Selected Applications of Evolutionary Algorithms

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

July 14, 2006

1 Introduction

The world of evolutionary computation is vast. A dozen or more annual international conferences are either dedicated to EAs or have special tracks for them. This amounts to a thousand or more EA papers each year, many of which involve novel applications. Hence, to write a **comprehensive** survey of EA applications would be a career in itself (for a small company).

The purpose of this section is much more modest: to give the reader a basic feel for EA usage in a variety of domains. The focus will be on the representations and fitness functions of these application domains with the dual intention of a) highlighting these creative attempts to model everyday problems in terms of genotypes and phenotypes such that evolution can solve them, and b) provide the reader with a few useful (and some classic) cases that may assist in her own EA problem solving.

2 Spook versus Spy

Dennis Shasha, a professor of computer science at the Courant Institute of New York, contributes a monthly puzzle column to Scientific American. His August, 2003 puzzle provides a nice toy-problem testbed for EAs.

In this *spy puzzle*, a CIA official (i.e., *spook*) attempts to send a series of N messages within a given time limit, T. The messages have different lengths, and their sending intervals can overlap with no limit on the number of parallel broadcasts. A spy is known to be tapping radio signals, but since (s)he must enter the sender's line of sight to steal signals, the spy must limit tapping to a single K-minute interval. To tap a message, the spy must intercept it from start to finish. The spook is willing to have at most M of the messages tapped.

The problem is to schedule the sending times of the N messages so that no k-minute interval between times 0 and T-1 fully contains more than M messages. Clearly, many versions of the problem exist, depending upon N, M, K and the message durations.

Consider a set of 7 messages of durations 2,3,4... 8 time units, as given in Shasha's article, with T = 15, K = 10 and M = 3.

A straightforward EA to solve this puzzle uses a genotype with 7 genes, one for each message. Each gene simply encodes the start time for its message. Whether a binary or integer gene representation is used makes little difference for this problem.

Each phenotype is a complete sending schedule for the messages: a list of sending intervals. These intervals are formed by forming the pairs (s, s+d-1) for each gene, where s is the start time and d is the message duration. For example, (3 7) is a 5 time-step message.

To assess fitness, calculate the *vulnerability*, V, of a phenotype by moving a 10-unit time window along the time axis from (0,9) to (1,10), to,...,to (26 35). At each location, calculate H (*hits*), the number of messages that would be fully intercepted if the spy were to perform the tap during this exact interval. If $H > M$, then increment V by the amount H-M.

After all tap windows have been tested, V should give a good indication of the sending schedule's total vulnerability, which should be inversely related to fitness. A simple fitness function is thus:

$$Fitness = \frac{1}{1+V} \tag{1}$$

Figure 1 shows one solution that is easily found using an EA with bit-vector genotype and a population of only 5 individuals. Adult selection was type A-I (full generational replacement) and mate selection was fitness-proportionate, although just about any setting for these would work on this simple problem.

Figure 2 shows the progress of evolution for this puzzle. This *fitness plot* is the most common graphic in EA documentation. This version includes the population's maximum, minimum and average fitness for each generation. Unfortunately, EA runs do not always always show this steady step-wise progression. There are often long periods of stasis with only occasional jumps in maximum fitness. This is similar to the *punctuated equilibria* theory of evolution.

To make the puzzle a bit more challenging, Shasha suggests adding 3 more 4-unit messages and only increasing the maximum time to 20 units. For this version, it helps to have a slightly larger population of 20.

Figure 4 shows a similar progression to the previous example. Incidentally, most attempts at this harder problem with a population size of only 5 were unsuccessful. A solution appears in Figure 3.

Finally, Shasha adds a final twist by returning to the original 7-message problem but a) reducing the number of acceptable intercepts from 3 to 1, and b) increasing the total time from 15 to 36

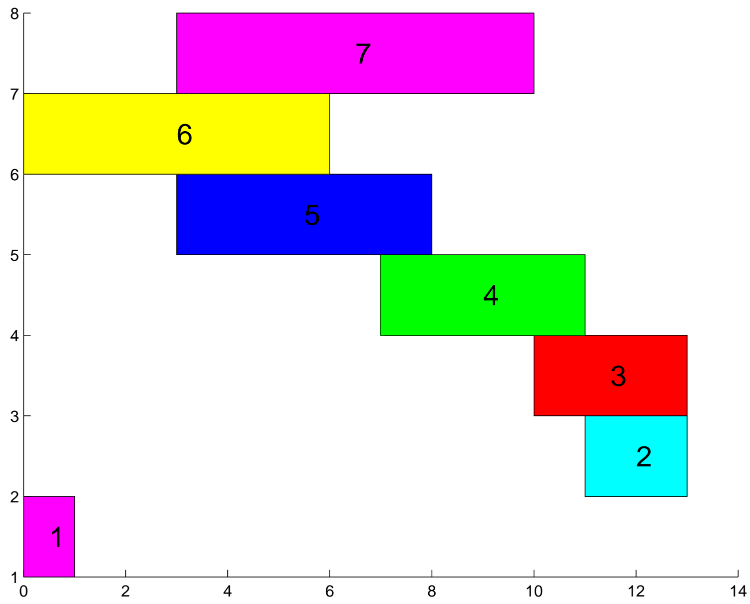

Figure 1: A solution to a 7-message spook-versus-spy problem with M=3 and K= 10.

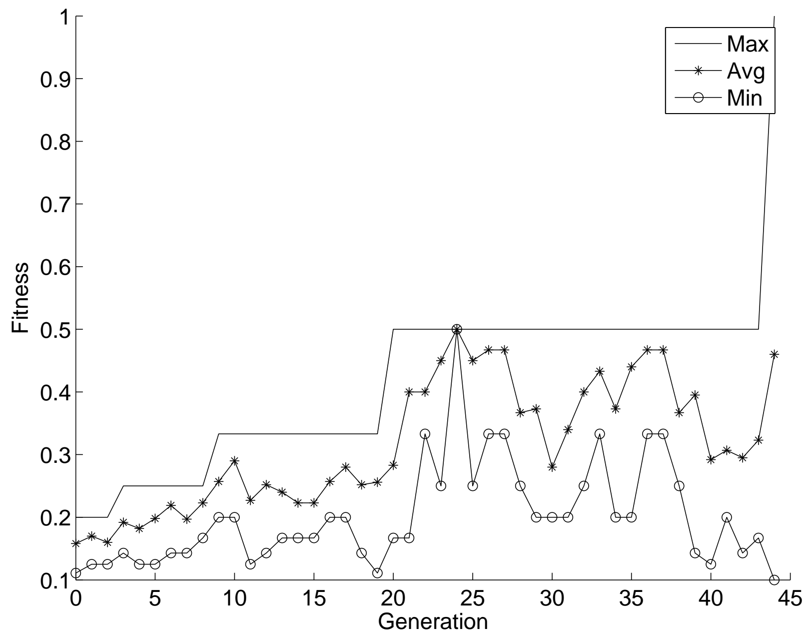

Figure 2: Evolutionary progression for the 7-message spook-versus-spy problem with M=3 and K=10.

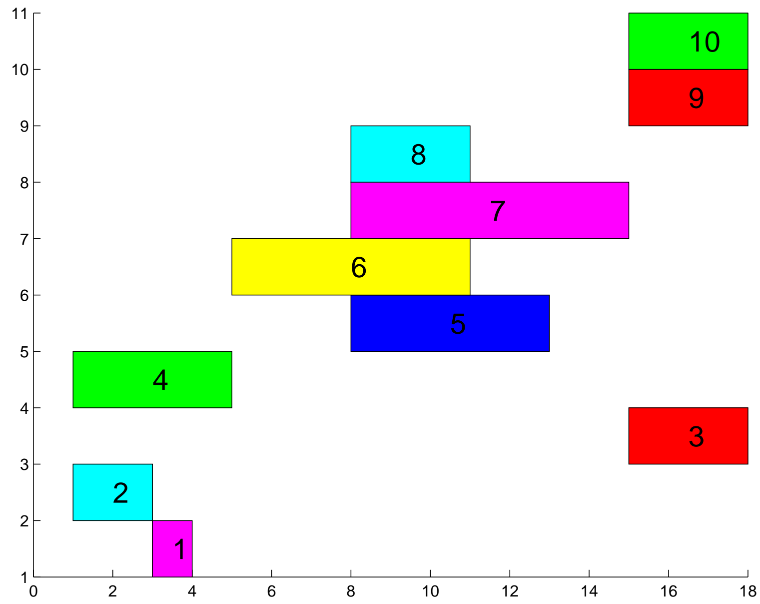

Figure 3: A solution to a 10-message spook-versus-spy problem with M=3 and K= 10.

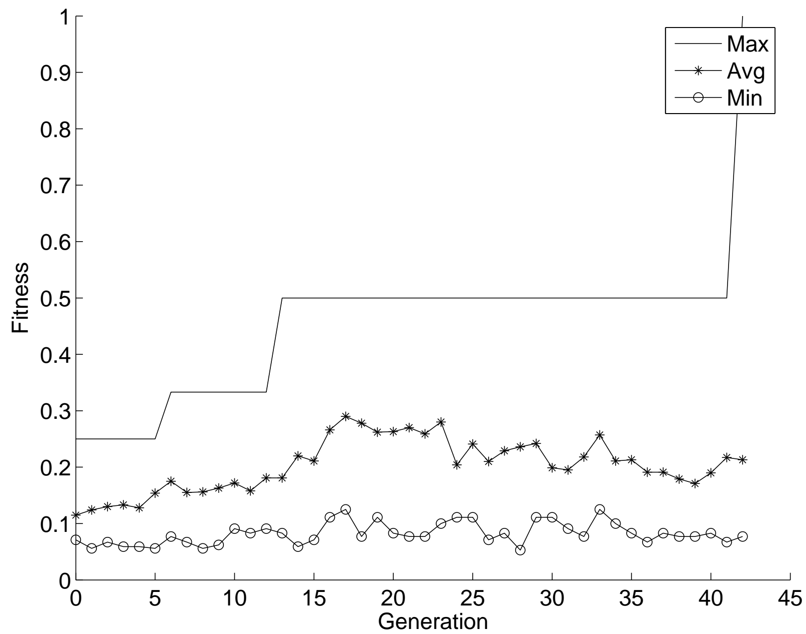

Figure 4: Evolutionary progression for the 10-message spook-versus-spy problem with M=3 and K=10.

time units. This proves very difficult for the above EA, even with a population of 1000. Shasha's solution is shown in Figure 5.

In rough fitness landscapes with only one or a few global optima, even an EA may fail unless either a) it gets lucky, or b) a fitness function can be devised to help smooth the landscape and provide partial credit that helps direct search toward the highest peaks.

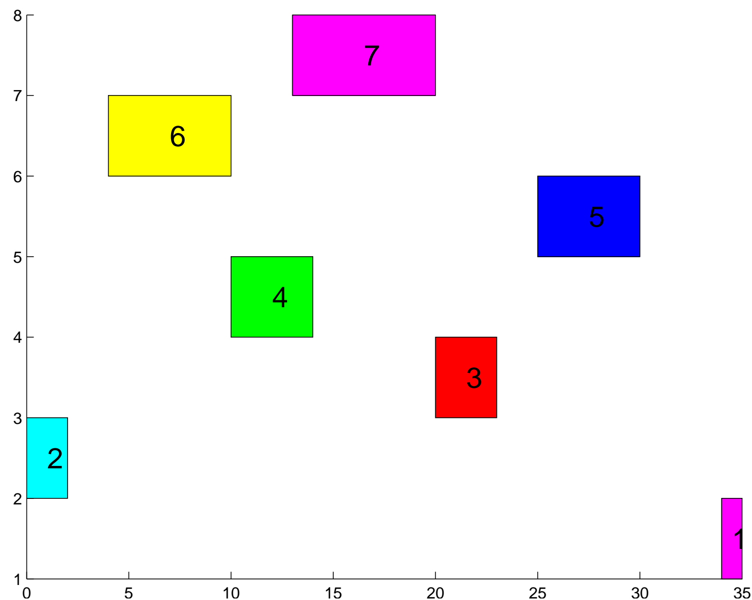

Figure 5: Shasha's solution to the 7-message spook-versus-spy problem with **M=1**, **T = 36**, and K= 10.

3 Travelling Salesman Problem (TSP)

The TSP is an all-time classic in computation theory [4]. Thousands of TSP solution techniques exist. Some are *brute force*, meaning that they methodically sweep the entire search space (in the worst case), thus guaranteeing an optimal solution. Unfortunately, as the size of a TSP increases, this guarantee can include the assumption that a thousand supercomputers will run for a thousand years to find it!

Although often called optimization tools, EAs can rarely guarantee an optimum, since they are decidedly not brute-force. EAs are more accurately termed *satisficing* problem solvers, since they can often find very good solutions. When solving TSPs, EAs often find the global optimum, but not always.

The TSP is easy to state but hard to solve. A salesman needs to drive to N different cities, returning to the start city at the end. No city can be visited more than once. In what order should he visit the cities in order to minimize his total driving distance? TSP, and the related Hamiltonian Circuit Problem [4], have a wide applicability in areas such as network design, delivery routing and

sequence scheduling, but our discussion will be restricted to cities and tours.

For any collection of cities, real or fictitious, if a complete table of distances between each pair of cities exists, then a TSP can be formulated for these cities. Small collections often support a TSP that can even be solved by hand, especially if the city coordinates are first plotted on a map. However, with larger city sets, the corresponding TSP often becomes extremely difficult.

To solve TSP with an EA, start with the phenotype and ask what the general structure of a solution will be. The answer is simply a list of cities corresponding to the salesman's tour. That is, if a_i and a_{i+1} are adjacent city names in the list, then the salesman will be driving the $a_i \Rightarrow a_{i+1}$ section of road without any intervening stops.

