Using Optimal Dependency-Trees for Combinatorial Optimization: Learning the Structure of the Search Space

Shumeet Baluja[1,2]

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

Scott Davies[2]

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

Abstract

Many combinatorial optimization algorithms have no mechanism for capturing inter-parameter dependencies. However, modeling such dependencies may allow an algorithm to concentrate its sampling more effectively on regions of the search space which have appeared promising in the past. We present an algorithm which incrementally learns pairwise probability distributions from good solutions seen so far, uses these statistics to generate optimal (in terms of maximum likelihood) dependency trees to model these distributions, and then stochastically generates new candidate solutions from these trees. We test this algorithm on a variety of optimization problems. Our results indicate superior performance over other tested algorithms that either (1) do not explicitly use these dependencies, or (2) use these dependencies to generate a more restricted class of dependency graphs.

1. INTRODUCTION

When performing combinatorial optimization, we wish to concentrate the generation of candidate solutions to regions of the solution space which have a high probability of containing solutions better than those previously sampled. Most optimization algorithms attempt to do this by searching around the location of the best previous solutions. There are many ways to do this:

Gradient Descent/Simulated Annealing: Only generate candidate solutions neighboring a single current solution. These algorithms make no attempt to model the dependence of the solution quality upon particular solution parameters.

Genetic Algorithms (GAs): attempt to implicitly capture dependencies between parameters and the solution quality by concentrating samples on combinations of high-performance members of the current population, through the use of the *crossover* operator. Crossover combines the information contained within pairs of selected "parent" solutions by placing random subsets of each parent's bits into their respective positions in a new "child" solution. Note that no explicit information is kept about which groups of parameters contributed to the quality of candidate solutions. Therefore, the crossover is randomized, and must be performed repeatedly, as most of the crossovers yield unproductive results.

Population-Based Incremental Learning (PBIL) [Baluja, 1997]*:* PBIL explicitly captures the dependencies between individual solution parameters and solution quality. PBIL uses a vector of unconditional probabilities to model a simple probability distribution over all bit-strings. The probability vector is adjusted to increase the likelihood of generating high-evaluation solutions. To do this, as the search progresses, the probability vector is moved towards the highest-performing individual in each generation. No attempt is made to model the inter-parameter dependencies. However, for many optimization problems drawn from the GA literature, the use of these statistics alone allows PBIL to outperform standard GAs and hill-climbers.

Mutual Information Maximization for Input Clustering (MIMIC) [De Bonet, et al., 1997]*:* This work extends PBIL by (1) capturing some of the pairwise dependencies between the solution parameters, and (2) providing a statistically meaningful way of updating the distribution from which samples are generated. MIMIC maintains at least the top N% of all previously generated solutions, from which it calculates pair-wise conditional probabilities. MIMIC uses 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 $H(X_{i+1} \mid X_i)$. After creating the full chain, it randomly generates more samples from the distribution specified by this chain. The entire process is then repeated.

The work presented in this paper extends MIMIC. Our algorithm maintains pairwise statistics similar to those employed by MIMIC. However, while MIMIC was restricted to a greedy heuristic for finding chain-based models, our algorithm uses a broader class of models, and finds the optimal model within that class. The algorithm will be described in Section 2. The empirical results described in Section 3 demonstrate that, in most cases examined, the performance of the combinatorial optimization algorithms consistently improves as the accuracy of the statistical model increases. In Section 4, we draw our conclusions and discuss avenues of future research.

2. ALGORITHM

We wish to model a probability distribution $P(X_1, ..., X_n)$ over bit-strings of length n, where $X_1, ..., X_n$ are variables corresponding to the values of the bits. Suppose we restrict our model $P'(X_1, ..., X_n)$ to models of the following form:

$$P'(X_1 \ldots X_n) = \prod_{i=1}^{n} P(X_{m(i)} | X_{m(p(i))}) \qquad (1)$$

Here, $m = (m_1, ..., m_n)$ is some unknown permutation of (1, ..., n); p(i) maps the integers $0 < i \leq n$ to integers $0 \leq p(i) < i$; and $P(X_i \mid X_0)$ is by definition equal to $P(X_i)$ for all i. In other words, we restrict P′ to factorizations in which the conditional probability distribution for any one bit depends on the value of at most one other bit. (In Bayesian network terms, this means we are restricting our models to networks in which each node can have at most one parent.)

A method for finding the optimal model within these restrictions is presented 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. The edges in the maximum spanning tree of G determine an optimal set of (n-1) conditional probabilities with which 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 Equation 1 model identical distributions. Among all trees, this algorithm produces the tree which maximizes the likelihood of the data when the algorithm is applied to empirical observations drawn from any unknown distribution.

