Combining Multiple Optimization Runs with Optimal Dependency Trees

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

June 30, 1997

CMU-CS-97-157

[1]Justsystem Pittsburgh Research Center
4616 Henry St.,
Pittsburgh, PA. 15213
baluja@cs.cmu.edu

[2]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA. 15213
scottd@cs.cmu.edu

Abstract

When trying to solve a combinatorial optimization problem, often multiple algorithms and/or multiple runs of the same algorithm are used in order to find multiple local minima. The information gained from previous search runs is commonly discarded when selecting initialization points for future runs. We present a method which uses information from previous runs to determine promising starting points for future searches. Our algorithm, termed COMIT, models inter-parameter dependencies present in the previously found high-evaluation solutions. COMIT incrementally learns optimal dependency trees that model the pairwise dependencies in a set of good solutions found in previous searches. COMIT then samples the probability distributions modeled by these trees to generate new starting points for future searches. This algorithm has been successfully applied to jobshop scheduling, traveling salesman, knapsack, rectangle packing, and bin-packing problems.

Scott Davies was supported by a Graduate Student Research Fellowship from the National Science Foundation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation.

Keywords:

Combinatorial Optimization, Dependency Trees, Probability Models, Bayesian Networks, Heuristic Search.

1 Introduction

Either implicitly or explicitly, all black-box combinatorial optimization algorithms concentrate the generation of candidate solutions on regions of the solution space close to the best previously found solutions. For example, hillclimbing and simulated annealing only generate candidate solutions directly neighboring previously discovered solutions. Genetic algorithms use the crossover operator to combine pairs of previously found good solutions into new candidate solutions by placing random subsets of the "parents'" bits into their respective positions in the "child" solution. By concentrating the generation of candidate solutions in this manner, such algorithms often manage to find good solutions without having to explore too much of the search space. The inevitable side effect of this concentration, however, is that these algorithms often converge to solutions which are "good" compared to closely related solutions, but which are far from optimal. Consequently, these algorithms are often restarted from random initialization points and rerun in the hope that they will find better local optima.

After performing several such restarts, however, there is information to be gained by analyzing the various local optima found in multiple search runs and looking for features they have in common. This information can be used to intelligently select new starting points for further searches. For example, if the best locally optimal solutions found so far all have bit 4 set to the opposite value of bit 23, it may make sense to favor such solutions when generating new initialization points for further searches. Such selection mechanisms can help the overall search procedure in two ways: first, by increasing the chance that the local optima found will be better than previously found local optima; and second, by decreasing the number of solution evaluations required before these local optima are attained. One approach to ascertaining this information was explored in [Boyan & Moore, 1997]; they predicted the eventual fitness achieved by future hillclimbing runs based on high-level information about previously found good solutions.

In this paper, we describe the COMIT algorithm (***C***ombining ***O***ptimizers with ***M***utual ***I***nformation ***T***rees). This algorithm uses optimal dependency trees as probabilistic models of the interparameter dependencies exhibited by good solutions found in previous optimization runs. By sampling these probabilistic models, we increase the probability of selecting future starting points that are in basins of the search space which lead to high-evaluation solutions. This method significantly improves the quality of solutions found within a fixed number of solution evaluations on a large set of optimization problems.

Instead of using these probabilistic models to generate starting points for other optimization algorithms, one could imagine inserting good bit-strings generated by the model directly back into the data set on which the model is based, and then immediately updating the model. This approach has been used in previous work [De Bonet, *et. al.*, 1997] [Baluja & Davies, 1997]. The MIMIC algorithm used a heuristic greedy search to generate a chain in which each variable is conditioned on the previous variable [De Bonet, *et. al.*, 1997]. [Baluja & Davies, 1997] extended this work to use a larger class of models – trees, and used a search technique which is guaranteed to find the optimal tree structure. Thus far, both of these approaches have been used for optimizing relatively small problems ($< 2^{256}$); extending these models to large problems is challenging because of (1) computational expense, and (2) the large sizes of data-sets needed to model the search space without prematurely converging the search. Overcoming these problems is currently an avenue of research. In this paper, we explore an alternate use of these models: combining the runs of multiple faster search algorithms. We gain the benefits of modeling the dependencies in the search space at a significantly reduced computational cost. This approach also has the advantage of being able to combine the strengths of several different search algorithms by allowing each algorithm to contribute its good solutions to the probabilistic model. This ability is important since each algorithm can have a different bias which causes it to search different parts of the solution space, thereby allowing it to contribute to the diversity of solutions from which the dependencies are modeled.

