Optimal Reinsertion: A new search operator for accelerated and more accurate Bayesian network structure learning

Andrew Moore AWM@CS.CMU.EDU
Weng-Keen Wong WKW@CS.CMU.EDU
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 USA

Abstract

We show how a conceptually simple search operator called Optimal Reinsertion can be applied to learning Bayesian Network structure from data. On each step we pick a node called the target. We delete all arcs entering or exiting the target. We then find, subject to some constraints, the globally optimal combination of in-arcs and out-arcs with which to reinsert it. The heart of the paper is a new algorithm called ORSearch which allows each optimal reinsertion step to be computed efficiently on large datasets. Our empirical results compare Optimal Reinsertion against a highly tuned implementation of multi-restart hill climbing. The results typically show one to two orders of magnitude speed-up on a variety of datasets. They usually show better final results, both in terms of BDEU score and in modeling of future data drawn from the same distribution.

1. Bayesian Network Structure Search

Given a dataset of R records and m categorical attributes, how can we find a Bayesian network structure that provides a good model of the data? Happily, the formulation of this question into a well-defined optimization problem is now fairly well understood (Heckerman et al., 1995; Cooper & Herskovits, 1992). However, finding the optimal solution is an NP-complete problem (Chickering, 1996a). The computational issues in performing heuristic search in this space are also severe, even taking into account the numerous ingenious and effective innovations in recent years (e.g. (Chickering, 1996b; Friedman & Goldszmidt, 1997; Xiang et al., 1997; Friedman et al., 1999; Elidan et al., 2002; Hulten & Domingos, 2002)), discussed in Section 4.

Problem: *From fully observed categorical data find an acyclic structure and tabular conditional probability tables (CPTs) that optimize a Bayesian Network scoring criterion. Assume no initial knowledge of the node ordering.*

The operation we need to perform is

$$\text{argmax}_D \; \textit{DagScore}(D) \tag{1}$$

where D is a directed acyclic graph (DAG) and *DagScore* is a complexity-penalized measure of how well the DAG explains the data. Common versions of *DagScore* can be broken into a sum of terms: one for each node in the DAG.

$$\textit{DagScore}(D) = \sum_{i=1}^{m} \textit{NodeScore}(\textit{Parents}(i) \to i) \tag{2}$$

where $\textit{Parents}(i)$ is the set of parents of attribute i in the DAG, and $\textit{NodeScore}(\textit{Parents}(i) \to i)$ scores the degree that these parents predict the conditional distribution of i given the parents (while penalizing for model complexity). Examples of *NodeScore* function that have been proposed are BIC (Schwartz, 1979), BD (Cooper & Herskovits, 1992), BDE (Heckerman et al., 1995) and BDEU (Buntine, 1991). The algorithms of this paper can be applied irrespective of the choice of *NodeScore* function.

1.1. Performing the optimization

The most common algorithm for optimizing $\textit{DagScore}(D)$ is a hill climbing algorithm which begins with an initial structure and then considers a finite set of tweaks to it. The tweaks usually include some or all of "Add arc", "Delete arc" and "Reverse arc", subject to never attempting a move that would introduce a cycle. Hill climbing proceeds by finding the best tweak and applying it. Then it finds the best tweak for the new DAG structure. This process continues until no more tweaks improve the structure. In practice it is well worth attempting to jump out of local minima. This is usually done by a combination of multiple restart hill climbing, simulated annealing and TABU search.

1.2. Optimal Reinsertion Overview

In this paper we introduce a new, much larger-scale, search operator called *Optimal Reinsertion* and (more importantly) show how to compute it efficiently.

Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC, 2003.

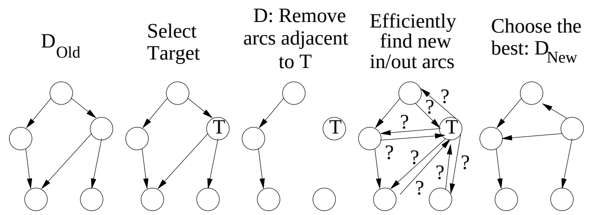


Figure 1. Optimal Reinsertion

Given a start structure D_{old} (Figure 1), pick one node (called the *Target*, T) and sever all connections in and out. Then find, subject to some constraints, the optimal set of in-arcs and out-arcs with which to reinsert it. This procedure continues, running through repeated cycles in which all nodes take turns at being the target, until no step changes the DAG structure. Each move in this search space finds the best of (typically) billions of changes to the current DAG, and so we can hope for faster and less local-optimum-prone search. With the algorithms described in subsequent sections each step finds the optimal reinsertion very quickly.