We employ a version of this algorithm for combinatorial optimization as follows. We incrementally learn pairwise statistics from previously seen "good" bit-strings; using a version of Chow and Liu's algorithm, we determine optimal subsets of these statistics with which to create model probability distributions $P'(X_1, ..., X_n)$ of the form assumed in Equation 1. These distributions are used to generate new candidate bit-strings which are then evaluated. The best bit-strings are used to update the pairwise statistics; these statistics are used to generate another dependency tree; and so forth, until the algorithm's termination criteria are met.

Our algorithm maintains an array $\boldsymbol{A}$ containing 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, where $\boldsymbol{A}[X_i=a, X_j=b]$ is as an estimate of how many recently generated "good" bit strings have had 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. 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.

We calculate unconditional and conditional probabilities $P(X_i)$ and $P(X_i, X_j)$ from the current values of $\boldsymbol{A}[X_i=a, X_j=b]$. From these probabilities we calculate the mutual information, $I(X_i, X_j)$, between all pairs of variables X_i and X_j:

$$I(X_i, X_j) = \sum_{a,b} P(X_i = a, X_j = b) \cdot \log \frac{P(X_i = a, X_j = b)}{P(X_i = a) \cdot P(X_j = b)} \qquad (2)$$

We then create a dependency tree containing an optimum set of n-1 dependencies. To do this, first, we select an arbitrary bit X_{root} to place at the root of the tree. Then, we add all other bits to the tree as follows: we find the pair of bits X_{in} and X_{out} – where X_{out} is any bit not yet in the tree, and X_{in} is any bit already in the tree – with the maximum mutual information $I(X_{in}, X_{out})$. We add X_{out} to the tree, with X_{in} as its parent, and repeat this process until all the bits have been added to the tree. By keeping track of which bit inside the tree has the highest mutual information with each bit still outside the tree, we can perform the entire tree-growing operation in $O(n^2)$ time. Because our algorithm is a variant of Prim's algorithm for finding minimum spanning trees [Prim, 1957], it can easily be shown that it constructs a tree that maximizes the sum

$$\sum_{i=1}^{n} I(X_{m(i)}, X_{m(p(i))}) \qquad (3)$$

which in turn minimizes the Kullback-Leibler divergence $D(P||P')$, as shown in [Chow & Liu, 1968]. For all i, our algorithm sets m(i) = j if X_j was the i^{th} bit added to the tree, and sets p(i) = j if the j^{th} bit to be added to the tree was the parent of the i^{th} bit to be added. (m(p(1)) is X_0, a

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 a 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:
     - Out of all the bits not yet in the tree, pick the bit X_{add} with the maximum mutual information I(X_{add}, bestMatchingBitInTree[X_{add}]), using $\boldsymbol{A}$ to estimate the relevant probability distributions.
     - Add X_{add} to the tree, with bestMatchingBitInTree[X_{add}] as its parent.
     - For each bit X_{out} still not in the tree, compare I(X_{out}, bestMatchingBitInTree[X_{out}]) with I(X_{out} , X_{add}). If I(X_{out}, X_{add}) is greater, set bestMatchingBitInTree[X_{out}] = X_{add}.

2. Generate K bit-strings based on the joint probability encoded by the dependency tree generated in the previous step. 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 S of these M, add 1.0 to every $\boldsymbol{A}[X_i=a, X_j=b]$ such that S has $X_i=a$ and X_j

Figure 1: Basic Algorithm. For each bit X_{out} not yet in the tree, bestMatchingBitInTree[X_{out}] maintains a pointer to a bit in the tree with which X_{out} has maximum mutual information.

dummy node with a constant value.) Our modeled probability distribution $P'(X_1, ..., X_n)$ is then specified by Equation 1. Among all distributions of the same form, this distribution maximizes the likelihood of the data when the data is a set of empirical observations drawn from any unknown distribution.

Once we have generated a model dependency tree specifying $P'(X_1, ..., X_n)$, we use it to generate K new bit-strings. Each bit-string is generated in O(n) time during a depth-first traversal of the tree, and then evaluated; the best M are chosen to update $\boldsymbol{A}$. First, we multiply all entries in $\boldsymbol{A}$ by a decay factor, α, between 0 and 1; this puts more weight on the more recently generated bit-strings. Then, we add 1.0 to $\boldsymbol{A}[X_i=a, X_j=b]$ for every bit-string out of the best M in which bit $X_i=a$ and bit $X_j=b$. After updating $\boldsymbol{A}$, we recompute the mutual information between all pairs of variables, use these to generate another dependency tree, use this tree to generate more samples, and continue the cycle until a termination condition is met. High-level pseudo-code is shown in Figure 1.