In the next section, we describe the COMIT algorithm. In section 3, we provide empirical results demonstrating the effectiveness of the algorithm. For this paper, we restrict our attention to combining the results from multiple hillclimbing runs; however, extending the algorithm to use other underlying combinatorial optimization algorithms is straightforward. In section 4, we close the paper with conclusions and directions for future research.

2 Modeling dependencies in COMIT

Suppose we have a set of good solutions, **S**, found over several previous hillclimbing runs. In this paper, we assume that the solutions are encoded as binary bitstrings; however, higher cardinality alphabets can also easily be used. We wish to discover what interparameter dependencies are exhibited by the bit strings in **S**, and use this information to generate good starting points for future hillclimbing runs. To do this, we try to model a probability distribution $P(\mathbf{X}) = P(X_1, ..., X_n)$ over bit-strings of length n, where $X_1, ..., X_n$ are variables corresponding to the values of the bits. 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.** As in [Baluja & Davies, 1997], we restrict our model $P'(X_1, ..., X_n)$ to the following form:

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

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 that in our models, 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.

From **S**, we calculate empirical probabilities of the form $P(X_i = a)$ and $P(X_i = a, X_j = b)$ for all combinations of i, j, a, and b (a & b are binary assignments to X_i & X_j). From these probabilities we calculate the mutual information, $I(X_i, X_j)$, between all pairs of variables X_i and X_j:

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

We then use a maximum spanning tree algorithm identical to the one used in [Baluja & Davies, 1997] (a slight variation of the algorithm originally proposed in [Chou & Liu, 1968]) to select a set of **n-1** dependencies which maximizes:

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

Among all distributions of the form specified by Equation 1, this distribution P' minimizes the Kullback-Leibler divergence D(P||P′):

$$D(P \parallel P') = \sum_{X} P(X) \log \frac{P(X)}{P'(X)} \tag{4}$$

As shown in [Chow & Liu, 1968] this distribution maximizes the likelihood of the dataset **S** (it optimally models the pairwise dependencies exhibited in **S). ** The tree generation algorithm, summarized in Figure 1, runs in time O(**|S|***n^2), where **|S|** is the size of **S** and **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:
  - 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 ***S*** to estimate the relevant probability distributions.
  - Add X_{add} to the tree, with bestMatchingBitInTree[X_{add}] as its parent.
  - For each bit X_{out} 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}.

Figure 1: Procedure for generating the dependency tree.

Once we have chosen a model P'(X_1...X_n), we use it to stochastically generate K candidate solutions. We evaluate these solutions, and use the best of these solutions as the starting point of a new hillclimbing run. Once this hillclimbing run has terminated, we select a subset of the solutions evaluated during the run to place in the dataset **S.** Up to MAX_INFLUENCE solutions in **S** are replaced by better solutions generated during the hillclimbing run; the size of **S** is kept constant. If MAX_INFLUENCE is too high, then a single hillclimbing can cause **S** to prematurely converge to a set of very similar solutions. If MAX_INFLUENCE is too low, then any hillclimbing run will not have a large enough influence on the history to affect future search. Figure 2 shows how hillclimbing is used as the inner-loop for finding the solutions from which to model the dependencies.

- Initialize the dataset **S** with bitstrings drawn from the uniform distribution
- While termination condition is not met:
  - Create an optimal dependency tree probabilistic model of **S**, as shown in Figure 1.
  - Use this model to stochastically generate K bitstrings. Evaluate these bitstrings.
  - Execute a hillclimbing run starting from the single best bitstring of these K.
  - Replace up to MAX_INFLUENCE bitstrings in **S** with the best bitstrings found during the hillclimbing run just executed.

