Search and Representation in Evolutionary Algorithms

December 7, 2013

1 Introduction

Imagine a (low-budget) mining company looking for rare minerals. They buy up tracts of land and then use a team of diggers (paid by the hour) to locate the mother lode, whose existence is normally promised by the seller of the property. Initially, the diggers are randomly distributed across the area, while a foreman moves about, checking their progress and occasionally relocating diggers.

The goal, of course, is to find the mother lode, but, from the surface, nobody can see it. Each digger serves as a probe into the terrain. If small traces of the mineral are found in a digger’s hole, then that indicates the potential presence of the mother lode in the vicinity, but no probes are conclusive until shovels full of the mineral appear. The foreman makes educated guesses concerning the placement of diggers based on the mineral contents (or lack thereof) of previous holes. For example, if one digger finds 2 grams of the mineral after an hour of digging, and another digger, just to his left, finds 8 grams (in a similar-sized hole), then the foreman might assign a 3rd digger even farther to the left based on these 2 pieces of partial information.

For the foreman directing this search process, the diggers (not the mineral) are his resource. He must allocate them in an intelligent manner to find the target site while incurring the lowest possible labor cost. Many computational search problems have a similar character. Assuming a problem, P, to be solved, educated guesses are made about the solutions to attempt. After being constructed, each solution is evaluated with respect to its performance on P. In essence, the solution is analogous to the precise location of a digger, while the evaluation corresponds to the amount of mineral taken out of his hole.

Typical strategic choices of the foreman include the following:

1. Move a digger from a location that does not appear promising.
2. Command a digger to make a small move in a random direction.
3. Deposit a digger in an area where other diggers have found some of the rare mineral.
4. Deposit a digger at an intermediate location between two diggers whose locations both show signs of a mineral deposit.

The judicious, parallel application of these options should, over time, allow the crew to home in on the mother lode.

In his classic book on genetic algorithms, Holland [5] uses a 2-armed bandit (a hypothetical type of slot machine) to illustrate and analyze search as resource allocation. In this problem, you walk into a casino with a small bucket of
quarters. In the corner stands a bright shining slot machine with 2 arms, the 2-armed bandit. The sign above the bandit says that one of the arms has better odds of hitting winners, but which arm? Your task is to find out which arm is best while simultaneously maximizing your total profit. Your only means of information gathering is pumping quarters into the machine, pulling one arm or the other, and recording the results.

Ideally, you can determine the higher-paying arm early in the process, using a small, but statistically-significant number of quarters. After this exploratory, information-gathering phase, you can commence an exploitative stage by investing all of the remaining quarters in the heretofore best arm. The catch, of course, is that you are never completely certain which is actually the best arm. Thus, there are obvious tradeoffs between exploring and exploiting: the former does allow you to gather more statistically-significant data, but at the price of many quarters invested in the inferior arm; while the latter can maximize profit, but it carries a risk, due to the uncertain identity of the best arm. Holland’s book provides a detailed mathematical analysis of this situation.

In the mining problem, the foreman has similar exploration-exploitation tradeoffs. To explore, he records the findings of each digger but does not move diggers to promising locations, only to new locations (in order to gather more information). To exploit, he uses the recorded findings to distribute diggers to promising spots. As with the 2-armed bandit, the foreman hopes to gather convincing data as quickly as possible (such that exploration can give way to exploitation), and thus to minimize the total digging effort needed to pinpoint the mother lode.

2 Evolutionary Search

In evolutionary algorithms, the individuals of the population are the resource, and we try to allocate them efficiently such that desirable points in design space can be found as quickly as possible, i.e. minimizing the product of the population size and the number of generations. The EA’s selection mechanisms and genetic operators have obvious analogies to the above strategic choices of the mining-company foreman.

The EA search process spans three interconnected spaces: genotype, phenotype, and fitness. Essentially, the EA searches in genotype space, but the trajectory of that search is strongly determined by the phenotypes and fitness values to which they are mapped.

Consider the simple example shown in Figure 1. An EA is used to find a satisfactory assignment of M items, of various weights, to K containers (labelled 0 to K-1), each with a maximum weight limit of W, such that the final weights of each filled container are as close as possible to being equal (i.e., the variance in the weights is minimized).

Assume that each genotype is a bit string of length $M[\log_2 K]$: one gene of length $[\log_2 K]$ for each item, where the gene specifies the container in which to place the item.

A phenotype is then a simple list of container labels, of length M. For simplicity, assume that K is a power of 2, so each $\log_2 K$ substring of bits translates into a unique integer between 0 and K-1.