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.[1]

Unlike the algorithm used in [De Bonet, et al., 1997], we do not maintain a record of data from previous generations. This eliminates the time and memory that would be required to keep a large number of old bit-strings sorted according to their evaluations. Our method is also closer to that employed by PBIL, which make empirical comparisons with PBIL more meaningful. On the other hand, the relationship between the entries in $\boldsymbol{A}[X_i=a, X_j=b]$ and the data the algorithm has generated in the past is less clear. In this paper, we compare the performance of our dependency tree algorithm with an algorithm similar to MIMIC that produces chain-shaped dependency graphs. Both algorithms update the values of $\boldsymbol{A}[X_i=a, X_j=b]$ in the manner described above. In future work, we will investigate the effects of explicitly maintaining and using previously generated solutions.

Example dependency graphs shown in Figure 2 illustrate the types of probability models learned by PBIL, our dependency chain algorithm, 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 in which there is a 0.5 probability of adding 1 to the evaluation function for every edge constraint satisfied by the candidate solution. It is interesting to note how closely the tree algorithm models the structure of the problem graph.

1. The values of $\boldsymbol{A}$ are, in some respects, similar to the coefficients of Dirichlet distributions — a correspondence which suggests the possibility of using Bayesian scoring metrics in place of information-theoretic ones in the future, such as those used in [Cooper and Herskovits, 1992] and [Heckerman, et al., 1995].

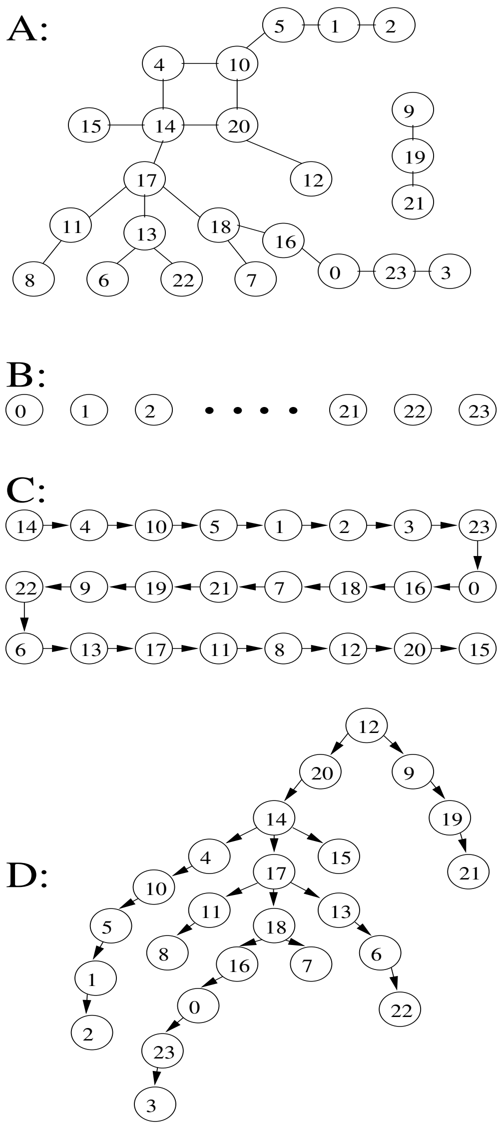

Figure 2: A: A noisy two-color graph coloring problem. B: the empty dependency graph used by PBIL. C: the graph learned by our implementation of the dependency chain algorithm. D: the graph learned by our dependency tree algorithm.

3. EMPIRICAL COMPARISONS

In this section, we present an empirical comparison of four algorithms on five classes of problems. For each problem, each algorithm was allowed 2000 generations, with 200 evaluations per generation. All algorithms were run multiple times, as specified in the individual problem descriptions. To measure the significance of the differences between the Tree and Chain algorithms, the Mann-Whitney rank-sum test is used. This is a non-parametric equivalent of the standard two-sample *t*-test. The algorithms compared are described below:

Trees: 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 ***A*** matrix). All parameters were held constant through all of the runs.

Chain: This algorithm is identical to the Tree algorithm, except the dependency graphs are restricted to chains, the type of dependency graph used in the MIMIC algorithm. All of the other parameters' values are the same as for Trees, and were held constant in all of the runs.