USER DEFINED CONSTANTS:

|**S**|: Constant size of the dataset **S.**

K: Number of samples to generate from tree before hillclimbing

MAX_INFLUENCE: Max number of elements in **S** to replace after a single hillclimbing run.

Figure 2: The COMIT algorithm: integrating hillclimbing and the dependency tree.

3 Empirical Comparisons

3.1 Algorithm Details

Hillclimbing (HC): To test the ideas presented in this paper, we use next-ascent, stochastic hillclimbing. As is commonly done with heuristic combinatorial optimization, the solutions are encoded as binary vectors. The hillclimbing algorithm used has two interesting properties. First, it allows moves to solutions with higher or equal evaluation; this is extremely important for hillclimbing to work well in many complicated spaces, since this allows HC to explore plateaus. Before restarting, we allow up to PATIENCE evaluations which are worse than the best evaluation seen so far in the run. Evaluations which are equal to the best evaluation seen so far are not counted towards the PATIENCE evaluations (they are, of course, still counted towards the total evaluations). This parameter has a large impact on the effectiveness of hillclimbing in large search spaces. Therefore, for each problem, multiple settings were tried for this parameter. The range of values was (1*|**X**|) to (10*|**X**|), where (|**X**| is the length of the solution encoding). In the results, the best solution found over all of these settings is reported. Second, it is a next-ascent hillclimber; which means that as soon as a better solution is found, it is accepted. This contrasts steepest-ascent hillclimbing, which searches all possible single-bit flips and accepts the one with the largest improvement. Steepest ascent hillclimbing did not work as well on the problems explored here. See Figure 3 for the hillclimbing algorithm shown in detail.

```
while (evaluations < MAX_EVALS)
        V = randomly_generate_binary_vector ();
        Best = Evaluate (V);
        iterations_with_worse_eval =0;
        while (iterations_with_worse_eval < PATIENCE && evaluations < MAX_EVALS)
                evaluations = evaluations + 1;
                bit = random (1..SOLUTION_ENCODING_LENGTH);
                V[bit] = Flip_Bit (V,bit);
                New_Eval = Evaluate (V);

                if (New_Eval < Best)
                        V[bit] = Flip_Bit (V,bit);
                        iterations_with_worse_eval ++;
                else
                        Best = New_Eval;
                        if (New_Eval > Best) iterations_with_worse_eval = 0;
```

USER DEFINED CONSTANTS:

SOLUTION_ENCODING_LENGTH: length of the solution vector; problem specific.

PATIENCE: max number of evaluations that are worse than the best seen so far.

MAX_EVALS: max number of evaluations allowed per run.

Figure 3: Detailed Description of a Hillclimbing Algorithm, set to ***maximize*** the returned evaluation. Details are given for reproducibility.

COMIT: We experiment with two versions of the COMIT algorithm, with K=100 (termed COMIT-100), which samples the tree 100 times before selecting the best point, and with K=1000 (termed COMIT-1000), which samples the tree 1000 times. Note that these extra

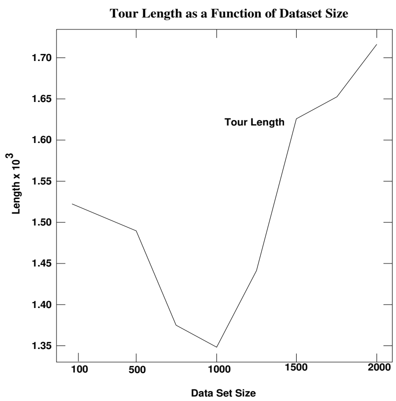

Figure 4: Final solution quality in the TSP domain as a function of the size of the dataset used to learn the dependency trees. For these experiments, MAX_INFLUENCE=100. Experiments with changing MAX_INFLUENCE are underway.

evaluations are counted in the total number of evaluations allowed.

