POOL-WISE CROSSOVER IN GENETIC ALGORITHMS: AN INFORMATION-THEORETIC VIEW

Shumeet Baluja[1,2] & Scott Davies[2]

[1] Justsystem Pittsburgh Research Center
4616 Henry St.
Pittsburgh, PA. 15213

[2] School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA. 15213

email:
baluja@cs.cmu.edu, scottd@cs.cmu.edu

Pool-Wise Crossover in Genetic Algorithms: An Information-Theoretic View

Abstract

Several recent studies have examined the effects of replacing the pair-wise recombination operators often used by genetic algorithms with pool-wise recombination operators. In their simplest form, pool-wise operators create probabilistic models that represent each parameter independently. Instead of using traditional crossover operators, these models are sampled to generate a new population. Although these simple probabilistic models are competitive with pair-wise recombination operators on many benchmark problems, pool-wise operators may not perform as well on problems which have a high degree of interdependence between the parameters. We wish to discover whether this is a fundamental limitation of pool-wise crossover or whether the poor performance is due to the simple probabilistic models which are often used. This paper presents a crossover operator, based on information theory, which explicitly captures pair-wise interdependence between parameters. By sampling this probabilistic model to generate the next population, promising results are obtained in comparison to both simple pool-wise and pair-wise operators.

1 Introduction

By maintaining a population of points, genetic algorithms (GAs) can be viewed as creating *implicit* probabilistic models of the solutions seen in the search. In many standard GAs, new sampling points are generated by applying randomized pair-wise recombination operators to the high-performance members of a population [Goldberg, 1989][Holland, 1975][De Jong, 1975]. Pair-wise recombination operators, such as one-point, two-point or uniform crossover, randomly select non-overlapping subsets of two "parent" solutions to place into a "child" solution. By using a crossover which preserves groups of parameters from parent to children strings, GAs attempt to implicitly capture dependencies between the parameters. Crossover's randomization is necessary because no information about which parameter interdependencies are important is explicitly maintained. Therefore, when combining two very different solutions, numerous crossover operations may be required before a useful child solution is produced.

One of the first steps towards making the GA's probabilistic model more explicit was the "Bit-Based Simulated Crossover (BSC)" operator [Syswerda, 1993]. Instead of combining pairs of solutions, population-level statistics were used to generate new solutions. The BSC operator works as follows. For each bit position[1], the number of members that contain a one in that bit position is counted. Each member's contribution is weighted by its fitness with respect to the target optimization function. The same process is used to count the number of zeros. Instead of using pair-wise crossover operators to generate new solutions, BSC generates new query points by stochastically assigning each bit's value with the probability of having seen that value in the previous population (the value specified by the weighted count) [Syswerda, 1993].

The ideas in Population-based incremental learning (PBIL) [Baluja, 1995] were similar to those used in BSC. While BSC used a population of solutions from which the sampling statistics were entirely rederived after each generation, PBIL incrementally adjusts its sampling statistics after each generation. PBIL is similar to a cooperative system of discrete learning automata in which the automata choose their actions independently, but all automata receive a common reinforcement dependent upon all their actions [Thathachar & Sastry, 1987]. Unlike most previous studies of learning automata, which have commonly addressed optimization in noisy but very small environments, PBIL was used to explore large deterministic spaces. The algorithm maintains a real-valued probability vector from which solutions are generated. As search progresses, the values in the probability vector are gradually shifted to represent high-evaluation solution vectors.

1. Note that in this paper, we will discuss combinatorial optimization with the solutions represented as binary vectors. However, all of the results can be trivially extended to higher cardinality alphabets.

Note that the probabilistic model created in PBIL and BSC is extremely simple. *There are no inter-parameter dependencies modeled; each bit is handled independently.* Although this simple probabilistic model was used, PBIL was successful when compared to a variety of standard genetic algorithm and hillclimbing algorithms on numerous benchmark and real-world problems [Baluja, 1997][Greene, 1996]. A more theoretical analysis of PBIL can be found in [Juels, 1997][Kvasnicka *et al*., 1995][Hohfeld & Rudolph, 1997]. However, as described in [Eshelman et. al, 1996], PBIL and BSC may not perform as well as pair-wise operators when tested on problems created to have a high degree of interdependence between parameters.

To facilitate the understanding of how interdependencies between parameters can be explicitly modeled, in the next section we describe PBIL, which uses a simple model that assumes each parameter is independent. Section 3 extends PBIL's probabilistic models to capture pair-wise dependencies. Section 4 provides empirical results with this new probabilistic model. Finally, Section 5 closes the paper with conclusions and suggestions for future work. Extending the models beyond pair-wise dependencies is discussed.