PBIL: This is the basic version of the PBIL algorithm described in [Baluja, 1995]. The algorithm and parameters are shown in Figure 3. The parameters were only changed on the peaks problems; parameter settings for the peaks problem were taken from [Baluja & Caruana, 1995].

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

while (NOT termination condition)
   for i:=1 to SAMPLES do
          sample_vectors[i] := generate_according_to_probabilities (P);
          evaluations[i] := evaluate(sample_vectors[i]);
   best_vector := vector_with_best_eval (sample_vectors, evaluations);
   for i:=1 to LENGTH do
          P[i] := P[i]*(1.0-LR) + best_vector[i]*(LR);
   for i:=1 to LENGTH do
          if (random (0..1) < MUT_RATE);
               direction := random (0 or 1);
               P[i] := P[i]*(1.0-MUT_SHIFT) + direction*(MUT_SHIFT)
```

PBIL: USER DEFINED CONSTANTS (Values Used in this Study):
SAMPLES: number of vectors generated before update of the probability vector (200).
LR: the learning rate, how fast to exploit the search performed (0.1).
LENGTH: the number of bits in a generated vector (problem specific).
MUT_RATE: chances of mutation (0.02)
MUT_SHIFT: magnitude of mutation (0.05).

Figure 3: PBIL algorithm for a binary alphabet, adapted from [Baluja & Caruana, 1995].

Genetic Algorithm (GA): Except when noted otherwise, the GA used in this paper had the following parameters: 80% crossover rate (the chances that a crossover occurs with two parents; if crossover does not occur, the parents are copied to the children without modification); uniform crossover (in each child, there is a 50% probability of inheriting the bit value from parent A, 50% from parent B [Syswerda, 1989]), mutation rate 0.1% (probability of mutating each bit), elitist selection (the best solution from generation_g replaces the worst solution in generation_{g+1}) [Goldberg, 1989], and population size 200. The GAs used fitness proportional selection (this means that the chances of selecting a member of the population for recombination is proportional to its evaluation). 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. Other rank-based selection metrics were also explored; however, they did not reveal consistently better results. Of all the algorithms, the most effort was spent tuning the GA's parameters. Additionally, for many problems multiple GAs were attempted with different parameter settings. For these problems, the performance for several GAs are given.

3.1 Checkerboard

This problem was originally suggested by [Boyan, 1993]. The object is to create a checkerboard pattern of 0's and 1's in an NxN 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; therefore, the problem size is 256 bits, and the maximum evaluation is 784. 30 trials were conducted for each algorithm. The distribution of results are shown in Figure 4.

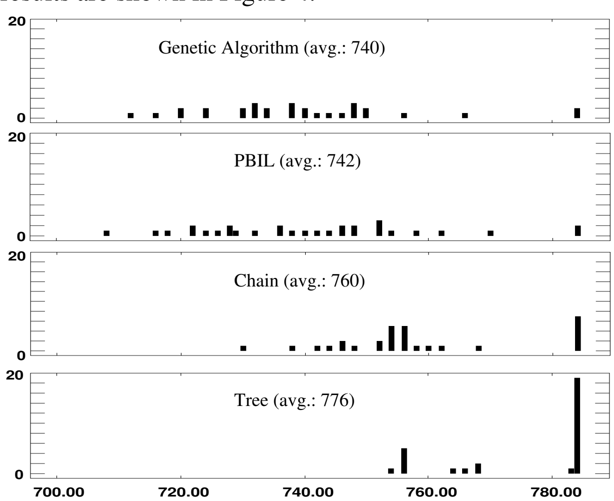

Figure 4: Distribution of results for the Checkerboard Problem.

3.2 Tree-Max

In this problem set, we randomly generate probability distributions of the form shown in Equation 1. For a given problem, a single tree-shaped network is generated and the probabilities associated with the nodes in these networks are randomly generated. The value of a given bit-string is the probability with which the randomly generated network would generate that bit-string; the task is to find a bit-string that maximizes this value. This problem is designed to test the extent to which other algorithms can capture the same statistics for which our tree algorithm is specifically designed. We show results for three different classes of problems: in the first, the probabilities associated with the nodes are chosen uniformly between 0 and 1; in the second, all probabilities are chosen to lie either between 0 and 0.2 or between 0.8 and 1.0; in the third, all probabilities are chosen uniformly between 0.4 and 0.6. 200 problems from each of the three distributions were tried; the size of the problem was 120 bits. The results, in Figure 5, show that Chains can capture many of the dependencies which Trees capture, but PBIL and GA cannot.

