UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques
i Informàtica

BACHELOR THESIS IN COMPUTER SCIENCE AND SOFTWARE ENGINEERING

Treball Final de Grau d'Enginyeria Informàtica

ALGORITHMIC CAUSAL EFFECT IDENTIFICATION

Author: Martí Pedemonte Bernat

Advisors: Dr. Jordi Vitrià Marca
Álvaro Parafita Martínez

Made at: Department of Mathematics and Computer Science

Barcelona, June 20, 2021

Acknowledgements

Voldria donar les gràcies a la meva família per tot el suport que he rebut durant el transcurs del treball. Gràcies Joan, Susanna, Ferran i Sara per estar sempre aquí al meu costat.

A en Gabri, per començar amb mi el que semblava impossible i per animar-me a acabar-ho.

A l'Íngrid, per escoltar totes les meves històries i fer veure que t'interessen.

A la Marta i a la Júlia, per cada hora que hem passat treballant junts però separats.

A la Núria, per donar-me un cop de mà amb els dibuixos de grafs.

I would also like to thank Dr. Tikka for his rapid and helpful response in some subtle doubts I had.

A l'Álvaro, per totes les hores dedicades i els consells que m'has regalat.

Al meu tutor, Jordi, per haver-me ajudat a entrar en un camp tan interessant com la causalitat, i haver-me guiat en tot el procés. Gràcies.

Abstract

Our evolution as a species made a huge step forward when we understood the relationships between causes and effects. These associations may be trivial for some events, but they are not in complex scenarios. To rigorously prove that some occurrences are caused by others, causal theory and causal inference were formalized, introducing the *do*-operator and its associated rules. The main goal of this project is to understand and implement in Python some algorithms to compute conditional and non-conditional causal queries from observational data. To this end, we first present some basic background knowledge on probability and graph theory, before introducing important results on causal theory, used in the construction of the algorithms. We then thoroughly study the identification algorithms presented by Shpitser and Pearl in 2006 [SP 2006a, SP 2006b], explaining our implementation in Python alongside. The main identification algorithm can be seen as a repeated application of the rules of *do*-calculus, and it eventually either returns an expression for the causal query from experimental probabilities or fails to identify the causal effect, in which case the effect is non-identifiable. We introduce our newly developed Python library and give some usage examples towards the end of the dissertation.

L'evolució de la nostra espècie va fer un enorme salt endavant quan vam aconseguir entendre relacions de causa-efecte. Aquestes relacions poden ser trivials en certs casos, però no ho són en general. Per a demostrar rigorosament que certs esdeveniments són causats per altres, es van formalitzar la teoria de la causalitat i la inferència causal, tot introduint l'operador *do* i les seves regles associades. L'objectiu central d'aquest projecte és entendre i implementar en Python alguns algorismes per a calcular distribucions causals, tant condicionades com no condicionades. Per a fer-ho, en primer lloc presentem algunes definicions i propietats bàsiques de probabilitats i grafs, seguides d'una introducció a la teoria de la causalitat, on es mostren resultats importants per a la construcció dels algorismes. A continuació estudiarem en detall els algorismes d'identificació desenvolupats per Shpitser i Pearl l'any 2006 [SP 2006a, SP 2006b], tot explicant la nostra implementació en Python. L'algorisme d'identificació principal es pot entendre com l'aplicació reiterada de les regles del càlcul *do*, i finalment o bé retorna una expressió de la consulta causal en termes de probabilitats experimentals, o bé falla i determina que l'efecte és no identificable. També introduïm la llibreria de Python desenvolupada al llarg d'aquest projecte, i cap al final de la memòria en mostrem alguns exemples d'ús.

2010 Mathematics Subject Classification. 05C20, 05C85, 62-07, 62-09, 68R10, 68W01, 90C35.

Contents

Introduction

As a kid, I had a game designed to annoy my parents which consisted of repeatedly asking why. I started with a simple question, such as "*Why do I have to go to school?*" or "*Why should I wear pyjamas when sleeping?*", and then kept asking why at every response I received. Of course, this was not a game the first time I tried it, but it became one when I saw how it bothered my interlocutor. I am confident that every child has done this at least once, and this is because asking why is an innate aspect of the human condition. Asking ourselves why things happen is a distinct trait humanity has acquired after thousands of generations of evolution. Many millennia ago, humans began to realize that certain things cause other things to happen, and our ancestors took this opportunity to their advantage. As a consequence of this cognitive improvement, small villages were established, which then turned into towns and, much later on, were the foundations of the organized society we live in.

We use this causal-effect thinking subconsciously in our everyday routine. For instance, if I wake up early in the morning feeling tired I might ask myself why, what is the cause of this tiredness, and perhaps reach the conclusion that had I not been awake until so late watching TV last night I would not be feeling this way this morning. This causal reasoning is useful not only in small everyday decisions but also in other more professional fields like law or science. In legal procedures often the phrase *but for* is used in sentences like "*Our client would certainly be walking on its own, but for the car accident*", implying a causal relationship between the accident and the inability of the victim to walk properly. In science, more precisely in medicine, causal inference is especially significant. We are often listening to statements like how smoking can lead to the development of some cancers, or how a treatment increases the life expectancy of patients suffering from a certain illness. To be able to make these affirmations rigorously we have had to mathematize the concept of causality.

But how do we define *causality*? This philosophical concept has dazzled great minds for centuries, and its definition has been debated many times [PM 2018, Chapter 8]. A possible definition is that causality is the influence by which one event (a cause) contributes to the production of another event (an effect) where the cause is partly responsible for the effect, and the effect is partly dependent on the cause. Nevertheless, the concept and definition of causation is still an ongoing debate between contemporary philosophers, but is out of the scope of this dissertation. Instead, we are interested in how can we answer causal-effect questions, and a very helpful concept to have in mind when asking those questions is the *Ladder of Causation* [PM 2018, Chapter 1].

The Ladder of Causation is a metaphor to classify three distinct levels of cognitive ability: seeing, doing and imagining. It consists of three fundamentally different rungs:

Rung 1: Association. This is the first, most basic level of the Ladder of Causation, and it involves the observation of data and extraction of regularities from these observations. Examples would be how a dog figures out where a ball is going to land when its owner throws it at the park, or how IBM's Deep Blue analysed thousands of chess games to extract the moves associated with a higher percentage of wins. It is characterized by questions like "*What if I see...?*" or "*How would seeing X change my belief in Y?*". For instance, what does a survey tell us about the election results? All the questions related to this level of the Ladder of Causation can be answered using standard statistical methods. Note that we cannot answer causal queries, we can only make associations (like, for example, compute the correlation of variables). Many animals and present-day Artificial Intelligence algorithms are considered to be in this rung.

Rung 2: Intervention. The second level of the Ladder of Causation involves intervening or doing a certain action to produce the desired outcome. Examples would be when we take paracetamol to cure a headache (we are intervening on the amount of paracetamol in our body to produce a reduction in headache pain), or when we study to pass an exam (we act on the things we learn to produce a better mark in the exam). It is characterized by questions like "*What if I do...?*" or "*How would Y be if I do X?*". For instance, what would be my weight at the end of the year if I were to jog every day for thirty minutes? To answer questions in this rung of the Ladder of Causation we need to either physically perform the intervention or make use of the recently defined *do*-calculus (which will thoroughly be explained in this project). Unlike the first level, this one allows us to make causal associations between variables. Babies and also primitive humans that used intentionally-made tools are considered to be in this rung.

Rung 3: Counterfactuals. The highest level of the Ladder of Causation involves imagination and understanding because it compares our real world with an imaginary world. The real world is the world we live in when we do an action, and the imaginary, counterfactual world is the alternative reality in which my action would have been different. It is characterized by questions like "*What if I had done...?*" or "*If X had not occurred, would Y have happened?*". For instance, would the Theory of General Relativity had been created if Einstein had not existed? Humankind entered this rung of the Ladder when it started to imagine fictional things that they had not seen in real life before, such as divinities, religions or events that could have happened but did not. It is this counterfactual thinking that makes us different from all other intelligent life on Earth and helps us make decisions, by imagining all possible outcomes.

What is important of this ladder is that one cannot answer queries from a level with information of lower levels alone. For instance, to be able to determine causal effects we do not have enough with only observational data, we need something else from rung two of the Ladder of Causation (or above). We will use the so-called *do*-calculus to perform interventions to our probabilities, so instead of having $P(Y|X)$, which would be read as "*the probability of Y when X is seen*", we will have expressions of the form $P(Y|do(X))$, meaning "*the probability of Y when X is artificially imposed*".

This tool will allow us to compute causal effects from observational data, but it will not always work. There will be cases where the mental model of the problem will not allow us to compute these causal relationships, and we will be forced to either change the model or perform a physical intervention in a real-life experiment. An example where we cannot compute a causal effect between two variables X and Y is when there exist some background unmeasurable variables that affect both X and Y. In these cases where we cannot use *do*-calculus to obtain the causal effect, we say that the causal effect is *not identifiable* or *unidentifiable*.

The goal of this project is to address the identifiability problem (to detect in which cases we can identify a causal effect and in which cases we cannot). To do so we will study a few algorithms devised by Shpitser and Pearl [SP 2006a, SP 2006b] and implement them in Python, developing a package available for everyone in the scientific community to use. In this journey, we will also study thoroughly the necessary results used in these algorithms, and we will try to explain them in the most accessible way to reach the widest audience possible. To this end we have organized this dissertation as follows.

In the first chapter, we present a succinct historical background of causality, and we explain why this project is relevant and of general interest.

In chapter two we present important tools used in the context of causal theory. We first recall some basic probability theory definitions and theorems, before focusing on crucial aspects about graphs and more precisely about direct acyclic graphs (DAGs). We then enter the realm of probabilistic causal models and introduce *do*-calculus, an indispensable tool when querying causal effects from experimental observations. We present the identifiability problem, and we end the chapter by studying some criteria to identify causal effects through the so-called confounded components.

The third chapter is devoted to study and explain the implementation of some algorithms that can identify causal relationships from causal diagrams. We first present the **ID** algorithm, useful for unconditional causal queries, and we explain how we encode probability distributions and causal diagrams in our implementation of that algorithm. After having meticulously explored every line of the algorithm alongside its implementation, we introduce an algorithm to solve conditional causal effects, called **IDC**. We then explain how we have implemented it in our package, and give some examples of how to call these functions. We finalize this third chapter by concisely giving an idea of how algorithms for counterfactual queries may be constructed.

Some conclusions on this project are then presented, after which a link with the source code is provided.

Chapter 1

Why Is the Identification Problem Significant?

For decades causation was seen for most statisticians as a special case of correlation, and we owe this misleading association to the English statistician [1]Sir Francis Galton and especially his disciple, [2]Karl Pearson. Pearson strongly believed that with data and traditional statistical methods (such as the correlation of variables) one could explain causation. [3]Sewall Wright, an American geneticist, was against that belief and thought that in causal analysis one must incorporate some understanding of the process that produces the data. He applied this technique when he constructed a path diagram to quantify the influence of developmental factors in a guinea pig's womb on the colour of the fur of its offspring. This path diagram, seen now as one of the first causal diagrams (where arrows are drawn from causes to effects), was the kind of resource Pearson was against, for he stated that different (subjective) models would lead to different conclusions, and that was not rigorous. He could not stand this idea of introducing additional, biased information into the deduction process proposed by Wright, and opposed it outright.

His influence persisted, and it was not until the late 1980's that causal theory made a significant step forward. Judea Pearl, an Israeli-American philosopher and computer scientist, was studying how to manage uncertainty in artificial intelligence systems with Bayesian networks, but this approach could not solve causal-effect queries (recall that one cannot answer questions from rung two of the Ladder of Causation with just information about rung one). With this problem in mind, he then devoted the following years of his career to the formalization of causal theory, obtaining a methodology to compute, in some cases, causal effects from causal diagrams and observational data.

Before the mathematization of causal theory, some philosophers tried to express the sentence "X *causes* Y" as "X *raises the probability of* Y" by writing $P(Y|X) > P(Y)$, but this is wrong at its core. Note that "raises" is a causal concept from the second level of the Ladder of Causation, while the expression $P(Y|X) > P(Y)$ uses data from observations and thus lies on the first level. This inequality really affirms that "*if I see* X*, then the probability of* Y *increases*",

[1]Sir Francis Galton. English statistician, 1822 - 1911.

[2]Karl Pearson. English mathematician and biostatistician, 1857 - 1936.

[3]Sewall Green Wright. American geneticist, 1889 - 1988.

but this increase in probability could be for other reasons, like a third variable Z being the cause of X and Y.

According to Pearl, who introduced the *do*-operator, for X to be the cause of Y we need to state that "*doing X raises the probability of Y*", which would be written as $P(Y|do(X)) > P(Y)$. This concept of doing or intervening is from rung two, and thus we can borrow this operator to solve causal queries. Note that doing is fundamentally different from seeing: by *doing X* we do not care if a third variable is causing X and Y because it is I who is forcing the value on X and not some other background factor. If we conclude that the probability of Y while I force the value of X is bigger than without forcing it, then X is partially responsible for Y.

Before the definition of the *do*-operator we could not solve causal queries because we were simply not asking the correct questions, we did not have the necessary tools to even formulate them. This operator has not only allowed us to ask the right questions, but it has also provided us with a set of rules that can help us resolve these queries. These rules constitute what is known as *do*-calculus, and under some conditions, they can be used to compute causal effects from observational data. When this is possible, this is, when we can use *do*-calculus to compute the effect of a causal relationship, we say that this effect is *identifiable*.

This definition of identifiability is completely different from the one we have in statistics. In classical statistics, a statistical model $\mathcal{P} = \{P_\theta | \theta \in \Theta\}$ is *identifiable* if the mapping $\theta \mapsto P_\theta$ is one-to-one, this is, if for different values of the parameter we obtain different probability distributions. In simultaneous equations models, this problem of identification arises when the value of one or more parameters of the equations in the model cannot be determined from observable variables. Note that, in this context, identification depends profoundly on the equations of the model. The concept of identifiability that Pearl introduced does not depend on the form of the equations, but only on the relationship between variables. We will study the identification problem in detail in the following chapter.

But why is the identification problem relevant in the framework of causal models? When trying to compute a causal effect we could perform the actual intervention in the real world, fixing the value of a variable X and then measuring the other variable Y, and seeing if $P(Y|do(X)) > P(Y)$. This is not always feasible, sometimes because it is unethical or sometimes because it is simply not viable. Therefore it is of great importance to have a way of computing these interventions without having to actually perform them in reality. This became possible with the introduction of Pearl's *do*-calculus, but lacked a systematic way of calculating causal queries. Years later, a technique to mechanize the estimation of causal effects was eventually developed. This method takes shape as algorithms, designed by Shpitser and Pearl [SP 2006a, SP 2006b], that use the rules of *do*-calculus to compute a certain causal effect, when possible, and that raise an error when the causal effect is not identifiable.

Our project will consist of studying thoroughly the theory behind these algorithms to be able to implement them in Python, and developing a package to perform these causal effect calculations. This work is relevant because it gathers a set of recent results which are unknown to many computer scientists and statisticians in general. Most of them know about *do*-calculus, but some of them are unaware of the existence of deterministic algorithms that mechanize the process of computing causal effects. There are even some scientists who still think that identifiability and the calculation of causal effects is an open problem. Through

this project we want to reach more people, and to make the extraction of causal effects from observational data an effortless procedure.

There is already one implementation of these algorithms for R, by Tikka and Karavanen [TK 2017] under the name of `causaleffect`, but we believe that implementing them in Python, a very popular programming language amongst data scientists, will make them more known worldwide. According to the TIOBE index [TIOBE 2021], at the moment of this writing Python is the second most popular programming language in the whole world just after C, and R falls back to 14^{th} place, so we strongly believe that developing this package for Python will boost the popularity of the results by Shpitser and Pearl.

Additionally, with this dissertation we will also try to explain the results that support the algorithms designed by Shpitser and Pearl [SP 2006a, SP 2006b] in a more clear, understandable way. The notation used to formalize causal theory is effective and very well constructed, but lacks transparency, so we believe that, in order to be accessible to a wider audience, results have to be properly organized and formulated in a friendlier way. We tried to do so in this thesis by first introducing a few necessary concepts of probability and graph theory, before entering the world of causal theory.

Chapter 2

Background Theory

The main purpose of this chapter is to lay a foundation of definitions and results needed later on in the definition and discussion of the algorithms that are the main interest of this dissertation. Some of them can be found in a standard introductory probability book such as [Ash 1970], others in a basic graph theory book like [BLW 1986]. The more specific results on causal models and causal diagrams are available mostly in *Causality* [Pea 2000] by Judea Pearl and *Causal Inference in Statistics* [PGJ 2016] by Pearl *et al.*. But there are also some recent results included in this chapter cited from the original sources, such as [Pea 1993, Pea 1995, SP 2006a, Ver 1993, VP 1988], and for the curious reader, [PM 2018] is a great accessible book for wider audiences by the father of modern causality, Judea Pearl.

The first section will be focused on establishing and recalling some well-known probability definitions and basic theorems, for they will be used in the context of probabilistic causal models. Then some basic notion of graphs will be presented in the second section, given that certain types of graphs are an essential tool in causality. The chapter will end with the definition of causal models and causal diagrams, and with the introduction of a criterion for knowing if a causal effect is identifiable from a causal diagram.

2.1 Probability Theory

To be able to identify causal effects, we must first recall some basic results of probability theory. One could wonder why probability, a branch of mathematics that works with randomness and doubt, has anything to do with causality. Perhaps the most common answer would be that we live in a world surrounded by uncertainty, and in every chain of events there are observations we cannot make or factors we cannot control. For instance, the sentence "*if you don't study, you will fail the exam*" may be true most of the time, but there are unknown and noisy factors, like chance, luck, or recalling a single memory of that only class you attended, for example, that may influence the outcome of that exam. That is why the language of probabilities is used widely in science to model not only social sciences, but natural sciences as well, and why it is also used in causal theory.

Suppose we have an event A. Then the probability $P(A)$ is always bounded between 0 and 1, i.e., $0 \leq P(A) \leq 1$, where $P(A) = 0$ when that event is impossible and cannot happen, and

$P(A) = 1$ when A always happen. If we now have another event B, the expression $P(A, B)$ refers to the probability of both events happening.

A basic result of probability theory is the Law of Total Probability, which will be used to simplify probability expressions in upcoming sections.

Theorem 2.1. (Law of Total Probability) *Let A be an arbitrary event, and $B_1, \ldots, B_n$ mutually exclusive events such that $\sum_{i=1}^{n} P(B_i) = 1$. Then,*

$$P(A) = \sum_{i=1}^{n} P(A, B_i) \ .$$

If B is a binary event, then $P(A) = P(A, B) + P(A, \overline{B})$, where $\overline{B}$ is the complementary of B.

We can also wonder how an event happening influences the probability of another event. To deal with these dependent probabilities we must state some basic results on conditional probability.

Definition 2.2. (Conditional Probability and Independence) Let A, B two events. Then, the *conditional probability* of A under the condition B, denoted by $P(A|B)$, is the probability that the event A occurs given that the event B has already occurred. It can be computed from the probability of joint events,

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

which leads to a useful relation to keep in mind, $P(A, B) = P(A|B)P(B)$. We say that two events are *independent* if $P(A|B) = P(A)$, meaning that knowing about either event has no effect on the likelihood of the other. Using the last derived relation, independence can also be expressed as $P(A, B) = P(A)P(B)$.

Notation. Given two independent events A and B, we will write $A \perp\!\!\!\perp B$. If two events A and B are independent given a third event C, we will write $A \perp\!\!\!\perp B|C$.