2 Unconditional Probabilities in PBIL

As described earlier, PBIL only maintains unconditional probabilities. The algorithm works as follows: instead of using recombination to create a new population, a real-valued vector, **P**, is sampled. **P** specifies the probability of generating a 1 in each bit position. Initially, all values in **P** are set to 0.5, so that random, uniformly distributed, solutions are generated. A number of solution vectors are generated by stochastically sampling **P**; each bit is sampled independently. The probability vector is then moved towards the solution vectors for which the evaluation function returns the best values, according to Equation 1. The update rule is similar to the updates used in unsupervised competitive learning [Hertz, *et al.* 1991].

$$P_{t+1,i} = (1-\alpha) \cdot P_{t,i} + \alpha \cdot BestSolutionVector_i \quad (1)$$

$P_{t,i}$ is the value of the probability vector at time t, for parameter i. $BestSolutionVector_i$ is the value of parameter i in the vector being used to update the probability vector. α is a learning rate parameter that determines how much each new datapoint changes the value of the probability vector. The basic version of the PBIL algorithm and its parameters are shown in Figure 1. The final result of the PBIL algorithm is the best solution generated throughout the search. Numerous extensions to this basic algorithm are possible - many are similar to those commonly used with genetic algorithms, such as variable or constant mutation rates, elitist selection, or local optimization heuristics.

Unlike BSC, not all population members contribute to the probability vector. Instead, only the top **M** members are used. In practice, **M** is kept extremely small; in these experiments **M** is either 1 or 2. Another dif-

Initialization.

```
for i :=1 to LENGTH do P[i] := 0.5;
```

Generate Samples.

```
while (NOT termination condition)
   for i :=1 to K do
      solution_vectors[i] := generate_vector_with_probabilities (P);
      evaluations[i] :=Evaluate_Solution (solution_vectors[i]);
```

Sort Vectors according to evaluations.

```
   best_solution_vectors =
      sort_solutions_from_best_to_worst (solution_vectors,evaluations);
```

Update Towards Best vectors.

```
   for i := M downto 1 do
      for j :=1 to LENGTH do
            P[j] := P[j] * (1.0 - α) + best_solution_vectors[i][j]* (α);
```

Terminate Search - return best ever found

```
Return the best solution generated throughout the entire search.
```

PBIL CONSTANTS:
K: # of vectors generated before update of the probability vector (200).
α: the learning rate, how fast to exploit the search performed (0.1)
M: number of vectors in the population that are used to update P (1).
LENGTH: # of bits in the solution encoding (problem dependent).

Figure 1: Basic PBIL algorithm for a binary alphabet.

ference is that in BSC, the probability vector is completely regenerated after each generation; in PBIL, the probability vector is incrementally updated after each generation.

As an example of how the PBIL algorithm works, we can examine the values in the probability vector through multiple generations. Consider the following maximization problem: 1.0/|(366503875925.0 - X)|, $0 \leq X < 2^{40}$. Note that 366503875925 is represented in binary as a string of 20 pairs of alternating '01'. The evolution of the probability vector is shown in Figure 2. Note that the most significant bits are pinned to either 0 or 1 very quickly, while the least significant bits are pinned last. This is because during the early portions of the search, the most significant bits yield more information about high evaluation regions of the search space than the least significant bits. The final solution returned is the solution generated during the search that had the highest evaluation.

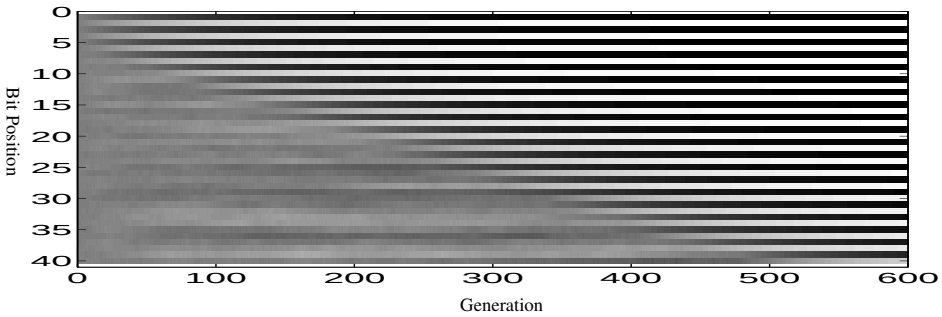

Figure 2: Evolution of the probability vector over successive generations. White represents a high probability of generating a 1, black represents a high probability of generating a 0. Intermediate grey represents probabilities close to 0.5 - equal chances of generating a 0 or 1. Bit 0 is the most significant, bit 40 the least.

Unlike the standard genetic algorithm which uses pair-wise crossover operators, PBIL employs a simple probabilistic model to independently model the values of the bits in the high evaluation solutions. Only the top few members contribute to the next generation's population and the rest of the members are discarded [Baluja, 1995]; this is akin to using a very severe selection criteria in standard genetic algorithms. PBIL and BSC illustrate the similar nature of genetic algorithms and cooperative systems of learning automata [Thathachar & Sastry, 1987].