An N-city TSP genotype must therefore encode a city tour such that all N cities appear exactly once. Already, this simple constraint causes a few problems compared to the spook-vs-spy problem. Consider a simple bit-vector genotype, with N segments, each segment of length $\lceil log_2 N \rceil$. For ease of explanation, assume that N is a power of 2. Thus, each segment encodes a city index (between 0 and N-1). By converting each bit segment into a city index, an N-city tour is formed. Unfortunately, this method cannot guarantee a *complete tour*: a tour in which every city is included exactly once.

Of course, the initialization process of genotypes for generation 0 could enforce a completeness constraint such that all initial tours are complete, but then the genetic operators would have to check for uniqueness every time a bit was mutated or two bit vectors were crossed over. That's a lot of checking, modification and re-checking!

A better approach involves moving the genotype to the phenotype level and simply encoding the genotype as a permutation of the integers 1..N. The genetic operators would then work directly with integer lists - no bits.

However, the genetic operators must operate differently than at the bit level. First, mutation cannot simply change one integer to another, since this would destroy the completeness of the tour. Similarly, unchecked crossover of two permutations cannot guarantee a complete tour. For example, assuming N = 4, the parents are 1234 and 4231, and they are crossed in the middle, then the children are 1231 and 4234, neither of which is complete.

Defining mutation for permutations is easy: swap elements. So one possible mutation of 1234 would swap 1 with 4 to yield 4231.

Crossover is trickier. Many versions now exist for permutations, but one of the earliest, Partially-Mapped Crossover (PMX) [6] is easy to implement and still in wide use today.

Given two parent permutations, PMX does the following:

1. Copy each of the parents to produce children C1 and C2.
2. Select two equal length segments, S1 and S2, from C1 and C2, respectively, and swap them.
3. Use corresponding elements in S1 and S2 as the *map*.

4. For any item,X, in C1-S2 that is also found in S2 (i.e., a duplicate), use the mapping between S2 and S1 to find a new value, X*, that is not in C1. Do the same for all duplicates in C2.

The PMX algorithm is illustrated in Figure 6. Notice that after segment swapping, some items in the original (unswapped) portion will be redundant. The mapping is then used, often repeatedly, to find a unique new integer, i.e., one not in the newly swapped-in segment nor in other parts of the original.

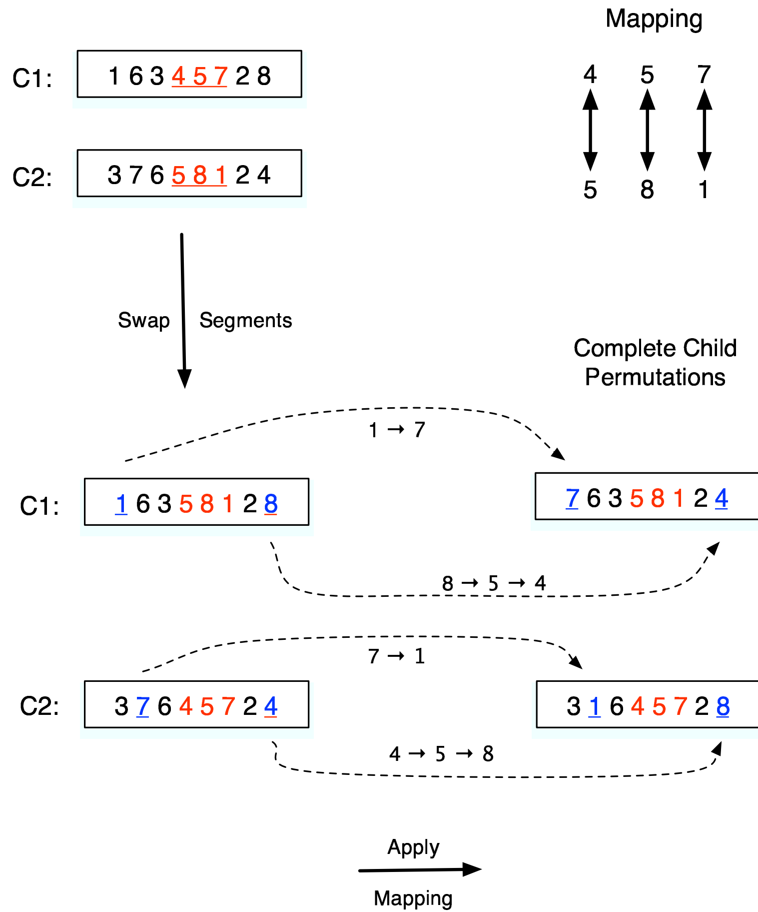

Figure 6: An example of Partially-Mapped Crossover (PMS). In the upper left, the underlined numbers are the segments to be swapped. In the lower left, the underlined numbers are redundant and must be replaced using the mapping.

With mutation and crossover defined for permutations, the EA can use standard modules for the rest of the job. A typical fitness function computes the total distance of the tour and uses its inverse as the fitness value. Since these tours are often thousands of kilometers long, the fitness function will often scale the total distance to avoid very small fitness values. In this example, we use the following:

$$fitness(tour) = \frac{1}{\sqrt[3]{distance(tour)}} \tag{2}$$

Figure 7 shows the fitness progression of an EA with permutation genotypes and PMX crossover working on a 29-town TSP from the Sahara Desert. The EA uses a population of size 500. It does find the optimum, 27603 kilometers, although repeated runs do not always do so; many satisfice by coming within 4-6% of optimum.

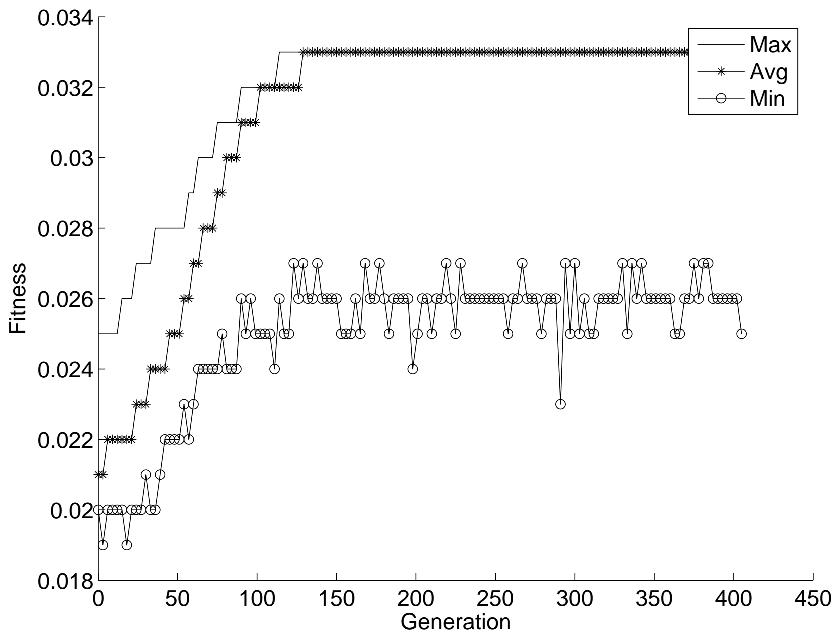

Figure 7: Fitness plot for a 500-individual EA using a permutation genotype and PMX crossover on the 29-city Sahara Desert TSP.

3.1 An Indirect Binary TSP Representation

Despite the problems with binary TSP representations discussed above, a different interpretation of a binary genome does permit its usage. The bits cannot be directly decoded as cities, but indirectly, as indices into a stack of unused cities, as shown in Figure 8. This scheme allows genomic redundancy (i.e., several of the indices can be identical). Thus, the standard mutation and crossover operators for bit vectors are valid, and the only specialized code is for translating the bit vectors into city tours.

Figure 9 shows the fitness plot for a 500-individual EA using this indirect representation. The performance is lower than that of Figure 7. Although the indirect representation is a nice trick, it has practical weaknesses due to the strong interdependence of its genes. In biology, this interaction of genes is known as *epistasis*.

Epistasis in the indirect genome is clearly evident. If a single index is mutated, say from 2 to 6,

then all indices that are a) to its right on the genome, and b) between 2 and 5 (inclusive), will have a different meaning, in that they will translate into different cities than before the mutation. Hence, one small change has large effects on the phenotype. Although desirable at times, this phenotypic oversensitivity to genotypic change - an indication of a poor correlation between genotype and phenotype space - can hinder evolutionary progress.

Furthermore, heritability often suffers when indirect representations undergo crossover. Consider the following simple example of two parents in a 6-city TSP. On the left is their indirect representation; on the right is the city tour (i.e., phenotype):

P1: (1 1 1 1 1 1) $\Longrightarrow$ (A B C D E F)

P2: (2 3 2 3 2 3) $\Longrightarrow$ (B D C F E A)

Performing single-point crossover after the 2nd integer produces two children:

C1: (1 1 2 3 2 3) $\Longrightarrow$ (A B D F E C)

C2: (2 3 1 1 1 1) $\Longrightarrow$ (B D A C E F)

To quantify the degree to which a child, C, resembles P1 and P2, calculate:

- *Position Inheritance:* The number of cities in C which appear in the same location as in P1 or P2.
- *Edge Inheritance:* The number of inter-city edges in C that appear in either P1 or P2.

In C1, cities A, B, E and F appear in the same locations as in P1 or P2, so position inheritance is 66% in C1, and, coincidentally, in C2 as well (via cities B, D, E and F). Thus, $\frac{2}{3}$ of the child positions come from the parents, which is a reasonably high level of inheritance.

However, the essence of a tour is not the positions of its cities, since for any N-city TSP there are N different orderings for exactly the same tour, e.g., (A B C D), (B C D A), (C D A B) and (D A B C) when N = 4. The essence lies in the inter-city links/edges, since the sum of these constitutes the solution's cost. The inheritance of edges is therefore a much better indicator of the ability of parent solutions to pass on **useful** traits.

In the example above, the edges found in the parents are:

P2: A $\leftrightarrow$ B, B $\leftrightarrow$ C, C $\leftrightarrow$ D, D $\leftrightarrow$ E, E $\leftrightarrow$ F, F $\leftrightarrow$ A

P2: B $\leftrightarrow$ D, D $\leftrightarrow$ C, C $\leftrightarrow$ F, F $\leftrightarrow$ E, E $\leftrightarrow$ A, A $\leftrightarrow$ B

Notice that all edges are bi-directional: the shortest routes between any two cities can be travelled in either direction.

A quick analysis of the edges in C1 and C2 reveals that C1 only inherits 50% of its edges from a parent, while C2 has a mere 33% edge inheritance. Clearly, the key aspects of each parent are poorly transmitted to the offspring, and consequently, the rate of evolutionary progress suffers.

In comparison, the edge inheritances of the two child tours in the PMX crossover example of Figure 6 are 75% and 62.5%. Clearly, heritability can be a significant difference between direct and indirect representations.

It is important to remember these issues of epistasis and heritability when contemplating the use of an indirect representation. As we will see, some problems are simply not amenable to direct representations.

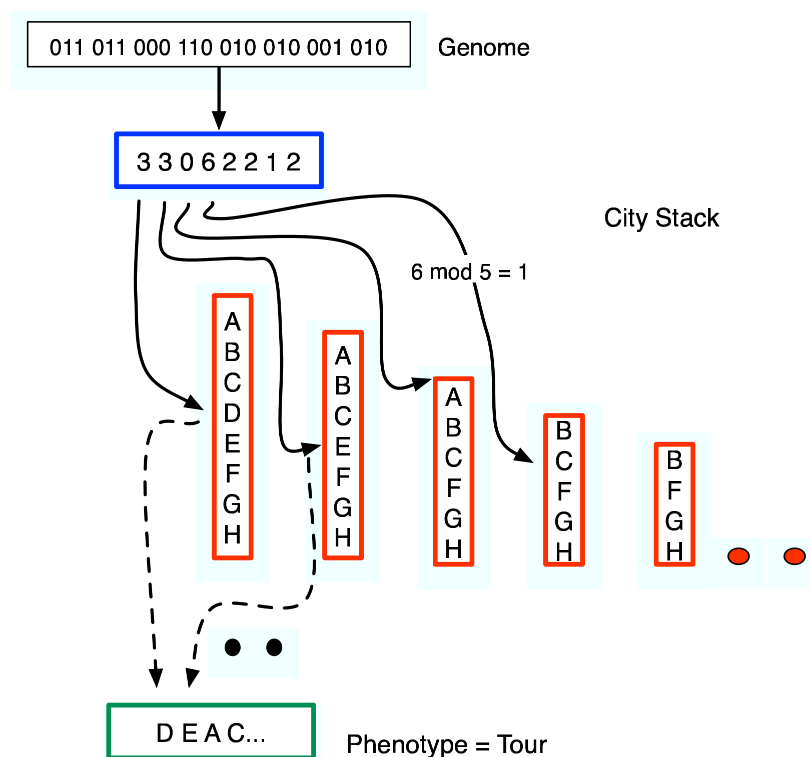

Figure 8: Decoding of an indirect representation for the TSP. Each integer from the bit-vector genome denotes a (zero-based) index into the stack of unused cities. An integer larger than the stack size is reduced via the modulo of that size. The letters A-H denote cities.

