Journal of Machine Learning Research 3 (2002) 507-554 Submitted 1/02; Published 11/02

Optimal Structure Identification With Greedy Search

David Maxwell Chickering DMAX@MICROSOFT.COM
Microsoft Research
One Microsoft Way
Redmond, WA 98052

Editor: Craig Boutilier

Abstract

In this paper we prove the so-called "Meek Conjecture". In particular, we show that if a DAG $\mathcal{H}$ is an independence map of another DAG $\mathcal{G}$, then there exists a finite sequence of edge additions and covered edge reversals in $\mathcal{G}$ such that (1) after each edge modification $\mathcal{H}$ remains an independence map of $\mathcal{G}$ and (2) after all modifications $\mathcal{G} = \mathcal{H}$. As shown by Meek (1997), this result has an important consequence for Bayesian approaches to learning Bayesian networks from data: in the limit of large sample size, there exists a two-phase *greedy* search algorithm that—when applied to a particular sparsely-connected search space—provably identifies a perfect map of the generative distribution if that perfect map is a DAG. We provide a new implementation of the search space, using equivalence classes as states, for which all operators used in the greedy search can be scored efficiently using *local* functions of the nodes in the domain. Finally, using both synthetic and real-world datasets, we demonstrate that the two-phase greedy approach leads to good solutions when learning with finite sample sizes.

1. Introduction

Over the last decade, there has been an enormous amount of work in the machine-learning literature on the problem of learning Bayesian networks from data. In a recent Ph.D. dissertation on the topic, Meek (1997) put forth a conjecture that, if true, leads to the following and somewhat surprising result: given that the generative distribution has a perfect map in a DAG defined over the observables, then there exists a sparse search space (that is, a space in which each state is connected to a small fraction of the total states) to which we can apply a *greedy* search algorithm that, in the limit of large number of training cases, identifies the generative structure. The so-called "Meek Conjecture" can be stated as follows. Let $\mathcal{H}$ and $\mathcal{G}$ denote two DAGs such that $\mathcal{H}$ is an *independence map* of $\mathcal{G}$. In other words, any independence implied by the structure of $\mathcal{H}$ is also implied by the structure of $\mathcal{G}$. Then there exists a finite sequence of edge additions and covered edge reversals that can be applied to $\mathcal{G}$ with the following properties: (1) after each edge change, $\mathcal{G}$ is a DAG and $\mathcal{H}$ remains an independence map of $\mathcal{G}$ and (2) after all edge changes $\mathcal{G} = \mathcal{H}$. Although intuitively plausible, the validity of Meek's Conjecture has, until now, remained unknown. Kočka, Bouckaert and Studený (2001a) proved that the conjecture is true if $\mathcal{G}$ and $\mathcal{H}$ differ by exactly one edge.

In this paper, we prove Meek's Conjecture. We provide an algorithm for determining a specific sequence of edge modifications to $\mathcal{G}$ that transforms it into $\mathcal{H}$ such that after each

©2002 David Maxwell Chickering.

modification, $\mathcal{H}$ remains an independence map of $\mathcal{G}$. Assuming that initially there are m edges in $\mathcal{H}$ that do not appear in any orientation in $\mathcal{G}$ and r edges in $\mathcal{H}$ that appear in the opposite orientation in $\mathcal{G}$, the sequence includes at most $2m + r$ edge modifications.

Our algorithm is similar to the one proposed by Kočka, Bouckaert and Studený (2001b). In particular, the choice of an edge to modify depends on the parents and children of some node in $\mathcal{G}$ that is a sink node (i.e., a node with no children) in $\mathcal{H}$. For some configurations of parents and children of this node, it is reasonably easy both to (1) identify an edge modification and (2) prove that $\mathcal{H}$ remains an independence map after performing that modification; for such configurations, our algorithm and proof are essentially the same as that provided by Kočka et al. (2001b). There is a particular configuration of parents and children, however, for which it is more difficult to choose an edge to modify. For this configuration, Kočka et al. (2001b) conjecture that an appropriate edge modification exists, but are unable to construct a procedure to identify one.

Under the assumption that the conjecture is true, Meek (1997) devised a two-phase greedy algorithm that applies a Bayesian scoring criterion to identify the (unique) equivalence class of DAGs that is a perfect map of the generative distribution, assuming such an equivalence class exists. The algorithm can be summarized as follows. We start with an equivalence class corresponding to no dependencies, and greedily add dependencies by considering all possible single-edge additions that can be made to all DAGs in the current equivalence class. Once the greedy algorithm stops at a local maximum, we apply a second-phase greedy algorithm that considers at each step all possible single-edge deletions that can be made to all DAGs in the current equivalence class. The algorithm terminates with the local maximum identified in the second phase. The fact that the algorithm identifies (in the limit) the optimal equivalence class is rather remarkable given the sparsity of the search space; each state in the search space is connected to only as many other states as there are possible single-edge additions to or single-edge deletions from the DAGs in that state. Assuming that the generative model is small, we expect that this number of additions or deletions will also be small for those states we encounter during the search.

Given that the two-phase greedy algorithm has theoretical justification in light of Meek's Conjecture being true, the obvious question is whether the algorithm works well in practice. In other words, without regard to whether the generative distribution has a perfect map in a DAG or to whether there is enough data to support the asymptotic properties of the Bayesian scoring criterion, does the local maximum reached by the algorithm applied to real-world data correspond to a model that is close in score to the global maximum? Although we are unlikely to be able to answer this question without exhaustively enumerating and scoring all possible equivalence classes, we can compare the two-phase algorithm with other traditional search algorithms.

In order to perform the desired greedy search, we must be able to score all possible single-edge additions and deletions from all DAGs contained within an equivalence class. In principle, this might involve an actual enumeration of the DAGs within an equivalence class, and for each DAG, all edge changes could be scored. Fortunately, Chickering (2002) has formulated a search space that allows the efficient traversal of equivalence classes directly, as opposed to the more traditional approach of traversing in DAG space. Although the operators defined by Chickering (2002) do not correspond to the connectivity of equivalence classes necessary for the two-phase search, we can leverage the existing results to derive the

appropriate operators with relative ease. We show that all of the operators can be scored as local functions of the nodes and their neighbors in the equivalence-class representation of a search state, and thus the space shares the computational advantages of traditional DAG-based search spaces.

This paper is organized as follows. In Section 2, we describe our notation and introduce previous relevant work. In Section 3, we discuss Meek's conjecture and detail the algorithm we use to identify each edge modification necessary in the transformation. We postpone a rigorous proof of the conjecture to Appendix A, but provide some intuition for how we prove the most difficult step. In Section 4, we discuss the asymptotic properties of a Bayesian scoring criterion and show how these properties, in conjunction with the validity of Meek's Conjecture, imply the optimality of the two-phase greedy search algorithm. In Section 5, we describe a search space where the states of the search correspond to equivalence classes of DAGs, and for which the operators correspond to single edge additions and deletions to member DAGs. We show how all operators can be scored as local functions of the nodes in the search-state representation. In Section 6, we apply the two-phase greedy algorithm to both synthetic and real-world datasets of different sizes. We compare solution quality of the algorithm to (1) a traditional DAG-based greedy search algorithm and (2) a greedy search algorithm applied to an equivalence-class space defined by Chickering (2002). Using the synthetic data, we show that the two-phase algorithm is superior to the others at the task of reconstructing the generative structure. Using the real-world data, we show that the two-phase algorithm is competitive with the others—although slightly slower due to a more densely connected search space—at the task of identifying high-scoring models. In Section 7, we conclude with a summary and discussion of future relevant research. Detailed proofs of the main results of this paper are contained in the appendix.

2. Background and Notation

In this section, we introduce our notation and discuss previous relevant work. Throughout the paper, we use the following syntactical conventions. We denote a variable by an upper case letter (e.g., A, B_i, Y, Θ) and a state or value of that variable by the same letter in lower case (e.g., a, b_i, y, θ). We denote a set of variables by a bold-face capitalized letter or letters (e.g., $\mathbf{X}, \mathbf{Pa}_i, \mathbf{NA}_{i,j}$). We use a corresponding bold-face lower-case letter or letters (e.g., $\mathbf{x}, \mathbf{pa}_i, \mathbf{na}_{i,j}$) to denote an assignment of state or value to each variable in a given set. We use calligraphic letters (e.g., $\mathcal{G}, \mathcal{B}, \mathcal{E}$) to denote statistical models (both parameterized and not).

2.1 Bayesian-Network Models and DAG Models

A *parameterized Bayesian-network model* $\mathcal{B}$ for a set of variables $\mathbf{U} = \{X_1, \ldots, X_n\}$ is a pair $(\mathcal{G}, \boldsymbol{\theta})$. $\mathcal{G} = (\mathbf{V}, \mathbf{E})$ is a directed acyclic graph—or *DAG* for short—consisting of (1) nodes $\mathbf{V}$ in one-to-one correspondence with the variables $\mathbf{U}$, and (2) directed edges $\mathbf{E}$ that connect the nodes. $\boldsymbol{\theta}$ is a set of parameter values that specify all of the conditional probability distributions; we use $\boldsymbol{\theta}_i \subset \boldsymbol{\theta}$ to denote the subset of these parameter values that define the conditional probability of node X_i given its parents in $\mathcal{G}$. A parameterized Bayesian network represents a joint distribution over $\mathbf{U}$ that factors according to the structure $\mathcal{G}$ as

follows:

$$p_{\mathcal{B}}(X_1 = x_1, \ldots, X_n = x_n) = \prod_{i=1}^{n} p(X_i = x_i | \mathbf{Pa}_i^{\mathcal{G}} = \mathbf{pa}_i^{\mathcal{G}}, \boldsymbol{\theta}_i) \tag{1}$$

where $\mathbf{Pa}_i^{\mathcal{G}}$ is the set of parents of node x_i in $\mathcal{G}$. The structure $\mathcal{G}$ of a Bayesian network is itself a model that represents the independence constraints that must hold in any distribution that can be represented by a Bayesian network with that structure. The set of all independence constraints imposed by the structure $\mathcal{G}$ via Equation 1 can be characterized by the *Markov conditions*, which are the constraints that each variable is independent of its non-descendants given its parents. That is, any other independence constraint that holds can be derived from the Markov conditions (see, e.g., Pearl, 1988). We use $A \perp\!\!\!\perp_{\mathcal{G}} B | \mathbf{S}$ to denote the assertion that DAG $\mathcal{G}$ imposes the constraint that A is independent of B given set $\mathbf{S}$. When the DAG $\mathcal{G}$ is clear from context we use $A \perp\!\!\!\perp B | S$. When $\mathbf{S} = \emptyset$, we use $A \perp\!\!\!\perp_{\mathcal{G}} B$ (or $A \perp\!\!\!\perp B$) instead.

Throughout this paper we make numerous comparisons among statistical models; for example, we compare DAG models with each other and we compare properties of probability distributions with corresponding properties of DAGs. To simplify the discussion, we will assume that when any such comparison is made, the models are defined over the same set of variables. Thus when we say, for example, that two DAGs $\mathcal{G}$ and $\mathcal{G}'$ represent the same independence constraints, we assume that $\mathcal{G}$ and $\mathcal{G}'$ are defined over the same set of nodes.

The *descendants* of a node Y in $\mathcal{G}$—denoted $\mathbf{De}_Y^{\mathcal{G}}$—is the set containing Y and all nodes reachable by a directed path from Y. The *ancestors* of a node Y in $\mathcal{G}$ is the set of nodes that can reach Y by a directed path of length one or more. For any subset $\mathbf{A}$ of the nodes in $\mathcal{G}$, we say that a node $A \in \mathbf{A}$ is *maximal* if there is no other node $A' \in \mathbf{A}$ such that A' is an ancestor of A in $\mathcal{G}$.

2.2 Equivalence and Independence Maps

Two DAGs $\mathcal{G}$ and $\mathcal{G}'$ are *distributionally equivalent* if for every Bayesian network $\mathcal{B} = (\mathcal{G}, \boldsymbol{\theta})$, there exists a Bayesian network $\mathcal{B}' = (\mathcal{G}', \boldsymbol{\theta}')$ such that $\mathcal{B}$ and $\mathcal{B}'$ define the same probability distribution, and vice versa. Two DAGs $\mathcal{G}$ and $\mathcal{G}'$ are *independence equivalent* if the independence constraints in the two DAGs are identical. In most applications, researchers assume that the conditional distribution for each node in the Bayesian-network model comes from some specific family of distributions. For example, we might assume that the conditional probability of each continuous variable is a sigmoid distribution. Such distributional assumptions can sometimes impose non-independence constraints on the joint distribution that lead to DAGs that are independence equivalent but not distributionally equivalent. For the remainder of this paper, however, we will adopt the common distribution assumptions found in the literature on Bayesian-network learning; namely, we assume Gaussian distributions for continuous variables and unconstrained multinomial distributions for discrete variables. Under these assumptions, the two notions of equivalence are identical, and we will say that two DAGs $\mathcal{G}$ and $\mathcal{G}'$ are *equivalent* to indicate that they are both distributionally and independence equivalent.

We use $\mathcal{G} \approx \mathcal{G}'$ to denote that $\mathcal{G}$ and $\mathcal{G}'$ are equivalent. Because equivalence is reflexive, symmetric, and transitive, the relation defines a set of equivalence classes over network structures. We use $\mathcal{E}$ to denote an equivalence class of DAG models. Note that we use

the *non-bold* character $\mathcal{E}$; although arguably misleading in light of our convention to use bold-face for sets of variables, we use the non-bold character to emphasize the interpretation of $\mathcal{E}$ as a model for a set of independence constraints as opposed to a set of DAGs. We do, however, use the set-containment operator to denote DAG-elements of an equivalence class. Thus, we write $\mathcal{G} \in \mathcal{E}$ to denote that $\mathcal{G}$ is in equivalence class $\mathcal{E}$. To denote a particular equivalence class to which a DAG model $\mathcal{G}$ belongs, we sometimes write $\mathcal{E}(\mathcal{G})$. Note that $\mathcal{G} \approx \mathcal{G}'$ implies $\mathcal{G}' \in \mathcal{E}(\mathcal{G})$ and $\mathcal{G} \in \mathcal{E}(\mathcal{G}')$.

The *skeleton* of any DAG is the undirected graph resulting from ignoring the directionality of every edge. A *v-structure* in DAG $\mathcal{G}$ is an ordered triple of nodes (X, Y, Z) such that (1) $\mathcal{G}$ contains the edges $X \rightarrow Y$ and $Z \rightarrow Y$, and (2) X and Z are not adjacent in $\mathcal{G}$. Verma and Pearl (1991) provide the following characterization of equivalent DAG models.

Theorem 1 (Verma and Pearl, 1991) *Two DAGs are equivalent if and only if they have the same skeletons and the same v-structures.*

For any DAG $\mathcal{G} = (\mathbf{V}, \mathbf{E})$, we say an edge $X \rightarrow Y \in \mathbf{E}$ is *covered* in $\mathcal{G}$ if X and Y have identical parents, with the exception that X is not a parent of itself. That is, $X \rightarrow Y$ is covered in $\mathcal{G}$ if $\mathbf{Pa}_Y^{\mathcal{G}} = \mathbf{Pa}_X^{\mathcal{G}} \cup X$. The significance of covered edges is evident from the following result:

Lemma 2 (Chickering, 1995) *Let $\mathcal{G}$ be any DAG model, and let $\mathcal{G}'$ be the result of reversing the edge $X \rightarrow Y$ in $\mathcal{G}$. Then $\mathcal{G}'$ is a DAG that is equivalent to $\mathcal{G}$ if and only if $X \rightarrow Y$ is covered in $\mathcal{G}$.*

The following *transformational* characterization of equivalent DAG models will prove to be important to the main results of this paper.

Theorem 3 (Chickering, 1995) *Let $\mathcal{G}$ and $\mathcal{G}'$ be any pair of DAG models such that $\mathcal{G} \approx \mathcal{G}'$ and for which there are δ edges in $\mathcal{G}$ that have opposite orientation in $\mathcal{G}'$. Then there exists a sequence of δ distinct edge reversals in $\mathcal{G}$ with the following properties:*

1. *Each edge reversed in $\mathcal{G}$ is covered*
2. *After each reversal, $\mathcal{G}$ is a DAG and $\mathcal{G} \approx \mathcal{G}'$*
3. *After all reversals $\mathcal{G} = \mathcal{G}'$*

A DAG $\mathcal{H}$ is an *independence map* of a DAG $\mathcal{G}$ if every independence relationship in $\mathcal{H}$ holds in $\mathcal{G}$. We use $\mathcal{G} \leq \mathcal{H}$ to denote that $\mathcal{H}$ is an independence map of $\mathcal{G}$. The symbol '$\leq$' is meant to express the fact that if $\mathcal{G} \leq \mathcal{H}$ then $\mathcal{H}$ contains more edges than does $\mathcal{G}$. We can use the independence-map relation to compare *any* pair of models—not just DAG models—that impose independence constraints over a set of variables. We reserve the use of the symbol '$\leq$', however, to comparisons between DAG models.

An edge $X \rightarrow Y$ in $\mathcal{G}$ is *compelled* if that edge exists in every DAG that is equivalent to $\mathcal{G}$. If an edge $X \rightarrow Y$ in $\mathcal{G}$ is not compelled, we say that it is *reversible*. In light of Theorem 1, for any reversible edge $X \rightarrow Y$ in $\mathcal{G}$, there exists a DAG $\mathcal{G}'$ equivalent to $\mathcal{G}$ in which the edge is oriented in the opposite direction (i.e., $X \leftarrow Y$).

We say that a distribution $p(\cdot)$ is *contained* in a DAG $\mathcal{G}$ if there exists a set of parameter values $\boldsymbol{\theta}$ such that the parameterized Bayesian-network model $(\mathcal{G}, \boldsymbol{\theta})$ represents p exactly.

2.3 Learning Models from Data

As discussed in Section 1, our proof of Meek's conjecture leads to an optimal greedy algorithm for learning graphical models from data. We concentrate on Bayesian methods for learning graphical models, the roots of which date back to the work of Jeffreys (1939). We refer the reader to Heckerman (1995) or Buntine (1996) for a review of these methods and a more complete list of relevant references. As we discuss below, however, the algorithm can be used in conjunction with alternative learning methods.

Approaches to the Bayesian-network learning problem typically concentrate on identifying one or more DAG models that fit a set of observed data $\mathbf{D}$ well according to some scoring criterion $S(\mathcal{G}, \mathbf{D})$; once the structure of a Bayesian network is identified, it is usually straightforward to estimate the parameter values for a corresponding (parameterized) Bayesian network. In the Bayesian approach to learning DAG models we define, for each model $\mathcal{G}$, the *hypothesis* $\mathcal{G}^h$ that the observed data is a set of iid samples from a distribution that contains exactly the independence constraints implied by $\mathcal{G}$. The scoring criterion is then defined to be the relative posterior (or relative log posterior) of $\mathcal{G}^h$ given the observed data. A more detailed discussion of the Bayesian scoring criterion, as well as a discussion of alternative definitions of $\mathcal{G}^h$, is given in Section 4.

For any scoring criterion $S(\mathcal{G}, \mathbf{D})$, we say that S is *decomposable* if it can be written as a sum of measures, each of which is a function only of one node and its parents. In other words, a decomposable scoring criterion S applied to a DAG $\mathcal{G}$ can always be expressed as:

$$S(\mathcal{G}, \mathbf{D}) = \sum_{i=1}^{n} s(X_i, \mathbf{Pa}_i^{\mathcal{G}}) \tag{2}$$

Note that the data $\mathbf{D}$ is implicit in the right-hand side Equation 2. When we say that $s(X_i, \mathbf{Pa}_i^{\mathcal{G}})$ is only a function of X_i and its parents, we intend this also to mean that the *data* on which this measure depends is restricted to those columns corresponding to X_i and its parents. To be explicit, we could re-write the terms in the sum of Equation 2 as $s(X_i, \mathbf{D}(\{X_i\}), \mathbf{Pa}_i^{\mathcal{G}}, \mathbf{D}(\mathbf{Pa}_i^{\mathcal{G}}))$, where $\mathbf{D}(\mathbf{X})$ denotes the data restricted to the columns corresponding to the variables in set $\mathbf{X}$. We find it convenient, however, to keep the notation simple.

Most scoring criteria derived in the literature are decomposable. An important property of decomposable scoring criteria is that if we want to compare the scores of two DAGs $\mathcal{G}$ and $\mathcal{G}'$, we need only compare those terms in Equation 2 for which the corresponding nodes have different parent sets in the two graphs. This proves to be particularly convenient for search algorithms that consider single edge changes to DAGs; in Section 5 we show how using a decomposable scoring criterion leads to an efficient implementation of the two-phase greedy search algorithm of Meek (1997).

