From Logs to Causal Analysis: A Guided User Interface for Causal Graph Discovery

by

Trinity Gao

S.B., Computer Science and Engineering, Massachusetts Institute of Technology (2023)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2024

Authored by: Trinity Gao
Department of Electrical Engineering and Computer Science
December 14th, 2023

Certified by: Michael J. Cafarella
Principal Research Scientist, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Mater of Engineering Thesis Committee

From Logs to Causal Analysis: A Guided User Interface for Causal Graph Discovery

by

Trinity Gao

Submitted to the Department of Electrical Engineering and Computer Science
on December 14th, 2023 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

In a world full of digital systems, logs are found everywhere. From distributed systems logging network events to stock exchanges logging transactions, preserving information in logs is a widely-used practice. Our group's hope is that logs can preserve events and system states at various points in time, which can later be leveraged to answer causal questions about the system.

However, analyzing logs is currently far from a smooth experience. Some system dynamics might only be partially captured by log variables, while others are drowned out by the sheer volume of uninteresting, "common-case" log-lines. It is not always possible to require the logging format to match our analysis, since most systems rely on infrastructure code and libraries that cannot be altered directly. We would also be throwing away a considerable amount of existing logs.

An existing system, Sawmill, is able to parse and process log data in order to answer causal questions. Sawmill's main functionalities include presenting the user with candidate answers to causal questions, and relies on user input to accept or reject them. Doing this iteratively allows a user to build up a causal graph for a system's logs. However, the user currently has no way to verify Sawmill's answers. So if a user incorrectly accepts or rejects an edge representing a causal relationship based off of Sawmill's answers on average treatment effect (ATE), this will be integrated into the user's causal graph and can cause even more errors further down the line.

In this master's thesis, we extend Sawmill's capability by identifying and presenting key assumptions which greatly impact Sawmill's answer to a causal question. The existence or non-existence of these assumptions informs the user about possible different states of the causal graph, providing more context about the log and ultimately allowing the user to be more confident in drawing causal conclusions. This also mitigates cascading effect of a single error in the construction of a causal graph. Importantly, we continue to leverage the user's knowledge about the log, relying on their ability to accept and reject assumptions.

Thesis supervisor: Michael J. Cafarella
Title: Principal Research Scientist

Acknowledgments

I would like to thank Mike Cafarella for inviting me into his lab and his guidance on this project, as well as his remarkable insights in the world of causality. I would also like to thank Markos Markakis for welcoming me into the Sawmill project and his indefinite patience and thoughtfulness in explaining everything to me.

I am also grateful to everyone else on the project, including but not limited to Brit, Chunwei, Sylvia, Rana, Ibrahim, and Peter. Of course, I cannot forget the rest of the group for welcoming me into the graduate school world and for all the invaluable advice.

Finally, thank you to my friends and family for supporting me through MIT, both my undergraduate years and during the MEng.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Causality and Logs

1.1 Introduction

It can be difficult to reason about the behavior of complex systems based solely on their outputs. Is the output wrong because one line of code, out of thousands, was buggy? Or was it that the OS scheduled some threads in an unanticipated pattern? Could increasing the amount of available memory help get better performance? Why was the data of a particular customer corrupted? These questions could, in theory, be answered with a counterfactual experiment, but this is often impractical due to the difficulty and cost of such an experiment. For example, a study found that the median preparation cost of a randomized control trial (RCT) in 2016 was $72,600 with a 92 days approval time [1].

Faced with the infeasibility of such a strategy, developers would rather capture as much interesting information as possible during the normal course of software execution in the form of a log: a semi-structured chronological account of key events, appended to a text file as they happen. Logs can later be inspected to reconstruct the series of events that led to a particular outcome of interest.

Many questions in data science, as well as the questions we would like to answer regarding logs, are fundamentally causal questions, in which a key component is *intervention* [2]. Unlike statistical learning, which provides a description of reality that only holds when the experimental conditions are held, causal learning models the effect of interventions and distribution changes. In this way, we can identify what variables might be a cause, and what variables might actually be a confounder.

In many cases, and especially with logs where the user might be a system manager, the user has valuable information about the data. For example, while an algorithm might identify a strong correlation between the number of CPUs and the latency of a system, a system manager may be able to confidently say that the relationship is causal. Similarly, an algorithm can discover that there is a high correlation between the cost of an electricity bill and presence of snow, but most humans can identify that there is no causal relationship. In fact, both are caused by lower temperatures. As such, we aim to leverage user knowledge, both

common-sense and system specific, in a guided interface for causal graph discovery.

1.2 Background

Causality is a large and well-studied field, and there have also been many studies on analyzing logs, so we will define some key terms in this section.

1.2.1 Log Anomalies

Developers are often interested in uncommon outcomes, usually labeled in literature as *anomalies*. This is why most of the automated tools that have been developed to analyze logs focus on anomaly detection, which is achieved by building patterns based on a stream of log messages and then detecting outliers. Automated anomaly detection has been done in various ways, including using NLP and deep learning methods [3]–[5]. However, the information captured in log files can be used to aid a much richer type of system understanding. Beyond simply detecting anomalies, Sawmill leverages logs to understand the causal relationships among the logged system variables *during the course of normal system operation*.

1.2.2 Modeling and Estimating Causality

Correlation is very easily measured, most popularly with the Pearson correlation coefficient [6].

Definition 1 (Correlation) *Correlation is defined as any statistical relationship, whether* ***causal or not****.*

Causality however, is much harder to determine. Causality is a much studied field, and there exists many frameworks to model and estimate causality. Two common methods include Rubin's Potential Outcome Model [7], and Pearl's Graphical Causal Model [8]. Pearl's method accounts for an *interventional setting*, where the value of T is assigned to each unit at random instead of being left to be partially determined by $\boldsymbol{Z}$ - like an RCT *assigning* the creation of an index only to some users, as opposed to studying the users that happened to have one available. Formally, the action of assigning the value t to T is denoted using Pearl's `do`-operator as $\texttt{do}(T = t)$, to distinguish it from simply *observing* that T took the value t [8]. In such a setting, we can then define the Average Treatment Effect [9] of (a binary) T on Y as follows:

Definition 2 (Average Treatment Effect (ATE))

$$ATE(T, Y) = E[Y \mid \texttt{do}(T = 1)] - E[Y \mid \texttt{do}(T = 0)]$$

However, it is often infeasible, whether for ethical or resource reasons, to run RCTs. For example consider the question "does smoking cause lung cancer?". It would take both a great deal of time and a compromising of ethics to apply the treatment, which might be smoking for some frequency over some length of time, to a random group of people.

As such, establishing causality in an *observational* setting requires controlling for confounders.

1.2.3 Potential Confounders

Informally, controlling for confounders with observational data consists of computing the ATE for each value of $\boldsymbol{Z}$ where $\boldsymbol{Z}$ is the set of confounders and weighing the results appropriately. For this to work, two assumptions come into play [10]:

- **Unconfoundedness**: The entire set of confounders $\boldsymbol{Z}$ is observed, so that it can be controlled for.
- **Positivity** (or **Overlap**): For each value of $\boldsymbol{Z}$, we have at least one observation for each possible value of T to use for the ATE calculation.

Under them, we can calculate the ATE from observational data. See Figure 1.1 for an example of a confounding variable.

Definition 3 (Average Treatment Effect (ATE) from observational data) *satisfying unconfoundedness and positivity is*

$$ATE(T,Y) = E_Z\left[E\left[Y|T=1,\boldsymbol{Z}=\boldsymbol{z}\right] - E\left[Y|T=0,\boldsymbol{Z}=\boldsymbol{z}\right]\right]$$

Proof-of-concept studies on uncovering interpretable potential confounders have also been conducted with success. The Zeng et al. study [11] explored how unstructured clinical text can be used as an alternative to RCTs in order to reduce selection bias. They examined localized prostate and lung cancer patients, using NLP techniques to identify terms in electronic medical records that were predictive of both the treatment *and* the survival outcome, labeling these as potential confounders. These potential confounders are then evaluated by calculating the hazard ratio with and without the confounders, where hazard ratio is defined as the ratio of hazard rates corresponding to a set of conditions. The study successful showed that the hazard ratio is shifted to be closer to that of the established RCTs, thus indicating the viability of the potential confounders identified.

1.2.4 Causal Graph Discovery

Causal graph discovery involves inferring causal relationships between variables from observational data. In Pearl's traditional model of inference, causal graph discovery is frowned upon due to a host of reasons [10], [12]–[15]:

1. Identifiability: there can be multiple causal models that explain observed correlations between variables.
2. Direction of Causation: Observational data alone cannot determine direction of causation.

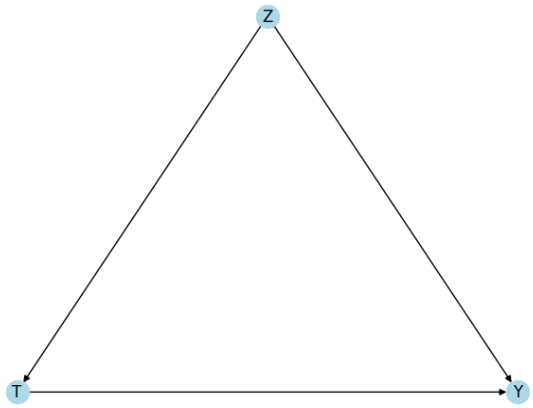


Figure 1.1: Example of confounder Z on treatment T and outcome Y

3. Hidden Mechanisms: Processes that are not directly observable may be involved in the true causal structure.

However, our leveraging of an existing user question and user knowledge makes causal graph discovery more reasonable in our case. In addressing issue #3 (hidden mechanisms), because we are processing logs, it is reasonable that a system manager chooses an adequate level of logging in order to discovery important causal relationships to the system. For issue #1 (identifiability), we build upon a user's specific question of the causal effect of T on Y, which already limits the number of causal models that can explain observed correlations. In addition, we present all findings to the user to accept/reject and assume that they have the knowledge to discern the true causal model amongst reasonable ones. Finally, in addressing #2, our methodology as described in Section 3 builds the graph around the user given causal question, which makes it such that direction of causation does yield different results, so our system is able to discern the difference.

1.2.5 DoWhy

While these ideas have been around for decades, DoWhy [16] is an example of a newly developed tool to make causal analysis on observational data more accessible. It provides a framework to estimate causal effects by modeling the causal question in a graphical model, which may or may not contain a confounder. It supports multiple different causal estimands as well as several different ways to attempt to refute the estimate, such as introducing confounders. This end-to-end tool is robust and easily extensible, making it commonly used in causal analysis. Since we assume that all causal relationships are linear (as explained later in Assumption 1), we use DoWhy's backdoor criterion along with linear regression estimator.