This following result is extremely useful despite its simple formulation, and helps us change from conditioning on one variable to conditioning on another.

Theorem 2.3. (Bayes' Theorem) *Let A, B two different events with $P(B) \neq 0$. Then,*

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

Proof. From the definition of conditional probability, we have

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

□

Notation. We will write $P(x|y)$ as a shorthand for $P(X = x|Y = y)$.

Example 2.4. This example illustrates how we might use the previous stated results and definitions to simplify and rewrite probability expressions. Suppose we have the following expression: $\sum_w P(w|z)P(x|z,w)P(y|x,z,w)$. Then, making use of the derived expression of the conditional probability definition, we can write it as

$$\sum_w P(w|z)P(x|z,w)P(y|x,z,w) = \sum_w P(w|z)P(x,y|z,w) = \sum_w P(x,y,w|z) \ ,$$

and by the Law of Total Probability,

$$\sum_w P(x,y,w|z) = P(x,y|z) \ .$$

It is sometimes useful to decompose a joint probability into individual, conditional probabilities, because if one knows information about the independence between events this procedure may lead to simpler, easier expressions. Nevertheless, the decomposition is obviously not unique. For example, $P(x,y,z) = P(x)P(y,z|x) = P(x)P(y|x)P(z|x,y)$, but also $P(x,y,z) = P(z)P(y,x|z) = P(z)P(y|z)P(x|z,y)$.

This information about dependency between events can be visualized in a graphical form, by drawing probabilistic graphical models.

Definition 2.5. (Probabilistic Graphical Model) A *Probabilistic graphical model* is a probabilistic model for which a graph shows the conditional dependence between the random variables present in the model. An example would be a *Bayesian network*, which uses directed acyclic graphs to encode variable dependencies.

An example of a probabilistic graphical model is shown below.

Example 2.6. The directed graph in Figure 2.1 is an example of a probabilistic graphical model, particularly a Bayesian network. This graph encodes the dependencies between three binary variables: whether it is summer, whether it is sunny and whether I wear sunscreen.

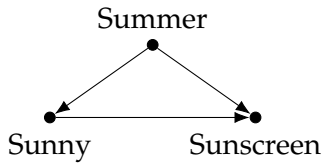


Figure 2.1: Probabilistic graphical model.

Clearly, the fact of being summer affects both being a sunny day and my decision to wear sunscreen, but this is not true in reverse: the season will not change depending on my decision of wearing sunscreen, nor today being sunny. Additionally, my choice of wearing sunscreen will also depend on the weather, hence the directed arrow from *Sunny* to *Sunscreen*, but I cannot change the weather by putting on some sunscreen (thus the lack of an arrow from *Sunscreen* to *Sunny*).

From a Bayesian network one can compute many things, but all from the first rung of the Ladder of Causation. In such models we only *observe* the variables of the model, and we do

not *intervene*, which, as we will see further down this chapter, is the key to solve causality, a concept from the second rung of the Ladder of Causation.

In the previous definition we have talked about directed acyclic graphs, and to understand what they are we must introduce some concepts on Graph Theory. These will help us not only to deal with Bayesian networks but also to establish the foundations of causality.

2.2 Basics of Graph Theory

A very useful tool when talking about causes and effects are graphs, more precisely directed graphs. Not only they are really intuitive to draw, but they are surprisingly powerful as well, as we will see throughout these following sections. The fact that even a six-year-old could sketch a causal diagram (directed graph where arrows imply causality from one node to another) and draw some basic conclusions from it makes causality often a mere puzzle. To understand how these causal diagrams are useful we first need to go through some results on graph theory.

Definition 2.7. (Graph) A *graph* is an ordered pair $G = (\boldsymbol{V}, \boldsymbol{E})$, where $\boldsymbol{V}$ is a finite not-empty set of *vertices* or *nodes* and $\boldsymbol{E} \subset \boldsymbol{V} \times \boldsymbol{V}$ a set of *edges* or *links* that connect some pairs of vertices. Two nodes X and Y connected by an edge are called *adjacent*, and we say that X and Y are *neighbours*. When the edges are ordered, represented by $X \to Y := (X, Y)$, we have a *directed graph*. If X and Y are two nodes connected by a directed edge from X to Y ($X \to Y$), we say X is the *parent* of Y and Y is a *child* of X.

We can also create new, smaller graphs by selecting only some nodes and edges of a bigger graph.

Definition 2.8. (Subgraph and Induced Subgraph) Let $G = (\boldsymbol{V}, \boldsymbol{E})$, $G' = (\boldsymbol{V'}, \boldsymbol{E'})$ be two graphs such that $\boldsymbol{V'} \subset \boldsymbol{V}$ and $\boldsymbol{E'} \subset \boldsymbol{E} \cap (\boldsymbol{V'} \times \boldsymbol{V'})$. Then, G' is a subgraph of G, and we denote it by $G' \subset G$. If the equality $\boldsymbol{E'} = \boldsymbol{E} \cap (\boldsymbol{V'} \times \boldsymbol{V'})$ holds, then G' is the *node induced subgraph* by the set $\boldsymbol{V'}$, and we write $G' = G[\boldsymbol{V'}]$.

It is usually convenient to study not only links between two nodes, but also the *paths* between two non-adjacent nodes that are further away.

Definition 2.9. (Path) Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a directed graph and $u, v \in \boldsymbol{V}$ two vertices. Consider a sequence of vertices $p = \{u = X_1, X_2, \ldots, X_k, X_{k+1} = v\}$, $X_i \in \boldsymbol{V}$ $\forall i$, such that

(a) every pair of consecutive nodes is an edge, i.e., $(X_i, X_{i+1}) \in \boldsymbol{E}$ or $(X_{i+1}, X_i) \in \boldsymbol{E}$,

(b) all edges joining consecutive nodes in p are different, and

(c) all vertices (except $u = X_1$ and $v = X_{k+1}$) are different.

Then, p is a *path* from u to v (or from v to u). A path that starts and ends at the same node ($u = v$) is called a *cycle*. If any node in the path does not have two incoming or outgoing edges, u has an outgoing edge and v and incoming edge, we say it is a *directed path from u to v*. If a path is a directed path that starts and ends at the same node ($u = v$), then it is called a *directed cycle*. A path from u to v is called a *back-door* path if it contains an arrow into u.

A notation that will be exhaustively used all through this project is the concept of parents and children of nodes in a directed graph.

Definition 2.10. (Parents, Children, Ancestors and Descendants) Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a directed graph and $\boldsymbol{X} \subset \boldsymbol{V}$ a subset of nodes. Then, the *parents* of $\boldsymbol{X}$, denoted by $\text{Pa}(\boldsymbol{X})_G$, is the set consisting of the parents of every node in $\boldsymbol{X}$ while also containing $\boldsymbol{X}$. Analogously, the *children* of $\boldsymbol{X}$, denoted by $\text{Ch}(\boldsymbol{X})_G$, is the set consisting of the children of every node in $\boldsymbol{X}$ while also containing $\boldsymbol{X}$. A node Y is an *ancestor* of a node $X_i \in \boldsymbol{X}$ if there exists a directed path $p \subset G$ from Y to X_i, and the set of all ancestors of $\boldsymbol{X}$ while also containing $\boldsymbol{X}$ is denoted by $\text{An}(\boldsymbol{X})_G$. A node Y is a *descendant* of a node $X_i \in \boldsymbol{X}$ if there exists a directed path $p \subset G$ from X_i to Y, and the set of all descendants of $\boldsymbol{X}$ while also containing $\boldsymbol{X}$ is denoted by $\text{De}(\boldsymbol{X})_G$. When possible we will omit the subscript G to ease comprehension.

Remark 2.11. Given a directed graph $G = (\boldsymbol{V}, \boldsymbol{E})$ and $\boldsymbol{X} \subset \boldsymbol{V}$, it is clear that $\boldsymbol{X} \subset \text{Pa}(\boldsymbol{X}) \subset \text{An}(\boldsymbol{X})$ and $\boldsymbol{X} \subset \text{Ch}(\boldsymbol{X}) \subset \text{De}(\boldsymbol{X})$.

Another useful set of vertices of a graph is the so-called *root set*, composed by those vertices with no descendants other than themselves.

Definition 2.12. (Root Set) Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a directed graph. Then the *root set of* G is the set of nodes with no descendants, $\text{Rt}(G) = \{X \in \boldsymbol{V} \mid \text{De}(X)_G \setminus X = \varnothing\}$.

As we will see down this chapter, performing interventions will have the effect of erasing some directed edges of a graph. To this end we present the following notation.

Notation. Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a directed graph and $\boldsymbol{X} \subset \boldsymbol{V}$. Then we will denote by $G_{\overline{\boldsymbol{X}}}$ (resp. $G_{\underline{\boldsymbol{X}}}$) the graph obtained from G by removing all incoming (resp. outgoing) edges of $\boldsymbol{X}$.

There is a special family of directed graphs that turns out to be very handy when dealing with causal effects, presented below.

Definition 2.13. (Directed Acyclic Graph) A graph that contains no cycles is called *acyclic*. A directed graph which has no directed cycles is called *directed acyclic graph (DAG)*. These last structures will be used throughout the dissertation, since they are a fundamental part of causal theory.

Remark 2.14. Every DAG G has a non empty root set, $\text{Rt}(G) \neq \varnothing$. Note that if $\text{Rt}(G) = \varnothing$, then every node of G would have at least one child, and since the set of vertices is finite, it would contain a cycle.

Definition 2.15. (Topological Ordering) Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a DAG. Then a *topological ordering* π of G is an ordering of its nodes, where for all pair of nodes $X, Y \in \boldsymbol{V}$ with $X \neq Y$ one has $X > Y$ or $Y > X$ such that if X is an ancestor of Y in G, then $X < Y$.

Example 2.16. To take all these definitions in, consider the following example. In Figure 2.2 (a) G is a directed acyclic graph, DAG. One possible path in G could be $\{X, Y, Z\}$, which could also be represented as $X \rightarrow Y \leftarrow Z$, and a directed path could be $\{Z, X, W, Y\}$, $Z \rightarrow X \rightarrow W \rightarrow Y$. Since we will mostly work with directed graphs, from now on we will

represent paths making use of the latter representation, i.e., specifying the directions. The parents of X are $\mathrm{Pa}(X) = \{X, Z\}$, the ancestors of W are $\mathrm{An}(W) = \{W, X, Z\}$, the descendants of W are $\mathrm{De}(W) = \{W, Y\}$ and the root set of this graph is just $\mathrm{Rt}(G) = \{Y\}$. A topological ordering of G would be $\{Z, X, W, Y\}$. Figure 2.2 (b) represents graph H, which is the induced subgraph of G by the set $\{X, Y, Z\}$.

(a) Graph G.

(b) Induced subgraph $H = G[\{X, Y, Z\}]$.

Figure 2.2: Examples of directed acyclic graphs (DAG).

Definition 2.17. (Connected Graph) Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a directed graph. Then, G is *connected* if there exists a path $p \subset G$ between every pair of nodes $X, Y \in \boldsymbol{V}$.

At this point we are able to consider an example of a Bayesian network and how it may answer some questions.

Example 2.18. Let us retrieve the probabilistic graphical model in Example 2.6. It is the same graph as H in Figure 2.2 (b), with renamed variables: Z is the binary variable "*Summer/Not summer*", X the binary variable "*Sunny/Not sunny*" and Y the binary variable "*I wear sunscreen/I do not wear sunscreen*". It is reasonable to think that the season of the year strongly affects the weather on a particular day ($Z \to X$), and also the probability of me wearing sunscreen ($Z \to Y$). Additionally, it is feasible to think that the decision of whether or not I must use sunscreen also depends on the weather of that particular day ($X \to Y$). The conditional probabilities between these random variables can be found in Table 2.3. Now, decomposing $P(X, Y, Z)$ into $P(X, Y, Z) = P(Y|X, Z)P(X|Z)P(Z)$, we can use the given conditional probabilities to compute, for example, what is the probability of being summer knowing I do wear sunscreen:

Summer	
Yes	No
0.25	0.75

	Sunny	
Summer	Yes	No
Yes	0.90	0.10
No	0.70	0.30

		Sunscreen	
Sunny	Summer	Yes	No
Yes	Yes	0.99	0.01
Yes	No	0.20	0.80
No	Yes	0.60	0.40
No	No	0.05	0.95

Table 2.3: Conditional probability tables of the different variables of the Bayesian network in example 2.18.

$$P(Z = \mathrm{T}|Y = \mathrm{T}) = \frac{P(Y = \mathrm{T}, Z = \mathrm{T})}{P(Y = \mathrm{T})} = \frac{\sum_{x \in \{\mathrm{T},\mathrm{F}\}} P(X = x, Y = \mathrm{T}, Z = \mathrm{T})}{\sum_{x,z \in \{\mathrm{T},\mathrm{F}\}} P(X = x, Y = \mathrm{T}, Z = z)} =$$
$$\frac{0.99 \cdot 0.90 \cdot 0.25 + 0.60 \cdot 0.10 \cdot 0.25}{0.99 \cdot 0.90 \cdot 0.25 + 0.60 \cdot 0.10 \cdot 0.25 + 0.20 \cdot 0.70 \cdot 0.75 + 0.05 \cdot 0.30 \cdot 0.75} = \frac{0.23775}{0.354} \approx 0.67.$$

Therefore, the probability that it is summer knowing I wear sunscreen is around 67%, according to the conditional probabilities shown in Table 2.3 and the dependencies encoded in the Bayesian network in Figure 2.2 (b).

Bayesian networks are a powerful tool to compute probabilities from a dependency graph of variables, but they do not shed light on the problem of causality. To understand how we can tackle such a puzzle we must introduce more results, starting with causal models.

2.3 Causal Models

Causal models are a mathematical representation that will help us solve how a first action causes a second, and are the central construction of the algorithms that will be analysed in the following chapter. But before formally defining what a causal model is, consider this plausible made-up example.

Example 2.19. Suppose we want to know how the salary of an employee in a given company depends on the gender of the worker. It is reasonable to suppose that salary (Y) could depend on the level of education (E) of the employee (better academic records usually translate to higher payroll), the field the employee is working in (F) and the amount of time he has been working for the company (S for seniority). Clearly, the age (A) of the worker influences both the level of education and the seniority, and gender (X) may have an impact on the seniority as well as the field of the employee. In addition, we might think that both age and gender may be related through a third unobserved variable (U), which could be that perhaps the company used to hire only men in the past, and thus the average age of men is higher than that of women in the company.

The so-called *exogenous variables* would be the unobserved, unmeasurable variables, U in this example, and the variables Y, E, F, S, A, X are called *endogenous variables*. These variables, together with the relations of dependence described above, form what is known as *causal model*.

Definition 2.20. (Causal Model) A *causal model* is a triple $M = (\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{F})$, where:

(a) $\boldsymbol{U}$ is a set of background random variables, called exogenous variables, determined by factors from outside the model.

(b) $\boldsymbol{V} = \{V_1, \ldots, V_n\}$ is a set of random variables, called endogenous variables, that are determined by variables in the model, i.e., by variables in $\boldsymbol{U} \cup \boldsymbol{V}$.

(c) $\boldsymbol{F} = \{f_i, \ldots, f_n\}$ is a set of functions such that for every $V_i \in \boldsymbol{V}$, there is a mapping $f_i : \boldsymbol{S_i} \cup \{U_{V_i}\} \to V_i$, and such that the whole set $\boldsymbol{F}$ forms a mapping from $\boldsymbol{U}$ to $\boldsymbol{V}$. That is, for every $V_i \in \boldsymbol{V}$ there is a mapping (named structural equation) $f_i \in \boldsymbol{F}$ such that

$$V_i = f_i(\boldsymbol{S_i}, U_{V_i}),\ i \in \{1, \ldots, n\}\ ,$$

where $U_{V_i} \in \boldsymbol{U}$ is the error term linked to V_i, and $\boldsymbol{S_i} \subset (\boldsymbol{U} \cup \boldsymbol{V}) \setminus \{V_i, U_{V_i}\}$, known as the *parent set*.

It is important to emphasise the concept of exogenous variables. Those variables can affect endogenous variables, which are measurable in our model, but we cannot see nor measure said exogenous variables. They encompass all kinds of unmeasurable perturbations, including the small deviations due to error terms or noise.

Every causal model M has its corresponding directed acyclic graph $G = (\boldsymbol{W}, \boldsymbol{E})$, where the node set $\boldsymbol{W} = \boldsymbol{U} \cup \boldsymbol{V}$ contains a node for each observed (endogenous, $\boldsymbol{V}$) and unobserved (exogenous, $\boldsymbol{U}$) variable of the model. We usually ignore the unobservable vertices U_{V_i} correspondent to the error term of measurable variables, for we know that they are always there and are implicitly taken into account in the model. Then, the set of edges $\boldsymbol{E}$ is determined by the functional relationships between the variables in the model, meaning that $\boldsymbol{E}$ contains an edge coming into V_i from every node required to uniquely define f_i. This graph G is known as the *causal diagram* of M.

Example 2.21. The DAG induced by the the causal model in Example 2.19 can be seen in Figure 2.4.

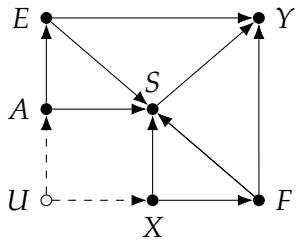


Figure 2.4: DAG G encodes the relations between variables of Example 2.19.

Definition 2.22. (Probabilistic Causal Model) A *probabilistic causal model* is a tuple $M = (\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{F}, P(\boldsymbol{U}))$, where $(\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{F})$ is a causal model and $P(\boldsymbol{U})$ is the joint distribution of the variables in $\boldsymbol{U}$. The distribution on $\boldsymbol{V}$ induced by $P(\boldsymbol{U})$ and $\boldsymbol{F}$ will be denoted $P(\boldsymbol{V})$.

Definition 2.23. (Semi-Markovian Causal Model) Given a causal model M, if every unobserved node is a parent of exactly two observed nodes, then M is called a *semi-Markovian causal model*.

We are going to focus only on semi-Markovian causal models, since by the result proved by Verma [Ver 1993], any causal model with unobserved variables can be redesigned into a semi-Markovian causal model while conserving all dependencies between variables.

Figure 2.5: (a) U is a confounder of nodes X and Y. (b) We will usually denote confounded nodes with a dashed bidirected edge.

Figure 2.5 (a) shows how unobserved nodes in semi-Markovian causal models are often represented. In this example, node U would be an exogenous variable with two children, X

and Y. Despite exogenous variables can be explicitly drawn in the causal, we will usually omit unobserved variables, since we cannot measure nor control them. We will represent their effect with a dashed bidirected edge between nodes X and Y, which corresponds to the effect of an unobserved confounding variable, this is, a hidden common cause. Note that this bidirected edge is not equivalent to two directed edges $X \to Y$ and $X \leftarrow Y$, as this would form a cycle in the graph which is not allowed in DAGs.

Definition 2.24. (*d*-separation) Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a DAG, p a path in G and $\boldsymbol{Z}$ a set of nodes $\boldsymbol{Z} \subset \boldsymbol{V}$. Then, the path p is *d-separated* by the set $\boldsymbol{Z}$ in G if and only if either

(a) p contains a chain $I \to M \to J$ or a fork $I \leftarrow M \to J$, such that $M \in \boldsymbol{Z}$ and $I, J \in \boldsymbol{V}$, or

(b) p contains an inverted fork or collider $I \to M \leftarrow J$, such that $\mathrm{De}(M)_G \cap \boldsymbol{Z} = \varnothing$.

