Exploiting Markov Equivalence for Fast Inference

MASTER THESIS

Submitted for the degree of
MASTER OF SCIENCE (M.SC.)

FRIEDRICH SCHILLER UNIVERSITY JENA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
MATHEMATICS

Author
Jenette SELLIN

Born December 22, 1995
in CALIFORNIA, USA

Advisor
Andreas GORAL

Supervisor
Prof. Dr. Joachim GIESEN

Jena, June 30, 2020

Abstract

As observational data collection becomes more accessible, the importance of modeling causality and answering inference queries efficiently increases. Causal relationships are often encoded in directed acyclic graphs (DAGs) and associated probability distributions, which together comprise a Bayesian network. Bayesian networks have the property of non-identifiability, meaning that several structurally distinguishable networks can encode the same probability distribution. Resultingly, the same inference queries can be asked over different Bayesian networks while producing statistically indistinguishable results. This thesis explores whether it can be computationally beneficial to exploit this non-identifiability property for faster inference; that is, whether one can achieve speedup in answering a sequence of inference queries by changing the representation of a Bayesian network on the fly to serve the queries. This exploration yields a short survey of related topics, a theorem for identifying the set of vertices a query depends on, an algorithm for making beneficial network transformations, and a discussion of their efficacy.

Keywords: Bayesian Networks, causal models, inference queries, optimization, non-identifiability, Markov Equivalence.

Zusammenfassung

Schlagwörter:

Acknowledgements

Notation and Abbreviations

DAG	Directed Acyclic Graph
iff	If and only if
$G = (V, E)$	A graph G with vertices V and edges E
$G, G', G_1, G_2, ..., G_n$	directed graphs
$H, H', H_1, H_2, ..., H_n$	undirected and mixed graphs
$B, B', B_1, B_2, ..., B_n$	Bayesian networks
Capital letters e.g. X, Y, Z	random variables
Lowercase letters e.g. x, y, z	values attained by random variables
$p(X = x)$	The probability that the random variable X takes on the value x.
$p(X = x, Y = y)$	The probability of $X = x$ and $Y = y$ simultaneously.
$p(X = x \mid Y = y)$	The probability that $X = x$ given the condition $Y = y$.
$p(X)$	The joint probability function of possible values attained by X.
$\alpha(X)$	the set of ancestors of a vertex X in a graph G
$\prod_X^G$	the set of parents of a vertex X in a graph G
$\Delta(G, q)$	the number of vertices involved in a query q on a graph G.
$\Delta*$	$\arg\min_{G' \in [G]}(\Delta(G', q))$.
$[G]$	The Markov equivalence class of G
MEC	Markov Equivalence Class
$G \approx_M G'$	G is Markov equivalent to G'
G_*	essential graph of G

Contents

Chapter 1

Introduction

1.1 Motivation

Many fields rely on graphical models as a framework for drawing inference on observational data, as they allow one to encode multivariate probability distributions. Inference on such models typically involves answering inference queries over a Bayesian network, that is, a directed acyclic graph (DAG) paired with an associated joint probability distribution. Computing inference queries is the process of reducing one probability distribution (typically via *conditioning* and *marginalizing*) into another simplified probability distribution from which we can draw samples, yielding a most probable configuration of the random variables involved.

For instance, in the medical field, we may have data about a large number of patients (such as age, sex, and blood type) and we may wish to predict whether the patient has a specific disease or not. In this scenario, one would represent the data as a Bayesian network and query the model for the likelihood that a patient has the disease. This amounts to evaluating the Bayesian network having observed the patient's data, and sample the resulting reduced probability distribution. This serves as the inference process, and allows for informed decision making.

The models discussed in this thesis have significant applications and adaptions in medicine [1], [2], [3], [4], [5], behavioral and social sciences [6], [7], and statistics [8], [9]. Furthermore, significant theoretical research has been done on graphical models themselves [10] [11], [12]. An extensive survey of their genesis, development, and applications can be found in [13].

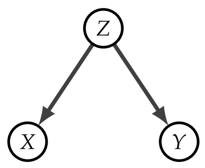
Naturally, especially as the use of statistical inference becomes increasingly widespread, it is in our best interest to explore methods for modeling and querying which allow us to do inference as quickly and efficiently as possible. This thesis therefore take advantage of the Bayesian networks' property of non-identifiability, meaning that there can be several networks which encode the same probability distribution. Such networks are called *Markov equivalent*. Markov equivalence informs us that a single inference query evaluated over structurally distinguishable networks can, in certain cases, yield in statistically indistinguishable results.

We therefore explore the following question: given that the same probability distribution can be encoded in different Bayesian networks, **can it be computationally beneficial to change the representation of the probability distribution in order to answer a sequence of queries more efficiently?**

Furthermore, in certain settings such as probabilistic programming, one does not evaluate an inference query over a probability distribution only once, but may instead wish to evaluate hundreds if not thousands or queries. Repeated evaluation of queries is a significant motivation for this work, as it justifies the overhead costs of transforming the networks.

1.2 Background

The family of models explored in this thesis are Bayesian networks over discrete probability distributions. The purpose of Bayesian networks is to encode information about vectors of random variables, namely the interdependencies satisfied by the vector's joint probability function. For example, consider a vector of three random binary variables (X, Y, Z). Suppose that the values of X and Y depend on Z, but X and Y are independent from one another given Z. These dependencies can be encoded in the graph in Figure 1.2.1, which is specified by its joint probabilisty function.

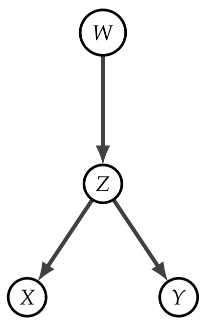


$$p(X, Y|Z) = p(X|Z) \cdot p(Y|Z)$$

Figure 1.2.1

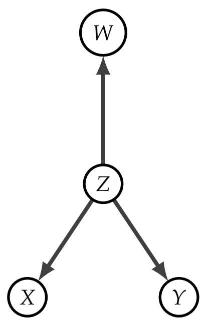
In the case of the directed models we explore in this thesis, there is not necessarily a unique graph which satisfies the dependency constraints of the joint probability function. A random vector and its constraints may be represented by a variety of different graph structures which ultimately describe the same underlying probability distribution. To understand how the complexity of answering a query depends on graph structure, consider the two models in Figure 1.2.2.

If we wish to answer a query about the random variable Y in G_1, for instance



G_1

$$p(X,Y,Z,W) = \\ p(X|Z) \cdot p(Y|Z) \cdot p(Z|W) \cdot p(W)$$



G_2

$$p(X,Y,Z,W) = \\ p(X|Z) \cdot p(Y|Z) \cdot p(W|Z) \cdot p(Z)$$

Figure 1.2.2

the probability that Y attains the value y, notated $p(Y = y)$, we must also compute the probabilities of each vertex it depends on either directly or indirectly: in this case, all of its **ancestors** (namely Z and W) in addition to Y itself. Alternatively, if we wish to answer a query $p(Y = y)$ graph G_2, we need only to compute Z and Y. Then the number of variables which must be computed is reduced when we consider G_2 instead of G_1. The set of vertices a query depends on can somewhat vary depending on the query, but the motivation for transforming a network remains the same.

In a context where answering queries about a single vertex may require us to compute arbitrarily many other vertices in the graph, we seek an equivalent graphical representation of our probability distribution which minimizes the number of vertices that must be computed when answering a query. Further, we wish to explore the optimal sequence of tranformations allowing us to answer a sequence of queries most efficiently.

1.3 Goal

The central concept of this thesis is the exploitation of non-identifiability of Bayesian networks (Markov equivalence) for faster inference. Given that multiple Bayesian networks may encode the same probability distribution, the goals of our research are as follows:

Given an inference query q and a Bayesian network B (which represents a single factorization of the probability distribution we wish to query), find a Markov equivalent Bayesian network B' which minimizes the number of variables that must be evaluated to answer the query. The purpose of the minimization is to reduce the cost of evaluating q in order to achieve faster inference, as well as evaluate the gained speed-up versus the cost of transforming the representation.

To achieve this, we first define the set of nodes which must be evaluated to answer q over a given network B, notated $\Delta(B,q)$. Then, we search for a Markov equivalent Bayesian network B' such that the size of $\Delta(B',q)$ is minimized. Finally, we compare the costs of computing the query over B to the costs of computing the query over the minimized graph B' plus the costs of identifying and transforming to B'. In short, the minimization problem is to determine the network Δ^* defined as

$$\Delta^* = argmin_{B' \in [B]} \Delta(B',q).$$

Subsequently, we consider how to identify optimal transformations for a known sequence of queries $\{q_0, q_1, ...q_n\}$ such that the overhead of transformation is most beneficial compared to the gained speedups by transforming.

1.4 Motivating Example

The following example demonstrates how answering a query on one graph can be sped up by asking the same query over a Markov equivalent graph with a different structure. Consider the DAG in Figure 1.4.1.

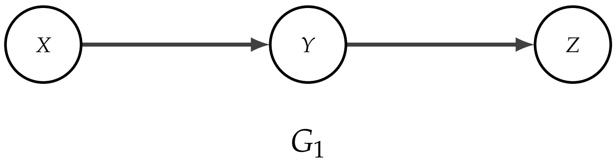


Figure 1.4.1

with conditional probabilities given by

$$\begin{aligned} &p(X=1)=0.2 && \\ &p(Y=1|X=0)=0.3 && p(Y=1|X=1)=0.4 \\ &p(Z=1|Y=0)=0.1 && p(Z=1|Y=1)=0.9 \end{aligned}$$

and the joint probability

$$p(X,Y,Z)=p(X)\cdot p(Y|X)\cdot p(Z|Y)$$

Here, if we wish to answer the query $p(Z|Y)$, our computation relies on all three random variables, since Z relies on Y and Y relies on X. However, we can find a Markov equivalent graph in which we can answer the same query while relying on fewer random variables. Consider the DAG in Figure 1.4.2.

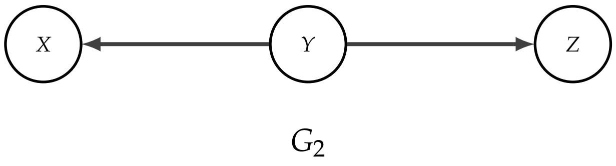


Figure 1.4.2

We compute the conditional probabilities of G_2 and claim that the model can equivalently describe our joint probability distribution from G_1, and therefore be used to answer the query $p(Z|Y)$. We then observe how the structure of the new model affects our computation of the query. By applying Bayes' Theorem, which states that $p(A|B)=\frac{p(B|A)p(A)}{p(B)}$, we can compute

$$p(X|Y)=\frac{p(Y|X)\cdot p(X)}{p(Y)}=\frac{p(Y|X)\cdot p(X)}{\sum_X p(Y|X)\cdot p(X)}$$

Using our pre-defined conditional probabilities,

$$\begin{aligned} p(Y=0) &= \sum_X p(Y=0|X) \cdot p(X) \\ &= 0.7 \cdot 0.8 + 0.6 \cdot 0.2 \\ &= 0.56 + 0.12 \\ &= 0.68 \end{aligned}$$

and

$$\begin{aligned} p(Y=1) &= \sum_X p(Y=1|X) \cdot p(X) \\ &= 0.3 \cdot 0.8 + 0.4 \cdot 0.2 \\ &= 0.24 + 0.08 \\ &= 0.32. \end{aligned}$$

which allow us to compute

$$p(X=1|Y=0) = \frac{p(Y=0|X=1) \cdot p(X=1)}{p(Y=0)} = \frac{0.6 \cdot 0.2}{0.68} \approx 0.176$$

and

$$p(X=1|Y=1) = \frac{p(Y=1|X=1) \cdot p(X=1)}{p(Y=1)} = \frac{0.4 \cdot 0.2}{0.32} = 0.25$$

Then, since X and Y do not depend on Z, the conditional probability of Z remains unchanged. Therefore, model G_2 is an equivalent model to G_1. If we answer the query $p(Z|Y)$ on model G_2, we do not need to consider X, since Y (and consequently Z) no longer depends on X.

Through this example, we see that it may be possible to find an equivalent graph to our original model, and to use the new graph structure to answer certain queries more efficiently. It is important to note that the efficiency of the new model for answering a query relies both on the conditional probabilites we are interested in, as well as the content of our query (as opposed to an arbitrary query).

1.5 Related work

Significant work has been done on individual aspects of the subject, though this thesis pioneers in using them together for fast inference.[10] give a framework for understanding how the same probability distribution can be encoded into two distinct graphs, as well as present a simplified criterion for when two graphs describe indistuingishable distributions. [14] present a survey-style overview of Markov Equivalent graph structures, reduction of graph structures to simplified representations of equivalent distributions, as well as providing a strong background of relevant graph theory and probability preliminaries. [11] explores parameters of equivalent networks, presents a procedure for altering graphs without changing the described probability distribution, and quantifies the complexity of several such manipulations. [12] make similar headway by exploring reductions of graphs to representative form, as well as describing procedures by which one can alter the structure.

Adjacently, [15] confronts the same goal of faster inference of queries using a different method in a modified setting. His task focuses on eliminating unnecessary nodes from the computations directly, rather than by reversing edges.

1.6 Outline

In Chapter 2, we introduce preliminary concepts from graph theory and Bayesian statistics to robustly define our models and to aid our understanding of graph manipulations and the process of querying over a graph. Chapter 3 details the problem and proposed querying method, then aims to quantify the computational costs of answering an arbitrary query over a graph by this method. Chapter 4 builds context for understanding Markov equivalence between graphs. In Chapter 5, we examine the circumstances under which a Markov equivalence can be utilized to answer a query more efficiently, and then quantify the computational costs of finding a suitable Markov equivalent graph for faster inference. Finally, in Chapter 6, we compare the quantities explored in Chapters 3 and 5 to determine whether our proposed algorithm indeed increases inference speed, and if so, where it is effective.

Chapter 2

Preliminaries

In this chapter we introduce preliminary concepts from graph theory and Bayesian statistics. These concepts will allow us to robustly define our models and problem statements, give us critial background for evaluating queries and exploring Markov Equivalence, and serve as a basis for the content of the rest of the thesis.