A scoring criterion $S(\mathcal{G}, \mathbf{D})$ is *score equivalent* if, for any pair of equivalent DAGs $\mathcal{G}$ and $\mathcal{G}'$, it is necessarily the case that $S(\mathcal{G}, \mathbf{D}) = S(\mathcal{G}', \mathbf{D})$.

2.4 Completed PDAGs

In our implementation of the greedy search algorithm presented in Section 5, we search through equivalence classes of DAG models as opposed to DAG models themselves. As is

done by Chickering (2002), we use *completed PDAGs*—which we define below—to represent equivalence classes.

An acyclic partially directed graph, or *PDAG* for short, is a graph that contains both directed and undirected edges, and can be used to represent an equivalence class of DAGs. Let $\mathcal{P}$ denote an arbitrary PDAG. We define the equivalence class of DAGs $\mathcal{E}(\mathcal{P})$ corresponding to $\mathcal{P}$ as follows: $\mathcal{G} \in \mathcal{E}(\mathcal{P})$ if and only if $\mathcal{G}$ and $\mathcal{P}$ have the same skeleton and the same set of v-structures.[1] From Theorem 1, it follows that a PDAG containing a directed edge for every edge participating in a v-structure and an undirected edge for every other edge uniquely identifies an equivalence class of DAGs. There may be many other PDAGs, however, that correspond to the same equivalence class. For example, any DAG interpreted as a PDAG can be used to represent its own equivalence class.

If a DAG $\mathcal{G}$ has the same skeleton and the same set of v-structures as a PDAG $\mathcal{P}$ *and* if every directed edge in $\mathcal{P}$ has the same orientation in $\mathcal{G}$, we say that $\mathcal{G}$ is a *consistent extension* of $\mathcal{P}$. Any DAG that is a consistent extension of $\mathcal{P}$ must also be contained in $\mathcal{E}(\mathcal{P})$, but not every DAG in $\mathcal{E}(\mathcal{P})$ is a consistent extension of $\mathcal{P}$. If there is at least one consistent extension of a PDAG $\mathcal{P}$, we say that $\mathcal{P}$ *admits a consistent extension*.

We use *completed* PDAGs to represent equivalence classes of DAGs. Recall that a compelled edge is an edge that exists in the same orientation for every member of an equivalence class, and that a reversible edge is an edge that is not compelled. The completed PDAG corresponding to an equivalence class is the PDAG consisting of a directed edge for every compelled edge in the equivalence class, and an undirected edge for every reversible edge in the equivalence class. Given an equivalence class of DAGs, the completed-PDAG representation is unique. Also, every DAG in an equivalence class is a consistent extension of the completed PDAG representation for that class. Figure 1a shows a DAG $\mathcal{G}$, and Figure 1b shows the completed PDAG for $\mathcal{E}(\mathcal{G})$. PDAGs are called *patterns* by (e.g.) Spirtes, Glymour and Scheines (1993) and completed PDAGs are called *essential graphs* by (e.g.) Andersson, Madigan and Perlman (1997) and *maximally oriented graphs* by Meek (1995).

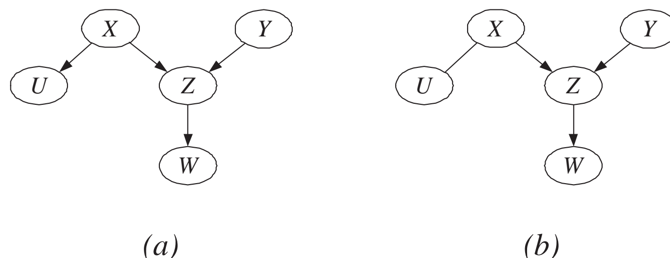

Figure 1: (a) a DAG $\mathcal{G}$ and (b) the completed PDAG for $\mathcal{E}(\mathcal{G})$

1. The definitions for the skeleton and set of v-structures for a PDAG are the obvious extensions to these definitions for DAGs.

3. Meek's Conjecture

In this section, we discuss Meek's conjecture and detail the constructive algorithm used to prove that the conjecture is true. We provide some examples to help illustrate the algorithm and to give some insight into why researchers have been unable to solve this problem. The detailed proof is postponed to Appendix A.

Recall the transformational characterization of equivalence from Section 2.2, which states that $\mathcal{G} \approx \mathcal{G}'$ if and only if we can transform $\mathcal{G}$ into $\mathcal{G}'$ by a sequence of covered edge reversals. Meek's conjecture is an analogous characterization of the independence-map relation. In particular, Meek's conjecture states that $\mathcal{G} \leq \mathcal{H}$ if and only if we can transform $\mathcal{G}$ into $\mathcal{H}$ by a sequence of (1) covered edge reversals and (2) single edge additions. More formally, we now state the main result of this paper.

Theorem 4 *Let $\mathcal{G}$ and $\mathcal{H}$ be any pair of DAGs such that $\mathcal{G} \leq \mathcal{H}$. Let r be the number of edges in $\mathcal{H}$ that have opposite orientation in $\mathcal{G}$, and let m be the number of edges in $\mathcal{H}$ that do not exist in either orientation in $\mathcal{G}$. There exists a sequence of at most $r + 2m$ edge reversals and additions in $\mathcal{G}$ with the following properties:*

1. *Each edge reversed is a covered edge*
2. *After each reversal and addition $\mathcal{G}$ is a DAG and $\mathcal{G} \leq \mathcal{H}$*
3. *After all reversals and additions $\mathcal{G} = \mathcal{H}$*

Our proof of Theorem 4 is constructive: we define an algorithm, shown in Figure 2, that takes as input two DAGs $\mathcal{G}$ and $\mathcal{H}$ such that $\mathcal{G} \leq \mathcal{H}$, and identifies an edge in $\mathcal{G}$ that can either be added or reversed. We show that after the edge modification is made by the algorithm (1) $\mathcal{H}$ remains an independence map of $\mathcal{G}$ and (2) $\mathcal{G}$ is "closer" to $\mathcal{H}$ in the sense that it has either fewer adjacency differences or the same number of adjacency differences and fewer orientation differences. Theorem 4 is an immediate consequence of the validity of ALGORITHM APPLY-EDGE-OPERATION because we can convert $\mathcal{G}$ into $\mathcal{H}$ by calling the algorithm repeatedly, replacing $\mathcal{G}$ after each call with the result of the algorithm, until $\mathcal{G} = \mathcal{H}$.

In words, the algorithm works as follows. First, all common sink nodes that have identical parents in the two DAGs are removed from both DAGs. By "remove" we mean "remove from consideration"; in practice, the input DAGs need not be modified, and it is to be understood that ALGORITHM APPLY-EDGE-OPERATION uses a "by value" calling convention so that both $\mathcal{G}$ and $\mathcal{H}$ are local variables that the algorithm can modify without side effects. The algorithm next identifies a sink node Y in $\mathcal{H}$. If Y is also a sink node in $\mathcal{G}$, the algorithm chooses any parent X of node Y in $\mathcal{H}$ that is not a parent of Y in $\mathcal{G}$, and adds the edge $X \rightarrow Y$ to $\mathcal{G}$. Otherwise, there is at least one edge $Y \rightarrow Z$ in $\mathcal{G}$ that is oriented in the opposite direction in $\mathcal{H}$, and the algorithm identifies a unique such edge via Step 5 (this step will be discussed in more detail below). If the edge $Y \rightarrow Z$ is covered in $\mathcal{G}$, the algorithm reverses the edge and terminates. Otherwise, it follows by definition of a covered edge that in $\mathcal{G}$ there is either (1) a parent X of Y that is not a parent of Z, in which case the algorithm adds the edge $X \rightarrow Z$ or (2) a parent X of Z that is not a parent of Y, in which case the algorithm adds the edge $X \rightarrow Y$.

Algorithm Apply-Edge-Operation($\mathcal{G}, \mathcal{H}$)
Input: DAGs $\mathcal{G}$ and $\mathcal{H}$ where $\mathcal{G} \leq \mathcal{H}$ and $\mathcal{G} \neq \mathcal{H}$
Output: DAG $\mathcal{G}'$ that results from adding or reversing an edge in $\mathcal{G}$.

1. **Set** $\mathcal{G}' = \mathcal{G}$.

2. **While** $\mathcal{G}$ and $\mathcal{H}$ contain a node Y that is a sink in both DAGs and for which $\mathbf{Pa}_Y^{\mathcal{G}} = \mathbf{Pa}_Y^{\mathcal{H}}$, remove Y and all incident edges from both DAGs.

3. Let Y be any sink node in $\mathcal{H}$

4. **If** Y has no children in $\mathcal{G}$, then let X be any parent of Y in $\mathcal{H}$ that is not a parent of Y in $\mathcal{G}$. Add the edge $X \rightarrow Y$ to $\mathcal{G}'$ and **Return** $\mathcal{G}'$.

5. Let $\mathbf{De}_Y^{\mathcal{G}}$ denote the descendants of Y in $\mathcal{G}$, and let $D \in \mathbf{De}_Y^{\mathcal{G}}$ denote the (unique) maximal element from this set within $\mathcal{H}$.[2] Let Z be any maximal child of Y in $\mathcal{G}$ such that D is a descendant of Z in $\mathcal{G}$.

6. **If** $Y \rightarrow Z$ is covered in $\mathcal{G}$, reverse $Y \rightarrow Z$ in $\mathcal{G}'$ and **Return** $\mathcal{G}'$.

7. **If** there exists a node X that is a parent of Y but not a parent of Z in $\mathcal{G}$, then add $X \rightarrow Z$ to $\mathcal{G}'$ and **Return** $\mathcal{G}'$.

8. Let X be any parent of Z that is not a parent of Y. Add $X \rightarrow Y$ to $\mathcal{G}'$ and **Return** $\mathcal{G}'$.

Figure 2: Algorithm that identifies and applies an edge modification

In the examples to follow, we will assume that the reader is familiar with the *d-separation* criterion used to test independence relationships in DAG models. Those who are not familiar with this criterion can refer to Appendix A for a detailed definition and description. In Figure 3, we give an example application of the algorithm. Consider the two DAGs $\mathcal{G}$ and $\mathcal{H}$ shown in Figure 3a and Figure 3b, respectively. It is easy to verify that $\mathcal{G} \leq \mathcal{H}$ by testing that the unique Markov conditions in $\mathcal{H}$ (i.e., $A \perp\!\!\!\perp B, A \perp\!\!\!\perp E|\{C\}, B \perp\!\!\!\perp E|\{C\}$) also hold via d-separation in $\mathcal{G}$. Now consider a call to Algorithm Apply-Edge-Operation($\mathcal{G}, \mathcal{H}$). There are no common sink nodes, so the algorithm does not remove any nodes in Step 2. Node E is the only sink node in $\mathcal{H}$, and because C is the only child of E in $\mathcal{G}$, it is easy to see that the edge tested in Step 6 is $E \rightarrow C$. This edge is covered in $\mathcal{G}$, so the algorithm reverses it and terminates. The resulting DAG is shown in Figure 3c. We can now call Algorithm Apply-Edge-Operation($\mathcal{G}', \mathcal{H}$) once again, using the DAG $\mathcal{G}'$ that was returned from the previous call to the function. For this call, both DAGs contain the sink node E with the single parent C, and thus node E is removed from consideration from both DAGs. After this removal, there are no remaining common sinks with the same parents, so the algorithm proceeds to Step 3 and identifies C as a sink node in $\mathcal{H}$. Again there is only a single child to identify in Step 5, and thus the edge tested in Step 6 is $C \rightarrow A$. This edge is covered, and thus the algorithm terminates with the DAG shown in Figure 3d.

2. D is guaranteed to be unique by Lemma 29 in Appendix A.

We call Algorithm Apply-Edge-Operation a final time, with the first DAG equal to the one that was returned in the previous call. After removing E from both DAGs, the algorithm adds the edge $B \to C$ in Step 4, and the resulting DAG—shown in Figure 3e—is identical to $\mathcal{H}$.

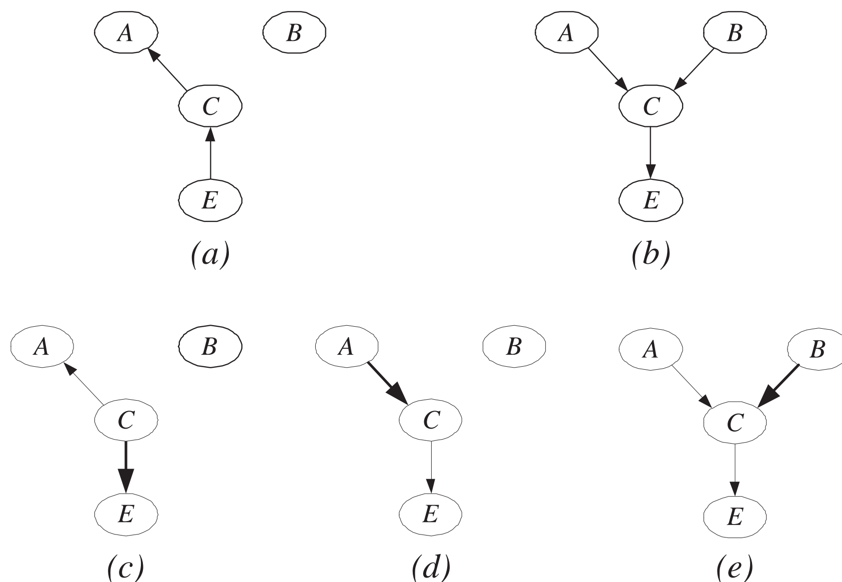

Figure 3: DAGs in an example application of Algorithm Apply-Edge-Operation. (a) Original DAG $\mathcal{G}$ and (b) DAG $\mathcal{H}$. (c), (d), and (e) show the DAGs resulting from successive calls to the algorithm.

As mentioned in Section 1, the validity of all but one of the edge modifications can be proved with relative ease. The difficult case to prove is when there is a child of the sink node Y that has a parent that is not a parent of Y. In this case, Step 8 will be encountered, and the key step is the selection of the specific such child Z in Step 5 of the algorithm. If Step 8 were never encountered, we could use a *much* simpler method for choosing Z in Step 5. In particular, it would suffice to choose any maximal child of Y. To illustrate the difficult case, we consider a single step of an example that was given by Kočka et al. (2001b) and is shown in Figure 4. Given the choice of adding either $X_1 \to T$ or $X_2 \to T$ to $\mathcal{G}$, only the second addition yields a DAG $\mathcal{G}'$ such that $\mathcal{H}$ remains an independence map. In particular, if we add the edge $X_1 \to T$, then the independence $X_1 \perp\!\!\!\perp_{\mathcal{H}} C_2$ does not hold in the resulting DAG. We now show that the correct choice between these two additions is made by a call to our algorithm. There are no common sink nodes, and the unique sink node from $\mathcal{H}$ is T. The set of descendants of T in $\mathcal{G}$ is $\{T, C_1, C_2\}$, and in $\mathcal{H}$ the maximal element D in this set is $D = C_2$. The maximal child of T in $\mathcal{G}$ that has $D = C_2$ as a descendant is C_2 itself, and thus the edge $T \to C_2$ is chosen by the algorithm to be considered for Steps 6 to 8. This edge is not covered in $\mathcal{G}$, and C_2 has the parent X_2 that is not a parent of T, and thus the algorithm adds the edge $X_2 \to T$ in Step 8.

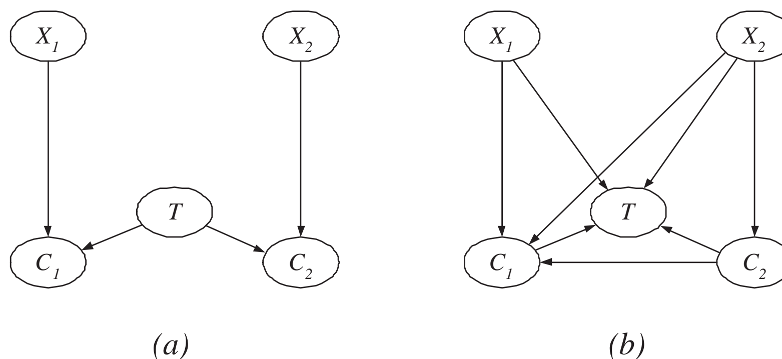

Figure 4: DAGs (a) $\mathcal{G}$ and (b) $\mathcal{H}$ in an example application of Algorithm Apply-Edge-Operation.

To fully understand how the selection of Z in Step 5 guarantees that the addition is valid, the reader should study the proof in Appendix A. For those who would like simply to gain some intuition, however, we now provide some insight into Step 5. Our discussion assumes familiarity with the d-separation criterion, as well as familiarity with the concept of an *active path* that defines the criterion.[3] Again, readers not familiar with these concepts can consult Appendix A. We provide relevant portions of both $\mathcal{G}$ and $\mathcal{H}$ in Figure 5a and Figure 5b, respectively, to help clarify the discussion.

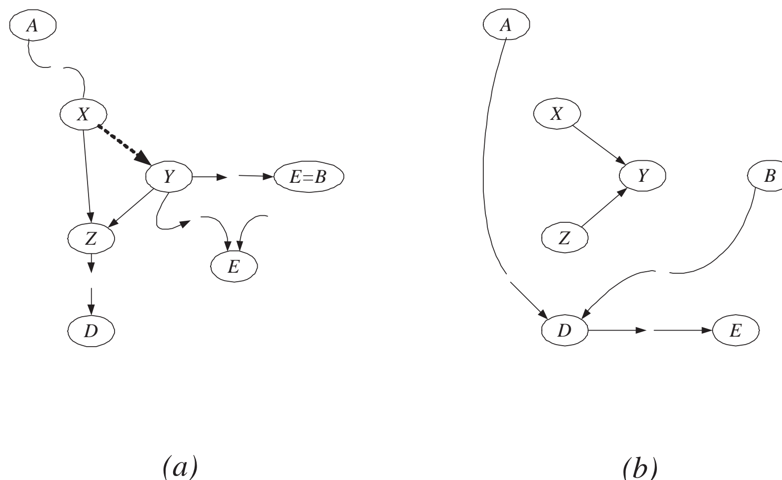

Figure 5: Relevant portion of (a) $\mathcal{G}$ and (b) $\mathcal{H}$ that demonstrate how Step 5 leads to a valid edge addition in Step 8.

3. We use a non-standard definition of an active path in Appendix A, but the standard definition will suffice in the present discussion.

Recall that node D from Step 5 is the maximal element of $\mathbf{De}_Y^{\mathcal{G}}$ with respect to $\mathcal{H}$. That is, in Step 5 we look at the descendents of Y in $\mathcal{G}$, and then pick the maximal descendant with respect to $\mathcal{H}$. Note that because Y is a sink in $\mathcal{H}$, $D \neq Y$.

The potential problem of the addition of the edge $X \rightarrow Y$ to $\mathcal{G}$ is that some active path between two nodes A and B given a conditioning set $\mathbf{S}$ exists in the resulting graph $\mathcal{G}'$, where no such active path exists in either $\mathcal{G}$ or $\mathcal{H}$. It is reasonably easy to show that the following three properties hold if there exists such an active path in $\mathcal{G}'$: (1) there must exist at least one such path that includes the edge $X \rightarrow Y$, (2) neither Z nor any descendant of Z (including D) in $\mathcal{G}$ can belong to the conditioning set, and (3) Y is not in the conditioning set.

The first conclusion we make from these three properties (see Figure 5a) is that there must be a descendant E of Y in $\mathcal{G}'$—and hence E is also a descendant of Y in $\mathcal{G}$—that is either in $\mathbf{S}$ or is one of the endpoints (B in the figure). This follows because any first non-descendant node along the path after $X \rightarrow Y$ follows a head-to-head junction (i.e., collider) along the path.

The second conclusion we make is that, because of our choice of Z, there must be an active path between both (1) A and D and (2) B and D in the DAG $\mathcal{G}$ (it is easy to show that there are active paths between both endpoints and Z, and there is a directed path in $\mathcal{G}$ from Z to D that does not pass through any node in $\mathbf{S}$). Thus in $\mathcal{H}$ (see Figure 5b), there must also exist active paths between both endpoints and D. Furthermore, because $D \notin \mathbf{S}$ (Property 2), both of these paths must end with an edge *into* D (i.e., $A - \ldots \rightarrow D$ and $B - \ldots \rightarrow D$) or else we could concatenate them together and identify an active path between A and B in $\mathcal{H}$. But this implies that (*) none of the descendants of D in $\mathcal{H}$ can be in $\mathbf{S}$, or else the concatenation of the active paths would be active, and (**) none of the descendants of D in $\mathcal{H}$ can be an endpoint, or else the concatenation of the directed path from D to that endpoint (through nodes not in $\mathbf{S}$, where the first edge is *away* from D) can be connected with the active path from the *other* endpoint to form an active path.

Now the logic of the choice of D in Step 5 becomes clear. Because D is the maximal node in $\mathcal{H}$ out of all of the descendants of Y in $\mathcal{G}$, D is an *ancestor* in $\mathcal{H}$ of all of these nodes as well (see Lemma 29 in Appendix A). This means that D is an ancestor of E in $\mathcal{H}$, which from (*) and (**) yield a contradiction.