3.2 Edge Recombination for the TSP

The insight that edges, not city positions, are critical for solving the TSP with evolutionary algorithms, is due to Whitley et. al. [19]. They developed the Edge Recombination Operator (ERO)

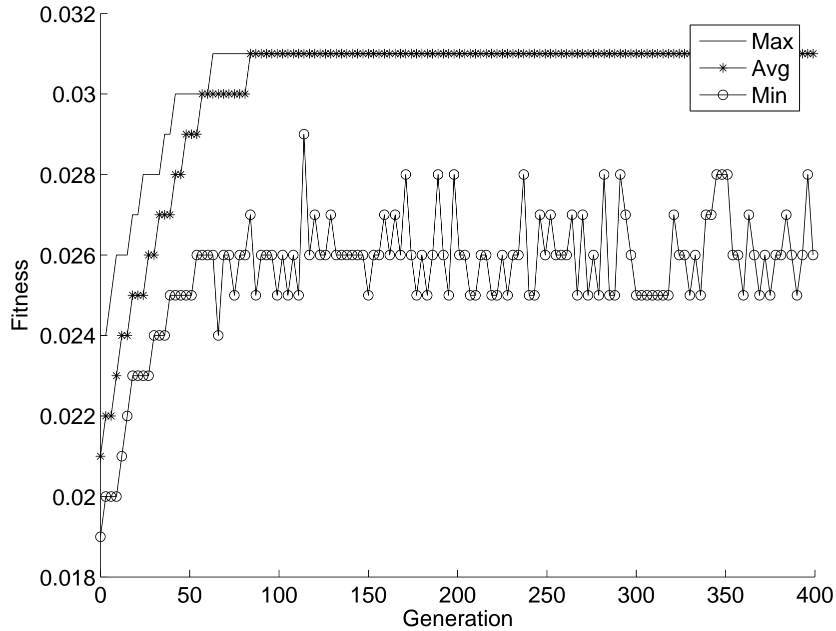

Figure 9: Fitness plot for a 500-individual EA using an indirect binary representation on the 29-city Sahara Desert TSP.

to perform crossover with the explicit emphasis on edge inheritance. The algorithm is quite simple and can easily be added as an option to any EA that handles permutation-list genotypes.

The cornerstone of ERO is an *edge map*, which is a lookup table in which the entry for a node is a list of all other nodes that it is directly connected to in either of the parent permutations. As a simple example, consider a 6-city, direct-encoded TSP in which the following two parents will be recombined:

P1: (A C D E F B)

P2: (E A B C F D)

The edge map for P1 and P2 is then a set of *neighbor lists*, one for each node:

A {C B E }

B {A C F}

C {A B D F }

D {C E F }

E {A D F }

F { B C D E }

Genotype	Crossover	Optimal Runs	Average Tour (km)	St. Dev. Tour
Indirect Binary	Standard Binary	4	28244	642
Direct Permutation	PMX	0	30796	1880
Direct Permutation	ERO	4	**27932**	**314**

Table 1: Summary of 30 different EA runs, 10 for each of the 3 genotype/crossover combinations. Each run uses a population of 500 for 200 generations on the 29-city Sahara TSP, which has an optimal path of 27603 kilometers.

Although not evident in this example, the edge map includes wrap around connections in the permutation from the last to the first element, since the tour is a connected circuit.

Once the edge map is created for a parent pair, the ERO works as follows to produce one child permutation, K:

1. Initialize K = $\emptyset$
2. Let N be the (randomly chosen) first node of either P1 or P2.
3. While length(K) < length(P1) do:
   (a) K $\leftarrow$ K $\cup$ {N}
   (b) Remove N from all neighbor lists in the edge map.
   (c) If N has a non-empty neighbor list,
       then let N* be the neighbor of N with the fewest items in its neighbor list.
       else let N* be a randomly chosen element of the set P1 - K. (e.g. any city not already in K)
   (d) N $\leftarrow$ N*

Below, a simple trace of this algorithm is shown, with parent P1 randomly chosen such that A is the first node. Between each updated version of K, the neighbor list of the newest node, N, is shown (above the arrow), along with the lengths of each neighbor's neighbor list (in parentheses), with N having just been deleted from those lists.

$$A \overset{C(3),B(2),E(2)}{\longrightarrow} AE \overset{D(2),F(3)}{\longrightarrow} AED \overset{C(2),F(2)}{\longrightarrow} AEDC \overset{B(1),F(1)}{\longrightarrow} AEDCB \overset{F(0)}{\longrightarrow} AEDCBF \quad (3)$$

Notice that in the child tour, AEDCBF, only one edge, FA, does not come from either P1 or P2; that is a heritability of 83%, which is actually quite low for ERO!

Table 1 compares the three approaches to the TSP described above. Although ERO does outperform the others, as expected, the binary-coded genome does fare quite well. The position-coded permutation representation with PMX is significantly worse than the other two. However, the ERO does run 6 times slower than PMX!

4 Tantrix

In 1987, Mike Mcmanaway, a New Zealand backgammon champion and puzzle-shop owner, invented *The Mind Game*, a two-person board game involving two-colored hexagonal tiles. The game enjoyed regional popularity and by 1991 had evolved into *Tantrix*, a versatile set of 4-color tiles supporting both a 2-4 player game and host of individual puzzles. A set of 5 *Super* puzzles was added in 1994, the Tantrix internet site (www.tantrix.com) appeared in 1996, and the first world championships were held in 1998. Tantrix now appeals to a broad international audience, with many of the best players coming from Hungary, Israel and New Zealand.

Basic Artificial Intelligence (AI) has been on the scene since 1999, when the first automated players of multi-person Tantrix were released. Today, the Tantrix web site includes a "robot" competition in addition to the highly competitive human tournament.

The wide range of Tantrix puzzles can be co-opted to provide everything from a) simple and nicely graphical examples of indirectly-coded GAs, to b) challenging EC homework and project assignments, to c) extremely difficult search problems is highly-deceptive landscapes, to d) currently unsolved holy-grail puzzles that would probably tax even the largest EC-dedicated Beowulf.

This section introduces Tantrix-GA, a genetic algorithm [9, 5] for solving Tantrix puzzles. We have used it to solve all 5 of the Tantrix *Rainbow* Puzzles, the 3- (trivial) to 30-block (very difficult) *Discovery* Puzzles, and the 5 Super Puzzles. The *Unsolvable* puzzles are currently well-beyond its reach. So, in addition to gaining basic insights into the application of GA to Tantrix, the reader may become inspired to join the hunt for solutions to the most perplexing Tantrix brain-twisters.

4.1 Tantrix Basics

The Tantrix *Game Pack* consists of 56 hexagonal tiles, each containing 3 arcs of different colors. Each of a tile's 6 edges is intersected by one of the 3 arcs. The tiles are numbered from 1 to 56, and the 4 groups of tiles 1-14, 15-28, 29-42, and 43-56 have a special significance with respect to the coloring scheme. All told, there are 4 possible arc colors in the 56-tile set: red, green, blue and yellow. However, each of the 4 groups of 14 employs a different set of 3 colors: 1) red, yellow, blue; 2) red, yellow, green; 3) red, green, blue; and 4) blue, green, yellow, respectively. To view the complete Tantrix tile set, see www.tantrix.com.

Orthogonal to this color scheme is a second 5-color labelling of the tile numbers. Each tile's number is written in one of these 5 colors on the back of the tile. There is no simple explanation for the assignment of these 5 colors, but they partition the 56 tiles into 5 sets: 1) Green - 10 tiles, 2) Yellow - 12 tiles, 3) White - 9 tiles, 4) Blue - 10 tiles, 5) Red - 15 tiles. These groups are the basis for the Rainbow puzzles explained below. In general, any reference to, say, the blue tiles refers to these 10 tiles and not to all tiles containing a blue arc.

In general, the goal of all Tantrix games and puzzles is to form a single two-dimensional cluster of tiles (called the *tantrix*) containing long lines or loops of the arc colors, while abiding by *The Golden Rule*: the shared edge between any two tiles must have the same arc color on both sides. For

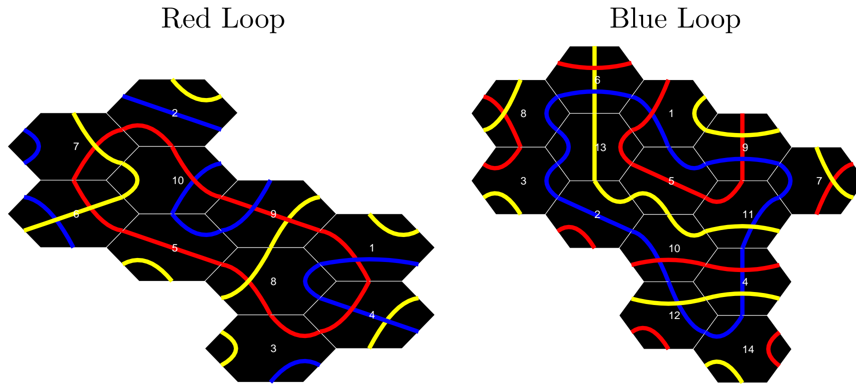

Figure 10: Solutions to the 10- and 14-tile Discovery puzzles.

example, the tantrix on the left of Figure 10 shows a red loop involving tiles 1-10. This constitutes a solution to the 10-puzzle from the Discovery set. Similarly, the rightmost tantrix solves the 14-puzzle, using a different color, blue.

Along with The Golden Rule, a second constraint is important in puzzle solving: the tantrix cannot contain holes, i.e., open cells completely enclosed within the tantrix. Figure 11 shows a 10-tile tantrix that contains a hole and thus is not a valid solution.

The Golden Rule and the hole restriction are the only hard constraints in Tantrix puzzle solving. A puzzle then consists of a given set of tiles, e.g. all 10 green tiles, and a (very general) description of the desired pattern, e.g., *a loop containing all 10 green arcs*. In no case is the exact shape of the complete curve or cycle given in the problem description, but in a few cases, the form of the tantrix is specified as a pyramid. Otherwise, the cluster's shape is also unconstrained, as long as it has no holes.

In the multi-person Tantrix game, each player has a different color and tries to form long curves or loops with it. However, all players share the same tantrix and alternate adding tiles to it. The rules for tile placement are more complicated than for the puzzles, involving a 3-step process in which one player can conceivably add dozens of tiles to the tantrix on a single turn. Further information on the multi-player game is available at the Tantrix web site.

This section focuses on the puzzles, of which there are many. However, most fall into one of the 4 categories below. Tantrix puzzles have been proven NP-complete via a reduction of the circuit-synthesis problem to a Tantrix task [10].

4.1.1 Discovery Puzzles

These puzzles are the simplest to describe but vary in complexity from trivial to extreme. They involve the tiles numbered 1-30. For the k-tile ($3 \leq k \leq 30$) Discovery puzzle, the problem is simply to use the tiles numbered 1-k to build a tantrix with a single k-segment loop. The required color

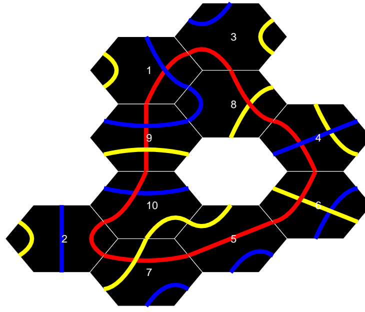

Figure 11: An invalid solution to the 10-puzzle, due to the hole.

Blue Pyramid Red Pyramid

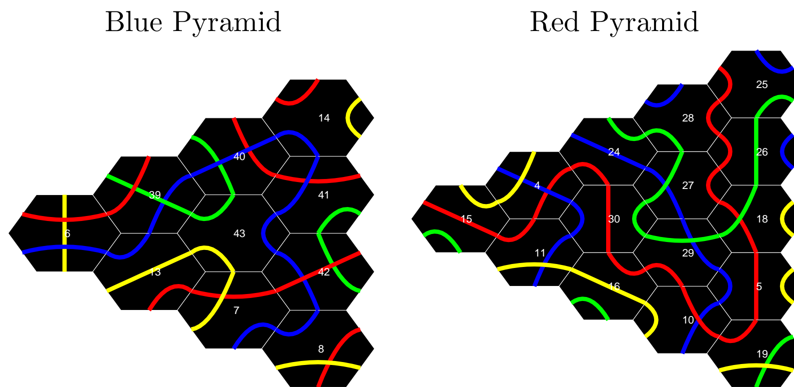

Figure 12: Solutions to the blue and red Rainbow puzzles.

of the loop corresponds to the color code of the kth tile. For example, the 10th tile is in the red group, while the 14th is blue.

4.1.2 Rainbow Puzzles

These involve the 5 color groups and are named after the color of the group. The specifications vary. The green (10-tile) and yellow (12-tile) puzzles require a tantrix containing a loop of 10 (12) green (yellow) arcs. The blue (10-tile) and red (15-tile) puzzles require a pyramid-shaped tantrix containing a non-looping segment of 10 (15) blue (red) arcs. Finally, the white puzzle requires a tantrix containing a loop (in an unspecified color) of the 9 white tiles. For example, Figure 12 shows the solutions to the two pyramid puzzles.

4.1.3 Super Puzzles

For these puzzles, a diverse set of 10 or 12 tiles are given, and the solution involves one **or two** loops or segments using all arcs of the given color(s). The tile sets, with their official names, are

as follows:

1. Junior (10 tiles) - 3,5,8,12,14,43,46,50,52,54
2. Student (10 tiles) - 19, 21, 24, 25, 29, 31, 32, 40, 41, 42
3. Professor (12 tiles) - 2, 11, 15, 17, 20, 30, 38, 39, 44, 45, 51, 56
4. Master (12 tiles) - 18, 22, 23, 26, 27, 33, 34, 35, 36, 47, 53, 55
5. Genius (12 tiles) - 1, 4, 6, 7, 9, 10, 13, 16, 28, 37, 48, 49

The Junior, Student and Master puzzles require the formation of a single 10- (12-) arc loop in an unspecified color. The Professor puzzle requires the formation of two loops of unspecified colors, where each loop involves all the arcs of its color, while the Genius puzzle requires two non-looping segments using all arcs in two unspecified colors. There are 3 known solutions to the Genius puzzle, two involve red and yellow, and one involves blue and red.

For puzzles in which the colors of the target pattern are not specified in the problem statement, some colors can be eliminated from consideration by a simple arc-counting method. For each color, there are only 3 types of arcs: straight line, sharp 120° turn, and gentle 60° turn. If a set of arcs form a loop, then we can begin at any point on the loop and follow the arcs, updating our current orientation on each new tile. At the end of the loop, the orientation must be the same as at the start. For this to occur, there must be an even number of 60° turns. Hence, any color with an odd number of these gentle turns cannot form a loop with all its arcs. This simple test usually filters out a few colors. For example, it helps in determining that the loop color for the Master puzzle is green. Figure 13 shows one solution.