2.1 Foundations of Graph Theory

Definition 2.1.1 (Graph (directed, undirected, mixed)). *A **graph** is defined as a pair $G = (V, E)$ in which V is a finite set of vertices, and $E \subset V \times V$ is a finite set of edges. Vertices (also sometimes called **nodes**) will generally be denoted by capital letters, i.e. X, Y, Z. G is called **undirected** if every edge in G is undirected, meaning that for each edge $(X, Y) \in E$ (denoting an edge from vertex X to vertex Y), the edge (Y, X) is also in E, for $X \neq Y$. Otherwise, G is called **directed**. We denote such undirected edges by $\{X, Y\}$. A **mixed graph** (also called a hybrid graph) is a graph G which contains both directed and undirected edges.*

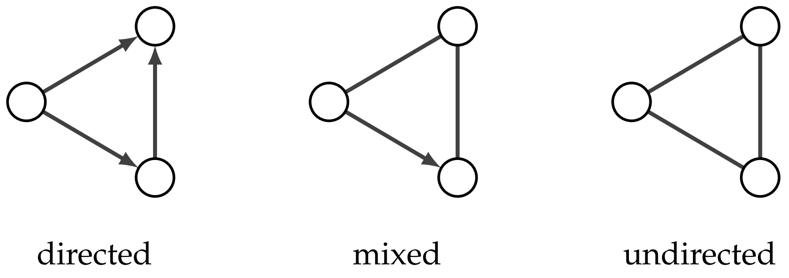


Figure 2.1.1

Definition 2.1.2 (Neighbor, adjacent). *Two vertices $X, Y \in V$ in a graph $G = (V, E)$ are called **adjacent** if there is an edge between them, either (X,Y), (Y,X), or X,Y. X and Y are also called **neighbors** in this scenario.*

Definition 2.1.3 (Route). *A **route** in a graph $G = (V, E)$ is a sequence $X_1, X_2, ..., X_k$ of vertices in V such that there is an edge connecting X_i to X_{i+1} (independent of the direction of the edge), for $i = 1, ..., k-1, k \geq 1$. The integer k is the length of the route.*

Definition 2.1.4 (Path, directed cycle). *A (directed) **path** in a directed graph $G = (V, E)$ is a route where vertices X_i and X_{i+1} are connected by a directed edge (X_i, X_{i+1}). A directed path which begins and ends at the same vertex is called a **directed cycle**.*

Remark. The distinction between a path and a route will be significant in later chapters. One should retain that a path is a directed sequence while a route is undirected.

Definition 2.1.5 (Parent, ancestor). *Given two vertices $X, Y \in V$ in a directed graph $G = (V, E)$, Y is called a **parent** of X if there is an edge $(Y, X) \in E$. The set of parents of a vertex X in a graph is denoted $\prod_X^G$. Likewise, X is called a **child** of Y. The set of **ancestors** of X, denoted $\alpha(X)$, is the set of all vertices Y such that there exists a directed path from Y to X, but no path from X to Y. In the context of this thesis, this is the set of vertices upon which a vertex X depends, either directly or indirectly. Likewise, if Y is an ancestor of X, then X is a **descendant** of Y.*

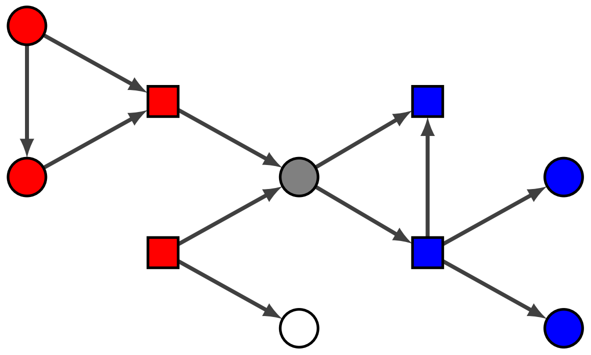


Figure 2.1.2: Red vertrices are ancestors of the gray vertex (squares denoting parents), and blue vertices are descendants of it (squares denoting children). The white vertex is neither an ancestor nor a descendent of the gray node.

Remark. $\prod_X^G \subseteq \alpha(X)$. Likewise, the set of children of X is a subset of the set of descendents of X.

Definition 2.1.6 (Chain graph, directed acyclic graph (DAG)). *A mixed graph $G = (V, E)$ is called a **chain graph** if it contains no directed cycles. A **directed acyclic graph** is a chain graph which is directed. This is abbreviated as DAG.*

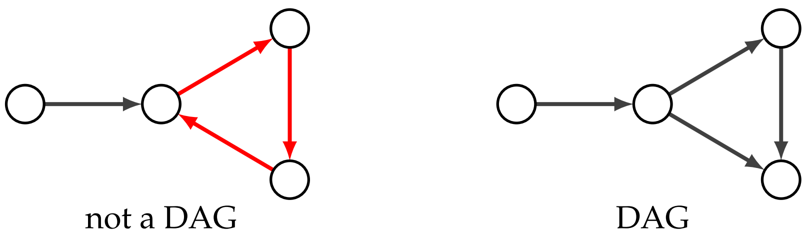


Figure 2.1.3: A directed cycle shown in red.

Remark. Every undirected graph is a chain graph.

2.2 Foundations of Bayesian Statistics

Definition 2.2.1 (Bayesian Network [16]). *A **Bayesian network** is a pair $B = (G, P)$ where $G = (V, E)$ is a DAG and P is a joint probability distribution defined on a set of random variables $\mathcal{X}$. That is, it is a DAG and an associated joint probability distribution.*

Vertices in G have a one-to-one correspondence to the random variables in $\mathcal{X}$ associated with P, meaning the set of vertices V represents both the vertices of G and the random variables of the joint probability distribution P.

P factorizes on the sets $\{X \cup \prod_X^G | X \in G\}$. That is, for each $X \in G$ there is a factor in P that depends on X and $\prod_X^G$, namely $P_X(X|\prod_X^G)$.

Example 2.2.1 (Simple Bayesian Network). The DAG in Figure 2.2.1 paired with the joint probability distribution $p(X_1 = x_1, X_2 = x_2) = p(X_1 = x_1) \cdot p(X_2 = x_2 | X_1 = x_1)$ is a Bayesian network.

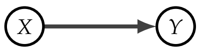


Figure 2.2.1

Remark. In the context of this thesis, we only consider graphs G which are DAGs unless explicitly stated otherwise. Further, for our purposes, all such DAGs are directed graphical models with associated probability distributions. Therefore we will refer to DAGs $G = (V, E)$ and Bayesian networks interchangably, with necessary details clarified in context.

Definition 2.2.2 (Sample space, event [17]). *Given a random experiment (an experiment whose outcome depends on chance), the* ***sample space*** *of the experiment is the set of all possible outcomes. Each subset of a sample space is called an* ***event****, that is, an outcome or a collection of outcomes of the random experiment. Therefore, event $\subseteq$ sample space.*

Example 2.2.2. In the context of Bayesian networks, we are interested in possible values of random variables which are represented by vertices. As a simple example, suppose we have the random binary variables X and Y. Then the sample space of X is $\{0,1\}$, and likewise for Y. Generally, we will be interested in answering questions such as the probability that the random variable X takes on a value $x \in \{0,1\}$; these queries are written $p(X=0)$, $p(X=1)$, or more generally, $p(X=x)$. Some possible queries one might encounter include the following:

- $p(X=x, Y=y)$, the probability that $X=x$ and $Y=y$ simultaneously, computed $p(X=x) \cdot p(Y=y)$,
- $p(X=x|Y=y)$, the probability that $X=x$ given that we have observed the condition $Y=y$.

Let the marginal probabilities of these random vectors be given as follows:

$$\begin{aligned} p(X=1) &= \frac{3}{10} & p(X=0) &= \frac{7}{10} \\ p(Y=0) &= \frac{6}{10} & p(Y=1) &= \frac{4}{10} \end{aligned}$$

where for X and Y binary variables, $p(X=1) = 1 - p(X=0)$ since $\sum_{x\in\{0,1\}} p(X=x) = 1$. This joint probability function $p : \{0,1\} \times \{0,1\} \to [0,1]$ can also be encoded as a table:

X	1	1	0	0
Y	1	0	1	0
$p(X,Y)$	$\frac{3}{25}$	$\frac{21}{100}$	$\frac{7}{25}$	$\frac{21}{50}$

So, the joint probability function $p(X,Y)$ is not a value, but a function of the likelihood of potential values from the sample space of Z and Y. Then, we can answer the query $p(X=1,Y=0)$ by reading off the table: $\frac{3}{25}$.

Next, suppose we have observed the condition $Y=0$. Then, we can *condition* the distribution, meaning we update the probability function such that it reflects our observation. We do so by looking at the scenarios in which $Y=0$:

$$p(X=0|Y=0)=\frac{21}{50} \qquad \text{and} p(X=1|Y=0)=\frac{21}{100}.$$

Then, using the fact that the probabilities should sum up to 1, we normalize:

$$\begin{aligned} p(X=0|Y=0) &= \frac{p(X=0|Y=0)}{p(X=0,Y=0)+p(X=1,Y=0)} \\ &= \frac{p(X=1,Y=0)}{p(Y=0)} \end{aligned}$$

and

$$\begin{aligned} p(X=1|Y=0) &= \frac{p(X=1,Y=0)}{p(X=0,Y=0)+p(X=1,Y=0)} \\ &= \frac{p(X=1,Y=0)}{p(Y=0)} \end{aligned}$$

Thus, with $p(Y=0)=\frac{63}{100}$, these give us

$X|Y=0$	1	0
$p(X|Y=0)$	$\frac{2}{3}$	$\frac{1}{3}$

Definition 2.2.3 (Probabilistic query). *A probabilistic query q on a graph $G=(V,E)$ is an inference question of the form $p(X_1,...,X_n|Y_1,...,Y_m)$ where $X_i, i \in [0,n] \in V$ and $Y_j, j \in [0,m] \in V$. X_i are the **targets** of q while Y_j are the **conditions** or **observations**.*

Theorem 2.2.1 (Bayes' Theorem [17]). *Given two random variables X and Y with $p(Y) \neq 0$*

$$p(X|Y)=\frac{p(Y|X)\cdot p(X)}{p(Y)}.$$

Remark. Bayes' Theorem will serve as a significant basis for many of the calculations necessary for transforming graphs in later chapters.

Definition 2.2.4 ((Statistical) independence [17]). *Let X and Y be random variables. Then X and Y are called* ***independent*** *whenever*

$$P(X,Y) = p(X)p(Y)$$

or equivalently if

$$p(X) = p(X|Y)$$

and vice versa.

Remark. 2.2.4 can be intuitively understood as follows: learning about Y has no effect on our knowledge concerning X and vice versa.

Definition 2.2.5 (Conditional independence [17]). *Let X, Y, and Z be random variables. Then X is said to be* ***conditionally independent*** *of Y given Z iff $p(X|Y,Z) = p(X|Z)$. Otherwise, they are said to be* ***conditionally dependent****.*

Remark. Definition 2.2.5 can be intuitively understood as follows: If we know about Z, then learning about Y has no effect on our expectation concerning X. Likewise, learning about X has no effect on our expectation concerning Y if we know about Z.

Remark. Definition 2.2.4 and Definition 2.2.5 also hold for disjoint sets A, B, and $C \subset V$ with p a joint probability distribution defined on V.

Example 2.2.3. Suppose Evin and Lisa take turns flipping a single coin which either results in heads (H) or tails (T). Naturally, one would assume that the probability of Lisa flipping heads is independent of the probability that Evin flips heads, that is,

$$p(Lisa = H|Evin = H) = p(Lisa = H),$$

since flipping a fair coin once does not dictate the outcome of a second flip. This means the two experiments are independent.

Now suppose we learn that the the coin is biased, calling this event Z. If the first coin flip results in heads, we would guess that the coin is more likely biased toward heads, and expect that the second coin flip to be heads. That is, the event of Evin flipping heads gives us information about the likelihood that Lisa will flip heads, given that we have observed Z. Then, whereas $p(Evin = H)$ and $p(Lisa = H)$ were previously independent from one another, observing event Z makes them depend on one another. Therefore the two events are conditionally dependent given Z:

$$p(Lisa = H|Z, Evin = H) \neq p(Lisa = H|Evin = H).$$

Dependence, independence, and conditional independence have an intuitive manifestation within graph structure, which follows from Definition 2.2.4.

Given a graph $G = (V, E)$ such as any of the graphs in Figure 2.2.2, if two nodes $X, Y \in V$ have an edge between them, either (X, Y) or (Y, X), then they are dependent variables by construction of our graphs.

If there exists a route between two nodes X and Y, but X and Y are not adjacent, then they are conditionally independent, where the conditions are the nodes on that route. This is because information about X can give us information about an intermediary node on the route and allow us to gain information about Y (or vice versa).

Finally, if two nodes are not connected by a route (disconnected) then they are independent, since information about X cannot give us information about Y under any circumstances.

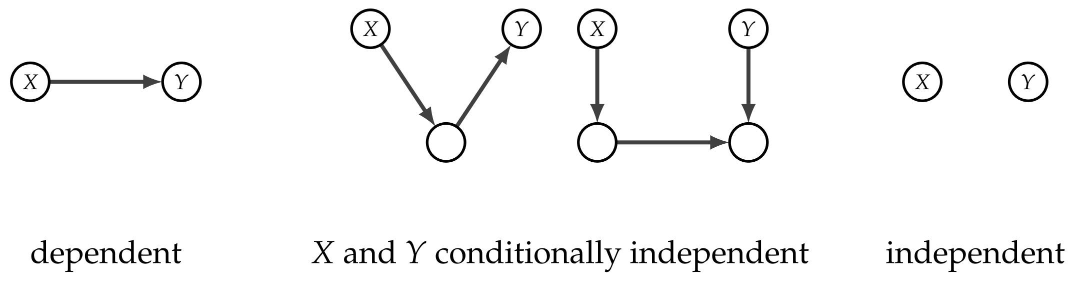


Figure 2.2.2

Chapter 3

Querying

3.1 Outline of the Task

The first task outlined by our goal is to determine the set of random variables that must be involved in the computation to answer a given query over G. This will serve as our cost function for the minimization which occurs in later chapters; the minimization will be to reduce the cost, meaning to find a Markov equivalent graph G' to G such that this number of vertices involved is minimized.

Naturally, before we consider the minimization itself, we begin by identifying such a set of vertices. This chapter will focus on identifying this set of vertices which must be computed to answer a query.

3.2 Structure of a Query

An inference query is a question which askes the probability of an event given some observations. Queries can take advantage of multiple observations and have multiple targets. For example, one can ask for the probability that the random variable X has takes on the value x given that we have observed $Y = y$. Let x be a value attained by the random variable X, and let y be a value attained by the variable Y. This is written $p(X = x|Y = y)$. The general form of a query is as follows:

Definition 3.2.1 (Query). *Given a graph $G = (V, E)$ with $X_1, ...X_n, Y_1, ...Y_n \in V$, the general form of a query q is $p(X_1, ..., X_n|Y_1, ...Y_n)$, meaning the probability of variables $X_1, ..., X_n$ given that we have conditioned on observed values attained by variables $Y_1, ..., Y_n$.*

Remark. Note that while a query on $G = (V, E)$ must have at least one target (otherwise there is nothing to compute), it does not necessarily need to include a condition. For example, $p(X = x)$ is a valid query for $X \in V$.