3.3 The Peaks Set of Problems

This set of three problems is based on the four-peaks problem, which was originally presented in [Baluja and 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: 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 also 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 will quickly get trapped in these local optima. For hill-climbing to work well here, it must repeatedly make "correct" decisions while searching large plateaus; this is extremely unlikely in practice. 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.

A version of the modification to the four-peaks problem suggested by [De Bonet et. al, 1997] is tried here. In this problem, "Six-Peaks", two more peaks are added. Optimization methods may oscillate between multiple answers, since the two optimal solutions require opposite values in many of the bit positions.

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

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

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

$$SixPeaks(T, X) = MAX(head(X_0, X), tail(\neg X_0, X)) + R(T, X)$$

$$R(T, X) = \begin{cases} 100 & \text{if} \quad (head(X_0, X) > T) \wedge (tail(\neg X_0, X) > T \\ 0 & \text{otherwise} \end{cases}$$

The final version of the peaks problem which was examined was the "Continuous-Peaks" problem. 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.

For all of the peaks problems, N=80 (the number of

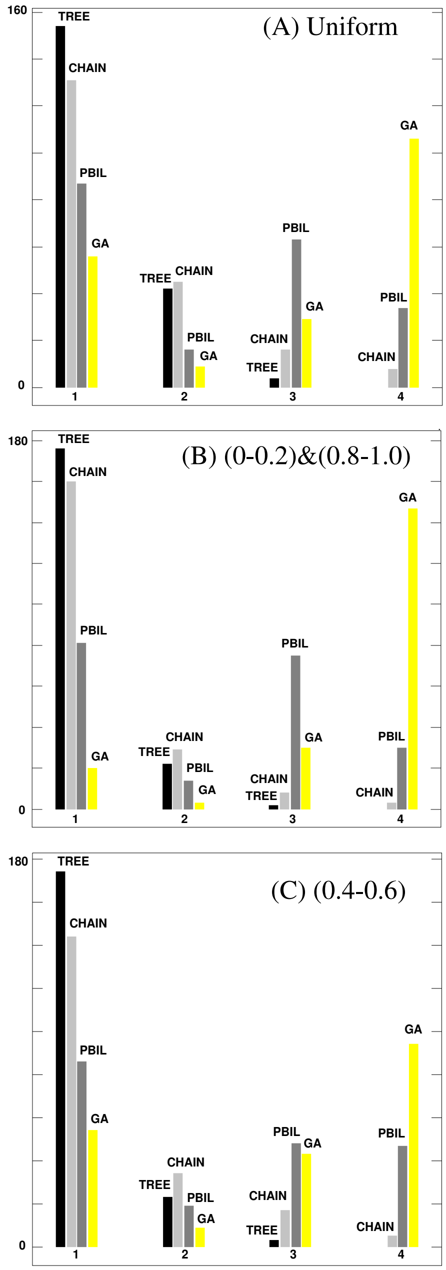

Figure 5: Results for Tree-Max. For each problem, the relative rank of the four algorithms was computed (1=Best, 4=Worst). All members of a tie are assigned the value of the highest rank. A histogram of ranks is presented. For example, in (A) Trees ranked first 154 times, Chains 131 times, GAs ranked first 56 times, and last 106 times. (Note that on many of these problems, there were many ties). (A) Uniform Distribution. (B) Probabilities between 0.0-0.2 and 0.8-1.0. (C) Probabilities between 0.4-0.6.

bits). 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

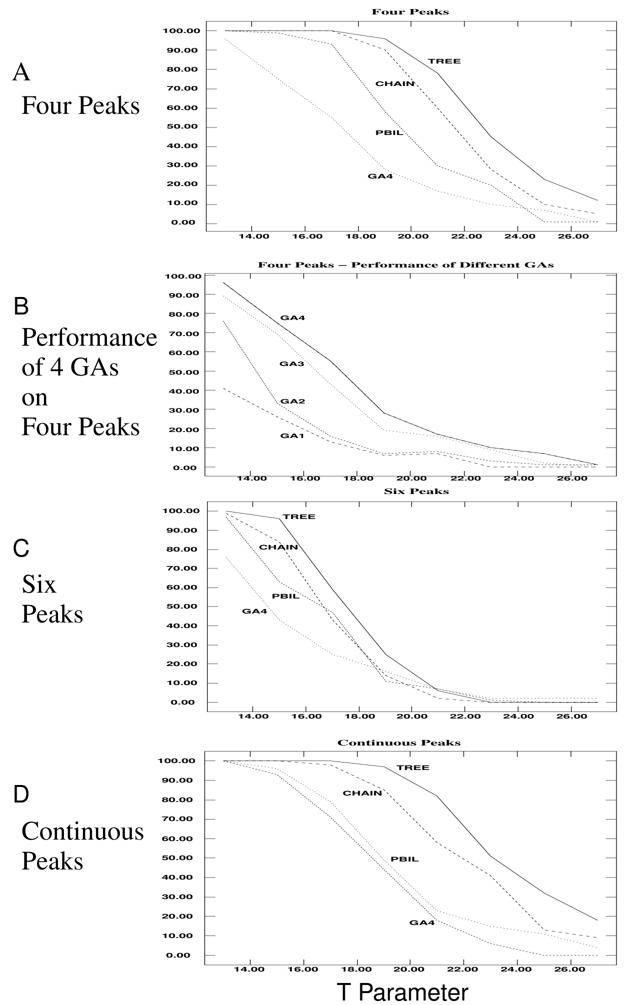

Figure 6: Number of runs (out of 100) that the reward was found (A) Four Peaks, (C) Six Peaks, (D) Continuous Peaks. In (B) the performance of the different GAs is measured on Four Peaks.

bit length is less than the previously studied version), therefore, four GAs were experimented with here. All used *one-point* crossover [Goldberg, 1989] 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.

The results, shown in Figure 6, show that the Tree algorithm outperforms the others in almost all scenarios tested.

3.4 Real-Valued Functions

It is common to use the types of optimization algorithms discussed in this paper on real-valued functions which have been discretized to an arbitrary precision. In this section, three multi-dimensional optimization problems are tested. These problems were chosen because none of the parameters can be set independently. For many real-value optimization problems, especially those which are encoded using Gray-Coding, mutation plays a crucial role in finding good solutions. Therefore, in all of these problems, three GAs are tried (with mutation rates of 0.001, 0.005 and 0.02); results for all of the experiments are given.

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:

$$-0.16 \le s_i \le 0.15 \quad i = 1 \ldots N \qquad y_1 = s_1 \qquad y_i = s_i + y_{i-1} \quad i = 2 \ldots N$$

$$C = \frac{1}{100000} \qquad f = \frac{1.0}{C + \left|\sum_{1}^{N} |y_i|\right|}$$