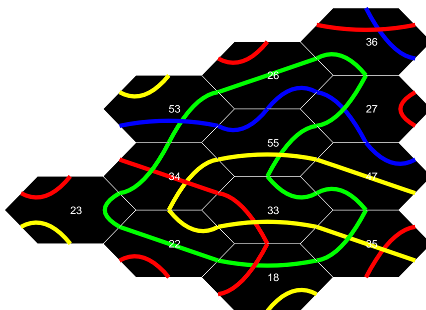

Figure 13: One of a few possible solutions for the Master puzzle.

4.1.4 Unsolved Puzzles

There are two holy-grail, *Unsolved* Tantrix puzzles, one of which remains unsolved as of this writing. Both involve the entire 56-tile set. The *Curve puzzle* requires a tantrix that contains 4 continuous curves, one in each color, while the *Loop puzzle* requires that the 4 curves be closed.

Each color appears on $\frac{3}{4}$ of the 56 tiles, so there are 42 arcs of each color. Thus, in theory, a perfect solution to the Loop or Curve puzzle would use all 4×42 arcs, yielding a score of 168, where the score is simply the sum of the number of arcs used in the longest segment or loop of each color. However, according to the Tantrix manual, computer analyses have shown that the maximum possible scores for the Curve and Loop puzzles are 146 and 136, respectively.

The manual also reveals that Jack Kuiper solved the Loop puzzle in 2003 (without the aid of a computer), while the best recorded score on the Curve puzzle is 140. Somewhat counterintuively, in multi-color puzzles, several loops are easier to discover than several open-ended segments.

And if this is not enough of a challenge, Tantrix products include 10 different, but compatible, 10-tile Discovery packs that enable extensions of the basic Discovery puzzles from 30 to 100 tiles.

4.2 The EA Challenge and Appeal of Tantrix

The combinatorics of Tantrix solutions are quite daunting. First of all, most puzzles involve an unspecified form for the cluster/tantrix; the only constraint is that it is connected and contains no holes. The Golden Rule (i.e., all colors match) greatly restricts the possibilities, but it is difficult to compute its quantitative effect upon this or other contributions to search-space size.

Second, given a pre-defined cluster form, the k tiles must be placed within it and rotated in one of 6 ways. For all except the first tile placed in a cluster, at most 2 of the rotations will match colors with the adjacent tiles in the current tantrix. Hence, given a pre-defined cluster shape, the worst-case size of the search space is:

$$Size_{ss} = 6(k!)2^{k-1} \tag{4}$$

This yields 11147673600 for the 10-tile case and 7653247968377485393920000 for the 20-tile case, and this does not include the combinatorics of the space of legal cluster shapes!

Finally, the fitness landscape (given most straightforward fitness measures) is quite rugged and deceptive. The neighbors formed by swapping any pair of tiles of a k-1 tile loop may all be non-loops or even illegal configurations. Although some k-tile solutions stem from small modifications to k-1 tile solutions, this is hardly a general rule. In fact, in the Discovery puzzles, the solution to the k-1 puzzle may involve a different color entirely from that of the k-tile puzzle.

4.3 Tantrix-GA

Figure 14 illustrates the conversion from genotype to phenotype (i.e., development) in Tantrix-GA. The chromosome consists of two regions, one for determining the growth pattern of the tantrix cluster, and the other for prioritizing the blocks and selecting their orientations (in cases where several rotations are valid).

The shape region simply encodes a sequence of k-1 moves for a k-tile problem, where each move is either South (0), Southeast (1), Northeast (2), North (3), Northwest (4) or Southwest (5). The move dictates the next cell to consider filling; it is always a neighbor of the previously-filled cell.

The second half of the genome consists of k pairs, one for each of the k blocks in the puzzle. The first element of each pair gives a priority to the block - low numbers indicate high priorities - thus determining its placement within the sorted priority list. The second member of the pair is used to choose an orientation for the block in cases where 2 or more orientations would validly match the existing tantrix. For the first block in the list, the tantrix will be empty, so this number will choose one of 6 orientations with which to begin building. All other blocks will have at most 2 feasible orientations. For target patterns with a fixed shape (given by a template to be filled by the tiles), this choice among 6 rotations of the first tile is important. When the shape template is not given, any initial orientation will suffice.

The developmental process is straightforward. The first block is removed from the priority list (which is sorted by ascending priority number) and placed in the middle of the board. The first move is then read from the developmental sequence and the specified neighbor cell becomes the one to fill. The block list is then searched from start to finish until a block that fits into the cell is found. To fit, the block must not only obey The Golden Rule, but it must match up with at least one tantrix edge containing a *focal color*, i.e., a color that is believed to comprise one of the solution loops. For example, in the blue Rainbow puzzle, the focal color is blue, while in the Genius puzzle, there are 2 focal colors. Development continues until either all blocks are used or the next chosen cell cannot be filled by any remaining blocks.

Let us trace through the development of the 5-tile Discovery puzzle in Figure 14, with red as the focal color. The shape region of the chromosome yields a 4-move sequence: North, Southeast, North, and Northwest. The priorities in the second half of the chromosome dictate the following block ordering: 3,5,1,4,2. Hence, the 3rd block is placed on the board and rotated 5 60° units clockwise, since 65 is the value of tile 3's orientation gene, and 65 mod 6 = 5. The neutral (0°) orientations are given in the Tantrix manual and are shown for blocks 1-5 in Figure 14, directly beneath their respective chromosomal regions.

Control then moves to the Northern neighbor of tile 3, and tile 5 (next on the priority list) is given the first chance to fill the spot. At this early stage of the solution, two orientations of tile 5 are valid: 0° and 180° clockwise rotations. Either will match the border color(s) to that cell, and in this case, the only border color is the red arc emanating from tile 3's northern edge. Tile 5's orientation gene, 131 is used to choose among these 2 alternatives: 131 mod 2 = 1, so the latter rotation, 180°, wins.

Control then moves to tile 5's Southeast neighbor, which is also tile 3's Northeast neighbor. Tile 1 gets the first chance to fill this spot and successfully does so, but with only one of its rotations, so the disambiguating orientation gene is not needed. The next move is straight North, and tile 4 satisfies this spot with a unique rotation. Finally, tile 4's Northwest neighbor is filled by tile 2 and the puzzle is successfully completed.

The entire tile-placement algorithm is slightly more complicated than shown in the above example. From newly-placed tile T, the shape portion of the genome actually specifies the *first* of T's neigh-

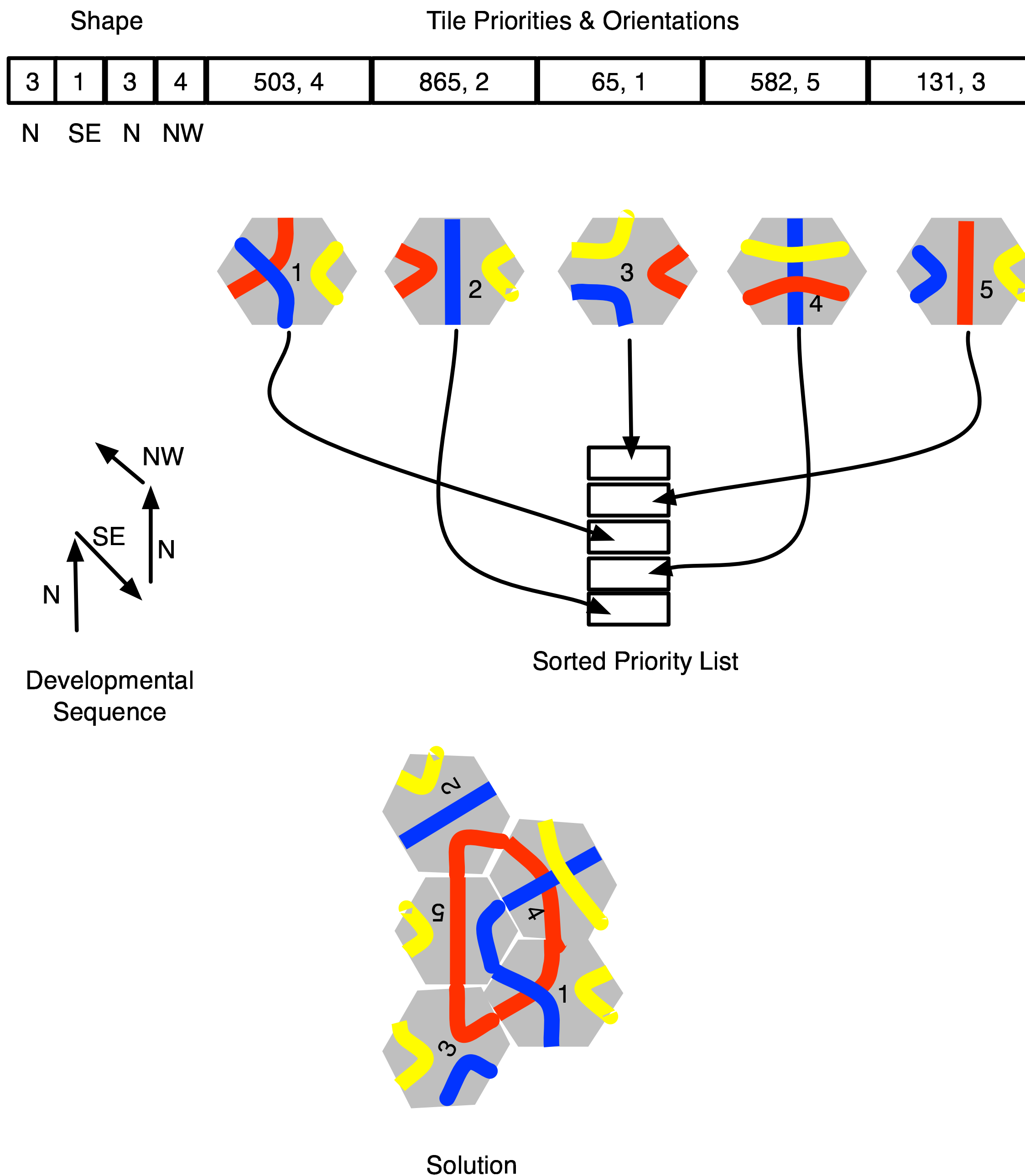

Figure 14: Development of a solution to the 5-tile Discovery puzzle from genotype to phenotype. The tiles labeled 1-5 are the first 5 in the Tantrix Game Pack. The rightmost part of the chromosome determines the priority ordering of these tiles and their preferred orientations, while the leftmost segment defines the overall shape of the tantrix via a growth sequence.

bors, N_1, to investigate. If N_1 is not on a side that matches a focal-color arc (emanating either from T or from one of the other neighbors of N_1), then the algorithm departs from N_1 and moves counter-clockwise around T in search of the first neighbor, N*, that does border on a focal-color arc. The building process halts if N* is not found. Otherwise, N*, and the remaining neighbors moving counterclockwise from N* around T, are returned as a set, C. Each remaining tile is then tested for a color match with N*. If no match is found, the next element in C is tested against all remaining tiles, and so on. This algorithm strongly biases the tantrix growth process toward the formation of long segments and loops in the focal color(s).

Note that nothing in this developmental scheme safeguards against hole creation, nor the formation of long thin segments with no chance of looping back upon themselves. In the multi-person Tantrix game, special move constraints protect against hole formation, while in Tantrix-GA, these restrictions are handled implicitly (and imperfectly) by the fitness function.

4.3.1 The Fitness Function

In Tantrix GA, fitness assessment of a tantrix involves 4 factors:

1. Segments (S)
2. Cycles (C)
3. Compactness (M)
4. Holes (H)

For each of the focal colors, the algorithm counts the longest segment and longest cycle in that color. Then either both, or their maximum, become addends for the fitness function. Compactness is simply the average number of tiles surrounding each tile. It is positively weighted in the fitness function to discourage long thin phenotypes, which, typically, cannot involve long loops unless they take wide turns and leave big holes in the middle of the tantrix. Any open cell in the interior of the tantrix constitutes a hole and is penalized severely.

The complete fitness function appears in equation 5:

$$F_{sum} = k_{co}M - k_{ho}H + \sum_{i=1}^{N_{fc}} k_{seg}S_i + k_{cyc}C_i \tag{5}$$

where $1 \ldots N_{fc}$ are indices of the focal colors, and S_i and C_i are the largest segment and cycle, respectively, in focal-color i. M is the average compactness of all tiles, and H denotes the number of holes in the tantrix. Unless otherwise stated, the evolutionary runs presented in this article used the following parameter values: $k_{co} = k_{seg} = k_{cyc} = 1$, and $k_{ho} = 10$.

F_{sum} denotes the fact that sizes of the largest segment and cycle are summed for each focal color. An alternate fitness measure, F_{max} (equation 6) used for the 12-piece, two-colored Super puzzles, takes the maximum of the segment and cycle contributions for each focal color.

$$F_{max} = k_{co}M - k_{ho}H + \sum_{i=1}^{N_{fc}} max(k_{seg}S_i, k_{cyc}C_i) \tag{6}$$

In this case, k_{cyc} should exceed k_{seg} in order to favor cycles over non-cyclic segments. We use $k_{cyc} = 1.5$ and $k_{seg} = 1$ for the Professor puzzle, and $k_{cyc} = 0$ and $k_{seg} = 1$ for the Genius puzzle, which requires non-looping solutions.

In both fitness functions, cycles should not be favored over simple segments to too large a degree, otherwise small (i.e., sub-optimal) cycles dominate the solutions. Ideally, long, convoluted segments evolve and eventually loop back upon themselves.

4.3.2 Genetic Operators

Tantrix chromosomes are subjected to a variety of genetic operators. First, standard bit-flipping mutation and single-point crossover are employed, with all crossover points restricted to gene boundaries. In addition, Tantrix GA uses *headless chicken crossover* [11, 1], wherein parents are occasionally crossed over with randomly-generated individuals. This often functions as a macro-mutation. In the runs reported below, the (bit-wise) mutation rate is 0.01, and the crossover rate is 0.5, wherein 10% of these are the headless-chicken variety.