Scenario	Query	Interpretation	Example
Query a single target, single observation	$p(X=x\|Z=z)$	Probability that $X=x$ given that we have observed $Z=z$.	Probability that a patient has lung cancer given that they are a tobacco smoker.
Query with multiple targets, single observation	$p(X=x,Y=y\|Z=z)$	Probability that $X=x$ and $Y=y$ given that we have observed $Z=z$.	Probability that a patient has lung cancer and high blood pressure given that they are a tobacco smoker.
Query with a single target, multiple observations	$p(X=x\|W=w,Z=z)$	Probability that $X=x$ given that we have observed $W=w$ and $Z=z$.	Probability that a patient has lung cancer given that they are over 50 years of age and a tobacco smoker
Query with multiple targets, multiple observations	$p(X=x,Y=y\|W=w,Z=z)$	Probability that $X=x$ and $Y=y$ given that we have observed $W=w$ and $Z=z$.	Probability that a patient has lung cancer and high blood pressure given that they are over 50 years of age and a tobacco smoker.

Let G be a DAG and q a query. Define $\Delta(G,q)$ to be the number of nodes involved in the computation of q on G. The goal of this chapter is to find this value by determining the members of the set. Ultimately, we intend to find $\arg\min_{G' \in [G]}(\Delta(G,q))$. We call this minimized graph Δ^*.

3.3 Vertices Required to Compute a Query

In the following section, we determine which sets of vertices a query q on a DAG G depends on, notated $\Delta(G,q)$. First we consider queries with single targets and single conditions, then move on to scenarios with multiple conditions, before finally generalizing to scenarios with multiple targets and conditions. We will see that the structue of the DAG (namely, position of targets in G in relation to the position of conditions) plays a significant role in whether certain vertices must be considered to compute a query.

Determining $\Delta(G,q)$ for an arbitrary q will later allow us to quantify the speedups gained by transforming the graph: given a DAG G and a Markov-equivalent transformed DAG G', will compare the $\Delta(G,q)$ to $\Delta(G',q)$. If $\Delta(G',q)$ is a smaller set, then one can conclude that the transformation has reduced the cost of the query. Then the question becomes whether the overhead of transformation is worth the earned speedup. Once again, we will take advantage of the definition of conditional independence:

$$P(A|B) = \frac{P(A,B)}{P(B)}.$$

Lemma 3.3.1. *Computing a conditionless query $q = p(X_0, X_1, ..., X_n)$ over a DAG G depends on vertices $X_0, X_1, ..., X_n$ and all of their ancestors: $\alpha(X_0), \alpha(X_1), ..., \alpha(X_n)$.*

Proof. Let $q = p(X_0, X_1, ..., X_n)$ on a DAG G. Using the definition of a Bayesian network, we have that the joint probability distribution

$$p(X_0, X_1, ...X_n) = \prod_{i=0}^{n} p(X_i | \prod_{X_i}^{G}).$$

Therefore q indeed depends on all ancestors of the targets. □

Example 3.3.1. Let $q = p(X_1 = x_1)$ be a query on $G = (V, E)$ in Figure 3.3.1. The computation of q is as follows.

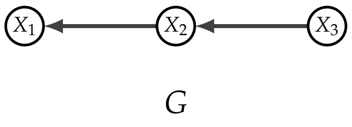


Figure 3.3.1

$$p(X_1 = x_1) \quad = \quad \sum_{x_2}\sum_{x_3} p(X_1 = x_1 | X_2 = x_2) \cdot p(X_2 = x_2 | X_3 = x_3) \cdot p(X_3 = x_3)$$

and therefore requires considering the target vertex X_1 and its ancestors, namely X_2 and X_3.

Theorem 3.3.2. *Let $q = p(X|Y)$ be a query on a DAG G. The the computation of q requires the following sets of vertices depending on the position of Y in G:*

- ***Special case*** *If X is conditionally independent of all ancestors of Y given Y, then q depends on X, $\alpha(X) \setminus \alpha(Y)$, and all nodes on a route from X to Y excluding Y itself.*

- ***General case*** *Otherwise, in the case that* X *is* ***not*** *independent of* $\alpha(Y)$ *given* Y, q *depends on both* X *and* Y *in addition to the sets* $\alpha(X)$ *and* $\alpha(Y)$, *as well as all nodes on a route from* X *to* Y.

Lemma 3.3.3. *The conditions for the cases in 3.3.2 can be rephrased as follows:*

- *The condition* Y *is in the* **special case** *if, when* Y *is removed from* G, *the resulting graph* G' *consists of at least two disconnected subgraphs* G'_1 *and* G'_2 *such that* $X \in G'_1$ *and all vertices in* $\alpha(Y) \in G'_2$. *That is, removing* Y *disconnects* $\alpha(Y)$ *from* X.

- *The condition* Y *is in the* **general case** *otherwise. That is, whenever the removal of* Y *does not result in a disconnected graph such that* X *and all vertices in* $\alpha(Y)$ *are in separate subgraphs.*

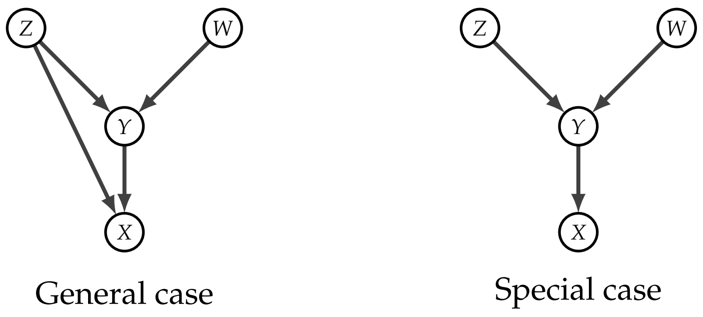


Figure 3.3.2: Example of one DAG in the general case and one in the special case.

When considering queries with more than one condition, one cannot classify the DAG G as being either in the special case or general case, but rather, must classify each condition Y_j into the cases. This is because a target X might be independent of $\alpha(Y_0)$ given a condition Y_0, but might still be dependent on $\alpha(Y_1)$ given Y_1. Then, Y_0 would be in the special case, whereas Y_1 would be in the general case. Resultingly, the query would not depend on Y_0 and its ancestors, but would still depend on Y_1 and its ancestors.

Furthermore, when considering multiple targets $X_0, X_1, ...X_i$, one must check whether a set of ancestors $\alpha(Y_j)$ is independent of all vertices $X_i, i \in [0, m]$ given Y_j. Otherwise, if one of the targets was dependent on an ancestor of Y_j, the query q would depend on that ancestor. From this, we arrive at Theorem 3.3.4.

Theorem 3.3.4. *Let $q = p(X_0, X_1, ..., X_m | Y_0, Y_1, ..., Y_n)$ be a query on a DAG G. Then the computation of q requires the following sets of vertices depending on the structure of G:*

1. *q always depends on X_i ,*
2. *q always depends on all vertices along a route from each vertex $X_i, i \in [0, m]$ to each vertex $Y_j, j \in [0, n]$, not including Y_j itself,*
3. *For each $Y_j, j \in [0, n]$ in the general case, that is, whenever all targets $X_i, i \in [0, m]$ are dependent on $\alpha(Y_j)$ given Y_j, q depends on vertices Y_j and $\alpha(Y_j)$.*

Therefore, if Y_j is in the special case, q does not depend on Y_j nor $\alpha(Y_j)$.

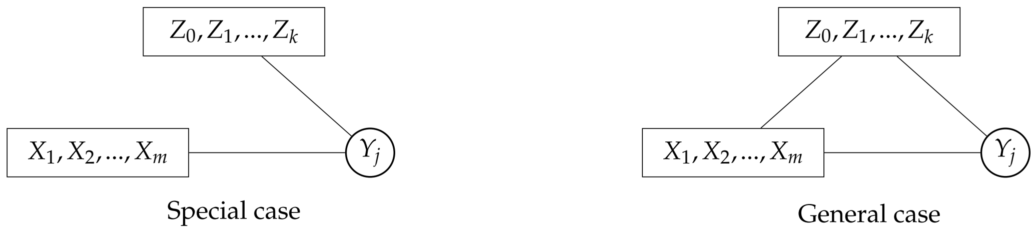


Figure 3.3.3: Graphical structure of the two cases: in the general case, X_is are not conditionally independent from all elements $Z_k \in \alpha(Y_j)$ given Y_j. In the special case, they are.

Example 3.3.2. Here, we apply Theorem 3.3.4 to determine the set of vertices a query q over the DAG G in Figure 3.3.4 depends on. An explicit calcuation to verify this is not given here; instead, we simply identify the set. Such calculations will take place in later examples.

Suppose $q = p(X|Y_0, Y_1)$. Then, for each of the conditions Y_0 and Y_1, we determine whether it is in the general case of the special case. Immediately, we know that the query depends on at least the target X.

- Y_0 has only one ancestor, namely Z_0. Z_0 is conditionally independent of X given Y_0. Therefore, the condition Y_0 is in the special case. Resultingly, q does not depend on Y_0 nor Z_0.
- Y_1 has two ancestors, namely W_0 and W_1. X is not conditionally independent from W_0 given Y_1, since we can find a route to W_0 which does not pass through Y_1. Therefore, Y_1 is in the general case, and we conclude that the query still additionally depends on W_0, W_1 and Y_1

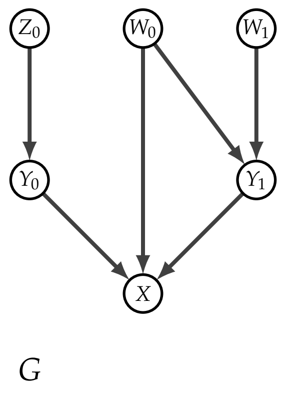


Figure 3.3.4

We conclude that $\Delta(G,q) = \{X, Y_1, W_0, W_1\}$.

The following examples verify which vertices are in the set $\Delta(G,q)$ outlined in Theorem 3.3.4. First we present several computations in the general case, then computations in the special case. Finally, we show how two different queries over the same DAG G can fall into each case.

Example 3.3.3. General Case. Let $q = p(X_3 = x_3, X_4 = x_4)$ be a query on $G = (V,E)$ in Figure 3.3.5. The computation of q is as follows.

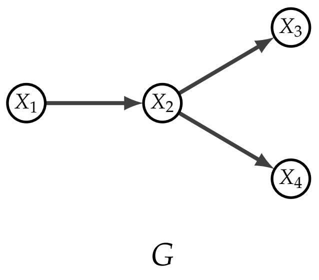


Figure 3.3.5

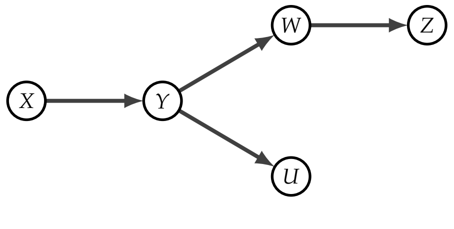
$$\begin{aligned}
q &= p(X_3 = x_3, X_4 = x_4) \\
&= \sum_{x_2}\sum_{x_1} p(X_3 = x_3, X_4 = x_4 | X_2 = x_2) \cdot p(X_2 = x_2 | X_1 = x_1) \cdot p(X_1 = x_1) \quad (1) \\
&= \sum_{x_2}\sum_{x_1} p(X_3 = x_3 | X_2 = x_2) p(X_4 = x_4 | X_2 = x_2) \cdot p(X_2 = x_2 | X_1 = x_1) \cdot p(X_1 = x_1) \quad (2) \\
&= \sum_{x_2}\sum_{x_1} p(X_3 = x_3 | X_2 = x_2) \cdot p(X_4 = x_4 | X_2 = x_2) \cdot p(X_1 = x_1, X_2 = x_2) \quad (3)
\end{aligned}$$

where equality (1) is by construction of the Bayesian network and equality (2) comes from the definition of conditional probability since X_3 and X_4 are conditionally independent. Equality (3) comes from the following application of conditional probability:

$$p(X_2 = x_2 | X_1 = x_1) \cdot p(X_1 = x_1) \;=\; p(X_1 = x_1, X_2 = x_2).$$

The query q depends on the targets X_4 and X_3 and their parents, X_2 and X_1.

Example 3.3.4. General case. The following example follows similar structure to Example 3.3.3 and uses real values for the conditional probabilities. Consider the graph $G = (V, E)$ from 3.3.6, where a vertex V takes a binary value $v \in [0,1]$, and suppose we are given a query $q = p(Y = 1 | Z = 1)$.



G

Figure 3.3.6

Let the conditional probabilities be given by

$$
\begin{array}{ll}
p(X=1)=0.2 & \\
p(Y=1|X=0)=0.6 & p(Y=1|X=1)=0.4 \\
p(W=1|Y=0)=0.4 & p(W=1|Y=1)=0.3 \\
p(Z=1|W=0)=0.7 & p(Z=1|W=1)=0.3 \\
p(U=1|Y=0)=0.8 & p(U=1|Y=1)=0.4
\end{array}
$$

Then, the joint probability is given by

$$p(X,Y,W,U,Z)=p(X)\cdot p(Y|X)\cdot p(U|Y)\cdot p(W|Y)\cdot p(Z|W).$$

The aim of this example is to demonstrate that our query depends on the set of ancestors $\alpha(Y)$ and the nodes on route from Z to Y, in this case, the route $Y \to W \to Z$. For this computation, we set $Z = 1$, and recall that since we are in a binary setting, for a vertex V we have $p(V = 0) = 1 - p(V = 1)$. Then, to determine $p(Y=1|Z=1)$, we first compute $p(Y=1)$:

$$
\begin{aligned}
p(Y=1) &= \sum_X p(Y=0|x)\cdot p(x) \\
&= 0.6\cdot 0.8+0.4\cdot 0.2 \\
&= 0.48+0.08 \\
&= 0.56 \\
P(Y=0) &= 1-p(Y=1)=0.44
\end{aligned}
$$

By similar calculation and plugging in the values $p(Y=1)$ and $p(Y=0)$, we obtain that $p(W=0)=\sum_Y p(W=0|Y)p(Y)\approx 0.65$ and $P(W=1)\approx 0.34$. Likewise, we compute $p(Z=1)\approx 0.56$ and $p(Z=0)\approx 0{,}43$.