Many adequate fitness functions are possible, and their details are unimportant at this point, but in general, they will give higher scores to partitions in which a) none of the container weights exceeds W, and b) the variance among the container weights is low.

In terms of the connections between the three spaces, the mapping between genotype and phenotype space will be bijective (i.e., each genotype will map to a unique phenotype, and all phenotypes will be mapped to by exactly one genotype), while that from phenotype to fitness space will be many-to-one, since several phenotypes may be awarded the same fitness.
Figure 1: A simple search problem of assigning objects (balls) to containers (cylinders) so as to a) put all objects in containers without exceeding any container’s weight limit, and b) minimize container weight variance.

Figure 2 shows the 3 spaces and their connectivity for a container problem with $K = 4$. The $n$th pair of bits on the genome encodes the destination container for the $n$th item. The developmental process is a simple conversion of bit pairs to integers (between 0 and 3), as shown at the phenotypic level in the diagram. The phenotypes then map to fitness values in the upper landscape.

Various features of the EA determine the nature of and interactions among these three spaces, and consequently, the difficulty of the evolutionary search. Starting on top, the fitness landscape is probably the most critical of the three. If it is relatively smooth, with gently rolling hills and gradual inclines leading to a summit corresponding to the optimal phenotype, then many search algorithms, not just EAs, will stand a good chance of finding that summit. However, if the summit sits atop a very steep peak that rises straight up from the zero plane, then search becomes a needle-in-the-haystack endeavor. Similarly, if there are many local (but not global) maxima on the landscape, then the search process is easily detoured toward and bogged down at these deceptive sites.

The fitness landscape is merely a visualization of the fitness function, so a landscape with one Mount Everest rising from the plains corresponds to a fitness function that gives no partial credit to any of the phenotypes directly under the plains. Hence, when the EA generates such phenotypes, it literally does not know which way is up and can only guess by searching out in random directions from the zero-fitness phenotypes. Hence, the mining scenario is a more appropriate real-life metaphor for evolutionary search than, say, wilderness navigation. Normally, hikers can see more than their immediate vicinity, and global peaks are often the most visible of all landmarks from almost any viewpoint. Diggers, on the other hand, have holes as their only windows onto the big picture, just as search algorithms only have the evaluations of previously-generated individuals as their patchwork vision of the world. However, as an illustration of a search space and the current status of an algorithm grappling with it, the mountainous fitness landscape is both popular and appropriate.

A landscape with many peaks reflects a fitness function that gives a lot of partial credit to phenotypes. Unfortunately, the generous allotment of credit is not sufficient to insure a successful search. The basis for smooth landscapes with gradual ascents to promising summits is the correlation between phenotypes and fitness. Given a phenotype, $P$, and its fitness $f(P)$, if most phenotypes in the neighborhood of $P$ also have a fitness close to $f(P)$, then the two spaces are
well correlated. And clearly, if search moves a small distance from P, the movement in the fitness landscape will only be a small ascent or descent, if not a neutral movement along a local plateau.

Unfortunately, devising fitness functions that insure this correlation is nearly impossible in many problem domains. Even in the container example, areas of the space are almost guaranteed to have low correlation. For example, assume that the items are ordered from heaviest to lightest. Then, small changes to the last container assignment (i.e., that for the lightest item) will probably not have a great effect upon fitness, since switching the container of the lightest item will not change total weights very much. Hence, in regions of phenotype space where all container assignments are the same (across all phenotypes in the region) except the last assignment, we can expect a rather high correlation between phenotypes and fitness.

However, in regions where only the first container assignment varies among neighbors, we can imagine very little correlation, since the phenotypes vary with respect to their assignment of the heaviest item, and there are surely many situations in which switching the container of the heaviest item can cause a transition from a reasonably good and completely legal solution to one in which a container has a total weight exceeding W. Since most fitness functions would punish such load violations severely, fitness would drop precipitously between neighboring phenotypes.

Assuming that the fitness and phenotype spaces are well correlated, the EA still has no guarantee of success. The mapping between genotypes and phenotypes must also show a smooth correspondence. Although genotypes and phenotypes are equivalent in some types of EAs, they often are not, and a non-trivial developmental process is required to convert genes into traits. Now, genetic operators such as mutation and crossover operate on genotypes, not phenotypes, so when a high-fitness parent genotype is mutated slightly (by, say, flipping a single bit), then if genotype and phenotype space are well correlated, the child should have similar traits to the parent and have a comparably high fitness. Thus, using mutation as a vehicle of change, search can move smoothly about the fitness landscape, without frequent, abrupt dips and hops. Crossover normally incurs major changes to genotypes, so even in a well-correlated
landscape, its does not produce smooth search.