Two specialized operators are also included: inversion and priority-swapping. During inversion, a random-length sequence of shape genes (the first part of the chromosome) is sliced out of its current location and spliced into a random new location within the shape portion of the same chromosome. This permits basic spatial patterns to change locations within a tantrix. Inversion is performed on child chromosomes with probability p_{inv}, which has a value of .02 in the runs reported herein.

Priority swapping simply exchanges the alleles of two priority genes. However, only certain types of genes can swap alleles. This process mimics a common strategy used by humans to solve Tantrix puzzles: tiles with the same arc angle for a focal color are swapped. At the beginning of a run, 3 priority-swapping bins are formed, one for each of the 3 possible subtended arc angles: $60°, 120°, 180°$. Tiles are then placed in all bins for which they have a focal-colored arc of the corresponding angle. Then, during priority-swapping mutation, only genes representing tiles that share a bin can swap alleles. The probability of applying a swapping mutation to a child chromosome, p_{swap}, takes a value of .2 in the simulations reported below. Also, after deciding to swap, Tantrix GA may perform a random number of swaps from the uniform distribution $1 \ldots N_{sw}$, where $N_{sw} = 3$ in the reported simulations.

4.3.3 A Slightly Unusual Selection Strategy

Tantrix-GA uses full generational replacement (type A-I) for adult selection. Hence, during each generation, all child genotypes are converted into phenotypes, evaluated for fitness, and moved into the parent group. Mating selection begins with truncation: the worst 50% of the population are removed. Those that remain are subjected to sigma-scale selection, which helps combat premature convergence to suboptimal solutions.

A small fraction of each generation stems from elitism, wherein the top 5% genotypically-distinct individuals are copied, without mutation, to the next generation. 85% of the next generation is formed by crossover and mutation of the top 50% from the previous generation, while the remaining 10% comes from randomly-generated individuals. This continuous re-injection helps to avoid convergence in a search space where a) many randomly-generated individuals have reasonable fitness, and b) small modifications to good solutions are often lethal. Basically, many useful genotypes drop out due to mutation and crossover, so random replenishment helps to maintain a viable gene pool.

4.4 Tantrix-GA Results

Tantrix-GA runs were performed on all Discovery, Rainbow and Super puzzles. Solutions were found for each; every tantrix figure in this document was discovered by Tantrix-GA. Although the algorithm performs well on most of the smaller puzzles and manages to find multiple solutions for the 30-tile Discovery puzzle, it has major problems with the 12-piece *Genius* puzzle, which, to date, it has only solved once (in hundreds of attempts).

For the Discovery puzzles of 20 tiles or less, Tantrix-GA with a population size of 100 and a maximum of 100 generations consistently discovers solutions. Beyond 20 tiles, larger populations and generations are required, as shown in Table 2. Still, it manages to find 4 (completely different) solutions to the 30-tile puzzle in 20 attempts. Unlike the Genius puzzle, which has only 3 known solutions, the many-tiled Discovery puzzles have several solutions. However, they are difficult to find due to many misleading local optima in the search space.

Figure 17 shows solutions to the 26- to 29-tile puzzles. Note that none is a simple modification of the other.

To illustrate the difficulty of the search space, at many times during the simulation, the best-of-generation individual is tested by analyzing all immediate neighbors in genotype space (i.e., those within a 1-bit Hamming distance). In almost all cases, the neighborhood in the fitness landscape resembles a plateau with many small cracks (leading to precipitous drops in fitness), as shown in Figure 15. Most such cracks map to the high-order bits of the priority genes. In the Discovery problems, fewer cracks map to the shape genes, although these become more essential for the multi-segment/loop targets in the Super puzzles. Most importantly, there are no spikes leading upwards. Only when genotypes are tested in very early generations does the occasional upward spike appear. Hence, the fitness landscape appears full of high plateaus that are very hard to hill-climb toward.

As a more standard measure of landscape ruggedness, Figure 16 provides a scatter plot of pairs $(\triangle G, \triangle F)$: differences in genotype (Hamming distance) versus differences in fitness. Here, the best-of-generation individual, B, is compared to 5000 randomly-generated genotypes that are between 1 and 100 bit mutations away from B, with 50 genotypes generated for each mutation/Hamming distance. This example yields a Pearson correlation coefficient [2] just over 0.2, and the plot clearly shows no signs of a correlation. Several similar tests during different Tantrix-GA runs on different puzzles yield the same general result: a correlation coefficient between 0.1 and 0.35 but no visible relationship between $\triangle G$ and $\triangle F$. In short, the fitness landscapes appear quite rugged.

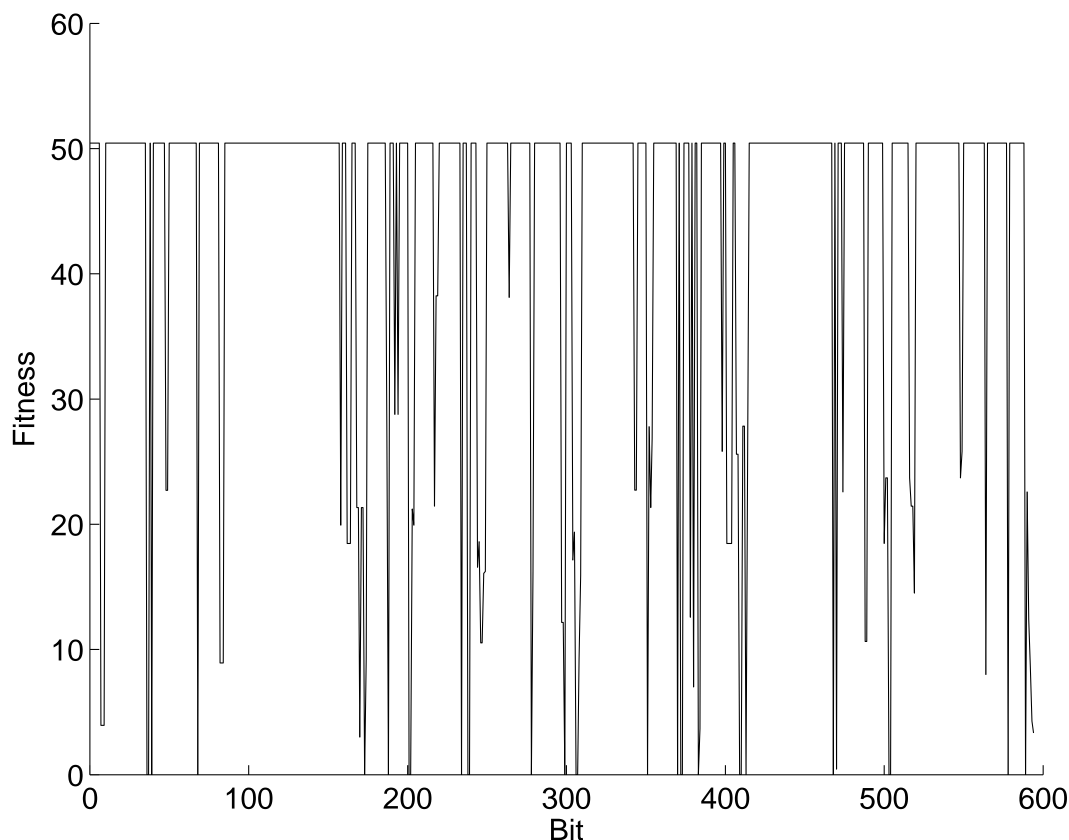

Figure 15: Fitness of all genotypes within a Hamming distance of 1 from a best-of-generation individual during a run of Tantrix-GA on the 30-tile Discovery puzzle. Note the increased sensitivity to mutations of the priority and orientation bits, which begin at location 150, while the shape bits (0-149) are less significant.

4.5 Alternate Representations and Strategies

The quest for improved search efficiency on the Red Rainbow puzzle, the large Discovery puzzles, and the Genius puzzle inspired a variety of alternate genome representations, genotype-phenotype mappings, fitness functions and selection strategies. Unfortunately, none yielded noticeable improvement, and several were largely disastrous.

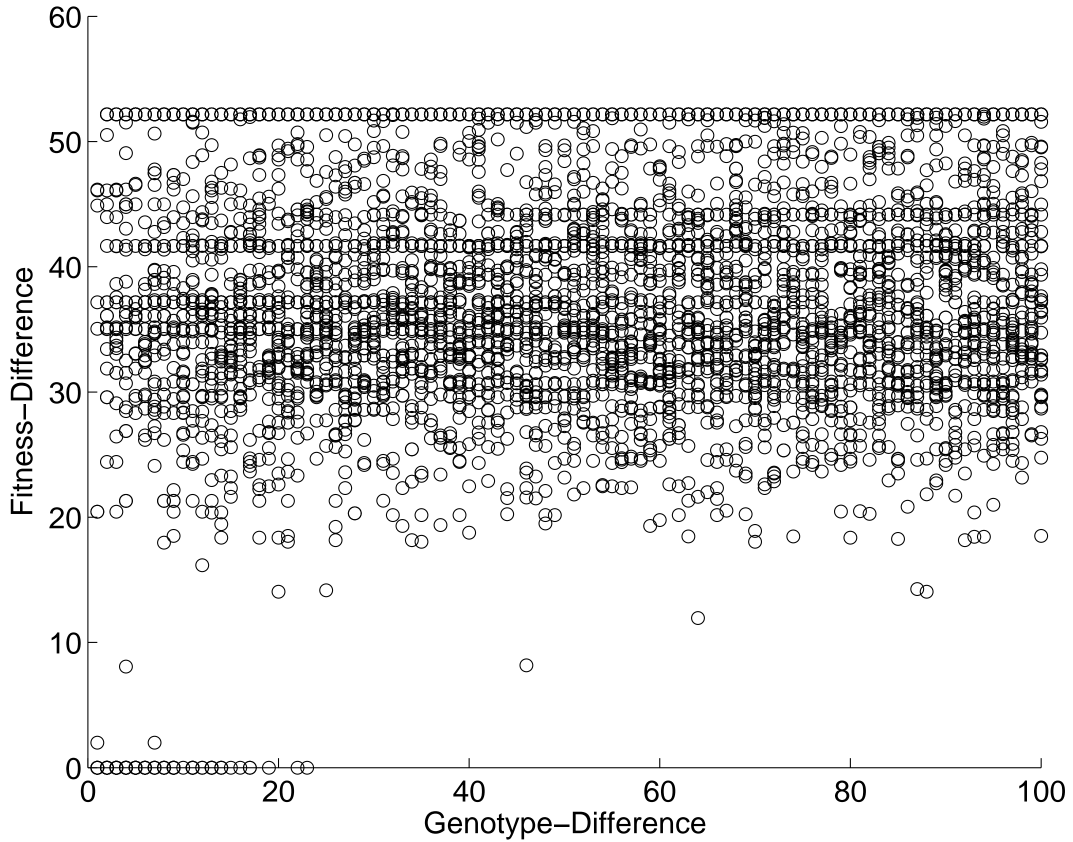

Figure 16: Relationship between genotypic Hamming distance and fitness difference. Given B, the best-of-generation individual (fitness = 52.2) after 100 generations of a run of Tantrix-GA on the 30-tile Discovery puzzle, this looks at mutation classes about B involving 1 to 100 bits, with 50 random samples taken from each class. For each sample, S, the fitness difference between S and B is plotted as a function of the Hamming distance between S and B. All genotypes for this run have a total length of 595 bits. The Pearson correlation coefficient is 0.22

Puzzle	Population Size	Generation Limit	Trials	Number Solved	Average Solution Generation
10-tile	100	100	20	20	5.5
12-tile	100	100	20	20	11.7
15-tile	100	100	20	20	20.8
20-tile	100	100	20	4	51.0
20-tile	200	200	20	15	78.9
25-tile	200	200	20	4	142.2
25-tile	300	300	20	6	148.0
30-tile	500	300	20	4	186.8

Table 2: Summary of multiple Tantrix-GA runs on assorted *Discovery* puzzles.

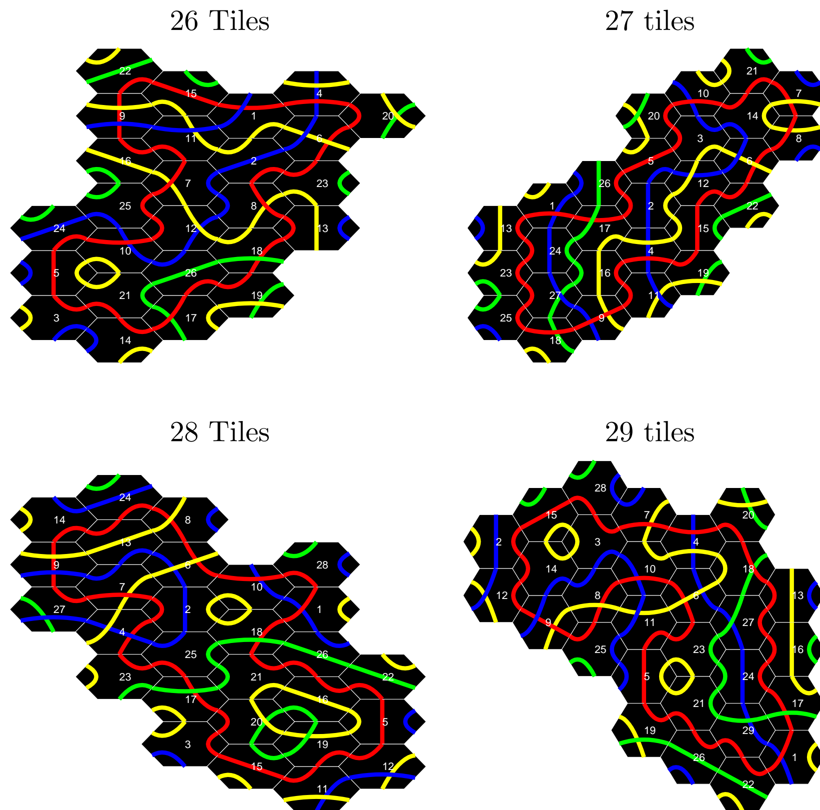

Figure 17: Solutions found by Tantrix GA for the 26- to 29-tile puzzles, all having red as the focal color.