3 Modeling Dependencies

One of the goals of crossover is to combine "building blocks" from two different solutions into a new child solution. It is clear that neither PBIL nor BSC propagate building blocks in a manner similar to standard GAs, since all parameters are examined independently. As discussed previously, pair-wise crossover must be randomized since the constituent parameters for building blocks are not *a priori* known. Ideally, we would like to know which parameters are correlated so that those building blocks can be propagated intact. The probabilistic model described in this section attempts to capture dependencies, or more specifically mutual information, between the bit parameters. Section 3.1 describes the basics of the proposed algorithm, and Section 3.2 describes the details of generating the probabilistic model.

3.1 Algorithm Basics

The algorithm works in a manner similar to PBIL. After evaluating each member of the current generation, the best members of that population are used to update a probabilistic model from which the next generation's population will be generated. The best few samples from each generation are added into a dataset, termed **S**. Rather than recording the individual members of **S**, our algorithm maintains a sufficient set of statistics in an array $\boldsymbol{A}$ that contains a number $\boldsymbol{A}[X_i=a, X_j=b]$ for every pair of variables X_i and X_j and every combination of binary assignments to a and b. $\boldsymbol{A}[X_i=a, X_j=b]$ is as an estimate of how many recently generated "good" bit strings (from **S**) have bit $X_i=a$ and bit $X_j=b$. Since the probability distribution will gradually shift towards better bit strings, we put more weight on more recently generated bit-strings by decaying the contributions of bitstrings that were previously added to the dataset. All $\boldsymbol{A}[X_i=a, X_j=b]$ are

initialized to some constant C_{init} before the first iteration of the algorithm; this causes the algorithm's first set of bit-strings to be generated from the uniform distribution. See Figure 3.

Initialization

For all bits i and j and all binary assignments to *a* and *b*, initialize $\boldsymbol{A}[X_i=a, X_j=b]$ to C_{init}.

Main Loop

Repeat until Termination Condition is Met

1. Generate probabilistic model based on **A**. See Figure 4.
2. Stochastically generate K bit-strings based on the probabilistic model. Evaluate these bit-strings.
3. Multiply all of the entries in $\boldsymbol{A}$ by a decay factor α between 0 and 1.
4. Choose the best M of the K bit-strings generated in step 2.
   For each bit-string V of these M, add 1.0 to every $\boldsymbol{A}[X_i=a, X_j=b]$ such that V has $X_i=a$ and $X_j=b$.

CONSTANTS (Values used in this study)

C_{init}: Constant used to initialize matrix **A** - Number of examples "seen" at initialization (1000).
K: Number of samples generated in each iteration. This is the population size (200).
M: Number of best samples (from the K generated) that are used to update the statistics (4).
α: How much to decay the effect of older examples (0.99).

Figure 3: Outline of proposed algorithm for using a probabilistic model instead of pair-wise crossover.

The values of $\boldsymbol{A}[X_i=a, X_j=b]$ at the beginning of an iteration may be thought of as specifying a prior probability distribution over "good" bit-strings: the ratios of the values within $\boldsymbol{A}[X_i=a, X_j=b]$ specify the distribution, while the magnitudes of these values, multiplied by α, specify an "equivalent sample size" reflecting how confident we are that this prior probability distribution is accurate.

Like PBIL, we only select the top few members of the population to contribute to the probabilistic model. Like pair-wise operators, we attempt to capture some of the dependencies between the bits. However, as shown in the next section, the dependencies are explicitly captured, rather than being captured through repeated randomized trials.

3.2 Dependency Trees

Given a dataset, **S**, of previously generated good bitstrings, we try to model a probability distribution $P(\mathbf{X}) = P(X_1, ..., X_n)$ of bit-strings of length **n**, where $X_1, ..., X_n$ are variables corresponding to the values of the bits. We try to learn a simplified model $P'(X_1, ..., X_n)$ of the empirical probability distribution $P(X_1, ..., X_n)$ entailed by the bitstrings in **S.** We restrict our model $P'(X_1, ..., X_n)$ to the following form:

$$P'(X_1 \ldots X_n) = \prod_{i=1}^{n} P\left(X_i \middle| \Pi_{X_i}\right) \tag{2}$$

where Π_{X_i} is X_i's single "parent" variable. We require that there be no cycles in these "parent-of" relationships: formally, there must exist some permutation $m = (m_1, ..., m_n)$ of $(1, ..., n)$ such that $(\Pi_{X_i} = X_j) \Rightarrow m(i) < m(j)$ for all *i*. (The "root" node, X_R, will not have a parent node; however, this case can be handled with a "dummy" node X_0 such that $P(X_R \mid X_0)$ is by definition equal to $P(X_R)$.) In other words, we restrict P' to factorizations representable by Bayesian networks in which each node (except X_R) has one parent, *i.e.,* tree-shaped graphs. As will be described later, more complex Bayesian Networks could be used; however, for simplicity trees are used here.