From here, we determine how the observation $Z = 1$ affects the likelihood of values of W and consequently Y. Bayes' theorem allows us to do the calcultion using quantities specified in the conditional probabilities:

$$p(W=1|Z=1) = \frac{p(Z=1|W=1)\cdot p(W=1)}{p(Z=1)} \approx \frac{0.3\cdot 0.34}{0.56} \approx 0.18.$$

Subsequently $p(W=0|Z=1) \approx 0.81$. Now that we see how W is affected by the observation that $Z=1$, we can move upward in our route to determine who Y is affected by. Continuing with our condition that $Z=1$ and applying Bayes' theorem again,

$$p(Y=1|W=1) \;=\; \frac{p(W=1|Y=1)\cdot p(Y=1)}{p(W=1)} \;\approx\; \frac{0.18\cdot 0.44}{0.18} \;\approx\; 0.48.$$

Finally,

$$\begin{aligned} p(Y=1|Z=1) &= p(Y=1|W=0)\cdot p(W=0) + p(Y=1|W=1)\cdot p(W=1) \\ &\approx (0.59\cdot 0.81) + (0.48\cdot 0.18) \\ &\approx 0.17. \end{aligned}$$

Thus, by tracing the effects of the observation $Z=1$ on a route to $Y=1$ and using the ancestors $\alpha(Y)$, we have answered the query $q = p(Y=1|Z=1)$. Notice that this computation did not involve the vertex U, which is neither an ancestor of Y nor along the route between Y and Z.

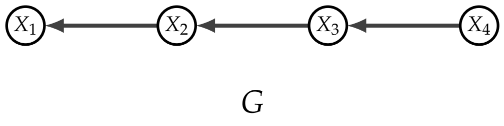


Figure 3.3.7

Example 3.3.5. Special Case. Let $q = p(X_4 = x_4|X_2 = x_2)$ be a query on G in Figure 3.3.7. Then the query only involves vertices X_4, X_3, and X_2, so there is no need to consider vertex X_1. This is because X_4 is conditionally dependent on $\alpha(X_2) = X_1$.

$$p(X_4=x_4|X_2=x_2) = \sum_{x_3} p(X_4=x_4|X_3=x_3)\cdot p(X_3=x_3|X_2=x_2).$$

Example 3.3.6. This example compares two queries over the same graph G in Figure 3.3.8. Let $q_1 = p(X_2=x_2|X_3=x_3)$ and let $q_2 = p(X_3=x_3|X_2=x_2)$.

Note that X_2 is an ancestor of X_3, and trivially X_2 is not conditionally indepen-

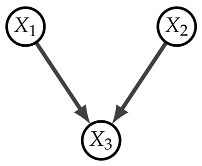


Figure 3.3.8

dent from X_2. Therefore q_1 is in the general case of Theorem 3.3.4, since the target of q_1 (namely X_2) is not disjoint from the ancestors of the condition, $\alpha(X_3)$.The computation of q_1 therefore requires the target X_2, the ancestors $\alpha(X_2)$, the condition X_3, and $\alpha(X_3)$. These sets together involve X_1, X_2, and X_3. This dependence is verified:

$$\begin{aligned}
q_1 &= p(X_2 = x_2 | X_3 = x_3) \\
&= \frac{p(X_2 = x_2, X_3 = x_3)}{p(X_3 = x_3)} \\
&= \frac{\sum_{x_1} p(X_1 = x_1) \cdot p(X_2 = x_2) \cdot p(X_3 = x_3 | X_1 = x_1, X_2 = x_2)}{\sum_{x_1} \sum_{x_2} p(X_1 = x_1) \cdot p(X_2 = x_2) \cdot p(X_3 = x_3 | X_1 = x_1, X_2 = x_2).}
\end{aligned}$$

Now we consider q_2. In this case, the conditional variable X_2 has no ancestors, and therefore we do not need to consider its distribution, since we are in the special case of Theorem 3.3.4. We show below that q_2 only relies only on X_3 and X_1.

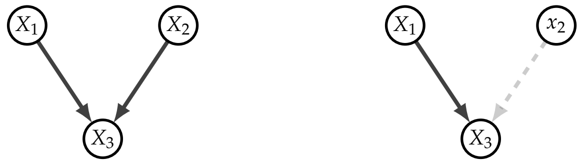


Figure 3.3.9: X_2 effectively removed from the model by conditioning $X_2 = x_2$.

Intuitively, setting $X_2 = x_2$ effectively removed the variable from the model, since we are conditioning other variables on this observation. Once the other variables are adjusted, X_2 no longer plays a role since it already has an established value and does not have parents that must be considered. That is, we are setting $p(X_3 = x_3 | X_1 = x_1) = p(X_3 = x_3' | X_1 = x_1', X_2 = x_2)$ for all x_1', x_2'. Thus:

$$\begin{aligned}
q_2 &= p(X_3 = x_3 | X_2 = x_2) \\
&= \frac{p(X_3 = x_3, X_2 = x_2)}{p(X_2 = x_2)} \\
&= p(X_3 = x_3) \\
&= \sum_{x_1} p(X_3 = x_3 | X_1 = x_1, X_2 = x_2) \cdot p(X_1 = x_1) \\
&= \sum_{x_1} p(X_3 = x_3 | X_1 = x_1) \cdot p(X_1 = x_1).
\end{aligned}$$

Therefore, q_2 relies only on X_3 and X_1 as described in the special case, whereas q_2 relies on X_2 as well, as described in the general case.

3.4 Complexity of Computing a Query

Once the number of nodes the computation a query depends on has been established, we can consider the actual complexity of the computations. The goal of this section is to determine the arithmetic cost of the query, that is, the number of operations (+, *) necessary to compute a query. We will calculate this arithmetic cost in terms of how it scales with the length of the query.

Example 3.4.1. Let G be the DAG in fig. 3.4.1 with binary variables X, Y, and Z

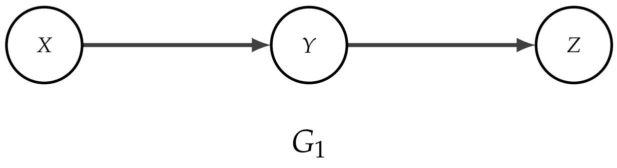


Figure 3.4.1

with conditional probabilities given by

$$\begin{aligned}
&p(X = 1) = 0.2 && \\
&p(Y = 1 | X = 0) = 0.3 && p(Y = 1 | X = 1) = 0.4 \\
&p(Z = 1 | Y = 0) = 0.1 && p(Z = 1 | Y = 1) = 0.9
\end{aligned}$$

and suppose we wish to compute the query $p(Z|Y)$. Then, the table of the query will need to represent the values achieved by Z in conjunction with possible results of the conditional, Y, and will have the following form:

Y	1	1	0	0
Z	1	0	1	0
$p(Z\|Y)$	0.9	0.1	0.1	0.9

Then, since the number of conditions is 1 (namely Y) and the number of targets is 1 (namely Z) the number of entries in the table is $2^{1+1} = 2^2$. This is because we must know how our target's value responds to values taken by the condition of q, and the 2 comes from the fact that we are assuming binary variables X, Y and Z.

Theorem 3.4.1. *Let G be a DAG with binary variables and let $q = p(X_0, X_1, ..., X_m | Y_0, Y_1, ..., Y_n)$ be a query on the vertices of G. Then, the data table for the resulting distribution of q will have $2^{n+m}(n+m)$-many entries. In particular, each row will have 2^{m+n} entries, and there will be $(m+n)$ rows.*

X_1	1	1	...	0
X_2	1	1	...	0
...				
X_m	1	1	...	0
Y_1	1	1	...	0
Y_1	1	1	...	0
...				
Y_n	1	0	...	0
q	q_1	q_2	...	$q_{2^{n+m}}$

Remark. If G instead has categorical variables which can each take on c-many possible values, then the number of entries in the data table for $q = p(X_0, X_1, ..., X_m | Y_0, Y_1, ..., Y_n)$ will be $c^{n+m}(m+n)$.

3.4.1 Cost of Marginalizing

Example 3.4.2. Suppose we are given a data table of three variables such as the following:

X	1	1	1	1	0	0	0	0
Y	1	1	0	0	1	1	0	0
Y	1	0	1	0	1	0	1	0
$p(X,Y,Z)$	$\frac{1}{12}$	$\frac{7}{32}$	$\frac{1}{12}$	$\frac{7}{96}$	$\frac{1}{4}$	$\frac{1}{32}$	$\frac{1}{4}$	$\frac{1}{96}$

As expected, this binary table of $p(X,Y,Z)$ his 2^3 entries since $m = 3, n = 0$, for m corresponding to the number of targets $X_0, ...X_m$ and n corresponding to conditions $Y_0, ..., Y_n$. If, for example, we wish to reduce the table to a specialized distribution in order to answer the query $p(Z,X)$, we can do so by *marginalizing* variables which the query doesn't depepend on. In this case, we disregard the values of Y by summing over the probabilities where $Y = 0$ and $Y = 1$:

X	1	1	0	0
Z	1	0	1	0
$p(X,Y=1,Z)$	$\frac{1}{12}$	$\frac{7}{32}$	$\frac{1}{4}$	$\frac{1}{32}$
$+p(X,Y=0,Z)$	$\frac{1}{12}$	$\frac{7}{96}$	$\frac{1}{4}$	$\frac{1}{24}$
$=p(X,Z)$	$\frac{1}{6}$	$\frac{7}{24}$	$\frac{1}{2}$	$\frac{1}{24}$

By summing over the possible values of Y, we are able to answer the query $p(X,Z)$. Notice that marginalizing had the following costs:

$$\text{For } Y = 0: \quad (\frac{8}{2}) - 1 = \frac{\text{number of entries in a row}}{\text{number of values attainable by each variable}} - 1 \quad = \frac{2^{n+m}}{2} - 1$$

$$\text{For } Y = 1: \quad \frac{2^{n+m}}{2} - 1$$

which together give us $2(\frac{2^{n+m}}{2} - 1)$, where the minus 1 comes from the fact that adding n numbers together only requires $n - 1$ addition operations. Marginalizing a single variable results in a table with $(\frac{n+m}{2})(n + m - 1)$ entries.

Theorem 3.4.2. *Let m be the number of targets in a query q and n be the number of conditions. Marginalizing a single variable X_i in a distribution $p(X_0, ..., X_n | Y_0, ..., Y_n)$ requires $2(\frac{2^{n+m}}{2} - 1)$ arithmetic operations, and results in a data table with $\frac{n+m}{2}(n + m - 1)$-many entries.*

3.4.2 Cost of Conditioning

Example 3.4.3. Suppose that, continuing example 3.4.2, we have the following probability table and we wish to further compute $q = p(Z|X=1)$.

X	1	1	0	0
Z	1	0	1	0
$p(Z\|X)$	$\frac{1}{6}$	$\frac{7}{24}$	$\frac{1}{2}$	$\frac{1}{24}$

Then, we must condition on the case $X=1$. From the above table, we can read

$$p(X=1, Z=1) = \frac{1}{6}, \qquad p(X=1, Z=0) = \frac{7}{24}$$

Notice that these values have not been normalized, and therefore we cannot yet use them as probabilities; they must be normalized in order represent the probabilities of observing $Z=1$ or $Z=0$ given $X=1$. Normalization is done as follows:

$$\begin{aligned} p(Z=1|X=1) &= \frac{p(X=1,Z=1)}{p(X=1,Z=1)+p(X=1,Z=0)} \\ &= \frac{p(X=1,Z=0)}{p(X=1)} \end{aligned} \qquad (1)$$

and

$$\begin{aligned} p(Z=0|X=1) &= \frac{p(X=1,Z=0)}{p(X=1,Z=0)+p(X=1,Z=1)} \\ &= \frac{p(X=1|Z=1)}{p(X=1)} \end{aligned} \qquad (2)$$

which results in the following data table:

$Z\|X=1$	1	0
$p(Z\|X=1)$	$\frac{4}{11}$	$\frac{7}{11}$

The above normalization equation (1) generalizes as follows:

Given a data table $p(X_0, X_1, ...X_m, Y_0, Y_1, ..., Y_n)$ and a condition $X_0 = 1$ without loss of generality, conditioning on $X_0 = 1$ requires us to evaluate

$$p(X_1,...,X_m,Y_0,Y_1,...,Y_n|X_0 = 1)$$

$$= \frac{p(X_0,X_1,...,X_m,Y_0,Y_1,...,Y_n)}{p(X_0 = 1,X_1 = 0,...Y_n = 0) + ... + p(X_0 = 1,X_1 = 1,...,Y_n = 1)}$$

where the denominator covers all possible combinations of values for which $X_0 = 1$. This denominator amounts to exactly half the the values in a given row in our binary setting. Then, since a given row has 2^{n+m}-many values, and half a row has $\frac{2^{n+m}}{2}$ values, we require $\frac{2^{n+m}}{2} - 1$ operators. The numerator is simply a single value from our data table, and therefore does not require an operator. Then, adding the fact that we must divide the numerator's single value with the denominators single value (after evaluation), the total number of operators to condition becomes $\frac{2^{n+m}}{2} - 1 + 1 = \frac{2^{n+m}}{2} = 2^{n+m-1}$.

Since, in the binary case, the above opertons must be done twice (for example solving both (1) and (2) above), we multiply the number of operations by 2 to determine the total arithmetic cost of conditioning a variable: 2^{n+m}.

The resulting data table has $\frac{n+m}{2}$ entries in each row, and $(n + m - 1)$ rows, resulting in $\frac{n+m}{2}(n + m - 1)$-many entries.

Theorem 3.4.3. *Let m be the number of targets in a query q and n be the number of conditions. Conditioning on a single variable $X_i = x$ in a distribution $p(X_0,...,X_n|Y_0,...,Y_n)$ requires 2^{n+m} arithmetic operations, and results in a data table with $\frac{n+m}{2}(n + m - 1)$ entries.*

3.4.3 Total Arithmetic Cost

ANDREAS this is the best guess that I mentioned but I haven't proved it and I still feel quite uncertain! :

In the first section of this chapter, we established the elements of the set $\Delta(G,q)$ for a DAG G and a query q. When calculating q, we do not need to condsider elements from outside of this set in the computation. Then, we determine which variables need to be marginalized and which need to be conditioned in order to quantify the overall number of arithmetic operations necessary to answer q. Recall that each of the elements in $\Delta(G,q)$ is at least one of the following:

1. A target X_i

2. A condition Y_j,
3. On the path between a target X_i and a condition Y_j,
4. A parent of X_i or of Y_j.

Remark. Of course, elements in the above categories are not necessarily *in* $\Delta(G,q)$ as demonstrated by the Special case, but all elements in $\Delta(G,q)$ are in those categories.

Let Z be a vertex which is in the third category, such as an ancestor of Y. Then, Z itself does not need to be conditioned upon, since the query does not ask about the effects of the conditions $Y_0,...,Y_n$ on Z. Therefore, we only need to marginalize Z in order to create a more specialized distribution. This holds for all nodes which are *only* in the third category.

Next consider a condition vertex Y_j. Naturally, we must compute the probabilities of the condition, $p(Y_j = y), y \in \{0,1\}$.