4.5.1 Relaxing The Golden Rule

The Tantrix manual recommends solving puzzles by initially ignoring The Golden Rule and focusing on composing the desired focal-colored loop. Once formed, tiles with equivalent arcs in the focal color can be swapped until The Golden Rule is eventually satisfied.

To incorporate this possibility into Tantrix-GA, the concept of *slack* was defined as the number of mismatches that a tile could have with its neighboring arcs and still be considered legally deployed. A mismatch penalty was added to both fitness functions (F_{sum} and F_{max}) so that tantrices with mismatches scored worse than those without. With slack > 0, Tantrix-GA could easily compose long-looped clusters with many mismatches, but coming up with the proper tile swaps (if they even existed) to remove the mismatches proved nearly impossible. Adding slack seems to have only increased the size of the search space without providing any scaffolding that could be exploited by our genome or genotype-phenotype mapping. Future work could involve changes to these latter elements to better accomodate search in a space dominated by illegal (but potentially helpful) clusters.

4.5.2 A More Direct Representation

Slack also plays a role in a more direct form of genotype-phenotype mapping that, at least theoretically, could be employed to solve fixed-topology problems such as the pyramid puzzles. In these cases, the shape genes are unecessary: a fixed sequence of cells in the pyramid can simply be filled in order by the priority-sorted tile genes. As explained earlier, Tantrix-GA's standard approach simply uses the first tile in the priority list that legally fills a cell (and matches a focal color arc), then it moves on to the next cell in the list. A more direct-encoded solution simply pairs up the cell list and the prioritized tile list such that the kth member of the tile list is always placed in the kth cell, and then oriented according to its rotation gene. Of course, this leads to mismatches, which are penalized by the fitness function. Unfortunately, by expanding the phenotype space to include this multitude of illegal solutions, the direct encoding only seems to make the search problem more difficult.

In effect, the direct encoding forces the use of slack $= 6$, although in practice the effects of all slack values of 3 or more are the same, since the filling algorithm for pyramids never places a new tile in a cell with more than 3 occupied neighbors; hence a maximum of 3 mismatches are possible at tile-placement time.

Figure 18 illustrates the effects of slack on the fitness landscape. Each graph depicts the fitness of all genotypes a Hamming distance of 1 from the best-of-generation individual. Note that with Tantrix-GA's standard (indirect) coding and no slack, there are many flat plateaus, indicating neutral local landscapes. Provided that these plateaus are not too large, they can be advantageous for evolutionary search [12, 20, 14, 15]. With the addition of more slack, the plateaus shrink. Since direct coding necessitates high slack, it produces difficult, low-neutrality landscapes.

In general, runs using slack > 0 rarely find solutions to even the simple puzzles. Slack appears to increase ruggedness and decrease neutrality. It may also increase the deceptiveness of landscapes,

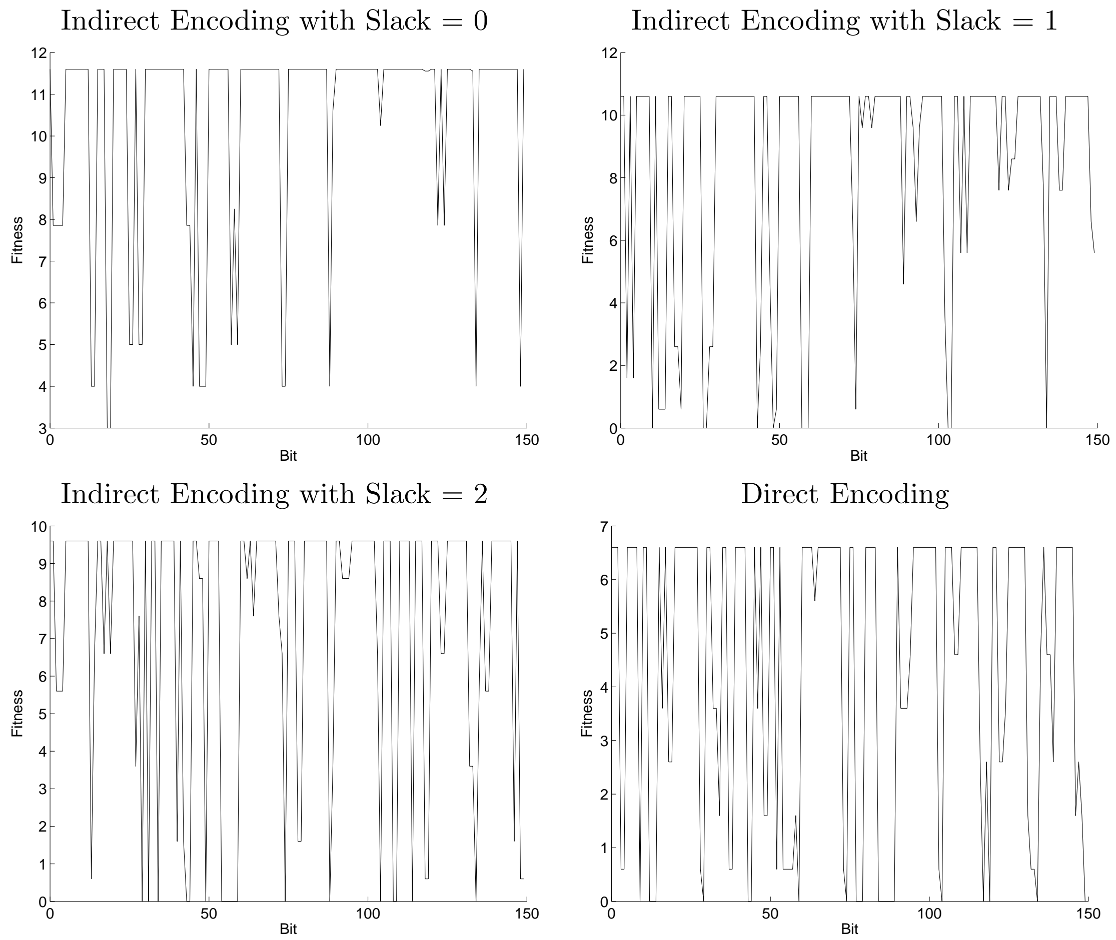

Figure 18: Comparison of fitness landscapes around the best-of-generation individual after 100 generations with a population size of 100 on the 10-tile Blue pyramid puzzle.

since it adds more legal partial solutions that may *appear* to be just a few tile swaps away from perfection, but are not. Although this may not rule out the use of all direct representations for Tantrix puzzles, the developmental approaches seem to show greater promise, particularly when cluster shapes are not pre-determined.

For more information on Tantrix-GA, see [3].

5 Developmental Genetic Programs

Traditionally, genetic programs (GPs) use genotypes that very closely resemble phenotypes. In many cases, the genotype merely needs to be *packaged* so that it actually compiles and runs. However, with a developmental approach to GP, the genotypes maintain the same basic syntax, that is, they are still nearly-executable computer programs, but they now encode a *recipe* for growing the phenotype through a multi-step process. This is analogous to the growth of a human body from DNA code, although, of course, the process is much less complicated in GP.

5.1 Cellular Encoding

Frederic Gruau [8, 7] made a key contribution to this area in the mid 1990's with his *cellular encoding* (CE) approach to evolving artificial neural networks (ANNs). In this model, each genetic programming tree is interpreted as a set of growth instructions, with individual GP functions performing simple operations on the growing ANN such as splitting a node into two, assigning a weight to an arc, etc.

Figure 19 shows the basic components of the CE developmental process. The ANN begins as an *embryo* consisting of the known inputs and outputs plus a single internal node. Pointers from the ANN to the GP tree (drawn as dotted arrows) denote *read heads*, which indicate the GP command that will be used to modify that ANN node. After a command executes, it sprouts new read heads for each of its children in the GP tree while relinquishing its own pointer.

As shown in Figure 20, the execution of the top S command implies the creation of a new node to be attached *serially* with the original node. The original node and the new node then each receive a read head for the next round. These heads point to the two children of the root S command.

The P command calls for a parallel node split, wherein a new node is placed parallel to the original node; it receives copies of all inputs and outputs.

Figures 21, 22 and 23 work through the entire developmental procedure for this simple example. Not shown are commands such as those for modifying ANN arc weights or for making recursive calls to the GP tree (i.e., moving a read head back to the root), which CE also includes.

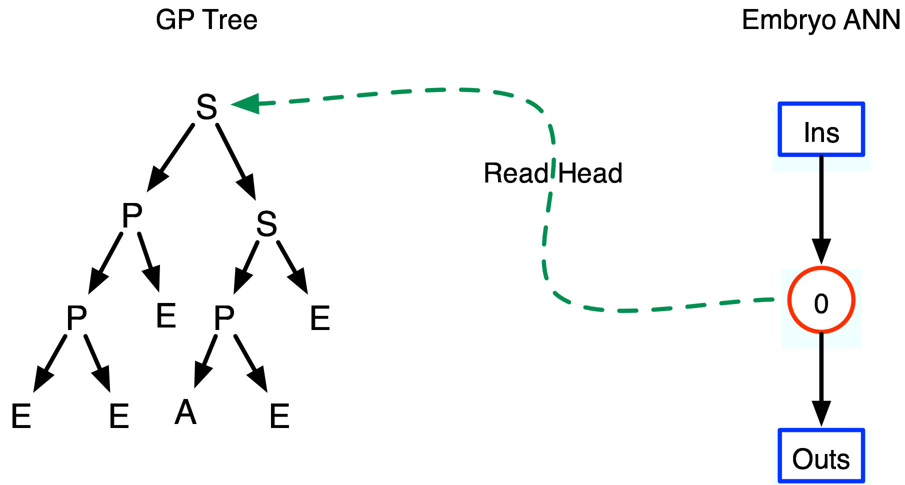

Figure 19: Initial configuration for the Cellular Encoding development process. The neural network may have many inputs and outputs, which are summarized as a single connection to and from the internal node, drawn as a circle. The read head points to the start of the GP Tree.

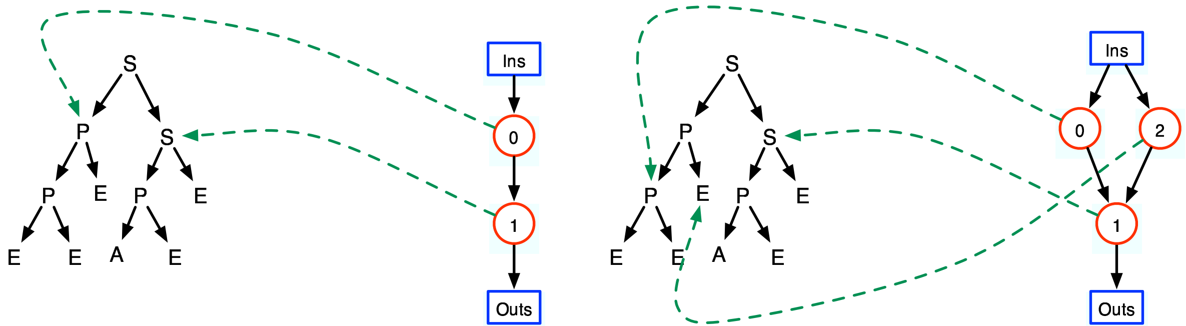

Figure 20: First two steps in the development of a neural network via cellular encoding. (Left) The topmost S command produces a serial expansion of the original node (N0), making the new node, N1, a child of N0.(Right) The level-2 P node then dictates a parallel expansion of NO, with copies of NO's input and output arcs attached to the new node, N2.

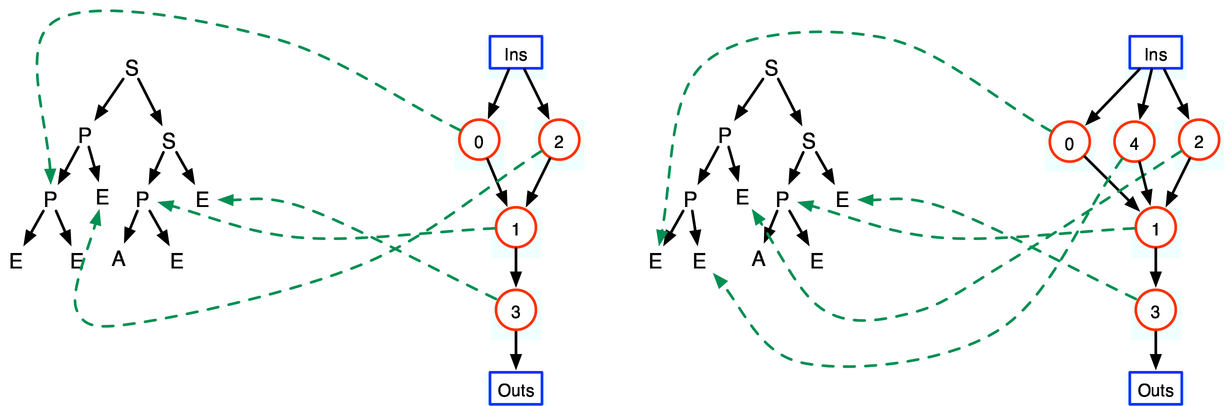

Figure 21: Steps 3-4 in CE development (Left) Node N1 performs the S command, creating node N4 in series with N1. (Right) Node N0 performs a second parallel split by executing the P command, thus producing a 3-node top layer.