Definition 4 (Backdoor Criterion) *A set of variables Z satisfy the **backdoor criterion** [17] relative to an ordered pair of variables X_i, X_j if*

- *No node in Z is a descendant of X_i*

```
2023-04-15T12:00:00.000000Z INFO user_1 SDK.UserLog System version: 1
2023-04-15T12:00:01.000000Z INFO user_2 SDK.UserLog System version: 1
2023-04-15T12:00:02.000000Z INFO user_3 SDK.UserLog System version: 2
2023-04-15T12:00:03.000000Z INFO user_4 SDK.UserLog System version: 2
2023-04-15T12:00:04.000000Z INFO user_5 SDK.UserLog System version: 1
2023-04-15T12:00:05.000000Z INFO user_6 SDK.UserLog System version: 1
2023-04-15T12:00:06.000000Z INFO user_7 SDK.UserLog System version: 2
2023-04-15T12:00:07.000000Z INFO user_1 SDK.UserLog User exited state INIT in 3 ms
2023-04-15T12:00:08.000000Z INFO user_1 SDK.UserLog User exited state PROCESSING in 77 ms
2023-04-15T12:00:09.000000Z INFO user_6 SDK.UserLog User exited state INIT in 8 ms
2023-04-15T12:00:10.000000Z INFO user_6 SDK.UserLog User exited state PROCESSING in 88 ms
2023-04-15T12:00:11.000000Z INFO user_3 SDK.UserLog User exited state INIT in 4 ms
2023-04-15T12:00:12.000000Z INFO user_2 SDK.UserLog User exited state INIT in 6 ms
2023-04-15T12:00:13.000000Z INFO user_2 SDK.UserLog User exited state PROCESSING in 86 ms
2023-04-15T12:00:14.000000Z INFO user_3 SDK.UserLog User exited state PROCESSING in 81 ms
2023-04-15T12:00:15.000000Z INFO user_2 SDK.UserLog User exited state FINISHED in 22 ms
2023-04-15T12:00:16.000000Z WARN user_0 SDK.StorageManager Local storage is full; deleting backed up files
2023-04-15T12:00:17.000000Z INFO user_1 SDK.UserLog User exited state FINISHED in 18 ms
2023-04-15T12:00:18.000000Z INFO user_6 SDK.UserLog User exited state FINISHED in 21 ms
2023-04-15T12:00:19.000000Z WARN user_0 SDK.StorageManager Local storage is full; deleting backed up files
2023-04-15T12:00:20.000000Z INFO user_4 SDK.UserLog User exited state INIT in 7 ms
2023-04-15T12:00:21.000000Z INFO user_4 SDK.UserLog User exited state PROCESSING in 76 ms
2023-04-15T12:00:22.000000Z INFO user_5 SDK.UserLog User exited state INIT in 2 ms
2023-04-15T12:00:23.000000Z INFO user_5 SDK.UserLog User exited state PROCESSING in 72 ms
2023-04-15T12:00:24.000000Z INFO user_7 SDK.UserLog User exited state INIT in 5 ms
2023-04-15T12:00:25.000000Z INFO user_5 SDK.UserLog User exited state FINISHED in 19 ms
2023-04-15T12:00:26.000000Z INFO user_7 SDK.UserLog User exited state PROCESSING in 80 ms
```

Figure 1.2: Log used for running example demonstrating Sawmill's functionality

- *Z blocks every path between X_i and X_j that contains an arrow into X_i*

As such, when given a graph, DoWhy searches for variables the satisfy the backdoor criterion and control for them. It then uses the linear regression estimator to calculate the ATE.

1.3 Sawmill

Our existing system, Sawmill, is able to parse a log and process the data such that it is amenable to causal analysis. It can then answer questions about candidate causal variables and confounders. It is ultimately up to the user to accept or reject these candidates. For this section, we will use a running example starting from the log shown in Figure 1.2. Sawmill was developed primarily by Markos Markakis, with help from me on experiments.

1.3.1 Log Parsing

First, because of the unstructured format of logs, Sawmill must parse the logs in order to transform them into structured events, which can then be used to learn about critical system behavior. Traditionally, log parsing relies heavily on regular expressions [18]. However, as

the sheer volume of logs increases rapidly and the variables of interest become more irregular, this approach has become more and more infeasible. As such, Sawmill uses Drain [19], which provides an online log parsing approach which does not require knowledge about the structure of the underlying log. In fact, it does not require anything other than raw log messages. It utilizes a fixed depth parse tree to avoid constructing a deep and unbalanced tree, and encodes specially designed parsing rules in the parse nodes. Drain is able to achieve higher accuracy on most datasets at over 50% improvement in runtime compared to the state-of-the-art online parser at the time. After parsing, we have a table with one row per log-line and one column per discovered variable.

1.3.2 Defining Causal Units

After parsing, we obtain a table in which each row represents a log line, and each column represents a parsed variable. This table is often incomplete, however, because it is highly unlikely that each log line contains every variable. To address this issue, observations will be merged based on a user-defined causal unit. To define causal units, a user must first select a base variable (column of the parsed table); examples may include user IDs, machine IDs, and machine locations, although the exact choices will depend on the log and the user's intended analysis. To maximize the utilization of the available data, the chosen base variable should have a low number of missing values in the parsed table. With Drain [19], this can be ensured by explicitly capturing the value of this variable in a pre-processing step (e.g. through a regular expression), rather than automatically extracting it in the normal course of the algorithm. In certain cases, the user may also select a "granularity" parameter to allow for subdivisions (e.g., subnet-based grouping of IP addresses).

Beyond data availability, the choice of a causal unit definition should reflect the Stable Unit Treatment Value Assumption related assumptions, as described below in Definition 5. Informally, causal units should ideally be chosen so that they have no interaction among them. In particular, the choice of causal units should reflect the following condition:

Definition 5 (Stable Unit Treatment Value Assumption (SUTVA)) *The outcome of each unit does not depend on the treatments of* other units*. For example, if we choose machines to be the causal unit, what happens on one machine most not affect the behavior of other machines [20].*

1.3.3 Generating Variables

After selecting a causal unit definition, there might exist more than one value for a specific variable in each unit. Even worse, it might still be the case there is no value for that variable in a unit. This might happen if that variable was unobserved across all lines in the causal unit, in which case we need to decide how to handle the unit as a whole. Our framework addresses these problems by generating appropriate aggregates for each variable and selectively imputing missing values, as described below.

Aggregating values

Log lines that are mapped to the same causal unit form a *causal unit group*. For downstream analysis, we would like to condense all the information within each causal unit group into a single observation. Within each causal unit group, there might be **base variables** that are observed on multiple log lines. However, how to best reconcile them may often be unclear a priori. Thus, for each causal unit group, we derive a fixed set of **prepared variables** for each base variable, by applying a consistent collection of aggregation functions. Users can define their own aggregation function(s) for each log variable (or class of log variables) in the parsed table if the context requires it. For example, the user might be interested in a custom aggregation function that computes total cloud resource costs from usage information in the log. In simple cases, users can use reasonable defaults like the following:

- **Numerical variables:** Minimum, maximum and mean.
- **Categorical variables:** Latest value, earliest value.
- **Timestamps:** Earliest/latest value in each causal unit group.
- **Log Template Indicator Variables:** Sum, OR, AND.

This means that, for example, when using the default aggregation functions, each numerical base variable will lead to three prepared variables, representing its minimum, maximum and mean value within each causal unit group, regardless of the number of observations of said variable within each causal unit group.

The prepared variables that correspond to categorical variables need to be converted to numerical variables for downstream use. To achieve this, we use one-hot encoding to turn each categorical variable into a collection of binary ones.

Imputing missing values

At the other extreme, there might be base variables that are never observed in a causal unit group. For example, if we use user-based causal units, some particular error message that reports the number of failed jobs might only be observed for one user. One solution to this issue is to simply disregard any causal units with missing values. However, this may both drop an excessive number of causal units (in the example we just gave, we would be left with only one) and result in *selection bias*, where certain samples are preferentially excluded from the data. Selection bias may, in turn, impact the accuracy of causal inference.

An alternative approach that is often available is to impute missing values with a reasonable default based on domain knowledge. In the example from the previous paragraph, it may be reasonable to impute the "number of failed jobs" variable with the value 0 for any users for which the corresponding error message was not observed. Care must be taken in this kind of imputation to ensure that the SUTVA [20] is not violated - the imputed values should be *known but unobserved*, not *derived from other causal units* (e.g. we should not impute the mean). For this care to be exercised, we leave this step at the discretion of the user.

```
s.set_causal_unit("ID")
s.prepare()
```

✓ 0.4s Python

```
Causal unit set to ID (tag: ID)
Calculating aggregates for each causal unit...
```

Performing imputation...: 100% 19/19 [00:00<00:00, 1216.76it/s]

One-hot encoding categorical variables...: 100% 7/7 [00:00<00:00, 310.70it/s]

Dumping prepared log to pkl file...: 100% 1/1 [00:00<00:00, 78.68it/s]

Dumping prepared variables to pkl file...: 100% 1/1 [00:00<00:00, 95.14it/s]

```
Successfully prepared the log with causal unit ID (tag: ID)
```

Figure 1.3: Sawmill setting the causal unit and preparing the log

CausalUnitIdentifier	Version_lastseen	FINISHED_count
user_1	1	1
user_2	1	1
user_3	2	0
user_4	2	0
user_5	1	1
user_6	1	1
user_7	2	0

Figure 1.4: Setting causal unit to user ID

Any causal units with missing values after the possible imputation step are disregarded from calculations involving the variable(s) they are missing values for.

In our running example, if we set the causal unit to be "ID", and ask it to **prepare** the variables, we can see how the system first groups variables by causal unit, then performs aggregations within each causal unit and imputes missing values as shown in Figure 1.3. Then, the resulting data is shown in Figure 1.4, with `Version_lastseen` and `FINISHED_count` as the aggregated variables.