Now consider a vertex W on a path between X_i and Y_j. If we have observed the condition $Y_j = y$, we must calculate the effects of that condition on every vertex between Y_j and our targets X_i. Therefore, every intermediate vertex such as W must be conditioned. For example, if we have a joint probability distribution of the form

$$p(X,Y,Z,W) = p(X) \cdot p(Y|X) \cdot p(W|Y) \cdot p(Z|W),$$

and we wish to answery the query $p(Y|Z=1)$, we must compute $p(Z=1)$, $p(W|Z=1) = p(W')$, and $p(Y|W') = p(Y')$, and marginalize out Z. This is true for targets X_i themselves as well.

Chapter 4

Markov Equivalence

4.1 Goal

In order to reduce the cost of a query on a graph G, we will exploit the observation that two Bayesian networks with distinguishable structures can model the same probability distribution. In this case, the corresponding graphs are called *Markov equivalent*. Given a query on a graph G, we search for a Markov equivalent graph G' such that the cost of the query is minimized when considered on G' instead of G.

To do so, we must develop a robust understanding of Markov equivalence. This includes: how to identify whether two graphs are Markov equivalent, how a graph can be altered while remaining in the set of Markov equivalent graphs (called the *Markov equivalence class*), how computations of queries on a graph are altered when the graph structure is altered, and how to search for minimizing graphs.

In this chapter, we present definitions, theorems, and examples to establish a sufficient understanding of Markov equivalence for this purpose.

4.2 Basic Properties of Markov Equivalence

Definition 4.2.1 (Markov equivalence [10]). *Two directed acyclic graphs G and G' are* ***Markov equivalent*** *if for every Bayesian network $B = (G, \theta_G)$, there exists a Bayesian network $B' = (G', \theta_{G'})$ such that B and B' describe the same probability distribution, and vice versa. This is denoted $G \sim_M G'$.*

Definition 4.2.2 (Markov equivalence class [10]). *The set of all DAGs which are Markov equivalent to a DAG G is called the **Markov equivalence class**(MEC) of G, denoted $[G]$.*

Significant research has been done to characterize Markov equivalence classes ([11], [10], [12]). This research has identified several key features of a graphs which allow us to more easily identify reversible edges and construct MECs. The following definitions outline several properties of graphs (and edges within them) which we will use to determine whether two graphs lie in the same MEC. Furthermore, it will help us build an understanding of how a graph can be manipulated without altering which MEC it belongs to.

Definition 4.2.3 (Covered Edge [11]). *Given a DAG $G = (V, E)$, an edge $e = (X, Y) \in E$ is called **covered** in G iff $\prod_Y^G = \prod_X^G \cup X$. That is, (X, Y) is covered in G iff X and Y have identical parents in G with the exception that X is not a parent of itself.*

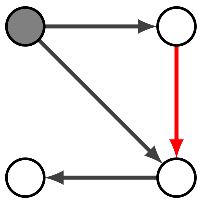
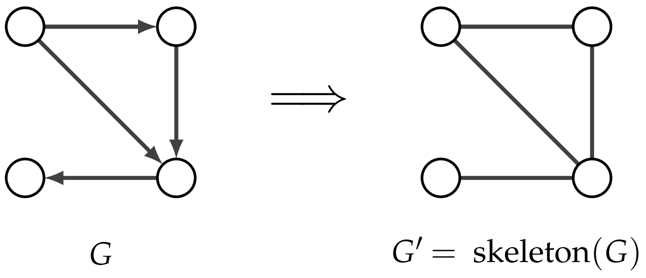


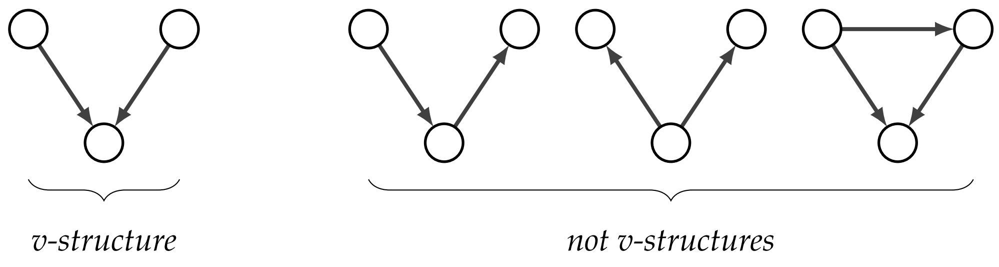
Figure 4.2.1

In Figure 4.2.1, for example, the red edge is the only covered edge, since the nodes it connects have a shared set of parents, namely, the single gray node.

Definition 4.2.4 (Skeleton). *Let $G = (V, E)$ be a directed graph. The undirected graph $G' = (V, E')$ which is constructed by removing the orientation of all edges in E is called the **skeleton** of G.*



Definition 4.2.5 (v-structure). *A set of three vertices X, Y, Z of a graph $G = (V, E)$ is called a **v-structure** (or immorality) iff $(X, Y) \in E$ and $(Z, Y) \in E$ but X and Z are not adjacent.*



Definition 4.2.6 (Irreversible edge [12]). *Let G be a DAG. A directed edge $(X, Y) \in G$ is called **irreversible** in G if changing (X, Y) to (Y, X) either creates or destroys a v-structure, or creates a directed cycle. An edge which is not irreversible is called **reversible**.*

Lemma 4.2.1 (Criterion for Markov equivalence [10]). *Two graphs are equivalent iff they have the same skeleton and v-structures.*

Proof. A sketch of the proof is provided, as the complete proof involves many definitions and details which are not presented in this paper. First, we use the definition of an **active route** (sometimes called active path) with respect to a set $\mathbb{Z}$: a route $[X_0, X_1, ... X_n]$ in which all nodes X_i are themselves **active**, meaning that (1) every middle node of a v-structure with respect to $\mathcal{Z}$ either is or has a descendant in $\mathcal{Z}$, and (2) every other node in the route is outside $\mathcal{Z}$.

The proof depends on two lemmas presented in the same paper: **Lemma 1** has the consequence that adjacency of nodes is invariant among equivalent DAGs. **Lemma 2** has the consequence that one can determine the presence or absence of v-structures by observing certain seperation properties of the graph alone, and vice versa. The two lemmas are unified with an inductive step to show that active paths in one DAG correspond to active paths in a Markov equivalent DAG. Furthermore, properties of active paths allow us to conclude that the DAGs must have the same skeleton and v-structures. □

Lemma 4.2.2 (Covered edges are exactly reversible edges [11]). *Let $G = (V, E)$ be a DAG, let $X, Y \in V$ and let $e = (X, Y) \in E$. Let $G' = (V, E')$ be the DAG constructed from G by reversing the edge (X, Y) to (Y, X). Then G and G' are equivalent iff e is a covered edge in G.*

Proof. Let $G = (V, E)$ be a DAG. A sketch of the proof is as follows (see [11] for the full proof):

(**if**) Suppose $e = (X, Y)$ is a covered edge in G. Let G' be equivalent to G, except that e is reveresed to $e' = (Y, X)$. Then:

- G' is also a DAG. To show this, suppose that G' is not a DAG. Since G and G' only differ by $e' = (Y, X)$, there must be a directed cycle incuding e' in G'. Then there is a path in G from Y to X. By assumption, Y and X have a shared set of parents. However, one can conclude from this that G also contains a cycle because every cycle mut include at least one node from $\prod_x^G$, and therefore is not a DAG, a contradiction. By this contradiction, we conclude that G' must be a DAG.
- $G' \sim_M G$. Firstly, G and G' trivially have the same skeleton, since they only differ by the orientation of edge e. Then, if they were *not* Markov equivalent graphs, they would differ by a v-structure. If G' has a v-structure not present in G, it must include the edge (Y, X). However, this would imply that X has a parent which is not a neighbor to Y, contradicting the assumption that e is covered. A similar argument holds for the scenario where G has a v-structre not present in G'.

(**only if**) We now show that if $e = (X, Y)$ is not a covered edge in G, we are in one of two cases: either G' contains a directed cycle, or G' is a DAG which is not equivalent to G. If (X, Y) is not a covered edge in G, then at least one of the following hold:

1. There is a node $Z \neq X$ which is a parent of Y but not of X.
2. There s a node W which is a parent of X but not of Y.

Let $Z \neq X$ be a parent of Y in G but not a parent of X in G. If Z and X are not neighbors, then there is a v-structure consisting of edges (X, Y) and (Y, Z) in G which does not exist in G'. If X is a parent of Z in G, then it must also be a parents of Z in G', which would imply that G' contains a directed cycle and is therefore not a DAG.

The argument for the second case is equivalent, assuming W is a parent of X in G and deriving a v-structure with $(W, X) and (X, Y)$ which exists in G' but not G. We conclude that G would have to contain a directed cycle. In both scenarios, we have derived a contradiction which shows that (X, Y) must be covered in G.

□

Example 4.2.1 (Members of Markov equivalence class [10]). Consider the four DAGs in Figure 4.2.2.

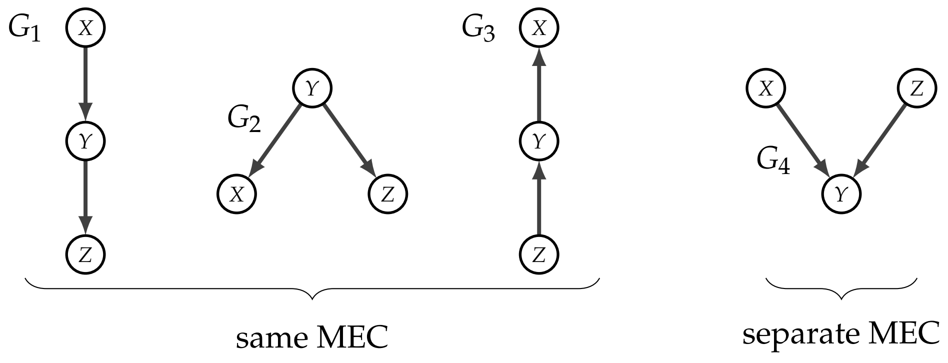


Figure 4.2.2

The fact that G_1, G_2, and G_3 are in the same MEC while G_4 is not can be seen by considering the joint probability distribution $p(X,Y,Z)$ of each of the graphs.

$$\begin{aligned}
G_1 : &\quad p(X,Y,Z) = p(X) \cdot p(Y|X) \cdot p(Z|Y) \\
G_2 : &\quad p(X,Y,Z) = p(Y) \cdot p(X|Y) \cdot p(Z|Y) \\
G_3 : &\quad p(X,Y,Z) = p(Z) \cdot p(Y|Z) \cdot p(X|Y)
\end{aligned}$$

By Bayes' theorem, all of the values of G_2 can be completely determined from values from G_1:

$$p(X)p(Y|X) = p(X,Y) = p(Y)p(X|Y)$$

and $p(Z|Y)$ is unchanged. A similar application of Bayes' theorem shows that G_3 is completely determined by G_1 and G_2:

$$p(z) \cdot p(Y|Z) = p(Z,Y) = p(Y) \cdot p(Z|Y)$$

where $p(Y)$ can be attained directly from G_2 and $p(Z|Y)$ can be attained from G_1. Therefore, we can conclude that since all three describe the same distribution, they lie in the same equivalence class. In contrast, the joint probability for G_4 is given by

$$p(X,Y,Z) = p(X) \cdot p(Z) \cdot p(Y|X,Z)$$

which can not be determined from the values of G_1, G_2 and G_3. This is because in [G_1] X is conditionally independent from Y and Z, that is, X is only independent in the circumstance that we are given values for Y and Z. Meanwhile, in G_4, X is marginally independent of Z, that is, independent in the circumstance that we ignore Y. Conditional independence does not provide information about marginal independence, and conversely, marginal independence does not imply conditional independence. Therefore, G_4 describes a different distribution, and does not lie in $[G_1]$.

Corollary 1. *Lemma 4.2.1 and Lemma 4.2.2 together imply that the graph G' which results from only reversing the orientation of a covered edge $(X, Y) \in G$ is in the MEC of $[G]$.*

Proof. This follows from the fact that edge reversal does not change the skeleton of the graph G (since no edges are added nor removed) and reversing covered edges does not alter the v-structures of G (Lemma 4.2.2), thus the v-structures and skeleton are unchanged. □

4.3 Effects of Edge Reversal

Lemma 4.3.1. *Given a single directed edge between two parentless nodes in which the joint probability $p(X_1, X_2) = p(X_1) \cdot p(X_2|X_1)$, the new factorization of the joint probability distribution $p(X_1, X_2)$ can be computed as follows using only the known quantities $p(X_1)$ and $p(X_2|X_1)$:*

$$p(X_1, X_2) \;=\; \sum_{X_1} p(X_2|X_1) \cdot p(X_1) \frac{p(X_2|X_1) \cdot p(X_1)}{\sum_{X_1} p(X_2|X_1) \cdot p(X_1)}.$$

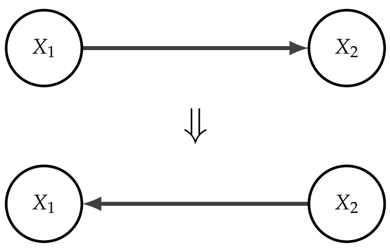


Figure 4.3.1

Proof. This follows directly from Bayes' theorem, which gives us

$$\begin{aligned} p(X_1|X_2) &= \frac{p(X_2|X_1) \cdot p(X_1)}{p(X_2)} \\ &= \frac{p(X_2|X_1) \cdot p(X_1)}{\sum_{X_1} p(X_2|X_1) \cdot p(X_1)}. \end{aligned}$$

□

Lemma 4.3.2. *Given a directed edge between two nodes X_1 and X_2 with a shared parent node X_P, the joint probability $p(X_1, X_2, X_p)$ after switching edge (X_1, X_2) to (X_2, X_1) can be computed as follows using only known quantities $p(X_p)$, $p(X_1|X_p)$, and $p(X_2|X_1, X_p)$:*

$$p(X_1, X_2, X_P) = [\frac{p(X_2|X_1, X_p) \cdot p(X_1|X_p)}{\sum_{X_1} p(X_1|X_p) \cdot p(X_2|X_1, X_p)}] \cdot [\sum_{X_1} p(X_1|X_p) \cdot p(X_2|X_1, X_p)] \cdot p(X_p).$$

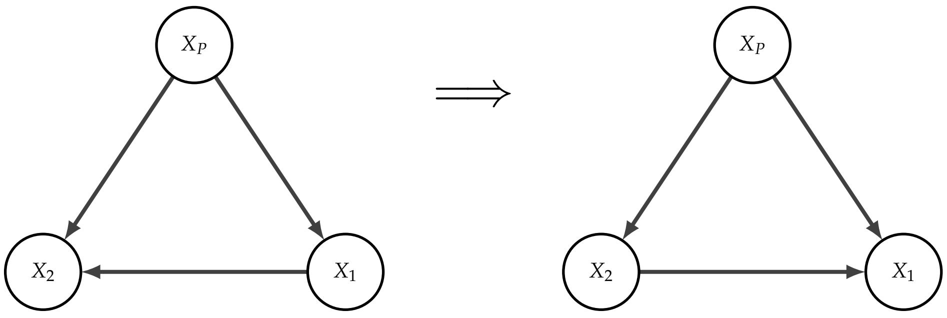


Figure 4.3.2

Remark. Furthermore, because this result does not depend on $p(X_p)$, we can conclude that for a set of shared parents $P = \{X_{P_1}, X_{P_2}, \ldots X_{P_i}\}$ of two nodes X_1 and X_2, the same equality holds. Additionally, note that any reversal of an edge with a set of shared parents is MEC-invariant, since all such edges are covered (corollary 1).