4. The Optimality of Greedy Search

In this section, we describe the two-phase greedy search algorithm proposed by Meek (1997), and show that in the limit of large samples, the algorithm identifies the DAG corresponding to the generative model if such a model exists. Here we are concerned with the theoretical properties of the algorithm; we postpone discussing implementation details to Section 5.

To be more precise about the optimality result in this section, we need the following notation. Given a DAG $\mathcal{G}$ and a probability distribution $p(\cdot)$, we say that $\mathcal{G}$ is a *perfect map* of p if (1) every independence constraint in p is implied by the structure $\mathcal{G}$ and (2) every independence implied by the structure $\mathcal{G}$ holds in p. If there exists some DAG that is a perfect map of a probability distribution $p(\cdot)$, we say that p is *DAG-perfect*.

Assumption 1 *Each case in the observed data* $\mathbf{D}$ *is an iid sample from some DAG-perfect probability distribution* $p(\cdot)$.

We allow there to be missing values in each iid sample, but our results implicitly depend on the assumption that the parameters of each Bayesian network are identifiable. We will therefore assume for the remainder of this section that the empirical distribution defined by the data **D** converges to $p(\cdot)$ as the number of records grows large.

The remainder of this section is organized as follows. In Section 4.1, we explore the asymptotic behavior of the Bayesian scoring criterion, and in Section 4.2, we detail the two-phase greedy algorithm and show how it takes advantage of that asymptotic behavior to identify the optimal solution. Finally, in Section 4.3, we discuss the applicability of the algorithm to non-Bayesian scoring criteria and to Bayesian scoring criteria for which the definition of the structure hypothesis differs from the one we presented in Section 2.3. We also discuss how violations of Assumption 1 can affect the solution quality of the algorithm.

4.1 Asymptotic Behavior of the Bayesian Scoring Criterion

Recall from Section 2 that the Bayesian scoring criterion for a DAG $\mathcal{G}$ measures the relative posterior or relative log posterior of the hypothesis $\mathcal{G}^h$ that the independence constraints in $\mathcal{G}$ are precisely the independence constraints in the generative distribution. Without loss of generality, we express the Bayesian scoring criterion S_B using the relative log posterior of $\mathcal{G}^h$:

$$S_B(\mathcal{G}, \mathbf{D}) = \log p(\mathcal{G}^h) + \log p(\mathbf{D}|\mathcal{G}^h) \tag{3}$$

where $p(\mathcal{G}^h)$ is the prior probability of $\mathcal{G}^h$, and $p(\mathbf{D}|\mathcal{G}^h)$ is the *marginal likelihood*. The marginal likelihood is obtained by integrating the likelihood function (i.e., Equation 1) applied to each record in **D** over the unknown parameters of the model.

Definition 5 (Consistent Scoring Criterion)
Let **D** *be a set of data consisting of m records that are iid samples from some distribution $p(\cdot)$. A scoring criterion S is* consistent *if in the limit as m grows large, the following two properties hold:*

1. *If $\mathcal{H}$ contains p and $\mathcal{G}$ does not contain p, then $S(\mathcal{H}, \mathbf{D}) > S(\mathcal{G}, \mathbf{D})$*
2. *If $\mathcal{H}$ and $\mathcal{G}$ both contain p, and $\mathcal{G}$ contains fewer parameters than $\mathcal{H}$, then $S(\mathcal{G}, \mathbf{D}) > S(\mathcal{H}, \mathbf{D})$*

Geiger, Heckerman, King and Meek (2001) show that the models we consider in this paper (i.e., those containing Gaussian or multinomial distributions) are *curved exponential models*. The details of this class of model are not important for our results, but Haughton (1988) shows that (under mild assumptions about the parameter prior) the Bayesian scoring criterion is consistent for curved exponential models. In particular, Haughton (1988) shows that Equation 3 for curved exponential models can be approximated using Laplace's method for integrals, yielding

$$S_B(\mathcal{G}, \mathbf{D}) = \log p(\mathbf{D}|\hat{\boldsymbol{\theta}}, \mathcal{G}^h) - \frac{d}{2}\log m + O(1) \tag{4}$$

where $\hat{\boldsymbol{\theta}}$ denotes the maximum-likelihood values for the network parameters, d denotes the dimension (i.e., number of free parameters) of $\mathcal{G}$, and m is the number records in **D**.

The first two terms in this approximation are known as the *Bayesian information criterion* (or BIC). The presence of the $O(1)$ error means that, even as m approaches infinity, the approximation can differ from the true relative log posterior by a constant. As shown by Haughton (1988), however, BIC is consistent. Furthermore, it is easy to show that the leading term in BIC grows as $O(m)$, and therefore we conclude that because the error term becomes increasingly less significant as m grows large, Equation 3 is consistent as well. Because the prior term $p(\mathcal{G}^h)$ does not depend on the data, it does not grow with m and therefore is absorbed into the error term of Equation 4. Thus the asymptotic behavior of the Bayesian scoring criterion depends only on the marginal likelihood term.

Consistency of the Bayesian scoring criterion leads, from the fact that BIC is decomposable, to a more practical property of the criterion that we call *local consistency.* Intuitively, if a scoring criterion is locally consistent, then the score of a DAG model $\mathcal{G}$ (1) *increases* as the result of adding any edge that eliminates an independence constraint that does not hold in the generative distribution, and (2) *decreases* as a result of adding any edge that does not eliminate such a constraint. More formally, we have the following definition.

Definition 6 (Locally Consistent Scoring Criterion)
Let $\mathbf{D}$ *be a set of data consisting of* m *records that are iid samples from some distribution* $p(\cdot)$. *Let* $\mathcal{G}$ *be any DAG, and let* $\mathcal{G}'$ *be the DAG that results from adding the edge* $X_i \rightarrow X_j$. *A scoring criterion* $S(\mathcal{G}, \mathbf{D})$ *is* locally consistent *if the following two properties hold:*

1. *If* $X_j \not\perp\!\!\!\perp_p X_i | \mathbf{Pa}_j^{\mathcal{G}}$, *then* $S(\mathcal{G}', \mathbf{D}) > S(\mathcal{G}, \mathbf{D})$
2. *If* $X_j \perp\!\!\!\perp_p X_i | \mathbf{Pa}_j^{\mathcal{G}}$, *then* $S(\mathcal{G}', \mathbf{D}) < S(\mathcal{G}, \mathbf{D})$

Lemma 7 *The Bayesian scoring criterion is locally consistent.*

Proof: The proof follows from the fact that in the limit, the criterion ranks models in the same order as BIC. Because BIC is decomposable, the increase in score that results from adding the edge $X_i \rightarrow X_j$ to any DAG $\mathcal{G}$ is *the same* as the increase in score that results from adding the edge to any other DAG $\mathcal{H}$ for which X_j has the same parents. We can therefore choose a particular $\mathcal{H}$—where $\mathbf{Pa}_j^{\mathcal{H}} = \mathbf{Pa}_j^{\mathcal{G}}$—for which adding the edge $X_i \rightarrow X_j$ results in a *complete* DAG $\mathcal{H}'$; that is, $\mathcal{H}'$ has an edge between every pair of nodes. Because the complete DAG imposes no constraints on the joint distribution, the lemma follows immediately from the consistency of BIC. □

From Lemma 7, we see that as long as there are edges that can be added to a DAG that eliminate independence constraints not contained in the generative distribution, the Bayesian scoring criterion will favor such an addition. If the DAG contains the distribution, then Lemma 7 guarantees that any deletion of an "unnecessary" edge will be favored by the criterion. These properties allow us to prove the optimality of the greedy search algorithm presented in the following section.

4.2 A Two-Phase Optimal Greedy Search Algorithm

In this section, we first detail the two-phase greedy search algorithm, called *Greedy Equivalence Search* or GES by Meek (1997). Then we present—using the results of Section 4.1—a version of the proof of Meek (1997) that GES is optimal in the limit of large datasets.

For the remainder of this section, we will assume that we are using the Bayesian scoring criterion in conjunction with the search algorithm.

Up to this point, we have concentrated on DAG models in our discussion of learning from data. We find it convenient now to switch to an equivalence-class interpretation of both DAG hypotheses and the Bayesian scoring criterion in order to more clearly present the GES algorithm. From the definition of $\mathcal{G}^h$, it follows that all DAGs in the same equivalence class correspond to the same hypothesis. That is, if $\mathcal{G} \approx \mathcal{H}$ then $\mathcal{G}^h = \mathcal{H}^h$. Thus we can use $\mathcal{E}^h$ to denote the hypothesis corresponding to the (identical) hypotheses of the DAGs contained within $\mathcal{E}$. Furthermore, by definition of the Bayesian scoring criterion, the score of a DAG model in equivalence class $\mathcal{E}$ is the (relative) log posterior of $\mathcal{E}^h$; thus, the Bayesian scoring criterion is well defined for equivalence classes, and can be evaluated using any DAG member of the class. We will use $S_B(\mathcal{E}, \mathbf{D})$ to denote the score for equivalence class $\mathcal{E}$ using the Bayesian scoring criterion.

Before proceeding, we show that the equivalence class that is a perfect map[4] of the generative distribution is the optimal solution.

Proposition 8 *Let $\mathcal{E}^*$ denote the equivalence class that is a perfect map of the generative distribution $p(\cdot)$, and let m denote the number of records in $\mathbf{D}$. Then in the limit of large m, $S_B(\mathcal{E}^*, \mathbf{D}) > S_B(\mathcal{E}, \mathbf{D})$ for any $\mathcal{E} \neq \mathcal{E}^*$.*

Proof: Suppose this is not the case, and there exists a higher-scoring equivalence class $\mathcal{E} \neq \mathcal{E}^*$. Because the scoring criterion is consistent, it must be the case that $\mathcal{E}$ contains p; furthermore, because $\mathcal{E}^*$ is a perfect map of p, it follows that $\mathcal{E}$ must be an independence map of $\mathcal{E}^*$. Let $\mathcal{G}$ be any DAG in $\mathcal{E}^*$, and let $\mathcal{H}$ be any DAG in $\mathcal{E}$. Because $\mathcal{G} \leq \mathcal{H}$, we know from Theorem 4 that there exists a sequence of covered edge reversals and edge additions that transforms $\mathcal{G}$ into $\mathcal{H}$. After each covered edge reversal, the score of $\mathcal{G}$ remains the same because (by Lemma 2) it remains in the same equivalence class. After each edge addition, however, the number of parameters in the DAG necessarily increases, and because the scoring criterion is consistent, the score necessarily *decreases*. Because $\mathcal{E}$ is optimal, there can therefore be no edge additions in the transformation, which contradicts the supposition that $\mathcal{E} \neq \mathcal{E}^*$. $\square$

As suggested by the name, GES is a greedy algorithm that searches over equivalence classes of DAGs. Greedy search (in general) proceeds at each step by evaluating each *neighbor* of the current state, and moving to the one with the highest score if doing so improves the score. The set of neighbors of each state in the search defines the connectivity of the search space. GES consists of two phases. In the first phase, a greedy search is performed over equivalence classes using a particular connectivity between equivalence classes. Once a local maximum is reached, a second phase proceeds from the previous local maximum using a second connectivity. When the second phase reaches a local maximum, that equivalence class is returned as the solution.

We use $\boldsymbol{\mathcal{E}}^+(\mathcal{E})$ to denote the neighbors of state $\mathcal{E}$ during the first phase of GES. In words, an equivalence class $\mathcal{E}'$ is in $\boldsymbol{\mathcal{E}}^+(\mathcal{E})$ if and only if there is some DAG $\mathcal{G} \in \mathcal{E}$ to which we can add a single edge that results in a DAG $\mathcal{G} \in \mathcal{E}'$. Given Theorem 3, an alternative way of describing $\boldsymbol{\mathcal{E}}^+(\mathcal{E})$ is as follows. Let $\mathcal{G}$ be any DAG in $\mathcal{E}$ and let $\mathcal{G}'$ be any DAG in $\mathcal{E}'$. Then

4. The definition of *perfect map* for equivalence classes is the obvious extension of the definition for DAGs.

$\mathcal{E}' \in \boldsymbol{\mathcal{E}}^+(\mathcal{E})$ if and only if there exists a sequence of covered edge reversals followed by a single edge addition followed by another sequence of covered edge reversals that transform $\mathcal{G}$ into $\mathcal{G}'$. We use $\boldsymbol{\mathcal{E}}^-(\mathcal{E})$ to denote the neighbors of state $\mathcal{E}$ during the second phase of GES. The definition of $\boldsymbol{\mathcal{E}}^-(\mathcal{E})$ is completely analogous to that of $\boldsymbol{\mathcal{E}}^+(\mathcal{E})$, and contains equivalence classes that are obtained by *deleting* a single edge from DAGs in $\mathcal{E}$.

In Figure 6a, we show a particular DAG $\mathcal{G}$, and in Figure 6b, we show all members of $\mathcal{E} = \mathcal{E}(\mathcal{G})$. In Figure 6c, we show all DAGs reachable by a single edge addition to a member of $\mathcal{E}$. The union of the corresponding equivalence classes constitutes $\boldsymbol{\mathcal{E}}^+(\mathcal{E})$; because all of the DAGs in Figure 6c are equivalent, $\boldsymbol{\mathcal{E}}^+(\mathcal{E})$ contains a single equivalence class (corresponding to the "no independence constraints" hypothesis). In Figure 6d, we show all DAGs reachable by a single edge deletion from a member of $\mathcal{E}$. The union of the two corresponding equivalence classes constitutes $\boldsymbol{\mathcal{E}}^-(\mathcal{E})$.

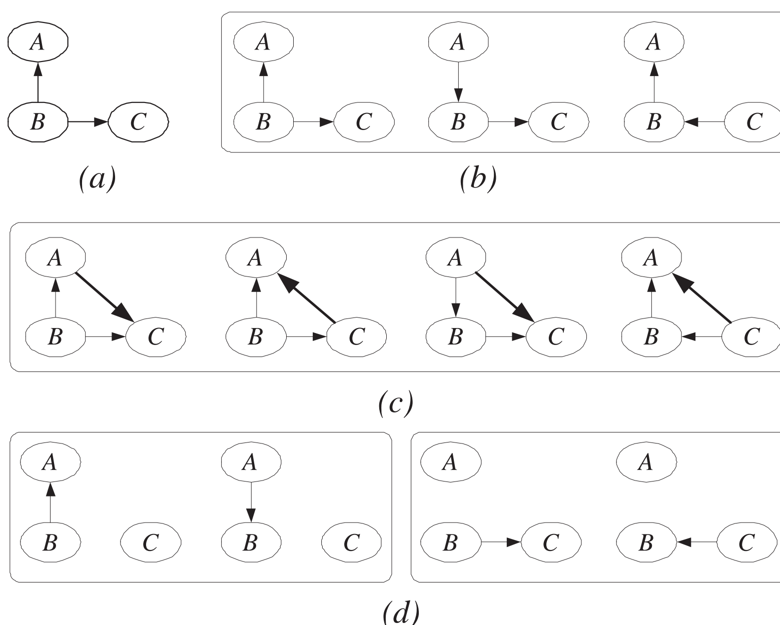

Figure 6: (a) DAG $\mathcal{G}$, (b) $\mathcal{E} = \mathcal{E}(\mathcal{G})$, (c) the single member of $\boldsymbol{\mathcal{E}}^+(\mathcal{E})$, and (d) the two members of $\boldsymbol{\mathcal{E}}^-(\mathcal{E})$.

GES can now be described as follows. We first initialize the state of the search to be the equivalence class $\mathcal{E}$ corresponding to the (unique) DAG with no edges. That is, the first state of the search corresponds to all possible marginal and conditional independence constraints. In the first phase of the algorithm, we repeatedly replace $\mathcal{E}$ with the member of $\boldsymbol{\mathcal{E}}^+(\mathcal{E})$ that has the highest score, until no such replacement increases the score. Once a local maximum is reached, we move to the second phase of the algorithm and repeatedly replace $\mathcal{E}$ with the member of $\boldsymbol{\mathcal{E}}^-(\mathcal{E})$ that has the highest score. Once the algorithm reaches a local maximum in the second phase, it terminates with its solution equal to the current state $\mathcal{E}$.

We prove that GES correctly identifies the optimal solution in the limit using two steps. First, we show that the local maximum reached in the first phase of the algorithm contains the generative distribution. Then, we use Theorem 4 to show that the equivalence class reached at the end of the second phase must be a perfect map of the generative distribution.

The proof for the first phase of GES relies on the fact that the generative distribution is DAG-perfect. Any such distribution must obey the *composition* independence axiom described in Pearl (1988), the contrapositive of which can be stated as follows: if variable X is not independent of the set $\mathbf{Y}$ given set $\mathbf{Z}$, then there exists a singleton element $Y \in \mathbf{Y}$ such that X is not independent of Y given set $\mathbf{Z}$.

Lemma 9 *Let $\mathcal{E}$ denote the equivalence class that results at the end of the first phase of GES, let $p(\cdot)$ denote the distribution from which the data $\mathbf{D}$ was generated, and let m denote the number of records in $\mathbf{D}$. Then in the limit of large m, $\mathcal{E}$ contains p.*

Proof: Suppose not, and consider any $\mathcal{G} \in \mathcal{E}$. Because $\mathcal{G}$ contains some independence constraint not in p, and because the independence constraints of $\mathcal{G}$ are characterized by the Markov conditions, there must exist some node X_i in $\mathcal{G}$ for which $X_i \not\perp\!\!\!\perp_p \mathbf{Y}|\mathbf{Pa}_i$, where $\mathbf{Y}$ is the set of non-descendants of X_i. Furthermore, because the composition axiom holds for $p(\cdot)$, there must exist at least one singleton non-descendant $Y \in \mathbf{Y}$ for which this dependence holds. By Lemma 7, this implies that the DAG $\mathcal{G}'$ that results from adding the edge $Y \rightarrow X_i$ to $\mathcal{G}$ (which cannot be cyclic by definition of $\mathbf{Y}$) has a higher score than $\mathcal{G}$. Clearly, $\mathcal{E}(\mathcal{G}') \in \boldsymbol{\mathcal{E}}^+(\mathcal{E})$, which contradicts the fact that $\mathcal{E}$ is a local maximum. $\square$

We now use Theorem 4 to show that in the second phase, GES will add independence constraints (by "deleting edges") until the equivalence class corresponding to the generative distribution is reached.

Lemma 10 *Let $\mathcal{E}$ denote the equivalence class that results from GES, let $p(\cdot)$ denote the DAG-perfect distribution from which the data $\mathbf{D}$ was generated, and let m denote the number of records in $\mathbf{D}$. Then in the limit of large m, $\mathcal{E}$ is a perfect map of p.*

Proof: Given Lemma 9, we know that when the second phase of the algorithm is about to commence, the current state of the search algorithm contains p. We are guaranteed that $\mathcal{E}$ will continue to contain p throughout the remainder of the algorithm by the following argument. Consider the first move made by GES to a state that does not contain p. By definition of $\boldsymbol{\mathcal{E}}^-(\mathcal{E})$, this move corresponds to an edge deletion in some DAG. But it follows immediately from the fact that the score is consistent that any such deletion would *decrease* the score, contradicting the fact that GES is greedy.

To complete the proof, assume that the algorithm terminates with some sub-optimal equivalence class $\mathcal{E}$, and let $\mathcal{E}^*$ be the optimal equivalence class. From Proposition 8, we know that $\mathcal{E}^*$ is a perfect map of p, and because $\mathcal{E}$ contains p, it follows that $\mathcal{E}$ must be an independence map of $\mathcal{E}^*$. Let $\mathcal{H}$ be any DAG in $\mathcal{E}$, and let $\mathcal{G}$ be any DAG in $\mathcal{E}^*$. Because $\mathcal{G} \leq \mathcal{H}$, we know from Theorem 4 that there exists a sequence of covered edge reversals and edge additions that transforms $\mathcal{G}$ into $\mathcal{H}$. There must be at least one edge addition in the sequence because by assumption $\mathcal{E} \neq \mathcal{E}^*$ and hence $\mathcal{G} \not\approx \mathcal{H}$. Consider the DAG $\mathcal{G}'$ that precedes the *last* edge addition in the sequence. Clearly $\mathcal{E}(\mathcal{G}') \in \boldsymbol{\mathcal{E}}^-(\mathcal{E})$ and because $\mathcal{G}'$ has fewer parameters than $\mathcal{H}$, we conclude from the consistency of the scoring criterion that $\mathcal{E}$ cannot be a local maximum, yielding a contradiction. $\square$

4.3 Discussion

In this section, we discuss some subtle issues about the GES algorithm, and consider what happens when some of our assumptions are violated.