Algorithm 1.3.1 Prune: Pruning prepared variables based on variables V.

```
function PRUNE(prep_var_data, graph, V)
    exclusion_set ← graph.nodes ∪ V
    for var in excl_set do
        excl_set.add(prep_var_data.sameBaseVariableAs(var))
    end for
    done ← False
    while done == False:
        X ← prep_var_data.drop(excl_set)
        X, scale ← StandardScaleNumericalColumns(X)
        coefs ← MultiTaskLasso(X, V)
        unscaled_coefs ← scale.Unscale(coefs)
        mapping ← {prep_var_data.var_names : unscaled_coefs}
        mapping.drop(unscaled_coefs ==0)
        pruned ← mapping.var_names
        done ← True
        if pruned.baseVars.hasDups() then
            done ← False
            pruned.sortby(abs(unscaled_coefs), descending)
            excl_set.add(pruned.lowRankedDups())
        end if
    end while
    return pruned
end function
```

Algorithm 1.3.4 CandidateConfounders: Finds candidate confounders for an edge S->D.

```
function CANDIDATECONFOUNDERS(prep_var_data, graph, S,D)
    // Get baseline effect of S on D.
    nodes, edges ← [S,D] , [[S,D]]
    g ← DiGraph(nodes, edges)
    baseline_effect, _ ← GetEffect(prep_var_data, S, D, g)
    // Evaluate candidate confounders.
    candidates ← Prune(variables, graph, S,D)
    scored ← []
    for C in candidates do
        nodes, edges ← [S,D,C] , [[S,D], [C,S], [C,D]]
        g ← DiGraph(nodes, edges)
        effect, p_value ← GetEffect(prep_var_data, S, D, g)
        scored.append(candidate, effect, p_value, effect/baseline_effect)
    end for
    scored.sortby(effect/baseline_effect, ascending)
    return scored
end function
```

Algorithm 1.3.2 GetEffect: Calculates the average treatment effect (ATE) of T on O, as well as its its significance.

```
function GetEffect(data, T, O, graph)
    m ← CausalModel(data[graph.nodes], T, O, graph)
    e ← m.identify_effect()
    s ← m.estimate_effect(e)
    effect ← s.value
    p_value ← s.test_stat_significance()
    return effect, p_value
end function
```

Algorithm 1.3.3 CandidateCauses: Finds candidate causes for variable O.

```
function CandidateCauses(prep_var_data, graph, O)
    candidates ← Prune(variables, graph, O)
    scored ← []
    for C in candidates do
        nodes, edges ← [C,O] , [[C,O]]
        g ← DiGraph(nodes, edges)
        effect, p_value ← GetEffect(prep_var_data, C, O, g)
        scored.append(candidate, effect, p_value)
    end for
    scored.sortby(p_value, ascending)
    return scored
end function
```

Interactive Causal Discovery

Sawmill supports interactive causal discovery by providing functionality for discovering candidate causes as shown in Algorithm 1.3.3, as well as candidate confounders as shown in Algorithm 1.3.4. For each of these, because most logs already contain hundreds to thousands of variables which is then compounded by the Sawmill's aggregations, Sawmill first prunes the variables according to Algorithm 1.3.1. Then, for finding candidate causes, it calculates the ATE of each candidate cause on the the given variable O using Algorithm 1.3.2, and ranks them by their p_values. For candidate confounders, it considers the graph in which the confounder exists and scores it by evaluating the resulting change in ATE. Algorithm 1.3.2 leverages the existing package, DoWhy [16] which calculates ATEs.

An example of calling `explore_candidate_causes` is shown in Figure 1.5.

1.4 Motivation

However, while Sawmill is able to produce candidate causes with confidence scores for a variable, it only considers the ATEs in the context of the causal graph that has already been built out. This poses two problems:

1. On the first step with none of the graph built out, the ATEs produced are calculated

```
c = s.explore_candidate_causes("Mean Latency")
print(c.show())
```
✓ 0.3s Python

	Candidate	Effect	P-value	Standard Error	Tag
0	be392a96_12+latest=False.	4.827848	[2.496549004629065e-13]	[0.5700038640770166]	Index Presence
1	b9fc65c4_6+latest=Android	4.006784	[6.943275553135476e-09]	[0.6316033520777586]	Platform
2	aa51d219_7+latest=OFF	-3.771285	[6.927547361396966e-08]	[0.6462305365704367]	Power Saving Mode

Figure 1.5: Example output of Sawmill after calling `explore_candidate_causes`

in a vacuum, with no regards to other causal relationships.

2. If a causal edge has been accepted, it will influence results in later iterations. If the accepted edge was incorrect, the errors will propagate.

In regards to the first challenge, we would like to present key edges which significantly impact the ATE of a candidate cause on the outcome of interest. These edges represent assumptions that are made about causal relationship, or assumed to be false, which impact the validity and magnitude of the ATE in question. This solution also mitigates the second challenge, by allowing users to accept edges in the graph with more confidence. We will demonstrate the insights these key edges present in the example below.

1.5 Running Example

Let us define a running example which will be referenced throughout the paper. Consider a log that periodically writes out values for 3 variables: x, y, z. In this example, the values of x are determined by z, while the values of y are determined by both x and z. Namely, the values for each variable at each time step is determined by Algorithm 1.5.1. Thus, the ground truth causal graph is shown in Figure 1.6. Additionally, let us define some notation what will be used throughout the paper:

Definition 6 *ATE(T, Y, G) = the average treatment effect of T on Y, given a causal graph G as calculated by the DoWhy [16] package.*

As can be seen from Figure 1.6, the true ATE of x on y should be 1. However, an model which does not take into account any other causal edges would incorrectly find:

$$ATE(x, y, G(e = \{\})) = 3$$

since

$$y = 2x + x + \epsilon = 3x + \epsilon$$

However, it turns out that it is important to consider the edges (z, x) and (z, y), since their existence would drastically change the ATE:

$$ATE(x, y, G(e = \{(z, x), (z, y)\})) = 1$$

Algorithm 1.5.1 XYZGen: how values for x, y, z are determined

```
1: function XYZGen
2:     z = randint(1, 10)
3:     // Let ϵ be some random noise
4:     x = 2*z + ϵ
5:     y = 4*z + x + ϵ
6: end function
```

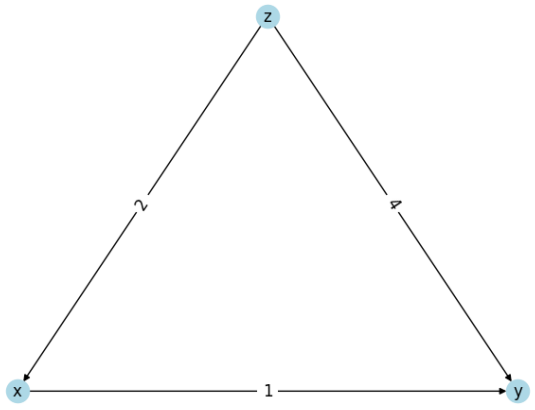


Figure 1.6: XYZ Causal Graph

Chapter 2

Processing Text-Based Logs: An Aside

One limitation of Sawmill is that it is largely equipped to handle numerical variables, and not higher-dimensional values such as strings. In this section, we describe an extension to Sawmill that allows it to answer questions about complex string variables as well. This work is very preliminary, as the focus of the project has pivoted since May 2023, when this work was done, and simply demonstrates proof-of-concept for processing text-based logs. I led the work in creating the synthetic log and adapting Sawmill to run this experiment.

2.1 Experimental Setup

In order to conduct our experiment, we needed a reasonable log that used complex string variables (also referred to as **text**), but also one in which the true causal graph was known to us. As such we generated a log in which there is a clear causal relationship between variables, one of them being text. To do so, we set up an Amazon AWS RDS instance with instance class db.t3.micro and 1 GB of RAM. It also utilizes 5 GiB of magnetic storage. This instance is configured as such so as to amplify the slowdown of large-compute workloads. We then set up two SQL databases, called `small_database` and `big_database`, each with one table, called `small_table` and `big_table`, respectively.

- `big_table`
  - columns x, y, and z
  - each column ranges from 0 to 100
  - 5M rows
- `small_table`
  - columns id and x
  - id is unique and ranges from 0 to 999
  - x is a random integer from 0 to 9
  - 1K rows

Algorithm 2.1.1 Query A: Performs a full scan of table T

```
function QUERY A
    PERFORM * FROM big_table;
end function
```

Algorithm 2.1.2 Query B: randomly updates row and counts number of rows where x=5 and prints

```
function QUERY B
    UPDATE small_table
    SET x = FLOOR(RANDOM()*10)
    WHERE id=FLOOR(RANDOM()*1000);
    RAISE NOTICE 'Number of entries where x=5 is %', (
      SELECT COUNT(*)
      FROM small_table
      WHERE x=5
    );
end function
```

We then define two queries. **Query A** is on `big_table` and simply does a scan of the entire table. The scan is wrapped in a function called foo() in order to suppress the output. **Query B** is on `small_table`, and randomly selects a single row and updates x to be equal to a random value between 0 and 10. It then counts the number of rows where x = 5 and prints it.

Then, query A is fired every minute and query B is fired every 2 seconds. Even though the two queries are on two different databases, because they are running on the same RDS instance and sharing computing power, query A's scan of `big_table` will result in a slowdown of query B's execution. We then have AWS log each query's duration. A truncated example of the log can be seen in Figure 2.1, where the normal duration of a query on `small_database` is on the order of 1 ms. However, when there is a query on `big_database` that is running in parallel, the queries on `small_database` slow down to the order of 50 ms.

2.2 Methodology

In order to allow Sawmill to work with this log, we must find a lower-dimensional, latent representation for the text variables. As can be seen from Figure 2.2, the "Query" variable, representing the PostgreSQL code for Algorithms 2.1.1 and 2.1.2, is logged and parsed as a string. Our preliminary approach is to vectorize the queries by encoding them using bag-of-words, then using scikit-learn's KMeans algorithm to cluster values for this variable. Each cluster is then assigned a unique, numerical value, which Sawmill can work with.

We then used Sawmill with a causal unit of 2 seconds, since Query B was fired every 2 seconds, allowing us the finest granularity of information possible. We used the **mean** aggregation function, but min, max, and mode would have all yielded the same result due to

```
2023-04-04 22:31:12 UTC:18.29.111.229(61344):trinityg@small_database:[1870]:LOG: duration: 1.346 ms statemen…
2023-04-04 22:31:14 UTC:18.29.111.229(61344):trinityg@small_database:[1870]:LOG: duration: 1.292 ms statemen…
2023-04-04 22:31:16 UTC:18.29.111.229(61344):trinityg@small_database:[1870]:LOG: duration: 1.462 ms statemen…
2023-04-04 22:31:18 UTC:18.29.111.229(61344):trinityg@small_database:[1870]:LOG: duration: 1.320 ms statemen…
2023-04-04 22:31:20 UTC:18.29.111.229(61344):trinityg@small_database:[1870]:LOG: duration: 1.357 ms statemen…
2023-04-04 22:31:22 UTC:18.29.111.229(61344):trinityg@small_database:[1870]:LOG: duration: 0.635 ms statemen…
2023-04-04 22:31:24 UTC:18.29.111.229(61344):trinityg@small_database:[1870]:LOG: duration: 1.290 ms statemen…
2023-04-04 22:31:26 UTC:18.29.111.229(61344):trinityg@small_database:[1870]:LOG: duration: 1.300 ms statemen…
2023-04-04 22:31:28 UTC:18.29.111.229(61344):trinityg@small_database:[1870]:LOG: duration: 0.987 ms statemen…
2023-04-04 22:31:30 UTC:18.29.111.229(61344):trinityg@small_database:[1870]:LOG: duration: 53.549 ms stateme…
2023-04-04 22:31:32 UTC:18.29.111.229(61344):trinityg@small_database:[1870]:LOG: duration: 39.491 ms stateme…
2023-04-04 22:31:34 UTC:18.29.111.229(61344):trinityg@small_database:[1870]:LOG: duration: 120.525 ms statem…
2023-04-04 22:31:36 UTC:18.29.111.229(61344):trinityg@small_database:[1870]:LOG: duration: 50.782 ms stateme…
2023-04-04 22:31:37 UTC:18.29.111.229(61341):trinityg@big_database:[1835]:LOG: duration: 9131.186 ms stateme…
```

Figure 2.1: Example Snippet of SQL Log

	Name	Occurrences	Preceding 3 tokens	Examples
0	Date	297	[]	['2023-04-04']
1	Time	297	[]	['22:30:28', '22:30:30', '...
2	Host	297	[]	['18.29.111.229(61344)', '...
3	User	291	[]	['trinityg']
4	Database	291	[]	['small_database', 'big_da...
5	Process Id	291	[]	['[1870]', '[1835]']
6	Duration	291	[]	['3.619', '46.446', '28.51...
7	Query	291	[]	["DO\$$\\\\BEGIN\\\\UPDAT...
8	439221a9_8	3	['checkpoint', 'complete:'...	['8', '12', '7']
9	439221a9_21	3	['recycled;', 'write', '=']	['0.803', '1.131', '0.635']

Figure 2.2: Variables extracted by Sawmill for the SQL Log

the causal unit we chose.

