majority, of today's leading EA researchers are either students or *grandstudents* of Holland. Adjectives such as *monumental* and *groundbreaking* only begin to describe his contributions to EA.

Today, the main difference between GAs and other EAs is the bit-vector representation, which continues to be a staple in the field. In this respect, GAs are the only EAs that maintain any significant syntactic distance between genotypes and phenotypes (with one key exception being developmental GPs, discussed in a later chapter). Also, whereas EP forbids crossover and ES uses it sparingly, GA adherents normally consider it the most significant source of variation. GP researchers have a similar view.

Holland's seminal theoretical contribution comes from the *Schema Theorem*, which helps predict the frequency changes of useful gene combinations (a.k.a. *building blocks*) over the course of evolution. It takes both fitness and the sequence-disrupting effects of mutation and crossover into account to

derive the minimum speed at which a useful building block will dominate a population, and an inferior one will disappear. Not incidentally, the theorem also justifies the use of binary genotypes as opposed to closer-to-phenotype representations.

Holland also popularized fitness-proportionate mate selection; classic GAs use type A-I, full generational replacement, for parent selection.

Today, the top GA researchers seem open to just about any type of representation and selection strategy, although there still appear to be strong ideological (if not purely political) divides between the GA and GP communities. In his recent book, *Evolutionary Computation: A Unified Approach*, De Jong [13], a long-time GA standout (and student of Holland), only mentions GP on occasion and even fails to include it in key EC-overview tables. So clearly, there is still some *unifying* to be done.

14 Genetic Programming

Genetic Programming (GP) is clearly the most radical of the EAs. Few serious computer scientists would pursue the wild idea of randomly combining fully-functioning computer programs to generate new, better ones. John Koza (also a Holland' student) was the clever exception, and his perseverance has yielded a gold-mine of results that push EA well beyond the level of interesting novelty to potent problem-solving and creative design tool. GP applications run the gauntlet from musical composition and artistic design to state-of-the-art (in some cases patentable or patent-infringing) inventions of electrical circuits, antennas and factory controllers.

Additions to the original simplified-LISP version of GP include looping constructs, memory, and strong typing. Perhaps the most important enhancements were subroutines, which greatly improved the efficiency of GP search, as thoroughly documented in [15].

The ease of linearizing GP trees permitted a shift from LISP to C as the basis for large GP systems, although GP genotypes are still typically drawn as trees.

The discovery of cellular encoding (CE) [11] (described in more detail in a later chapter) set the stage for a long line of *human competitive* GP results. Originally designed to evolve artificial neural networks (ANNs) with GP, CE proved effective in evolving graph topologies for a broad range of domains, including biochemistry, electronics and control theory. CE also widened the (normally thin) gap between GP genotypes and phenotypes by using the genotype as a developmental procedure for *growing* the phenotype.

15 Convergence of EA types

Evolutionary biologists often speak of *convergence* in reference to similar traits that evolve independently. Eyes are an excellent example, having evolved on many separate occasions.

Evol Alg	Gtype Level	Mutate	Crossover	Parent Sel Proto	Parent Sel Mech	Mate Sel Mech
ES	Real Vector	Yes	Yes/No	A-II, A-III	Truncate	Uniform
EP	Int Vector	Yes	No	A-II, A-III	Truncate	Uniform
GA	Bit Vector	Yes	Yes	A-1	Uniform	FitPro
GP	Code Tree	Yes	Yes	A-1	Uniform	FitPro

Table 1: Brief overview of the 4 EA types and their traditional characteristics.

The general field of Evolutionary Computation has displayed this form of convergence, in that ES, EP and GA researchers all (relatively independently) realized the huge potential of the evolutionary metaphor in computational problem solving.

Now, another form of convergence, the mixing and homogenization of ideas, is clearly evident in all EA subfields, as each approach becomes less and less distinct from the others. Only GP remains somewhat segregated, since the evolution of programs is quite a bit different from that of variable lists.