Note that the first phase of GES does not depend on Theorem 4. In fact, the first phase is not even needed to get the large-sample optimality. It is used only to identify an equivalence class that contains the generative distribution, so we could simply start with the *complete* equivalence class (i.e., no independence constraints) and move immediately to the second phase. The problem, of course, with starting from the complete model is that for any realistic domain, the number of parameters in the model will be prohibitively large. The hope is that the first phase will identify a model that is as simple as possible. There exist generative distributions (e.g., the distribution with no independence constraints) for which the first phase will, in fact, have to reach the complete model in order to identify an appropriate equivalence class, but we hope that in practice the first phase will reach a local maximum that is reasonably sparse. In Section 6, we will see that for many real-world domains, this is exactly what happens.

The optimality proofs in the previous section do not depend on the scoring criterion being the Bayesian criterion. Lemma 9 (the first phase of GES) holds for any scoring criterion that is locally consistent, which means that the result holds for any consistent criterion that is decomposable in the limit (recall from Section 4.1 that we used consistency and decomposability to get local consistency). The proof of Proposition 8 (the optimal structure is perfect with respect to the generative distribution) and the proof of Lemma 10 (the second phase of GES) used Theorem 4 to compare the score of two equivalence classes by comparing the scores of two *particular* DAGs in those equivalence classes. For any score-equivalent criterion (such as the Bayesian criterion), this approach is clearly justified. Score equivalence is not needed, however, for the large-sample optimality of GES. In particular, as long as all DAGs in an equivalence class have the same number of parameters—a property that is easy to show for the models we consider in this paper (i.e., those containing Gaussian or multinomial distributions)—these proofs remain valid for any consistent criterion. To see this, we consider the following result:

Proposition 11 *Let $\mathcal{G}$ and $\mathcal{H}$ be any two DAGs that contain the generative distribution and for which $\mathcal{G}$ has fewer parameters than $\mathcal{H}$, and let S be any consistent (DAG) scoring criterion. If all DAGs in an equivalence class have the same number of parameters, then for every $\mathcal{G}' \approx \mathcal{G}$ and for every $\mathcal{H}' \approx \mathcal{H}$, $S(\mathcal{G}', \mathbf{D}) > S(\mathcal{H}', \mathbf{D})$.*

Proof: Because $\mathcal{G}'$ and $\mathcal{H}'$ both contain the generative distribution, and given that all DAGs in an equivalence class have the same number of parameters, the result follows immediately from the definition of consistency. $\square$

Given Proposition 11, it is easy to see that the proofs for Proposition 8 and Lemma 10 hold without modification for any consistent scoring criterion, regardless of whether or not the criterion is score equivalent. Things get a bit tricky when the scoring criterion is not score equivalent, however, because if we are interested in the highest-scoring DAG model, we may still have work to do after identifying the optimal equivalence class. In particular, there can be an enormous number of DAGs contained within an equivalence class, and we must search through these DAGs to find the best model. Depending on the particulars of the scoring criterion, this search problem may or may not be difficult.

An example of a popular scoring criterion that is not score equivalent is the (Bayesian) K2 scoring criterion. Cooper and Herskovitz (1992) derive this closed-form criterion for multinomial conditional distributions by making some assumptions about network parameter priors. It turns out that for two DAGs $\mathcal{G}_1$ and $\mathcal{G}_2$ that are in the same equivalence class, we can get different values for the marginal-likelihood terms $p(D|\mathcal{G}_1^h)$ and $p(D|\mathcal{G}_2^h)$ in the K2 criterion. Strictly speaking, this means that the hypothesis corresponding to a DAG in the K2 score cannot be simply a hypothesis about independence constraints. In fact, the reason that the K2 scoring criterion is not score equivalent is that Cooper and Herskovits (1992) constrain the conditional-parameter priors in the DAGs to come from a particular restricted family of distributions. Researchers often use K2 because it is easy to implement and is very fast to evaluate. Furthermore, the score differences between members within the same equivalence class are typically very small compared to the score differences between members of different equivalence classes. As a result, researchers often use the criterion to identify a good DAG, and then interpret the result to mean that the algorithm identified the equivalence class corresponding to that DAG.

As opposed to the “accidental” non-score-equivalence of K2, Heckerman, Geiger, and Chickering (1995) discuss a Bayesian scoring criterion for learning *causal* networks. In this case, they define the hypothesis corresponding to a DAG model to assert, in addition to the independence properties about the generative distribution, that each edge in the DAG corresponds to a cause-effect relationship. It turns out that the resulting scoring criterion is locally consistent, and thus—as described above—we can use GES to identify a single equivalence class of models in which we can then search for a high-scoring (causal) model.

For most real domains, it is unlikely that the generative distribution will be DAG perfect in the sense that there is a DAG *defined over the observables* that is perfect. In this case, we need to refine our definition of the hypothesis corresponding to a DAG because otherwise we are admitting that *none* of these hypotheses are true. We can relax the hypothesis $\mathcal{G}^h$ to denote, for example, the assertion that $\mathcal{G}$ is a DAG model with fewest parameters that can represent the joint distribution over the observables.[5] If we make the assumption that there exists a DAG defined over *some* set of variables that is a perfect map of the generative distribution—of which the observables are a subset—then the composition axiom still holds so we are guaranteed (in the limit) to identify an independence map of the optimal hypothesis in the first phase of GES. All we know about the second phase in this case, however, is that the resulting equivalence class will be a *minimal* independence map of the optimal solution. That is, there is no DAG in the class for which we can remove an edge and still contain the generative distribution. In Section 6, we explore the potential problems with the DAG-perfect assumption by applying the GES algorithm to real-world data. As we show in that section, the GES algorithm performs well in these domains, regardless of whether the large-sample guarantees are justified.

5. An Efficient Search Space

In the previous section, we provided a theoretical justification for using the GES algorithm by proving that in the limit of large datasets, the algorithm will identify the optimal model.

5. A technical difficulty with this definition is that two non-equivalent DAGs might both satisfy these conditions, and thus the hypotheses are not mutually exclusive.

Such a result is of little importance unless the search algorithm can be implemented in a reasonably efficient manner. To make this point clear, consider the (provably optimal) search algorithm that exhaustively enumerates and evaluates every possible structure; because the number of DAGs grows super-exponentially with the number of variables in the domain, and recent results from (e.g.) Gillispie and Perlman (2001) suggest that the number of equivalence classes grows super-exponentially as well, such an algorithm is of no practical importance except for very small domains (for only eight variables, there are over 700 billion DAGs and over 200 billion equivalence classes).

The feasibility of applying any search algorithm in practice depends on the complexity of both the algorithm and the search space to which that algorithm is applied. Because we are using a greedy search algorithm over edges, it is easy to show that the total number of search states visited by GES in a domain of n variables can never exceed $n \cdot (n-1)$. Furthermore, we have found that in practice the number of states visited generally grows linearly with n.

Of greater concern to us—given the simplicity of the algorithm—is the complexity of the search space: for each state visited by the greedy search algorithm, we need to generate and evaluate all states that are reachable by the application of a single operator. If the number of such neighbor states grows very large, or if each neighbor state takes too long to evaluate, even the simple greedy algorithm may not terminate quickly enough. Chickering (1996) shows that the problem of learning the optimal structure using the Bayesian scoring criterion is NP-hard; this negative result suggests that in the worst case, the connectivity of the search space that the algorithm encounters will be a problem. Our hope is that in practice, this worst-case scenario will not occur, and that for real-world problems the portion of the search space traversed by GES will be sparse. If we do, in fact, encounter portions of the search space that are too dense to search efficiently, we can choose to consider only a heuristically-selected subset of the candidate neighbors at each step, albeit at the cost of losing the large-sample optimality guarantee. We should point out that the density of the search space has never been a problem in any of the experiments we have performed, including those presented in Section 6.

In this section, we describe a method for efficiently generating and evaluating the neighbors of a given search state in the GES algorithm. The approach we take builds upon the work of Chickering (2002), where completed PDAGs (described in Section 2.4) are used to represent states in the search, and where operators are defined that can be used (by any algorithm) to search the space of equivalence classes efficiently.

We now define a *search space* corresponding to each of the two phases of the GES algorithm presented in Section 4.2. A search space has three components:

1. A set of states
2. A representation scheme for the states
3. A set of operators

The set of states represents the logical set of solutions to the search problem, the representation scheme defines an efficient way to represent the states, and the set of operators is used by the search algorithm to transform the representation of one state to another in

order to traverse the space in a systematic way. The two phases of GES correspond to a greedy search algorithm applied to two different search spaces that differ by the set of operators they contain.

In Section 4.2, both the states of GES and the connectivity of the search space in the two phases are defined. In particular, the states of the search are equivalence classes of DAGs, and the neighbors of a particular state $\mathcal{E}$ are either $\boldsymbol{\mathcal{E}}^+(\mathcal{E})$ or $\boldsymbol{\mathcal{E}}^-(\mathcal{E})$, depending on whether GES is in the first or second phase, respectively. Furthermore, we will use completed PDAGs—described in Section 2.4—to represent the states of the search. Thus all that remains to defining the search space is an implementation of the operators.

Given a state of the search represented as a completed PDAG $\mathcal{P}^c$, we define the following two sets of operators that can be used to define the connectivity of the two phases of GES. In these definitions and elsewhere, a pair of nodes X and Y in a PDAG are *neighbors* if they are connected by an undirected edge, and they are *adjacent* if they are connected by either an undirected edge or a directed edge.

Definition 12 $Insert(X, Y, \mathbf{T})$
For non-adjacent nodes X and Y in $\mathcal{P}^c$, and for any subset $\mathbf{T}$ of the neighbors of Y that are not adjacent to X, the $Insert(X, Y, \mathbf{T})$ operator modifies $\mathcal{P}^c$ by (1) inserting the directed edge $X \rightarrow Y$, and (2) for each $T \in \mathbf{T}$, directing the previously undirected edge between T and Y as $T \rightarrow Y$.

Definition 13 $Delete(X, Y, \mathbf{H})$
For adjacent nodes X and Y in $\mathcal{P}^c$ connected either as $X - Y$ or $X \rightarrow Y$, and for any subset $\mathbf{H}$ of the neighbors of Y that are adjacent to X, the $Delete(X, Y, \mathbf{H})$ operator modifies $\mathcal{P}^c$ by deleting the edge between X and Y, and for each $H \in \mathbf{H}$, (1) directing the previously undirected edge between Y and H as $Y \rightarrow H$ and (2) directing any previously undirected edge between X and H as $X \rightarrow H$.

We use *Insert* operators to implement the connectivity for the first phase of GES, and we use *Delete* operators to implement the connectivity for the second phase of GES. We use '**T**' to denote the set-argument of the *Insert* operator because every node in this set becomes a "tail" node in a new v-structure as a result of the operator. Similarly, we use '**H**' for the *Delete* operator because every node in this set becomes a "head" node in a new v-structure.

After applying an operator to a completed PDAG, the resulting PDAG is not necessarily completed. Therefore we may need to convert that PDAG to the corresponding completed PDAG representation of the resulting equivalence class; this is accomplished in two steps by first extracting a consistent extension from the (not completed) PDAG, and then constructing the completed PDAG from that DAG. In Appendix C, we provide the implementation of Chickering (2002) for both steps of this conversion algorithm. If the (not completed) PDAG that results from an operator admits a consistent extension, we say that the operator is *valid*. Otherwise, we say the operator is *not valid* and do not allow its application to the search space.

The algorithm in Appendix C that converts PDAGs to completed PDAGs takes time $O(|E| \cdot k^2)$ in the worst case—where $|E|$ is the number of edges in the PDAG and k is the maximum number of parents per node—which could potentially be a problem for domains

with a large number of variables. As we show below, however, in GES all of the operators for a given search state can be generated and evaluated efficiently without ever needing to construct the representation of the resulting states. Thus the only time that the completed PDAG representation for a state needs to be constructed is when GES "moves" to that state (i.e., when the best neighbor state is identified and the current state is replaced with that neighbor state). Furthermore, the algorithm does not depend on the number of records in the data, and because it is applied infrequently compared to the number of times operators are evaluated, its contribution to the overall run time of GES is insignificant.

There are easily testable conditions for both *Insert* and *Delete* operators to ensure that they are valid. To define these conditions, we first need to define a *semi-directed* path. This is the same as a directed path except that any of the edges may be undirected. More formally we have:

Definition 14 *A* semi-directed path *from Y to X in a PDAG is a path from Y to X such that each edge is either undirected or directed away from Y.*

The following two theorems and corresponding corollaries demonstrate (1) how to determine efficiently whether or not an *Insert* or *Delete* operator is valid and (2) how to score each such operator. We have simplified our notation to make the results easy to read: $\mathbf{Pa}_Y$ denotes the parents of node Y in the completed PDAG representation of the current state. We use $\mathbf{Pa}_Y{}^{+X}$ and $\mathbf{Pa}_Y{}^{-X}$ as shorthand for $\mathbf{Pa}_Y \cup \{X\}$ and $\mathbf{Pa}_Y \setminus \{X\}$, respectively. We use $\mathbf{NA}_{Y,X}$ to denote the set of nodes that are neighbors of node Y and are adjacent to node X in the current state. The proofs of these results, which are summarized in Table 1, are given in Appendix B.

Theorem 15 *Let $\mathcal{P}^c$ be any completed PDAG, and let $\mathcal{P}^{c\prime}$ denote the result of applying an $Insert(X, Y, \mathbf{T})$ operator to $\mathcal{P}^c$. There exists a consistent extension $\mathcal{G}$ of $\mathcal{P}^c$ to which adding the edge $X \rightarrow Y$ results in a consistent extension $\mathcal{G}'$ of $\mathcal{P}^{c\prime}$ if and only if in $\mathcal{P}^c$*

1. *$\mathbf{NA}_{Y,X} \cup \mathbf{T}$ is a clique*
2. *Every semi-directed path from Y to X contains a node in $\mathbf{NA}_{Y,X} \cup \mathbf{T}$*

Corollary 16 *For any score-equivalent decomposable scoring criterion, the increase in score that results from applying a valid operator $Insert(X, Y, \mathbf{T})$ to a completed PDAG $\mathcal{P}^c$ is*

$$s(Y, \mathbf{NA}_{Y,X} \cup \mathbf{T} \cup \mathbf{Pa}_Y{}^{+X}) - s(Y, \mathbf{NA}_{Y,X} \cup \mathbf{T} \cup \mathbf{Pa}_Y)$$

Theorem 17 *Let $\mathcal{P}^c$ be any completed PDAG that contains either $X \rightarrow Y$ or $X - Y$, and let $\mathcal{P}^{c\prime}$ denote the result of applying the operator $Delete(X, Y, \mathbf{H})$ to $\mathcal{P}^c$. There exists a consistent extension $\mathcal{G}$ of $\mathcal{P}^c$ that contains the edge $X \rightarrow Y$ from which deleting the edge $X \rightarrow Y$ results in a consistent extension $\mathcal{G}'$ of $\mathcal{P}^{c\prime}$ if and only if $\mathbf{NA}_{Y,X} \setminus \mathbf{H}$ is a clique.*

Corollary 18 *For any score-equivalent decomposable scoring criterion, the increase in score that results from applying a valid operator $Delete(X, Y, \mathbf{H})$ to a completed PDAG $\mathcal{P}^c$ is*

$$s(Y, \{\mathbf{NA}_{Y,X} \setminus \mathbf{H}\} \cup \mathbf{Pa}_Y{}^{-X}) - s(Y, \{\mathbf{NA}_{Y,X} \setminus \mathbf{H}\} \cup \mathbf{Pa}_Y)$$

Table 1: Necessary and sufficient validity conditions and (local) change in score for each operator

Operator	Validity Tests	Change in Score
$Insert(X, Y, \mathbf{T})$	$\mathbf{NA}_{Y,X} \cup \mathbf{T}$ is a clique Every semi-directed path from Y to X contains a node in $\mathbf{NA}_{Y,X} \cup \mathbf{T}$	$s(Y, \mathbf{NA}_{Y,X} \cup \mathbf{T} \cup \mathbf{Pa}_Y{}^{+X})$ $-\, s(Y, \mathbf{NA}_{Y,X} \cup \mathbf{T} \cup \mathbf{Pa}_Y)$
$Delete(X, Y, \mathbf{H})$	$\mathbf{NA}_{Y,X} \setminus \mathbf{H}$ is a clique	$s(Y, \{\mathbf{NA}_{Y,X} \setminus \mathbf{H}\} \cup \mathbf{Pa}_Y{}^{-X})$ $-\, s(Y, \{\mathbf{NA}_{Y,X} \setminus \mathbf{H}\} \cup \mathbf{Pa}_Y)$

The final step in an implementation of GES is a method to generate candidate operators after each move. We note that the majority of the operators at a given step of the algorithm both will remain valid and will have the same score at the next step of the algorithm. Given that we need to generate or re-generate a set of operators corresponding to a pair of nodes X and Y, the most obvious approach is to use Definition 12 and Definition 13 directly to generate those operators without regard to the validity conditions, and then test the validity conditions for every one. This procedure is detailed in the following paragraph.

In the first phase of GES, only those nodes that are not adjacent will have a corresponding set of operators. For such pair X and Y whose corresponding operators need to be generated, we define $\mathbf{T_0}$ to be the set of all neighbors of Y that *are not* adjacent to X. Let $\mathbf{T_0}*$ denote the power set of $\mathbf{T_0}$; that is, $\mathbf{T_0}*$ contains all possible subsets of $\mathbf{T_0}$. We then test the validity of (and possibly score) the result of $Insert(X, Y, \mathbf{T})$ for every $\mathbf{T} \in \mathbf{T_0}*$. In the second phase of GES, only those nodes that *are* adjacent will have a corresponding set of operators. For such a pair X and Y whose corresponding operators need to be generated—and for which there is either an undirected edge between X and Y or a directed edge from X to Y—we define $\mathbf{H_0}$ to be the set of all neighbors of Y that *are* adjacent to X. Let $\mathbf{H_0}*$ denote the power set of $\mathbf{H_0}$. We then test the validity of (and possibly score) the result of $Delete(X, Y, \mathbf{H})$ for every $\mathbf{H} \in \mathbf{H_0}*$.

For a set of nodes $\mathbf{S}$ of size k, there are 2^k elements in the power set of $\mathbf{S}$. It follows that the feasibility of this implementation for GES will, to a large degree, depend on the number of neighbors of the nodes in the completed PDAGs we encounter; if there is a node with too many neighbors, we may simply have too many operators to test. In particular, during the first phase of the algorithm, in order to generate the operators for a pair of non-adjacent nodes X and Y, the implementation can be slow if Y has many neighbors that are not adjacent to X. Similarly, during the second phase of the algorithm, the implementation may be slow for (adjacent) X and Y if Y has many neighbors that *are* adjacent to X.

There are a number of tricks we can apply to generate more efficiently the candidate operators corresponding to a pair of nodes. Consider the first validity condition for the *Insert* operator given in Table 1: namely, that the set $\mathbf{NA}_{Y,X} \cup \mathbf{T}$ must be a clique. If this test fails for some set $\mathbf{T}$, then it will also fail for any $\mathbf{T}'$ that contains $\mathbf{T}$. Thus if we are careful, we can gain enormous savings by not generating candidates that we know are not valid. A similar optimization can be made for the *Delete* operator, except that we save only the cost of performing the validity test. In particular, if the validity test for the *Delete* operator passes for some set $\mathbf{H}$, then we know it will also pass for any set $\mathbf{H}'$ that contains $\mathbf{H}$ as a subset. We can also save time by noting that if the *second* validity condition for the *Insert* operator passes for some $\mathbf{T}$, then it will also pass for any $\mathbf{T}'$ that contains $\mathbf{T}$. Finally, we note that if we are careful, we can avoid generating distinct operators that result in the same neighbor state. For example, $Delete(X, Y, \mathbf{H})$ and $Delete(Y, X, \mathbf{H})$ result in the same state,[6] so only one of them need be generated. A similar result for the *Insert* operator when the set $\mathbf{T}$ is empty is given by Chickering (2002): if X and Y have the same parents, then $Insert(X, Y, \emptyset)$ and $Insert(Y, X, \emptyset)$ result in the same state.

Unfortunately, in the worst case there can be an exponential number of valid operators for a particular state in the search. As was mentioned above, we can prune neighbors heuristically in this situation to make the search practical. For example, we might choose to search only through equivalence classes where the member DAGs have some upper bound k on the number of parents for each node. In this case, we need consider only a polynomial number of "v-structure sets" for each pair of nodes. In all of the experiments we have performed, however, including those presented in the next section, we have yet to encounter a domain for which GES encounters a state that has too many neighbors.