2. Algorithms for Optimal Reinsertion

2.1. The *maxParams* Parameter

During each Optimal Reinsertion step we forbid certain structures. We do not allow any of the conditional probability tables to contain more than *maxParams* non-zero entries. We make this restriction for its computational benefit: we can control the total number of CPTs we will ever need to look at. We can hope that the effects of this approximation will be mitigated by the fact that large CPTs will receive large penalties in the scoring function and so would not be part of the optimal solution in any case. We will resort to empirical studies to see the actual consequences. In practice we typically set *maxParams* between 10 and 100, and usually a target node has somewhere between thousands up to millions of possible parent sets.

2.2. Cached Node Scores

In the algorithm that follows, it will be necessary to quickly look up the $NodeScore(PS \to T)$ value for a wide variety of Parent-Sets PS and all possible target nodes T. Let us use the convention that given a dataset with m attributes, the attributes (and hence the corresponding nodes in the DAG) are identified by integers $1, 2, \ldots m$ respectively. A parent-set PS is thus a subset of the integers 1 through m. Similarly, a target node T is an integer between 1 and m.

We create a cache of $NodeScore(PS \to T)$ values, so that once the cache is constructed any lookup may occur in time independent of R (the number of records) or m (the total number of attributes). We only cache $NodeScore(PS \to T)$ combinations that produce conditional probability tables with *maxParams* or fewer parameters.

Creating this cache looks expensive. With $maxParams = 2^k$ and m binary-valued attributes, there are $\binom{m-1}{k}$ CPTs for each of the m target nodes, meaning $m\binom{m-1}{k}$ tables in total, each needing $O(R)$ work to construct naively.

Constructing all these tables is a job suited for AD-search, introduced in (Moore & Schneider, 2002) and an extension of AD-trees (Moore & Lee, 1998). There is no space to review AD-search here, except to mention costs. Searching all contingency tables of dimension k would normally require $R\binom{m}{k}$ operations. In contrast, AD-search requires

$$R \sum_{j=0}^{k} \lambda^j \binom{m}{j} \tag{3}$$

operations, where λ is a dataset-specific quantity that is always in the range $[0, 0.5]$, and is smaller for datasets with larger degrees of inter-dependence between attributes. Empirically, λ is usually between 10^{-2} and 10^{-1}.

2.3. Searching for the Optimal Reinsertion

Let D_{old} = DAG before the Optimal Reinsertion of T.
Let $D = D_{\text{old}}$ with all arcs into and out of T removed.
Let $D_{\text{new}} = D$ after Optimal Reinsertion (see Figure 1).

2.3.1. Legal Supplements of a Parent-Set.

We search over parent-sets of T. During search, if our current parent-set is PS, then the next sets to be inspected will be defined as $LegalSupplements(PS)$ where $Supplements(PS) = \{PS \cup \{q+1\}, PS \cup \{q+2\}, \ldots PS \cup \{m\}\}$ with $q = \max(PS)$ (e.g. if $PS = \{2,4\}, \max(PS) = 4$) and $LegalSupplements(PS)$ defined as those members of $Supplements(PS)$ that produce a CPT with *maxParams* or fewer parameters. Define $q = 0$ if $PS = \{\}$.

2.3.2. All Specializations of a Parent Set.

One final definition concerns the complete set of legal specializations of PS. This set of legal specializations, called $Specializations(PS)$ are those parent sets that are supplements, supplements of supplements, or supplements to the nth degree of PS. Formally, $Specializations(PS)$ is the closure of $LegalSupplements(PS)$: $PS' \in Specializations(PS)$ if and only if $PS' = PS$ or $PS' \in Specializations(PS'')$ for some $PS'' \in LegalSupplements(PS)$.

Assume that $\{3,5\}, \{3,5,6\}$ and $\{3,4,5\}$ all produce fewer than *maxParams* parameters. Note that $\{3,5,6\} \in Specializations(\{3,5\})$ but $\{3,4,5\} \notin Specializations(\{3,5\})$ (because supplements of a PS involve only nodes denoted by a higher index than is currently in PS). A further example is given in part of Figure 2.

Note that depth first traversal through the space of all parent sets, beginning at $\{\}$ and using *LegalSupplements* as the search operator, will visit each legal parent set exactly once.