Additionally, because our generated log did not log the beginning of the query, we calculated when a query's execution started by using the timestamp and the logged duration. We were then able to label each row with the number of Query A's and the number of Query B's that were executing at a given time. We can see in Figure 2.3, for example, Query A was mapped to cluster 1 while Query B was mapped to cluster 0. On line 34, Query A was logged to have finished executing after a duration of 9.13 seconds. As such, `count_1` holds the value 1 for lines 30 to 34 (keep in mind that the causal unit is 2 seconds). The rows above line 30 have `count_1 = 0`. `count_0` is always 1 since the causal unit is 2 seconds, and Query B is fired every 2 seconds.

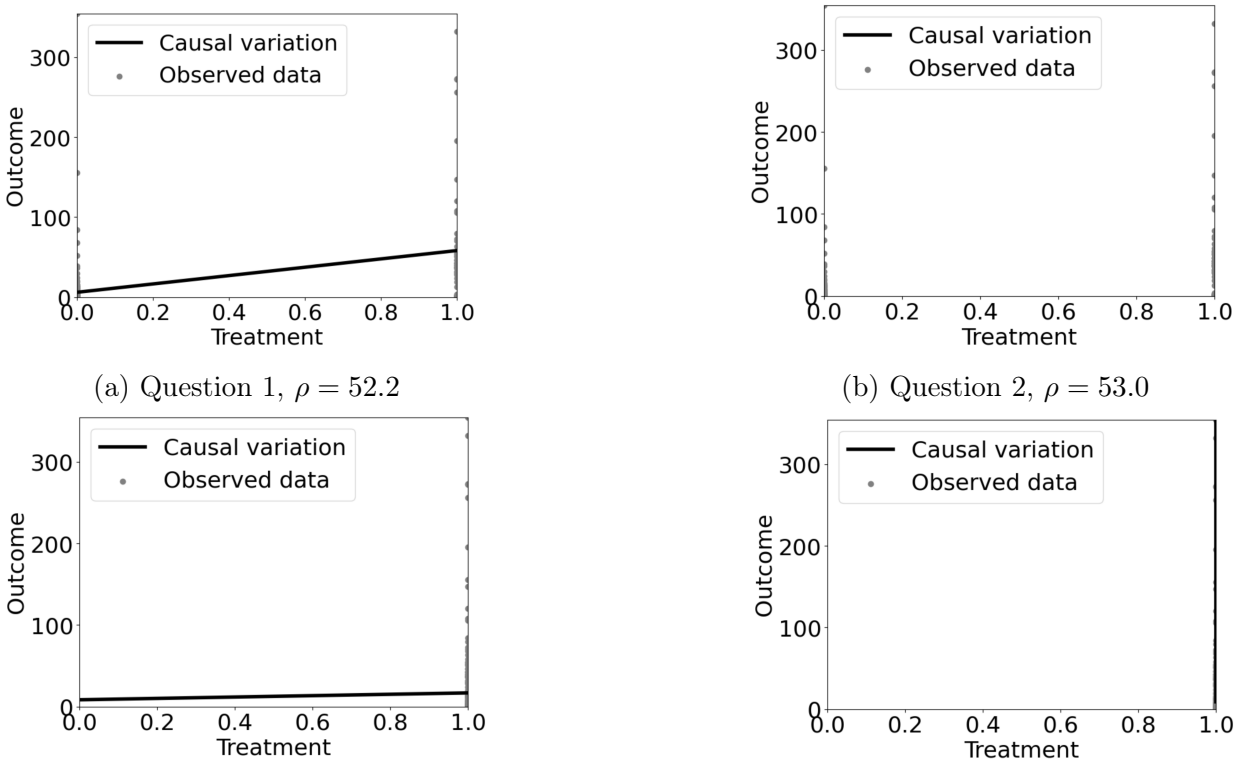
	Duration_0_mean	Duration_1_mean	causal_unit	count_0	count_1
20	1.225	NaN	20.0	1	0
21	1.209	NaN	21.0	1	0
22	1.346	NaN	22.0	1	0
23	1.292	NaN	23.0	1	0
24	1.462	NaN	24.0	1	0
25	1.320	NaN	25.0	1	0
26	1.357	NaN	26.0	1	0
27	0.635	NaN	27.0	1	0
28	1.290	NaN	28.0	1	0
29	1.300	NaN	29.0	1	0
30	0.987	NaN	30.0	1	1
31	53.549	NaN	31.0	1	1
32	39.491	NaN	32.0	1	1
33	120.525	NaN	33.0	1	1
34	50.782	9131.186	34.0	1	1

Figure 2.3: SQL Log after process from 3.2.4

2.3 Results

Finally, we use Algorithm 1.3.2 to ask several questions about our system. Full results can be seen in Figure 2.4. As a reminder, the true causal relationships in our system is that `count_0` affects `Duration_0_mean` and that `count_1` affects `Duration_1_mean`. However, `count_1` should **also** affect `Duration_0_mean` because of our experimental setup: shared computing resources between the two queries, and the high amount of resources Query A requires.

- Question 1: Figure 2.4a
  - Cause: `count_1`
  - Effect: `Duration_0_mean`
  - Confounder: None
  - ρ: 52.2
- Question 2: Figure 2.4b
  - Cause: `count_1`
  - Effect: `Duration_0_mean`
  - Confounder: `count_0`
  - ρ: 53.0
- Question 3: Figure 2.4c
  - Cause: `count_0`



(a) Question 1, $\rho = 52.2$

(b) Question 2, $\rho = 53.0$

(c) Question 3, $\rho = 8.42$

(d) Question 4, $\rho = -14 \times 10^{15}$

Figure 2.4: Results of the SQL experiment

  - Effect: `Duration_0_mean`
  - Confounder: None
  - ρ: 8.42

- Question 4: Figure 2.4d
  - Cause: `count_0`
  - Effect: `Duration_0_mean`
  - Confounder: `count_1`
  - $\rho : -14 \times 10^{15}$

As can be seen, our system correctly finds that there is a strong causal relationship between the number of Query A's executing and the average duration of Query B, *even with the number of Query B's executing given as a confounder* (see Question 1 vs Question 2). Our system also finds that there is a causal relationship between the number of Query B's running and the average runtime of Query B, but it is not as strong (8.42 vs > 50). However, when asked about the number of Query A's as a confounder on this relationship, it correctly identifies that the number of Query A's is a confounder.

Chapter 3

Identifying Key Edges in the Causal Graph

While Sawmill presents users with candidate causes of a variable and a p_value, there exists the implicit assumption that no other edges exist in the causal graph beyond what is already built out by the user. In other words, the candidate causes are presented along with an ATE value, but this ATE value assumes that there are no other causal dependencies in the data. This is not true in many cases.

However, assuming the non-existence of an edge often does not effect the ATE of a candidate cause on our variable of interest. For example, consider the XYZ example from Section 1.5. Let's say there's another random variable w, such that the true causal graph is as shown in Figure 3.1. Since w is completely random and not correlated to y, the ATE calculation to remain the same:

$$ATE(x, y, G(e = \{(w, y)\})) = 3$$

As such, we propose an algorithm which, given a treatment variable T and outcome variable Y, identifies key edges which represent assumptions that the user should consider, since the existence/non-existence of such an edge in the causal graph would have a significant impact on the ATE of T on Y. I led the algorithmic development as well as experimentation portion of this work.

3.1 Methodology

There exists two main challenges to our problem:

1. The large number of variables in a typical log poses a computational challenge.
2. The domain of ATE values largely depend on the domain of the input data, so it is impossible to set a fixed threshold for determining whether a difference in ATE is considered significant.

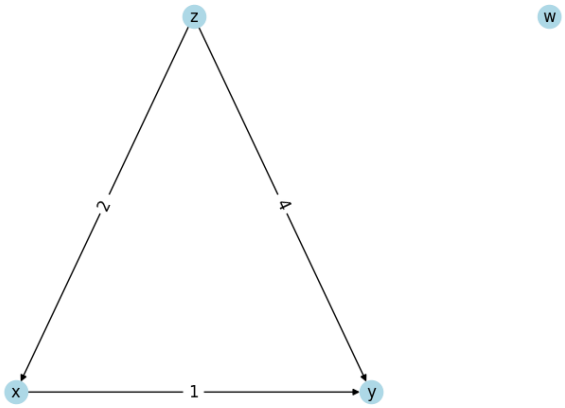


Figure 3.1: XYZW Causal Graph

3.1.1 Challenge #1: Computational Challenge

A naive approach in order to determine whether an edge represents an important assumption for the user to consider or not is to evaluate

$$\|ATE(T, O, G(e = \{\})) - ATE(T, O, G(e = \{e\}))\|$$

However, just because an edge does not contribute any difference to the ATE calculation, does not mean it is not significant. Consider the causal graph shown in Figure 3.2. Evaluating

$$\|ATE(x, y, G(e = \{\})) - ATE(x, y, G(e = \{(z, w)\}))\| = 0$$

yields 0, but it is clear that (z, w) is still an important edge to consider.

Thus, we must consider causal graphs as a whole. Then, if there are V nodes in a graph, there are $O(V^2)$ edges. Each edge has three states: non-existent, existing in one direction, existing in another direction. Thus, the total number of graphs is $O(3^{V^2})$. Not all of these will be DAGs, but the number of graphs we must consider is still exponentially large.

3.1.2 Challenge #2: Determining When an ATE Change is Significant

Additionally, once we are able to find ATE values to compare, how can we determine whether the difference is significant or not? Namely, for what value τ will

$$\|ATE_1 - ATE_2\| \geq \tau$$

yield useful information? To illustrate this challenge, let us again consider the example from Section 1.5. In this example, we had

$$\|ATE(x, y, G(e = \{(z, x), (z, y)\})) - ATE(x, y, G(e = \{\}))\| = 2$$

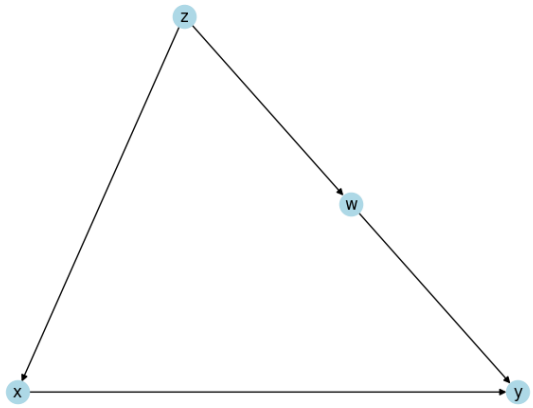


Figure 3.2: XYZ-W Causal Graph to demonstrate ineffectiveness of naive solution proposed in 3.1.1

Algorithm 3.1.1 XYZGen: how values for x, y, z are determined, part 2

```
1: function XYZGen
2:     z = randint(1, 10)
3:     x = 2*z + ϵ
4:     y = 0.5*z + x + ϵ
5: end function
```

If our threshold was $\tau = 0.5$, we would successfully find that these two ATEs are significantly different. However, if we simply had generated the log using Algorithm 3.1.1 instead, then we would have

$$y = 0.25x + x + \epsilon$$

$$ATE(x, y, G(e = \{\})) = 1.25$$

$$ATE(x, y, G(e = \{(z, x), (z, y)\})) = 1$$

$$\|ATE(x, y, G(e = \{(z, x), (z, y)\})) - ATE(x, y, G(e = \{\}))\| = 0.25$$

This value, 0.25, is $< \tau$ and we would discard this difference in ATE as insignificant, even though the user may find it interesting and want to learn about the edge (z, x) and (z, y).