In EAs, mutation is critical for zeroing in on nearby maxima (whose general neighborhood may have been discovered by crossover), but it requires good genotype-phenotype and phenotype-fitness correlation in order to succeed. Once again, superior correlations are hard to guarantee. As the developmental process becomes more complex, the genotype-phenotype correlation often deteriorates dramatically.

However, even trivial developmental links can cause poor correspondence. For example, if bit strings are translated into integers in the straightforward manner of a base-2 to base-10 conversion, then the correlation always breaks down for the higher-order bits. To wit, the genotypes 0001 and 1001 are neighbors, since they differ by a single bit, but their phenotypes, 1 and 9, are quite distant. If these phenotypes represent settings for a thermostat, for example, then 1 and 9 would probably give much more divergent outcomes (and thus a larger fitness difference) than 1 versus 2. So in this case, the lack of genotype-phenotype correlation could easily cause a poor genotype-fitness correlation and make evolutionary search difficult.

Conversely, in a K=4 container problem, 1 and 9 would just represent container labels, so the difference between 1 and 9 versus 1 and 2 would not necessarily mean anything in terms of the actual problem solution and its fitness: moving an item between containers 1 and 9 need not be significantly better or worse than moving it between 1 and 2.

In general, the correlations between these spaces are rarely perfect, and when they are, the problem is usually either trivial or better solved using a traditional search method. EAs are designed to handle the tough cases: those with fitness landscapes containing scattered local maxima and steep peaks jutting out of flat planes, and those with non-trivial mappings between the syntax (i.e., genotype) of a problem solution and its semantics (i.e., the phenotype). When applied to a particular problem, the discussion above gives us concrete quantitative criteria for a) assessing approximately how hard these inevitably difficult searches will be, and b) designing fitness functions and genotypic and phenotypic representations (along with their mapping) so as to make evolutionary search as computationally tractable as possible.

3 Exploration versus Exploitation in Evolutionary Search

As mentioned above, the fitness landscape is a visualization of the fitness function. Unfortunately, for complex phenotype spaces, a complete visualization of the fitness values for each phenotype is computationally impossible. Hence, we cannot simply look at the fitness landscape (in a multi-dimensional space) and eyeball the global maxima. Also, we cannot simply solve the fitness function (the way we solve $x^2 - 3x - 18 = 0$ for $x$) using symbolic techniques to find the optimal phenotype.

Instead, we must perform search, meaning that partial or complete solutions must be generated, tested and modified, with changes taken in directions that appear to lead to improved solutions, i.e., directions that seem to head toward a maximal summit of the fitness landscape.

Whereas classic AI search techniques such as A* deal with partial solutions, evolutionary algorithms work with complete solutions/designs. Also, while A* tends to process these partial solutions serially and independently, EAs work in parallel with many different solutions and often combine them to produce hybrid children.

Search is a partially blind process, since the search module can never see the entire fitness landscape; it only knows the terrain near its attempted solutions. Each such attempt is therefore a probe point into the fitness landscape, and the intelligence in search comes from the judicious choice of these probes such that global optima can be discovered using as few probes as possible.
A good search algorithm strikes a proper balance between exploration and exploitation, typically by beginning in an exploratory mode and then gradually becoming more exploitative. In biological terms, search procedures begin by accentuating variation before gradually moving to a more inheritance-centered strategy wherein *apples never fall far from the tree*.

In general, mutation embodies an exploitative change, whereas recombination via crossover is more explorative. Since EAs often do both, the *frequency* of each genetic operator is often a measure of the degrees of exploration and exploitation. From a very general perspective, all genetic operators can be considered explorative, since they do produce variation, whereas selection strategies embody exploitation by giving priority to the best individuals. But again, this is a very broad-brush approximation, since both genetic operators and selection strategies have tuning mechanisms to adjust their balance of exploration and exploitation. Figure 3 illustrates the broad and more-detailed views of these interactions.

Essentially, selection pressure mirrors exploitation. When pressure is high, the EA strongly shifts focus to the high-fitness solutions, at the expense of the lesser solutions. This often means that one or a few highly-fit individuals produce a large fraction of the next generation. Hence, population diversity drops, which effectively reduces exploration.