Proof. From before the edge reversal, we have access to the values $p(X_p), p(X_1|X_p)$, and $p(X_2|X_1, X_p)$. Our goal is to determine the new joint probability distribution $p(X_1, X_2, X_p)$ using this information. Therefore, we must find $p(X_p), p(X_2|X_p)$, and $p(X_1|X_2, X_p)$. First, $p(X_p)$ is already trivially available. Then, to compute $p(X_1|X_2, X_p)$,

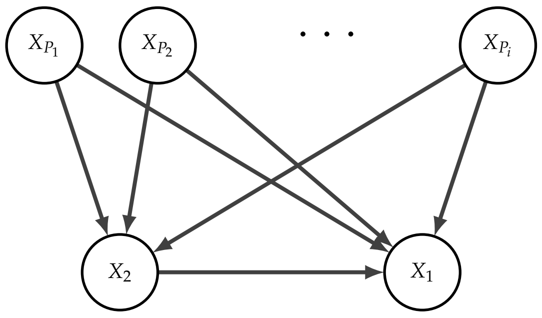


Figure 4.3.3

$$\begin{aligned}
p(X_1|X_2,X_p) &= \frac{p(X_1,X_2,X_p)}{p(X_2,X_p)} \\
&= \frac{p(X_2|X_1,X_p)\cdot p(X_1,X_p)}{p(X_2,X_p)} \\
&= \frac{p(X_2|X_1,X_p)\cdot p(X_1|X_p)\cdot p(X_p)}{p(X_2|X_1,X_p)} \\
&= \frac{p(X_2|X_1,X_p)\cdot p(X_1|X_p)}{p(X_2|X_p)}
\end{aligned}$$

which depends only on known quantities once we compute

$$\begin{aligned}
p(X_2|X_p) &= \frac{p(X_2,X_p)}{p(X_p)} \\
&= \frac{1}{p(X_p)}\sum_{X_1} p(X_1,X_2,X_p) \\
&= \frac{1}{p(X_p)}\sum_{X_1} p(X_1,X_p)\cdot\frac{p(X_2,X_1,X_p)}{p(X_1,X_p)} \\
&= \sum_{X_1} p(X_1|X_p)\cdot p(X_2|X_1,X_p).
\end{aligned}$$

Together, these give us

$$\begin{aligned} p(X_1, X_2, X_p) &= p(X_1|X_2, X_p) \cdot p(X_2|X_p) \cdot p(X_p) \\ &= [\frac{p(X_2|X_1, X_p) \cdot p(X_1|X_p)}{\sum_{X_1} p(X_1|X_p) \cdot p(X_2|X_1, X_p)}] \cdot [\sum_{X_1} p(X_1|X_p) \cdot p(X_2|X_1, X_p)] \cdot p(X_p). \end{aligned}$$

□

Remark. Notice that the above cancels to show that the two graphs are Markov equivalent, since $p(X_1|X_2, X_p) \cdot p(X_2|X_p) \cdot p(X_p) = p(X_2|X_1, X_p) \cdot p(X_1|X_p) \cdot p(X_p)$.

4.4 Identifying Reversible Edges

Definition 4.4.1 (Essential edge [12], [14]). *An **essential edge** (also called a compelled edge [11]) of a graph $G = (V, E)$ is an edge $(X, Y) = e \in E$ such that for every graph $G' = (V, E')$ in $[G]$, the orientation of $e \in E$ is unchanged. That is, there is no graph $G' = (V, E')$ in $[G]$ such that $(Y, X) \in E'$.*

Definition 4.4.2 (Essential graph [12]). *The **essential graph** of a Markov equivalence class $[G]$ is a mixed graph G_* such that the only directed edges in G_* are essential edges of $[G]$. That is,*

- *The directed edge $e = (X, Y) \in G_*$ iff $e \in G$ for every graph $G \in [G]$,*
- *The undirected edge $X, Y \in G_*$ iff there are graphs G_1 and $G_2 \in [G]$ such that (X, Y) in G_1 and (Y, X) in G_2.*

Example 4.4.1. In this example, we identify the essential edges of a Markov equivalence class and construct the essential graph. Consider the following DAG $G = (V, E)$:

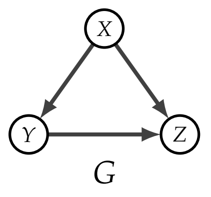


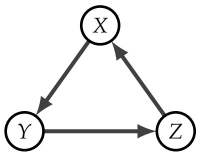
Figure 4.4.1

We use the following realizations to show that the graph has no essential edges.

If we switch any edge in G, the resulting graph G' becomes one of two cases:

- G' is cyclic, and therefore no longer a DAG, meaning this edge was irreversible.

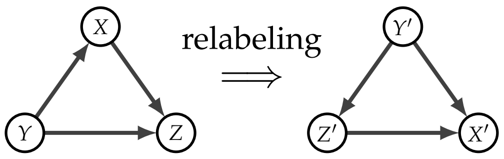
  Example: reversing edge (X, Z) as in **(A)** creates a cycle.



(A)

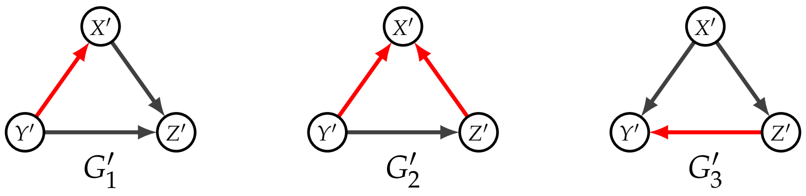
- G' is equivalent to G up to relabeling, and therefore in the same MEC.

  Example: reversing (X, Y) and changing labels according to $X \to Y'$, $Y \to Z'$, and $Z \to X'$, as in **(B)**.

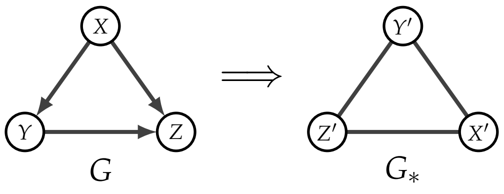


(B)

Thus the following graphs are included in $[G]$, though they are not the entire MEC. The red edges in each are those with orientation which differs from edges in G. Notice, however, that some relabeling was required, meaning these edges do not necessarily correspond (under this new labeling) to covered edges.



Therefore, under some circumstance, for every edge $(U, V) = e \in G$, there is a DAG $G' \in [G]$ such that $(V, U) \in G'$. This means that G has no essential edges. We construct the essential graph G_* of G by removing the orientation of all non-essential edges in G:



Remark. Andersson et. al. [12] demonstrate that essential graphs are consistent across a MEC and unique to it, meaning an essential graph can be used as a representative for a given MEC. Furthermore, essential graphs are not necessarily DAGs, and therefore do not generally belong to the MEC that they represent.

This can be seen by the fact that the distinction between two MECs lies entirely in either differences in their skeletons or in the orientations of their compelled edges.

"It is only in the heart that one can see rightly; what is essential is invisible to the eye."
-Antoine de Saint-Exupéry, The Little Prince.

Remark. One should take care to note the distinction between implications about non-essential edges versus covered edges:

- For $G = (V, E)$, an edge $e \in E$ is non-essential iff there exists a graph G' in $[G]$ such that e is reversed,
- An edge e in G is covered iff the graph G' derived from *only* reversing e, is in the MEC of G. (Follows from corollary 1.)

That is, reversing *e exclusively* while remaining in the MEC of G requires e to be covered. Meanwhile, a non-essential edge $e \in G$ may require other edges in G to be reversed as well to remain in the same MEC. This means that not all non-essential edges can be reversed at a given time. The following example demonstrates that some configurations of non-essential edge reversals are not possible if we wish to remain in the MEC of G.

Example 4.4.2. Again consider members of the Markov equivalence class seen in example 4.2.1. The positions of the nodes are visually rearranged for clarity, but no properties are changed.

Notice that edge $(Y, Z) \in G_1$ is non-essential, since there exists another graph (namely G_3) in $[G_1]$ such that $(Z, Y) \in G_3$. Therefore, we know that there exists some combination of edge reversals such that (Y, Z) can be reversed without leaving $[G_1]$. However, if we choose to switch only (Y, Z) to (Z, Y), we have exactly graph G_4, which is no longer in the same MEC. Therefore, the fact that an edge is non-essential does not provide information about the circumstances in which the

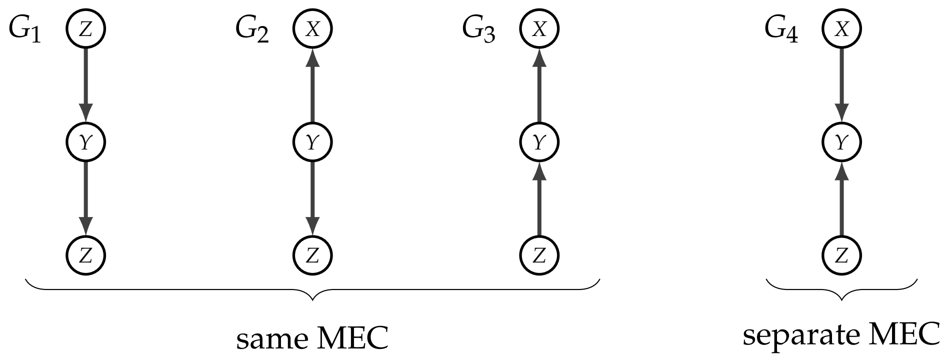


Figure 4.4.2

edge can be reversed without altering the MEC; only that in some context, it can be. This realization leads us to corollary 2.

Corollary 2. *Let E_c be the set of covered edges in G, and let E_e be the set of essential edges in G. Then $E_c \subseteq \bar{E}_e$, the set of non-essential edges in G.*

Proof. This follows directly from the realization that non-essential edges can be switched while remaining in the same MEC under the correct conditions, while covered edges can be switched immediately, without regard to other edge conditions (the conditions for their reversal are always met). □

Example 4.4.3. We use the graph G from example 4.4.1 to demonstrate corollary 2. Of course, this is not sufficient for a proof, but rather helps to contextualize the lemma.

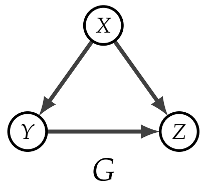


Figure 4.4.3

In example 4.4.1 we determined that all edges $e_i \in E$ of $G = (V, E)$ are non-essential, so the set of non-essential edges $E^N_G = \{(X,Y),(Y,Z),(X,Z)\}$. We can quickly see that the only covered edge of G is (Y,Z) since Y and Z share a single parent X, not including Z itself. Then, indeed, $\{(Y,Z)\} \subset E^N_G$.

Remark. There are two scenarios we consider for finding reversible edges, that is, edges which can be reversed without altering the MEC of the graph:

- Firstly, we may wish to find the set of edges which can be immediately reversed in one step, without regard to other edges in the graph. This coincides with the set of covered edges.
- Secondly, we may wish to find the set of edges which can ever be reversed within a graph, under the correct conditions of other edges being reversed beforehand/simultaneously. This coincides with the set of non-essential edges.

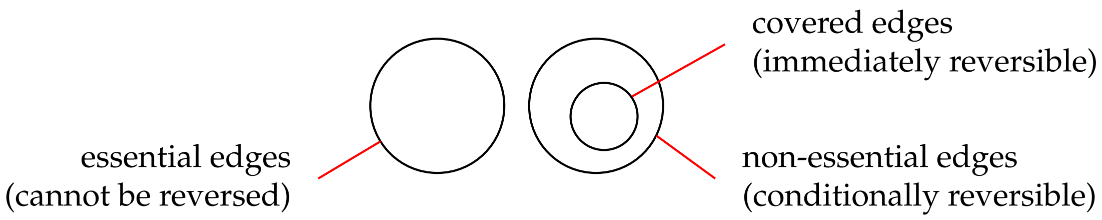


Figure 4.4.4

Theorem 4.4.1 (Edge reversal sequence. [11]). *Given two Markov equivalent DAGs G and G', there exists a sequence of distinct edge reversals in G with the following properties:*

- *Each edge reversed in G is a covered edge,*
- *After each reversal, G is a DAG and $G \sim_M G'$*
- *After all reversals, $G = G'$.*

That is, we can transform G into G' via a finite sequence of covered edge reversals, such that all of the intermittently resulting graphs $G_1, G_2, ...G_n \sim_M G, G'$.

Proof. Chickering's [11] proof defines the procedure **Procedure Find-Edge** which takes a DAG G as input. At each step, the procedure identifies a next edge to reverse. To be reversible, such an edge must be covered, by Lemma 4.2.2. The proof demonstrates that such an edge is identifiable when G and G' remain unequal, and each reversal reduces the number of edges in total which must be reversed, eventually transforming G into G'. □

Chapter 5

Exploiting Markov Equivalence using Greedy Strategy

5.1 Motivation

One complication of minimizing the cost of a query over a Markov equivalence class is that Markov equivalence classes can be quite large. Constructing all members of the class then searching for a minimal graph among those members may be very costly.

Additionally, searching for covered edges (edges reversible at a given moment) is a siginificantly less costly task than searching for essential edges (edges which can be reversed under the correct circumstances, but not necessarily at a given moment). One can take advantage of these circumstances by searching for an optimal single edge reversal over the set of reversible edges in a graph G.

We therefore develop the greedy strategy, an algorithm in which each step entails searching for a single edge that can be reversed without leaving the original Markov equivalence class, while simultaneously reducing the number of vertices which must be computed to answer our query q on G, written $\Delta(G,q)$.

There are two aspects we wish to consider while searching for a reversible edge which reduces the cost of answering a query:

- Does reversing the edge alter existing v-structures in the graph, either by creating a new v-structure or destroying an existing one?
- Does switching the edge indeed benefit our query, ie, reduce the number of variables that must be considered while answering the query?

In this section, we present some relevant background, create a procedure to identify reversible edges, develop a framework for when edge switching is beneficial, and analyze the efficacy of the procedure for minimization.

5.2 Background and Construction

5.2.1 Finding Reversible Edges

The greedy algorithm takes advantage of Lemma 4.2.2, which states that reversible edges are exactly covered edges. Therefore, to search for reversible edges, we simply search for covered edges. We define the following algorithm to search for covered edges using Definition 4.2.3 directly, which tells us that covered edges (X_1, X_2) have the same set of parents excluding X_1 itself.