A method for finding the optimal model within these restrictions is given in [Chow and Liu, 1968]. A complete weighted graph **G** is created in which every variable X_i is represented by a corresponding vertex V_i, and in which the weight W_{ij} for the edge between vertices V_i and V_j is set to the *mutual information* $I(X_i,X_j)$ between X_i and X_j:

$$I(X_i, X_j) = \sum_{a,b} P(X_i = a, X_j = b) \cdot \log \frac{P(X_i = a, X_j = b)}{P(X_i = a) \cdot P(X_j = b)} \tag{3}$$

The empirical probabilities of the form $P(X_i = a)$ and $P(X_i = a, X_j = b)$ are computed directly from **S** for all combinations of i, j, a, and b (a & b are binary assignments to X_i & X_j). Once these edge weights are computed, the maximum spanning tree of **G** is calculated, and this tree determines the structure of the network used to model the original probability distribution. Since the edges in **G** are undirected, a decision must be made about the directionality of the dependencies with which to construct P′; however, all such orderings conforming to the restrictions described earlier model identical distributions. Among all trees, this algorithm produces a tree that maximizes:

$$\sum_{i=1}^{n} I(X_{m(i)}, X_{m(p(i))}) \tag{4}$$

which in turn minimizes the Kullback-Leibler divergence, $D(P \| P')$, between P (the true empirical distributions exhibited by **S**) and P′ (the distribution modeled by the network):

$$D(P \| P') = \sum_{X} P(X) \log \frac{P(X)}{P'(X)} \tag{5}$$

As shown in [Chow & Liu, 1968], this produces the tree-shaped network that maximizes the likelihood of **S**. This tree generation algorithm, summarized in Figure 4, runs in time $O(n^2)$, where **n** is the number of bits in the solution encoding.

Generate an optimal dependency tree:
- Set the root to an arbitrary bit X_{root}
- For all other bits X_i, set bestMatchingBitInTree[X_i] to X_{root}.
- While not all bits have been added to the tree:
  - Of all the bits not yet in the tree, pick bit X_{add} with the maximum mutual information $I(X_{add}$, bestMatchingBitInTree[X_{add}]), using **A** (which contain sufficient statistics for **S**) to estimate the relevant probability distributions.
  - Add X_{add} to tree, with bestMatchingBitInTree[X_{add}] as parent.
  - For each bit X_{out} not in the tree, if $I(X_{out}$, bestMatchingBitInTree[X_{out}]) < $I(X_{out}, X_{add})$. then set bestMatchingBitInTree[X_{out}]=X_{add}.

Figure 4: Procedure for generating the dependency tree.

Once we have generated a dependency tree modeling $P(X_1, ..., X_n)$, we use it to generate **K** new bitstrings. Each bitstring is generated in O(n) time during a depth-first traversal of the tree, and then evaluated. The best **M** of these bitstrings are selected and effectively added to **S** by updating the counts in **A**. Based on the updated **A**, a new dependency tree is created, and the cycle is continued.

3.3 Summary

The first extension to PBIL that captured pair-wise dependencies was termed *Mutual Information Maximization for Input Clustering (MIMIC)* [De Bonet *et al.*, 1997]. MIMIC used a greedy search to generate a chain in which each variable is conditioned on the previous variable. The first variable in the chain, X_1, is chosen to be the variable with the lowest unconditional entropy $H(X_1)$. When deciding which subsequent variable X_{i+1} to add to the chain, MIMIC selects the variable with the lowest conditional entropy $H(X_{i+1} \mid X_i)$. While MIMIC was restricted to a greedy heuristic for finding chain-based models, the algorithm described in this paper uses a broader class of models, trees, and finds the optimal model in the class.

Example dependency graphs shown in Figure 5 illustrate the types of probability models learned by PBIL, a dependency-chain algorithm similar to MIMIC, and our dependency tree algorithm. We use Bayesian network notation for our graphs: an arrow from node X_p to node X_c indicates that X_c's probability distribution is conditionally dependent on the value of X_p. These models were learned while optimizing a noisy version of a two-color graph coloring problem (shown in Figure 5a) in which there is a 0.5 probability of adding 1 to the evaluation function for every edge constraint satisfied by the candidate solution.

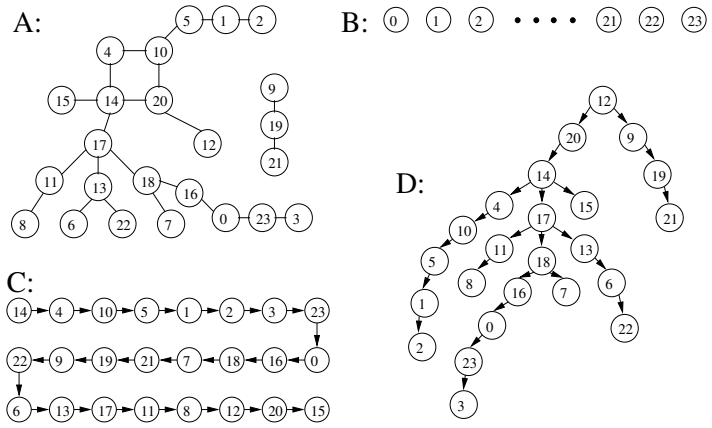

Figure 5: A: A noisy two-color graph coloring problem. B: the empty dependency graph used by PBIL. C: the graph learned by our implementation of the dependency chain algorithm. D: the graph learned by our dependency tree algorithm.