There are two other parameters which need to be set for COMIT. In this study, we set the size of **S** to 1000 and MAX_INFLUENCE to 100. These are kept constant in all of our experiments. Before the first hillclimbing run, **S** is initialized with bitstrings randomly chosen from a uniform distribution.

Given that MAX_INFLUENCE is 100, the size of **S** was chosen by very brief experimentation on the Traveling Salesman Problem (TSP). Results for this are shown in Figure 4. Initially, we see that as the size of **S** increases, the solution quality increases. This is because more solutions are used for modeling the dependencies. However, when the size is increased beyond 1000, the solution quality declines. This is because each run is given a fixed number of evaluations. Since **S** is initialized with random bitstrings, only after these are replaced can **S** be effectively used. With a large |S|, it takes more hillclimbing runs to replace these bitstrings. One way to avoid the latter problem could be to start with a smaller data-set and let it grow as more data becomes available. However, this approach could suffer from narrowing the search too quickly in the initial stages of the algorithm.

Augmented Hillclimbing (AHC): A possible confounding factor in determining how effective COMIT is in comparison to HC, is the fact that COMIT-***K*** examines **K** points before choosing one to use for hillclimbing. To ensure that it is not simply the process of selecting these **K** before hillclimbing that gives performance gains, we augment hillclimbing as follows. Before the beginning of each run, AHC-***K*** examines **K** randomly chosen points. It selects the best one as the starting position for the hillclimber. (The difference between this and COMIT is that COMIT samples K points generated by the dependency tree). Two versions of AHC are examined, AHC-100, and AHC-1000.

3.2 Problem Descriptions

3.2.1. Traveling Salesman Problems (TSP)

The encoding used in this study requires a bit string of size $\mathbf{N}\log_2\mathbf{N}$ bits, where N is the number of cities in the problem. Each city is assigned a substring of length $\log_2\mathbf{N}$ which is interpreted as an integer. The city with the lowest integer value comes first in the tour, the city with the second lowest comes second, etc. In the case of ties, the city whose substring comes first in the bit string comes first in the tour. This encoding was taken from [Syswerda, 1992]. To minimize the tour length, the evaluation used is 1.0/**Tour_Length**.

3.2.2. Jobshop Scheduling Problems

The problem is encoded in two ways. The first encoding is derived from [Fang *et. al*, 1993]. The exact encoding can be found in [Fang *et al*., 1993] and [Baluja, 1994]. The difference between this encoding and that used by Fang is that in this study, bit strings were used to encode the integers (in standard binary encoding) in the range of 1..J. In contrast, Fang used a genetic algorithm and applied crossover to only selected portions of the bitstring. As the makespan is to be minimized, the evaluation of the potential solution is (1.0/ **makespan**). Two standard test problems are attempted, a 10 job, 10 machine problem and a 20-job, 5-machine problem. A description of the problems can be found in [Muth & Thompson, 1963].

The second encoding is very similar to the encoding used in the Traveling Salesman Problem. The drawback of this encoding is that it uses more bits than the previous one. Nonetheless, empirically, it revealed improved results. Each job is assigned M entries of size $\log_2$(J*M) bits. The total length of the encoding is J*M*$\log_2$(J*M). The value of each entry (of length $\log_2$(J*M)) determines the order in which the jobs are scheduled. The job which contains the smallest valued entry is scheduled first, etc. The order in which the machines are selected for each job depends upon the ordering required by the problem specification.

3.2.3. Knapsack Problem

In this version of the knapsack problem, there is a single bin of limited capacity, and **M** elements of varying sizes and values. The problem is to select the elements which will yield the greatest summed value without exceeding the capacity of the bin. The evaluation of the quality of the solution is judged in two ways: If the solution selects too many elements, such that the summed size of the elements is larger than the bin size, the solution is judged by how much it exceeds the capacity of the bin – the less it exceeds the capacity, the better the solution. If the sum of the element sizes is within the capacity of the bin, the sum of the values of the selected elements is used as the evaluation. To ensure that the solutions which overfill the bin are not competitive with those which do not, their evaluations are multiplied by a small constant. This makes the invalid solutions competitive only when there are no solutions which are valid. The evaluations are described below.

$$10^{-10} \times \left(\sum_{allElements} size - \sum_{selectedElements} size \right)$$

if size is greater than capacity of bin

$$\sum_{selectedElements} value$$

if size is less than or equal to capacity of bin

The weights and values for each problem were randomly generated.