As is evident from the simplicity of the validity conditions from Table 1, there are a number of ways to efficiently update (i.e., regenerate) the valid operators after each step of GES. For example, consider the set of *Insert* operators corresponding to the nodes X and Y. Suppose that all the operators have been generated and scored at a given step of (the first phase of) GES, and we want to know whether these operators remain valid and have the same score after applying some operator. From Table 1, we see that if the neighbors of Y have not changed, the first validity condition must still hold for all previously-valid operators; because we are adding edges in this phase, any clique must remain a clique. Furthermore, if the parents of node Y have not changed, we need only check the second validity condition (assuming the first holds) if the score of the operator is higher than the best score seen so far; otherwise, we know that regardless of whether the operator is valid or not, it will not be chosen in the next step.

Finally, we note that an obvious optimization that we use for both GES and the alternative search algorithms described in the next section is to cache away previously-computed local scores corresponding to a node. Thus when we transition to the second phase of GES, many of the operators can be scored without an explicit call to the scoring function.

6. These operators are only both defined if the edge between X and Y is undirected; note that the definition of $Delete(X, Y, \mathbf{H})$ is not symmetric in X and Y.

6. Experimental Results

In this section, we evaluate the GES algorithm using both synthetic and real-world data. In Section 6.1, we use synthetic data to evaluate GES in terms of how well the algorithm can identify the generative structure given datasets that are finite. In Section 6.2, we use real-world data to evaluate the solution quality and total search time of GES when it is applied to real data.

In all of our experiments, we compare GES to two alternative greedy search algorithms. The first such algorithm, which we call *D-space search*, is a traditional DAG-space greedy algorithm that considers adding, removing, and reversing edges at each step. The second search algorithm, which we call *E-space search*, is a greedy search through equivalence classes using the following operators defined by Chickering (2002): (1) all valid $Insert$ operators for which $\mathbf{T}$ is empty (no v-structures are created that contain previously undirected edges), (2) all valid $Delete$ operators where the set $\mathbf{H}$ is empty, (3) a directed edge can be reversed if the result is a PDAG that admits a consistent extension and (4) for any length-two path of undirected edges $X - Y - Z$, if X and Z are not adjacent, then the edges can be directed as $X \rightarrow Y \leftarrow Z$ if the result is a PDAG that admits a consistent extension. As shown by Chickering (2002), all of these operators can be tested and scored efficiently.

We use the Bayesian *BDeu* scoring criterion for discrete variables—derived by Heckerman et al. (1995)—in all of our experiments. The BDeu criterion uses a parameter prior that has uniform means, and requires both a prior equivalence sample size and a structure prior to specify. For all of our experiments, we use a prior equivalent sample size of ten, and a structure prior of 0.001^f, where f is the number of free parameters in the DAG. Let q_i denote the number of configurations of the parent set $\mathbf{Pa}_i$, and let r_i denote the number of states of variable X_i. Then the version of the BDeu criterion used in our experiments is:

$$S_{BDeu}(\mathcal{G}, \mathbf{D}) = \log \prod_{i=1}^{n} 0.001^{(r_i-1)q_i} \prod_{j=1}^{q_i} \frac{\Gamma(\frac{10}{q_i})}{\Gamma(\frac{10}{q_i} + N_{ij})} \cdot \prod_{k=1}^{r_i} \frac{\Gamma(\frac{10}{r_i \cdot q_i} + N_{ijk})}{\Gamma(\frac{10}{r_i \cdot q_i})} \tag{5}$$

where N_{ijk} is the number of records in $\mathbf{D}$ for which $X_i = k$ and $\mathbf{Pa}_i$ is in the jth configuration, and $N_{ij} = \sum_k N_{ijk}$. We also use the (non-Bayesian) constraint that each parameter needs to have a corresponding sample size of at least five. Note from Equation 5 that our scoring criterion is decomposable.

6.1 Synthetic-Data Experiments

In our experiments with synthetic data, we generated datasets of various sample sizes from a *gold standard* Bayesian network with known structure and parameters. In order to make the connectivity of the gold standard "realistic", we constructed each generative network as follows. First, we took a real-world dataset (the MediaMetrix dataset described in detail in Section 6.2), consisting of roughly 5000 records in a domain of 13 discrete (three-valued) variables, and ran the D-space search algorithm to identify a local maximum. Then, we performed ten random D-space edge operations (i.e., additions, deletions, and reversals) to that local maximum, and the resulting structure defined the edges in our gold standard. Finally, we parameterized the gold standard by sampling all of the conditional (multinomial) parameters from a uniform Dirichlet distribution.

The synthetic data experiments can be described as follows. We generated 100 random gold standards as described above, and considered sample sizes from 500 to 10000 in increments of 500 samples. For each sample size, we created a dataset with the appropriate number of records from each of the 100 gold standards. For each sampled dataset, we learned three Bayesian networks using each of the three greedy search algorithms, and checked whether or not these networks were equivalent to the gold standard. Figure 7 contains the results of these experiments. The figure plots, for each of the three algorithms, the number of the learned networks that were equivalent to the gold standard as a function of the sample size.

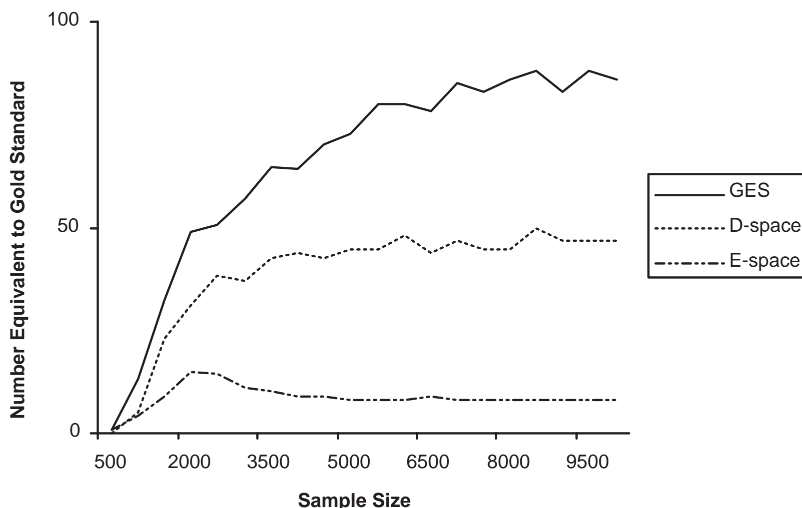

Figure 7: Number of learned networks that are equivalent to the generative structure as a function of the sample size.

As we see from the figure, GES proved to be superior to the competing algorithms when tasked with identifying the generative structure. Rather surprising is that the models identified using D-space were more often equivalent to the generative structure than those identified using E-space; one explanation for this is that by virtue of generating the gold standards using D-space, we may be biasing the experiment in favor of that space. To gauge the complexity of the domain, we recorded the number of edges, the number of parameters, and the maximum number of parents for each of the 100 gold-standard models. The averages of these measurements were 9.1 ($\pm$1.1), 98.0 ($\pm$36.1), and 2.4 ($\pm$0.7), respectively, which demonstrate that for this experiment, the optimal equivalence classes were sparse.

6.2 Real-World Data Experiments

We used the following six real-world datasets in our experiments. For all datasets, we assume that values are *not* missing at random. In particular, we treat "missing" as a distinct, discrete state.

1. **Microsoft Web Training Data (MSWeb)**

   This dataset, which is available via anonymous ftp from the UCI Machine Learning Repository, contains 32711 instances of users visiting the www.microsoft.com web site on a day in 1996. For each user, the data contains a variable indicating whether or not that user visited each of the 292 areas (i.e., "vroots") of the site. We used the 50 most popular areas and a sample of 5,000 users.

2. **Nielsen**

   The Nielsen dataset contains data about television-watching behavior during a two-week period in 1995. The data was made available courtesy of Nielsen Media Research. The data records whether or not each user watched five or more minutes of network TV shows aired during the given time period. There were 3314 users in the study, and 402 television shows. We used the most popular 50 shows in our experiments.

3. **EachMovie**

   The EachMovie dataset consists of viewer ratings on movies. The data was collected during an 18-month period beginning in 1995. We used the ratings of the 50 most popular movies by a sample of 5,000 viewers. The rating is a discrete variable that is either missing, or is provided as an integer from one to five.

4. **MediaMetrix**

   This dataset contains demographic and internet-use data for 4808 individuals during the month of January 1997. We used only the internet-use variables in our experiments; there are 13 such variables that indicate the category of web site visited.

5. **1984 United States Congressional Voting Records (HouseVotes)**

   This dataset contains the 1984 congressional voting records for 435 representatives voting on 17 issues, and is available via anonymous ftp from the UCI Machine Learning Repository. Votes are all three-valued: yes, no, or unknown. For each representative, the political party is given; this dataset is typically used in a classification setting to predict the political party of the representative based on the voting record.

6. **Mushroom**

   The Mushroom dataset, available via anonymous ftp from the UCI Machine Learning Repository, contains physical characteristics of 8124 mushrooms, as well as whether each mushroom is poisonous or edible. There are 22 physical characteristics for each mushroom, all of which are discrete.

For our experiments, we considered some variants of the GES algorithm that we deemed to be better suited for real-world domains. Our inclusion of these variants was motivated

by a number of observations. First, we found that after running phase two of GES, it was often the case that we could further increase the score by applying more *Insert* operators; in other words, the state reached after phase two was not a local maximum with respect to the phase-one operators. Second, we found that in the first phase of GES, the best *Delete* operator would often have a better score than the best *Insert* operator, even though the best *Insert* operator increased the score; note that this situation is impossible in the large-sample limit. Finally, we noticed that in practice, the number of v-structures induced by the sets $\mathbf{T}$ and $\mathbf{H}$ for the best *Insert* and *Delete* operators, respectively, was in almost all cases either zero or one. Thus we can often restrict the size of $\mathbf{T}$ and $\mathbf{H}$ to size one and get the same local maximum as we would with no such restrictions. As discussed in Section 5, such a restriction reduces the number of operators we need to evaluate and thus will speed up the implementation.

We ran experiments using three specific variants of GES. The first variant, which we call GES*, simply applies GES repeatedly until neither phase one nor phase two increases the score. We use GES* instead of GES in our experiments because it is guaranteed to find a solution that is at least as good (in terms of the score) as GES. The second variant, which we call OPS, performs a greedy search using *both* the *Insert* operators and the *Delete* operators at each step. Finally, the third variant, which we call OPS-1, is identical to OPS except that we only consider *Insert* and *Delete* operators for which $|\mathbf{T}| \leq 1$ and $|\mathbf{H}| \leq 1$, respectively.

The results of our experiments are given in Table 2 and Table 3. In Table 2 we report, for each dataset, the score of the maximum reached by each algorithm. In Table 3 we report, for each dataset, the total learning time in seconds for each algorithm.

Table 2: Scores of the model selected by each of the algorithms.

Dataset	GES*	OPS	OPS-1	D-space	E-space
MSWeb	-38599.7	-38599.7	-38599.7	-38602.0	-38618.4
Nielsen	-42787.8	-42787.8	-42787.8	-42800.3	-42916.4
EachMovie	-258531.0	-258531.0	-258531.0	-258531.0	-258531.0
MediaMetrix	-46341.3	-46341.3	-46341.3	-46369.8	-46341.3
HouseVotes	-6061.1	-6061.1	-6061.1	-6061.1	-6061.1
Mushroom	-177351	-177351	-177351	-177408	-177351

Table 3: Total learning time in seconds for each algorithm.

Dataset	GES* Time	OPS Time	OPS-1 Time	D-space Time	E-space Time
MSWeb	54	54	52	28	24
Nielsen	36	36	36	30	12
EachMovie	25	24	24	16	16
MediaMetrix	3	3	3	2	2
HouseVotes	0.5	0.5	0.5	0.3	0.3
Mushroom	14	14	13	5	4

Rather surprising, we see from Table 3 that all of the algorithms performed about the same in terms of the resulting score. Although the GES variants always identified a model that had the same score or better than the two competing approaches, we do not believe the differences are significant.[7] Upon closer examination of the models, we found some interesting properties. For HouseVotes and EachMovie, all of the algorithms resulted in the same local maximum, and this model contained no compelled edges. For MediaMetrix and Mushroom, all of the algorithms except for D-space resulted in the same local maximum, and this model contained no compelled edges. For all datasets, the GES variants traversed the same set of states and resulted in the same local maximum. All of the models learned were reasonably sparse.

Because the local maxima from the experiments can be identified without applying many (if any) operators that create v-structures, all algorithms essentially traversed over the same set of states. We expect that in domains for which there are more complicated dependencies, the GES-based algorithms will identify different models both from themselves and the two competing algorithms. Given the results in Section 6.1, we also have reason to hope that these algorithms will identify better models. From Table 3 we see that the running times of the GES variants are generally larger than the running times of the two alternative algorithms. To investigate the source of the increase in time, we recorded the number of times that the local scoring function was called by each of the algorithms. In Table 4 we report the total learning time—in *milliseconds*—divided by the number of times the evaluation function was called; as we see, for each of the datasets, this time is roughly constant across all of the algorithms.

Table 4: Total learning time in milliseconds divided by the number of calls to the evaluation function for each algorithm.

Dataset	GES* Time	OPS Time	OPS-1 Time	D-space Time	E-space Time
MSWeb	5.01	5.01	4.99	5.27	5.40
Nielsen	4.72	4.77	4.78	4.88	7.31
EachMovie	10.01	9.59	9.73	9.56	9.56
MediaMetrix	5.96	5.96	6.04	6.45	6.40
HouseVotes	0.73	0.70	0.76	1.06	0.95
Mushroom	12.44	12.50	12.29	13.00	12.25

Because we cache the local scores of the nodes during all searches, each operator re-score (due to a change in the local connectivity of the state) requires on average a single call to the scoring function. Therefore the times in Table 4 are roughly equal to the time spent per operator re-score. Because this time is constant, we conclude that the increase in time is due entirely to the additional operators that we need to score (and re-score) at each step. Furthermore, we conclude that our validity tests for the *Insert* and *Delete* operators

7. Chickering (2002) compared D-space to E-space using a 70% sample of the full datasets, and the winning algorithm was different than our results for three of them.

are efficient enough that the time to traverse the search space using the GES variants is dominated by the time spent scoring the operators.

7. Conclusion

In this paper, we proved the so-called "Meek Conjecture" and showed how the result leads to the asymptotically optimal two-phase greedy search algorithm GES that was originally proposed by Meek (1997). We provided a new implementation of the search space to which GES can be applied such that all operators used by the algorithm can be scored efficiently using local functions of the nodes in the domain. Using synthetic data, we demonstrated that (1) the GES algorithm can identify the generative structure when given enough data and (2) the GES algorithm is superior in this regard to a greedy search using two alternative search spaces. We applied GES to six real-world datasets and saw that the solution quality was roughly the same as the two alternative greedy approaches. Although the time per evaluation-function call was the same as the competing algorithms, we found that the larger number of neighbors per state for the GES algorithm resulted in slightly slower run times.

An interesting extension to this work would be to investigate whether or not there are any large-sample optimality guarantees to GES (or a variant of GES) when the generative structure is not a DAG defined over the observables. As discussed in Section 4.3, if the generative structure is a DAG that includes hidden variables, the composition axiom of independence still holds among the observables, and the first phase of GES will lead to an independence map of the optimal model. We know that result of the *second* phase of the algorithm is a minimal such independence map, but can we say anything stronger? In a recent paper, Chickering and Meek (2002) consider situations when the composition axiom is guaranteed to hold, and investigate the optimality guarantees of GES in these situations.

It is unfortunate that the real-world dataset experiments did not provide a good test bed for our algorithms. It does suggest, however, the following search strategy to apply when faced with real data: First run a simple (and fast) DAG-based greedy algorithm. If the resulting model is simple (e.g., there are no compelled edges and there are only a few edges), we probably will not be able to find a better solution with a more sophisticated algorithm. If the model is reasonably complicated, on the other hand, we may try to apply GES or one of its variants.

Recall that the OPS algorithm from Section 6.2 considers both the *Insert* and *Delete* operators simultaneously. An interesting extension would be to implement an algorithm that considers, in addition to these operators, the "extra" operators from the E-space algorithm of Chickering (2002) that connect states that are not adjacent in the OPS space; these operators are the edge-reversal operator and the operator that makes a v-structure by directing two undirected edges. This extension would increase the number of evaluations that would need to be performed at each state, but perhaps the combined search algorithm would perform better.

Acknowledgments

Special thanks to Michael Perlman, who revived my interest in Meek's conjecture; I had long ago given up after many months in pursuit of a proof. My discussions with Michael

Perlman, Milan Studený, Tomáš Kočka, Robert Castelo and Steve Gillispie proved to be extremely useful, and I am grateful to them all. I would also like to thank Chris Meek—who initially introduced me to his conjecture in 1995—for the many helpful discussions on this work. Others who provided useful comments on earlier drafts include Remco Bouchaert, David Heckerman, Rich Neapolitan, and two anonymous reviewers.

Appendix A: Detailed Proof of Theorem 4

In this appendix, we provide a detailed proof of Theorem 4. The theorem is an immediate consequence of Lemma 30, which demonstrates the correctness of Algorithm Apply-Edge-Operation.

Almost all of the results presented here are proved using properties of the *d-separation* criterion. This criterion—which is detailed by (e.g.) Pearl (1988)—is used to test whether or not certain independence constraints are implied by a DAG model. In particular, two nodes A and B are said to be d-separated in a DAG $\mathcal{G}$ given a set of nodes $\mathbf{S}$ if and only if there is no *active path* in $\mathcal{G}$ between A and B given $\mathbf{S}$. The standard definition of an active path is a *simple* path for which each node W along the path either (1) has converging arrows and W or a descendant of W is in $\mathbf{S}$ or (2) does not have converging arrows and W is not in $\mathbf{S}$. By simple, we mean that the path never passes through the same node twice.

To simplify our proofs, we use an equivalent definition of an active path—that need not be simple—where each node W along the path either (1) has converging arrows and W is in $\mathbf{S}$ or (2) does not have converging arrows and W is not in $\mathbf{S}$. In other words, instead of allowing a segment $\rightarrow W \leftarrow$ to be included in a path by virtue of a descendant of W belonging to $\mathbf{S}$, we require that the path include the sequence of edges from W to that descendant and then back again. For those readers familiar with the celebrated "Bayes ball" algorithm of Shachter (1998) for testing d-separation, our expanded definition of an active path is simply a valid path that the ball can take between A and B.

More formally, we have the following definitions.

Definition 19 (Collider) *Let $\pi(W_1, W_n)$ denote any path between W_1 and W_n. A node W_i is called a* collider at position i *of the path if $W_i \notin \{W_1, W_n\}$ and the path contains the converging arrows $W_{i-1} \rightarrow W_i \leftarrow W_{i+1}$ at W_i.*

Definition 20 (Active Path) *A path $\pi(A, B)$ between A and B in DAG $\mathcal{G}$ is* $\mathbf{S}$-active *in $\mathcal{G}$ if the following conditions hold:*

1. *$A \notin \mathbf{S}$ and $B \notin \mathbf{S}$*
2. *If $W \in \mathbf{S}$ is an element of $\pi(A, B)$, then W is a collider at every position in $\pi(A, B)$*
3. *If $W \notin \mathbf{S}$ is an element of $\pi(A, B)$, then W is not a collider in any position in $\pi(A, B)$*

The direction of each *terminal* edge—that is, the first and last edge encountered in a traversal from one end of the path to the other—in an active path is important for determining whether we can append two active paths together to make a third active path. We say that a path $\pi(A, B)$ is *into* A if the terminal edge incident to A is oriented toward

A (i.e., $A \leftarrow$). Similarly, the path is into B if the terminal edge incident to B is oriented toward B. If a path is not into an endpoint A, we say that the path is *out of* A.

The following lemma demonstrates when we can create an active path by simply appending two other active paths together.

Lemma 21 *Let $\pi(A, B)$ be an* **S***-active path between A and B, and let $\pi(B, C)$ be an* **S***-active path between B and C. If either path is out of B, then the concatenation of $\pi(A, B)$ and $\pi(B, C)$ is an* **S***-active path between A and C.*

Proof: Because at least one of the paths is out of B, the junction between $\pi(A, B)$ and $\pi(B, C)$ cannot be a collider. Furthermore, because $B \notin \mathbf{S}$, the concatenation satisfies all of the conditions of Definition 20. □

For example, consider the DAG shown in Figure 8, and assume $\mathbf{S} = \{C\}$. If we let $\pi(A, D) = \{A \to B \to D\}$ and $\pi(D, E) = \{D \to C \leftarrow D \leftarrow E$, it follows from Lemma 21 that because (1) both paths are **S**-active and (2) $\pi(D, E)$ is out of D, the concatenation $\pi(A, E) = A \to B \to D \to C \leftarrow D \leftarrow E$ is **S**-active.

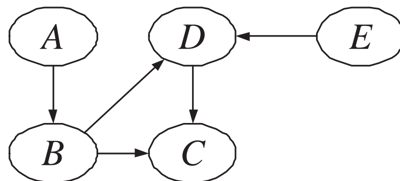

Figure 8: Example DAG with a **S**-active path between A and E, where $\mathbf{S} = \{C\}$.