Two disjoint sets $\boldsymbol{X}$ and $\boldsymbol{Y}$ are *d-separated* by $\boldsymbol{Z}$ in G if all paths from $\boldsymbol{X}$ to $\boldsymbol{Y}$ are d-separated by $\boldsymbol{Z}$ in G. A path that is not d-separated is said to be *d-connected*.

The following is an important result proved by Verma and Pearl [VP 1988], although we present the clearer and more succinct version proposed by Shpitser and Pearl in [SP 2006a].

Theorem 2.25. (Theorem 1 in [SP 2006a]**)** *Let M be a causal model with the corresponding DAG $G = (\boldsymbol{V}, \boldsymbol{E})$, and $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z} \subset \boldsymbol{V}$ be sets of variables or nodes in G. If $\boldsymbol{X}$ and $\boldsymbol{Y}$ are d-separated by $\boldsymbol{Z}$, then $\boldsymbol{X}$ is independent of $\boldsymbol{Y}$ given $\boldsymbol{Z}$ in G, i.e., $(\boldsymbol{X} \perp\!\!\!\perp \boldsymbol{Y} | \boldsymbol{Z})_G$.*

Example 2.26. Consider the DAG G shown in Figure 2.6, from [Pea 2000, p.p. 17-18]. There are two different paths between X and Y in G, the one that uses the bidirected arc, $X \to Z_1 \dashleftarrow\dashrightarrow Z_3 \leftarrow Y$, and $X \to Z_1 \leftarrow Z_2 \leftarrow Z_3 \leftarrow Y$. Note that, without measuring $\{Z_1, Z_2, Z_3\}$,

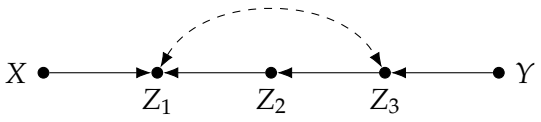


Figure 2.6: DAG G showing d-separation in Example 2.26.

X and Y are d-separated, since both paths have a collider ($X \to Z_1 \leftarrow U$, where U is a confounder of Z_1 and Z_3, and $X \to Z_1 \leftarrow Z_2$). Nevertheless, when Z_1 is measured the path $X \to Z_1 \dashleftarrow\dashrightarrow Z_3 \leftarrow Y$ becomes unblocked. This is so because measuring $\boldsymbol{Z} = \{Z_1\}$ unblocks both colliders at Z_1 and Z_3: at $X \to Z_1 \leftarrow U$ we have $\mathrm{De}(Z_1) \cap \{Z_1\} = \{Z_1\} \neq \varnothing$, and at $U \to Z_3 \leftarrow Y$ we have $\mathrm{De}(Z_3) \cap \{Z_1\} = \{Z_1, Z_2, Z_3\} \cap \{Z_1\} \neq \varnothing$, and thus none of the conditions in Definition 2.24 are met, making X and Y d-connected given Z_1.

2.3.1 Causal Effects, *do*-Calculus and Identifiability

Does smoking cigarettes increase the likelihood of developing lung cancer? This apparently obvious question was not that obvious sixty years ago before the foundations of causality were established. A typical argument against the claim that smoking caused lung cancer was that it could be an unknown "evil" gene confounding the variables "Smoking" and "Lung cancer", fully accounting for the correlation between both variables. Such a gene would make

an individual crave nicotine (and thus consume more tobacco) while at the same time would make them prone to developing lung cancer (see Figure 2.7 (b)).

Figure 2.7: (a) Causal diagram where Smoking is causally related to Lung cancer, while confounded by an "Evil" gene. (b) Causal diagram where Lung cancer is not a causal effect of Smoking.

A randomized controlled trial (RTC) was (and it still is) usually performed in similar problems (for example, in drug testing) to get rid of confounders or other sources of bias. This experiment consists of separating the subjects into two or more groups *randomly* and treating them differently, so one can be sure that the differences between them after the experiment are only a product of the different treatments given. The key point in these trials is the random selection process, which eliminates (or at least reduces) the biases of known and unknown factors. In our situation performing an RTC would not be possible: if both groups were separated by mere observation and not randomness it would not be an RTC because the "evil" gene would be present on most of the smokers, thus inducing a higher probability of cancer in that group. On the other hand, creating the groups at random and forcing people to smoke for twenty years to see their evolution and harming their health in the process is greatly unethical.

We have already stated that to solve problems from the second level of the Ladder of Causation (such as causal effects queries) we must fix, act on some variables to remove confounder effects. So the point now is, how can we intervene in an experiment without actually physically doing so? The so-called *do-operator* can solve this issue.

Definition 2.27. (*do*-operator) Let $M = (\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{F}, P(\boldsymbol{U}))$ be a probabilistic causal model, and $\boldsymbol{X}$ a set of variables of the model. Then the *do-operator* is the intervention that sets the values of $\boldsymbol{X}$ to $\boldsymbol{x}$, and it is denoted by $do(\boldsymbol{X} = \boldsymbol{x})$. This is, every action $do(\boldsymbol{X} = \boldsymbol{x})$ on M produces a new model $M_{\boldsymbol{x}} = (\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{F}_{\boldsymbol{x}}, P(\boldsymbol{U}))$, where $\boldsymbol{F}_{\boldsymbol{x}}$ is obtained by, for every $X \in \boldsymbol{X}$, replacing $f_X \in \boldsymbol{F}$ with a new constant function of value x given by $do(\boldsymbol{X} = \boldsymbol{x})$.

Remark 2.28. Despite being apparently similar, we must not misinterpret intervening for conditioning over a variable. $P(Y = y|X = x)$ is the probability that $Y = y$ conditional on finding $X = x$, in other words, we are considering the distribution of Y among individuals whose X value is x. $P(Y = y|do(X = x))$, however, is the probability that $Y = y$ when we intervene to make $X = x$, namely, we are now considering the distribution of Y if every individual in the population had their X value fixed at x.

Definition 2.29. (Causal Effect) Let $M = (\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{F}, P(\boldsymbol{U}))$ be a probabilistic causal model and $\boldsymbol{X}, \boldsymbol{Y} \subset \boldsymbol{V}$. Then the *causal effect* of $do(\boldsymbol{X} = \boldsymbol{x})$ on $\boldsymbol{Y}$ in M is the marginal distribution of $\boldsymbol{Y}$ in $M_{\boldsymbol{x}}$, noted by $P(\boldsymbol{Y}|do(\boldsymbol{X} = \boldsymbol{x})) = P_{\boldsymbol{x}}(\boldsymbol{Y})$.

Remark 2.30. For every intervention $do(\boldsymbol{X} = \boldsymbol{x})$, to ensure that $P_{\boldsymbol{x}}(\boldsymbol{V})$ and its marginals are well defined, is required that $P(\boldsymbol{x}|\,\mathrm{Pa}(\boldsymbol{X})_G \setminus \boldsymbol{X}) > 0$. It is not possible to force $\boldsymbol{X}$ to have values not observed in the data.

A special case of causal effects are direct effects, where the intervened variables are the parents of the studied variable.

Definition 2.31. (Direct Effect) Given a probabilistic causal model M with variables $\boldsymbol{V}$ and $\boldsymbol{Y} \subset \boldsymbol{V}$, a *direct effect* is a causal effect of the form $P(\boldsymbol{Y}|do(\mathrm{Pa}(\boldsymbol{Y}) \setminus \boldsymbol{Y} = \boldsymbol{y'}))$, this is, when the parents of the variables are the ones intervened.

The ultimate goal of solving a causal effect problem of some variables $\boldsymbol{X}$ over $\boldsymbol{Y}$ given a causal model M has therefore been reduced to finding the probability $P_{\boldsymbol{x}}(\boldsymbol{Y})$. Nevertheless, how can we obtain a value for this probability given that, most of the time, we will only have access to data from observational studies (information of the first rung of the Ladder of Causation)? In 1993 the computer scientist and philosopher Judea Pearl [Pea 1993] showed that, when some conditions are fulfilled, the intervening probability can be computed from just observational data, making use of the so-called *back-door criterion*.

Definition 2.32. (Back-Door Criterion) Let $M = (\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{F}, P(\boldsymbol{U}))$ be a probabilistic causal model with DAG G and $\boldsymbol{Z} \subset \boldsymbol{V}$, $X_i, X_j \in \boldsymbol{V}$ with $X_i \neq X_j$. Then, we say that $\boldsymbol{Z}$ satisfies the *back-door criterion* relative to (X_i, X_j) if:

(a) $\boldsymbol{Z} \cap \mathrm{De}(X_i) = \varnothing$, and

(b) $\boldsymbol{Z}$ blocks every path between X_i and X_j that contains an arrow into X_i (back-door path).

More generally, if $\boldsymbol{X}, \boldsymbol{Y} \subset \boldsymbol{V}$ with $\boldsymbol{X} \cap \boldsymbol{Y} = \varnothing$, then $\boldsymbol{Z}$ satisfies the *back-door criterion* relative to $(\boldsymbol{X}, \boldsymbol{Y})$ if it satisfies the criterion for every pair $(X_i, X_j) \in \boldsymbol{X} \times \boldsymbol{Y}$.

If such a condition is fulfilled, then the next result follows.

Theorem 2.33. (Back-Door Adjustment [Pea 1993]**)** *If a set of variables $\boldsymbol{Z}$ satisfies the back-door criterion relative to $(\boldsymbol{X}, \boldsymbol{Y})$, then the causal effect of X on Y is given by*

$$P_{\boldsymbol{x}}(\boldsymbol{y}) = \sum_{z} P(\boldsymbol{y}|\boldsymbol{x}, z)P(z).$$

Although this criterion was a big step in the right direction, it couldn't be applied to every scenario, so more adjustments like the previous were required to account for all possible causal diagrams. This dream of his of obtaining causal information such as $P_{\boldsymbol{x}}(\boldsymbol{Y})$ from observational data was finally a reality thanks to the development of *do-calculus* by Pearl and some of his colleagues in 1995 [Pea 1995]. His theory is constructed on three simple, yet powerful rules that allow the removal of the $do-$operator under some specific scenarios, thus letting one travel from unmeasurable probabilities involving *do* expressions to observational, standard probabilities. These rules were first proven by Pearl himself in [Pea 1995].

Theorem 2.34. (Rules of *do*-Calculus [Pea 2000, Theorem 3.4.1]**)** *Let $M = (\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{F}, P(\boldsymbol{U}))$ be a probabilistic causal model and G its associated DAG. For any pairwise disjoint subsets of nodes $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z} \subset \boldsymbol{V}$, the following rules apply:*

Rule 1 *(Insertion and deletion of observations)*

$$P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{z},\boldsymbol{w}) = P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{w}), \quad \textit{if} \quad (\boldsymbol{Y} \perp\!\!\!\perp \boldsymbol{Z}|\boldsymbol{X},\boldsymbol{W})_{G_{\overline{\boldsymbol{X}}}}$$

Rule 2 *(Exchanging actions and observations)*

$$P_{\boldsymbol{x},\boldsymbol{z}}(\boldsymbol{y}|\boldsymbol{w}) = P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{z},\boldsymbol{w}), \quad \textit{if} \quad (\boldsymbol{Y} \perp\!\!\!\perp \boldsymbol{Z}|\boldsymbol{X},\boldsymbol{W})_{G_{\overline{\boldsymbol{X}},\underline{\boldsymbol{Z}}}}$$

Rule 3 *(Insertion and deletion of actions)*

$$P_{\boldsymbol{x},\boldsymbol{z}}(\boldsymbol{y}|\boldsymbol{w}) = P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{w}), \quad \textit{if} \quad (\boldsymbol{Y} \perp\!\!\!\perp \boldsymbol{Z}|\boldsymbol{X},\boldsymbol{W})_{G_{\overline{\boldsymbol{X}},\overline{Z(\boldsymbol{W})}}}$$

where $Z(\boldsymbol{W}) = \boldsymbol{Z} \setminus \mathrm{An}(\boldsymbol{W})_{G_{\overline{\boldsymbol{X}}}}$*, i.e., the set of nodes in* $\boldsymbol{Z}$ *that are not ancestors of any node in* $\boldsymbol{W}$ *in* $G_{\overline{\boldsymbol{X}}}$*.*

Remark 2.35. Rule 1 of insertion and deletion of observations is a generalization of *d*-separation (Theorem 2.25) in a graph with interventions (that is why independence in $G_{\overline{\boldsymbol{X}}}$ is required). Rule 2 of exchanging actions and observations is a generalization of the back-door criterion: the only paths in DAG $G_{\overline{\boldsymbol{X}},\underline{\boldsymbol{Z}}}$ between $\boldsymbol{Z}$ and $\boldsymbol{Y}$ are back-door paths, and if we block those paths conditioning over $\boldsymbol{X}$ and $\boldsymbol{W}$ we can swap the intervention for the observation. Rule 3 provides conditions for introducing or deleting other interventions.

These rules have proven to be enough to compute interventional probabilities whenever these probabilities can be identified. Informally speaking, when it is possible to compute the interventional term of a distribution from just observational data, we say that the effect is *identifiable*.

Theorem 2.36. (*do*-Calculus Completeness [SP 2006a, Theorem 7]**)** *The three rules of do-calculus, together with standard probability manipulations, are complete for determining identifiability of all effects of the form* $P_{\boldsymbol{x}}(\boldsymbol{Y})$*.*

So we know that, if the effect is identifiable, using only the three rules of *do*-calculus we can obtain an algebraic expression for $P_{\boldsymbol{x}}(\boldsymbol{Y})$ which does not involve the *do*-operator, meaning that we can use only observable data to infer causal conclusions. But what does it really mean for an effect to be *identifiable*?

Definition 2.37. (Causal Effect Identifiability) Let $M = (\boldsymbol{U},\boldsymbol{V},\boldsymbol{F},P(\boldsymbol{U}))$ be a probabilistic causal model with DAG G, and $\boldsymbol{X},\boldsymbol{Y} \subset \boldsymbol{V}$. The causal effect of an action $do(\boldsymbol{x})$ on $\boldsymbol{Y}$ such that $\boldsymbol{X} \cap \boldsymbol{Y} = \varnothing$ is said to be *identifiable* from P in G if $P_{\boldsymbol{x}}(\boldsymbol{Y})$ is uniquely computable from $P(\boldsymbol{V})$ in any causal model which induces G. This is, if for every pair of causal models M^1 and M^2 such that $P^1(\boldsymbol{V}) = P^2(\boldsymbol{V})$, the causal effect coincides, $P^1_{\boldsymbol{x}}(\boldsymbol{Y}) = P^2_{\boldsymbol{x}}(\boldsymbol{Y})$.

In the following example we will use *do*-calculus to determine the causal effect of a very similar causal diagram to Figure 2.7 (a).

Example 2.38. Consider the causal diagram G shown in Figure 2.8 (a), where the variables S, T and C stand for Smoking, Tar and Cancer. This causal diagram is a slight variation of the one shown in Figure 2.7 (a). Here, the main modification of the model is that we are

Figure 2.8: (a) Causal diagram where Smoking is causally related to Lung cancer through Tar, and Smoking and Cancer confounded. (b) Same causal diagram but we explicit the unobserved confounder for ease when applying *do*-calculus.

supposing that lung cancer only develops through tar deposited in the lungs (and not from smoking directly), which in turn is *only* produced by smoking. In this scenario, Tar is called a *mediator* between S and C, because it is the variable that explains the causal effect of S on C.

Our goal is to determine the causal effect $P_{\boldsymbol{s}}(c)$ from Figure 2.8. First of all, we use the Law of Total Probability, followed by some conditional dependence manipulations (similar to Example 2.4):

$$P_{\boldsymbol{s}}(c) = P(c|do(s)) = \sum_t P(c,t|do(s)) = \sum_t P(c|t,do(s))P(t|do(s))$$

Then we apply rule 2 of do calculus, exchanging the t for $do(t)$ in the first term, since $(C \perp\!\!\!\perp T|S)_{G_{\overline{S},\underline{T}}}$ (see Figure 2.9 (a)):

$$\sum_t P(c|t,do(s))P(t|do(s)) = \sum_t P(c|do(t),do(s))P(t|do(s))$$

Applying rule 2 again we can change $do(s)$ for s in the second term, because $(T \perp\!\!\!\perp S)_{G_{\underline{S}}}$ (see Figure 2.9 (b)). The fact that $(T \perp\!\!\!\perp S)_{G_{\underline{S}}}$ follows from Theorem 2.25, since T and S are d-separated in $G_{\underline{S}}$ by the collider C:

$$\sum_t P(c|do(t),do(s))P(t|do(s)) = \sum_t P(c|do(t),do(s))P(t|s)$$

We now see that $(C \perp\!\!\!\perp S|T)_{G_{\overline{T},\overline{S}}}$ (see Figure 2.9 (c)), so we can apply rule 3 by deleting the intervention $do(s)$ from the first term:

$$\sum_t P(c|do(t),do(s))P(t|s) = \sum_t P(c|do(t))P(t|s)$$

Using some probability axioms as in the first step, we can write:

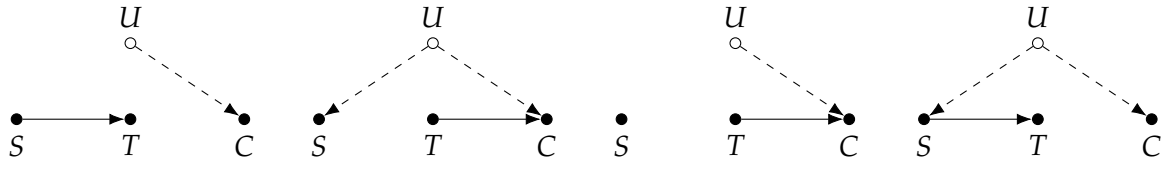
$$\sum_t P(c|do(t))P(t|s) = \sum_{s'}\sum_t P(c,s'|do(t))P(t|s) = \sum_{s'}\sum_t P(c|do(t),s')P(s'|do(t))P(t|s)$$

Using Theorem 2.25 once again we see that $(C \perp\!\!\!\perp T|S)_{G_{\underline{T}}}$ (see Figure 2.9 (d)), because the chain $U \to S \to T$ while conditioning on S d-separates the only path between C and T. So using rule 2 once more we can replace $do(t)$ for t in the first term:

$$\sum_{s'}\sum_{t} P(c|do(t),s')P(s'|do(t))P(t|s) = \sum_{s'}\sum_{t} P(c|t,s')P(s'|do(t))P(t|s)$$

Finally we are able to delete the only *do*-expression in the second term, $do(t)$, by using rule 3, because $G_{\underline{S}} = G_{\overline{T}}$ and we have seen previously that $(S \perp\!\!\!\perp T)_{G_{\overline{T}}}$ (see Figure 2.9 (b)):

$$\sum_{s'}\sum_{t} P(c|t,s')P(s'|do(t))P(t|s) = \sum_{s'}\sum_{t} P(c|t,s')P(s')P(t|s) = \sum_{t} P(t|s)\left(\sum_{s'} P(c|t,s')P(s')\right).$$



(a) $G_{\overline{S},\underline{T}}$. (b) $G_{\underline{S}} = G_{\overline{T}}$. (c) $G_{\overline{T},\overline{S}}$. (d) $G_{\underline{T}}$.

Figure 2.9: Different subgraphs used throughout Example 2.38.