2.3.3. THE ORSEARCH ALGORITHM

The search algorithm could be of this form:

> *For all possible (PS,CS) pairs, consider a version of D in which T is given PS as its parents and CS as its children...*

In fact, Section 2.1 added the restriction

> *...disallowing (PS,CS) pairs in which any CPT has more than maxParams non-zero parameters.*

Even with this restriction there are many (PS,CS) pairs to consider, sometimes trillions. Happily, we can make the search tractable with two steps. First, by analytically computing the optimal CS to associate with each PS. Second, by pruning the search tree over parent sets in cases where we can prove no specialization can be better.

2.3.4. CHOOSING CHILD SETS

In Figure 2 we are considering parent-set $\{2,4\}$. Without introducing cycles in the presence of this parent set, the potential children of T are $SafeNodes(PS) = \{3,6,7,8\}$. In general define that $i \in SafeNodes(PS)$ if and only if (a) there is no directed path from i to T in D', and (b) adding T to the parents of i would not cause more than *maxParams* parameters for node i.

Each node in $SafeNodes(PS)$ can independently consider whether there is benefit in adding a $T \to i$ link. We can thus define the optimal Child Set to associate with Parent Set PS as $\{i \in SafeNodes(ps) : NodeScore(P_i \cup \{T\} \to i) > NodeScore(P_i \to i)\}$ where P_i are the original parents of i in DAG D.

2.3.5. BRANCH-AND-BOUND OVER PARENT SETS

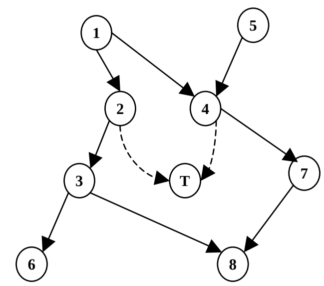
ORSearch takes four parameters:

- D, the current DAG (in which T has no in-arcs and no out-arcs).
- T, the current target node.
- PS_{in}, a parent set.
- PS_{known}, another parent set: the best set found so far in the search.

Define $ORScore(PS)$ to be the score obtained with PS as the parents of T and $OptChildren(PS)$ as T's children. The **ORSearch** procedure outputs PS_{out} defined as:

$$\underset{PS \in \{PS_{\text{known}} \cup Specializations(PS_{\text{in}})\}}{\text{argmax}} ORScore(PS)$$

Table 1 gives the implementation of **ORSearch**. In Step 3, we compute $ORScore(PS)$. In Step 4, we compute the best possible score of any of the legal specializations of PS. This requires us to obtain $bestNodeScore(PS \to T)$, defined as:



$PS = \{2,4\}$

$LegalSupplements(PS) = \{\{245\}\{246\}\{247\}\{248\}\}$

$Specializations(PS) = \{\{24\}\{245\}\{246\}\{2456\}$
$\{247\}\{2457\}\{2467\}\{24567\}\{248\}\{2458\}$
$\{2468\}\{24568\}\{2478\}\{24578\}\{24678\}\{245678\}$

$SafeNodes = \{3,6,7,8\}\}$

Figure 2. An illustration of *LegalSupplements, Specializations* and *SafeNodes* for the illustrated parameter set. We assume that *maxParams* is sufficiently large that no parent-sets are discounted for producing too many parameters.

$$\max_{PS' \in Specializations(PS)} NodeScore(PS' \to T) \tag{5}$$

This information is computed once, with one pass over the cache immediately after the cache of *NodeScore*s is created, so is a constant-time lookup during execution of **ORSearch**.

Step 5 is our opportunity to abort this portion of the recursive search. If the condition is satisfied we have proved that there is no point in looking at any specialization of *PS*. Note how the decision to bound the search is dependent on the structure of the rest of D: depending on the structure we may or may not abort because *PS* may or may not cause loops that lose us the benefit of some of our children.

Step 7 is the recursive call. We look at the immediate legal supplements of the current parent-set, each time possibly improving PS^*. Notice that by passing PS^* (the best parent set so far) as the fourth argument we give as much opportunity as possible for recursive calls to prune.

At all points in the algorithm the *NodeScore* value, *NodeScore* values, *ChildBenefit* values, *DagScore* values and *bestNodeScore* values can all be obtained in constant time from the cached *NodeScore* tables, or by recalling values computed earlier on in the search. The original dataset is neither needed nor accessed.