In the proofs that follow, we will make extensive use of Lemma 21, but we will do so implicitly to simplify the presentation. In many of the results, for example, we prove the existence of an **S**-active path between two nodes A and B by showing that (1) there is an **S**-active path between A and some node X_1, (2) there is an **S**-active path between B and X_2, and (3) there is an **S**-active path between X_1 and X_2 that is out of both X_1 and X_2. To conclude from these properties that there is an **S**-active path between A and B, we need to make an awkward argument about applying Lemma 21 twice, whereas the conclusion is obvious given the lemma.

The following lemma and its two corollaries provide the main tools we use to prove that Algorithm Apply-Edge-Operation is correct. In particular, these results expose properties about active paths that must hold in light of an edge addition to some DAG.

Lemma 22 *Let $\mathcal{G}$ be any DAG, and let $\mathcal{G}'$ be the DAG that results by adding the edge $X \to Y$ to $\mathcal{G}$. Let $\pi(A, B)$ be any* **S***-active path in $\mathcal{G}'$ such that there is no S-active path between A and B in $\mathcal{G}$. Then the following properties hold:*

1. *$\pi(A, B)$ contains the edge $X \to Y$*
2. *$X \notin$* **S**
3. *If Y is an endpoint, then in $\mathcal{G}$ there is an active path between the other endpoint and X*

4. *If $Y \notin \mathbf{S}$, then in $\mathcal{G}$ either there are active paths between both endpoints and X, or there is an active path between one endpoint and X and an active path between the other endpoint and Y.*

5. *If $Y \in \mathbf{S}$, then in $\mathcal{G}$ either there are active paths between both endpoints and X, or there is an active path between one endpoint and X and an active path between the other endpoint and all other parents of Y that are not in $\mathbf{S}$.*

Proof: (1) follows immediately because otherwise the path is active in $\mathcal{G}$. Given (1), (2) follows because otherwise the path would not be active. (3) follows from (2) and the fact that every sub-path of an active path between nodes not in $\mathbf{S}$ is by definition active.

We now prove (4) and (5) by considering the following two traversals of $\pi(A, B)$: consider a traversal of $\pi(A, B)$ from A along $\pi(A, B)$ until the edge between X and Y is about to be traversed for the first time. Similarly, consider the same traversal except starting at B.

First we show that at least one of the traversals ends at X, and thus we establish for both (4) and (5) that there exists in $\mathcal{G}$ an active path between one of the endpoints and X. Suppose to the contrary that both traversals end at node Y. If $Y \in \mathbf{S}$, the last edge in both traversals must be into Y and thus we could append them together to form an active path that violates property (1). Similarly, if $Y \notin \mathbf{S}$, it follows that because the next edge (i.e., $X \rightarrow Y$) along both traversals is into Y, the last edge in both traversals is out of Y, and again we can form an active path in violation of property (1).

Without loss of generality, assume there is an $\mathbf{S}$-active path between A and X in $\mathcal{G}$. Property (4) now follows immediately because the traversal from B must have ended at X or at Y; because $Y \notin \mathbf{S}$, this sub-path is active. To prove property (5), we assume $Y \in \mathbf{S}$. If the traversal from B ended at X, the property follows immediately. Otherwise, the last edge in the traversal must have been into Y, and thus the next-to-last node is some parent X' of Y that is not in $\mathbf{S}$, and thus we conclude that there is an active path between B and X'. Now consider any other parent X'' of Y that is not in $\mathbf{S}$: we can form an active path between B and X'' by appending the active path between B and X' with the (active) path $X' \rightarrow Y \leftarrow X''$ in $\mathcal{G}$ that is out of X'. $\Box$

Corollary 23 *Let $\mathcal{G}$ be any DAG, and let $\mathcal{G}'$ be the DAG that results by adding the edge $X \rightarrow Y$ to $\mathcal{G}$. Let $\mathcal{H}$ be any DAG such that $\mathcal{G} \leq \mathcal{H}$. Then for any $\mathbf{S}$-active path π in $\mathcal{G}$ identified with properties 3, 4, or 5 from Lemma 22, there is a corresponding $\mathbf{S}$-active path between the endpoints of π in $\mathcal{H}$.*

Proof: Follows immediately because $\mathcal{G} \leq \mathcal{H}$. $\Box$

The following second corollary is convenient for our main proof because two of the additions made by the algorithm are edges into a node that is a sink in the independence map.

Corollary 24 *Let $\mathcal{G}$ be any DAG, and let $\mathcal{G}'$ be the DAG that results from adding the edge $X \rightarrow Y$ to $\mathcal{G}$. Let $\mathcal{H}$ be any DAG such that (1) $\mathcal{G} \leq \mathcal{H}$, (2) Y is a sink node in $\mathcal{H}$, and (3) $\mathcal{H}$ contains the edge $X \rightarrow Y$. Let $\pi(A, B)$ be any $\mathbf{S}$-active path in $\mathcal{G}'$ such that there is no S-active path between A and B in $\mathcal{G}$. If $Y \in \mathbf{S}$, then there is an $\mathbf{S}$-active path between A and B in $\mathcal{H}$.*

Proof: From Corollary 23 (Property 5) we know that in $\mathcal{H}$, there is an S-active path between one of the endpoints and X, and there is an S-active path between the other endpoint and either X or a parent of Y from $\mathcal{G}$ that is not in $\mathbf{S}$. Without loss of generality, assume there is an $\mathbf{S}$-active path between A and X in $\mathcal{H}$. Given the three preconditions of the corollary, it follows that every parent of Y in $\mathcal{G}'$ is a parent of Y in $\mathcal{H}$. Thus there is an $\mathbf{S}$-active path in $\mathcal{H}$ between B and some parent W of Y that is not in $\mathbf{S}$ (with $W = X$ a possibility). Consequently we can construct an $\mathbf{S}$-active path between A and B in $\mathcal{H}$ by connecting the two active paths from the endpoints with the active path $X \rightarrow Y \leftarrow W$ that is out of both X and W. $\square$

The next lemma is the key idea of Step 2 of the algorithm: it allows us to remove nodes from both of the input DAGs in order to simplify the problem.

Lemma 25 *Let $\mathcal{G}$ and $\mathcal{H}$ be two DAGs containing a node Y that is a sink in both DAGs and for which $\mathbf{Pa}_Y^{\mathcal{G}} = \mathbf{Pa}_Y^{\mathcal{H}}$. Let $\mathcal{G}'$ and $\mathcal{H}'$ denote the subgraphs of $\mathcal{G}$ and $\mathcal{H}$, respectively, that result by removing node Y and all its in-coming edges. Then $\mathcal{G} \leq \mathcal{H}$ if and only if $\mathcal{G}' \leq \mathcal{H}'$.*

Proof: (If) For this case, we assume that $\mathcal{G}' \leq \mathcal{H}'$, and show that any active path in $\mathcal{G}$ must also exist in $\mathcal{H}$, thus establishing that $\mathcal{G} \leq \mathcal{H}$. Let $\pi(A, B)$ be any $\mathbf{S}$-active path between A and B in $\mathcal{G}$.

If Y never appears in $\pi(A, B)$, then $\pi(A, B)$ is $\mathbf{T}$-active in $\mathcal{G}'$, where $\mathbf{T} = \mathbf{S} \setminus \{Y\}$, and thus by assumption, there is a corresponding $\mathbf{T}$-active path $\pi'(A, B)$ between A and B in $\mathcal{H}'$. Furthermore, because $\mathcal{H}'$ is a subgraph of $\mathcal{H}$, and because Y cannot appear in $\pi'(A, B)$ (Y does not exist in $\mathcal{H}'$), we conclude that $\pi(A, B)$ is $\mathbf{S}$-active in $\mathcal{H}$, thus proving the result. For the remainder of the proof, we assume that Y appears in $\pi(A, B)$.

Suppose $Y \in \mathbf{S}$. This implies that Y occurs as a collider at every position in $\pi(A, B)$. Consider a traversal from A to B along $\pi(A, B)$, and let $X \rightarrow Y \leftarrow X'$ and $Z' \rightarrow Y \leftarrow Z$ be the first and last occurrence of Y (as a collider) on the traversal. Because every sub-path of an $\mathbf{S}$-active path between members not in $\mathbf{S}$ is by definition active, and because neither X nor Z can be in $\mathbf{S}$ (else $\pi(A, B)$ would not be active through the identified colliders), we conclude that in $\mathcal{G}$ there exists (1) an $\mathbf{S}$-active path between A and X that does not pass through Y and (2) an $\mathbf{S}$-active path between B and Z that does not pass through Y. Clearly both of these paths are $\mathbf{T}$-active in $\mathcal{G}'$ given $\mathbf{T} = \mathbf{S} \setminus \{Y\}$, and by assumption that $\mathcal{G}' \leq \mathcal{H}'$, it follows that there exist corresponding $\mathbf{T}$-active paths $\pi'(A, X)$ and $\pi'(B, Z)$ in $\mathcal{H}'$. Because $\mathcal{H}'$ is a sub-graph of $\mathcal{H}$, both $\pi'(A, X)$ and $\pi'(B, Z)$ are $\mathbf{T}$-active in $\mathcal{H}$. Furthermore, because neither path contains Y, they are both $\mathbf{S}$-active in $\mathcal{H}$ as well. This means we can append them together with the $\mathbf{S}$-active path $X \rightarrow Y \leftarrow Z$ that is out of both X and Z to create an $\mathbf{S}$-active path between A and B in $\mathcal{H}$.

Suppose $Y \notin \mathbf{S}$. Then because Y is a sink node in $\mathcal{G}$, the only time it can occur in $\pi(A, B)$ is as an endpoint. Without loss of generality, assume $Y = A$. In the degenerate case when Y is also B, the result follows trivially, so we assume that there is at least one edge in $\pi(A, B)$. Because Y is a sink in $\mathcal{G}$, we know the first edge in $\pi(A, B)$ is into Y: let $X \rightarrow Y$ denote this first edge. Clearly X cannot be in $\mathbf{S}$, which means that there is an $\mathbf{S}$-active path between X and B that does not include Y. This means that the same path is $\mathbf{S}$-active in $\mathcal{G}'$, and therefore because we are assuming $\mathcal{G}' \leq \mathcal{H}'$, there must exist an $\mathbf{S}$-active path $\pi'(B, X)$ between B and X in $\mathcal{H}'$. Because the parent sets of Y are identical in $\mathcal{G}$ and

$\mathcal{H}$, it follows that the edge $X \rightarrow Y$ exists in $\mathcal{H}$ and constitutes an $\mathbf{S}$-active path in $\mathcal{H}$ that can be appended to $\pi'(B, X)$ to create an $\mathbf{S}$-active path between $Y = A$ and B in $\mathcal{H}$.
(Only If) For this case, we assume that $\mathcal{G} \leq \mathcal{H}$, and show that any active path in $\mathcal{G}'$ must also exist in $\mathcal{H}'$, thus establishing that $\mathcal{G}' \leq \mathcal{H}'$. Let $\pi'(A, B)$ be any $\mathbf{S}$-active path between A and B in $\mathcal{G}'$. Because Y does not exist in $\mathcal{G}'$ we can assume, without loss of generality, that $Y \notin \mathbf{S}$. Because $\mathcal{G}'$ is a subgraph of $\mathcal{G}$, $\pi'(A, B)$ is $\mathbf{S}$-active in $\mathcal{G}$. By the assumption that $\mathcal{G} \leq \mathcal{H}$, it follows that there exists a corresponding $\mathbf{S}$-active path $\pi(A, B)$ between A and B in $\mathcal{H}$. Because $Y \notin \mathbf{S}$ and Y is a sink node in $\mathcal{H}$, Y cannot be in $\pi(A, B)$, and thus this path is $\mathbf{S}$-active in $\mathcal{H}'$. □

We are now almost ready to present the main proof; we first need some simple intermediate results, the first of which was proved by Verma and Pearl (1991).

Lemma 26 (Verma and Pearl, 1991) *If nodes X and Y are not adjacent in some DAG $\mathcal{G}$, then for the set $\mathbf{S} = \mathbf{Pa}_X^{\mathcal{G}} \cup \mathbf{Pa}_Y^{\mathcal{G}}$, there is no $\mathbf{S}$-active path between X and Y in $\mathcal{G}$.*

Proposition 27 *Let $\mathcal{G}$ and $\mathcal{H}$ be two DAGs such that $\mathcal{G} \leq \mathcal{H}$. If there is an edge between X and Y in $\mathcal{G}$, then there is an edge between X and Y in $\mathcal{H}$.*

Proof: Follows immediately from Lemma 26 and the fact that an edge between X and Y constitutes an $\mathbf{S}$-active path for any $\mathbf{S}$ that does not include X or Y. □

Lemma 28 *Let $\mathcal{G}$ and $\mathcal{H}$ be two DAGs such that $\mathcal{G} \leq \mathcal{H}$. If $\mathcal{G}$ contains the v-structure $X \rightarrow Z \leftarrow Y$, then either $\mathcal{H}$ contains the same v-structure or X and Y are adjacent in $\mathcal{H}$.*

Proof: Suppose this is not the case and $\mathcal{H}$ does not contain the v-structure and X and Y are not adjacent in $\mathcal{H}$. From Proposition 27, we know that in $\mathcal{H}$, Z must be adjacent to both X and Y. By our supposition, Z is a parent of either X or Y in $\mathcal{H}$. This implies by Lemma 26 that there exists a conditioning set $\mathbf{S}$ that includes node Z (and does not include either node X or node Y) for which no active path exists between X and Y in $\mathcal{H}$. But the path $X \rightarrow Z \leftarrow Y$ in $\mathcal{G}$ is active given any conditioning set that includes Z (and excludes X and Y), including the set $\mathbf{S}$, which contradicts the fact that $\mathcal{G} \leq \mathcal{H}$. □

Lemma 28 was also proven by Kočka et al. (2001b). For the next lemma, recall from the definition of $\mathbf{De}_Y^{\mathcal{G}}$ that Y is included in this set.

Lemma 29 *Suppose $\mathcal{G} \leq \mathcal{H}$. For any node Y, there is a unique maximal element in $\mathcal{H}$ from the set $\mathbf{De}_Y^{\mathcal{G}}$.*

Proof: Suppose not, and let D_1 and D_2 be any two maximal elements in $\mathcal{H}$. Because these nodes are both descendants of Y in $\mathcal{G}$, there is an $\mathbf{S}$-active path in $\mathcal{G}$ between them for any $\mathbf{S}$ that does not contain any node in $\mathbf{De}_Y^{\mathcal{G}}$. By definition of D_1 and D_2, in $\mathcal{H}$ neither has a parent from $\mathbf{De}_Y^{\mathcal{G}}$, and thus by Lemma 26, $\mathbf{S} = \mathbf{Pa}_{D_1}^{\mathcal{H}} \cup \mathbf{Pa}_{D_2}^{\mathcal{H}}$ constitutes precisely such a set that renders them independent in $\mathcal{H}$, contradicting the fact that $\mathcal{G} \leq \mathcal{H}$. □

We can now prove that Algorithm Apply-Edge-Orientation is correct.

Lemma 30 *Let $\mathcal{G}$ and $\mathcal{H}$ be two DAGs such that $\mathcal{G} \leq \mathcal{H}$ and $\mathcal{G} \neq \mathcal{H}$. Let $\mathcal{G}'$ denote the graph returned by* Algorithm Find-Edge-Operation$(\mathcal{G}, \mathcal{H})$. *Then $\mathcal{G}'$ is a DAG such that $\mathcal{G}' \leq H$ and if the operation was an edge reversal, then the edge was covered in $\mathcal{G}$.*

Proof: In Step 2 of the algorithm the input DAGs are simplified by repeatedly removing common sink nodes that have the same parents in both DAGs. Let $\mathcal{G}_S$ and $\mathcal{H}_S$ denote these simplified versions of the input DAGs. It follows immediately from Lemma 25 that if we find an edge modification to $\mathcal{G}_S$ such that $\mathcal{H}_S$ is an independence map of the resulting DAG, then $\mathcal{H}$ is an independence map of the DAG that results from that same edge modification in $\mathcal{G}$. Furthermore, because only sink nodes are removed, any covered edge reversal in $\mathcal{G}_S$ corresponds to a covered edge reversal in $\mathcal{G}$. Thus we can concentrate on identifying an edge to modify using the simplified problem. For notational simplicity, we will use $\mathcal{G}$ and $\mathcal{H}$ to denote the simplified versions of the input DAGs for the remainder of the proof.

We know that after Step 2, $\mathcal{G}$ and $\mathcal{H}$ have at least two nodes, else we conclude that *all* nodes were removed in Step 2, contradicting the fact that $\mathcal{G} \neq \mathcal{H}$. Thus the node Y identified in Step 3 must exist.

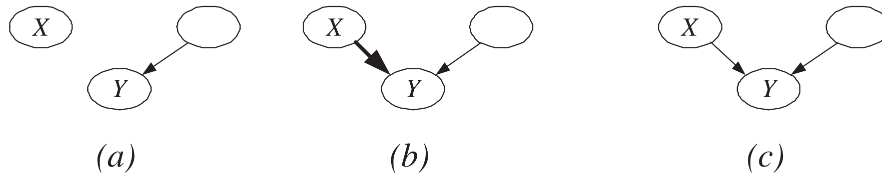

Figure 9: Relevant portions of the DAGs for the edge addition at Step 4: (a) $\mathcal{G}$, (b) $\mathcal{G}'$ resulting from the edge addition, and (c) $\mathcal{H}$.

If there are no children of Y in $\mathcal{G}$ (Step 4), we simply choose any node X that is a parent of Y in $\mathcal{H}$ but not a parent of Y in $\mathcal{G}$ (See Figure 9), and we return the DAG $\mathcal{G}'$ that results from adding $X \rightarrow Y$ to $\mathcal{G}$. We know that such an X exists, else we would have removed Y in Step 2. Consider any $\mathbf{S}$-active path $\pi(A, B)$ in $\mathcal{G}'$ that is not active in $\mathcal{G}$. Recall from Lemma 22 (Property 1) that Y must be an element of $\pi(A, B)$. Because Y is a sink in $\mathcal{G}'$, we know that it must either be an endpoint of $\pi(A, B)$ or it must be a member of $\mathbf{S}$. If Y is an endpoint, we know by Corollary 23 (Property 3) that in $\mathcal{H}$ there is an $\mathbf{S}$-active path from the other endpoint and X. Thus by appending this path with the edge $X \rightarrow Y$ (which is out of X), we have identified an $\mathbf{S}$-active path between A and B in $\mathcal{H}$. If Y is in $\mathbf{S}$, it follows immediately from Corollary 24 that there is an $\mathbf{S}$-active path between A and B in $\mathcal{H}$. Thus we conclude that if $\mathcal{G}'$ is returned by the algorithm at Step 4, then $\mathcal{G}' \leq \mathcal{H}$.

If we get to Step 5, there is at least one child of Y in $\mathcal{G}$, and we apply a somewhat complicated rule for choosing a *particular* child Z on which to concentrate (see Figure 10). We first use the DAG $\mathcal{G}$ to identify the set $\mathbf{De}_Y^{\mathcal{G}}$ of descendants of Y in $\mathcal{G}$. Then, we turn our attention to DAG $\mathcal{H}$ and identify the maximal element D of the set $\mathbf{De}_Y^{\mathcal{G}}$ with respect to $\mathcal{H}$. From Lemma 29, this maximal element is necessarily unique, and because Y is a sink node in $\mathcal{H}$, it follows that $D \neq Y$. Thus in $\mathcal{G}$, D must be a descendant of some maximal child of Y, and therefore the node Z at Step 5 is well defined.

For Step 6 of the algorithm, if $Y \rightarrow Z$ is covered in $\mathcal{G}$ then by Lemma 2 the DAG $\mathcal{G}'$ that results from reversing the covered edge $Y \rightarrow Z$ in $\mathcal{G}$ is equivalent to $\mathcal{G}$, and thus $\mathcal{G}' \leq \mathcal{H}$. If the edge $Y \rightarrow Z$ is not covered in $\mathcal{G}$, then by definition of a covered edge there is either a

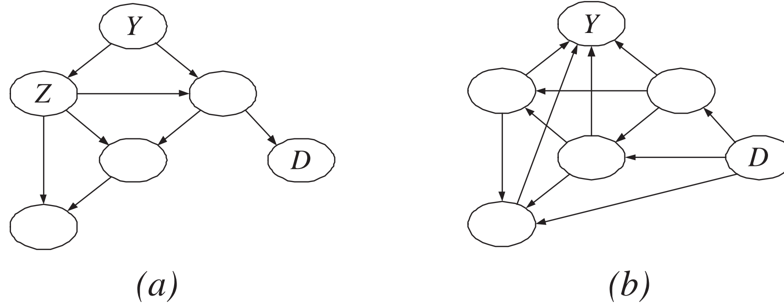

Figure 10: Selection of node Z at Step 5: (a) example DAG $\mathcal{G}$ and (b) corresponding DAG $\mathcal{H}$. All nodes (including Y) are members of $\mathbf{De}_Y^{\mathcal{G}}$.

parent of Y that is not a parent of Z, or there is a parent of Z that is not a parent of Y. These two cases are tested in Step 7 and Step 8, respectively.