![General Relationships Diagram](image)

**Figure 3:** Broad and more detailed views of the relationships between exploration, exploitation, genetic operators and selection.
Figure 4 illustrates exploration versus exploitation for evolutionary search. At time T, the population is reasonably well spread across genotype space and the fitness landscape. An exploratory strategy encourages even further spreading, with no extra emphasis on points near the middle (where a high-fitness genotype was discovered at time T). Conversely, the exploitative strategy shifts more resources (i.e., genotypes) to the areas that have already yielded high fitness values. In terms of EA genetic operators, the explorative strategy may have employed more crossover to produce new genotypes that are distant from the parents, while exploitation might have slightly mutated the two best individuals from time T to produce neighboring genotypes at T+1.

Figure 4: Exploration versus exploitation in evolutionary search. Circles denote the population of (5) genotypes, while stars represent their fitness. This assumes a perfect correlation between genotype and phenotype space such that genotypes produce phenotypes directly above them in the diagram.

To formalize the contributions of exploration and exploitation to evolutionary progress, De Jong [7] (pp. 160-162) includes an enlightening description of Price’s classic equation:

$$\Delta Q = \frac{Cov(z, q)}{\bar{z}} + \frac{\sum z_i \Delta q_i}{N\bar{z}}$$  \hspace{1cm} (1)

Here, $\Delta Q$ is the change in some trait/quality, Q, over the entire population, while $q_i$ is the value of that quality in parent
The trait is assumed to be graded, to some degree, so that individuals will have more or less of the trait, with $q_i$ quantifying that amount. Also, $\Delta q_i$ denotes the difference between the average $q$ value in parent $i$’s children and $q_i$.

As mentioned earlier, fitness in evolutionary biology is normally measured in terms of reproductive success, so the fitness correlate in Price’s equation is $z_i$, the number of children to which parent $i$ contributes portions of its genotype; $\bar{z}$ is the population average of the $z_i$. Also, $N$ is the population size, and Cov($z$,q) is the covariance of $z$ and $q$, as defined by:

$$Cov(z,q) = \sum_{i=1}^{N} \frac{(z_i - \bar{z})(q_i - \bar{q})}{N}$$

In a nutshell, the covariance will be a high positive value if there is a direct relationship between reproductive success and the degree of trait $q$. It will be a large negative value if the two are inversely related, i.e. many individuals with high $q$ values produce few offspring. It will be close to zero if there is no clear relationship between $q$ values and reproductive success.

The first term in Price’s equation 1 denotes the selective pressure, since the covariance of $q$ and $z$ indicates the degree to which possession of the trait correlates with reproductive success. When selection pressure is low, any value of $q$ could yield just as many offspring as any other value, which would be reflected in a low Cov($z$,q) value. Conversely, when selection pressure is high, certain traits (in Price’s model, high values of $q$) give a definite reproductive advantage over others, reflected in many (high-$q$, high-$z$) pairs in the population, and thus a high Cov($z$,q) value.

The second term in 1 captures the degree to which children differ from their parents, with respect to trait $q$. It therefore represents population variability from generation to generation, the opposite of inheritance.

Clearly then, Price’s equation shows that the rate of evolutionary change is directly proportional to the combination of selective pressure and variability. If both are high, evolutionary change should be swift, but if either is low, the rate is reduced.

Consider each of these options. If variation is high but selection low, then many new genotypes (i.e., many diverse $q$ values) are produced, but all receive approximately equal priority in terms of reproduction. Hence the population will not deviate much from some average value of $q$. Conversely, if selection is high but variation is low, then an initial population (of presumably similar $q$ values) will not diversify. Thus, selection, no matter how strong, will have very little to favor, since all $q$ values will be similar.

For evolution to attain optimal, but hard to reach, areas of a search space, it needs to progress through many rounds of ruthless filtering of high-variance gene pools. Each filtering round moves the population’s center of gravity to a slightly higher point, from which many exploratory moves are taken by the genetic operators, with only the best such moves resulting in useful new points, to which evolution shifts its focus and produces new feelers in search space.

The assumption of homogeneous initial $q$ values normally holds in nature but not in EAs, which are often initialized with random genotypes. However, even with a diverse gene pool as starting point, evolution will stagnate under conditions of high selection and low variance. To wit, high selection will initially favor one or a few of the slightly better genotypes, causing many of the new offspring to have $q$ values within a tight range. From there, exploration will be quite slow due to low variation. In effect, selection forces the population to converge to homogeneity so early in evolution that the favored individuals probably only represent local maxima in the search space; evolution then gets stuck there due to low variation.
4 Genotype-Phenotype Mappings

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 automata 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.