Note that the dependency tree algorithm is able to discover the underlying structure of the graph, in terms of which bits are dependent on each other (as shown in Figure 5D). This type of information could be invaluable when trying to find building blocks to propagate to the next generation's population.

4 Empirical Comparisons

Recently, there have been a number of empirical comparisons of pool-wise and pair-wise operators in genetic search, see for example [Juels, 1996][Baluja, 1997][Greene, 1996][Eshelman et. al, 1996]. In many benchmark problems, pool-wise operators perform as well, or better than, pair-wise operators. However, when compared on problems which are designed to exhibit large amounts of interparameter dependencies the performance outcomes are less predictable [Eshelman et. al, 1996]. In this section, we present a large set of empirical tests on problems which exhibit obvious inter-parameter dependencies. We hope to address two questions. First, can pool-wise operators perform as well as pair-wise operators on these problems? Second, if we use more expressive dependency models, will optimization improve (for example, do trees perform better than chains, and do both trees and chains perform better than models without dependencies, such as used in PBIL?)

We present an empirical comparison of four algorithms. For each problem, each algorithm was allowed 2000 generations, with 200 evaluations per generation. All algorithms were run multiple times, as specified with the problem description. The algorithms compared are described below:

Trees: Pool-wise recombination with a tree-based probabilistic model: This is an implementation of the algorithm described in this paper. C_{init} is set to 1000, and the decay rate α is set to 0.99. (Other decay rates were empirically tested on a single problem; the value of 0.99 yielded the best results for both Trees and Chains.) K=200 samples are created per generation. The top 2% (M=4 samples) are used to update the statistics (the $\boldsymbol{A}$ matrix). All parameters were held constant through all runs.

Chains: Pool-wise recombination with a chain-based probabilistic model: This algorithm is identical to the Tree algorithm, except the dependency graphs are restricted to chains, the type of dependency graph used in MIMIC. All of the other parameters' values are the same as for Trees.

PBIL: Pool-wise Recombination with no dependencies: The algorithm and parameters are shown in Figure 1. The only addition is a mutation operator [Baluja, 1995]. This operator affects each entry in **P** with probability 0.02 per generation. When mutated, an entry is shifted in a random direction, with 0.05 magnitude.

Genetic Algorithm with Pair-wise Recombination: Except when noted otherwise, the GA used had the following parameters: 80% crossover rate, uniform crossover [Syswerda, 1989], mutation rate 0.1% per bit, elitist selection (the single best solution from generation_g replaces the worst solution in generation_{g+1}), and population size 200. The GAs used fitness proportional selection. With this method of selection, the GA is the only algorithm tested which is sensitive to the magnitudes of the evaluation (all the other algorithms work with ranks). To help account for this, the lowest evaluation in each generation is subtracted from all the evaluations before probabilities of selection are calculated. Rank-based selection metrics were also explored; however, they did not reveal consistently better results. On problems on which these settings did not work well, multiple GAs were attempted with many different parameter settings. For these problems, the performance for several GAs are given.

4.1 Checkerboard

The object is to create a checkerboard pattern of 0's and 1's in an **N**x**N** grid. Only the primary four directions are considered in the evaluation. For each bit in an (N-2)(N-2) grid centered in an NxN grid, +1 is added for each of the four neighbors that are set to the opposite value. The maximum evaluation for the function is (N-2)(N-2)*4. In these experiments N=16, so the maximum evaluation is 784. 30 trials were conducted for each algorithm. The distribution of results are shown in Figure 6.

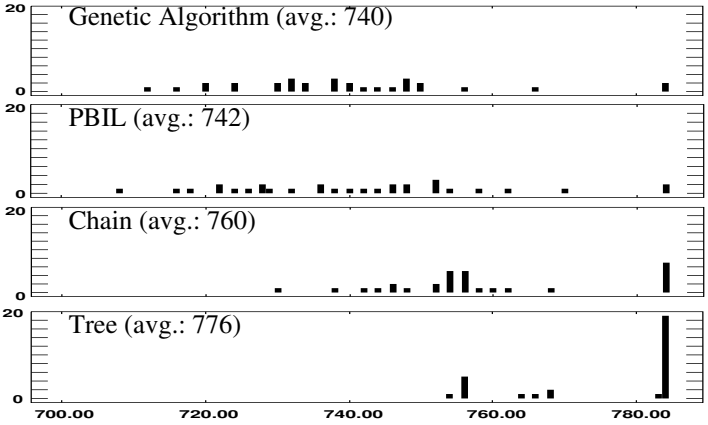

Figure 6: 30 trials of the checkerboard problem were tried per algorithm. Shown is a histogram of the final results obtained by each algorithm. Note that the optimal solution has an evaluation of 784. The average evaluation is also given.