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 problems runs were simulated. 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. When using algorithms which operate in a binary alphabet, numerical parameters are often encoded in Gray code. Gray code avoids the Hamming cliffs which can be present between consecutive numbers when encoded in standard base-2 representation. We repeated the experiments above (again with 100 trials per dimension setting) using Gray coding. The results are presented in Table II. 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.

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 such that there is guaranteed to be a solution to *DX=B*. The results for the algorithms are presented in Table III; 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 I: Summation Cancellation- Standard Binary Encoding.
Average value of best solution found over 100 runs of 2000 generations each. (Goal: Maximize Value)

Parameters	Bits	Tree	Chain	95% Significance Tree/Chain	PBIL	GA (mut=0.001)	GA2 (mut=0.005)	GA3 (mut=0.02)
10	50	**53.7**	34.1	**Yes**	21.0	13.0	20.2	23.6
12	60	**29.3**	24.1	**Yes**	16.1	9.3	14.5	18.8
15	75	16.8	16.9	No	11.2	5.8	11.3	14.6
17	85	13.8	13.7	No	9.5	4.7	8.9	11.8
20	100	11.0	10.9	No	7.5	3.3	6.8	8.9
23	115	**8.5**	6.4	**Yes**	6.0	2.4	5.7	4.5
25	125	**6.3**	4.2	**Yes**	5.0	2.2	5.1	2.9
27	135	**4.2**	3.0	**Yes**	4.4	1.9	4.4	2.3
30	150	**2.6**	1.9	**Yes**	3.6	1.6	3.7	1.6

Table II: Summation Cancellation- Gray Encoding.
Average value of best solution found over 100 runs of 2000 generations each. (Goal: Maximize Value)

Parameters	Tree	Chain	95% Significance Tree/Chain	PBIL	GA (mut=0.001)	GA2 (mut=0.005)	GA3 (mut =0.02)
10	92008	92008	No	100000	2038.2	98002	100000
12	42053.3	54041.8	No	100000	2020.4	95000	100000
15	4047	7044.1	No	100000	1008.9	81009	100000
17	26.0	**26.7**	**Yes**	98001.5	6.6	70010	100000
20	**14.4**	13.7	**Yes**	94005	5.1	47022	93001.4
23	**8.1**	7.6	**Yes**	90008.7	3.7	24025.1	10011.8
25	5.0	**5.3**	**Yes**	73019.1	2.9	15024.2	6.0
27	**4.4**	4.0	**Yes**	62028.6	2.9	19027.9	3.4
30	**3.0**	2.9	**Yes**	32039.3	2.2	3021.9	2.2

Table III: System of Linear Equations:
Average value of best solution found over 25 runs of 2000 generations each. (Goal: Minimize Error)

Standard Base-2 Encoding								Gray Coding						
Tree	Chain	95%Sig. Tree/ Chain	PBIL	GA (mut .001)	GA2 (mut 0.005)	GA3 (mut 0.02)		Trees	Chain	95%Sig. Tree/ Chain	PBIL	GA (mut .001)	GA2 (mut 0.005)	GA3 (mut 0.02)
648	520	No	2248	3561	1975	3641		335	341	No	1195	1341	865	2136
721	1537	**Yes**	3825	4132	1812	4096		778	799	No	1952	1246	955	2662
405	544	**Yes**	2981	3361	2493	4069		387	346	No	1011	1524	785	1909
706	1347	**Yes**	3204	3554	2313	3921		830	911	No	2120	1628	990	2218
848	1255	**Yes**	3031	3370	2198	3511		**736**	851	**Yes**	1985	1253	834	2265
313	393	**Yes**	2306	3994	2779	3737		330	335	No	621	1205	720	2190
692	1034	**Yes**	3331	3719	2252	3719		809	723	No	1987	1769	1066	2132
708	1029	**Yes**	2996	3100	2076	3841		684	641	No	1753	1311	934	2584
577	904	No	3135	3537	1875	3837		333	308	No	1209	1732	777	2511

Problem 3: Equal Products

In this problem, there are N randomly selected real valued numbers, each in the range (0..5). The object is to select a set of these numbers such that $\prod(selected(X)) = \prod(notSelected(X))$. The evaluation function, which must be minimized, is the absolute difference between the products.

To test the robustness of algorithms as the problem size changes, problems with N= 40,50,60,70,80, and 90 numbers were tried. For each value of N, 16 problems were tried, with 100 runs per problem. Because of the large number of trails, only the chain and tree algorithms were attempted. The chain version did better on the smallest problem, while the tree version did better on the larger problems. In Table IV, the “Tree Better” column shows how many times (out of 16) the Tree algorithm outperformed Chains. The “signif (>95)” column shows the number of problems in which the difference between the chains and trees was significant to 95% (using the Mann-Whitney Test). The same statistics are shown for Chains.

Table IV: Results for Equal Products