So we conclude that, in the causal diagram shown in Figure 2.8, the causal effect $P_{\boldsymbol{s}}(c)$ is identifiable and can be computed from observational data as

$$P_{\boldsymbol{s}}(c) = \sum_{t} P(t|s)\left(\sum_{s} P(c|t,s)P(s)\right). \tag{2.1}$$

The causal diagram in Figure 2.8 appears often in bigger causal diagrams, and the formula obtained to identify the causal effect in Equation 2.1 is known as the *Front Door Adjustment*. The diagrams shown in Figure 2.8 and the resolution applying *do*-calculus are from Pearl [PM 2018, pp. 236].

Even though with the three rules of *do*-calculus we can solve any identifiable causal effect, we do not have a specific policy on the order we should use those rules, and more importantly, we do not know a priori if a certain causal effect is identifiable or not. If this is the case, we might be using the rules uselessly without being able to reach a successful *do*-free algebraic expression. To overcome this problem, Shpitser and Pearl devised an algorithm [SP 2006a] that checks if a causal effect is identifiable, and if so, returns a *do*-free algebraic expression. This will be the main topic in the following chapter, and its implementation in Python the central goal of this project.

2.3.2 Identifiability Criteria

Before introducing the algorithm we should understand some criteria on identifiability, since they are the basis of said algorithm. The question we want to answer now is simple: how can we check if the causal effect of one variable on another is identifiable in a certain causal model? If this causal effect were not identifiable, we would not even bother to try to use *do*-calculus to compute it.

A useful but partial solution was found by Pearl for Markovian models, those which do not have bidirected edges (i.e., confounders).

Theorem 2.39. (Identifiability of Markovian Models [Pea 2000, Corollary 3.2.6]**)** *Given the causal diagram G of any Markovian model (i.e., that do not contain bidirected edges) in which all variables are measured, all effects are identifiable.*

But we do have confounders, so we need another approach for semi-Markovian causal models. What is essentially different between Markovian and semi-Markovian causal models is the appearance of bidirected edges, so it seems reasonable to study them in detail. To do so, we first need some definitions regarding properties of directed acyclic graphs, in particular, we will look at sets of nodes interconnected by bidirected paths, for they play an important role in identifiability.

Definition 2.40. (C-component) Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a graph. If there exists a set $\boldsymbol{F} \subset \boldsymbol{E}$ which contains only bidirected edges and the graph $(\boldsymbol{V}, \boldsymbol{F})$ is connected, then G is a *C-component* (*confounded component*).

Definition 2.41. (Maximal C-component) Let G be a graph and $S = (\boldsymbol{V}, \boldsymbol{E})$ a C-component with $S \subset G$. Then S is a *maximal C-component* (with respect to G) if, for every bidirected path in G containing at least one node of $\boldsymbol{V}$, that path is also a path in S.

If G is not a C-component, it can be uniquely partitioned into a set of graphs, each a maximal C-component with respect to G.

Lemma 2.42. *Every directed graph $G = (\boldsymbol{V}, \boldsymbol{E})$ can be decomposed into a unique set $C(G) = \{G[S_1], \ldots, G[S_k]\}$ of subgraphs such that every $G[S_i]$ $\forall i \in \{1, \ldots, k\}$ is a maximal C-component of G.*

Proof. Given two nodes $X, Y \in \boldsymbol{V}$, they belong to the same maximal C-component if and only if there exists a bidirected path between X and Y, from the definition of maximal C-component. Therefore every maximal C-component is unique, hence the bidirected paths of G define its maximal C-components. □

This decomposition will ultimately help us to reduce the identification problem into several smaller identification subproblems. A useful special case of C-components are C-trees, which are closely related to direct effects.

Definition 2.43. (C-tree) Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a C-component such that every node has at most one child. If there exists a node $X \in \boldsymbol{V}$ such that $\mathrm{An}(X)_G = \boldsymbol{V}$, then G is a *X-rooted C-tree*.

The following is just a generalization of a C-tree with multiple roots.

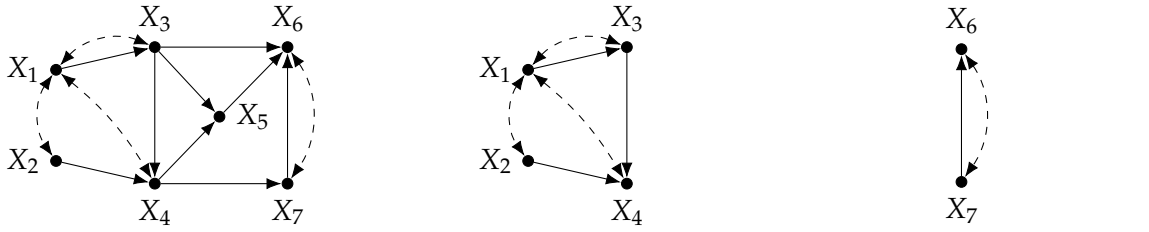
Definition 2.44. (C-forest) Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a C-component such that every node has at most one child. Then, if $\boldsymbol{X} = \mathrm{Rt}(G)$, i.e., if $\boldsymbol{X}$ have no descendants, we say G is a *$\boldsymbol{X}$-rooted C-forest*.

If a DAG contains a pair of different C-forests, under some conditions this pair of C-forests are called a *hedge*, structures that play a fundamental part in identifiability.

Definition 2.45. (Hedge) Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a directed graph, and $\boldsymbol{X}, \boldsymbol{Y} \subset \boldsymbol{V}$ disjoint sets of nodes, i.e., $\boldsymbol{X} \cap \boldsymbol{Y} = \varnothing$. Suppose that there exist two $\boldsymbol{R}$-rooted C-forests $F = (\boldsymbol{V}_F, \boldsymbol{E}_F)$, $\widetilde{F} = (\boldsymbol{V}_{\widetilde{F}}, \boldsymbol{E}_{\widetilde{F}})$ such that $\boldsymbol{X} \cap \boldsymbol{V}_F \neq \varnothing$, $\boldsymbol{X} \cap \boldsymbol{V}_{\widetilde{F}} = \varnothing$, $\widetilde{F} \subseteq F$ and $\boldsymbol{R} \subset \text{An}(\boldsymbol{Y})_{G_{\overline{\boldsymbol{X}}}}$. Then, F and $\widetilde{F}$ form a *hedge* for $P_{\boldsymbol{x}}(\boldsymbol{y})$ in G.

Example 2.46. Figure 2.10 (a) shows a causal diagram G of a probabilistic causal model M. It can easily be seen that G is not a C-component, because its bidirected edges do not connect all vertices in G. Nevertheless, G can be decomposed into three maximal C-components, $G[\boldsymbol{S}_1]$ and $G[\boldsymbol{S}_2]$, as seen in Figures 2.10 (b) and (c), and also the trivial component $G[\boldsymbol{S}_3] = \{X_5\}$, which is the graph containing X_5 with no edges. These three C-components are, in turn, also C-trees, because every node in $G[\boldsymbol{S}_1]$, $G[\boldsymbol{S}_2]$ and $G[\boldsymbol{S}_3]$ has at most one child: $G[\boldsymbol{S}_1]$ is a X_4-rooted C-tree, since $\text{An}(X_4)_{G[\boldsymbol{S}_1]} = \{X_1, X_2, X_3, X_4\}$, $G[\boldsymbol{S}_2]$ is a X_6-rooted C-tree, for $\text{An}(X_6)_{G[\boldsymbol{S}_2]} = \{X_6, X_7\}$, and finally $G[\boldsymbol{S}_3]$ is a X_5-rooted C-tree.

We can also detect a hedge for $P_{x_1}(x_6)$ in G. We can easily see that $G[\boldsymbol{S}_1]$ is a $\{X_4\}$-rooted C-forest with $\{X_1\} \cap \boldsymbol{S}_1 \neq \varnothing$ and $\{X_4\} \subset \text{An}(X_6)_{G_{\overline{X_1}}}$. Now consider F, the $\{X_4\}$-rooted C-forest formed only by the vertex X_4 and no edges. It is clear that $\{X_1\} \cap \boldsymbol{V}_F = \{X_1\} \cap \{X_4\} = \varnothing$, and that $F \subseteq \boldsymbol{S}_1$. Therefore, $G[\boldsymbol{S}_1]$ and F form a hedge for $P_{x_1}(x_6)$ in G.



(a) Graph G. (b) Maximal C-component $G[\boldsymbol{S}_1] \subset G$. (c) Maximal C-component $G[\boldsymbol{S}_2] \subset G$.

Figure 2.10: Causal diagram G contains three maximal C-components, $G[\boldsymbol{S}_1]$, $G[\boldsymbol{S}_2]$ and $G[\boldsymbol{S}_3]$, that are also C-trees.

The following result is finally what we were searching for in this section, a criterion to check if a causal effect is identifiable.

Theorem 2.47. (Hedge Identifiability Criterion [SP 2006a, Theorem 4]**)** *Let G be the causal diagram of a model $M = (\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{F}, P(\boldsymbol{U}))$, and $\boldsymbol{X}, \boldsymbol{Y} \subset \boldsymbol{V}$. Then the causal effect $P_{\boldsymbol{x}}(\boldsymbol{y})$ is not identifiable in G if and only if there exist two $\boldsymbol{R}$-rooted C-forests F and $\widetilde{F}$ that form a hedge for $P_{\boldsymbol{x}}(\boldsymbol{y})$ in G.*

Knowing this we can state that the causal effect $P_{x_1}(x_6)$ is not identifiable in Figure 2.10 (a), since we have seen in Example 2.46 that there is a hedge for $P_{x_1}(x_6)$.

Remark 2.48. In Example 2.46 we have introduced a well-known non-identifiable graph, $G[\boldsymbol{S}_2]$, which is the so-called *bow arc* graph. It is the simplest non-identifiable graph, easily provable with the hedge criterion. Consider the $\{X_6\}$-rooted C-forests $G[\boldsymbol{S}_2]$ and $\{X_6\}$, which clearly form a hedge for $P_{x_7}(x_6)$ in $G[\boldsymbol{S}_2]$.

A complete characterization of identifiability involves hedges, and will be the discussed in the coming chapter.

Chapter 3

Identification Algorithms and Their Implementation

do-calculus was the first step into solving causal effects from causal diagrams, but it has a major problem: even if the effect is identifiable, no one tells us in what order should we apply the three rules, and this gets even more complex when the number of variables in the model grows.

In this section, we will introduce two algorithms that help us compute causal queries, when those are identifiable, called **ID** and **IDC**. We will examine meticulously each and every line of both algorithms, justifying why the operations involved are correct, while also explaining our implementation in Python. This implementation is the central objective of this project, and all objects and functions involved form a newly developed package for Python named `causaleffect`.

We will first introduce an algorithm to solve non-conditional causal effects and its implementation, then we will do the same for an algorithm to compute conditional causal effects. To conclude this chapter we will shortly explain the existence of some algorithms to compute counterfactual queries from interventional distributions, which have not been developed in our package.

3.1 Identification of Interventional Distributions

We have seen that if the causal diagram G of a causal model M is not a C-component it can be decomposed into a unique set of maximal C-components (Lemma 2.42). This in turn will help to reduce the identification problem in more manageable subproblems, making use of the following result by Jin Tian [Tian 2002].

Lemma 3.1. **([**Tian 2002, Corollary 1 pp. 56**])** *Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be the induced DAG from the causal model $M = (\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{F}, P(\boldsymbol{U}))$, and $C(G) = \{G[\boldsymbol{S}_1], \ldots, G[\boldsymbol{S}_k]\}$ a decomposition of G in C-components, where $\boldsymbol{S}_i$ are the vertices in $G[\boldsymbol{S}_i]$. Then, we have*

(a) $P(\boldsymbol{v})$ *factorizes as*

$$P(\boldsymbol{v}) = \prod_{i=1}^{k} P(\boldsymbol{s_i}|do(\boldsymbol{v} \setminus \boldsymbol{s_i})) = \prod_{i=1}^{k} P_{\boldsymbol{v}\setminus\boldsymbol{s_i}}(\boldsymbol{s_i}) \ .$$

(b) Let a topological order over $\boldsymbol{V}$ *be* $V_1 < \cdots < V_n$*, and let* $\boldsymbol{V}_G^{(i)} = \{V_1, \ldots, V_i\}$ *for* $1 \leq i \leq n$*, and* $\boldsymbol{V}_G^{(0)} = \varnothing$*. Then every factor from the previous product is identifiable in* G *as*

$$P_{\boldsymbol{v}\setminus\boldsymbol{s_j}}(\boldsymbol{s_j}) = \prod_{\{i|V_i\in\boldsymbol{S}_j\}} P(v_i|\boldsymbol{v}_G^{(i-1)})$$

We are now able to define the identification algorithm presented in Figure 3.1.

ID: Identification algorithm

Input: Value assignments $\boldsymbol{x}$, $\boldsymbol{y}$, a joint probability distribution $P(\boldsymbol{v})$, a causal diagram $G = (\boldsymbol{V}, \boldsymbol{E})$.

Output: Expression for $P_{\boldsymbol{x}}(\boldsymbol{y})$ in terms of $P(\boldsymbol{v})$, or **FAIL**(F, F').

function ID$(\boldsymbol{y}, \boldsymbol{x}, P, G)$:

1. **if** $\boldsymbol{x} = \varnothing$, **then**
   return $\sum_{v\in\boldsymbol{v}\setminus\boldsymbol{y}} P(\boldsymbol{v})$
2. **if** $\boldsymbol{V} \neq \text{An}(\boldsymbol{Y})_G$, **then**
   return ID$(\boldsymbol{y}, \boldsymbol{x} \cap \text{An}(\boldsymbol{Y})_G, P(\text{An}(\boldsymbol{Y})_G), G[\text{An}(\boldsymbol{Y})_G])$
3. Let $\boldsymbol{W} = (\boldsymbol{V} \setminus \boldsymbol{X}) \setminus \text{An}(\boldsymbol{Y})_{G_{\overline{\boldsymbol{X}}}}$
   if $\boldsymbol{W} \neq \varnothing$, **then**
   return ID$(\boldsymbol{y}, \boldsymbol{x} \cup \boldsymbol{w}, P, G)$
4. **if** $C(G[\boldsymbol{V} \setminus \boldsymbol{X}]) = \{G[\boldsymbol{S}_1], \ldots, G[\boldsymbol{S}_k]\}$, **then**
   return $\sum_{v\in\boldsymbol{v}\setminus(\boldsymbol{y}\cup\boldsymbol{x})} \prod_{i=1}^{k}$**ID**$(\boldsymbol{s}_i, \boldsymbol{v} \setminus \boldsymbol{s}_i, P, G)$
   if $C(G[\boldsymbol{V} \setminus \boldsymbol{X}]) = \{G[\boldsymbol{S}]\}$, **then**
5. **if** $C(G) = \{G\}$, **then**
   throw FAIL$(G, G[\boldsymbol{S}])$
6. **if** $G[\boldsymbol{S}] \in C(G)$, **then**
   return $\sum_{v\in\boldsymbol{s}\setminus\boldsymbol{y}} \prod_{\{i|V_i\in\boldsymbol{S}\}} P\left(v_i|\boldsymbol{v}_G^{(i-1)}\right)$
7. **if** $\exists \boldsymbol{S}'$ with $\boldsymbol{S} \subset \boldsymbol{S}'$ such that $G[\boldsymbol{S}'] \in C(G)$, **then**
   return ID$(\boldsymbol{y}, \boldsymbol{x} \cap \boldsymbol{s}', \prod_{\{i|V_i\in\boldsymbol{S}'\}} P\left(V_i|\boldsymbol{V}_G^{(i-1)} \cap \boldsymbol{S}', \boldsymbol{v}_G^{(i-1)} \setminus \boldsymbol{s}'\right), G[\boldsymbol{S}'])$

Figure 3.1: Algorithm proposed by Shpitser and Pearl [SP 2006a] to compute $P_{\boldsymbol{x}}(\boldsymbol{y})$.

This algorithm systematically takes advantage of the properties of C-components to decompose, recursively, the identification problem into smaller subproblems until either we find an expression for $P_{\boldsymbol{x}}(\boldsymbol{y})$ or a hedge that indicates the causal effect is not identifiable.

In the same paper where Shpitser and Pearl defined this algorithm [SP 2006a], they also proved that it is sound, that is, that when **ID** returns an expression for $P_{\boldsymbol{x}}(\boldsymbol{y})$ it is correct. They also proved that it is complete.

Theorem 3.2. (Soundness and Completeness of ID [SP 2006a, Lemma 3 and Theorem 5]**)** ***ID*** *always terminates, and whenever it returns an expression for $P_{\boldsymbol{x}}(\boldsymbol{y})$, it is correct.*

The algorithm in Figure 3.1 makes use of topological ordering, which is encoded in the graph structure and can be computed beforehand. This is practical, since a topological ordering of a graph G is conserved for any subgraph, and it does not need to be computed again. It is also easy to see that one and only one line of the algorithm will be invoked at every step of the recursion because after checking if a certain condition is fulfilled, it either returns a probability expression, calls the **ID** again with other parameters or throws an error revealing a hedge in the DAG.

Before seeing in more detail every line of the algorithm and how it has been implemented we will show how the algorithm operates given a causal diagram. To do so, consider again the causal diagram introduced in Example 2.38.

Example 3.3. In the example where we showed the Front Door Adjustment we had the causal diagram shown again in Figure 3.2, with renamed variables to ease comprehension. We will also write expressions like $G[X, Y]$ instead of $G[\{X, Y\}]$ to avoid notation overload.

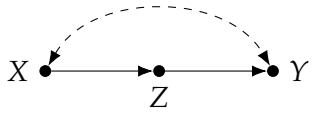


Figure 3.2: Causal diagram first introduced in Example 2.38.

We want to obtain the causal effect $P_x(y)$ from the probability distribution $P(X, Y, Z)$ and the causal diagram G shown in Figure 3.2. First we have to compute the topological ordering of the vertices, which will be encoded in the graph structure, but there is only a single possibility: $X < Z < Y$. We are now ready to apply the algorithm. The line triggered in this first step is line 4, since clearly $\boldsymbol{x} = \{x\} \neq \varnothing$, $\text{An}(\boldsymbol{Y})_G = \text{An}(Y)_G = \boldsymbol{V}$ and $\boldsymbol{W} = \varnothing$, but $C(G[\boldsymbol{V} \setminus \boldsymbol{X}]) = C(G[Y, Z]) = \{G[Y], G[Z]\}$. Hence we have

$$P_x(y) = \sum_z P_{x,z}(y) P_{x,y}(z) \ . \tag{3.1}$$

Now, for the first term the conditions in first three lines are not fulfilled by the same reason as before, but neither line 4, since $C(G[\boldsymbol{V} \setminus \boldsymbol{X}]) = C(G[Y]) = \{G[Y]\}$, so $\boldsymbol{S} = \{Y\}$. Additionally, $C(G) = \{G[X, Y], G[Z]\}$, so we have that $\boldsymbol{S} = \{Y\} \subset \{X, Y\} = \boldsymbol{S'}$ with $G[\boldsymbol{S'}] \in C(G)$. Therefore line 7 is triggered,

$$P_{x,z}(y) = P'_x(y) \ ,$$

where $G' = G[X, Y]$ (see Figure 3.3 (a)) and $P'(X, Y) = P(X)P(Y|X, z)$. Now line 2 is triggered, because $\text{An}(Y)_{G'} = \{Y\} \neq \{X, Y\}$, so

$$P'_x(y) = P''_{\varnothing}(y) \ ,$$

where $G'' = G'[Y] = \{Y\}$ and $P''(Y) = \sum_x P'(Y, x)$. Finally line 1 is triggered, and so we obtain $P''_{\varnothing}(y) = P''(y)$. Going backwards to write this probability in terms of P,

$$P_{x,z}(y) = P''(y) = \sum_x P'(x, y) = \sum_x P(x)P(y|x, z) \ .$$

(a) $G' = G[X, Y]$. (b) $G''' = G[X, Z]$.