3.2 Pruning Variables

In order to mitigate challenge #1 as described in Section 3.1.1, we first prune the number of variables we consider.

3.2.1 Pruning Aggregates

Sawmill handles multiple values for a single variable in a single causal unit by performing various aggregations, as described in Section 1.3.3. However, in the case where a variable is

Algorithm 3.2.1 PruneAggregates: prunes variables by selecting one aggregate variable to represent a variable when possible

```
function PruneAggregates(V)
    // max absolute differences across all causal units
    mean_diffs ← {}
    mean ← {}
    for v in V do
        // Calculate mean of the variable to use for normalization
        mean ← 1/n ∑ v
        mean_diffs[v] ← 1/(n*mean) ∑_{i∀cu} max(v_aggs) - min(v_aggs)
    end for
    kept ← []
    for v in V do
        if mean_diffs[v] = 0 or mean_diffs[v] ≤ Q_10(mean_diffs) then
            kept.append(v_aggs[0])
        else
            kept.extend(v_aggs)
        endif
    end for
    return kept
end function
```

numerical and there is a single occurrence in each causal unit, the different values for the aggregations may be the same. For example, consider variable x with one value, 5, in a certain causal unit. Then, x_mean, x_min, x_max will all hold 5.0. In this case, it is unnecessary to consider all three of them, as they essentially represent the same variable. Furthermore, if aggregates only differ by a small amount within each causal unit, the different aggregates probably have a very similar impact on ATEs.

As such, we apply Algorithm 3.2.1 in order to select one aggregate to represent a variable if the maximum absolute difference between aggregates is 0, or in the bottom 10% of maximum absolute differences amongst all variables after normalization.

3.2.2 Prune via ATE Impact

Next, we further prune variables by processing them one at a time. The idea is that there are only so many causal graphs possible containing three variables. Take into account the condition that there must be a path from the treatment to the outcome and how ATE is calculated, there are then only 5 graphs in which the 3rd variable has an impact on ATE. As such, we proceed by considering only these causal graphs, as shown in Figure 3.3. We then calculate $ATE(T, Y, G)$ for each G in the graphs previously described, and rank variables by the maximum absolute difference in ATE values produced. Finally, we return the top n variables as important variables we should consider enumerating DAGs, described later on. This process is described in Algorithm 3.2.2

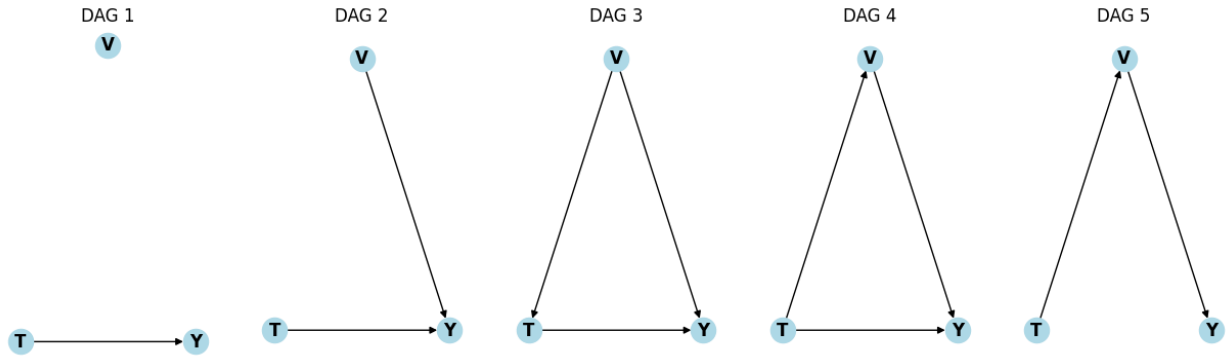


Figure 3.3: Simple graphs involving treatment T, outcome O, and variable V

Algorithm 3.2.2 PruneVariables: prune variables by ranking variables by the largest difference in ATE produced using different graphs, and taking the top n

```
function PruneVariables(T, Y, n)
    ate_diffs ← {}
    for v in V \{T, Y} do
        ates ← [ATE(T, Y, G) for G ∈ graphs shown in 3.3]
        ate_diffs[v] ← max(ates) - min(ates)
    end for
    ate_diffs.sort()
    return ate_diffs[:n]
end function
```

Proof of Pruning Algorithm

Let us first prove that the rightmost 4 graphs in Figure 3.3 are the only ones in which V may affect $ATE(T, Y)$ and thus considering these DAGs are sufficient to identify if a variable is impactful in the ground truth causal graph. We keep the leftmost graph as it served as the baseline for ATE comparisons.

Proof 1 (DAGs in Figure 3.3 are sufficient) *Similarly to Definition 3, ATE with observed data where the treatment can take on more than 2 values is*

$$ATE = \frac{1}{K}\sum_{k=1}^{K}(E[Y|T=k] - E[Y])$$

For the ATE calculation to change, there must be an observed covariate V, which is introduced by an edge from V to Y. The new ATE calculation is then

$$ATE = \frac{1}{K}\sum_{k=1}^{K}(E[Y|T=k, V] - E[Y|V])$$

With the new constraint of edge (V, Y), the only four possible DAGs are as shown.

Next, since these are the variables we will consider in order to discover key edges later on, we will prove that the set of variables found are a **superset** of the variables that comprise of interesting edges. In order to do so, we make two key assumptions:

Assumption 1 *All causal relationships are linear.*

Assumption 2 *Most ground truth causal graphs are sparse, so the number of variables that comprise of interesting edges are limited.*

Assumption 1 is often true with datasets, especially those derived from logs, but the inability to consider nonlinear causal relationships is a limitation to our system. Assumption 2 is empirically true as evidenced by the number of papers on algorithms for sparse causal graphs [21], so we believe that these two assumptions are very reasonable. Given this, we claim the following:

Lemma 1 *The set of variables that satisfies either*

1. *an ancestor of Y*
2. *an ancestor of T who is an ancestor of, or shares a common ancestor with Y*

*in the **ground truth causal graph** is a superset of the variables that comprise of the edges we are interested in.*

Proof 2 (Lemma 1) *The first point in the lemma follows from Proof 1. An ancestor of Y has a causal effect on Y, so whether or not we adjust for it as shown in the ATE equation is significant.*

The second describes nodes that satisfy the backdoor criterion as described in Definition 4, which will be controlled for by DoWhy.

Proof 3 (Our algorithm will always find the set of variables satisfying 1 in Lemma 1) *Following assumption 1, if V is an ancestor of Y, it will have a linear relationship with Y, which will be discovered via DAGs 2, 3, 4, and 5 in 3.3.*

Proof 4 (Our algorithm will always find the set of variables satisfying 2 in Lemma 1) *Following assumption 1, if V is an ancestor of T who is an ancestor of, or shares a common ancestor with Y, it will have a linear relationship with the common ancestor, who will have a linear relationship with Y, which will be discovered. It will also have a linear relationship with T which will be discovered via DAG 3 in 3.3.*

3.3 Enumerating DAGs

Next, we will enumerate all possible DAGs with at most k edges formed by the variables remaining to us after pruning. This limitation is another heuristic to improve the efficiency of our system, and is a reasonable one following Assumption 2. Finally, each DAG must contain an edge from T to Y, sine we are examining $ATE(T, Y, G)$, and the DoWhy package requires a path for the calculation[16]. The algorithm to enumerate DAGs is then described in Algorithm 3.3.1.

Algorithm 3.3.1 DagEnumeration: Enumerates all possible dags given a set of possible edges with at most k edges and an edge from T to Y

```
function DagEnumeration(edges, T, Y, k)
    dags ← []
    sequences ← lists of size len(edges) of {-1, 1, None} where sum(abs(list)) ≤ k-1
    for s in sequences do
        graph = G(edges=[s[i]*edges[i] for i in range(len(s))])
        graph.add_edge(T, Y)
        if is_dag(graph) then
            dags.append(graph)
        endif
        return dags
    end for
end function
```

3.4 Clustering ATEs

Now that we have enumerated all DAGs to be considered, we must calculate $ATE(T, Y, G)$ for each DAG, and cluster them. The idea here is that after we cluster the ATE values, we will be able to identify which DAGs produced a certain ATE, and more importantly, *which edges* differentiate the DAGs that produced significantly different ATEs. In order to combat Challenge 2 described in Section 3.1.2, we use a hierarchical clustering algorithm which aims to learn a clustering which does requires neither a distance threshold nor a specific number of clusters. In order to achieve this, we use the "ward" linkage method, defined as below:

Definition 7 (Ward Distance) *Ward distance measures the distance between two clusters u, v as*

$$d(u,v) = \sqrt{\frac{|v|+|s|}{T}d(v,s)^2 + \frac{|v|+|t|}{T}d(v,t)^2 - \frac{|v|}{T}d(s,t)^2}$$

where $T = |v|+|s|+|t|$ and $|*|$ is the cardinality of the argument. Additionally, u is a newly joined cluster consisting of clusters s and t

Then, clusters are hierarchically merged by increasing distance. This algorithm can also be understood as minimizing intra-cluster variance while maximizing inter-cluster variance. Since the algorithm starts by treating each point as a cluster and merging until there is only one cluster left, we must determine a point at which to stop merging. We choose the threshold to be

$$multiplier \times max(distances)$$

Thus, with $multiplier = 0.5$, we would only keep merges that at a distance higher than 0.5 times the distance of the last merge.

We implement this clustering algorithm using scipy's linkage and fcluster methods, as shown in 3.4.1. For ease of use we also allow the user the option to specify the number of clusters, which would not conduct the last $n-1$ merges. If this is specified, we do not use the distance threshold to determine when to stop merging.