N	Tree Better	Signif. (> 95)?	Chain Better	Signif. (> 95)?
40	0/16	n/a	16/16	16/16
50	8/16	0/8	8/16	0/8
60	15/16	5/15	1/16	0/1
70	14/16	6/14	1/16	0/1
80	15/16	1/15	2/16	0/2
90	14/16	4/14	2/16	0/2

3.5 SUMMARY OF RESULTS

In this section, we have presented a large number of empirical results. Five sets of problems were tried, using four algorithms (Trees, Chains, PBIL & GAs). In almost all of the problems, the Tree and Chain algorithms generally outperformed both PBIL and GAs. The exception to this was the “Summation Cancellation” problem utilizing Gray encodings. As can be seen from the results with the varieties of GAs used on the Gray-Code problems, the frequency with which mutation is applied can have a large impact on the performance of the algorithm. The version of the Tree and Chain algorithms used in this study did not have any form of mutation; however, they could easily be extended to include mutation-like operations. It is suspected that this addition will have a large beneficial impact on the quality of results obtained.

Between the Tree and Chain algorithms, the Tree algorithm performed at least as well as the Chain algorithm on almost every problem examined, and often performed significantly better. The notable exception to this is the smallest version of the “Equal Products” problems, in which the Chains performed better. We are currently analyzing what features of this problem cause Chains to perform better.

Perhaps the most important result to notice is that in most cases examined, the performance of the combinatorial optimization algorithms consistently improved as the accuracy of their statistical models increased.

4. SUMMARY & FUTURE DIRECTIONS

We have shown that using incrementally learned conditional probabilities to generate optimal tree-structured probabilistic networks can significantly improve the performance of combinatorial optimization algorithms. Two obvious questions arise: could these results be extended to handle higher-order dependencies, and would modeling such dependencies result in more effective combinatorial optimization algorithms?

Suppose we are given a set of variables X; a database D={C_1, ..., C_m}, where each C_i is an instance of all variables in X; a scoring metric M(D, B) which evaluates how well a network structure B models the dependencies in the data D; and a real value p. The k-LEARN problem may be described as follows: does there exist a network structure B defined over the variables in X, where each node in B has at most k parents, such that M(D, B) ≥ p?

The algorithm we have used in this paper provides a solution for the case in which k=1 and M(D, B) is the likelihood of the data D given the network B. Unfortunately, with the types of scoring metrics that we would probably wish to employ, it has been shown that k-LEARN is NP-complete for k > 1 [Chickering, et al., 1995]. However, search heuristics for finding approximate solutions have been developed for automatically learning Bayesian networks from data [Heckerman, et al., 1995]. A common approach is to perform hill-climbing over network structures, starting with a relatively simple network – often the optimal tree-shaped network produced by algorithms similar to the one presented here.

While it would be computationally expensive, it would be interesting to see whether learning more complex network structures through such hill-climbing procedures might allow combinatorial optimization algorithms similar to PBIL, MIMIC, and the algorithm developed in this paper to reach better solutions with fewer evaluations. It is not clear *a priori* that it is necessarily desirable to use a more complex model of the "good" bit-strings generated so far: the more tightly the model fits the old "good" data, the more heavily the algorithm will concentrate on exploitation rather than exploration. Nonetheless, we feel that experiments with more general probabilistic models would be invaluable.