EA researchers often differentiate between two general classes of genotype-phenotype mappings: direct and indirect, although details of the distinction often vary. These are essentially synonymous with direct/indirect encodings or representations; the genotype represents the phenotype since the former recreates the latter when processed by the mapping function.

A direct EA representation is one in which the genome can be divided into several components (i.e. genes), where each such segment independently encodes some aspect of the phenotype, e.g. a trait, and there is a 1-1 relationship between each gene and trait. For example, in the bin-packing problem above, each gene is a bit string that encodes the container for a particular ball. These are independent, since the placement of any ball (a trait) does not affect the ball-placement encoded by another gene. In terms of genetic operations, any mutation to one gene will only affect its own translation, not that of other genes. So independence has both developmental and evolutionary significance.

If the genome for a direct mapping always encodes a value for each phenotypic trait, then the representation is complete; these representations typically involve fixed-length genomes. On the other hand, a partial direct encoding has genomes that only denote some traits of the phenotype (e.g. the weights of only some connections in a neural network); partial-direct genomes in the same population will often have different lengths. In either case, the genes still determine traits independently.

Complete encodings can take advantage of genome position to simplify their syntax, whereas partial encodings cannot. For example, if the genome encodes the open/closed settings for N gates in a flow-routing problem, then under a complete encoding, the kth bit on the chromosome can denote the state of the kth gate, so the whole genome could be k bits long. Alternatively, with a partial representation (where, for example, all unmentioned gates are assumed closed), each gene would require both a gate index and a binary state.

Indirect mappings can be either bijective or generative. Like a complete direct mapping, the bijective indirect genome has one entry for each trait, but independence is no longer guaranteed. Hence, one gene-to-trait translation can depend upon another gene/trait. For the bin-packing problem, one possible bijective indirect genome has the same number of genes and bits as does the direct representation, but now, if the ith gene encodes the integer j, then the semantics is place the ith object in the first bin that can accommodate it, beginning with the jth bin, ascending thereafter, and possibly looping back to the beginning of the bin list. Clearly, the placement of low-index objects can affect the decoded bins of higher-index objects, and single mutations can have widespread consequences for the phenotype: the final placement of all objects.

Finally, a generative indirect representation guarantees neither a 1-1 mapping nor independence. Typically, these involve relatively small genotypes that produce much larger phenotypes. For bin-packing, a generative representation may encode a few parameters that control key decisions in a packing algorithm. For instance, assume an algorithm that takes an object (of size S) and the remaining space in each bin as input. It then computes the ratio of S to available space for each bin and then sorts these ratios in descending order. It then removes all ratios greater than 1 to produce a priority list of length L (where L ≤ K, the number of bins). The evolved parameters of this algorithm are K probabilities, which are normalized to sum to 1. Since each object can yield a priority list of a different length (due to the filtering of containers with insufficient space), the algorithm must combine the K probabilities into L numbers (that still sum to 1). By spinning a simulated roulette wheel, the algorithm can then choose among the L possible bins.
in a stochastic manner weighted by the evolved parameters. Since \( K \) is normally quite a bit less than \( M \) (the number of objects) for interesting bin-packing problems, the length-\( K \) genotype produces much larger (size-\( M \)) phenotypes.

These categories serve a useful purpose in supporting a comparison of genotypes with respect to the amount of effort required to convert them into phenotypes. However, a universally useful classification of EA representations seems far-fetched, since many thousands of EA applications (with specialized representations) exist, and different perspectives can be important in analyzing them.

While the above formulation keys on the relationship between genotype and phenotype, other classifications focus on one or the other. For example, De Jong [7] uses the syntax of the genotype as the defining property to produce 4 main classes:

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 that require 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.

Another alternative classification is based on the semantics of the information that the genotype contributes to the phenotype. It consists of two main genotypic categories:

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).

This perspective, along with the direct/indirect classification of mappings, is central to this book’s discussion of EA representations.

### 4.1 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 syntactic representations for genotypes and their corresponding mappings functions. 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 011 = 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 is 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.

Indirect bijective 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 elaborate, 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.
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.
4.1.1 A Robot Example

To illustrate the wide variety of 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.

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

Assume that the robot has 8 color sensors, 4 each for red and blue, 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 the carried object down in front of the robot.