3.2.4. Bin Packing/Equal Piles

In this version of the bin packing problem, there are **N** bins of varying capacities and **M** elements of varying sizes. The problem is to pack the bins with elements as tightly as possible, without exceeding the maximum capacity of any bin. In the problems attempted here, the error is measured by:

$$ERROR = \sum_{i=1}^{N} \sqrt{(CAP_i - ASSIGNED_i)^2}$$

CAP_i is the capacity of bin i

$ASSIGNED_i$ is the total size of the elements in bin i.

The solution is encoded in a bit string of length **M** * $\log_2$**N**. Each element to be packed is assigned a sequential substring of length $\log_2$**N** whose value indicates the bin in which the element is placed. In order to minimize the **ERROR**, the evaluation of the potential solution is 1.0/**ERROR**.

3.2.5. Rectangle Packing

The object of this problem is to pack **N** rectangles as tight as possible on a page, thereby leaving as much space at the bottom of the page as possible. The problem is encoded as $N*\log_2 N$ bits. Each rectangle is assigned a unique $\log_2 N$ bits, which are interpreted as a integer. The number dictates the order in which the rectangles are placed on the page. As a rectangle is placed, it is placed as high and to the left on the page as possible. Therefore, changing the order in which the rectangles are placed on the page will change the final configuration of the rectangles. The evaluation function tries to first maximize the number of rectangles which are on the page (the rectangles were chosen so that they all can fit on the page), and after this is accomplished to maximize the amount of room left at the bottom of the page.

3.2.6. 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:

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

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

For this problem, we used 75 parameters, and each parameter was represented with 9 bits, encoded in standard base-2, with the values uniformly spaced between -2.56 and +2.56.

3.3 Results

For each of the problems, we try the five algorithms mentioned above. For each algorithm on each problem, we try multiple settings of the PATIENCE parameter. The setting of the PATIENCE parameter which gives the best result is report here. This is done to give HC and AHC an advantage; in almost every case, the lowest setting of the PATIENCE parameter worked the best with COMIT. The results reported are the average of at least 25 runs. Each algorithm is given 200,000 function evaluations on each problem. The results are shown in Table I. A summary of the results, the relative ranks of the algorithms, are provided in Table II (1=best, 5=worst). Additionally, the significance in the difference in results is given; this is measured by the Mann-Whitney test (a non-parametric equivalent to the *t*-test).

Table I: Performance of HC, AHC and COMIT

Problem	|**X**| Problem Size: Bits	HC Mean (Std. Dev)	AHC-100 Mean (Std. Dev)	AHC-1000 Mean (Std. Dev)	COMIT-100 Mean (Std. Dev)	COMIT-1000 Mean (Std. Dev)
TSP - 100 City (Minimization)	700	1629 (106)	1599 (103)	1573 (87)	**1335** **(87)**	1336 (117)
Jobshop - Problem 1, Encoding 1 (Minimization)	500	998 (20)	988 (16)	982 (16)	978 (14)	**970** **(12)**
Jobshop - Problem 1, Encoding 2 (Minimization)	700	965 (8)	961 (11)	957 (9)	954 (9)	**953** **(6)**
Jobshop - Problem 2, Encoding 2 (Minimization)	700	1207 (13)	1200 (11)	1199 (11)	1196 (12)	**1195** **(8)**
Bin-Packing 168 elements, 8 bins (Minimization)	504	1.70e-03 (3.2e-04)	1.58e-03 (4.1e-04)	1.62e-03 (3.0e-04)	1.56e-03 (4.7e-04)	**1.45e-03** **(4.2e-04)**
Knapsack - 512 elements (Maximization)	512	3238 (135)	3377 (145)	3335 (99)	**6684** **(143)**	6259 (167)
Rectangle Packing(75 rectangles) (1.0/Evaluation) (Minimization)	525	8.32e+06 (1.4e+05)	8.23e+06 (1.6e+05)	8.25e+06 (1.6e+05)	8.24e+06 (1.3e+05)	**8.21e+06** **(1.7e+05)**
Sum. Canc. (75 params * 9 bits) (Minimization)	675	64 (3)	61 (3)	59 (3)	54 (4)	**52** **(4)**