Algorithm 3.4.1 Cluster: Given a list of DAGs, T, Y, calculate the $ATE(T, Y, G)$ for all DAGs and cluster the resulting ATEs.

```
function CLUSTER(dags, T, Y, n=None)
    ates ← {dag: ATE(T, Y, dag) for dag in dags}
    data ← ates.values()
    linked ← linkage(data, method="ward")
    if n is not None then
        inconsistencies ← linked[:2] - linked[:2].mean()
        m ← 1
        clusters = fcluster(linked, t=m*max(inconsistencies), criterion='distance')
    else
        clusters = fcluster(linked, t=n, criterion='maxclust')
    endif
    return clusters
end function
```

3.5 Identifying Key Edges

Now that we have ATEs and their corresponding DAGs clustered, we can use this information to identify key edges which represent important assumptions that the user should consider. To do so, we leverage the hierarchical clustering's structure to create a tree and annotate each node with the following

1. Edge counts: how many times does each edge occur within the DAGs of a cluster
2. Expectancy: how many times do we expect an edge to occur within the DAGs of a cluster. This is calculated using each node and its parents' edge counts.

$$expected = \frac{num\_dags}{num\_dags_{parent}} * edge\_count_{parent}$$

3. Percent over/under expectancy: how different does the actual frequency of an edge occurence differ from our expected.

$$\%\ over/underexpectancy = \frac{edge\_count - expected}{expected}$$

We then define an "outlier" edge as such.

Definition 8 (Outlier Edge) *An edge is an outlier within a cluster if its percent over/under expectancy in two clusters who share the same parent has different signs,* and *if the absolute value of the percent over/under expectancy is over some threshold value t, which is set to 0 if not defined by the user.*

This definition aims to identify edges whose presence greatly affects the ATE.

Chapter 4

Evaluation

We conducted our experiment on two logs. The first, which we will center our evaluation around, is a synthetic log in which we know the ground truth causal graph. This experiment demonstrates how our system was able to outperform other methods in accuracy while achieving a very realistic runtime complexity. The method in which we evaluate the suggestions our system makes is also further described in Section 4.1.2.

The second is a real life dataset on flight delays and cancellations from 2015 as collected by the U.S. Department of Transportation [22]. While there is no ground truth causal graph that we can compare to, this experiment demonstrates the insights into real life systems that we can discover using our proposed work. It also serves as an end-to-end demonstration of our system's functionality, as well as how it integrates with Sawmill.

4.1 Synthetic Log

4.1.1 Synthetic Log Creation

For our synthetic log, we have two main goals:

1. Simulate the complexity challenge posed by having a log with many variables.
2. Have knowledge of a ground truth DAG so that we can evaluate results.

To achieve goal #1, we decided to have a log with 100 variables, called `a0` to `a99`. For goal #2, we defined a base causal graph around which we will build the rest of the graph. The base graph following closely with the XYZ example as described in Section 1.5, and consists of nodes `a0, a1, a2` where there is a causal relationship between `a1` and `a2`, but `a0` is also a confounder, as shown in Figure 4.1.

Next, we fill in the rest of the graph randomly by considering each edge in a random order, and adding it with 0.03% probability. This probability was chosen to create a relatively sparse, but still interesting graph. Each edge is also assigned a random integer weight between 1 and 10, inclusive. These weights represent the true ATE of the edge. The full graph is shown in Figure 4.2, with the base graph described above denoted in red.

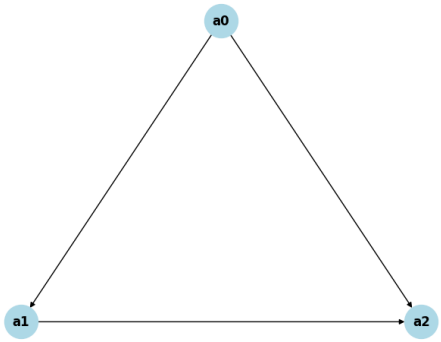


Figure 4.1: Base graph for synthetic log

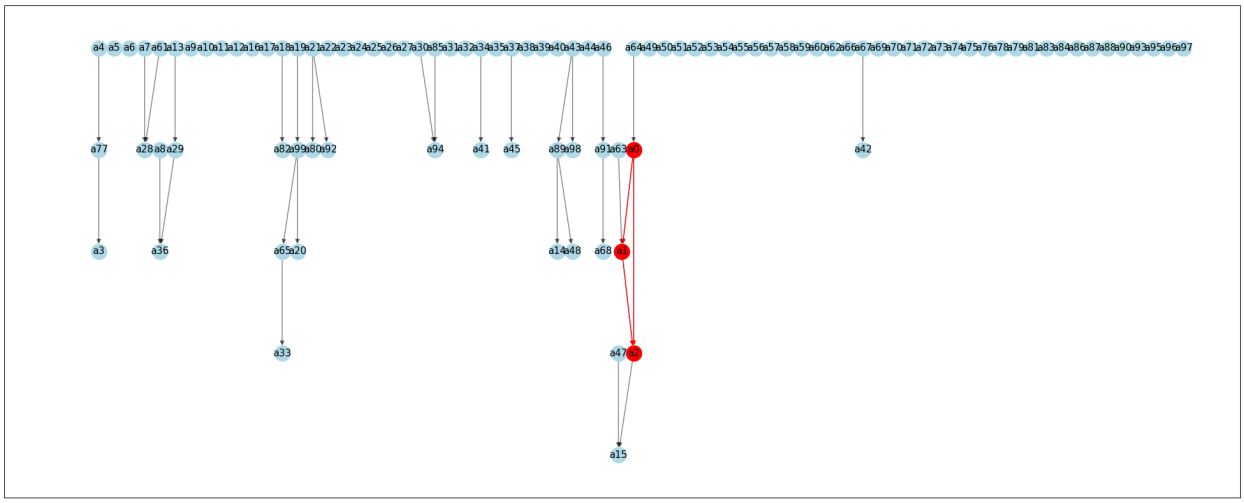


Figure 4.2: Full graph for synthetic log

Finally, we generate the log in similar fashion to the XYZ example, using the edge weights as coefficients and injecting some noise. For those variables with no incoming edges, the value is a randomly generated integer between 1 and 1000, inclusive, at each time-step. We generated the log for 25k time-steps, and a short snippet of the produced log can be seen in Figure 4.3.

4.1.2 Evaluation Methods

In order to evaluate the results of our system, we will define an accuracy scoring function for the edges identified. For this, we make the following key assumption:

Assumption 3 (User Knowledge) *We assume that the user is able responds perfectly to the system suggestions, and only accepts an edge when the edge exists in the ground truth causal graph, and rejects it otherwise. Notably, this assumes that the user can differentiate between direct and indirect causal relationships.*

Then, we define evaluate our system's suggestions as such:

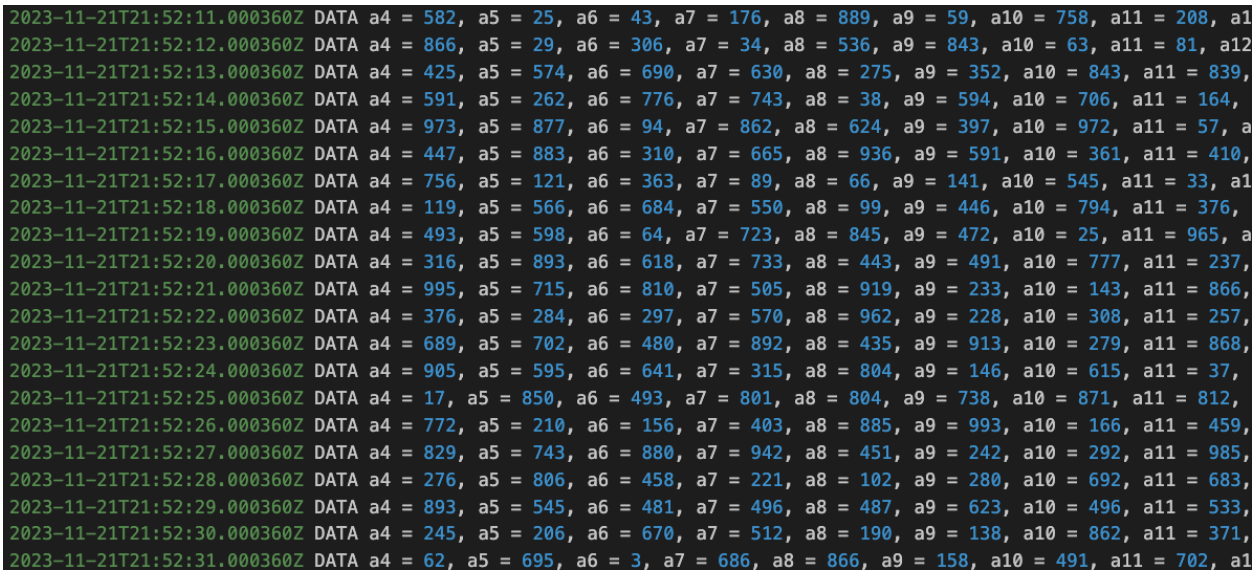


Figure 4.3: Snippet of synthetic log

Definition 9 (Accuracy Scoring Function) *We define G to be the graph consisting of all backdoor paths of T and Y as identified by the backdoor criterion in the ground truth causal graph. We then define G' to be the graph constructed by the user with our suggestions and according to Assumption 3. Then, we have*

$$score = \frac{len(edges_{G'})}{len(edges_G)}$$

In order to evaluate our system, we will examine how performance decreases with the number of variables considered, how accuracy improves with the number of variables considered, and how our model's performance and accuracy compares to other methods.

4.1.3 Results

Performance and Accuracy with Regards to Number of Variables

As described in Section 3.2, we rank the variables by their impact on ATE and take the top n. The goal of this step was to reduce the number of DAGs possible in order to reduce the runtime of the system. However, the less number of variables we keep, the more likely that we may prune variables that are actually important in the causal graph. As such, we measured how the number of variables we keep impacts both performance and accuracy.

The results are shown in Figure 4.4, where n is the number of variables we keep from the pruning process. As expected, performance degrades exponentially with n, while accuracy increases linearly before plateauing out after $n = 3$. While it is true that the value n at which accuracy plateaus is largely dependent on the dataset and the ground truth causal graph, Assumption 2 implies that this number is limited. Additionally, if a user has a certain budget, whether it be time or number of variables considered, then n is easily determined. The user can then be assured that increasing the budget can only increase accuracy.