We omit the elementary inductive proof of the correctness of the procedure. **ORSearch** is initially called with parameters: **ORSearch**$(D,T,\{\},\{\})$. It thus returns the best

ORSearch$(D, T, PS_{\text{in}}, PS_{\text{known}})$

1. Let $D' = D$ supplemented with $k \to T$ for all $k \in PS_{\text{in}}$ and $T \to i$ for all $i \in \mathit{OptChildren}(PS_{\text{in}})$.
2. Let *ChildBenefit* be the change in the score of D that is due to the positive benefit of the children derived from PS_{in}. $\mathit{ChildBenefit} = \sum_{i \in \mathit{OptChildren}(PS_{\text{in}})} \mathit{NodeScore}(P_i \cup \{T\} \to i) - \mathit{NodeScore}(P_i \to i)$
   where P_i are the current parents of node i in DAG D.
3. Let
   $$\mathit{myScore} = \mathit{DagScore}(D') = \mathit{BaseScore} + \mathit{NodeScore}(PS_{\text{in}} \to T) + \mathit{ChildBenefit} \tag{4}$$
   where *BaseScore* is the (static) sum of node scores for the rest of the net, thus $\mathit{BaseScore} = \sum_{j \neq T} \mathit{NodeScore}(P_j \to j)$. The middle term in Equation 4 adds in the contribution from T's Parent Set. The rightmost term adjusts for the effect upon new children of T.
4. Let $\mathit{bestDagScore} = \mathit{BaseScore} + \mathit{bestNodeScore}(PS_{\text{in}} \to T) + \mathit{ChildBenefit}$
5. If $\mathit{bestDagScore} \leq \mathit{ORScore}(PS_{\text{known}})$ define $PS_{\text{out}} = PS_{\text{known}}$ and return from this recursive call.
6. Let PS^* be a local variable denoting the best Parent Set encountered so far within this stack frame.
   If $\mathit{myScore} > \mathit{ORScore}(PS_{\text{known}})$ define $PS^* = PS_{\text{in}}$ else define $PS^* = PS_{\text{known}}$.
7. For each $PS' \in \mathit{LegalSupplements}(PS)$: PS^* := **ORSearch**(D, T, PS', PS^*)
8. $PS_{\text{out}} = PS^*$ (the best scoring out of PS_{known}, PS_{in} and the best of the results of the recursive calls).

Table 1. The **ORSearch** algorithm, described in Section 2.3.

available legal *PS*, which we then assign as the new parents of T. The new children of T are those legal, non-cycle-inducing nodes with strictly positive benefit.

2.4. The outer loop

We have defined a single Optimal Reinsertion operation, but the full search consists of repeated operations. One full pass of the outer loop consists of generating a random ordering of $\{1, 2, \ldots m\}$ then applying the operation to each node in this ordering in turn. We iterate this procedure, performing a full pass over and over again, with a newly randomized ordering on each pass. When a full pass causes no change to the current DAG, we terminate.

2.5. Multiple Restarts

In case the above procedure gets caught in a local minimum, we run it a large number of times—50 in the following experiments. Each run begins with a randomly corrupted version of the best DAG to date. Empirically, the use of multiple restarts does not appear to be critical: after the first run we have never observed a significant improvement from the remaining runs.

2.6. Final Stage of Hill climbing

After the specified number of restarts of the Optimal Reinsertion procedure we finally perform an iteration of conventional hill-climbing. This is in order to allow the current DAG to tweak itself to introduce CPTs bigger than allowed by the *maxParams* parameter. Thus, for this final iteration, we no longer apply the restriction that all contingency tables must have fewer than *maxParams.* In some cases, if *maxParams* was set to a low value, this final pass can significantly improve the score.

2.7. Sparse Candidate Optimal Reinsertion

The sparse candidate algorithm was introduced in (Friedman et al., 1999). It approximates standard hill climbing, but maintains a small set of candidate parents for each node instead of simply allowing any parent for the node. Initially, the candidate set for node i is chosen to be the k attributes best correlated with i, but the set can be refined as the search progresses. This has the strong benefit of reducing the number of *NodeScore* computations needed during hill climbing. Indeed, the first pass of hill climbing requires only $O(mk)$ such computations instead of $O(m^2)$.

We have incorporated a primitive version of Sparse Candidate into the Optimal Reinsertion search. For datasets with large numbers of attributes, we restrict the set of legal parents of T to be those k attributes most strongly correlated with T. In the experimental results section we use values of k ranging between 10 to 20, depending on the number of attributes in the dataset.

3. Empirical Results

Table 2 shows the datasets that were used.