Figure 3.3: Induced subgraphs used throughout Example 3.3.

Now we focus on the second term in Equation 3.1, and we see that line 2 is triggered, for $\text{An}(\boldsymbol{Y})_G = \text{An}(Z)_G = \{X, Z\} \neq \boldsymbol{V}$. Hence

$$P_{x,y}(z) = P'''_x(z) \ ,$$

where $G''' = G[X, Z]$ (see Figure 3.3 (b)) and $P'''(X, Z) = \sum_y P(X, Z, y)$. The next step is to see that $\text{An}(\boldsymbol{Y})_{G'''} = \text{An}(Z)_{G'''} = \{X, Z\} = \boldsymbol{V}$, and also $\boldsymbol{W} = \varnothing$, so we compute the confounded components of $G'''[\boldsymbol{V} \setminus \boldsymbol{X}]$ and G''': $C(G'''[\boldsymbol{V} \setminus \boldsymbol{X}]) = \{G'''[Z]\}$ and $C(G''') = \{G'''[X], G'''[Z]\}$, so line 6 is invoked:

$$P'''_x(z) = P'''(z|x) \ .$$

This means that

$$P_{x,y}(z) = P'''(z|x) = \sum_y P(z, y|x) = P(z|x) \ .$$

Putting together the two terms of Equation 3.1 we finally obtain the desired causal effect:

$$P_x(y) = \sum_z \left(\sum_x P(x)P(y|x, z) \right) P(z|x) = \sum_z P(z|x) \sum_x P(x)P(y|x, z) \ .$$

Indeed, we obtain the same result as in Example 2.38, as we see that Equation 2.1 is identical to the one obtained using the identification algorithm with renamed variables $\{X = S, Z = T, Y = C\}$.

ID algorithm in Figure 3.1 can also be used to detect unidentifiability. We are going to see another example of the algorithm that raises an error due to the existence of a hedge.

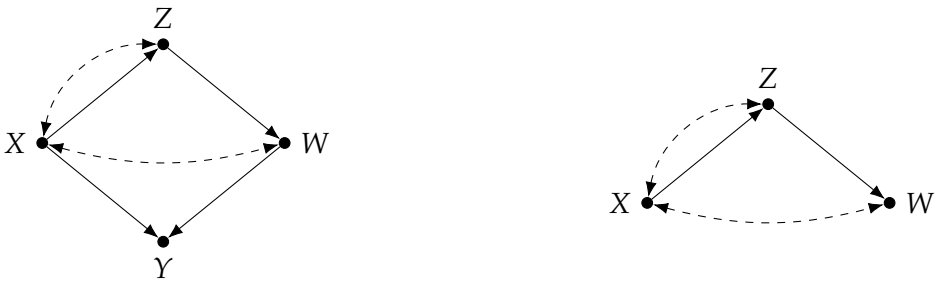
Example 3.4. Consider the DAG G in Figure 3.4 (a), and we will to compute $P_x(y)$. The only possible topological order would be $X < Z < W < Y$, and we see that the ancestors of Y in G are all the nodes in G.

First of all, line 4 will be executed, because the ancestors of Y (in G and in $G_{\overline{X}}$) is the set containing all vertices of G and also $C(G[\boldsymbol{V} \setminus X]) = \{G[Y], G[W], G[Z]\}$. So we will have

$$P_x(y) = \sum_{w,z} P_{x,w,z}(y)P_{x,y,z}(w)P_{x,y,w}(z) \ . \tag{3.2}$$

For the first term, line 6 is triggered, because $G[Y] \in C(G) = \{G[Y], G[X, W, Z]\}$, so

$$P_{x,w,z}(y) = P(y|x, w, z) \ .$$



(a) Graph G. (b) $G' = G[X, W, Z]$.

Figure 3.4: Causal diagrams in Example 3.4.

The second term invokes line 2, because $Y \notin \text{An}(W)_G$, thus we obtain

$$P_{x,y,z}(w) = P_{x,z}(w) \ ,$$

where $G' = G[X, W, Z]$ (see Figure 3.4 (b)) and $P'(X, W, Z) = \sum_y P(X, W, Z, y)$. Now we have $C(G'[W]) = \{G'[W]\}$ and $C(G') = \{G'\}$, so line 5 is triggered, throwing the hedge $(G', G'[W])$ for $P_{x,z}(w)$ in G'. Clearly both are C-components of G', they are W-rooted (with W being an ancestor of Y in $G'_{\overline{X}}$), $G'[W] \subset G'$, $G' \cap \{X\} \neq \varnothing$ and $G'[W] \cap \{X\} = \varnothing$. So we conclude that $(G', G'[W])$ form a hedge for $P_{x,z}(w)$ in G', and thus the original causal effect $P_x(y)$ is not identifiable in G.

We are now ready to explore in detail every line of the algorithm and to show our implementation of **ID** in Python.

3.1.1 Python Implementation of Graphs and Distributions

First of all, to handle graphs and probabilities we need some classes. In our implementation to deal with graphs we have used the `igraph` library for Python since it provides some useful methods (although, as we will see, we will also need to implement our own).

`def createGraph(edges, verbose=False):`		
Description	Creates a Graph object from a list of edges in string format.	
Parameters	`edges`	List of edges of the graph. Each edge must either be directed (`'X->Y'`) or bidirected (`'X<->Y'`).
	`verbose`	Boolean. If enabled, some useful debugging information will be printed.
Returns	A Graph object from the `igraph` library, with the directed and bidirected edges given as the `edges` parameter. Each bidirected edge will be encoded as two directed edges. It will contain exactly all vertices appearing in the edges list. Each edge will have a property called `confounding`, which will be 0 for directed edges and ± 1 for bidirected edges (edges of a bidirected pair will have opposite signs for the `confounding` property).	

Function 3.1: Implemented `createGraph` function.

The first thing we need to be able to do is to create a DAG in a simple, intuitive way. For this purpose, we devised the function `createGraph` (see Function 3.1).

Another useful function that helps visualize causal diagrams that we have implemented is `plotGraph` (see Function 3.2).

`def plotGraph(graph, name=None):`		
Description	Plots a causal diagram. Needs `pycairo` library.	
Parameters	`graph`	Graph object with an edge property named `confounding`.
	`name`	Name of the png file with the plotted graph. If not introduced it does not produce a png image.
Returns	Nothing. It makes use of the function `plot` of the `igraph` library to plot the causal diagram.	

Function 3.2: Implemented `plotGraph` function.

To ease the comparison of causal effects of different causal models between our implementation the one made with R by Tikka and Karvanen [TK 2017], we have created a function that "translates" our edge notation into theirs (see Function 3.3).

`def to_R_notation(edges):`		
Description	Converts a list of edges from our notation to the notation used in the R package `causaleffect`.	
Parameters	`edges`	List of strings containing the edges of a graph.
Returns	Tuple of three elements. The first is a string encoding the edges of a graph. The next two elements are integers, and are the indexes used by the `causaleffect` package to set confounding properties to the edges of the graph.	

Function 3.3: Implemented `to_R_notation` function.

To see how these three functions work we can create the causal diagram in Figure 3.4 (a) by executing the code in Figure 3.5 (a).

The output of the plot of graph G can be seen in Figure 3.5 (b), where bidirected edges are coloured in green and are slightly thinner than directed edges.

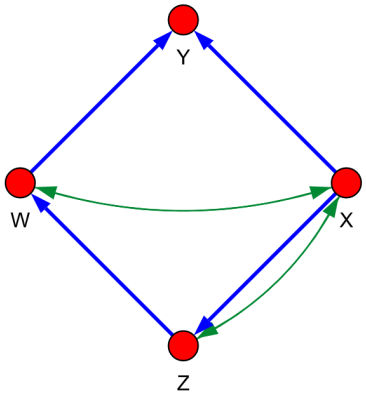
We have developed many more functions to obtain properties of causal diagrams needed in the implementation of **ID**, but they will be explained in due course when required by the algorithm. The next step is to manage distribution probabilities. For that purpose, we have created a Python class named **`Probability`**, which can be constructed recursively to embrace the nature of the algorithm, also recursive.

The **`Probability`** class has several attributes. The string sets **`var`** and **`cond`** allow simple conditional distributions (distributions of **`var`** conditioned on **`cond`**). For instance, the object `Probability(var={'Y'}, cond={'X'})` represents the distribution $P(Y|X)$. To model products of distributions, and mimicking [TK 2017], the boolean attribute **`recursive`** is defined, allowing multiple **`Probability`** objects to be nested inside one another when **`True`**. When this is the case, the attribute **`children`**, which is a set, is filled with **`Probability`** objects, and the variables **`var`** and **`cond`** are ignored. We can now have more complex probability distributions such as $P^*(X,Y,Z,W) = P(Z,X|W)P(Y|Z)P(W)$ by creating the object in Figure 3.6 (a).

We also need to consider how to manage marginal distributions, and thus a set containing variables to be summed over (in the discrete case, integrated in the continuous case) is defined.

```
>>> edges = ['X<->Z', 'X<->W', 'X->Z', 'Z->W', 'W->Y', 'X->Y']
>>> G = createGraph(edges)
>>> print(to_R_notation(edges))
('X-+Z, Z-+W, W-+Y, X-+Y, X-+Z, X+-Z, X-+W, X+-W', 5, 8)
>>> plotGraph(G)
```

(a) Code executed.



(b) Plot of graph G.

Figure 3.5: Executing `createGraph`, `to_R_notation` and `plotGraph`.

The attribute **sumset** is the set of variables that makes this possible. So if we wanted to represent $P^*(X,Y)$, we would need to set the attribute `sumset={'Z', 'W'}` for the object `p` previously defined.

Another level of complexity is needed when computing conditional probabilities. Sometimes to form conditional distribution expressions one has to introduce fractions, and that is why the **Probability** class has two last attributes, **fraction** and **divisor**. When the boolean attribute `fraction` is set to **True**, the **divisor** containing a probability distribution object is enabled. This final step allows us to represent complex conditional distributions such as

$$P^*(X|Y) = \frac{P^*(X,Y)}{P^*(Y)} = \frac{\sum_{W,Z} P(Z,X|W)P(Y|Z)P(W)}{\sum_{X,W,Z} P(Z,X|W)P(Y|Z)P(W)},$$

which can be stored as seen in Figure 3.6 (b).

To understand and read the **Probability** objects easily, a method to get the LaTeX string representation has been added, named `printLatex` (see Function 3.4).

`def printLatex(self, tab=0, simplify=True, complete=True, verbose=False):`		
Description	Constructs the LaTeX string representation of a **Probability** object.	
Parameters	**tab**	Integer, keeps track of the depth of the recursion.
	simplify	Boolean, simplifies the expression if enabled.
	complete	Boolean, performs additional simplifications if enabled.
	verbose	Boolean, prints some useful debugging information if enabled.
Returns	A string containing the LaTeX representation of the probability distribution.	

Function 3.4: Implemented `printLatex` function.

In the latter function defined we have seen that, in some cases, simplifications are made to

```
p1 = Probability(var={'X', 'Z'}, cond={'W'})
p2 = Probability(var={'Y'}, cond={'Z'})
p3 = Probability(var={'W'})
p = Probability(recursive=True, children={p1, p2, p3})
```

(a) $P^*(X, Y, Z, W)$.

```
p1 = Probability(var={'X', 'Z'}, cond={'W'})
p2 = Probability(var={'Y'}, cond={'Z'})
p3 = Probability(var={'W'})
p4 = Probability(sumset={'X', 'Z', 'W'}, recursive=True, children={p1, p2, p3})
p = Probability(sumset={'Z', 'W'}, recursive=True, children={p1, p2, p3}, fraction=
    True, divisor=p4)
```

(b) $P^*(X|Y)$.

Figure 3.6: Representing probability distributions with the `Probability` class.

the distribution. Those are handled by the `simplify` method in Function 3.5.

`def simplify(self, complete=True, decouple=True, verbose=False):`		
Description	Simplifies the probability distribution object.	
Parameters	`complete`	Boolean, performs additional simplifications if enabled.
	`verbose`	Boolean, prints some useful debugging information if enabled.
Returns	Nothing.	

Function 3.5: Implemented `simplify` function.

There are some steps to perform basic simplifications. The first step is to decouple the `Probability` objects when those have children that, in turn, have children. If possible, we will delete these children of children by moving them up one level in the recursion definition, to have just a single level of children. Then the basic simplifications are made for those distributions that are not recursive, which are $\texttt{newsumset} = \texttt{sumset} \setminus (\texttt{sumset} \cap \texttt{var})$ and $\texttt{newvar} = \texttt{var} \setminus (\texttt{sumset} \cap \texttt{var})$. This is an application of the Law of Total Probability,

$$\sum_{X,Y} P(Y, W|X) = \sum_{X} P(W|X) \ .$$

If the probability to simplify is a fraction it will do the same in the denominator, and then it will check if the variable set is empty. Additionally, if the conditioning set of the denominator is empty and the variable set of the denominator is a subset of the variables of the numerator, it will delete the fraction and change the numerator accordingly to the conditional probability. An example of this simplification is shown below:

$$\frac{\sum_X P(Y, W|X)}{P(Y)} = \sum_{X} P(W|X, Y) \ .$$

If the `complete` boolean is enabled and the probability distribution is recursive it will try to perform additional simplifications. For every different pair of non-recursive children (`p1`, `p2`) it will check if `p1.cond == (p2.var)`$\cup$`(p2.cond)`, and if this is the case, it will delete `p2` from

the children and change `p1.var = (p1.var)∪(p2.var)` and `p1.cond = (p1.cond)\(p2.var)`. This simplification is the same made in Example 2.4, and an example can be seen below:

$$P(Y, W|X, Z)P(X|Z) = P(X, Y, W|Z) \ .$$

With all these preparations discussed we find ourselves ready to explore the **ID** algorithm line by line and see how it has been implemented.

3.1.2 Python Implementation of the ID Algorithm

We have already reviewed how graphs and probability distribution objects will be encoded in our implementation, and later on, when used, some useful functions regarding these objects will be explained. The function implementing the **ID** algorithm in Figure 3.1 in our package is called `ID_rec`, referring to its recursive nature (see Function 3.6).

`def ID_rec(Y, X, P, G, ordering, verbose=False, tab=0):`		
Description	Recursive function that implements the identification algorithm **ID**, computing the causal effect $P_{\boldsymbol{x}}(\boldsymbol{y})$ of a DAG G.	
Parameters	`Y`	Set of strings containing the variables in $\boldsymbol{y}$.
	`X`	Set of strings containing the intervened variables in $\boldsymbol{x}$.
	`P`	`Probability` object with the probability distribution P.
	`G`	Graph object, encoding the DAG of the causal model G.
	`ordering`	List of strings containing a topological ordering of the nodes of G.
	`verbose`	Boolean, prints some useful debugging information if enabled.
	`tab`	Integer, keeps track of the depth of the recursion.
Returns	If the effect is identifiable it returns a `Probability` object with de computed causal effect $P_{\boldsymbol{x}}(\boldsymbol{y})$. If the algorithm encounters a hedge it raises an error, providing the two forests that form the hedge for $P_{\boldsymbol{x}}(\boldsymbol{y})$ in G.	

Function 3.6: Implemented `ID_rec` function.

When the function is called it first retrieves all vertices of G and stores them as a set in a variable `V`. It then separates the graph G in the directed (only containing directed edges) and the bidirected (only containing bidirected edged) parts, calling the developed function `get_directed_bidirected_graphs` in Function 3.7.

`def get_directed_bidirected_graphs(G):`		
Description	Decouples the causal diagram G into two separate graphs: one containing only bidirected edges, and the other containing directed edges.	
Parameters	`G`	Graph object G.
Returns	A tuple with two Graph objects (one with bidirected edges and the other with directed edges).	

Function 3.7: Implemented `get_directed_bidirected_graphs` function.

The observed portion of G will be saved as `G_dir`. We now proceed to explain every line of the algorithm in detail, revealing our implementation.

Line 1

1. **if** $\boldsymbol{x} = \varnothing$, **then**
   return $\sum_{v \in \boldsymbol{v} \setminus \boldsymbol{y}} P(\boldsymbol{v})$

This line is rather straightforward, since we do not have the *do*-operator any more ($\boldsymbol{x} = \varnothing$). Therefore, we are computing the marginal distribution $P(\boldsymbol{y})$ instead of a causal effect. This action of marginalizing considers two cases: if the probability distribution is simple, this is, if it is not a product of probabilities, changes the variables of P to the variables in $\boldsymbol{y}$. If, on the contrary, the probability is a product of probabilities, adds the pertinent variables to the sum set, `sumset` $=$ `sumset` $\cup$ (`V` $\setminus$ `Y`).

Line 2

2. **if** $\boldsymbol{V} \neq \text{An}(\boldsymbol{Y})_G$, **then**
   return ID$(\boldsymbol{y}, \boldsymbol{x} \cap \text{An}(\boldsymbol{Y})_G, P(\text{An}(\boldsymbol{Y})_G), G[\text{An}(\boldsymbol{Y})_G])$

This line eliminates all non-ancestors of Y in G. To see clearly how this is possible, consider the following result.

Lemma 3.5. **([**SP 2006a, Lemma 5**])** *Let* $\boldsymbol{X}' = \boldsymbol{X} \cap \text{An}(\boldsymbol{Y})_G$. *Then,* $P_{\boldsymbol{x}}(\boldsymbol{y})$ *obtained from* P *in* G *is equal to* $P'_{\boldsymbol{x}'}(\boldsymbol{y})$ *obtained from* $P' = P(\text{An}(\boldsymbol{Y})_G)$ *in* $G[\text{An}(\boldsymbol{Y})_G]$.

Proof. Let $\boldsymbol{W} = \boldsymbol{V} \setminus \text{An}(\boldsymbol{Y})_G$, and consider the submodel $M_{\boldsymbol{w}}$, i.e., where the variables in $\boldsymbol{W}$ are intervened. The induced causal diagram is $G[\boldsymbol{V} \setminus \boldsymbol{W}] = G[\text{An}(\boldsymbol{Y})_G]$, and the induced distribution is $P' = P_{\boldsymbol{w}}(\text{An}(\boldsymbol{Y})_G)$.

Now let $\boldsymbol{X}'' = \boldsymbol{X} \setminus \text{An}(\boldsymbol{Y})_G$, and clearly $\boldsymbol{X} = \boldsymbol{X}' \sqcup \boldsymbol{X}''$. Therefore, we have

$$P_{\boldsymbol{x}}(\boldsymbol{y}) = P_{\boldsymbol{x}', \boldsymbol{x}''}(\boldsymbol{y}) = P_{\boldsymbol{x}'}(\boldsymbol{y}) \ ,$$

where in the last equality we have used rule 3 of *do*-calculus. Indeed, note that we have $(\boldsymbol{Y} \perp\!\!\!\perp \boldsymbol{X}'' | \boldsymbol{X}')_{G_{\overline{\boldsymbol{X}'}, \overline{\boldsymbol{X}''}}} = (\boldsymbol{Y} \perp\!\!\!\perp \boldsymbol{X}'' | \boldsymbol{X}')_{G_{\overline{\boldsymbol{X}}}}$, because there are no paths from $\boldsymbol{Y}$ to $\boldsymbol{X}''$: directed paths from $\boldsymbol{X}''$ to $\boldsymbol{Y}$ are non-existing because, by definition, $\boldsymbol{X}''$ has no ancestors of $\boldsymbol{Y}$, and back-door paths are deleted by removing all incoming edges to $\boldsymbol{X}'' \subset \boldsymbol{X}$ in $G_{\overline{\boldsymbol{X}}}$. We apply rule 3 of *do*-calculus again, finally obtaining

$$P_{\boldsymbol{x}'}(\boldsymbol{y}) = P_{\boldsymbol{x}', \boldsymbol{w}}(\boldsymbol{y}) = P'_{\boldsymbol{x}'}(\boldsymbol{y}) \ ,$$

because $(\boldsymbol{Y} \perp\!\!\!\perp \boldsymbol{W} | \boldsymbol{X}')_{G_{\overline{\boldsymbol{X}'}, \overline{\boldsymbol{W}}}}$. By the same reasoning as before, there are no directed paths from $\boldsymbol{W} = \boldsymbol{V} \setminus \text{An}(\boldsymbol{Y})_G$ to $\boldsymbol{Y}$, neither back-door paths (deleted by looking at $G_{\overline{\boldsymbol{X}'}, \overline{\boldsymbol{W}}}$). □

To implement this line, we first have developed a function to compute the ancestors of a node or a set of nodes (see Function 3.8).