4.2 The Peaks Set of Problems

This set of three problems is based on the four-peaks problem, which was originally presented in [Baluja & Caruana, 1995]. The original four-peaks problem is defined as follows. Given an input vector **X**, which is composed of **N** binary elements, maximize the following:

$$FourPeaks(T, X) = MAX(head(1, X), tail(0, X)) + Reward(T, X)$$

$$Reward(T, X) = \begin{cases} 100 & \text{if} \quad (head(1, X) > T) \wedge (tail(0, X) > T) \\ 0 & \text{otherwise} \end{cases}$$

$$head(b, x) = \text{number of contiguous leading bits set to b in X}$$

$$tail(b, x) = \text{number of contiguous trailing bits set to b in X}$$

Fitness is maximized if a string is able to get both the REWARD of 100 and if the length of one of head(1,X) or tail(0,X) is as large as possible. The four peaks problems have two suboptimal local optima with fitnesses of N (independent of T). One of these is at tail(0,X)=N, head(1,X)=0 and the other is at tail(0,X)=0, head(1,X)=N. Hill-climbing can quickly get trapped in these local optima. By increasing T, the basins of attraction surrounding the inferior local optima increase in size exponentially while the basins around the global optima decrease at the same rate.

Two modifications of the four-peaks problem are also explored. In the first, "Six-Peaks" [De Bonet et. al,

1997], two more peaks are added by allowing a string of 0's to appear either at the beginning or end of the vector, with a string of 1's appearing at the opposite end. Optimization methods may oscillate between multiple answers, since the two optimal solutions require opposite values in many of the bit positions. In the second modification, "Continuous-Peaks", rather than forcing 0's and 1's to be at opposite ends of the solution string, they are allowed to form anywhere in the string. For this problem, a reward is given when there are greater than T contiguous bits set to 0, and greater than T contiguous bits set to 1.

Since the four-peaks problem has previously been studied, we used the same parameter settings as in the previous studies – PBIL (mutation rate = 0.0, update from the top two vectors). For the GA, the previously used settings did not work as well (perhaps since the total bit length is less than the previously studied version); therefore, four GAs were experimented with here. All used *one-point* crossover instead of uniform crossover, since these problems were custom-designed to benefit from this operator. GA1 used a population size of 200, with mutation rate 0.001; GA2 used a population size of 200, with mutation rate 0.01; GA3 used a population size of 500 with mutation rate 0.001; and GA4 used a population size of 500 with mutation rate 0.01. GA4 performed the best, so all the GA results are reported with it in all of the problems. Note that since each algorithm was allowed the same number of generations, GA3 & GA4 used 500*2000 evaluations, while all other algorithms used 200*2000. Results are shown in Figure 7.

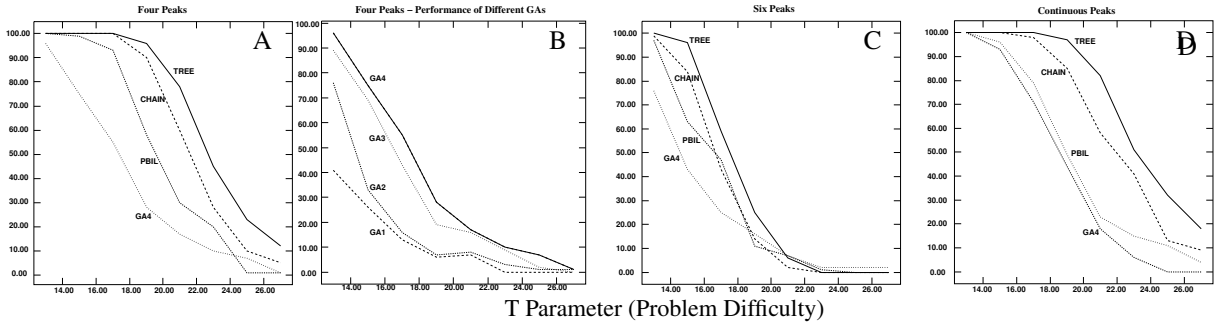

Figure 7: Number of runs (out of 100) that the reward was found (A) Four Peaks, (C) Six Peaks, (D) Continuous Peaks. In (B) the performance of the different GAs is measured on Four Peaks.

4.3 Real-Valued Functions

It is common to use evolutionary algorithms on real-valued functions which have been discretized to an arbitrary precision. In this section, two multi-dimensional optimization problems are tested. Both of the problems were chosen because of the clear parameter interactions. In both of these problems, the GAs used previously performed poorly. Often GA-based optimization benefits from higher mutation rates when the parameters are encoded in Gray code. The GA's parameters were hand-tuned; different amounts of mutation worked best with each problem. A total of three GAs were tried, GA1 with mutation rate 0.001, GA2 with mutation rate 0.005, and GA3 with mutation rate 0.02.