Table II: Relative Ranks of HC, AHC and COMIT (Algorithm with Rank=1 highlighted)

Problem	|**X**| Problem Size: Bits	HC	AHC-100	AHC-1000	COMIT-100	Significance > 90%? (COMIT-100 vs.:)			COMIT-1000	Significance > 90%? (COMIT-1000 vs.:)		
						HC	AHC-100	AHC-1000		HC	AHC-100	AHC-1000
TSP - 100 City	700	5	4	3	**1**	Y	Y	Y	2	Y	Y	Y
Jobshop - Problem 1, Encoding 1	500	5	4	3	2	Y	Y	N	**1**	Y	Y	Y
Jobshop - Problem 1, Encoding 2	700	5	4	3	2	Y	Y	Y	**1**	Y	Y	Y
Jobshop - Problem 2, Encoding 2	700	5	4	3	2	Y	N	N	**1**	Y	N	N
Bin-Packing 168 elements, 8 bins	504	5	3	4	2	Y	N	N	**1**	Y	Y	Y
Knapsack - 512 elements	512	5	3	4	**1**	Y	Y	Y	2	Y	Y	Y
Rectangle Packing(75 rectangles)	525	5	2	4	3	Y	N	N	**1**	Y	N	N
Sum. Canc. (75 params * 9 bits)	675	5	4	3	2	Y	Y	Y	**1**	Y	Y	Y

In every problem examined, COMIT significantly improves the performance over hill-climbing. In comparison to AHC and HC, COMIT-1000 or COMIT-100 performed the best in every problem. To provide some intuition about how the COMIT algorithm progresses, Figure 5 shows the values of each evaluation performed with the HC and COMIT-1000 algorithm in the TSP domain. There are four features which should be noticed. First, the spikes in the evaluations correspond to the beginning of hillclimbing runs. In the COMIT graph, the spikes also represent the K samples generated by sampling the tree. Second, for the COMIT algorithm, the random initial samples in the dataset **S** were entirely removed by evaluation #90,000 (this approximately corresponds to the number of evaluations used in the first 10 hillclimbing runs; each run contributed 100 samples to the dataset, and the size of **S** is 1000). Third, the magnitude of the spikes in the COMIT plot gradually decreases; this corresponds to the COMIT algorithm learning to seed the hillclimbing runs with high-quality solutions. Fourth, most importantly, even before the HC runs are *started* at noticeably better solutions, the *final* solutions found at each hillclimbing run have improved over standard hillclimbing; good initial points for search have been found. By using the interparameter dependency models to generate starting points, the hillclimbing runs are started are in basins of the search space which lead to high-evaluation solutions.