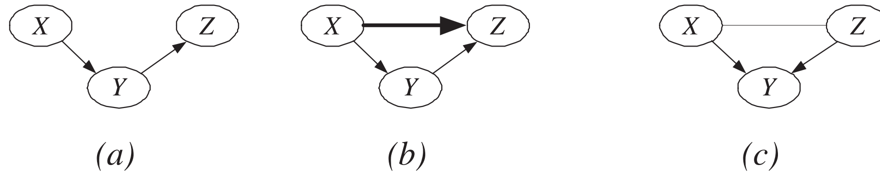

Figure 11: Relevant portion of DAGs for the edge addition at Step 7: (a) $\mathcal{G}$, (b) $\mathcal{G}'$ that results from the edge addition, and (c) $\mathcal{H}$.

If (in Step 7) Y has some parent X that is not a parent of Z in $\mathcal{G}$, then we return the DAG $\mathcal{G}'$ that results from adding $X \rightarrow Z$ to $\mathcal{G}$ (see Figure 11). First we note that because there is a directed path from X to Z in $\mathcal{G}$ (X is a parent of Y and Z is a child of Y), the addition will not create a cycle. To see that $\mathcal{H}$ remains an independence map, consider any $\mathbf{S}$-active path $\pi(A, B)$ between A and B in $\mathcal{G}'$ that is not active in $\mathcal{G}$. Recall from Lemma 22 (Property 1) that $\pi(A, B)$ must include the edge $X \rightarrow Z$. It must be the case that $Y \in \mathbf{S}$, or else we could replace every occurrence of the edge $X \rightarrow Z$ in $\pi(A, B)$ with the path $X \rightarrow Y \rightarrow Z$ to construct an $\mathbf{S}$-active path between A and B in $\mathcal{G}$. If $Z \notin \mathbf{S}$, we conclude from Corollary 23 (Property 4) that in $\mathcal{H}$ there is an S-active path between each endpoint and either X or Z. Because Y is a child of both X and Z in $\mathcal{H}$, we can connect these two active paths together in $\mathcal{H}$ (using either $X \rightarrow Y \leftarrow X$ or $X \rightarrow Y \leftarrow Z$; from property 4 we know at least one of the paths ends with X) to construct an S-active path between A and B in $\mathcal{H}$. If $Z \in \mathbf{S}$, we know from Corollary 23 (Property 5) that in $\mathcal{H}$ there is an active path between one of the endpoints and X, and an active path between the other endpoint and either X or some parent of Z not in $\mathbf{S}$. Without loss of generality, assume there is an $\mathbf{S}$-active path between A and X in $\mathcal{H}$. If the path from B ends at X we establish an $\mathbf{S}$-active path between A and B in $\mathcal{H}$ by connecting these paths together with the active

path $X \rightarrow Y \leftarrow X$ that is out of X. Otherwise, assume that the path from B ends at node W, where $W \notin \mathbf{S}$ is a parent of Z in both $\mathcal{G}$ and $\mathcal{G}'$. It must be the case that W is adjacent to Y in $\mathcal{H}$—and hence because Y is a sink, W must be a *parent* of Y in $\mathcal{H}$—by the following argument: if W is not adjacent to Y in $\mathcal{G}$, then $Y \rightarrow Z \leftarrow W$ is a v-structure not in $\mathcal{H}$ and the adjacency is established from Lemma 28; if W is adjacent to Y in $\mathcal{G}$ then the adjacency is established from Proposition 27. Because W is a parent of Y in $\mathcal{H}$, we can construct an $\mathbf{S}$-active path between A and B by connecting the two paths from A and B with the active path $X \rightarrow Y \leftarrow W$. Thus we conclude that if $\mathcal{G}'$ is returned by the algorithm at Step 7, then $\mathcal{G}' \leq \mathcal{H}$.

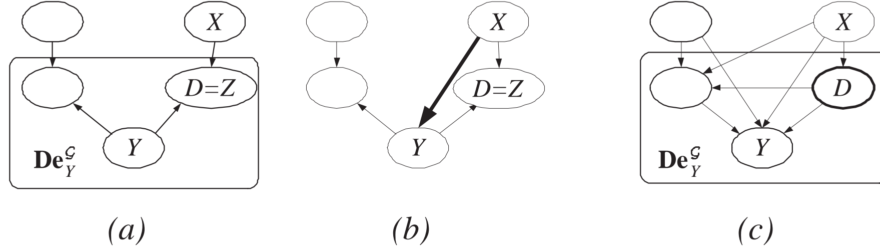

Figure 12: Edge addition at Step 8: (a) example DAG $\mathcal{G}$, (b) DAG $\mathcal{G}'$ resulting from the edge addition and (c) corresponding DAG $\mathcal{H}$.

Finally, if Step 8 is reached, we know that Z must have some parent X that is not a parent of Y, and we return the DAG $\mathcal{G}'$ that results from adding the edge $X \rightarrow Y$ to $\mathcal{G}$ (see Figure 12). We now argue that the edge addition cannot form a cycle. If it did, then there must be a directed path from Y to X in $\mathcal{G}$. The *first* node W in this path—which is a child of Y—is either equal to X or is an ancestor of X, which means that W is an ancestor of Z. But this contradicts the fact that Z is a maximal child of Y that has D as a descendant.

As above, let $\pi(A, B)$ be any $\mathbf{S}$-active path between A and B in $\mathcal{G}'$ that is not active in $\mathcal{G}$. Recall from Lemma 22 (Property 1) that $\pi(A, B)$ must include the edge $X \rightarrow Y$. We will now demonstrate that there must be a corresponding $\mathbf{S}$-active path between A and B in $\mathcal{H}$.

We first consider the case when $Y \in \mathbf{S}$. Because Y is a sink in $\mathcal{H}$, we know that $\mathcal{H}$ cannot include the v-structure $X \rightarrow Z \leftarrow Y$ that exists in $\mathcal{G}$ (but not $\mathcal{G}'$), and we conclude from Lemma 28 (and the fact that Y is a sink node in $\mathcal{H}$) that the edge $X \rightarrow Y$ must exist in $\mathcal{H}$. Thus from Corollary 24 it follows that there is an $\mathbf{S}$-active path between A and B in $\mathcal{H}$. For the remainder of the proof, we consider the case when $Y \notin \mathbf{S}$.

It must be the case that no member from $\mathbf{De}_Z^{\mathcal{G}}$ is in $\mathbf{S}$; otherwise, we could replace each occurrence of the edge $X \rightarrow Y$ with an active path $X \rightarrow Z \rightarrow \ldots \rightarrow S \leftarrow \ldots \leftarrow Z \leftarrow Y$ for any such descendant S, and thus construct an active path in $\mathcal{G}$ between A and B.

We now show that there must be $\mathbf{S}$-active paths between each endpoint and the node D chosen in Step 5 of the algorithm. First, from Lemma 22 (Property 4), there is an active path in $\mathcal{G}$ from each endpoint to either X or Y. Because D is a descendant of Z (and hence both X and Y) in $\mathcal{G}$, and because neither Y nor any of the descendants of Z (including Z itself) are in $\mathbf{S}$, we can append to each of these active paths a directed path to D to

construct **S**-active paths between both endpoints and D in $\mathcal{G}$. Because $\mathcal{G} \leq \mathcal{H}$, it follows that there exists corresponding **S**-active paths in $\mathcal{H}$ as well.

Given that there is an **S**-active path between both endpoints and D in $\mathcal{H}$, if we can identify a directed path in $\mathcal{H}$ from D to either (1) one of the endpoints or (2) an element of **S**, then we can easily identify an **S**-active path between A and B in $\mathcal{H}$: consider the shortest such directed path. If the path reaches an endpoint, then the directed path constitutes an **S**-active path in $\mathcal{H}$ between D and that endpoint that is out of D, which means we can append it to the **S**-active path between the *other* endpoint and D to create the desired path. If the path reaches an element S of **S**, then we can append to that directed path the same path in the opposite direction to create an **S**-active path between D and itself that is out of D on both endpoints (i.e., $D \rightarrow \ldots \rightarrow S \leftarrow \ldots \leftarrow D$), which can then be used to connect the two active paths between the endpoints and D to create the desired path.

All that remains is to show that there must be a descendant of D in $\mathcal{H}$ that is either one of the endpoints or an element of **S**. To do so, we turn our attention back to the active path $\pi(A, B)$ in $\mathcal{G}'$. Consider any segment of $\pi(A, B)$ that starts with the edge $X \rightarrow Y$, and then continues in the direction of the edge until either the path ends or an edge is encountered in the other direction. Clearly this directed path (which might end immediately at Y) ends at either an endpoint (i.e., A or B) or a member of **S**. In either case, the last node is a descendant of Y in $\mathcal{G}$. Because D is the *unique* (Lemma 29) maximal element of $\mathbf{De}_Y^{\mathcal{G}}$ within $\mathcal{H}$, it follows that any such descendant of Y in $\mathcal{G}$ is a descendant of D in $\mathcal{H}$. Thus we conclude that there is an **S**-active path between A and B in $\mathcal{H}$, and that if $\mathcal{G}'$ is returned by the algorithm at Step 8, then $\mathcal{G}' \leq \mathcal{H}$. □

Finally, we can prove the main result of this paper, which we state again below.

Theorem 4 *Let $\mathcal{G}$ and $\mathcal{H}$ be any pair of DAGs such that $\mathcal{G} \leq \mathcal{H}$. Let r be the number of edges in $\mathcal{H}$ that have opposite orientation in $\mathcal{G}$, and let m be the number of edges in $\mathcal{H}$ that do not exist in either orientation in $\mathcal{G}$. There exists a sequence of at most $r + 2m$ edge reversals and additions in $\mathcal{G}$ with the following properties:*

1. *Each edge reversed is a covered edge*
2. *After each reversal and addition $\mathcal{G}$ is a DAG and $\mathcal{G} \leq \mathcal{H}$*
3. *After all reversals and additions $\mathcal{G} = \mathcal{H}$*

Proof: Properties 1 through 3 follow immediately from Lemma 30 if we simply apply the edge operation from Algorithm Find-Edge-Operation($\mathcal{G}, \mathcal{H}$) until $\mathcal{G} = \mathcal{H}$. We now show that the algorithm is called at most $r + 2m$ times. If a covered edge is reversed in $\mathcal{G}$ by the algorithm, we know (see Step 6) that *after* the reversal, the edge has the same orientation as in $\mathcal{H}$, and therefore r is reduced by exactly one and m remains constant; thus the sum $r + 2m$ is reduced by exactly one. If an edge is added, it follows from Lemma 30 that $\mathcal{H}$ is an independence map of the resulting DAG, and thus by Proposition 27 m is necessarily reduced by one; in this case r either remains constant or is increased by one, and thus the sum $r + 2m$ is reduced by either two or one. □

Appendix B: Operator Proofs

In this appendix, we provide proofs for the main results in Section 5. We show that the conditions given in Table 1 are necessary and sufficient for an *Insert* and *Delete* operator to be valid for the first and second phase, respectively, of the GES algorithm. An immediate corollary of the proof for each operator type is the increase in score that results.

The appendix is organized as follows. In Appendix B.1, we provide numerous preliminary results, the majority of which are proved by Chickering (2002). Then in Appendix B.2 and B.3, we provide the main results for the *Insert* and *Delete* operators, respectively.

B.1 Preliminary Results

The main proofs in this appendix rely on many intermediate results, most of which are proven by Chickering (2002). In this section, we enumerate all of these intermediate results.

The following proposition characterizes the conditions under which a v-structure exists in one PDAG but not another.

Proposition 31 *Let $\mathcal{P}$ and $\mathcal{P}'$ denote any pair of PDAGs. Let $X \rightarrow Y \leftarrow Z$ be any v-structure in $\mathcal{P}'$ that is not in $\mathcal{P}$. Then one of the following conditions must hold: (1) $X \notin \mathbf{Pa}_Y^{\mathcal{P}}$, (2) $Z \notin \mathbf{Pa}_Y^{\mathcal{P}}$, or (3) X and Z are adjacent in $\mathcal{P}$.*

Proof: Follows immediately from the definition of a v-structure. □

The next several results show how the edge status—either compelled or reversible—of some of the edges in a PDAG can constrain the status of other edges. An edge is compelled (reversible) in a PDAG if the corresponding edge is compelled (reversible) in a consistent extension of that PDAG.

Proposition 32 (Chickering, 2002) *Let $\mathcal{P}$ be any PDAG that admits a consistent extension and contains a compelled edge $X \rightarrow Y$. If there is an edge, either directed or undirected, between Y and some node Z such that Z and X are not adjacent, then that edge is compelled.*

Proposition 33 (Chickering, 2002) *Let $\mathcal{P}$ be any PDAG that admits a consistent extension such that there is a directed path from X to Y consisting of compelled edges. If there is an edge between X and Y, it is compelled as $X \rightarrow Y$.*

Lemma 34 (Chickering, 1995) *Let $\{X, Y, Z\}$ be any three nodes that form a clique of size three in PDAG $\mathcal{P}$. If any two of the edges in the clique are reversible, then the third edge is reversible as well.*

Lemma 35 (Chickering, 2002) *For any directed edge $X \rightarrow Y$ in a completed PDAG, X is a parent of every node reachable by Y via undirected edges.*

Lemma 36 (Chickering, 2002) *Let $\mathbf{X} = \{X_1, ..., X_n\}$ be the nodes from any undirected clique of size n within some undirected component of a completed PDAG $\mathcal{P}^c$, and let τ denote any total ordering of the nodes in $\mathbf{X}$. There exists a consistent extension of $\mathcal{P}^c$ for which (1) the edge orientations among the nodes in $\mathbf{X}$ are consistent with τ, and (2) any edge between X_i and a neighbor Y that is not in $\mathbf{X}$ is oriented as $X_i \rightarrow Y$.*

The final set of results are properties of semi-directed paths (see Definition 14) from a completed PDAG.

Lemma 37 (Chickering, 2002) *Let $\mathcal{P}^c$ be a completed PDAG that contains a semi-directed path from X to Y. If there exists a directed edge $Z \rightarrow W$ in this path, then there exists a directed path from Z to Y in $\mathcal{P}^c$.*

Corollary 38 (Chickering, 2002) *Let $\mathcal{P}^c$ be a completed PDAG. If $\mathcal{P}^c$ contains a semi-directed path from X to Y consisting of intermediate nodes contained within some set $\mathbf{N}$, then the* shortest *semi-directed path whose intermediate nodes are contained in $\mathbf{N}$ consists of exactly two consecutive segments, where the first segment consists entirely of undirected edges and the second segment consists entirely of directed edges.*

Corollary 39 (Chickering, 2002) *Let $\mathcal{P}^c$ be a completed PDAG. If $\mathcal{P}^c$ contains a semi-directed path from X to Y consisting of intermediate nodes contained within some set $\mathbf{N}$, then for any* shortest *semi-directed path whose intermediate nodes are contained in $\mathbf{N}$, there is no edge in $\mathcal{P}^c$ that connects a pair of non-consecutive nodes along the path.*

Lemma 40 *Let $\mathcal{P}^c$ be a completed PDAG that contains a semi-directed path from X to Y, and let $X - W$ be the first edge along any* shortest *such semi-directed path. If the edge between X and W is directed as $X \rightarrow W$ in some consistent extension $\mathcal{G}$ of $\mathcal{P}^c$, then there is a directed path from X to Y in $\mathcal{G}$.*

Proof: Suppose not, and let $B \leftarrow C$ be the first edge that is directed *away* from Y along the path. From Corollary 38, we know that this edge must be reversible, as must be the edge $A \rightarrow B$ that precedes it. But from Corollary 39, A and C are not adjacent, and thus $A \rightarrow B \leftarrow C$ is a v-structure, yielding a contradiction. □

The conditions from Table 1 include checking that some set of neighbors of a node in a completed PDAG are a clique. It follows immediately from Lemma 34 that if any set of neighbors is a clique, then that set of neighbors is a clique of *undirected* edges. It is to be understood that in the sections to follow that when we use *clique*, we mean a clique of undirected edges.

We say that Y is a *reversible parent* of X in a PDAG or DAG if the edge $Y \rightarrow X$ is reversible. Similarly, we say that Y is a *compelled parent* of X if $Y \rightarrow X$ is compelled. We use analogous definitions for *reversible child* and *compelled child.*

B.2 The Insert Operator

In this section, we show that the conditions in Table 1 are necessary and sufficient for determining whether an *Insert* operator is valid during the first phase of GES. In particular, we show in Theorem 15 that the conditions hold if and only if we can extract a consistent extension $\mathcal{G}$ of the completed PDAG $\mathcal{P}^c$ to which adding a single directed edge results in a consistent extension $\mathcal{G}'$ of the completed PDAG $\mathcal{P}^{c\prime}$ that results from applying the operator. The "if" part of the proof is constructive; that is, we identify a specific $\mathcal{G}$ to which we can add the edge. The increase in score that results from the operator thus follows immediately.

First, we need the following result:

Lemma 41 *Let $\mathcal{P}^c$ be any completed PDAG with consistent extension $\mathcal{G}$. Let $\mathcal{P}^{c\prime}$ denote the completed PDAG that results from applying the operator $Insert(X, Y, \mathbf{T})$ to $\mathcal{P}^c$, where $\mathbf{T}$ is a clique consisting of nodes that are neighbors of Y that are not adjacent to X. Let $\mathcal{G}'$ denote the graph that results from adding $X \to Y$ to $\mathcal{G}$. Then $\mathcal{G}'$ has the same adjacencies and the same set of v-structures as $\mathcal{P}^{c\prime}$ if and only if the set of reversible parents of Y in $\mathcal{G}$ that are not adjacent to X is equal to $\mathbf{T}$.*

Proof: Clearly $\mathcal{G}'$ and $\mathcal{P}^{c\prime}$ have the same adjacencies. Because $\mathcal{G}$ is a consistent extension of $\mathcal{P}^c$, any difference in v-structures between $\mathcal{G}'$ and $\mathcal{P}^{c\prime}$ must have resulted from the modification to either the completed PDAG or the DAG. From Proposition 31 and the fact that the *Insert* operator does not undirect or reverse any directed edges, it is easy to see that the set of v-structures that are in $\mathcal{P}^c$ but not in $\mathcal{P}^{c\prime}$ are precisely the set of v-structures that are in $\mathcal{G}$ but not in $\mathcal{G}'$. In other words, the set of v-structures that we lose as a result of performing the *Insert* operator to $\mathcal{P}^c$ is the same as the set that we lose as a result of adding $X \to Y$ to $\mathcal{G}$; these are precisely the ones whose "tails" are made adjacent as a result of the edge addition. We establish the result by showing that the set of v-structures that we *gain* as a result of the two modifications is the same if and only if the set of reversible parents of Y in $\mathcal{G}$ is equal to $\mathbf{T}$.

Because $\mathbf{T}$ is a clique in $\mathcal{P}^c$ (and therefore a clique of undirected edges) we know from Lemma 35 that any parent of a node in $\mathbf{T}$ is a parent of *every* node in $\mathbf{T}$; this implies that any v-structure that includes a previously undirected edge must have $X \to Y$ as the other edge. It follows that the set of v-structures that are in $\mathcal{P}^{c\prime}$ but not in $\mathcal{P}^c$ are those of the form $X \to Y \leftarrow Z$, where Z is either a member of $\mathbf{T}$ or is a parent of Y that is not adjacent to X in $\mathcal{P}^c$. It is easy to see that the set of v-structures in $\mathcal{G}'$ that are not in $\mathcal{G}$ are of the form $X \to Y \leftarrow Z$, where Z is a parent of Y in $\mathcal{G}$ that is not adjacent to X.