The types of statistics maintained by our algorithm can be used to combine the outputs from other methods of optimization. For example, multiple hill-climbing or genetic algorithm runs are often used to find different local optima. The best solutions from multiple runs of any of these algorithms, or even from different algorithms, can be used to collect the pairwise statistics used in this study. Once these statistics are collected, and the dependency graph is created, the candidate solutions generated by our algorithm can be used either to update the statistics used by our algorithm, or to re-initialize the other algorithms with good starting points. If the second method is employed, rather than updating the statistics with the strings generated from the previous model, the model could be updated with the strings returned by the separate search procedure. Somewhat similar methods have been explored by [Boyan & Moore, 1997].

Future studies will investigate the feasibility of using these algorithms with larger problems. Here, the largest search space that was explored was 2^{256}. Larger problems, such as those explored in [Baluja, 1997], have search spaces larger than 2^{2000}. It is suspected that modification to the tree algorithm presented here will be required before it can handle problems of this size. For example, creating useful trees with this many dependencies will require larger sample sizes than those used here, to avoid being misled by spurious correlations between bit positions. These studies are currently underway.

Finally, it should be noted that many ideas used in other combinatorial search algorithms can easily be incorporated, such as mutation, weighting the contribution of candidate solutions according to their evaluations, and explicitly recording and using old solutions to model probability distributions. Additionally, our algorithm can easily be extended to handle variables with more than two values. We are also currently extending it to handle real-valued variables. There are many opportunities here for exploiting recent research, such as [Geiger and Heckerman, 1994], on learning Bayesian networks for real-valued functions.

5. ACKNOWLEDGEMENTS

The authors would like to gratefully acknowledge the help of Doug Baker, Justin Boyan, Lonnie Chrisman, Greg Cooper, Geoff Gordon, Andrew Moore, and Andrew Ng.

6. REFERENCES

Baluja, S. (1997) "Genetic Algorithms and Explicit Search Statistics," to appear in *Advances in Neural Information Processing Systems 1996*. Also available as CMU-CS-95-193.

Baluja, S. & Caruana, R. (1995) "Removing the Genetics from the Standard Genetic Algorithm", *International Conference on Machine Learning-12*. Morgan Kaufmann Publishers.

Boyan, Justin. (1993), Personal Communication.

Boyan, J. & Moore, A., (1997), "Using Prediction to Improve Combinatorial Optimization Search", to Appear: AI/Stats, 1997.

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 Transactions on Information Theory*, 14:462-467.

Cooper, G. and Herskovits, E. (1992) "A Bayesian method for the induction of probabilistic networks from data," *Machine Learning*, 9:309-347.

De Bonet, J., Isbell, C., and Viola, P. (1997) "MIMIC: Finding Optima by Estimating Probability Densities," to appear in *Advances in Neural Information Processing Systems 1996.*

Eshelman, L.J., Mathias, K.E, Schaffer, D. (1996), "Convergence Controlled Variation", in *Foundations of Genetic Algorithms-4, 1996*. Morgan Kaufmann Publishers.

Geiger, D. and Heckerman, D. (1994) "Learning Gaussian networks," in *Proceedings of Tenth Conference on Uncertainty in Artificial Intelligence*, 235-243.

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

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

Prim, R.C. (1957) "Shortest Connection Networks and Some Generalizations", *Bell Systems Technical Journal,*36:1389-1401.

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