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

\[ BC \land RR \implies TR \]

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

\[ BC \land \neg BB \land \neg RB \land RF \implies MB \land DROP \]

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

\[ RL \land \neg RR \land \neg BR \land \neg BC \land \neg RC \implies TR \land MF \]

*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 an 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 one 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 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
Sensory Input Gene (5 bits)

<table>
<thead>
<tr>
<th>T or F? (1)</th>
<th>Red or Blue? (1)</th>
<th>Direction or Carry? (1)</th>
<th>Direction (2)</th>
</tr>
</thead>
</table>

Action Gene (6 bits)

<table>
<thead>
<tr>
<th>Turn Left? (1)</th>
<th>Turn Right? (1)</th>
<th>Go Forward? (1)</th>
<th>Go Backward? (1)</th>
<th>Pickup? (1)</th>
<th>Drop? (1)</th>
</tr>
</thead>
</table>

Rule Segment (5K+9 bits)

Number Input Terms = K (3 bits) K Input Genes (K x 5 bits) Actions (6 bits)

Translating a Rule Segment

2 input terms

010

11101

00111

101001

T Red Dir Back

F Blue Carry Ignore

MF & TL & DROP

Complete Genotype: Set of Rules

3 5K 6 3 5K 6 3 5K 6

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 $RB \land \neg BC \implies MF \land TL \land DROP$. 
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. Though theoretically possible, these enumeration-based encodings become impractical for domains involving either many inputs or many distinct values for each input. The current robot example, however, lies well within their scope.

For the lower encoding strategy of Figure 9, 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 000001110 (i.e., 14 in binary), which denotes the conjunction:

$$\neg RF \land \neg RB \land \neg RL \land \neg RR \land \neg BF \land \neg BB \land BL \land BR \land RC \land \neg BC$$  \hspace{1cm} (6)
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:

\[ RB \land \neg BC \implies MF \land TL \land DROP \]  \hspace{1cm} (7)

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 7. One such rule is (with the core intensional component in boldface):

\[ RB \land \neg BC \land RF \land RL \land RR \land BF \land \neg BB \land \neg BL \land \neg BR \land \neg RC \implies MF \land TL \land DROP \]  \hspace{1cm} (8)

and another is:

\[ RB \land \neg BC \land \neg RF \land \neg RL \land \neg RR \land \neg BF \land BB \land BL \land BR \land RC \implies MF \land TL \land DROP \]  \hspace{1cm} (9)

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 7 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 \(MF \land TL \land DROP\)) 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 be molded into fixed-size 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 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 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.
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.
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 $RL \land \neg RL$ 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.

### 4.1.2 A Neural Network Robot Controller

Although evolutionary algorithms constitute fundamental approaches to Bio-AI, the above robot controllers employ a classic GOFAI representation: a set of if-then rules. This is one way to (relatively seamlessly) blend the two AI paradigms, and because you can evolve just about any aspect of an AI system, this type of hybrid is commonplace. In contrast, evolving artificial neural networks (EANNs) are a purely Bio-AI approach.

Consider the ANN of Figure 10, again used to control the cylinder-moving robot. This receives a binary vector of length 10 on its input layer (open circles), one bit for the boolean value of each sensory precondition. These values are weighted and fed forward to the hidden layer (dark circles) and output layer (pentagons), with the activation levels of the output neurons determining the actions taken by the robot. For example, all output nodes above a given threshold
may trigger their corresponding action, with conflicts (i.e. turn left and turn right) resolved in favor of nodes with higher activation levels.

The network may contain other connections, such as feedback links from the output to the hidden layer, but such details are irrelevant for the current discussion. The main point is that several parameters (i.e., the connection weights along with a few factors associated with the activation functions) will determine its decision-making logic. In theory, given a sufficient number of hidden nodes, this ANN can realize any of the robot rule sets described above. However, the precise combination of these parameters needed to mirror a given rule set can be very difficult to design by hand; this search for a proper parameter vector often calls for an evolutionary algorithm.

Given a fixed ANN topology, the size of this parameter vector is also fixed. Hence, the EA chromosome is simply a long array of values (using one of the many direct encodings mentioned above), with no need for padding to prevent the widespread disruption of semantic content by a single mutation. However, the Evolving ANN (EANN) research community often has problems with crossover: either they omit it completely, or they restrict it to certain chromosomal locations.

Under the common view of a neuron as a detector of particular patterns (on its afferents), the weights on all incoming arcs combined with the properties of a node’s activation function constitute a module that essentially defines each detector. These are the building blocks of the EANN chromosome, and once effective subset vectors arise, a good EA will maintain and combine them, although splitting and merging have more deleterious effects. Hence, many EANN systems include constraints such that crossover only occurs between (not within) these modules.