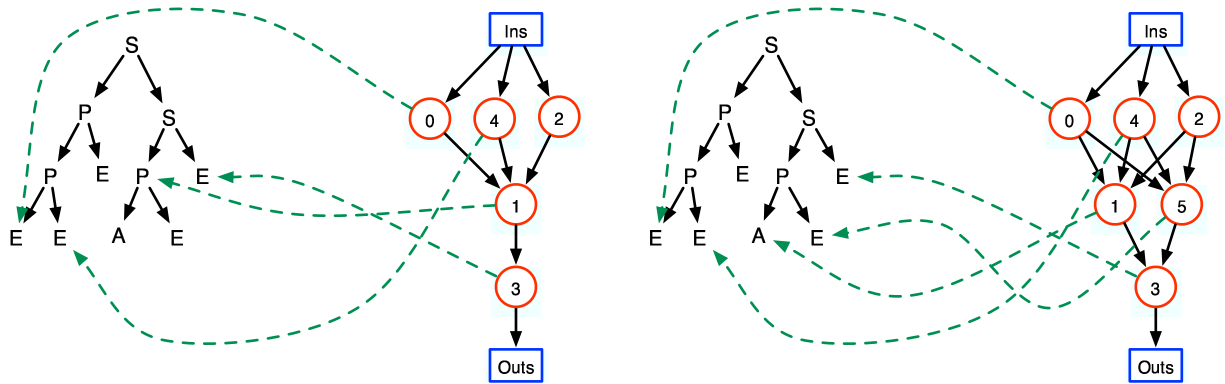

Figure 22: Steps 5-6 in CE development (Left) Node N2 executes an E (End) statement, terminating activity for that node and removing its read head (Right) Node N1 executes a P statement, creating a new parallel node, N5, which receives copies of all of N1's input and output arcs.

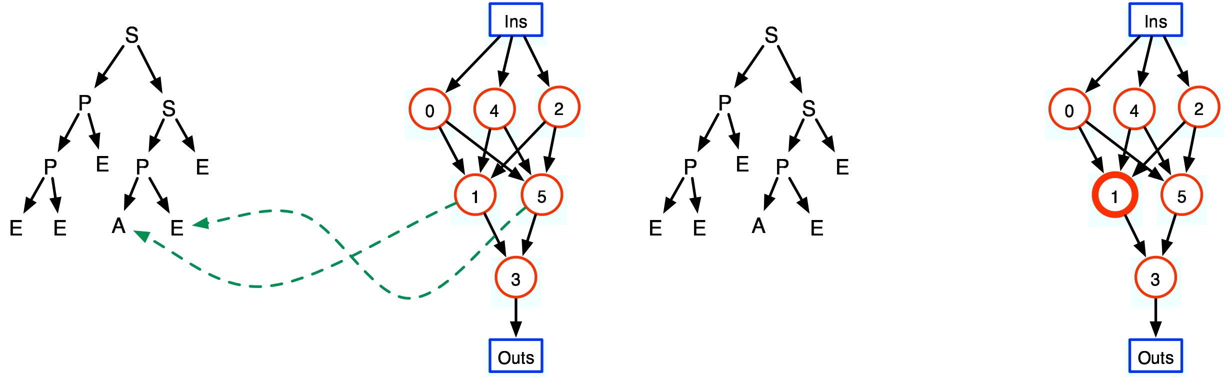

Figure 23: Steps 7-11 in CE development (Left) Nodes N3, N0 and N4 (in that order) execute E commands, terminating their activity and deleting their read heads. (Right) Node N1 executes the A command, which increments the threshold of its transfer function, denoted by a thicker circle, and then terminates activity. N5 executes an E command and also terminates to complete development.

5.2 Developmental Genetic Programming for Circuit Design

In [13], Koza et. al. describe developmental GPs for designing a wide range of electrical circuits. They elaborate upon the basic cellular-encoding procedure to include 5 categories of primitive functions:

1. Component-creating functions - for inserting a component into the topology.
2. Arithmetic functions - These are in the value-setting portions of the component-creating functions. Each component has a few key parameters values, such as resistance or inductance, that must be set.
3. Automatically-defined functions (ADFs) - subroutines that are defined in one part of the GP tree and called from other subtrees.
4. Topology-modifying functions - these form parallel and serial splits of the growing network.
5. Development-controlling functions - basic commands for terminating activity, for example.

These and most other advanced GP applications use *strongly-typed GPs* [17]. These replace the closure constraint with a more detailed set of restrictions on the inputs and outputs of each primitive function. Hence, particular arguments to certain functions may be restricted to certain value types that only some of the other functions can produce. For example, in Koza et. al.'s developing circuits, the component-creating commands typically have two subtrees/arguments, one of which evaluates to a number (representing a key parameter such as resistance), while the other embodies a textitcontinuation indicating the next operation to perform at the same point in the circuit (e.g., put in another resistor, or split the wire into two parallel tracts, etc.).

Another key issue with Developmental GP trees is execution order. In a normal LISP-like program tree, all arguments are evaluated **before** the function itself. This essentially leads to a depth-first traversal of the program tree. However, in typical developmental GP's, only some of a functions arguments are evaluated early, while others (the continuations) are executed afterwards, thus giving a more breadth-first look to tree processing.

As a simple example, Figure 24 shows a typical GP program tree for circuit building along with the default embryo circuit. Wires on the circuit have read heads into the GP tree, with the corresponding primitive function then prescribing a modification for that wire.

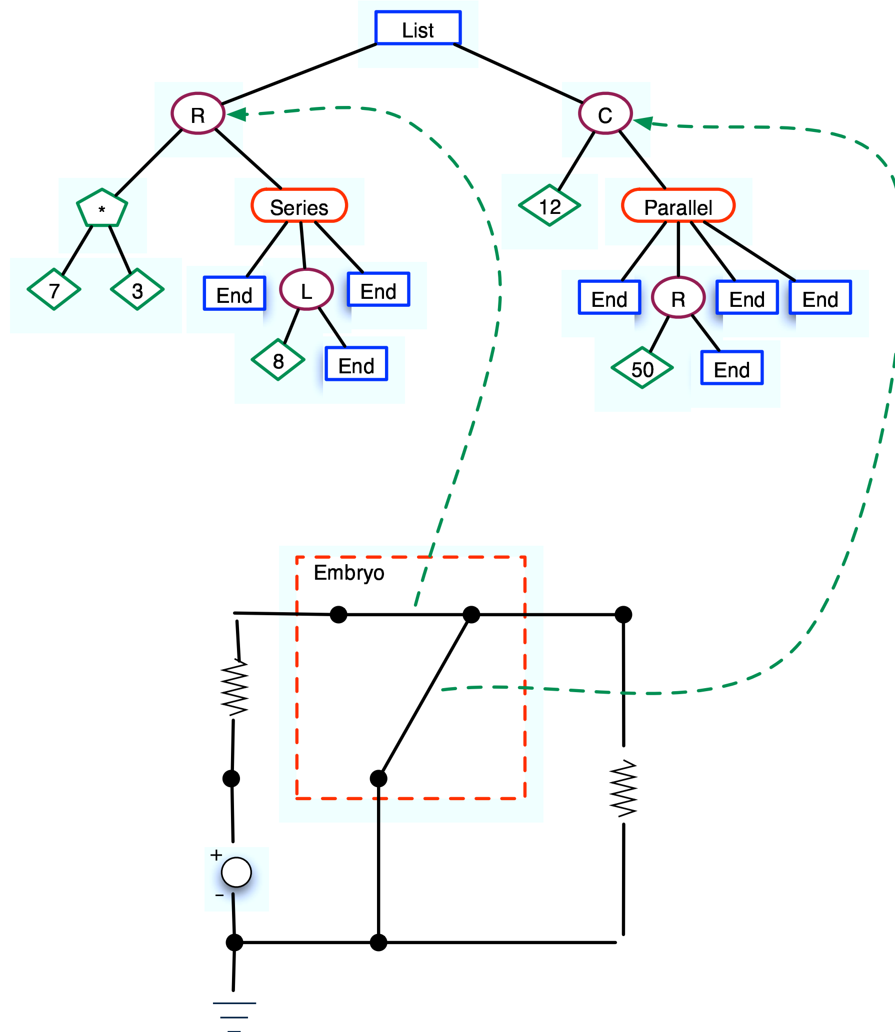

Figure 24: Initial situation for the Circuit GP developmental process. The root command, list, merely indicates that a series of commands will follow, while R and C are component-creation commands for resistors and capacitors, respectively.

Once the circuit is fully formed, Koza et. al.'s GP performs a fitness test using the SPICE circuit simulator. Fitness is typically error-based: inversely related to the deviation of the simulated circuit's output to the desired output, where the latter reflect the intended functionality of the

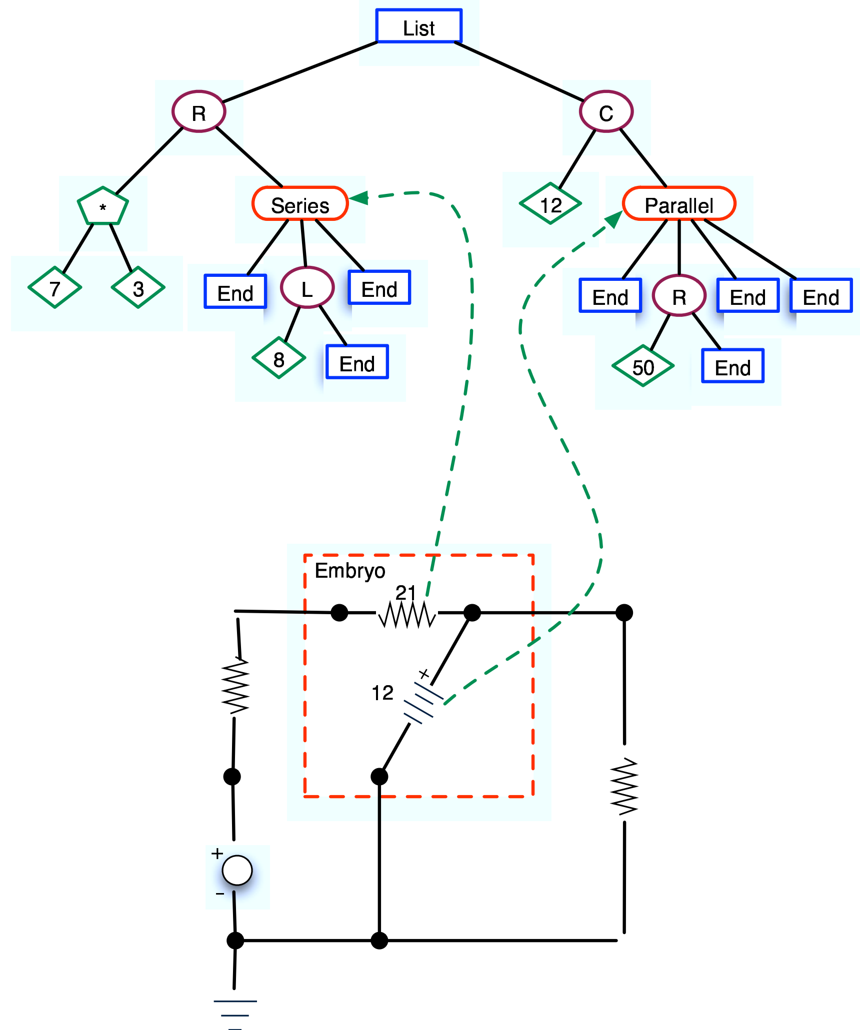

Figure 25: Circuit GP development after the addition of the resistor and capacitor. Note that the arithmetic values (12) and functions (7*3) were performed immediately to give parameter values for the new components - and without creating new read heads, while the Series and Parallel commands are continuations and thus require read heads.

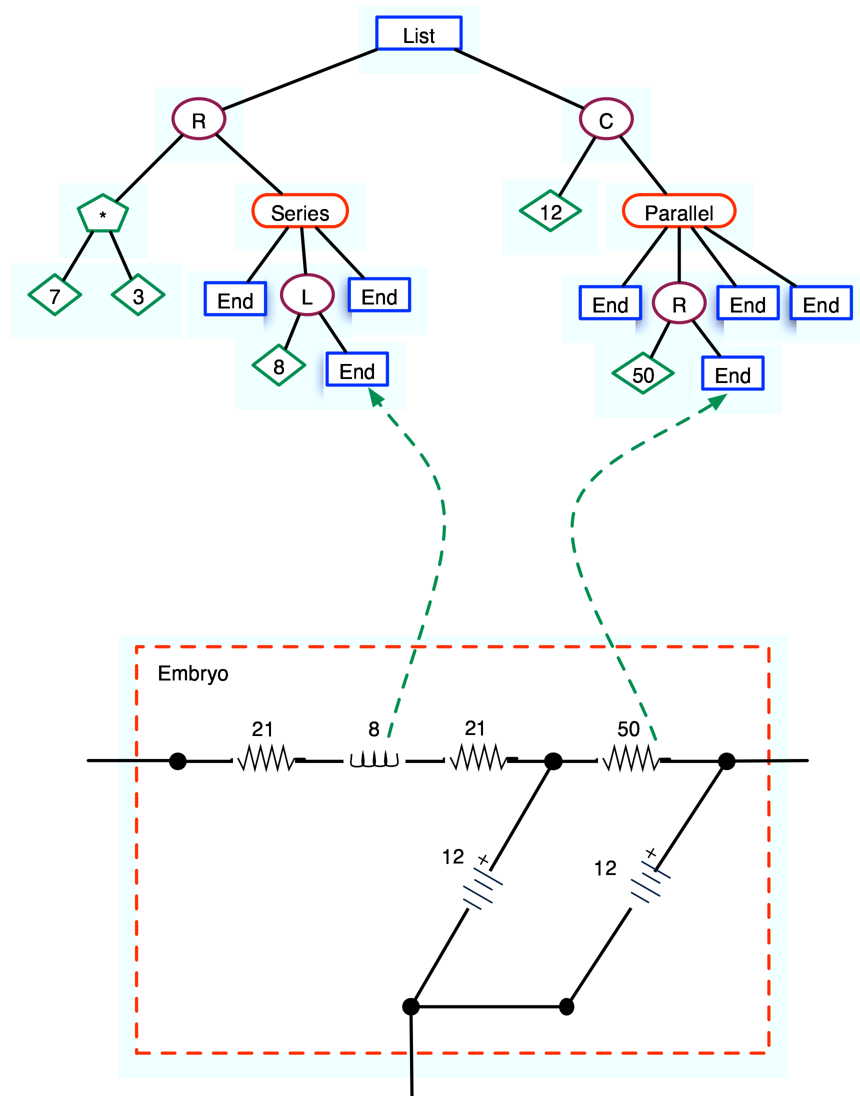

Figure 26: Circuit GP development nearing completion. Note that the series and parallel commands created a copy of the 21-ohm resistor and 12-nanofarad capacitor, respectively. The continuations then created the 8-microhenry inductor and the 50-ohm resistor. The series and parallel functions produce, respectively, 3 and 4 new segments (each with a read head). For the parallel command, the segments are the 4 sides of the parallelogram, and for the series command, the segments are the original component and its copy along with a segment between the two.

design. For example, if the target function is a low-pass filter, then high-frequency signals that pass through (or low-frequency signals that do not) will result in a fitness penalty.