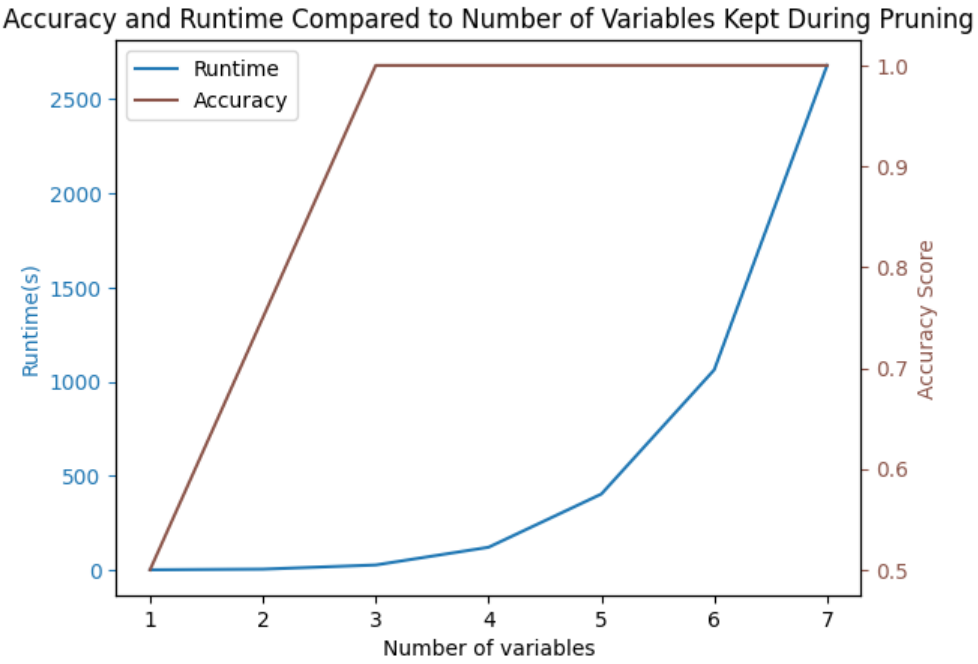


Figure 4.4: How runtime and accuracy scales with number of variables kept from the pruning process

Model Performance & Accuracy vs Other Methods

In this section, we will call our method `ate_prune`. We also compared our algorithm's performance and accuracy to two other methods:

- Lasso: Use a MultiTaskLasso object to measure the impact of each variable on the outcome variable Y. We will then select the top n variables.
- Random: Use a random selector to select n variables.
- Exhaustive: Use all variables without any pruning method.

For each of these, we will then follow the same methodology as described in Section 3 starting from DAG enumeration. Measuring each method's performance ($\frac{1}{runtime}$) and accuracy yields Figure 4.5. Note that for the random selector, we ran the experiment 10 times and took the averages of both runtimes and accuracies to ensure that an extremely lucky or unlucky run did not affect the results. Additionally, we selected 3 variables for the lasso, random, and `ate_prune` so that the accuracy results would be comparable. Also note that it was impossible to actually run the exhaustive search due to the computational power it requires, so we assigned it the max accuracy score possible, but set performance to be $1e-10$. As shown in these results, `ate_prune` was able to achieve just as high of an accuracy as the exhaustive search, with a much better performance. While `ate_prune` was slightly slower than lasso and random, it does much better. Lasso was able to achieve 25% accuracy, while over 10 runs, random only achieved 2.5%, likely from a lucky run.

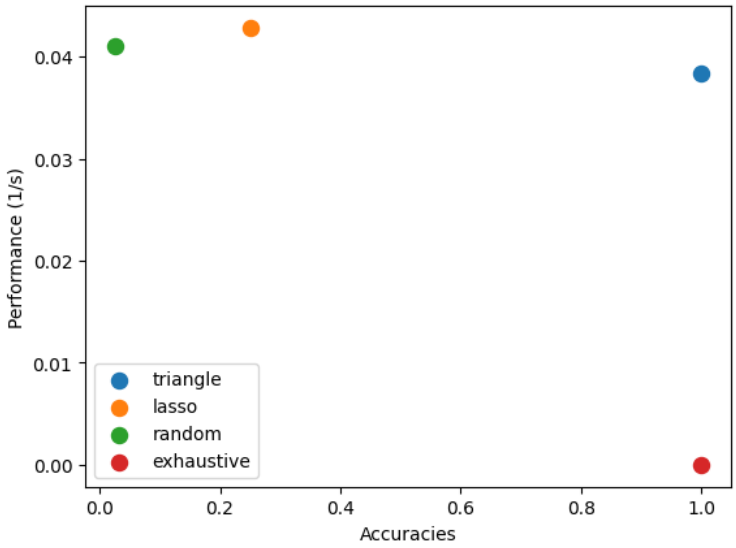


Figure 4.5: Comparison of performance and accuracy between our model, lasso, random, and exhaustive

4.1.4 End-to-End Demo

Here's a demo of how we can interact with the system to fully construct the edge of interest to us in the synthetic log. As a reminder the ground truth DAG is shown in Figure 4.2. The sub-graph that is connected to and thus may influence the ATE calculation is shown below in Figure 4.6 and we are interested in the question: What is the ATE of a1 on a2? As can be seen, the answer should be 3, but there are many other variables affecting a2 as well.

Variable Pruning Results

After we run the variable pruning algorithm as described in Section 3.2.2, we get a ranking of the variables, with the top 10 shown in Figure 4.1. We can see that our variables of interest in the ground truth causal DAG which are upstream from a1 and a2 all appear in the top 4 variables. Additionally, we can see that the impact significantly drops off after a15, correctly finding that the nodes below a15 have very little impact on our ATE calculation.

Results with $n = 3$

As seen from Section 4.1.3 and from the results of pruning, choosing the top 3 variables yields the best results in terms of accuracy and runtime. Running our algorithm with n=3 then correctly identifies the following assumptions for an ATE of 3.0:

- (a0, a1) EXISTS
- (a64, a1) EXISTS
- (a0, a2) EXISTS

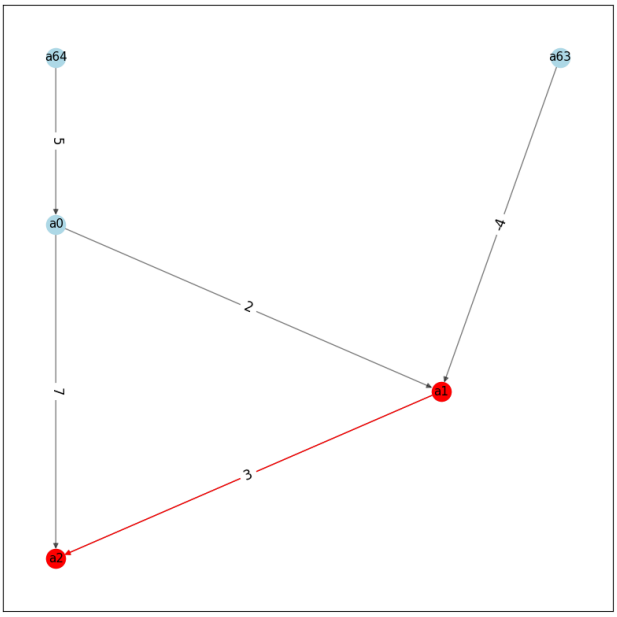


Figure 4.6: Sub-graph consisting of backdoor paths

And as expected, it identified that these edges did **not** exist for the DAGs clustered at ATE= 5.96. The full results can be seen at Figure 4.7, with the first two tables displaying outlier edges identified with each ATE. Figure 4.7c contains histogram shows the distribution of ATE values found by analyzing all of the DAGs we found, as well as how our clustering algorithm identified 3.0 and 5.96 as the two main possible ATEs. Finally, Figure 4.7d shows the specific frequencies of each edge as they appear in the DAGs for each cluster. As shown, we can see that if we asked the system for a fourth outlier edge, it also would have found the edge from a64 to a2, which does represent a true causal relationship that the user would find helpful to consider.

4.2 Flights Log

4.2.1 Log-ifying Flights Dataset

The raw dataset is a CSV file, as shown in Figure 4.8. As such, we first convert the data into a log form, as one might expect if a computer was writing out the data to a readable format as flights occurred. While this is not necessary and it would actually be easier to start with the CSV file, converting it to a log shows our system's end-to-end capability to process logs. A snippet of the log is shown in Figure 4.9.

4.2.2 Sawmill

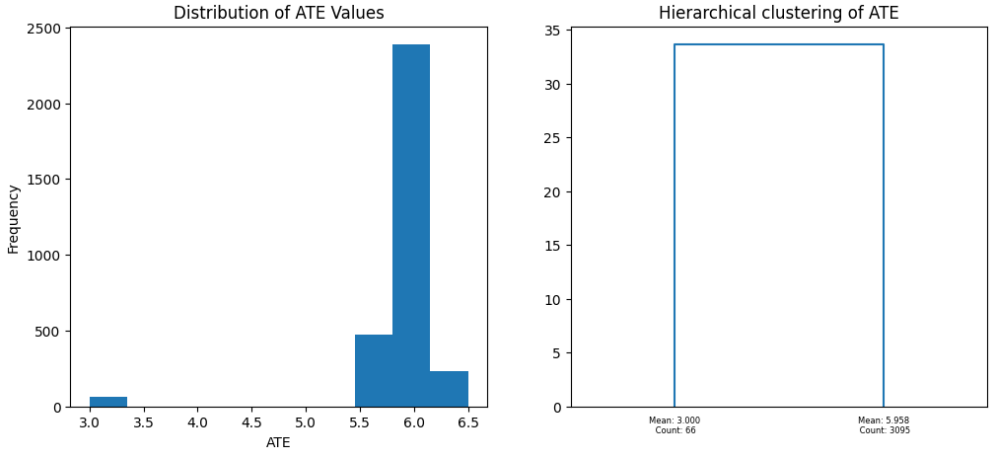
Now, we set up our Sawmill instance, asking it to parse the log we just created. In addition to the default regex dictionary Drain uses to parse, we give it custom regex definitions for the flight id: r"(?<=Flight\s)\d+", as well as airline code: r"(?<=airline\s)\w+". Other

Edge	% Expectancy	Status
(a0, a1)	4.02	EXISTS
(a64, a1)	3.71	EXISTS
(a0, a2)	3.60	EXISTS
(a1, a0)	-0.86	DOES NOT EXIST
(a1, a64)	-0.86	DOES NOT EXIST
(a2, a0)	-0.84	DOES NOT EXIST

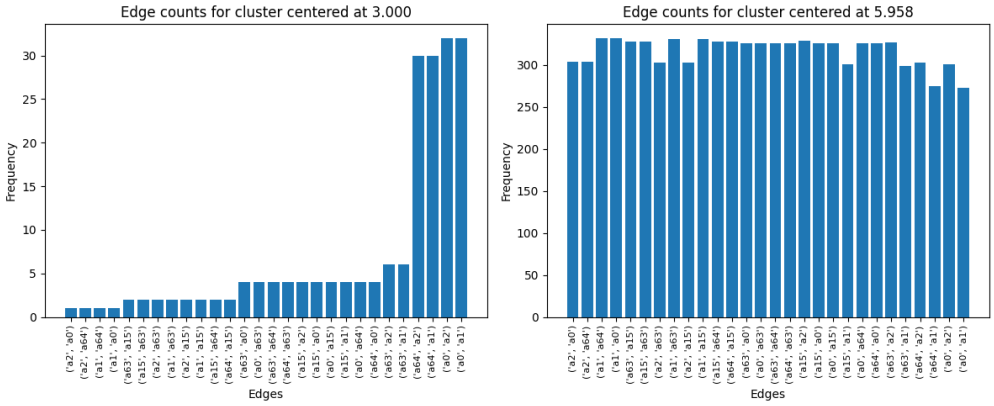
(a) Outlier edges identified in synthetic log experiment for cluster centered at ATE=3.00