![Figure 10: A sample ANN for controlling the robot of Figure 6. Only some connections are shown, and all connections have weights (W).](image)

These ANNs serve as intensional representations of a rule set. To derive the extension, simply input all 1024 sensory cases and record the resulting outputs. The beauty of ANNs is their ability (when combined with the right search technique) to formulate parameter vectors that perform complex mappings from inputs to outputs. The rule-based extensions of these mappings often consist of many specific rules whose combination embodies both general and case-specific behavior, akin to human knowledge. However, it is typically not the same rule set that a human engineer would formulate and, consequently, ascertaining the knowledge content can be very difficult.

In Figure 11, the given weights, firing thresholds and activation functions are those that a person might use to mimic the 4, robot-controlling, if-then rules shown above. For example, the rule \((RR \lor RL) \land BC \rightarrow DROP\) is realized by the nodes RR, RL, BC, c, f, and DROP (along with the links between them). Remembering that all input-layer nodes produce only 1’s and 0’s, the weighted input to node c will be either 0, 1 or 2. Since c has a firing threshold of 1, it will fire when one or both of RR and RL are active: when their disjunct is true. Node f is simpler and fires if and only
if BC is true. Finally, the output node DROP, with incoming weights of 0.5 and a threshold of 1, will only fire when both c and f have fired: it detects their conjunction.

Note that the weights shown are just a small fraction of the total network; all others are presumably zero. This constitutes the purest and simplest continuous-valued parallel to the set of 4 discrete rules. However, there are an infinite number of continuous alternatives to the discrete solution, each containing many more non-zero weights and non-unitary firing thresholds. A search algorithm operating in this continuous space could home in on any of them and return it as an optimal result.

![Diagram](image)

Figure 11: Weights (labels on edges) and firing thresholds (numbers in diamonds) that provide the same functionality as the set of 4, if-then, robot rules given earlier. All firing thresholds in the middle layer are 1.0, and all undrawn connections are assumed to have weights of zero. The activation function for each neuron is a step function that produces a 1 if the sum of weighted inputs is greater than or equal to the firing threshold, and a 0 otherwise.

Figure 12 displays one such network in which the dotted connections provide noisy additional consequents to each of the 4 rules. However, each has a small weight that a minor firing-threshold change can counterbalance. For example, RB and BB provide noise to node c; neither alone nor in combination can they fire c, but they can make it more difficult for RR or RL to activate c. Lowering c’s threshold to 0.9 effectively negates this noisy inhibition. A corresponding threshold reduction is also needed at DROP to maintain the integrity of the original rule.

The plethora functionally-equivalent solutions allows evolutionary search to find something useful, though probably not the needle-in-a-haystack solution of Figure 11. This requires very little special treatment in the chromosome, other than the above-mentioned (though not completely necessary) restriction of crossover points to module boundaries. Conversely, to bias search toward the purest continuous solutions, with, for example, a minimum of non-zero weights and non-unity firing thresholds, would require a fitness function with various penalties for impure values and/or complex restrictions upon mutation and crossover. In short, EANNs can find intensional solutions using relatively simple genotypes and genetic operators, due to the nature of this solution-rich, non-discrete, search space. ANNs may not be easy to understand, but useful ones need not be difficult for a search algorithm to find.

### 4.2 Program-Oriented Genotypes

In the robot example above, the genotype encoded either a set of if-then behavioral rules or a collection of ANN parameters. Thus, the genotype formed the basis for a rule-based or ANN-based controller. However, the genotype did not contain the if-then structure nor the neural integrate-and-fire architecture; those decisions are made by the EA designer, who interprets bit segments as if-then rules to be run by a rule-based system kernel or as weights and
Figure 12: A functionally equivalent network to that of Figure 11, but one whose behavior is much less transparent.

thresholds to govern an ANN.

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 [8, 9, 10] 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 ground-breaking 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: \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\). 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 13 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).

![Diagram of a genetic program tree]

\[
\text{(lambda (X) (+ (* (* (* X X) X) (+ 3 3)) (+ (* -2 X) (- -3 2))))}
\]

Figure 13: (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.

When a randomly-generated GP tree (call it GPT) combines the above functions and terminals, and is then wrapped within a lambda expression to form \((\text{lambda} (X) \text{GPT})\), it will compile and run as long as the following constraints have been respected:

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, this 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 14, merely swaps random subtrees of two GP trees. The closure property guarantees that the results of mutation and crossover are valid programs.