Algorithm 1: COVERED EDGES returns a set of covered edges in a DAG

```
Input: A directed acyclic graph $G$
Output: The set of covered edges in $G$
1 covered_edges = []
2 for $e = (X_1, X_2) \in Edges(G)$ do
3     if $\prod_{X_1}^{G} = \prod_{X_2}^{G} \setminus X_1$ then
4         add $e$ to covered_edges
5 end
6 return covered_edges
```

5.2.2 Determining the Benefit of a Reversal

Once we have identified the set of reversible edges, we must determine whether a switch is advantageous. Let $G = (V, E)$ be a DAG with $(X_1, X_2) \in E$, and let G' be the DAG achieved by reversing (X_1, X_2). Then, to determine whether reversing (X_1, X_2) is advantageous given a query q, we compare $|\Delta(G, q)|$ to $|\Delta(G', q)|$. If the latter is smaller, we have reduced the number of nodes the computation of q depends on, and therefore the reversal is advantageous.

Then, we can determine which edge reversal is most advantageous by searching for $max_{G'} |\Delta(G, q)| - |\Delta(G', q)|$ where, without loss of generality, G' is the DAG achieved by reversing a single covered edge in G.

Algorithm 2: IS ADVANTAGEOUS returns the most advantageous edge reversal over G given a query

```
Input: A directed acyclic graph $G$, a query $q$
Output: The most advantageous possible edge reversal in $G$
1 edge_advantages = {}
2 for $e \in$ covered_edges(G) do
3     $G' = G$ with $e$ reversed edge_advantages[$e$] = $|\Delta(G,q)| - |\Delta(G',q)|$
4 end
5 best_reversal = $e$ such that edge_advantages[$e$] = max(values in
    edge_advantages)
6 return best_reversal
```

5.2.3 Identifying $\Delta(G,q)$

Notice that Algorithm 2 requires algorithmically determining the set $\Delta(G,q)$. To do so involves two subtasks: first, we must determine which case (either special or general) each condition in the query is in. Then, once all conditional vertices have been classified, we must add nodes accordingly to the set $\Delta(G,q)$.

There are at least two possible methods by which one can determine which case a vertex is in. These correspond to the two ways of framing the cases, namely Theorem 3.3.2 and Lemma 3.3.3:

Theorem 3.3.2 suggests that we approach the cases by considering paths. In this work, we will use this method exclusively. Our goal is to determine whether, for a given condition Y, a target X is conditionally independent of $\alpha(Y)$ given Y. To do so, we check all route from X to each element in $\alpha(Y)$, calling these Y_{α_i}, $i \in [1, |\alpha(Y)|]$. If there exists a route from X to some Y_{α_i} which does not pass through Y, we know that X is not independent of $\alpha(Y)$ given Y. This would mean that Y is in the general case. If all paths from X to Y_{α_i} pass through Y, then $\alpha(Y)$ is independent of X given Y, and therefore Y is in the special case.

The alternative method corresponding to Lemma 3.3.3 involves creating a copy of the graph G called G' in which the vertex Y is removed from G', and checking whether

1. G' is made up of two disjoint subgraphs G_1 and G_2,
2. $X \in G_1$ and all elements in $\alpha(Y) \in G_2$.

If both of the above are true, then X is independent from $\alpha(Y)$ given Y, and therefore Y is in the special case. Otherwise, Y is in the general case.

The method outlined by Theorem 3.3.2 is presented in Algorithm 3:

Algorithm 3: FIND CASE returns the case of a single condition Y with respect to a target X, either Special or General

```
Input: A DAG $G$, a target $X$, a condition $Y$
Output: The case of condition $Y$ with respect to target $X$ in $G$
$G'$ = skeleton of $G$ #to ensure routes, not directed paths
paths = all simple paths from $X$ to $Y$ in $G'$ (no repeated vetices)
if $|\prod_G^Y| = 0$ then
    case = 'special'
    return case
for path in paths do
    for ancestor in $\alpha(Y)$ do
        if ancestor $\in$ path then
            case = 'general'
            return case
        else
        continue
    end
end
#only reaches this block if a case has not already been established
case = 'special'
return case
```

The logic of the general case block is as follows: if any path from X to Y passes through an element in $\alpha(Y)$, then there is a back-route to Y through its ancestors, meaning there is a path connecting X to $\alpha(Y)$ which does not pass through Y.

This algorithm is applied repeatedly until the cases of all conditions Y have been determined, at which point the appropriate vertices can be added to $\Delta(G, q)$ accordingly. These repeated application and collection of relevant vertices is handled by Algorithm 4.

Algorithm 4: CREATE DELTA SET returns the set $\Delta(G,q)$.

```
Input: A DAG $G$, a target $X$, a set of conditions $Ys$ which correspond to a
       query $q = p(X|Ys)$
Output: $\Delta(G,q)$
$G'$ = skeleton of $G$ #to ensure routes, not directed paths
$\Delta(G,q)$ = []
cases = {}
if len(Ys) = 0 then
    $\Delta(G,q) = alpha(X)$
    return $\Delta(G,q)$
Add $X$ to $\Delta(G,q)$
for $Y$ is $Ys$ do
    cases[Y] = find_case(G,X,Y)
    shortest_path = the shortest path from $X$ to $Y$ over $G'$
    if $Y$ is in the general case then
        for node in shortest_path do
            add node to $\Delta(G,q)$
        end
        for $Y_\alpha$ in $\alpha(Y)$ do
            add $Y_\alpha$ to $\Delta(G,q)$
        end
        if $Y$ is in the special case then
            for node in shortest_path do
                add node to $\Delta(G,q)$
            end
            Remove $Y$ from $\Delta(G,q)$
            Remove $\alpha(Y)$ from $\Delta(G,q)$
end
return $\Delta(G,q)$ with no repeating values
```

5.3 Complexity of the Greedy Strategy

In this section, we will find the complexity of each of the algorithms involved in our greedy strategy. Ultimately, the goal is to compare the costs of identifying and enacting a beneficial transformation versus computing the query without transforming the graph. To do so, we will compare the complexity of the following:

1. Computing a query q on a DAG G,
2. Identifying a beneficial edge reversal $e \in E$ which can be done on G,
3. Computing the probabilities of the new graph G' achieved by reversing e.
4. Computing a query q on G'.

which ultimately allows us to determine whether the cost of reversal was worth the speedups of answering q.

5.3.1 Restrictions on Graph Structure

In order to reasonably compute the complexity of these algorithms, we must lay some restrictions on the structure of our DAG G for the following reasons:

- Even in sparse graphs–graphs in which $|E| \in O(|V|)$– the number of parents is not inherently bounded. For example, the graph could be a single parent vertex connected directly by one edge to all other vertices in the graph. Therefore, even restricting ourselves to sparse graphs is not sufficient if we wish to keep our algorithms tractable. To combat this, we set a limitation of the maximum number of parents allowable for a given vertex.
- Likewise, the number of cycles (Definition 2.1.4) in a graph are not inherently restricted. This means that the number of paths between two arbitrary vertices in G are affected exponentially. We therefore also restrict the maximum number of cycles allowable in the graph.

Luckily, in practice, these restrictions are reasonable for a graphical model; graphical models tend to be sparse graphs by the nature of their usage. This is partially due to the fact that they often encode real data sets with a small number of parameters (for example, a medical office can only store so much data about patients' demographics and health backgrounds).

The particular restrictions we enforce are: (1) The maximum degree of a given vertex $v \in V$ is logarithmic, that is, maxdeg(G) $\in O(log(|V|))$, and (2) the maximum number of cycles in G is logarithmic; num_cycles $\in O(log(|V|))$.

Remark. One result of these restrictions is that a single conditional probability table of G contains at most $O(|V|)$-many values and $O(|V|)$-many paths between two given vertices.

5.3.2 Complexity of Components

Let $G = (V, E)$ be a sparse DAG with the aforementioned restrictions, that (1) maxdeg(G) $\in O(log(|V|))$ and num_cycles $\in O(log(|V|))$. Let the vertices in G be binary.

Covered Edges: In the general setting, Algorithm 1 has complexity $O(|E| * |V|)$. This is because the main component of the algorithm is an iteration through the set of edges E ($|E|$-many steps), wherein each iteration, two sets of vertices (bounded by the total number of vertices $|V|$) are compared. In the context of sparse graphs, in which $|E| \in O(|V|)$, this becomes $O(|V|^2)$. Furthermore, since the sets of vertices being compared are sets of parents, they are bounded by $log(|V|)$ by assumption. Therefore, the total complexity is $O((log(|V|))^2)$.

Find Case: Algorithm 3 has complexity $O(e^c|V|^2)$ where c is the number of cycles in G. By assumption $c \in O(log(|V|))$, and therefore this complexity becomes $O(e^{log(|V|)}|V|^2) = O(|V|^3)$.

Create Delta Set: Algorithm 4 relies on a breadth-first search to identify a shortest path from a vertex X to a vertex Y in $G' = skeleton(G)$. A breadth first-search has complexity $O(|V| + |E|)$, which in the case if a sparse graph, is $O(|V|)$ since $|E| \in O(|V|)$. However, Create Delta Set should be considered in conjunction with Find Case, which it calls for each condition Y_j in q. Since Y are vertices, $Y \in O(|V|)$. The dominating factor of calling Create Delta Set is calling Find Case. Therefore, Create Delta Set and Find Case together amount to a complexity of $O(|V| * |V|^3) = O(|V|^4)$ in the case of our restricted graphs.

Is Advantageous: Algorithm 2 called Create Delta Set twice for every covered edge $e \in E$. Naturally, the number of covered edges is bounded by $|E|$ which, in a sparse graph, is $O(|V|)$. Therefore, considering the complexity of Is Advantageous called $O(|V|)$-many times, we have $O(|V|^5)$.

Edge Switch: Reversing an edge (X, Y) to (Y, X) requires us to compute the new conditional probabilities of X and Y as outlined in Lemma 4.3.2. This reversal has complexity $O(2^{|\prod_G^X|})$ for a target vertex X. That is, it scales exponentially with respect to the number of parents of the target node. Under our assumption that $\prod_G^X \in O(log(|V|))$, the algorithm cas complexity $O(2^{log(|V|)})$, making it a moderate exponential increase.

Chapter 6

Efficacy of Transformation

6.1 Comparison of methods

A comparison of the speed of normal querying VS the speed of the new query + time required to exploit ME.
Ultimately trying to answer the questions:

- Can we do faster inference?
- Under what circumstances is it possible/effective?
- How much faster is it?

6.2 Conclusion

6.3 Future work

This work has relied on several restrictions in order to quantify the complexity of the algorithms used. Firstly, many of the scenarios explored were in the binary setting. With some patience, the logic of the binary setting can be extended to categorical variables. Further, in cases where quantifying the complexity of an algorithm is difficult– say, if the graph has an unrestricted number of parents– one could experimentally determine whether the greedy strategy is effective by timing it over a large number of randomly generated graphs. Then, even when the arithmetic complexity is unclear, one can gain an understanding of whether it is effective, and if so, under what circumstances.

We also primarily explored the greedy strategy, meaning we looked for a single effective edge reversal over the set of covered edges. This was done with respect to one query rather than a sequence of queries. In future work, one could explore other methods, such as generating the entire Markov equivalence class $[G]$ of a given input DAG G, and determining which graph $G' \in [G]$ minimizes the cost of the query. Likewise, one could try both the greedy strategy and this full-search strategy for a string of queries, possibly outweighing the overhead cost of manipulating the graph further.

In more detailed consideration, one could compare the method presented in section 5.2 of determining which case (special or general) a query is in to the other proposed method; that is, compare the method of searching among all paths between target nodes and condition parents to the method of copying the graph, removing the conditional vertex, and checking the structure of the newly created graph (is it disjoint? Are the parents of the condition in a separate subcomponent from the targets?). The author believes that the presented method is more efficient, but has not explored this.

Bibliography

[1] S. Greenland, J. Pearl, and J. Robins, "Causal diagrams for epidemiological research," *Epidemiology 10*, pp. 37–48, 1999.

[2] M. Glymour and S. Greenland, "Causal diagrams," *Modern Epidemiology, 3rd edition*, pp. 183–209, 2008. ed. Lippincott Williams & Wilkins, Philidelphia, PA.

[3] P. L. Miller and P. R. Fisher, "Causal models for medical artificial intelligence," *Selected Topics in Medical Artifical Intelligence*, 1988. ISBN: 978-1-4613-8779-4.

[4] P. Szolovits and S. G. Pauker, "Categorical and probabilistic reasoning in medical diagnosis," *Artificial Intelligence, Volume 11, Issues 1-2*, pp. 115–144, 1978.

[5] M. Shwe, B. Middleton, D. Heckerman, M. Henrion, E. Horvitz, H. Lehmann, and G. Cooper, "A probabilistic reformulation of the quick medical reference system," *Annual Symposium on Computer Application in Medical Care*, pp. 790–794, 1990.

[6] S. Morgan and C. Winship, "Counterfactuals and causal inference: Methods and principles for social research (analytical methods for social research)," 2007. Cambridge University Press, New York, N.

[7] H. Blalock, Jr., "Causal models in the social sciences," *The Economic Journal Vol. 82 No. 328.*, pp. 1420–1423, 1972. Published by Oxford University Press.

[8] D. Cox and N. Wermuth, "Causality: a statistical view," *International Statistical Review 72*, pp. 285–305, 2004.

[9] S. Lauritzen, "Causal inference from graphical models," *Complex Stochastic Systems*, pp. 63–107, 2001. Chapman and Hall/CRC Press, Boca Raton, FL.

[10] T. Verma and J. Pearl, "Equivalence and synthesis of causal models," *Proceeding of the Sixth Annual Conference on Uncertainty in Artificial Intelligence, UAI '90*, pp. 255–270, 1990. DOI: 10.1142/S0218488504002576.

[11] D. M. Chickering, "A transformational characterization of equivalent bayesian network structures," *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence UAI'95*, 1995. Morgan Kaufmann Publishers Inc.

[12] S. Andersson, D. Madigan, and M. Perlman, "A characterization of markov equivalence classes for acyclic digraphs," *Annals of Statistics 25*, 1997.

[13] J. Pearl, "Causal inference in statistics: an overview," *Statistics Surveys Vol. 3*, pp. 96–146, 2009.

[14] I. Flesch and P. Lucas, "Markov equivalence in bayesian-network structures," *J. Mach. Learn. Res., 2*, 2002.

[15] R. D. Schachter, "Evaluating influence diagrams," *Operations Research Vol. 34, No. 6 (Nov. - Dec. pp. 871-882)*, 1986.

[16] T. A. Stephenson, "An introduction to bayesian network theory and usage," *Dalle Molle Institute for Perceptual Artificial Intelligence*, 2000. IDIAP-RR 00-03.