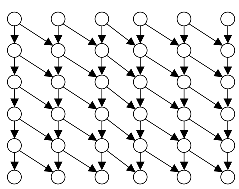
3.1. Generation of synthetic datasets

The synthetic datasets create local-minimum-prone search landscapes. They were generated from the structures shown in Figure 3. All nodes were binary. Nodes without parents chose their value with 50-50 probability. For nodes with parents,

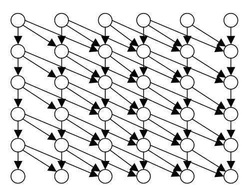
$$P(value = True|\text{parents}) = 0.1 \text{ if Parity(parents)=0}$$
$$P(value = True|\text{parents}) = 0.9 \text{ if Parity(parents)=1}$$

where Parity(Parents)=1 if and only if an odd number of parents have value "True". The nodes are thus noisy exclusive-ors and so it is hard to learn a set of parents incrementally.

Synth2



Synth3



Synth4

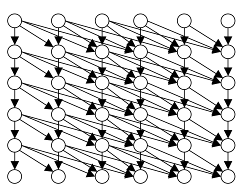


Figure 3. Synthetic datasets described in Section 3.1.

	R	m	AA	
adult	49K	15	7.7	Contributed to UCI by Ron Kohavi
alarm	20K	37	2.8	Data generated from a standard Bayes Net benchmark (Beinlich et al., 1989).
biosurv	150K	24	3.5	Anonymized, deidentified aggregate information about hospital admission rates
covtype	150K	39	2.8	Contributed to UCI by Jock Blackard.
con4	67K	43	3.0	Contributed to UCI by John Tromp
edsgc	300K	24	2.0	Data on 300,000 galaxies from the Edinburgh-Durham Sky Survey (Nichol et al., 2000)
synth2	25K	36	2.0	Generated from Figure 3
synth3	25K	36	2.0	Generated from Figure 3
synth4	25K	36	2.0	Generated from Figure 3
nursery	13K	9	3.6	Contributed to UCI by Marko Bohanec and Blaz Zupan
letters	20K	17	3.4	Contributed to UCI by David Slate

Table 2: Datasets used. R = Number of records, m = number of attributes and AA = average arity of attributes.

3.2. Processing of the real datasets

The empirical datasets were chosen from UCI Irvine datasets (Blake & Merz, 1998) that contained at least 10,000 records. Real valued attributes were automatically converted to binary-valued categorical attributes by thresholding them at their median value (treating real-valued variables in this way with Bayesian Nets is not generally a good idea but it does not favor either method in this evaluation). Several additional datasets of particular current interest within our research lab were also used.

To our knowledge, this is a relatively large study of a Bayesian Network structure finding algorithm—running on 11 datasets and analyzing both optimization performance and k-fold test set performance. The datasets we used, including our discretizations, are at *http://www.cs.cmu.edu/∼awm/optreinsert*.

3.3. The Hill climbing implementation

Following the methodologies of (Elidan et al., 2002; Friedman & Goldszmidt, 1997; Friedman et al., 1999; Hulten & Domingos, 2002), we benchmarked Optimal Reinsertion against an optimized version of traditional hill climbing. We used the same libraries and underlying efficient data structures to implement hill climbing. We tuned the multi-restart strategy to maximize performance and we are satisfied that our hill climber is highly optimized. For example, an efficient hash table is used to ensure that hill climbing never redundantly recomputes a *NodeScore* score from the dataset that it can obtain from the results of an earlier computation.

3.4. Optimizing the BDEU score: Experiments

All experiments were performed on an unloaded 2 gigahertz Pentium 4, with 2 gigabytes of RAM (although none of the experiments below used more than 1 gigabyte and most used less than 100 megabytes). The time measured for Optimal Reinsertion includes the costs of all components of the algorithm including the AD-search and building of the *NodeScore* cache.

Table 4 shows the performance of Optimal Reinsertion search against hill climbing. Table 3 shows the number of structures searched by Optimal Reinsertion for each dataset. In most (but not all) cases Optimal Reinsertion quickly finds a better solution than Hill-climbing ever finds.

3.5. Is there an acceleration?

Table 4 compares the performance of Optimal Reinsertion when given only one thirtieth the wall-clock time of hill climbing. It confirms that Optimal Reinsertion usually performs at least as well as hill climbing when Optimal Reinsertion is given only 100 seconds and hill climbing is given 3000 seconds.