![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.](image)

Figure 14: 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.

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 = ∅
- i = random index into the code vector, V.
- while count ≠ -1 do
  - count = count - 1 + arity(V(i))
  - append V(i) onto end of S
  - i = i + 1
Here, the arity of a function is the number of arguments that it takes, while the arity of a terminal is 0. Figure 15 shows a simple example of a subtree hunting in the program from Figure 13. 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.

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 16 shows a GP program of similar functionality to 4 sense-act rules:
1. \( (RR \lor RL) \land BC \implies DROP \)
2. \( RF \implies MB \)
3. \( \neg RF \implies MF \)
4. \( BF \implies PICK \)

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

Although a more complete comparison of GP and the other EAs appears in textbooks [1, 7], 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.

5 Conclusion

Just as the natural world depends upon evolution, Bio-AI relies heavily upon evolutionary algorithms. Though other Bio-AI tools, such as ANNs, swarms [2] and Lindenmayer systems [11] have, strictly speaking, no necessary tie to EAs, they all invoke emergent processes whose outcomes defy accurate prediction. Hence, the task of hand-designing local behaviors to generate target global patterns often becomes rather daunting, as does the formulation of intelligent, solution-building tools. In many cases, the trial-and-error combination of primitive components is the only feasible approach, with EAs as a popular alternative.

As proven mathematically by Holland [6], EAs skillfully manage the combination of exploration and exploitation, though the true optimality of this balance is highly representation dependent, pertaining mainly to direct encodings. Unfortunately, many evolutionary design problems require indirect encodings, moving us well outside the realm of guaranteed feasibility; crafting genotypes, phenotypes and their mappings becomes a very creative endeavor. As shown above, different syntactic choices can reduce phenotypic sensitivity to mutation such that inheritance remains relatively high, and the EA can methodically exploit a promising region of the search space. However, many of these options add a lot of extra baggage to the genotype in order to preserve the generality and expressibility of an intensional phenotype. Switching to an extensional phenotype alleviates many of these problems, but at the expense
of expressibility, since certain general traits become highly unlikely, though not impossible, to evolve. In restricted
domains, all preconditions, of both general and specific nature, can be enumerated, thus relieving many potential
problems at the genotype level while maintaining intensionality. However, these scale very poorly.

Artificial neural networks manifest quantitative reasoning engines at a level well below that of rule-based systems.
They can serve as generators of extensional rule sets, but the genotypes used to produce ANNs can employ direct or
indirect encodings. By evolving ANN parameters (e.g. weights, thresholds, etc.), the EA can retain a simple chromo-
some with direct encoding and standard genetic operators, while producing a structure that exhibits both general and
highly context-dependent behavior, without the need to enumerate all possible rules when defining the genotype (as in
Figure 9). As shown later, the direct encoding of all ANN parameters also becomes impractical for complex domains,
but the general approach of encoding a quantitative, behavior-generating machine (instead of discrete logical rules) in
the genome has considerable merit.

Genetic Programming conveniently supports the evolution of both a) intensional rule sets, by allowing myriad com-
bined combinations of primitive elements in preconditions and actions, and b) the control logic itself. It epitomizes the spirit of
Bio-AI by combining basic building blocks into an infinity of emergent designs. GP raises genotypes to the pheno-
typic level, a decision that would normally require a host of special-purpose, genetic operators and impose a strong
search bias just to insure syntactic validity. However, the homogeneous syntax of functional programs (in concert with
features such as closure) allows generic, tree-based, mutation and crossover mechanisms to produce a wide variety of
legal phenotypes. GP can even evolve programs that, when run, generate the topologies and weight sets for ANNs [4].

As in GOFAI, representational choice governs the success or failure of design search in Bio-AI. What confounds the
issue is the extra representational level of the genotype. Whereas GOFAI normally employs hand-crafted phenotypes
(that may adapt via learning), Bio-AI drops a level and searches in a purely syntactic world to find genotypes that
translate into successful phenotypes (that may also undergo learning). This is nature’s way, not necessarily AI’s; but a
long string of EA success stories [1, 3] testify to the wisdom of this general approach to computational creativity.

References

On the Automatic Evolution of Computer Programs and its Applications, Morgan Kaufmann, dpunkt.verlag,
1998.


1995.


Invention and Problem Solving, Morgan Kaufman, Apr. 1999.