Consider the set of parents of Y that are not adjacent to X in $\mathcal{G}$. Clearly this set consists of the union of (1) compelled parents of Y that are not adjacent to X and (2) reversible parents of Y that are not adjacent to X. Because this first set is precisely the set of parents of Y in $\mathcal{P}^c$ that are not adjacent to X, the lemma follows. $\square$

Theorem 15 *Let $\mathcal{P}^c$ be any completed PDAG, and let $\mathcal{P}^{c\prime}$ denote the result of applying an $Insert(X, Y, \mathbf{T})$ operator to $\mathcal{P}^c$. There exists a consistent extension $\mathcal{G}$ of $\mathcal{P}^c$ to which adding the edge $X \to Y$ results in a consistent extension $\mathcal{G}'$ of $\mathcal{P}^{c\prime}$ if and only if in $\mathcal{P}^c$*

1. *$\mathbf{NA}_{Y,X} \cup \mathbf{T}$ is a clique*
2. *Every semi-directed path from Y to X contains a node in $\mathbf{NA}_{Y,X} \cup \mathbf{T}$*

Proof: **(If)** Given that the first condition implies that $\mathbf{T}$ is a clique, it follows from Lemma 41 that we need only identify a consistent extension $\mathcal{G}$ of $\mathcal{P}^c$ with the following two properties: (*) the reversible parents of Y that are not adjacent to X are precisely the nodes in $\mathbf{T}$ and (**) there is no directed path from Y to X. Because $\mathbf{NA}_{Y,X} \cup \mathbf{T}$ is a clique, and because Y is a neighbor of all of these nodes, we conclude that $\mathbf{NA}_{Y,X} \cup \mathbf{T} \cup \{Y\}$ is also a clique. Therefore we conclude from Lemma 36 that there exists a consistent extension $\mathcal{G}$ of $\mathcal{P}^c$ for which the reversible parents of Y are precisely those nodes in $\mathbf{NA}_{Y,X} \cup \mathbf{T}$. Because all nodes in $\mathbf{NA}_{Y,X}$ are adjacent to X, (*) is satisfied for $\mathcal{G}$. It remains to be shown that there is no directed path from Y to X in $\mathcal{G}$. Suppose there does exist such a path. Clearly any

such directed path has a corresponding semi-directed path in $\mathcal{P}^c$. By the second condition of the lemma, however, this path must pass through a node in $\mathbf{NA}_{Y,X} \cup \mathbf{T}$, all of which are parents of Y in $\mathcal{G}$, yielding the contradiction that $\mathcal{G}$ is cyclic. Thus we conclude that (**) is satisfied for $\mathcal{G}$.

(Only if) Suppose $\mathbf{NA}_{Y,X} \cup \mathbf{T}$ is not a clique in $\mathcal{P}^c$. Then there are two nodes A and B in this set for which (1) $A - Y - B$ is in $\mathcal{P}^c$ and (2) A and B are not adjacent. Thus in any consistent extension $\mathcal{G}$, at least one of the corresponding edges must be directed away from Y, else $\mathcal{G}$ would contain a v-structure not in $\mathcal{P}^c$. Without loss of generality, assume the edge $Y \rightarrow A$ is in $\mathcal{G}$. If $A \in \mathbf{NA}_{Y,X}$, we know that the edge between A and X must be directed toward X, else $\mathcal{G}$ would contain the v-structure $X \rightarrow A \leftarrow Y$ that is not in $\mathcal{P}^c$. But this implies that the graph that results from adding $X \rightarrow Y$ to $\mathcal{G}$ is cyclic. If $A \in \mathbf{T}$, then the result of adding $X \rightarrow Y$ to $\mathcal{G}$ cannot include the v-structure $X \rightarrow Y \leftarrow A$; because this v-structure exists in $\mathcal{P}^{c\prime}$ as a result of the *Insert* operator, $\mathcal{G}'$ cannot be a consistent extension of $\mathcal{P}^{c\prime}$.

Suppose that there exists a semi-directed path from Y to X that does not pass through a node in $\mathbf{NA}_{Y,X} \cup \mathbf{T}$, and consider the shortest such path. If the first edge is directed away from Y, we conclude from Lemma 37 that there is a directed path from Y to X in $\mathcal{P}^c$ and thus the *Insert* operator results—before converting to the resulting completed PDAG representation—in a PDAG that contains a cycle; this PDAG does not admit a consistent extension. Otherwise, let $Y - A$ be the first edge in this path. By assumption, A is in neither $\mathbf{NA}_{Y,X}$ nor $\mathbf{T}$, and thus if $\mathcal{G}$ contains the edge $A \rightarrow Y$, $\mathcal{G}'$ contains the v-structure $X \rightarrow Y \leftarrow A$ that is not in $\mathcal{P}^{c\prime}$. If $\mathcal{G}$ contains the edge $Y \rightarrow A$, we conclude by Lemma 40 that there is a directed path from Y to X in $\mathcal{G}$ and consequently $\mathcal{G}'$ is cyclic. $\Box$

Corollary 16 *For any score-equivalent decomposable scoring criterion, the increase in score that results from applying a valid operator* $Insert(X, Y, \mathbf{T})$ *to a completed PDAG* $\mathcal{P}^c$ *is*

$$s(Y, \mathbf{NA}_{Y,X} \cup \mathbf{T} \cup \mathbf{Pa}_Y{}^{+X}) - s(Y, \mathbf{NA}_{Y,X} \cup \mathbf{T} \cup \mathbf{Pa}_Y)$$

Proof: Follows immediately from subtracting the score of $\mathcal{G}$ from the score of $\mathcal{G}'$, where $\mathcal{G}$ is defined in the "if" part of Theorem 15, and $\mathcal{G}'$ denotes the DAG that results from adding the edge $X \rightarrow Y$ to $\mathcal{G}$. $\Box$

B.3 The Delete Operator

In this section, we show that the conditions in Table 1 are necessary and sufficient for determining whether a *Delete* operator is valid during the second phase of GES. In particular, we show in Theorem 17 that the conditions hold if and only if we can extract a consistent extension $\mathcal{G}$ of the completed PDAG $\mathcal{P}^c$ to which deleting a single directed edge results in a consistent extension $\mathcal{G}'$ of the completed PDAG $\mathcal{P}^{c\prime}$ that results from applying the operator. The "if" part of the proof is constructive; that is, we identify a specific $\mathcal{G}$ from which we can delete the edge. The increase in score that results from the operator thus follows immediately.

First, we need the following result:

Lemma 42 *Let $\mathcal{P}^c$ be any completed PDAG with consistent extension $\mathcal{G}$ that includes the edge $X \to Y$. Let $\mathcal{P}^{c\prime}$ denote the completed PDAG that results from applying the operator $Delete(X, Y, \mathbf{H})$ to $\mathcal{P}^c$, where $\mathbf{H}$ consists of nodes that are neighbors of Y that are adjacent to X. Let $\mathcal{G}'$ denote the graph that results from deleting $X \to Y$ from $\mathcal{G}$. Then $\mathcal{G}'$ has the same adjacencies and same v-structures as $\mathcal{P}^{c\prime}$ if and only if the set of reversible children of Y in $\mathcal{G}$ that are children of X is equal to $\mathbf{H}$.*

Proof: Clearly $\mathcal{G}'$ and $\mathcal{P}^{c\prime}$ have the same adjacencies. Because $\mathcal{G}$ is a consistent extension of $\mathcal{P}^c$, any difference in v-structures between $\mathcal{G}'$ and $\mathcal{P}^{c\prime}$ must have resulted from the modification to either the completed PDAG or the DAG. From Proposition 31 and the fact that that the *Delete* operator does not undirect or reverse any directed edges, it is easy to see that the set of v-structures that are in $\mathcal{P}^c$ but not in $\mathcal{P}^{c\prime}$ are precisely the set of v-structures that are in $\mathcal{G}$ but not in $\mathcal{G}'$. In particular, the set of v-structures that we lose in both $\mathcal{P}^c$ or $\mathcal{G}$ are simply those v-structures that contain the edge between X and Y. We establish the result by showing that the set of v-structures that we *gain* as a result of the two modifications is the same if and only if the set of reversible children of Y in $\mathcal{G}$ that are children of X is equal to $\mathbf{T}$.

From Proposition 31 and the definition of the *Delete* operator, we immediately conclude that the set of v-structure in $\mathcal{P}^{c\prime}$ that are not in $\mathcal{P}^c$ are characterized by those v-structures in one of the two following forms: (1) $X \to Z \leftarrow Y$, where Z is either in $\mathbf{H}$ or both edges also exist in $\mathcal{P}^c$ (that is, the v-structure is formed because the adjacency between X and Y was removed, and zero or more of the edges were directed by the *Delete* operator) or (2) $A \to Z \leftarrow B$, where $Z \in \mathbf{H}$, and exactly one of the nodes A or B is either X or Y.

We now demonstrate that v-structures of form (2) never occur. Without loss of generality, assume that A is not a member of $\{X, Y\}$, and that $B = X$. By definition of the *Delete* operator, we know that $A \to Z$ must be directed in $\mathcal{P}^c$ (only edges incident to X or Y are made directed). Because the v-structure does not exist in $\mathcal{P}^c$, and the adjacency between A and B did not change as a result of the *Delete*, we conclude that the edge between Z and X must be undirected in $\mathcal{P}^c$. But from Proposition 32, this is impossible, and therefore we conclude that $A \to Z \leftarrow B$ must exist in $\mathcal{P}^c$.

Clearly, the set of v-structures gained as a result of deleting $X \to Y$ from $\mathcal{G}$ is characterized by those v-structures of the form $X \to Z \leftarrow Y$, where each Z in this case is simply a common child of both X and Y in $\mathcal{G}$. Thus the lemma follows if we can demonstrate that in $\mathcal{G}$, the common children of X and Y are precisely the common compelled children of X and Y unioned with the reversible children of Y.

Suppose that in $\mathcal{G}$ there exists a child Z of X and Y that is not a common compelled child and for which $Y \to Z$ is compelled. Because $X \to Z$ is not compelled, we conclude by Lemma 34 that $X \to Y$ is compelled. But this implies by Proposition 33 that $X \to Z$ *is* compelled, yielding a contradiction. □

Theorem 17 *Let $\mathcal{P}^c$ be any completed PDAG that contains either $X \to Y$ or $X - Y$, and let $\mathcal{P}^{c\prime}$ denote the result of applying the operator $Delete(X, Y, \mathbf{H})$ to $\mathcal{P}^c$. There exists a consistent extension $\mathcal{G}$ of $\mathcal{P}^c$ that contains the edge $X \to Y$ from which deleting the edge $X \to Y$ results in a consistent extension $\mathcal{G}'$ of $\mathcal{P}^{c\prime}$ if and only if $\mathbf{NA}_{Y,X} \setminus \mathbf{H}$ is a clique.*

Proof: **(If)** Suppose $\mathbf{NA}_{Y,X} \setminus \mathbf{H}$ is a clique. By definition of $\mathbf{NA}_{Y,X}$, it follows that $\{\mathbf{NA}_{Y,X} \setminus \mathbf{H}\} \cup \{Y\}$ is also a clique. If $X - Y$ exists in $\mathcal{P}^c$, then $\{\mathbf{NA}_{Y,X} \setminus \mathbf{H}\} \cup \{Y\} \cup \{X\}$ is a clique; otherwise, we know that $X \rightarrow Y$ is in $\mathcal{P}^c$. In either case, we conclude from Lemma 36 that we can extract a consistent extension $\mathcal{G}$ from $\mathcal{P}^c$—where $\mathcal{G}$ contains the edge $X \rightarrow Y$—whose directed edges are consistent with the following ordering: $\mathbf{NA}_{Y,X} \setminus \mathbf{H}$, then X, then Y, then the remaining nodes. Clearly in $\mathcal{G}$ the reversible children of Y that are adjacent to X are precisely the nodes in $\mathbf{H}$. Furthermore, because $\mathcal{G}$ contains the edge $X \rightarrow Y$, any child of Y that is adjacent to X is also a child of X. Thus we conclude from Lemma 42 that the result of deleting $X \rightarrow Y$ from $\mathcal{G}$ is a DAG $\mathcal{G}'$ that has the same adjacencies and v-structures as $\mathcal{P}^{c\prime}$. Because $\mathcal{G}$ is a DAG, $\mathcal{G}'$ must also be a DAG, and therefore the lemma follows.

(Only if) Suppose there exists a consistent extension $\mathcal{G}$ of $\mathcal{P}^c$ that contains the edge $X \rightarrow Y$ from which deleting the edge $X \rightarrow Y$ results in a consistent extension $\mathcal{G}'$ of $\mathcal{P}^{c\prime}$. From Lemma 42 we conclude that the set of reversible children of Y that are adjacent to X in $\mathcal{G}$ is precisely the set $\mathbf{H}$. Thus every element in $\mathbf{NA}_{Y,X} \setminus \mathbf{H}$ is a *parent* of Y in $\mathcal{G}$. Any pair of such parents A and B that were not adjacent would constitute the v-structure $A \rightarrow Y \leftarrow B$, which contradicts the fact that both these edges are reversible. □

Corollary 18 *For any score-equivalent decomposable scoring criterion, the increase in score that results from applying a valid operator Delete*$(X, Y, \mathbf{H})$ *to a completed PDAG* $\mathcal{P}^c$ *is*

$$s(Y, \{\mathbf{NA}_{Y,X} \setminus \mathbf{H}\} \cup \mathbf{Pa}_Y{}^{-X}) - s(Y, \{\mathbf{NA}_{Y,X} \setminus \mathbf{H}\} \cup \mathbf{Pa}_Y)$$

Proof: Follows immediately from subtracting the score of $\mathcal{G}$ from the score of $\mathcal{G}'$, where $\mathcal{G}$ is defined in the "if" part of Theorem 17, and $\mathcal{G}'$ denotes the DAG that results from deleting the edge $X \rightarrow Y$ from $\mathcal{G}$. □

Appendix C: Converting to a Completed PDAG

In Section 5, we defined the *Insert* and *Delete* operators to be local modifications to the completed PDAG representation of the current state. As described in that section, the result of applying an operator to a completed PDAG is a PDAG that is not necessarily completed. In this appendix, we describe a conversion algorithm that converts a PDAG to the completed PDAG representation of the corresponding equivalence class. Recall that this conversion algorithm—in light of the results of Section 5—need only be applied once for each state visited by GES; we can evaluate efficiently all adjacent states in the greedy search without using the conversion.

The conversion algorithm is, in fact, the combination of two algorithms described in much more detail by Chickering (2002). The first algorithm, which we refer to as PDAG-to-DAG, takes as input a PDAG representation for an equivalence class, and outputs a (DAG) member of that class. The second algorithm, which we refer to as DAG-to-CPDAG, takes as input a Bayesian-network structure, and outputs a completed PDAG representation of the equivalence class to which that structure belongs. Clearly, we can implement the desired conversion by first calling PDAG-to-DAG on the PDAG that results from applying an

operator, and then calling DAG-TO-CPDAG on the consistent extension obtained by the first algorithm.

We first consider a simple implementation of PDAG-TO-DAG due to Dor and Tarsi (1992). Let $\mathbf{N}_X$ denote the neighbors of node X in a PDAG $\mathcal{P}$. We first create a DAG $\mathcal{G}$ that contains all of the directed edges from $\mathcal{P}$, and no other edges. We then repeat the following procedure: First, select a node X in $\mathcal{P}$ such that (1) X has no out-going edges and (2) if $\mathbf{N}_X$ is non-empty, then $\mathbf{N}_X \cup \mathbf{Pa}_X$ is a clique. If $\mathcal{P}$ admits a consistent extension, the node X is guaranteed to exist. Next, for each undirected edge $Y - X$ incident to X in $\mathcal{P}$, insert a directed edge $Y \rightarrow X$ to $\mathcal{G}$. Finally, remove X and all incident edges from the $\mathcal{P}$ and continue with the next node. The algorithm terminates when all nodes have been deleted from $\mathcal{P}$.

Algorithm ORDER-EDGES($\mathcal{G}$)
Input: DAG $\mathcal{G}$
Output: DAG $\mathcal{G}$ with labeled total order on edges
1. Perform a topological sort on the NODES in $\mathcal{G}$
2. Set $i = 0$
3. **While** there are unordered EDGES in $\mathcal{G}$
4. Let Y be the lowest ordered NODE that has an unordered EDGE incident into it
5. Let X be the highest ordered NODE for which $X \rightarrow Y$ is not ordered
6. Label $X \rightarrow Y$ with order i
7. $i = i + 1$

Figure 13: Algorithm to produce a total ordering over the edges in a DAG. The algorithm is used by Algorithm LABEL-EDGES.

The version of DAG-TO-CPDAG that we provide was originally derived by Chickering (1995), and is asymptotically optimal on average. The algorithm labels all of the edges in a DAG as either "compelled" or "reversible"; given such a labeling, it is trivial to construct the corresponding completed PDAG. The first step of the algorithm is to define a total ordering over the edges in the given DAG. For simplicity, we present this step as a separate procedure listed in Figure 13. In Figure 13, a *topological sort* refers to any total ordering of the nodes where if X_i is an ancestor of X_j, then X_i must precede X_j in the ordering. To avoid confusion between ordered nodes and ordered edges, we have capitalized "node" and "edge" in the figure. In Figure 14, we show an algorithm of Chickering (1995) that labels the edges.

References

Andersson, S. A., Madigan, D., and Perlman, M. D. (1997). A characterization of Markov equivalence classes for acyclic digraphs. *Annals of Statistics*, 25:505–541.

Buntine, W. L. (1996). A guide to the literature on learning probabilistic networks from data. *IEEE Transactions on Knowledge and Data Engineering*, 8:195–210.

Algorithm Label-Edges($\mathcal{G}$)
Input: DAG $\mathcal{G}$
Output: DAG $\mathcal{G}$ with each edge labeled either “compelled” or “reversible”

1. Order the edges in $\mathcal{G}$ using **Algorithm Order-Edges**
2. Label every edge in $\mathcal{G}$ as “unknown”
3. **While** there are edges labeled “unknown” in $\mathcal{G}$
4. Let $X \rightarrow Y$ be the lowest ordered edge that is labeled “unknown”
5. **For** every edge $W \rightarrow X$ labeled “compelled”
6. **If** W is not a parent of Y
7. Label $X \rightarrow Y$ and every edge incident into Y with “compelled”
8. **Goto** 3
9. **Else**
10. Label $W \rightarrow Y$ with “compelled”
11. **If** there exists an edge $Z \rightarrow Y$ such that $Z \neq X$ and Z is not a parent of X
12. Label $X \rightarrow Y$ and all “unknown” edges incident into Y with “compelled”
13. **Else**
14. Label $X \rightarrow Y$ and all “unknown” edges incident into Y with “reversible”

Figure 14: Algorithm to label each edge in a DAG with “compelled” or “reversible”, which leads to an immediate implementation of DAG-to-CPDAG.

Chickering, D. M. (1995). A transformational characterization of Bayesian network structures. In Hanks, S. and Besnard, P., editors, *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 87–98. Morgan Kaufmann.

Chickering, D. M. (1996). Learning Bayesian networks is NP-Complete. In Fisher, D. and Lenz, H., editors, *Learning from Data: Artificial Intelligence and Statistics V*, pages 121–130. Springer-Verlag.

Chickering, D. M. (2002). Learning equivalence classes of Bayesian-network structures. *Journal of Machine Learning Research*, 2:445–498.

Chickering, D. M. and Meek, C. (2002). Finding optimal Bayesian networks. In Darwiche, A. and Friedman, N., editors, *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, pages 94–102. Morgan Kaufmann.

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

Dor, D. and Tarsi, M. (1992). A simple algorithm to construct a consistent extension of a partially oriented graph. Technical Report R-185, Cognitive Systems Laboratory, UCLA Computer Science Department.

Geiger, D., Heckerman, D., King, H., and Meek, C. (2001). Stratified exponential families: graphical models and model selection. *Annals of Statistics*, 29(2):505–529.

Gillispie, S. B. and Perlman, M. D. (2001). Enumerating Markov equivalence classes of acyclic digraph models. In Goldszmidt, M., Breese, J., and Koller, D., editors, *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, pages 171–177. Morgan Kaufmann.

Haughton, D. M. A. (1988). On the choice of a model to fit data from an exponential family. *The Annals of Statistics*, 16(1):342–355.

Heckerman, D. (1996). A tutorial on learning Bayesian networks. Technical Report MSR-TR-95-06, Microsoft Research.

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

Jeffreys, H. (1939). *Theory of Probability.* Oxford University Press.

Kočka, T., Bouckaert, R. R., and Studený, M. (2001a). On characterizing inclusion of Bayesian networks. In Breese, J. and Koller, D., editors, *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, pages 261–268. Morgan Kaufmann.

Kočka, T., Bouckaert, R. R., and Studený, M. (2001b). On the inclusion problem. Technical Report 2010, Academy of Sciences of the Czech Republic, Institute of Information Theory and Automation.

Meek, C. (1995). Causal inference and causal explanation with background knowledge. In Hanks, S. and Besnard, P., editors, *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 403–410. Morgan Kaufmann.

Meek, C. (1997). *Graphical Models: Selecting causal and statistical models.* PhD thesis, Carnegie Mellon University.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, CA.

Shachter, R. (1998). Bayes-ball: The rational pastime (for determining irrelevance and requisite information in belief networks and influence diagrams). In Cooper, G. and Moral, S., editors, *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 480–487. Morgan Kaufmann.

Spirtes, P., Glymour, C., and Scheines, R. (1993). *Causation, Prediction, and Search.* Springer-Verlag, New York.

Verma, T. and Pearl, J. (1991). Equivalence and synthesis of causal models. In Henrion, M., Shachter, R., Kanal, L., and Lemmer, J., editors, *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pages 220–227.