4.4 Problem 1: Summation Cancellation

In this problem, the parameters (s_1, ..., s_N) in the beginning of the solution string have a large influence on the quality of the solution. The goal is to minimize the magnitudes of cumulative sums of the parameters. Small changes in the first parameters can cause large changes in the evaluation. The evaluation is:

$$-0.16 \le s_i \le 0.15 \quad i = 1 \ldots N \qquad y_i = s_i + y_{i-1} \quad i = 2 \ldots N \quad y_1 = s_1 \qquad C = \frac{1}{100000} \qquad f = \frac{1.0}{C + \left| \sum_1^N |y_i| \right|}$$

To test the effects of increasing the dimensionality of the problem, we tried varying the dimensions (N) between 10 and 30. For each number of dimensions, 100 runs were executed. In all of these problems, each

parameter was represented with 5 bits, encoded in standard base-2, with the values uniformly spaced between -0.16 and 0.15. The results are shown in Table I. We repeated the experiments (again with 100 trials per dimension setting) using Gray coding. The results are presented in Table I(right). Note that because of the problem specification, small changes in the sum in the denominator of the evaluation function can lead to enormous differences in evaluation.

Table I: Summation Cancellation- Average best solution found over 25 runs. (Goal: Maximize Value)

Parameters (Bits)	Standard Binary Encoding							Gray-Coding					
	Tree	Chain	PBIL	GA1 (0.001)	GA2 (0.005)	GA3 (0.02)		Tree	Chain	PBIL	GA1 (0.001)	GA2 (0.005)	GA3 (0.02)
10 (50)	**53.7**	34.1	21.0	13.0	20.1	23.6		92008	92008	**100000**	2038	98002	**100000**
12 (60)	**29.3**	24.1	16.1	9.3	14.5	18.7		42053	54041	**100000**	2020	95000	**100000**
15 (75)	16.8	**16.9**	11.2	5.8	11.3	14.7		4047	7044	**100000**	1008	81009	**100000**
17 (85)	**13.8**	13.7	9.5	4.7	8.9	11.9		26.0	26.7	98001	6.6	70010	**100000**
20 (100)	**11.0**	10.9	7.5	3.3	6.9	8.9		14.4	13.7	**94005**	5.1	47022	93001
23 (115)	**8.5**	6.4	6.0	2.4	5.7	4.6		8.1	7.6	**90008**	3.7	24025	10011
25 (125)	**6.3**	4.2	5.0	2.2	5.1	3.0		5.0	5.3	**73019**	2.9	15024	6.0
27 (135)	4.2	3.0	**4.4**	1.9	**4.4**	2.3		4.4	4.0	**62028**	2.9	19027	3.4
30 (150)	2.6	1.9	3.6	1.6	**3.7**	1.6		3.0	2.9	**32039**	2.2	3021	2.2

4.5 Problem 2: Solving a System of Linear Equations

The goal of this problem is to solve for X in $DX=B$, when given a matrix D and vector B. Although there are many standard techniques for solving this problem, it also serves as a good benchmark for combinatorial optimization algorithms [Eshelman et. al, 1996]. Since all of the parameters are dependent on each other, optimization is difficult. In this study, D is a 9x9 matrix, B and X are vectors of length 9. Each value in X is an integer between [-256, 255]. The solution encoding is 81 bits. D is randomly generated, and B set so that there is guaranteed to be a solution to $DX=B$. The results for the algorithms are presented in Table II; a total of 9 problems were tried with 25 runs per problem. The goal is to minimize the summed parameter-by-parameter absolute difference of DX and B.

Table II: System of Linear Equations: Avg. value of best solution over 25 runs. (Goal: Minimize Error)

Problem	Standard Base-2 Encoding							Gray Coding					
	Tree	Chain	PBIL	GA1 (0.001)	GA2 (0.005)	GA3 (0.02)		Trees	Chain	PBIL	GA1 (0.001)	GA2 (0.005)	GA3 (0.02)
1	648	**520**	2248	2622	2136	3149		**335**	341	1195	1015	777	1722
2	**721**	1537	3825	5069	2447	3958		**778**	799	1952	1734	1275	2699
3	**405**	544	2981	4338	1665	4063		387	**346**	1011	832	585	2342
4	**706**	1347	3204	3702	1949	4136		**830**	911	2120	1661	1226	2588
5	**848**	1255	3031	3783	2376	3628		**736**	851	1985	1555	953	2350
6	**313**	393	2306	2620	1657	2770		**330**	335	621	550	587	1727
7	**692**	1034	3331	3382	1681	4044		809	**723**	1987	1543	1153	2590
8	**708**	1029	2996	3098	2441	3943		684	**641**	1753	1616	945	2063
9	**577**	904	3135	3624	2075	3738		333	**308**	1209	1406	954	2263

5 Conclusions & Future Work

We have presented results on a diverse set of problems which have a large number of interparameter dependencies. The pool-wise operators performed as well as, and often better than, the pair-wise operators

in the majority of problems. One problem in which pair-wise operators performed slightly better than pool-wise is the summation-cancellation problem when encoded in Gray-code. As can be seen from the results with the standard GA, the mutation probability has a large impact on performance. The version of the Tree and Chain algorithms used did not have any form of mutation; however, they could easily be extended to include mutation-like operations.