Koza's group has used this methodology to evolve double bandpass filters, Butterworth filters, 3-way source identification circuits, temperature-sensing circuits, amplifiers, and much much more. They have also used the technique to generate PID controllers, field-programmable gate arrays, and biochemical signalling pathways. Gruau's basic idea, in the hands of the tireless John Koza and his 1000-node Beowulf, has been a virtual gold-mine of interesting (and sometimes even patentable) GP results.

5.3 Edge Encoding for Graphs

In Travelling Salesman Problems, the EA designs tours through a fixed graph, while in the neural network and circuit design problems, actual graph topologies are generated and tested. However, in both cases, the issue of node- versus edge- encoding comes up.

In 1996, Luke and Spector [16] devised an edge-based version of cellular-encoding known as *edge encoding* (EE). One fundamental weaknesses of CE that they sought to remedy was the fact that whenever CE creates a new node, it creates arcs, often very many of them, with weights identical to those of their ancestor arcs. To traverse the network and fine-tune all these new arcs is very cumbersome in CE. Hence, the odds of evolving large ANNs with arcs properly tuned for complex problems are poor.

By focusing on arcs and allowing the GP operators to work on them (instead of the nodes), Luke and Spector hoped to generate more complex graph structures. In the end, what they produced was a technique for generating graphs that have a much lower edge/node ratio and important parameters on each arc. This is desirable for evolving Finite State Automata (FSAs) but not necessarily for neural networks, which normally demand reasonably dense connectivity.

The mode of operation in EE is similar to that of CE: Read heads connect the GP tree to the growing graph such that graph components (in this case edges) are modified according to their corresponding GP command. The GP function primitives are also similar to those shown above, with two serial operators, *split* and *bud*, and a parallel operator, *double*. These are briefly explained as they appear in the example below.

Figures 27, 28 and 29 illustrate the development of a non-deterministic finite state automata (NFA) for recognizing regular expressions of the form 10*01(00)*: a 1 followed by zero or more 0's, followed by 01, followed by zero or more 00 segments. The non-determinism stems from the two movement choices when reading a 0 after reading the initial 1; see node W in the figures.

To date, no detailed comparison of CE and EE has been performed. EE does not appear to evolve better neural networks, but it can more naturally label the arcs of networks that it does generate. Hence, FSAs and NFAs are probably easier designed with EE than CE. Since CE can produce densely connected graphs using only a few commands, it seems to be the wiser choice for evolving neural networks, which often require many inter-node links. For an excellent, and much broader,

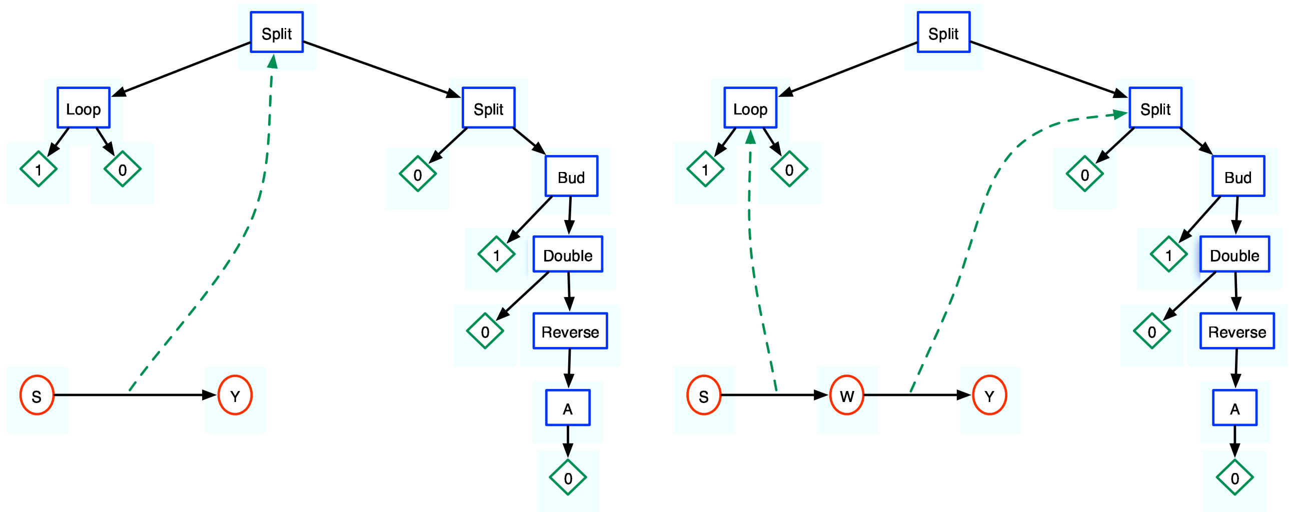

Figure 27: Initial stages of EE development, with the GP tree and embryo NFA. (Left) The NFA begins with a start state S and one other state, Y. The edge $S \rightarrow Y$ has a read head at the GP tree's root. (Right) The split operator creates a new node, W, to *split* the edge $S \rightarrow Y$. Each of the two new edges, $S \rightarrow W$ and $W \rightarrow Y$ gets a read head pointing to the children of the split command.

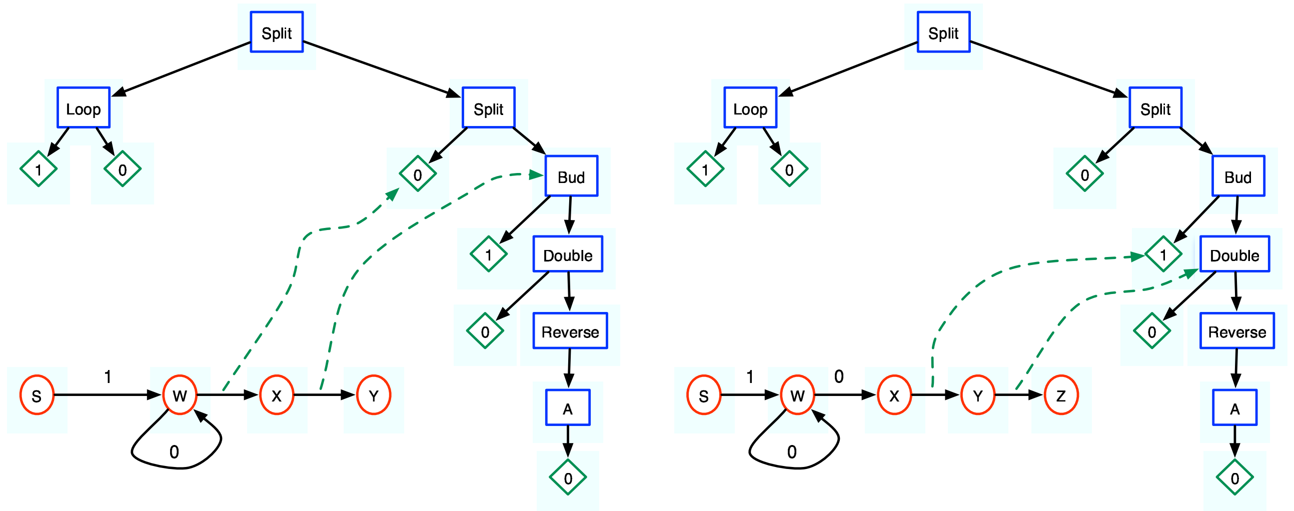

Figure 28: Intermediate stages of EE development (Left) The loop command creates a self-arc on the head (right side) of its edge. The original edge, $S \rightarrow W$ and new edge $W \rightarrow W$ each get a read head to a child of the loop command. Those children, 1 and 0, are terminal operators that place a label on their edge. The second split command converts $W \rightarrow Y$ into $W \rightarrow X$ and $X \rightarrow Y$, with each new edge getting a read head. (Right) After the 2nd split, $W \rightarrow X$ receives a 0 label, while $X \rightarrow Y$ performs a *bud* operation, which adds the edge $Y \rightarrow Z$.

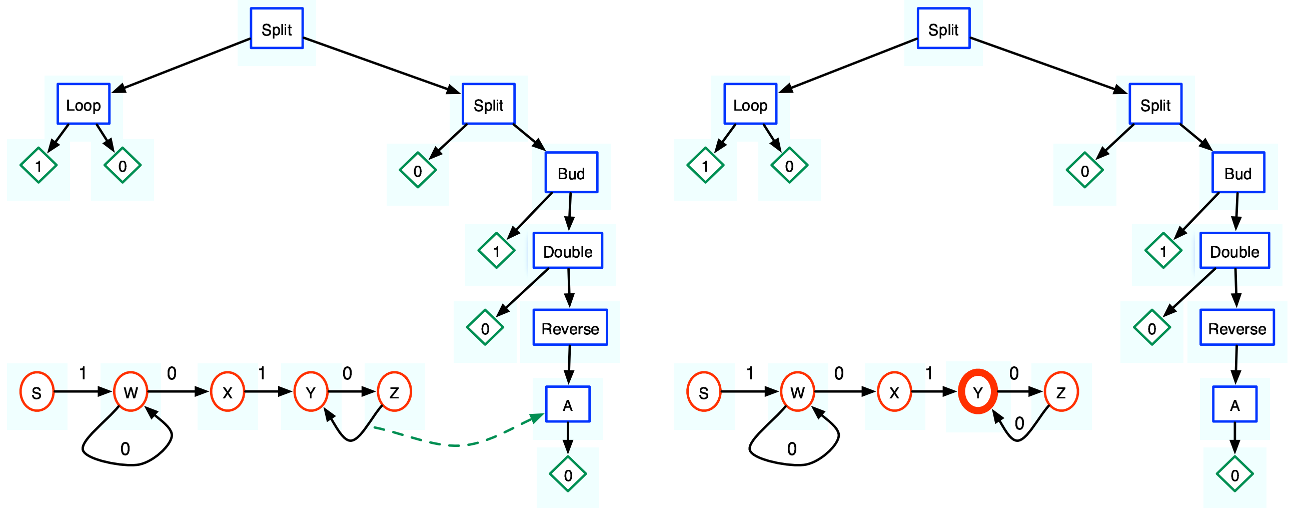

Figure 29: Final stages of EE development (Left) The $X \rightarrow Y$ edge gets a 1 label, and *double* creates a duplicate edge $Y \rightarrow Z$. The children of *double* then label the original $Y \rightarrow Z$ with a 0 and reverse the direction of the duplicate edge to become $Z \rightarrow Y$. (Right) The A command receives edge $Z \rightarrow Y$ and converts its head, Y, into an accepting state (drawn as a thick circle). Finally, A's child, 0, is the label for $Z \rightarrow Y$.

discussion of developmental EAs, see [18].

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] P. Cohen, *Empirical Methods for Artificial Intelligence*, The MIT Press, Cambridge, MA, 1995.

[3] K. Downing, *Tantrix: A minute to learn, 100 (genetic algorithm) generations to master*, Genetic Programming and Evolvable Machines, 6 (2005), pp. 381–406.

[4] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, 1979.

[5] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley Longman, Reading, Massachusetts, 1989.

[6] D. E. Goldberg and R. Lingle, *Alleles, loci, and the traveling salesman problem*, in Proc. of the International Conference on Genetic Algorithms and Their Applications, Pittsburgh, PA, 1985, pp. 154–159.

[7] 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.

[8] F. Gruau and D. Whitley, *Adding learning to the cellular development of neural networks*, Evolutionary Computation, 1 (1993), pp. 213–233.

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

[10] M. Holzer and W. Holzer, *Tantrix rotation puzzles are intractable*, Discrete Applied Mathematics, 144 (2004), pp. 345–358.

[11] T. Jones, *Crossover, macromutation and population-based search*, in Proceedings of the Sixth International Conference on Genetic Algorithms, L. Eshelman, ed., Pittsburgh, PA, Morgan Kaufmann, 1995, pp. 73–80.

[12] M. Kimura, *The Neutral Theory of Molecular Evolution*, Cambridge University Press, Boston, MA, 1983.

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

[14] M. Lones, *Enzyme Genetic Programming*, PhD thesis, University of York, York, England, 2003.

[15] M. A. Lones and A. M. Tyrrell, *Biomimetic representation in enzyme genetic programing*, Genetic Programming and Evolvable Machines, 3 (2002), pp. 193–217.

[16] S. Luke and L. Spector, *Evolving graphs and networks with edge encoding: Preliminary report*, in Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996, J. R. Koza, ed., Stanford University, CA, USA, 28–31 1996, Stanford Bookstore, pp. 117–124.

[17] D. J. Montana, *Strongly typed genetic programming*, Evolutionary Computation, 3 (1995), pp. 199–230.

[18] K. Stanley and R. Miikkulainen, *A taxonomy for artificial embryogeny*, Artificial Life, 9 (2003), pp. 93–130.

[19] D. Whitley, T. Starkweather, and D. Fuquay, *Scheduling problems traveling salesman: The genetic edge recombination operator*, in Proceedings of the Third International Conference on Genetic Algorithms, J. D. Schaffer, ed., San Mateo, CA, 1989, Morgan Kaufman.

[20] A. S. Wu and R. Lindsay, *Empirical studies of the genetic algorithm with non-coding segments*, Evolutionary Computation, 3 (1995), pp. 121–148.