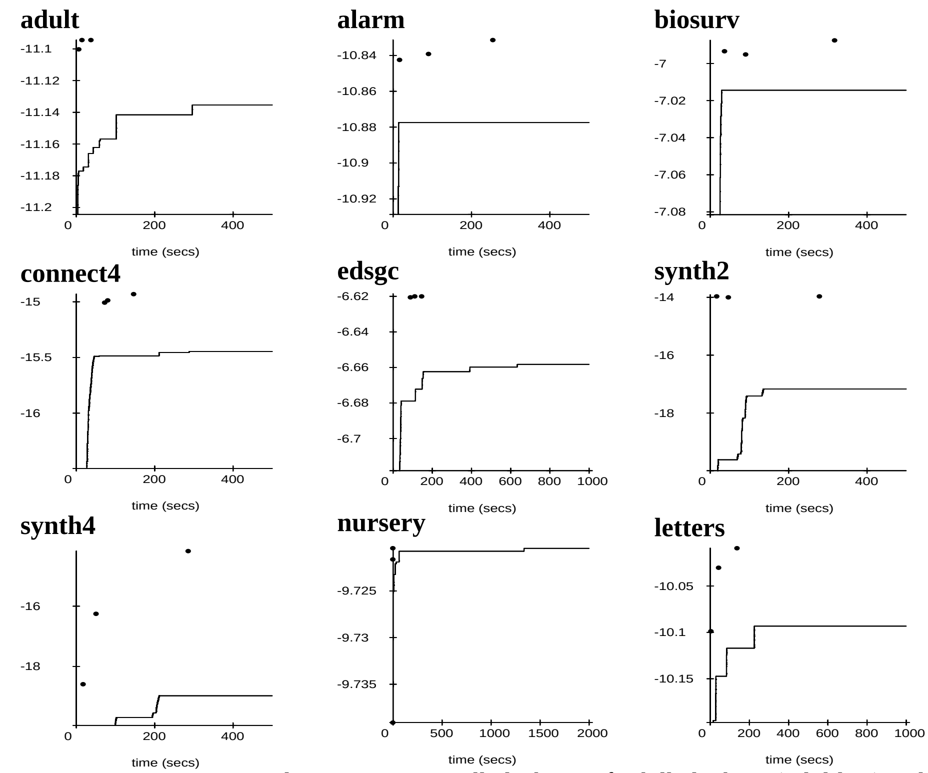


Figure 4. BDEU score per datapoint versus wall-clock time for hill-climbing (solid line) and three versions of Optimal Reinsertion search (shown as dots, corresponding left-to-right with *maxParams* $= 10$, 50 and 100). This version of Optimal Reinsertion is not an anytime algorithm which is why three dots are shown instead of three additional curves. Hill climbing was run for 3000 seconds on all datasets, but in all cases there is no non-negligible change in BDEU score for Hill climbing after the time period shown on the graphs. The timing for Optimal Reinsertion includes all the computation including the preprocessing, and the AD-search to generate the cached *NodeScore*s. A minimal vertical axis scale of 0.1 (corresponding to an average record probability change of about 10%) was used in all cases except where the difference between AD-search and Hill climbing was too large for such a scale.

Data Set	Num. Structures
adult	4.6×10^7
alarm	1.5×10^{15}
biosurv	4.4×10^{10}
connect4	7.6×10^{16}
covtype	2.0×10^{15}
edsgc	5.2×10^{10}
synth2	4.1×10^{14}
synth3	4.4×10^{14}
synth4	3.9×10^{14}
letters	1.7×10^8
nursery	2.5×10^5

Table 3. The number of DAG structures maximized over during the full Optimal Reinsertion search. There are duplicate structures in this search but we nevertheless know that billions of unique structures are maximized over during the majority of these searches.

Dataset	O. R. score after 100 seconds	HillClimb after 3000 seconds	Winner
adult	-11.094	-11.132	O. R.
alarm	-10.839	-10.870	O. R.
biosurv	-6.993	-7.012	O. R.
covtype	-5.695	-5.714	O. R.
connect4	-14.975	-15.377	O. R.
edsgc	$-\infty$	-6.650	HC
synth2	-13.960	-16.847	O. R.
synth3	-13.993	-18.308	O. R.
synth4	-16.245	-18.588	O. R.
letters	-10.030	-10.080	O. R.
nursery	-9.720	-9.720	Tie

Table 4. How highly does 100 seconds of Optimal Reinsertion search score in comparison with thirty times the search time applied to hill climbing? In all but two cases Optimal Reinsertion gives a better result in one thirtieth the time. For the edsgc dataset, O. R. had not finished within 100 seconds, and so was beaten by default by hill climbing.