If a single vertex is inputted, Function 3.8 computes its ancestors by exploring the directed acyclic graph using BFS (breadth-first search). If, on the other hand, a set of vertices is given, it calls itself for every single vertex in the set and performs the union of the results.

To check if the condition in line 2 is fulfilled we compute the ancestors of `Y` in G and save it as `anc`, before querying the length of the set `V` $\setminus$ `anc`. If this length is not zero, we create a `Probability` object which is the marginalized distribution of the one given (considering the

`def get_ancestors(G, V):`		
Description	Computes the set containing all ancestors of a vertex or a set of vertices, including itself.	
Parameters	`G`	Graph object, encoding the direct portion of the causal diagram G (which is a DAG).
	`V`	String with the name of a vertex, or a set of strings containing the names of the vertices.
Returns	A set of strings containing the names of vertices in G which are ancestors of `V`.	

Function 3.8: Implemented `get_ancestors` function.

two cases discussed in line 1), and we call `ID_rec` again with the new parameters specified. The only parameter worth mentioning is the induced subgraph, $G[\text{An}(\boldsymbol{Y})_G]$, which we have computed using the function `induced_subgraph` from the `igraph` library for efficiency purposes. This function takes a graph and a set of nodes and constructs a subgraph with all given nodes and edges between them as in the original graph.

Line 3

3. Let $\boldsymbol{W} = (\boldsymbol{V} \setminus \boldsymbol{X}) \setminus \text{An}(\boldsymbol{Y})_{G_{\overline{\boldsymbol{X}}}}$
   if $\boldsymbol{W} \neq \varnothing$, **then**
   return ID$(\boldsymbol{y}, \boldsymbol{x} \cup \boldsymbol{w}, P, G)$

This line adds interventions to the original causal effect, which is possible due to rule 3 of *do*-calculus.

Lemma 3.6. **([**SP 2006a, Lemma 6**])** *Let* $\boldsymbol{W} = (\boldsymbol{V} \setminus \boldsymbol{X}) \setminus \text{An}(\boldsymbol{Y})_{G_{\overline{\boldsymbol{X}}}}$. *Then,* $P_{\boldsymbol{x}}(\boldsymbol{y}) = P_{\boldsymbol{x},\boldsymbol{w}}(\boldsymbol{y})$, *where* $\boldsymbol{x}$ *are arbitrary values of* $\boldsymbol{W}$ *within its domain.*

Proof. By assumption, we have that $(\boldsymbol{Y} \perp\!\!\!\perp \boldsymbol{W} | \boldsymbol{X})_{G_{\overline{\boldsymbol{X}},\overline{\boldsymbol{W}}}}$, because there are no paths from $\boldsymbol{Y}$ to $\boldsymbol{W}$: directed paths from $\boldsymbol{W}$ to $\boldsymbol{Y}$ are non-existing because, by definition, $\boldsymbol{W}$ has no ancestors of $\boldsymbol{Y}$, and back-door paths are deleted by removing all incoming edges to $\boldsymbol{W}$ in $G_{\overline{\boldsymbol{X}},\overline{\boldsymbol{W}}}$. Hence the result holds by applying rule 3 of *do*-calculus. □

In our code, to construct $\boldsymbol{W}$ we first have created a copy of the graph and deleted all incoming edges into the set $\boldsymbol{X}$, thus constructing $G_{\overline{\boldsymbol{X}}}$. This has been achieved using the `igraph` function `delete_edges` and inputting the edges to remove using the method `select`, also provided by the package. We then have computed the ancestors of $\boldsymbol{Y}$ in $G_{\overline{\boldsymbol{X}}}$ using the previously discussed function, and we have successfully constructed $\boldsymbol{W}$ with set differences. By checking the length of the set $\boldsymbol{W}$ we have determined whether to execute line 3, and if the condition is fulfilled, the `ID_rec` function is called again with the appropriate parameters.

Note that Lemma 3.6 does not fix the values of added interventions $do(\boldsymbol{W} = \boldsymbol{w})$. This means that the resulting expression does not depend on the value assigned to $\boldsymbol{w}$, even though it appears in the expression $P_{\boldsymbol{x},\boldsymbol{w}}(\boldsymbol{y})$.

Line 4

4. **if** $C(G[\boldsymbol{V} \setminus \boldsymbol{X}]) = \{G[\boldsymbol{S}_1], \ldots, G[\boldsymbol{S}_k]\}$, **then**
   return $\sum_{v \in \boldsymbol{v} \setminus (\boldsymbol{y} \cup \boldsymbol{x})} \prod_{i=1}^{k} \mathbf{ID}(\boldsymbol{s}_i, \boldsymbol{v} \setminus \boldsymbol{s}_i, P, G)$

This follows directly from Lemma 3.1 (a), considering the submodel model $M_{\boldsymbol{x}}$. This model induces the causal diagram $G[\boldsymbol{V} \setminus \boldsymbol{X}]$ and probability distribution $P_{\boldsymbol{x}}(\boldsymbol{V} \setminus \boldsymbol{X})$. Then, if we marginalize the distribution we get

$$P_{\boldsymbol{x}}(\boldsymbol{y}) = \sum_{\boldsymbol{v} \setminus (\boldsymbol{y} \cup \boldsymbol{x})} P_{\boldsymbol{x}}(\boldsymbol{v}) = \sum_{\boldsymbol{v} \setminus (\boldsymbol{y} \cup \boldsymbol{x})} P_{\boldsymbol{x}}(\boldsymbol{v} \setminus \boldsymbol{x}) = \sum_{\boldsymbol{v} \setminus (\boldsymbol{y} \cup \boldsymbol{x})} \prod_{i=1}^{k} P_{\boldsymbol{v} \setminus \boldsymbol{s}_i, \boldsymbol{x}}(\boldsymbol{s}_i) = \sum_{\boldsymbol{v} \setminus (\boldsymbol{y} \cup \boldsymbol{x})} \prod_{i=1}^{k} P_{\boldsymbol{v} \setminus \boldsymbol{s}_i}(\boldsymbol{s}_i) \,,$$

where $do(\boldsymbol{X} = \boldsymbol{x})$ is in each term of the product, because $\boldsymbol{V} \setminus \boldsymbol{X} = \bigcup_i \boldsymbol{S}_i$ and thus $\boldsymbol{X} \subset \boldsymbol{V} \setminus \boldsymbol{S}_i$ for all $i \in \{1, \ldots, k\}$. A full alternative proof can be found in [SP 2006a, Lemma 4].

We have created a method (refer to Function 3.9) that computes the C-components of a causal diagram.

`def get_C_components(G):`	
Description	Computes the set of maximal C-components of a given graph.
Parameters	`G` Graph object, the causal diagram G.
Returns	A list of subgraphs containing all maximal C-components of G.

Function 3.9: Implemented `get_C_components` function.

Function 3.9 has been created by decomposing G into its directed and bidirected part. Then, for every directed edge in G, we have checked if both source and target vertices of said edge were in the same connected component in the bidirected part of G and if so, we have added this edge into the bidirected part. By performing this algorithm for all directed vertices we will obtain a modified graph of the old bidirected part, which may be no longer only bidirected. This new graph might be disconnected, and if so every connected subcomponent will be a maximal C-component of the original graph G. Finally, we have created a list of connected subcomponents of this potentially disconnected graph, which was the initial goal. In the making of this function, we have used the methods `subcomponent` and `decompose` of `igraph`, for retrieving a connected subcomponent of a graph and for making a list of connected subcomponents of a graph, respectively.

The length of the set containing the maximal C-components of $G[\boldsymbol{V} \setminus \boldsymbol{X}]$ will determine if line 4 is triggered or not. If invoked, a set of probabilities will be filled by calling the `ID_rec` function again, with the proper parameters, one time for each C-component. If identifiable, each `ID_rec` call will return a **`Probability`** object, stored in the set of probabilities aforementioned. Finally, a recursive **`Probability`** object will be returned, with the set of probabilities as children and the pertinent **`sumset`**.

If line 4 has not been triggered, then there is only one C-component, $C(G[\boldsymbol{V} \setminus \boldsymbol{X}]) = \{G[\boldsymbol{S}]\}$, so lines 5, 6 or 7 will be executed.

Line 5

5. **if** $C(G) = \{G\}$, **then**
   throw FAIL$(G, G[\boldsymbol{S}])$

The discovery of a hedge in G for $P_{\boldsymbol{x}}(\boldsymbol{y})$ is the only source of unidentifiability in the algorithm. The conditions of line 5 imply the existence of a hedge for the current recursion stage, as it states the following Theorem.

Theorem 3.7. ([SP 2006a, Theorem 6]) *Suppose that line 5 in* **ID** *is executed. Then there exist* $\boldsymbol{X}' \subseteq \boldsymbol{X}$ *and* $\boldsymbol{Y}' \subseteq \boldsymbol{Y}$ *such that the graph pair* G, $G[\boldsymbol{S}]$ *returned by the fail condition of* **ID** *contain as edge subgraphs two C-forests* F, F' *that form a hedge for* $P_{\boldsymbol{x}'}(\boldsymbol{y}')$.

Proof. Let $\boldsymbol{R}$ the root set of G (which in this step of the recursion will be a subgraph of the original input). Since G is a single C-component, it is possible to remove a set of directed arrows from G while preserving the root set $\boldsymbol{R}$ such that the resulting graph F is an $\boldsymbol{R}$-rooted C-forest (i.e., such that each node has at most one child). Additionally, by lines 2 and 3 of the algorithm, we know that $\boldsymbol{R} \subset \text{An}(\boldsymbol{Y})_{G_{\overline{\boldsymbol{X}}}}$.

Now consider the graph $F' = F \cap G[\boldsymbol{S}]$, which is also an $\boldsymbol{R}$-rooted C-forest because only single directed arrows were removed from $G[\boldsymbol{S}]$ to obtain F'. Moreover we know that $F' \subset F$, $\boldsymbol{V}_F \cap \boldsymbol{X} \neq \varnothing$ and $\boldsymbol{V}_{F'} \cap \boldsymbol{X} = \varnothing$ by construction, so we have a hedge for $\boldsymbol{X}$ and $\boldsymbol{Y}$, a subset of the original input. □

In our source code we have called once again the function `get_C_components` for the original graph. If it only produces a single maximal C-component it must be the graph itself, so by checking the length of the set of maximal C-components of G we decide whether to invoke line 5. If so, it raises a `HedgeFound` exception, which returns information about the pair $(G, G[\boldsymbol{S}])$.

Line 6

6. **if** $G[\boldsymbol{S}] \in C(G)$, **then**
   return $\sum_{v \in \boldsymbol{s} \setminus \boldsymbol{y}} \prod_{\{i | V_i \in \boldsymbol{S}\}} P\left(v_i | \boldsymbol{v}_G^{(i-1)}\right)$

This line also follows from Lemma 3.1, but this time from part (b). If line preconditions are met, this is, if $C(G[\boldsymbol{V} \setminus \boldsymbol{X}]) = \{G[\boldsymbol{S}]\}$, then $\boldsymbol{V} \setminus \boldsymbol{X} = \boldsymbol{S}$ follows. This implies that G local to that recursive call is partitioned into $G[\boldsymbol{S}] \in C(G)$ and $G[\boldsymbol{X}]$, with no bidirected arcs between these two partitions, and that $\boldsymbol{X} = \boldsymbol{V} \setminus \boldsymbol{S}$. Therefore,

$$P_{\boldsymbol{x}}(\boldsymbol{y}) = P_{\boldsymbol{v} \setminus \boldsymbol{s}}(\boldsymbol{y}) = \sum_{\boldsymbol{s} \setminus \boldsymbol{y}} P_{\boldsymbol{v} \setminus \boldsymbol{s}}(\boldsymbol{s}) = \sum_{\boldsymbol{s} \setminus \boldsymbol{y}} \prod_{\{i | V_i \in \boldsymbol{S}\}} P(v_i | \boldsymbol{v}_G^{(i-1)}) \,,$$

where the last equality follows from Lemma 3.1 (b) given that $G[\boldsymbol{S}] \in C(G)$.

Note that in line 4 we could not use Lemma 3.1 (b) for each product term $P_{\boldsymbol{v} \setminus \boldsymbol{s_i}}(\boldsymbol{s_i})$ as we have done here because, unlike this case, we did not know if $G[\boldsymbol{S}_i] \in C(G)$.

To implement this line of the algorithm we must first check if $G[\boldsymbol{S}] \in C(G)$, and we have constructed a function called `check_subcomponent` that does just that: it iterates through a list of graphs, called `components`, and returns `True` if a given graph is in that list. The complicated part is to create a function to compare graphs, this is, to analyse if two graphs are equal. This function, named `graphs_are_equal` in our package, has been implemented relying on another function, `check_subgraph` (see Function 3.10), that checks if a given graph G^1 is a subgraph of another graph G^2, $G^1 \subseteq G^2$. Provided with this function, to see if two graphs are equal

`def check_subgraph(G1, G2):`		
Description	Given two graphs G^1 and G^2 this function verifies if $G^1 \subseteq G^2$.	
Parameters	`G1`	Graph object, with both directed and bidirected parts.
	`G2`	Graph object, with both directed and bidirected parts.
Returns	A boolean whose value is `True` if `G1`$\subseteq$`G1`, `False` otherwise.	

Function 3.10: Implemented `check_subgraph` function.

we just have to see if $G^1 \subseteq G^2$ and $G^2 \subseteq G^1$, and that is precisely what the `graphs_are_equal` function does.

Function 3.10 first checks if the set of nodes of G^1 is a subset of the set of nodes of G^2, and then looks if the number of edges of G^1 is larger than the number of edges in G^2. If the former condition is not satisfied or the latter is fulfilled, the function would immediately return `False`. Then it checks if every edge in G^1 is also in G^2, and if all edges of G^1 satisfy the prior condition, returns `True`. If even a single edge in G^1 is not in G^2, it would return `False`.

If the condition of line 6 is fulfilled, we have to consider two different cases. The first one is when $\boldsymbol{S}$ consists of a single vertex ($|\boldsymbol{S}| = 1$) and hence we do not have a product of probabilities (the `Probability` object to return will not be recursive but instead very simple). We will obtain the conditional vertices in $\boldsymbol{v}_G^{(i-1)}$ by calling the `get_previous_order` method defined in Function 3.11.

`def get_previous_order(v, possible, ordering):`		
Description	This function computes the vertices prior to a given one in a certain topological ordering. It excludes the given vertex and all other vertices not present in the possible set of vertices passed.	
Parameters	`v`	String, name of the initial vertex of the topological ordering.
	`possible`	Set of possible vertices that can be in the output set.
	`ordering`	List of vertices containing an ordering of the graph G.
Returns	A set of vertices strictly smaller than the given vertex in the given topological ordering, intersected with the possible vertices set.	

Function 3.11: Implemented `get_previous_order` function.

Once we have the conditional variables of the output `Probability` object we must construct said object. If the probability distribution in the recursive call is simple and only consists of a single term, the output probability distribution will be also very simple. For example, if $P'(X,Y,Z) = P(X,Y,Z)$ and we want to calculate $P'(Y|X,Z)$, it is just $P(Y|X,Z)$. But if the given distribution is a little bit more complex and consists of a product or sums of probabilities the output distribution will also be complex. For instance, consider the previous example but with $P'(X,Y,Z) = P(X,Y)P(Z)$. Now we would have

$$P'(Y|X,Z) = \frac{P'(X,Y,Z)}{P'(X,Z)} = \frac{P'(X,Y,Z)}{\sum_Y P'(X,Y,Z)} = \frac{P(X,Y)P(Z)}{\sum_Y P(X,Y)P(Z)} = \frac{P(X,Y)P(Z)}{P(X)P(Z)} = P(Y|X)\ .$$

To take this into account we have devised a function named `get_new_probability` (explained in Function 3.12).

The simplest case is when `cond` $= \varnothing$. In this scenario, if the given distribution is not

`def get_new_probability(P, var, cond={}):`		
Description	Function that computes a conditional probability from a distribution `P` = P' of the form $P'(\texttt{var}|\texttt{cond})$.	
Parameters	`P`	`Probability` object, with a probability distribution P'.
	`var`	Set of variables of the new distribution.
	`cond`	Set of conditional variables of the new distribution.
Returns	A `Probability` object from the given distribution P' encoding $P'(\texttt{var}|\texttt{cond})$.	

Function 3.12: Implemented `get_new_probability` function.

recursive we only change the variables of the old distribution for `var`, while if it is a product of probabilities we add the variables $\texttt{V} \setminus \texttt{var}$ in the sum set (marginalizing). By contrast, if $\texttt{cond} \neq \varnothing$, we construct a new `Probability` object which is a fraction of the form $\frac{\sum_{\texttt{V}\setminus(\texttt{var}\cup\texttt{cond})} P'}{\sum_{\texttt{V}\setminus\texttt{cond}} P'}$, equal to $P'(\texttt{var}|\texttt{cond}) = \frac{P'(\texttt{var}\cup\texttt{cond})}{P'(\texttt{cond})}$. Before returning this probability we attempt to perform a simplification using `simplify` (Function 3.5).

Once we have this new probability, this first case of line 6 can return that `Probability` object summed over $\boldsymbol{S} \setminus \boldsymbol{Y}$.

We have considered when $\boldsymbol{S}$ has only one element, and the remaining case when $\boldsymbol{S}$ has two or more elements is now simple. We construct a set of probabilities, and for each vertex in $\boldsymbol{S}$ we create a `Probability` object in the same manner as in the first scenario. Then we construct and return a recursive `Probability` object with this set of probabilities as children, and summed over $\boldsymbol{S} \setminus \boldsymbol{Y}$.

Line 7

7. **if** $\exists \boldsymbol{S}'$ with $\boldsymbol{S} \subset \boldsymbol{S}'$ such that $G[\boldsymbol{S}'] \in C(G)$, **then**
   return ID$(\boldsymbol{y}, \boldsymbol{x} \cap \boldsymbol{s}', \prod_{\{i|V_i \in \boldsymbol{S}'\}} P\left(V_i | \boldsymbol{V}_G^{(i-1)} \cap \boldsymbol{S}', \boldsymbol{v}_G^{(i-1)} \setminus \boldsymbol{s}'\right), G[\boldsymbol{S}'])$

This line covers the last possible outcome. If the algorithm has reached this far it is clear that $C(G)$ has more than one C-component (otherwise line 5 would have been triggered), and that $G[\boldsymbol{S}]$ is not a maximal C-component of G (or then line 6 would have been executed). Therefore there are bidirectional edges between $G[\boldsymbol{S}]$ and $G[\boldsymbol{X}]$ in G, and so there has to be a maximal C-component of G, $G[\boldsymbol{S}']$, such that $G[\boldsymbol{S}] \subset G[\boldsymbol{S}']$. The equivalence of intervened probabilities in line 7 relays on the following result.