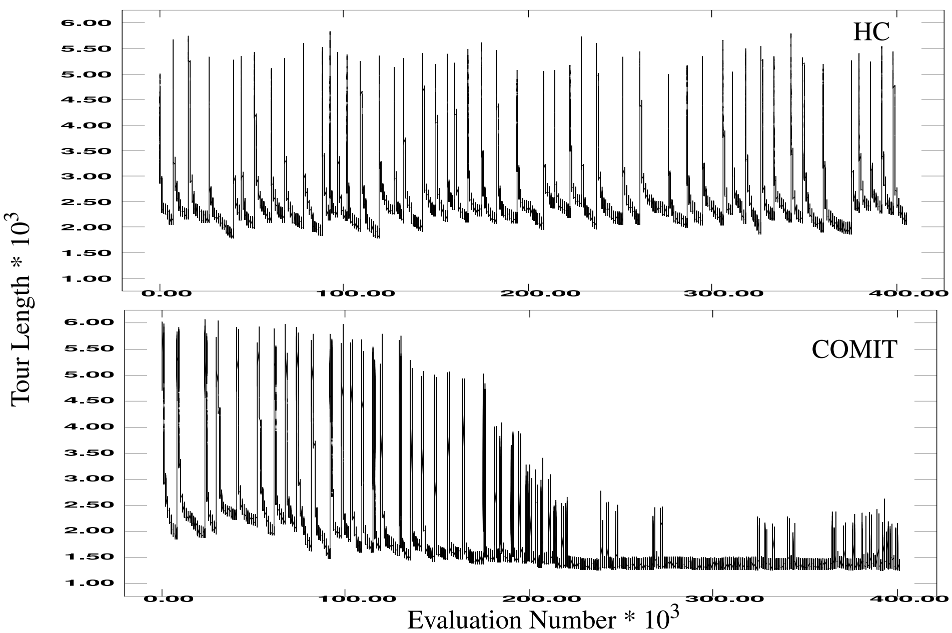

Figure 5: These graphs show the values of every evaluation performed in the HC (top) and COMIT (bottom) algorithms for the TSP domain. The object is to minimize the tour length. Note that these runs are extended to 400,000 evaluations.

4 Conclusions & Future Work

We have shown that information gathered from previous search runs can be effectively used for initializing new searches. By using a model of the interparameter dependencies in previously found good solutions, we generated starting points which allowed the search algorithm to concentrate its effort in promising regions of the search space. In all of the problems examined, this has led to the discovery of significantly better final solutions.

Since the probabilistic model is updated relatively infrequently by COMIT – that is, only between hillclimbing runs – it may be feasible to replace the dependency trees used here with more sophisticated but more computationally expensive models, such as general Bayesian networks. These can model arbitrary sets of dependencies. Rather than generating the network from scratch after every hillclimbing run, we can use the previous network as the starting point for a search for network structures with which to model the updated dataset **S.** If only a few modifications to **S** were made from the previous hillclimbing run, then searching for a new Bayesian network may be relatively inexpensive.

The purpose of this paper was not to advocate the use of COMIT with hillclimbing over more sophisticated optimization algorithms. The purpose was, instead, to show that when multiple runs of an optimizer are used, the information obtained from one run can be used to guide future searches. We showed how to incorporate COMIT with hillclimbing. However, it requires no change to be used with other search algorithms such as simulated annealing, genetic algorithms, hillclimbing, TABU search [Glover, 1989], or PBIL [Baluja, 1997]. In population-based methods, such as genetic algorithms, it can be used to initialize the population. Additionally, this method can be used when multiple search algorithms are used; the best solutions from any search algorithm can be used to update the pair-wise statistics. The trees generated can then be sampled to provide initial samples to begin any of these searches.

References

Baluja, S. (1997) "Genetic Algorithms and Explicit Search Statistics," *Advances in Neural Information Processing System*s, 1996. Mozer, M.C., Jordan, M.I, & Petsche, T. (Eds). MIT Press. Also Available as Technical Report from Carnegie Mellon University: CMU-CS-95-193 (http://www.cs.cmu.edu/~baluja).

Baluja, S. & Davies, S. (1997) "Using Optimal Dependency-Trees for Combinatorial Optimization: Learning the Structure of the Search Space", *Proc. 1997 International Conference on Machine Learning*. Also Available as Tech Report: CMU-CS-97-107.

Boyan, J. & Moore, A. (1997) "Using Prediction to Improve Global Optimization Search", Submitted.

Chou. C. and Liu, C. (1968) Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information 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).

Fang, H.L., Ross, P. & Corne, D. "A Promising GA Approach to Job-Shop Scheduling, Rescheduling and Open-Shop Scheduling Problems". In *Proc. Int. Conf. on Genetic Algorithms-95*. S. Forrest, (ed). Morgan Kaufmann.

Glover, F. (1989) "Tabu-Search - Part I", *ORSA Journal on Computing* 1:190-206.

Muth & Thompson (1963) *Industrial Scheduling* Prentice Hall International. Englewood Cliffs, NJ.

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