Data set	Significant winner on future data?
adult	No significant winner
alarm	Optimal Reinsertion
biosurv	No significant winner
connect4	Optimal Reinsertion
covtype	Optimal Reinsertion
edsgc	Optimal Reinsertion
synth2	Optimal Reinsertion
synth3	Optimal Reinsertion
synth4	Optimal Reinsertion
letters	No significant winner
nursery	Hill Climbing

Table 5. Which algorithm (if any) is significantly better at generalization to future data, given 300 seconds of computation? This was measured by a paired t-test at the 5% level on the results of 20-fold cross-validation.

3.6. Assessing Statistical Benefits

The above results give empirical support to the assertion that Optimal Reinsertion is faster and less local-minimum-prone at optimizing *DagScore*. But does that matter? In some applications it is the ability of the learned model to generalize to likelihood estimation of future data drawn from the same distribution that counts, and do the gains in *DagScore* translate to gains in performance on such future data? This question is not so much a test of our algorithm, but of whether the structure scoring metric (in these tests, BDEU) is doing its job adequately. Table 5 shows the results of 20-fold cross-validation. On each fold the left-out data is unused until the DAG and the Bayes Net parameters have been constructed from the training set. Then the log-likelihood of each held-out data point is recorded. This procedure is applied to both Optimal Reinsertion and Hill climbing, which are each allowed 5 minutes of computation. Table 5 shows that frequently, Optimal Reinsertion of BDEU has a significant generalization advantage (according to a paired t-test) over hill climbing optimization of BDEU.

3.7. Why is Hill-climbing beaten?

The synthetic cases provide the most obvious examples: given the XOR nodes there is no benefit in adding any one parent individually without the others and so hill-climbing can make no meaningful progress. We hypothesize that similar effects occur in some of the real datasets. The problems of single link removals and additions has been studied carefully in (Xiang et al., 1997). This Optimal Reinsertion implementation has an additional advantage over hill-climbing: the use of ADSEARCH means that once the cached node scores are computed there are no subsequent operations that require time proportional to the number of records.

3.8. Effects of *maxParams* and the Sparse Candidate k

The graphs in Figure 4 illustrate that as *maxParams* increases, so does both the time and quality of the solution. this is as expected. Additional results (not shown) illustrate a similar effect for the number of Sparse Candidates, k. As k grows we take longer and sometimes achieve better final results.

4. Related Work

We now discuss how the algorithms of this paper in the context of the most related recent work. We also discuss possible future developments to Optimal Reinsertion.

The Sparse Candidate Algorithm (Friedman et al., 1999). In its original form Sparse Candidate was a method to accelerate Hill climbing at the risk of a slightly lower final score. Empirically, the acceleration was large and the score sacrifice small. We believe that a combination of Sparse Candidate and Optimal Reinsertion will be superior to either alone. Our current use of a primitive Sparse Candidate approach (described in Section 2.7) could be generalized to a properly adaptive approach which iteratively updates its set of candidates.

Data Perturbation (Elidan et al., 2002). This very interesting new algorithm reduces local minimum problems very impressively by learning an initial net and then learning a second net with more weight given to records that were poorly modeled by the original. This process iterates. We believe this clever trick is orthogonal to Optimal Reinsertion and we believe that promising practical future work would implement data perturbation with Optimal Reinsertion as the inner loop search over the weighted data.

Massive Datasets. For truly massive datasets, many researchers have observed that working with a smaller sample may produce almost equal results compared with working with the full data. Several algorithms have been introduced that do this adaptively, with the algorithm dynamically determining from the data what sample size will be sufficient to very probably find a good answer, e.g. (Kaelbling, 1990; Maron & Moore, 1993; Hulten & Domingos, 2002; Pelleg & Moore, 2002). The most relevant recent example is (Hulten & Domingos, 2002) which learns Bayesian network structure from impressively massive datasets using adaptive sampling. For massive data the sampling algorithm uses only a tiny fraction of the full dataset with only moderate performance degradation in comparison to hill climbing. In methods such as this which work on small in-memory samples, it is possible that the increased speed and accuracy of Optimal Reinsertion methods may help their speed and accuracy even further.

Multi-link lookahead. (Xiang et al., 1997) show that a class of probabilistic domain models cannot be learned by algorithms that modify the network structure by a single link at a time. They propose a multi-link lookahead search for finding decomposable Markov Networks. This algorithm iterates over a number of levels where at level i, the current network is continually modified by the best set of i links until the entropy decrement fails to be significant. We plan to evaluate Optimal Reinsertion against an equivalent multi-link lookahead algorithm for Bayesian Networks.