Perhaps the most interesting result is that the performance of the combinatorial optimization algorithms consistently improved as the accuracy of their statistical models increased. Trees generally performed better than chains, and chains generally performed better than independent models as used in PBIL. This suggests the possibility of using even more complex models. Unfortunately, when we move toward models in which variables can have more than one parent variable, the problem of finding an optimal network with which to model a set of data becomes NP-complete [Chickering, *et al.*, 1995]. However, search heuristics have been developed for automatically learning Bayesian networks from data (for example [Heckerman, *et al.*, 1995]). A common approach is to perform hill-climbing over network structures, starting with a relatively simple network. A second potential drawback is that more complex model of the "good" bit-strings generated may fit the data too tightly, thereby forcing the algorithm to concentrate too heavily on exploitation rather than exploration.

In this paper, we have not attempted to propose a complete optimization "system". For a complete system, there are many heuristics, such as mutation, local hillclimbing, elitist selection etc., that can easily be incorporated. Instead, we have attempted to show that pool-wise matings can be competitive with pair-wise matings, even on problems in which there is a high degree of interparameter dependency. Previous pool-wise mating schemes were not effective on these problems because the appropriate statistics were not extracted from the population. This work makes a step towards *explicitly* capturing which parameters should be modeled together, to make the transfer of building blocks more effective.

References

Baluja, S. (1997) "Genetic Algorithms and Explicit Search Statistics," *Advances in Neural Information Processing System*s 9, 1996. Mozer, M.C., Jordan, M.I., & Petsche, T. (Eds). MIT Press.

Baluja, S. (1995),"An Empirical Comparison of Seven Iterative and Evolutionary Heuristics for Static Function Optimization" Technical Report CMU-CS-95-193, Carnegie Mellon University, Pittsburgh, PA.

Baluja, S. & Caruana, R. (1995), "Removing the Genetics from the Standard Genetic Algorithm," In (eds) Prieditis, A., Russel, S. *The International Conference on Machine Learning, 1995 (ML-95)*. Morgan Kaufmann Publishers. San Mateo, CA. pp. 38-46.

Chickering, D., Geiger, D., and Heckerman, D. (1995) "Learning Bayesian networks: Search methods and experimental results," *Proc.of Fifth Conference on Artificial Intelligence and Statistics*

Chou. C. and Liu, C. (1968) Approximating discrete probability distributions with dependence trees. *IEEE Trans. on Info. Theory*, 14:462-467.

De Bonet, J., Isbell, C., and Viola, P. (1997) "MIMIC: Finding Optima by Estimating Probability Densities," *Advances in Neural Information Processing Systems*, 1996. Mozer, M.C., Jordan, M.I, & Petsche, T. (Eds).

Eshelman, L.J., Mathias, K.E., Schaffer, J.D. "Convergence Controlled Variation", in *Proc. Foundations of Genetic Algorithms 4*. Morgan Kaufmann Publishers, San Mateo, CA.

Greene, J.R. (1996) "Population-Based Incremental Learning as a Simple Versatile Tool for Engineering Optimization". In *Proceedings of the First International Conf. on EC and Applications*. pp. 258-269.

Heckerman, D., Geiger, D., and Chickering, D. (1995) "Learning Bayesian networks: The combination of knowledge and statistical data," *Machine Learning* 20:197-243.

Hertz, J., Krogh A., & Palmer R.G. (1991), *Introduction to the Theory of Neural Computing.* Addison-Wesley, Reading, MA.

Hohfeld, M. & Rudolph, G. (1997) "Towards a Theory of Population-Based Incremental Learning", *International Conference on Evolutionary Computation*. pp. 1-5.

Juels, A. (1996) *Topics in Black-box Combinatorial Optimization*. Ph.D. Thesis, University of California - Berkeley.

Kvasnica, V., Pelikan, M, Pospical, J. "Hill Climbing with Learning (An Abstraction of Genetic Algorithm). *In Proceedings of the First International Conference on Genetic Algorithms (MENDEL, '95)*. pp. 65-73.

Pearl, J. (1988) *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann.

Syswerda, G. (1989) "Uniform Crossover in Genetic Algorithms," *Int. Conf. on Genetic Algorithms 3.* 2-9.

Syswerda, G. (1993) "Simulated Crossover in Genetic Algorithms," in (ed.) Whitley, D.L., Foundations of Genetic Algorithms 2, Morgan Kaufmann Publishers, San Mateo, CA. 239-255.

Thathachar, M, & Sastry, P.S. (1987) "Learning Optimal Discriminant Functions Through a Cooperative Game of Automata", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 17, No. 1.