Table 1 summarizes the features of the traditional versions of each EA. Of course, the partial integration of the 4 subfields has broken down many of the barriers, but, for example, EP folks still eschew crossover and ES researchers focus on engineering problems best represented by vectors of reals. In the GA and GP communities, fitness-proportionate selection has given up some ground to tournament selection, but almost all selection mechanisms can be found in a random sampling of GA and GP applications.

To the outside observer, with no bloodlines to the founding fathers of EA, the differences between the approaches seem vanishingly petite, and the disputes petty. For long-term success with EAs, the wise researcher will adopt (or design) a general system such as Sean Luke's ECJ (http://cs.gmu.edu/ eclab/projects/ecj/) and experiment with different representations, fitness functions and selection mechanisms with cautious disregard for which of the 4 EA types the eventual configuration most resembles.

References

[1] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*, Morgan Kaufmann, dpunkt.verlag, 1998.

[2] R. Beer, *The dynamics of active categorical perception in an evolved model agent*, Adaptive Behavior, 11 (2003), pp. 209–243.

[3] R. Brooks, *Cambrian Intelligence: The Early History of the New AI*, The MIT Press, Cambridge, MA, 1999.

[4] A. Clark, *Mindware: An Introduction to the Philosophy of Cognitive Science*, The MIT Press, Cambridge, MA, 2001.

[5] K. Downing, *Artificial life and natural intelligence*, in Proceedings of the 6th Genetic and Evolutionary Computation Conference, Seattle, WA, 2004, The MIT Press, pp. 81–92.

[6] D. Fogel, *Blondie24: Playing at the Edge of AI*, Morgan Kaufmann Publishers, San Francisco, 2002.

[7] L. Fogel, *Artificial Intelligence through Simulated Evolution*, John Wiley and Sons, New York, 1966.

[8] K. D. Forbus, *Qualitative reasoning*, in The Computer Science and Engineering Handbook, 1997, pp. 715–733.

[9] D. Futuyma, *Evolutionary Biology*, Sinauer Associates, Sunderland, MA, 1986.

[10] D. Goldberg, *The Design of Innovation*, Kluwer Academic Publishers, Norwell, Massachusetts, 1989.

[11] F. Gruau, *Genetic micro programming of neural networks*, in Advances in Genetic Programming, J. Kenneth E. Kinnear, ed., MIT Press, 1994, ch. 24, pp. 495–518.

[12] J. H. Holland, *Adaptation in Natural and Artificial Systems*, The MIT Press, Cambridge, MA, 2 ed., 1992.

[13] K. D. Jong, *Evolutionary Computation: A Unified Approach*, MIT Press, Cambridge, MA, 2006.

[14] J. R. Koza, *Genetic Programming: On the Programming of Computers by Natural Selection*, MIT Press, Cambridge, MA, 1992.

[15] ——, *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge, MA, 1994.

[16] J. R. Koza, David Andre, F. H. Bennett III, and M. Keane, *Genetic Programming 3: Darwinian Invention and Problem Solving*, Morgan Kaufman, Apr. 1999.

[17] G. Lakoff and R. Nunez, *Where Mathematics Comes From*, Basic Books, New York, 2000.

[18] C. Langton, *Artificial life*, in Artificial Life: Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems, C. Langton, ed., Addison-Wesley, Reading, Massachusetts, 1989, pp. 1–49.

[19] M. Mitchell, *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, MA, 1996.

[20] J. Roughgarden, *Theory of Population Genetics and Evolutionary Ecology: An Introduction*, Prentice Hall, Upper Saddle River, NJ, 1996.

[21] H. Schwefel, *Evolution and Optimum Seeking*, Wiley, New York, 1995.

[22] L. Steels, *Intelligence with representation*, Philosophical Transactions: Mathematical, Physical and Engineering Sciences, 361 (2003), pp. 2381–2395.

[23] D. H. Wolpert and W. G. Macready, *No free lunch theorems for optimization*, IEEE Transactions on Evolutionary Computation, 1 (1997), pp. 67–82.