Lemma 3.8. **([**SP 2006a, Lemma 8**])** *If the conditions of line 7 of* **ID** *are satisfied, $P_{\boldsymbol{x}}$ obtained from P in G is equal to $P'_{\boldsymbol{x}\cap\boldsymbol{s}'}$ obtained from $P' = \prod_{\{i|V_i \in \boldsymbol{S}'\}} P\left(V_i | \boldsymbol{V}_G^{(i-1)} \cap \boldsymbol{S}', \boldsymbol{v}_G^{(i-1)} \setminus \boldsymbol{s}'\right)$ in $G[\boldsymbol{S}']$.*

The implementation we have devised is somewhat similar to the one performed in line 6. First it checks, for each component $G[\boldsymbol{S}'_j]$ in $C(G)$, if $G[\boldsymbol{S}] \subset G[\boldsymbol{S}'_j]$, by iterating over the components in $C(G)$ and using the `check_subgraph` function earlier defined (see Function 3.10). Once it has found the maximal C-component of G that satisfies $G[\boldsymbol{S}] \subset G[\boldsymbol{S}'] \in C(G)$, it splits again the possible cases in $|\boldsymbol{S}'| = 1$ and $|\boldsymbol{S}'| > 1$.

When $|\boldsymbol{S}'| = 1$, we get the smaller vertices $\boldsymbol{V}_G^{(i-1)}$ using the `get_previous_order` function (Function 3.11), and perform the intersections, differences and unions necessary according to the conditional variables of the new probability stated by line 7. Next, using these con-

ditional variables, we compute a new probability distribution as we did in line 6, by using function `get_new_probability` (Function 3.12). We finally return a recursive call to `ID_rec` with $\boldsymbol{Y}$, $\boldsymbol{X} \cap \boldsymbol{S}'$, the new probability distribution and the induced subgraph $G[\boldsymbol{S}']$ (using `induced_subgraph` from the `igraph` library).

When $|\boldsymbol{S}'| > 1$ we define a set of probabilities and for each vertex in $\boldsymbol{S}'$ we create a `Probability` object in the same fashion as in the case $|\boldsymbol{S}'| = 1$. We finally construct a recursive `Probability` object with this set of probabilities as children, which will be the distribution passed in the recursive call to `ID_rec`. All the other parameters of this recursive call are the same as in the first scenario.

We have checked all lines of the **ID** algorithm, and since it is sound and complete, there are no more cases to explore. Our implementation of this algorithm, named `ID_rec` in the developed package, consequently finishes here. Despite being very useful in the majority of cases, this algorithm only computes identifiable causal effects of the form $P_{\boldsymbol{x}}(\boldsymbol{y})$, but we could also think of conditional interventions. Shpitser and Pearl, authors of the recently explained identification algorithm, devised also an algorithm for such cases, as we explain in the following section.

3.2 Identification of Conditional Interventional Distributions

We have seen an algorithm to compute the causal effect of the form $P_{\boldsymbol{x}}(\boldsymbol{y})$, but often we know that some variables hold, that is, the distribution is conditioned on a set of other variables. We now consider the problem of identifying distributions of the form $P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{z})$, where $\boldsymbol{X}, \boldsymbol{Y}$ and $\boldsymbol{Z}$ are disjoint sets of variables. The basic idea is to reduce the problem to a solved case when $\boldsymbol{Z} = \varnothing$, so we can use the algorithm **ID** seen in the former section. A possible way to do so is to use rule 2 of *do*-calculus, that exchanges actions and observations, $P_{\boldsymbol{x},\boldsymbol{z}}(\boldsymbol{y}|\boldsymbol{w}) = P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{z},\boldsymbol{w})$, under some conditions, and that is how the algorithm **IDC** is constructed. This conditional identification algorithm is presented in Figure 3.7.

IDC: Conditional identification algorithm

Input: Value assignments $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$, a joint probability distribution $P(\boldsymbol{v})$, a causal diagram $G = (\boldsymbol{V}, \boldsymbol{E})$.

Output: Expression for $P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{z})$ in terms of $P(\boldsymbol{v})$, or **FAIL**(F, F').

function IDC$(\boldsymbol{y}, \boldsymbol{x}, \boldsymbol{z}, P, G)$:

1. **if** $\exists Z' \in \boldsymbol{Z}$ such that $(\boldsymbol{Y} \perp\!\!\!\perp Z' | \boldsymbol{X}, \boldsymbol{Z} \setminus \{Z'\})_{G_{\overline{\boldsymbol{X}}, \overline{\boldsymbol{Z}'}}}$, **then**
   return IDC$(\boldsymbol{y}, \boldsymbol{x} \cup \{z'\}, \boldsymbol{z} \setminus \{z'\}, P, G)$
2. **else** let $P' = \mathbf{ID}(\boldsymbol{y} \cup \boldsymbol{z}, \boldsymbol{x}, P, G)$,
   return $\frac{P'}{\sum_{\boldsymbol{y}} P'}$

Figure 3.7: Algorithm proposed by Shpitser and Pearl [SP 2006b] to compute $P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{z})$.

Similarly to the first identification algorithm, Shpitser and Pearl defined this algorithm in [SP 2006b] and they also proved that it is sound and complete.

Theorem 3.9. (Soundness and Completeness of IDC [SP 2006b, Theorem 7 and Theorem 8]**)** *IDC always terminates, and whenever it returns an expression for $P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{z})$, it is correct.*

The first line of the **IDC** is a direct application of rule 2 of *do*-calculus where the exchange consists of a single variable $\{Z'\}$, because if $(\boldsymbol{Y} \perp\!\!\!\perp Z'|\boldsymbol{X}, \boldsymbol{Z} \setminus \{Z'\})_{G_{\overline{\boldsymbol{X}}, \underline{Z'}}}$, then $P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{z}) = P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{z} \setminus \{z'\}, \{z'\}) = P_{\boldsymbol{x} \cup \{z'\}}(\boldsymbol{y}|\boldsymbol{z} \setminus \{z'\})$ follows.

The full construction and justification of the conditional identification algorithm can be found in [SP 2006b].

3.2.1 Python Implementation of the IDC Algorithm

Having seen how we represent graphs and probability distribution objects, and how the **ID** algorithm works (which is an essential requirement), we are ready to reveal our implementation of **IDC** in Figure 3.7. In our package, algorithm **IDC** is called `IDC`, and it is defined in Function 3.13.

`def IDC(Y, X, Z, P, G, ordering, verbose=False, tab=0):`		
Description	Recursive function that implements the conditional identification algorithm **IDC**, computing the causal effect $P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{z})$ of a DAG G, by using the rule 2 of *do*-calculus and the **ID** algorithm.	
Parameters	`Y`	Set of strings containing the variables in $\boldsymbol{y}$.
	`X`	Set of strings containing the intervened variables in $\boldsymbol{x}$.
	`Z`	Set of strings containing the conditional variables in $\boldsymbol{z}$.
	`P`	`Probability` object with the probability distribution P.
	`G`	Graph object, encoding the DAG of the causal model G.
	`ordering`	List of strings containing a topological ordering of the nodes of G.
	`verbose`	Boolean, prints some useful debugging information if enabled.
	`tab`	Integer, keeps track of the depth of the recursion.
Returns	If the conditional causal effect is identifiable it returns a `Probability` object with de computed causal effect $P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{z})$. If the algorithm encounters a hedge it raises an error, providing the two C-forests that form the hedge for $P_{\boldsymbol{x}}(\boldsymbol{y}|\boldsymbol{z})$ in G.	

Function 3.13: Implemented `IDC` function.

We are now going to explain in detail how we have implemented these two lines of algorithm **IDC** in our package.

Line 1

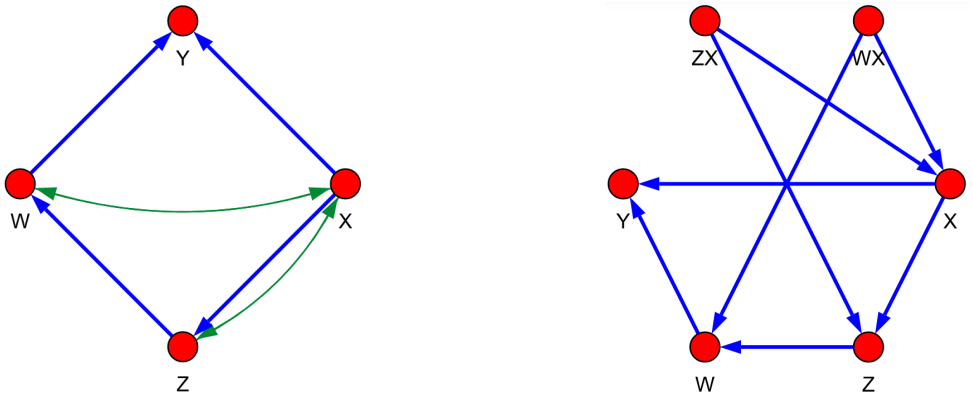
In the first line we have to see if there is a vertex Z' in $\boldsymbol{Z}$ fulfilling the independence suggested. To do so, we must first obtain the graph in which this independence will be checked, and note that the graph is different for each vertex Z' in $\boldsymbol{Z}$. Therefore, we first loop through the vertices in $\boldsymbol{Z}$, and then for each one we construct the required graph. To do so, we first use a newly defined function called `unobserved_graph`, explained in Function 3.14.

Function 3.14 changes the representation of a graph we have used up to this point to a more explicit representation, where exogenous variables confounding two nodes are now

`def unobserved_graph(G):`		
Description	Function that constructs a causal diagram where confounded variables have explicit unmeasurable nodes from a DAG of bidirected edges.	
Parameters	`G`	Graph object, encoding the DAG of the causal model G with bidirected edges.
Returns	A graph object with no bidirected confounding edges, but instead explicit exogenous variables and direct edges to the the affected nodes.	

Function 3.14: Implemented `unobserved_graph` function.

nodes in the graph object. This is necessary for the next steps (deleting arrows and computing d-separation of two sets of nodes), as we will see next, but let us first show how a graph object representation is changed when applying this function, by considering Figure 3.8.



(a) Graph with bidirected edges confounding variables.

(b) Graph with explicit unmeasurable confounders.

Figure 3.8: We always use the graph representation in (a), but for the purposes of checking d-separation we must explicitly state the direct arrows from confounders to variables as in (b).

In Figure 3.8 (a) we see the plot of a graph object where the pairs of variables (X, W) and (X, Z) are confounded by hidden common causes. We see that, after applying the `unobserved_graph` function, these unobserved common causes appear explicitly, named WX for the first pair and ZX for the second, in Figure 3.8 (b). Note that this explicit representation will *only* be used to compute d-separation of sets of measurable variables, and nowhere else.

The following step is to remove incoming edges to $\boldsymbol{X}$ and outgoing edges from Z'. This is where the previously constructed graph comes into play: imagine we have to remove outgoing edges from a confounded vertex X. In the first representation we would delete a directed confounding edge from the bidirected pair (see Figure Figure 3.9 (a)), while in reality a variable that is confounded only has incoming edges, not outgoing! To see this more clearly, consider Figure 3.9, where all outgoing edges of X have been deleted.

In Figure 3.9, if we were to remove outgoing edges from X in the bidirected representation we would be deleting an arrow that it really does not exist. In reality, X has no outgoing edges, and we need the explicit representation to be aware of this.

In our implementation, we have deleted the appropriate incoming and outgoing edges from the explicit version of G as stated in line 1, using the `delete_edges` and `select` functions provided by `igraph`, as we did in line 3 of **ID**. Then we have checked if $\boldsymbol{Y}$ and Z' are inde-

(a) Bidirected representation. (b) Explicit representation.

Figure 3.9: X and Y are confounded by an exogenous variable XY, and outgoing edges from X have been deleted (incorrectly in (a)).

pendent conditional to $\boldsymbol{X}$ and $\boldsymbol{Z} \setminus \{Z'\}$ in the modified graph, $(\boldsymbol{Y} \perp\!\!\!\perp Z' | \boldsymbol{X}, \boldsymbol{Z} \setminus \{Z'\})_{G_{\overline{\boldsymbol{X}}, \underline{Z'}}}$, by using Theorem 2.25 and seeing if they are d-separated. To do so, we have constructed Function 3.15, named `dSep`.

`def dSep(G, Y, node, cond, verbose=False):`		
Description	Checks if the set `Y` and the node `node` are d-separated in G conditional to a set of nodes `cond`.	
Parameters	`G`	Graph object, encoding the DAG of the causal model G.
	`Y`	Set of strings containing some nodes in G.
	`node`	String, node of the graph G.
	`cond`	Set of strings containing some nodes in G.
	`verbose`	Boolean, prints some useful debugging information if enabled.
Returns	Returns `True` if all paths between `Y` and the node `node` conditional to `cond` in G are d-separated, `False` if there is a d-connected path between a node in `Y` and `node`.	

Function 3.15: Implemented `dSep` function.

Function 3.15 iterates through all possible paths between all nodes of `Y` and `node`, and for each path it computes if it is d-separated. To do so we have devised a function to verify if a given path in a graph is d-separated by a set of vertices, called `is_path_d_separated` (see Function 3.16).

`def is_path_d_separated(G, path, cond, verbose=False):`		
Description	Checks if a given path `path` is d-separated in G conditional to a set of nodes `cond`.	
Parameters	`G`	Graph object, encoding the DAG of the causal model G.
	`path`	List if indexes of vertices in G, consisting of a path between the first and last elements of the list.
	`cond`	Set of strings containing some nodes in G.
	`verbose`	Boolean, prints some useful debugging information if enabled.
Returns	Returns `True` if the `path` conditional to `cond` in G is d-separated, `False` if it is d-connected.	

Function 3.16: Implemented `is_path_d_separated` function.

To certify that a path is d-separated in Function 3.16 we have iterated through the vertices of the path, looking for the necessary structures that make a path d-separated (see Definition 2.24). This is, we have looked for either a chain or a fork with a node of the conditional set in the middle ($I \to M \to J$ or $I \leftarrow M \to J$ with $M \in$ `cond` and $I, J \in$ `path`), or for a collider with

no descendants in the conditional set (i.e., $I \to M \leftarrow J$ with $\text{De}(M)_G \cap \textbf{cond} = \varnothing$). Therefore, as we did with the ancestors, we have also created Function 3.17 to compute the descendants of a set of nodes, called `get_descendants`.

`def get_descendants(G, V):`		
Description	Computes the set containing all descendants of a vertex or a set of vertices, including itself.	
Parameters	`G`	Graph object, encoding the direct portion of the causal diagram G (which is a DAG).
	`V`	String with the name of a vertex, or a set of strings containing the names of the vertices.
Returns	A set of strings containing the names of vertices in G which are descendants of `V`.	

Function 3.17: Implemented `get_descendants` function.

Analogously to the `get_ancestors` function, if a single vertex is inputted, Function 3.17 computes its descendants by exploring the directed acyclic graph using BFS (breadth-first search). If, on the other hand, a set of vertices is given, it calls itself for every single vertex in the set and performs the union of the results.

Once we know, using the aforementioned functions, that there is a vertex Z' such that $(\boldsymbol{Y} \perp\!\!\!\perp Z' | \boldsymbol{X}, \boldsymbol{Z} \setminus \{Z'\})_{G_{\overline{\boldsymbol{X}}, \underline{Z'}}}$, we simply call recursively the function `IDC` with exactly the same parameters except for the fact that now Z' is in the *do* set instead of the conditional set (following line 1 of the algorithm **IDC**).

Line 2

This second line of **IDC** is really straightforward, as we do not have to check any condition. We just call the function of the first identification algorithm, `ID_rec`, with the specified parameters, and then create a fractional `Probability` object with the numerator being the returned probability from `ID_rec` and the denominator the same but summed over the variables in $\boldsymbol{y}$.

3.3 Join Implementation of ID and IDC and Usage

In our package we have decided to join the two identification algorithms in just one function, named `ID` (refer to Function 3.18).

Function 3.18 is the one in charge of checking if the the three sets of vertices inputted are pairwise disjoint, as required by both algorithms, and if the directed part of the graph is a DAG. Then it creates the first `Probability` object by setting all the nodes in G as the variables of the distribution. It is also the one that has to compute a topological ordering of the vertices in G, which as we recall only has to be computed once. It does so by passing a graph object G to a function named `get_topological_ordering`, which makes use of the provided function `topological_sorting` of the `igraph` package over the directed part of G (omitting bidirected edges).

To use the developed identification algorithms in our `causaleffect` package, one first has

`def ID(Y, X, G, cond={}, verbose=False):`		
Description	Function that calls either the non-conditional (**ID**) or the conditional (**IDC**) identification algorithm, depending on the parameter `cond`.	
Parameters	`Y`	Set of strings containing the variables in $\boldsymbol{y}$.
	`X`	Set of strings containing the intervened variables in $\boldsymbol{x}$.
	`G`	Graph object, encoding the DAG of the causal model G.
	`cond`	Set of strings containing the conditional variables.
	`verbose`	Boolean, prints some useful debugging information if enabled.
Returns	If the parameter `cond` is empty, this is, if it does not contain any element, the `ID_rec` function is called. Otherwise, if it contains any element, then `IDC` is executed.	

Function 3.18: Implemented `ID` function.

to install `igraph` and `numpy`. If, additionally, one wants to draw plots of the causal diagrams, the package `pycairo` is also required. Once the `causaleffect` package has been imported, we can compute the causal effect in Example 3.3. As we can see from Figure 3.10, we obtain the same result, printed in LaTeX syntax.

```
>>> G = createGraph(['X->Z', 'Z->Y', 'X<->Y'])
>>> P = ID({'Y'}, {'X'}, G)
>>> print(P.printLatex())
\sum_{z}P(z|x)\left(\sum_{x}P(x)P(y|x, z)\right)
```

Figure 3.10: How to create and identify a causal effect with the `caualeffect` package for Python.

We present a few more examples of the usage of the developed package.

Example 3.10. Consider the causal diagram shown in Figure 3.11 (a), from [SP 2006a]. This

(a) Graph G. (b) Graph G'.

Figure 3.11: Causal diagrams.

diagram could be the induced graph of a model for studying how the level of a certain toxin affects the survival rate of a pregnant mother and her child. In this model, W and Z are the afflictions of the mother and the unborn child, respectively. X is the toxin produced in the mother's body due to the illness, and Y_1 and Y_2 are the survival rates of the mother and the child, respectively. Bidirected edges are reasonable confounding variables ($W \dashleftarrow\dashrightarrow Y_1$ would be a common hidden cause affecting the affliction of the mother and her survival expectancy, for example, the existence of a substance affecting both the affliction and the

chance of surviving). Suppose that we have a treatment that can artificially lower the amount of toxin X in the mother's body, and we are interested in how this reduction in X affects the survival rates of the mother and child. In causal theory this is equivalent to compute $P_x(y_1, y_2)$. To compute this causal effect in graph G (Figure 3.11 (a)), we just have to execute the code shown in Figure 3.12.

```
>>> G = createGraph(['W->X', 'X->Y_1', 'Z->Y_2', 'W<->Y_1', 'W<->Z', 'W<->Y_2', 'X
    <->Z'])
>>> P = ID({'Y_1', 'Y_2'}, {'X'}, G)
>>> print(P.printLatex())
\sum_{z}P(y_2, z)\left(\sum_{w}P(w)P(y_1|w, x)\right)
```

Figure 3.12: Computing $P_x(y_1, y_2)$ from G.