[17] C. M. Grinstead and J. L. Snell, "Introduction to probability," *American Mathematical Society*, 2003. Copyright (C) 2006 Peter G. Doyle.

Appendix

Python code for determining the set $\Delta(G,q)$

For a DAG G, the following code determines which nodes are in the se $\Delta(G,q)$ for a query q. It uses two functions:

1. `find_case`, a helper function which determines whether nodes are in the special case or the general case according to the logic from Theorem 3.3.4.

2. `create_delta_set`, the main function which populates the set $\Delta(G,q)$ depending on the result of `find_case`.

```
# Helper function for create_delta_set(); checks for paths between targets and
                                           parents of conditions.
# Inputs: a DAG $G$, a target vertex $X$, a single conditional vertex $Y \in
                                           \{Y_{0},Y_{1},...Y_{n}\}$ for the query
                                           $q = (X|Y_{0},Y_{1},....Y_{n})$.
#Returns: the case of individual vertex $Y$ in the query $q$, either special or
                                           general.

def find_case(G,X,Y,cases):
    skeleton =G.to_undirected()
    y_ancestors =list(nx.ancestors(G,Y))
    paths =list(nx.all_simple_paths(skeleton, X, Y, cutoff=None)) #use skeleton
                                               so they are undirected (routes not
                                               paths).

    if len(y_ancestors) ==0:
        cases[Y] ="special" #Yj in trivial special case
        return(cases)

    # The logic of the following is to find paths from X to Y: if any of them
                                               pass through an ancestor of Y then
                                               there is a "back route" to Y through
```

```
                                                  its ancestors, meaning there is a
                                                  path connecting X to a(Y)
    # which does not pass through Y.
    for path in paths:
        for y_ancestor in y_ancestors:
            if y_ancestor in path: #There exists a path from X to a(Y) which does
                                                          not pass through Y
                cases[Y] ="general" #Yj in general case
                return(cases)
            else:
                continue

    #Only reaches this block of code if all paths to a(Y) passed through Y; in
                                                  nontrivial special case.
    cases[Y] ="special" #Yj in the special case
    return(cases)
```

```
#Inputs: A DAG G, a target vertex X, and conditional vertices Ys =
                                              {Y_{0},Y_{1},...Y_{n}})$corresponding
                                              to a query q = p(X|Y_{0},
                                              Y_{1},...Y_{n}).
#Returns: the set Delta(G,q).

def create_delta_set(G,X,Ys): #Graph G, one target X, set of multiple conditions
                                              Ys = {y0, y1, ...}.
    skeleton =G.to_undirected()

    dependent =[]
    cases ={} #Dictionary with (key, value) pairs: (Y,case).
    if not nx.is_connected(skeleton): #Only consider connected graphs.
        return(0)
    if len(Ys) ==0:
        return(list(nx.ancestors(G,X)))

    for Y in Ys:
        cases =check_paths(G,X,Y,cases)

    shortest_path =nx.shortest_path(skeleton,X,Y) #use skeleton so we have
                                                  shortest route (undirected)
    x_ancestors =list(nx.ancestors(G,X))
    y_ancestors =list(nx.ancestors(G,Y))

    dependent.append(X)
    for x_ancestor in x_ancestors:
            dependent.append(x_ancestor)

    for Y in Ys:
        if cases[Y] =="general":
            for node in shortest_path:
                dependent.append(node) #Includes all nodes on shortest route form
                                                    X to Y including X and Y.
            for y_ancestor in y_ancestors:
                dependent.append(y_ancestor)

        if cases[Y] =="special":
            for node in shortest_path:
                dependent.append(node)
            dependent =list(set(dependent))
            dependent.remove(Y)
            for y_ancestor in y_ancestors:
                if y_ancestor in dependent:
                    dependent.remove(y_ancestor)

    return(list(set(dependent)))
```

Python code for identifying advantageous reversals

The following pieces of code determine which edge reversal is most advantageous at a single given time given a Dag G and a query $q = (X|Y_0, Y_1, ..., Y_n)$. The first piece, `find_covered_edges` searches for covered edges since they are equivalent to immediately reversible edges. It does so by comparing the parents of each of the two vertices corresponding to an edge in G.

```
# Inputs: A DAG G
# Return the set of covered edges in a graph. Independent of query.

def find_covered_edges(G):
    covered_edges =[]
    for edge in G.edges: #for edge (X,Y)
        source_parents =list(G.predecessors(edge[0])) #parents of X
        target_parents =list(G.predecessors(edge[1])) #parents of Y
        source_parents.append(edge[0]) #parents of X union X
        if set(source_parents) ==set(target_parents):
            covered_edges.append(edge)
    return(covered_edges)
```

Next, `is_advantageous` takes in a DAG G, a covered edge e, and the parameters of the query $q = p(X|Y_0, Y_1, ..., Y_n)$. It creates a graph $Grev$, the graph which results if e is reversed, and calculates $\Delta(Grev, q)$. Then, the difference between $\Delta(G, q)$ and $\Delta(Grev, q)$ is returned, indicating how advantageous the reversal is.

```
# Inputs: A DAG G, a covered edge, the parameters of the query q = p(X|Ys).
# Returns: The number of vertices Delta(G,q) is reduced by if the edge is
                                        reversed, that is, how advantageous the
                                        reversal is.


def is_advantageous(G,edge, X, Ys):
    vertex_cost_difference =0
    cost_before =len(find_case(G, X, Ys))

    x = edge[0]
    y = edge[1]
    Grev =G.copy()
    Grev =structure_reverse_edge(Grev,x,y)
    cost_after =len(find_case(Grev,X,Ys))
    if cost_after <cost_before:
        vertex_cost_difference =cost_before -cost_after
    return(vertex_cost_difference)
```

Finally, `best_reversal` iterates through the set of covered edges in G to determine which covered edge is most advantageous when reversed, and returns this edge.

```
# Input: A DAG G, target vertex X, the set of conditional vertices Ys which
                                        correspond to the query q = p(X|Ys).
# Returns: the most advantageous edge reversal possible at a given time.
def best_reversal(G, X, Ys):
    covered_edges =find_covered_edges(G)
    edge_advantages ={}
    for edge in covered_edges:
         edge_advantages[edge] =is_advantageous(G,edge,X,Ys)
    max_adv_edge =max(edge_advantages, key=edge_advantages.get)
    return(max_adv_edge)
```

Python code for encoding conditional node data

The following code assume binary variables, that is, for each vertex X in a DAG G corresponding to a random variable, $X = x \in \{0,1\}$. Encoding even binary variables is expensive and nontrivial, but the same arguments can be extended to categoricals for those patient enough to implement it.

Each vertex in G is specified by a set of conditionals corresponding to the values its parents can take. Therefore, the number of conditionals corresponds to the number of parents it has. For example, if X is a parentless vertex, it will be specified by $p(X=1)$. Since we are in the binary case, this is sufficient, since $p(X=0) = 1 - p(X=1)$.

If the vertex Y has a single parent X, it will be specified by $p(Y=1|X=0)$ and $p(Y=1|X=1)$ from which one can again determine $p(Y=0|X=0)$ and $p(Y=0|X=1)$. A node Z with parents X and Y will be specified by $p(Z=1|X=0,Y=0)$, $p(Z=1|X=0,Y=1)$, $p(Z=1|X=1,Y=0)$ and $p(Z=1|X=1,Y=1)$. Generally, a vertex with P parents will be specified by 2^P conditionals. Therefore, one must construct a dynamic data structure such as the following, which accounts for the variety in number of parents (and therefore conditionals) a vertex can have.

```python
import itertools
class NodeData:
    '''
    Attributes:
        parents: a list of strings that contains the names of the parent nodes
        conditionals: a list of size 2^p (where p is the number of parents)
            that contains the conditional probabilities of the RV of this node
                                                    given all possible values
                                                    for the parents
    '''
    def __init__(self, name, parents=None, conditionals=None):
        self.name =name
        self.parents =parents

        # initializing the data structures,
        # mostly stuff that ensures that everything also works for nodes without
                                                parents
        if parents ==None:
            self.num_parents =0
        else:
            self.num_parents =len(self.parents)
        if conditionals ==None:
            self.conditionals =[0 for _ in range(2**self.num_parents)]
        else:
            self.conditionals =conditionals


    def set_conditional(self, p, parent_assignment=None):
        '''
        parent_assignment: a dict that maps values to names of parents, e.g.
        {'x1' : 0, 'x2' : 1, 'x3' : 0, ...}
        this function computes the index of self.conditionals where the required
                                                      value is stored and sets the
                                                      conditional probability
        '''
        # handle special case of node without parents:
        if self.parents ==None:
            self.conditionals[0] =p
            return
        # general case:
        index =0
        for p_i in range(len(self.parents)):
            index +=parent_assignment[self.parents[p_i]] *2**p_i

        self.conditionals[index] =p


    def get_conditional(self, parent_assignment):
        '''
        parent_assignment: a dict that maps values to names of parents, e.g.
        {'x1' : 0, 'x2' : 1, 'x3' : 0, ...}
        this function computes the index of self.conditionals where the required
                                                      value is stored and returns the
                                                      conditional
        '''

        ##input of get conditionals is parent_assignment, dict which grows
                                                      linearly with number of parents.
        ##So we can calculate this in terms of numparsx.

        # handle special case of node without parents:
        if self.parents ==None:
            return self.conditionals[0]
        # general case:
        index =0
        for p_i in range(len(self.parents)):
            index +=parent_assignment[self.parents[p_i]] *2**p_i

        return self.conditionals[index]


    def print_conditionals(self):
        '''
        prints the full conditional distribution
        iterates through all possible assignments and prints the conditional
                                            distribution for each assignment
                                            in a seperate line
        '''
        all_possible_parent_assignments =[x for x in itertools.product([0,1],
                                            repeat=self.num_parents)]

        for a_i in range(2**self.num_parents):
            assignment ={self.parents[i] :all_possible_parent_assignments[a_i][i]
                                                for i in
                                                range(self.num_parents)}
            # some list-magic that produces a sufficiently nice string:
            assignment_string ="P("+self.name+" = 1 | " +', '.join([str(x) +" = "
                                                +str(assignment[x]) for x in
                                                assignment]) +" ) = " +
                                                str(self.get_conditional(assignment))
            print (assignment_string)
```

Python code for calculating edge reversals in a binary setting

Likewise to the data structures above, the following code assumes binary variables. It must also dynamically access the conditional probabilities specifying each node, since the length of specifiers of a vertex with P parents is 2^P.

```
def switch_edge(x, y):
    '''
    switches the edge (x,y) to (y,x)
    x, y: NodeData
    '''
    # preparation:
    # use itertools.product to get all possible assignments of values for the
                                            parent nodes:
```

```
all_possible_parent_assignments =list(itertools.product([0,1],
                                                        repeat=x.num_parents))
num_parents_assigments =len(all_possible_parent_assignments)

# get conditional joint distribution p(x,y | parents):
# initialization:
pxe0ye0 =[0 for _ in range(2**x.num_parents)]
pxe0ye1 =[0 for _ in range(2**x.num_parents)]
pxe1ye0 =[0 for _ in range(2**x.num_parents)]
pxe1ye1 =[0 for _ in range(2**x.num_parents)]


# iterate through all possible assignments for the common parents of x and y:
for a_i in range(num_parents_assigments):

    assignemnt_dict_x ={x.parents[i] :all_possible_parent_assignments[a_i][i]
                                        for i in range(x.num_parents)}
    assignemnt_dict_y ={x.parents[i] :all_possible_parent_assignments[a_i][i]
                                        for i in range(x.num_parents)}
    assignemnt_dict_y[x.name] =0

    # case x=0, y=1:
    pxe0ye1[a_i] =(1-x.get_conditional(assignemnt_dict_x)) *
                                        y.get_conditional(assignemnt_dict_y)
    # case x=0, y=0:
    pxe0ye0[a_i] =(1-x.get_conditional(assignemnt_dict_x)) *
                                        (1-y.get_conditional(assignemnt_dict_y))

    assignemnt_dict_y[x.name] =1
    # case x=1, y=1:
    pxe1ye1[a_i] =x.get_conditional(assignemnt_dict_x) *
                                        y.get_conditional(assignemnt_dict_y)
    # case x=1, y=0:
    pxe1ye0[a_i] =x.get_conditional(assignemnt_dict_x) *
                                        (1-y.get_conditional(assignemnt_dict_y))


# marginalize x from the conditional distribution of y:
pye0 = [pxe0ye0[i] +pxe1ye0[i] for i in range(num_parents_assigments)]
pye1 = [pxe0ye1[i] +pxe1ye1[i] for i in range(num_parents_assigments)]

# condition x on the possible values of y:
pxe0gye0 =[pxe0ye0[i]/pye0[i] for i in range(num_parents_assigments)]
pxe0gye1 =[pxe0ye1[i]/pye1[i] for i in range(num_parents_assigments)]
```

```
pxe1gye0 =[pxe1ye0[i]/pye0[i] for i in range(num_parents_assigments)]
pxe1gye1 =[pxe1ye1[i]/pye1[i] for i in range(num_parents_assigments)]

# construct new nodedata objects:
x_new_parents =[y.name]
if not x.parents ==None:
    x_new_parents +=x.parents
x_new =NodeData(x.name, parents=x_new_parents)

for a_i in range(num_parents_assigments):

    # construct the assignment-dictionary:
    assignment_dict ={x.parents[i] :all_possible_parent_assignments[a_i][i]
                                           for i in range(x.num_parents)}

    # case y=0:
    # add y to assignment_dict:
    assignment_dict[y.name] =0
    # set conditional for x_new:
    x_new.set_conditional(p=pxe1gye0[a_i], parent_assignment=assignment_dict)

    # case y=1
    # add y to assignment_dict:
    assignment_dict[y.name] =1
    # set conditional for x_new:
    x_new.set_conditional(p=pxe1gye1[a_i], parent_assignment=assignment_dict)

y_new_parents =[]

if not x.parents ==None:
    y_new_parents +=x.parents

y_new =NodeData(y.name, parents=y_new_parents)

for a_i in range(num_parents_assigments):
    # construct the assignment-dictionary:
    assignment_dict ={x.parents[i] :all_possible_parent_assignments[a_i][i]
                                           for i in range(x.num_parents)}
    y_new.set_conditional(p=pye1[a_i], parent_assignment=assignment_dict)

return x_new, y_new
```

Declaration of Authorship

I hereby declare that I have completed this work independently and using only the sources and tools specified. The author has no objection to making the present Master's thesis available for public use in the university archive.

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Seitens des Verfassers bestehen keine Einwände die vorliegende Masterarbeit für die öffentliche Benutzung im Universitätsarchiv zur Verfügung zu stellen.

Ort, Abgabedatum	Unterschrift der Verfasserin
(Place, date)	(Signature of the Author)