Searching Equivalence Classes. There are other approaches to DAG learning. One which also searches the equivalent of very many DAGs on each step is (Chickering, 1996b). This searches an underlying space of a subclass of partial DAGs. Evaluations in this space can also be accelerated by a cache of scores obtainable from a fast enumeration of contingency tables, such as AD-search, but it will require further work to discover whether the equivalent of an Optimal Reinsertion operation exists.

Structural EM. A very important problem is to learn Bayesian network structure in datasets where some attributes of some records are missing. (Friedman, 1997) and subsequent publications have pioneered an EM approach to this problem. The EM approach requires repeated Bayesian Network structure optimizations and we plan to apply Optimal Reinsertion to this application, as one which will benefit greatly from the ability to do extremely fast search.

5. Conclusion

We have described and empirically examined a new search operator for learning Bayesian Network structure from fully observable data. The results are promising in comparison with hill-climbing and there is reason to believe that Optimal Reinsertion could be combined with the work of several other authors to eventually produce even faster search.

Acknowledgements

Supported by DARPA award F30602-01-2-0569 and NSF Grant 0121671. Thanks to Jeff Schneider and anonymous reviewers for helpful comments and suggestions.

References

Beinlich, I. A., Suermondt, H. J., Chavez, R. M., & Cooper, G. F. (1989). The alarm monitoring system: A case study with two probabilistic inference techniques for belief networks. *Proc. Second European Conference on AI and Medicine* (pp. 247–256). Berlin: Springer-Verlag.

Blake, C., & Merz, C. (1998). UCI Repository of machine learning databases. http://www.ics.uci.edu/∼mlearn/MLRepository.html.

Buntine, W. (1991). Theory Refinement on Bayesian Networks. *Proceedings of the Seventh Conference on UAI* (pp. 52–60).

Chickering, D. M. (1996a). Learning Bayesian networks is NP-Complete. In D. Fisher and H. Lenz (Eds.), *Learning from data: Artificial intelligence and statistics v*, 121–130. Springer-Verlag.

Chickering, D. M. (1996b). Learning equivalence classes of Bayesian network structures. *Proceedings of the Twelfth Conference on UAI*, Portland, OR (pp. 150–157). Morgan Kaufmann.

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

Elidan, G., Ninio, M., Friedman, N., & Schuurmans, D. (2002). Data perturbation for escaping local maxima in learning. *Proceedings of AAAI-02* (pp. 132–139).

Friedman, N. (1997). Learning belief networks in the presence of missing values and hidden variables. *Proc. 14th ICML* (pp. 125–133). Morgan Kaufmann.

Friedman, N., & Goldszmidt, M. (1997). Sequential update of Bayesian network structure. *Proceedings of the Thirteenth Conference on UAI* (pp. 165–174).

Friedman, N., Nachman, I., & Peér, D. (1999). Learning Bayesian network structure from massive datasets: The “sparse candidate” algorithm. *Proceedings of the Fifteenth Conference on UAI* (pp. 206–215).

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

Hulten, G., & Domingos, P. (2002). Mining complex models from arbitrarily large databases in constant time. *Proc. 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Kaelbling, L. P. (1990). *Learning in Embedded Systems.* PhD. Thesis; Technical Report No. TR-90-04). Stanford University, Department of Computer Science.

Maron, O., & Moore, A. (1993). Hoeffding Races: Accelerating Model Selection Search for Classification and Function Approximation. *Advances in NIPS 6*. Morgan Kaufmann.

Moore, A. W., & Lee, M. S. (1998). Cached Sufficient Statistics for Efficient Machine Learning with Large Datasets. *JAIR*, *8*, 67–91.

Moore, A. W., & Schneider, J. (2002). Real-valued All-Dimensions search: Low-overhead rapid searching over subsets of attributes. *Conference on UAI* (pp. 360–369).

Nichol, R. C., Collins, C. A., & Lumsden, S. L. (2000). The Edinburgh/Durham Southern Galaxy Catalogue - IX. The Galaxy Catalogue. http://xxx.lanl.gov/abs/astro-ph/0008184.

Pelleg, D., & Moore, A. W. (2002). Using Tarjan’s Red Rule for Fast Dependency Tree Construction. *NIPS 15*. Morgan Kaufmann.

Schwartz, G. (1979). Estimating the dimensions of a model. *Annals of Statistics*, *6*, 461–464.

Xiang, Y., Wong, S., & Cercone, N. (1997). A microscopic study of minimum entropy search in learning decomposable markov networks. *Machine Learning*, *26*, 65–92.