We see how the graph G in Figure 3.11 (a) is constructed in the first line, and how the **ID** algorithm is called next. In this case, as we see in Figure 3.12, the effect is identifiable, and equal to

$$P_x(y_1, y_2) = \sum_z P(y_2, z) \left(\sum_w P(w) P(y_1|w, x) \right) = P(y_2) \sum_w P(w) P(y_1|w, x) \ .$$

This means that we can actually know how this variation in X would affect Y_1 and Y_2 without having to perform the potentially hazardous treatment in real patients.

Suppose now that our model is slightly changed due to recent studies that suggest that the affliction of the mother directly influences that of the child. Taking this new information into account in our causal diagram means adding one more edge, $W \to Z$ (see Figure 3.11 (b)). If we now try to identify the same causal effect $P_x(y_1, y_2)$ in this new causal diagram G' we will fail due to the existence of a hedge, as it can be seen in Figure 3.13.

```
>>> G = createGraph(['W->X', 'X->Y_1', 'Z->Y_2', 'W<->Y_1', 'W<->Z', 'W<->Y_2', 'X
    <->Z', 'W->Z'])
>>> P = ID({'Y_1', 'Y_2'}, {'X'}, G)
HedgeFound: Causal effect not identifiable. A hedge has been found:

C-Forest 1:
Vertices: W, X, Y_1, Z, Y_2
Edges: W->X, X->Y_1, Z->Y_2, W->Z, W<->Z, W<->Y_1, X<->Z, W<->Y_2

C-Forest 2:
Vertices: W, Y_1, Z, Y_2
Edges: W->Z, Z->Y_2, W<->Z, W<->Y_1, W<->Y_2
```

Figure 3.13: Trying to identify $P_x(y_1, y_2)$ from G'.

Indeed, we see that the first C-forest found by the algorithm is G' itself, which has roots $\{Y_1, Y_2\}$. Moreover, we have $\{Y_1, Y_2\} \subset \text{An}(\{Y_1, Y_2\})_{G'_{\overline{X}}}$ and $\{X\} \cap G' \neq \varnothing$. The other $\{Y_1, Y_2\}$-rooted C-forest is $G'[\boldsymbol{V} \setminus X]$, which trivially fulfils $\{X\} \cap G'[\boldsymbol{V} \setminus X] = \varnothing$ and also

$G'[\boldsymbol{V} \setminus X] \subseteq G'$. Hence G' and $G'[\boldsymbol{V} \setminus X]$ form a hedge for $P_x(y_1, y_2)$ in G', making this causal effect unidentifiable.

Consider now this following example of a conditional causal effect computation.

Example 3.11. Imagine we want to compute the causal effect $P_x(y|z)$ in the causal diagram G presented in Figure 3.14. If we execute the code in Figure 3.15 (a), we retrieve the desired

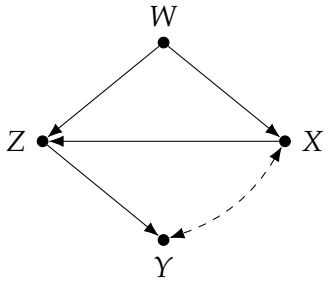


Figure 3.14: Causal diagram G.

```
>>> G = createGraph(['X<->Y', 'Z->Y', 'X->Z', 'W->X', 'W->Z'])
>>> P = ID({'Y'}, {'X'}, G, cond={'Z'})
>>> print(P.printLatex())
\frac{\sum_{x}P(x|w)P(y|w, x, z)}{\sum_{x, y}P(x|w)P(y|w, x, z)}
```

(a) Computing $P_x(y|z)$ from G in Figure 3.14.

```
>>> G = createGraph(['X<->Y', 'Z->Y', 'X->Z', 'W->X', 'W->Z'])
>>> P = ID({'Y'}, {'X'}, G)
>>> print(P.printLatex())
\sum_{w, z}P(w)P(z|w, x)\left(\sum_{x}P(x|w)P(y|w, x, z)\right)
```

(b) Computing $P_x(y)$ from G in Figure 3.14.

Figure 3.15: Computing two causal effects with the implemented package.

conditional causal effect,

$$P_x(y|z) = \frac{\sum_x P(x|w)P(y|w,x,z)}{\sum_{x,y} P(x|w)P(y|w,x,z)} .$$

Observe that, although our package is able to simplify a handful of expressions, the obtained result can be further reduced:

$$P_x(y|z) = \frac{\sum_x P(x|w)P(y|w,x,z)}{\sum_{x,y} P(x|w)P(y|w,x,z)} = \frac{\sum_x P(x|w)P(y|w,x,z)}{\sum_x P(x|w) \sum_y P(y|w,x,z)} = \sum_x P(x|w)P(y|w,x,z) .$$

This expression is interesting, since we do not fix nor condition on the variable W, and yet it appears in the final expression as a *free variable* (meaning it is not summed over all its possible values). In fact, despite appearing in the computed causal effect, said effect is independent of the value w, but in a practical setting some value within the domain of W must be chosen for w. The independence can be easily seen from Figure 3.14, because when intervening X and conditioning on Z we d-separate Y and W, thus making them independent.

If we now want to compute the causal effect $P_x(y)$ in the same causal diagram G, we just slightly change the call to the `ID` function, as we see in Figure 3.15 (b), and we obtain

$$P_x(y) = \sum_{w,z} P(w)P(z|w,x) \left(\sum_{x} P(x|w)P(y|w,x,z) \right) ,$$

which cannot be further simplified.

Consider the following example, from the causal model and data in Example 2.18.

Example 3.12. In Example 2.18 we had a causal diagram that modelled the probability of wearing sunscreen depending on the season and on the weather of that particular day. We want to see if a sunny day causes to wear sunscreen, and to do so we will compute $P(y)$ and $P_x(y)$ and compare them. First of all, we need to calculate an expression for the interventional distribution, and we do it with our `causaleffect` package, as in Figure 3.16.

```
>>> G = createGraph(['Z->X', 'Z->Y', 'X->Y'])
>>> P = ID({'Y'}, {'X'}, G)
>>> print(P.printLatex())
\sum_{z}P(y|x, z)P(z)
```

Figure 3.16: Creation of graph G from Example 2.18 and computation of $P_x(y)$ from G.

We have obtained $P_x(y) = \sum_z P(y|x,z)P(z)$, which is exactly the Back-Door Adjustment (note that Z fulfils the back-door criterion relative to (X,Y)). Now we will compute these probabilities from data in Table 2.3, but to do so we need to remember that we did not have $P(X,Y,Z)$, but instead the conditional probabilities: $P(X,Y,Z) = P(Y|X,Z)P(X|Z)P(Z)$. We first compute what is the probability of wearing sunscreen.

$$\begin{aligned} P(Y=\mathrm{T}) &= \sum_{x,z\in\{\mathrm{T,F}\}} P(X=x, Y=\mathrm{T}, Z=z) \\ &= \sum_{x,z\in\{\mathrm{T,F}\}} P(Y=\mathrm{T}|X=x, Z=z)P(X=x|Z=z)P(Z=z) \\ &= 0.99\cdot 0.90\cdot 0.25 + 0.20\cdot 0.70\cdot 0.75 + 0.60\cdot 0.10\cdot 0.25 + 0.05\cdot 0.30\cdot 0.75 \\ &= 0.354\ . \end{aligned}$$

Then, we compute the probability of wearing sunscreen imposing that it is sunny. This intervention cannot be done in real life, and that is the beauty of it!

$$\begin{aligned} P(Y=\mathrm{T}|do(X=\mathrm{T})) &= \sum_{z\in\{\mathrm{T,F}\}} P(Y=\mathrm{T}|X=\mathrm{T}, Z=z)P(Z=z) \\ &= 0.99\cdot 0.25 + 0.20\cdot 0.75 \\ &= 0.3975\ . \end{aligned}$$

To compare even more results, we also compute the probability of wearing sunscreen given that we see it is sunny.

$$
\begin{aligned}
P(Y = \mathrm{T}|X = \mathrm{T}) &= \frac{\sum_{z\in\{\mathrm{T,F}\}} P(X = \mathrm{T}, Y = \mathrm{T}, Z = z)}{\sum_{y,z\in\{\mathrm{T,F}\}} P(X = \mathrm{T}, Y = y, Z = z)} \\
&= \frac{\sum_{z\in\{\mathrm{T,F}\}} P(Y = \mathrm{T}|X = \mathrm{T}, Z = z)P(X = \mathrm{T}|Z = z)P(Z = z)}{\sum_{y,z\in\{\mathrm{T,F}\}} P(Y = y|X = \mathrm{T}, Z = z)P(X = \mathrm{T}|Z = z)P(Z = z)} \\
&= \frac{\sum_{z\in\{\mathrm{T,F}\}} P(Y = \mathrm{T}|X = \mathrm{T}, Z = z)P(X = \mathrm{T}|Z = z)P(Z = z)}{\sum_{z\in\{\mathrm{T,F}\}} P(X = \mathrm{T}|Z = z)P(Z = z)} \\
&= \frac{0.99 \cdot 0.90 \cdot 0.25 + 0.20 \cdot 0.70 \cdot 0.75}{0.90 \cdot 0.25 + 0.70 \cdot 0.75} = \frac{0.32775}{0.75} \\
&= 0.437 \,.
\end{aligned}
$$

We see that $P(Y = \mathrm{T}|do(X = \mathrm{T})) > P(Y = \mathrm{T})$, and this rise in probability reveals that the sun causes to wear sunscreen. Additionally, we also see that the conditional probability is bigger than the interventional, $P(Y = \mathrm{T}|X = \mathrm{T}) > P(Y = \mathrm{T}|do(X = \mathrm{T}))$. We believe that the probability of wearing sunscreen increases more when we see it is sunny than when we force it to be sunny because when it is sunny it is also more likely to be summer, and this also affects the probability of wearing sunscreen. On the other hand, when we intervene to be sunny it does not care if it is summer or not.

3.4 From Causal Effects to Counterfactual Queries

We have already seen how, using causal diagrams and the *do*-operator, we can answer causal queries from the second level of the Ladder of Causation when they are identifiable. The next natural step would be to study and formalize the concept of counterfactuals, and try to build a technique to answer counterfactual questions from observational data. This journey was undertaken by none other than Shpitser and Pearl [SP 2007], authors of the previous algorithms for computing causal effects. In this section, we will briefly explain an idea of their work regarding the identifiability of counterfactual queries and some algorithms made to evaluate them, although we have not implemented those in our package as they were far too complex and out of the scope of this project.

To be able to answer counterfactual queries, the authors introduced some notation. The reader can refer to [Pea 2000] for an extensive discussion on counterfactuals and the notation used. A variable Y affected by an intervention $do(x)$ is changed into a *counterfactual variable*, and it is denoted by Y_x. Then, questions such as "*what if* $\boldsymbol{X}$ *were* $\boldsymbol{x}$" would be represented as $P(Y_{\boldsymbol{x}}|e)$, where e are observations that induce the probability distribution. There is an intrinsic problem with these types of queries and it is that actions $\boldsymbol{x}$ and evidence e can stand in logical contradiction, and no experimental setup exists which would emulate both the evidence and actions. For instance, there is no experiment that allows us to know the percentage of deaths that could be avoided among people who received a given treatment, had they not taken the treatment. So it is unclear if counterfactual expressions like $P(Y_{\boldsymbol{x}}|e)$, with e and $\boldsymbol{x}$ incompatible, can be estimated consistently.

The authors designed two algorithms [SP 2007] to identify counterfactuals and evaluate them from interventional probabilities when identifiable. To do so, they introduce the con-

cept of *parallel world graphs*, which can be thought as multiple causal diagrams (each one of them representing a possible world, with a precise intervention) sharing the same exogenous variables. An example of a parallel world graph taken from [SP 2007] is shown in Figure 3.17.

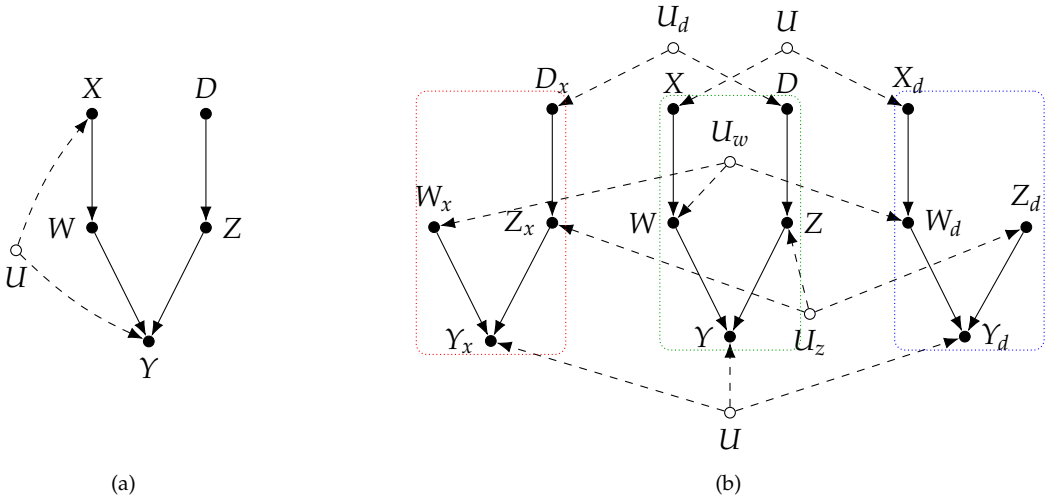


Figure 3.17: (a) Causal graph G of model M. (b) Parallel world graph of G for $P(y_x|x', z_d, d)$, formed by three worlds: the original model M (centre, green) and the submodels M_x (left, red) and M_d (right, blue).

Figure 3.17 (a) is the induced causal graph G of the causal model M. M models how drugs X and D affect intermediate symptoms W and Z, which in turn influence another symptom Y. Suppose that we want to know how likely the patient would be to have a symptom Y given a certain dose x of drug X, assuming we know that the patient has taken dose x' of drug X, dose d of drug D, and we also know how the intermediate symptom Z responds to treatment d. This causal query can be expressed as $P(y_x|x', z_d, d)$, and the parallel world graph would be formed by three worlds: the original model M and also the submodels M_x and M_d, as seen in Figure 3.17 (b). Note that we have not included nodes fixed by actions in the parallel world graph since we already know their values, which are constant.

Parallel world graphs can have duplicate nodes (for instance, in Figure 3.17 (b) we have $Z = Z_x$ since Z is not a descendant of X), and this could arise some errors when computing d-separation. So what they do is they merge duplicate nodes following rigorous criteria and create what they call *counterfactual graphs*, which is equivalent to counterfactual queries as a causal graph is to causal queries.

They present two algorithms similar to the ones for causal queries, called **ID*** (for unconditional queries) and **IDC*** (for conditional queries), that given a causal graph and a counterfactual query return either an error or an expression for computing the counterfactual query from interventional probabilities. Analogously to the causal case, they are recursive, and **IDC*** calls **ID*** as a subroutine. They have to work with counterfactual graphs, so they construct them from the inputted causal graphs using another algorithm also defined in the paper, named **make-cg**. Note that both **ID*** and **IDC*** return interventional probabilities, so to finally obtain results from observational data one has to use the causal effect algorithms already defined in this chapter.

The implementation of these counterfactual identifying algorithms is highly non-trivial and would require the construction of additional classes and functions.

Conclusions

Causal theory is a field in statistics that until recently had not been studied much. The influence of powerful statisticians, like Karl Pearson, discouraged the use of causal graphs to compute causal queries in the first half of the twentieth century, but in the last thirty years, thanks to Pearl and many other scientists, a formalized causal theory has been built.

This mathematization of causal questions reached a peak with the design of deterministic algorithms that identify and, when possible, compute causal effects from observational data. Although known by many scientists, these results are unknown to many others, and to bring them to a wider audience the author has implemented a new Python library that computes causal effects. This package is very practical since in only two lines of code one can construct a causal graph and query a causal effect.

This is not the first developed package with these results. To the best of the author's knowledge, the only implementation of these identification algorithms before the one presented in this work is the R package called `causaleffect`, by Tikka and Karvanen [TK 2017]. In this project, the author has also studied in utmost detail this package and its associated paper and has found a subtle bug in the implementation of **IDC**. This error was reported to the authors, and it has already been fixed in version 1.3.13 (June 14$^{\text{th}}$, 2021).

The implementation and analysis of the identification algorithms presented in this thesis have required the author to go through the necessary background mathematical results on the theories involved. I believe this has been vital to present the results in this dissertation in a more organized and clearer way, and I expect this to be a gateway for more scientists to discover Pearl's remarkable results.

As the author has stated in this work, a logical extension of this project would involve the implementation of counterfactual identification algorithms. To do so, one would have to sail through the obscure notation of counterfactual formalization. If successful, this would finally enable the computation of counterfactual queries, questions of the third level of the Ladder of Causation.

Appendix A

Source Code

The source code of the developed Python library, `causaleffect`, can be found in the following GitHub repository:

`https://github.com/pedemonte96/causaleffect`

Every figure containing code in this dissertation has its equivalent Python script in the provided repository.

Bibliography

[Ash 1970] Ash, R. B. (1970). *Basic probability theory*. Wiley New York.

[BLW 1986] Biggs, N., Lloyd, E. K. and Wilson, R. J. (1986). *Graph Theory, 1736-1936*. Clarendon Press.

[Pea 1993] Pearl, J. (1993). *Comment: graphical models, causality and intervention*. Stat. Sci. **8** (3) pp. 266-269.

[Pea 1995] Pearl, J. (1995). *Causal Diagrams for Empirical Research*. Biometrika **82** (4) pp. 669-688.

[Pea 2000] Pearl, J. (2000). *Causality: Models, Reasoning, and Inference*. Cambridge University Press.

[PGJ 2016] Pearl, J., Glymour, M. and Jewell, N. P. (2016). *Causal inference in statistics: a primer*. Wiley Chichester.

[PM 2018] Pearl, J. and Mackenzie, D. (2018). *The Book of Why*. Penguin Random House UK.

[SP 2006a] Shpitser, I. and Pearl, J. (2006). *Identification of Joint Interventional Distributions in Recursive Semi-Markovian Causal Models*. AAAI Press. pp. 1219-1226.

[SP 2006b] Shpitser, I. and Pearl, J. (2006). *Identification of Conditional Interventional Distributions*. AUAI Press. pp. 437-444.

[SP 2007] Shpitser, I. and Pearl, J. (2007). *What Counterfactuals Can Be Tested*. AUAI Press. pp. 352–359.

[Tian 2002] Tian, J. (2002). *Studies in Causal Reasoning and Learning*. PhD Dissertation, Department of Computer Science, University of California.

[TIOBE 2021] TIOBE Software BV (2021). *TIOBE Index for June 2021*. `https://www.tiobe.com/tiobe-index/`.

[TK 2017] Tikka S. and Karvanen J. (2017). *Identifying Causal Effects with the R Package causaleffect*. J. Stat. Softw. **76** (12) pp. 1-30.

[Ver 1993] Verma, T. S. (1993). *Graphical Aspects of Causal Models*. UCLA Cognitive Systems Laboratory, Technical Report (R-191).

[VP 1988] Verma, T. S. and Pearl, J. (1988). *Causal Networks: Semantics and Expressiveness*. UAI.