Edge	% Expectancy	Status
(a1, a0)	0.018	EXISTS
(a1, 64)	0.018	EXISTS
(a2, a0)	0.018	EXISTS
(a0, a1)	-0.086	DOES NOT EXIST
(a64, a1)	-0.079	DOES NOT EXIST
(a0, a2)	-0.077	DOES NOT EXIST

(b) Outlier edges identified in synthetic log experiment for cluster centered at ATE=5.96



(c) Hierarchical clustering of synthetic results



(d) Edge frequency for each cluster for synthetic results

Figure 4.7: Results for end-to-end synthetic demo

	AIRLINE	FLIGHT_NUMBER	ORIGIN_AIRPORT	DESTINATION_AIRPORT	ARRIVAL_DELAY	DEPARTURE_DELAY
0	AS	98	ANC	SEA	-22.0	-11.0
1	AA	2336	LAX	PBI	-9.0	-8.0
2	US	840	SFO	CLT	5.0	-2.0
3	AA	258	LAX	MIA	-9.0	-5.0
4	AS	135	SEA	ANC	-21.0	-1.0
5	DL	806	SFO	MSP	8.0	-5.0
6	NK	612	LAS	MSP	-17.0	-6.0
7	US	2013	LAX	CLT	-10.0	14.0
8	AA	1112	SFO	DFW	-13.0	-11.0
9	DL	1173	LAS	ATL	-15.0	3.0

Figure 4.8: Raw flights dataset

```
2015-01-01T00:05:00.000000Z id_0000000 Flight 98 by airline AS departed from ANC with a delay of -11.0 minutes
2015-01-01T04:30:00.000000Z id_0000000 Flight 98 by airline AS reached SEA with a delay of -22.0 minutes
2015-01-01T00:10:00.000000Z id_0000001 Flight 2336 by airline AA departed from LAX with a delay of -8.0 minutes
2015-01-01T07:50:00.000000Z id_0000001 Flight 2336 by airline AA reached PBI with a delay of -9.0 minutes
2015-01-01T00:20:00.000000Z id_0000002 Flight 840 by airline US departed from SFO with a delay of -2.0 minutes
2015-01-01T08:06:00.000000Z id_0000002 Flight 840 by airline US reached CLT with a delay of 5.0 minutes
2015-01-01T00:20:00.000000Z id_0000003 Flight 258 by airline AA departed from LAX with a delay of -5.0 minutes
2015-01-01T08:05:00.000000Z id_0000003 Flight 258 by airline AA reached MIA with a delay of -9.0 minutes
2015-01-01T00:25:00.000000Z id_0000004 Flight 135 by airline AS departed from SEA with a delay of -1.0 minutes
2015-01-01T03:20:00.000000Z id_0000004 Flight 135 by airline AS reached ANC with a delay of -21.0 minutes
2015-01-01T00:25:00.000000Z id_0000005 Flight 806 by airline DL departed from SFO with a delay of -5.0 minutes
2015-01-01T06:02:00.000000Z id_0000005 Flight 806 by airline DL reached MSP with a delay of 8.0 minutes
2015-01-01T00:25:00.000000Z id_0000006 Flight 612 by airline NK departed from LAS with a delay of -6.0 minutes
2015-01-01T05:26:00.000000Z id_0000006 Flight 612 by airline NK reached MSP with a delay of -17.0 minutes
2015-01-01T00:30:00.000000Z id_0000007 Flight 2013 by airline US departed from LAX with a delay of 14.0 minutes
2015-01-01T08:03:00.000000Z id_0000007 Flight 2013 by airline US reached CLT with a delay of -10.0 minutes
2015-01-01T00:30:00.000000Z id_0000008 Flight 1112 by airline AA departed from SFO with a delay of -11.0 minutes
2015-01-01T05:45:00.000000Z id_0000008 Flight 1112 by airline AA reached DFW with a delay of -13.0 minutes
2015-01-01T00:30:00.000000Z id_0000009 Flight 1173 by airline DL departed from LAS with a delay of 3.0 minutes
2015-01-01T07:11:00.000000Z id_0000009 Flight 1173 by airline DL reached ATL with a delay of -15.0 minutes
```

Figure 4.9: Log-ified flights dataset

Variable	Max ATE Diff
a0	3.01
a64	3.01
a63	0.49
a15	0.01
a51	0.00
a7	0.00
a42	0.00
a26	0.00
a74	0.00
a77	0.00

Table 4.1: Top 10 variables for synthetic log

	Name	Occurrences	Preceding 3 tokens	Examples	Type	Tag
0	Timestamp	200000	[]	[2015-01-01T00:05:00.000000Z, 2015-01-01T04:30:00.000000Z, 2015-01-01T00:10:00.000000Z, 2015-01-01T07:50:00.000000Z, 2015-01-01T00:20:00.000000Z]	date	Timestamp
1	id	200000	[]	[id_0000000, id_0000001, id_0000002, id_0000003, id_0000004]	str	id
2	flight	200000	[]	[98, 2336, 840, 258, 135]	num	flight
3	airline	200000	[]	[AS, AA, US, DL, NK]	str	airline
4	90a25666_9	100000	[<*3>, departed, from]	[ANC, LAX, SFO, SEA, LAS]	str	90a25666_9
5	90a25666_14	100000	[a, delay, of]	[-11.0, -8.0, -2.0, -5.0, -1.0]	num	90a25666_14
6	cbef448f_8	100000	[airline, <*3>, reached]	[SEA, PBI, CLT, MIA, ANC]	str	cbef448f_8
7	cbef448f_13	100000	[a, delay, of]	[-22.0, -9.0, 5.0, -21.0, 8.0]	num	cbef448f_13

Figure 4.10: Variables parsed from the flights log

variables, such as departure airport, arrival airport, departure delay, and arrival delay, have been given hashed names, so we further tag the variables for better readability. The resulting parsed variables are shown in Figure 4.10.

Finally, we set the causal unit to be "id", and perform the necessary aggregations in order to get the prepared variables.

4.2.3 What is the ATE of Delta Airlines on Arrival Delay?

In an experiment from the Sawmill paper, we discovered that the most likely causal factor of arrival delay is the airline being Delta, with an ATE of -14. This implies that on average, flying on Delta will result in an early arrival by 14 minutes. We wish to use our system to tell the user if there are any major assumptions being made in producing this answer. Thus, treatment T is `airline+latest=DL` and outcome Y is `arrdelay+mean`.

Pruning Variables

With 645 variables, our system would find $O(3^{645^2}) = 10^{198494}$ number of potential graphs to consider. This would take a tremendous amount of computational power to process,

but we are successfully able to prune the variables as described in Section 3.2. Pruning aggregates resulted in eliminating `flight+min` and `flight+max`, using just `flight+mean`. It also kept just `depdelay+mean` for the `depdelay` aggregates. Next, pruning via ATE impact as described in Section 3.2.2 ranked the variables, and we took the top 6. As can be seen in Figure 4.12, the ATE impact decreases rapidly, and 6 variables is more than enough. Now, with these 6 variables, `airline+latest=DL`, `arrdelay+mean`, we are down to **11523** $\approx 10^4$ potential DAGs. This is 10^{198490} times less the number of DAGs as we would have had to consider with all the variables!

Variable	Max ATE Diff
depdelay+mean	9.88
depairport+latest=ATL	1.29
airline+latest=MQ	1.19
depairport+latest=ORD	0.79
airline+latest=US	0.72
flight+mean	0.71
arrairport+latest=ATL	0.67
airline+latest=WN	0.61
arrairport+latest=ORD	0.58
airline+latest=AS	0.52
arrairport+latest=DFW	0.51
depairport+latest=DEN	0.46
arrairport+latest=EWR	0.43
depairport+latest=DFW	0.43
airline+latest=AA	0.34

Table 4.2: Variables ranked by impact on ATE as described in Section 3.3

Clustering

After the $ATE(T, Y, G)$ is calculated for each DAG and we cluster the results according to Section 3.4, the system visualizes the results via a histogram and a dendrogram as shown in Figure 4.11.

Key Edge Detection

Finally, we identify key edges by following the method as described in Section 3.5. The outlier edges for each cluster is displayed as shown in Figures 4.12a and 4.12b, while the specific edge counts are shown in a bar chart so the user can manually identify if any edges were overlooked, as shown in 4.13.

Results

Thus, we can see that the system identifies two major possible answers for our question: "What is the ATE of Delta Airlines on Arrival Delay?". The original answer, -14, appears

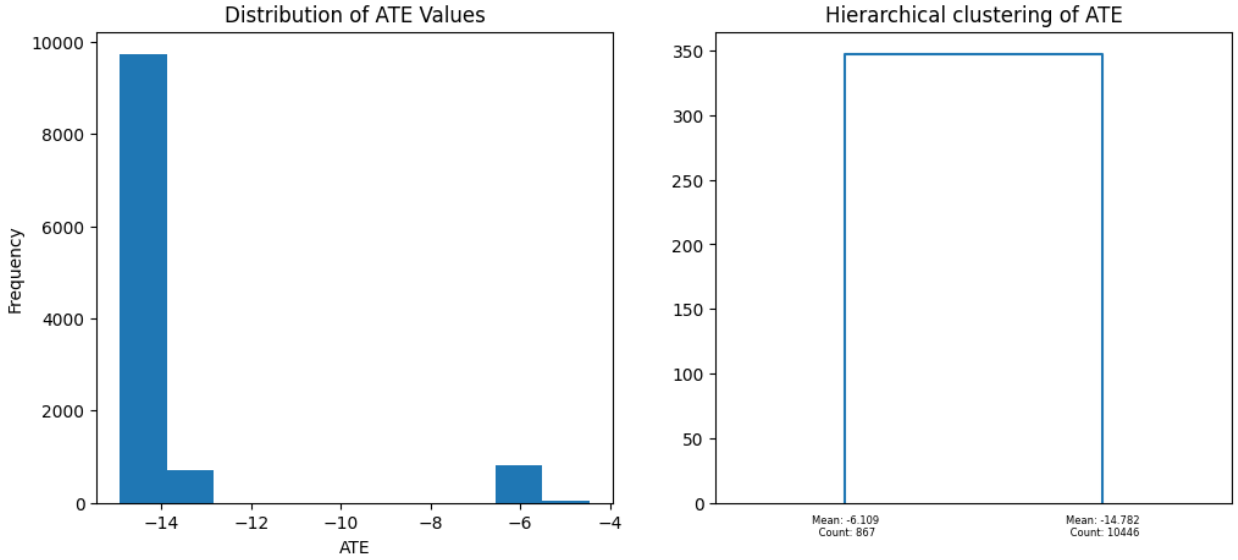


Figure 4.11: Distribution and clustering of ATE(`airline+latest=DL`, `arrdelay+mean`).

Edge	% Expectancy	Status
(depdelay+mean, arrdelay+mean)	11.34	EXISTS

(a) Outlier edges identified in flights experiment for cluster centered at ATE=-6.11

Edge	% Expectancy	Status
(depdelay+mean, arrdelay+mean)	-0.94	DOES NOT EXIST

(b) Outlier edges identified in flights experiment for cluster centered at -14.78

Figure 4.12: Outlier edges for flights experiment

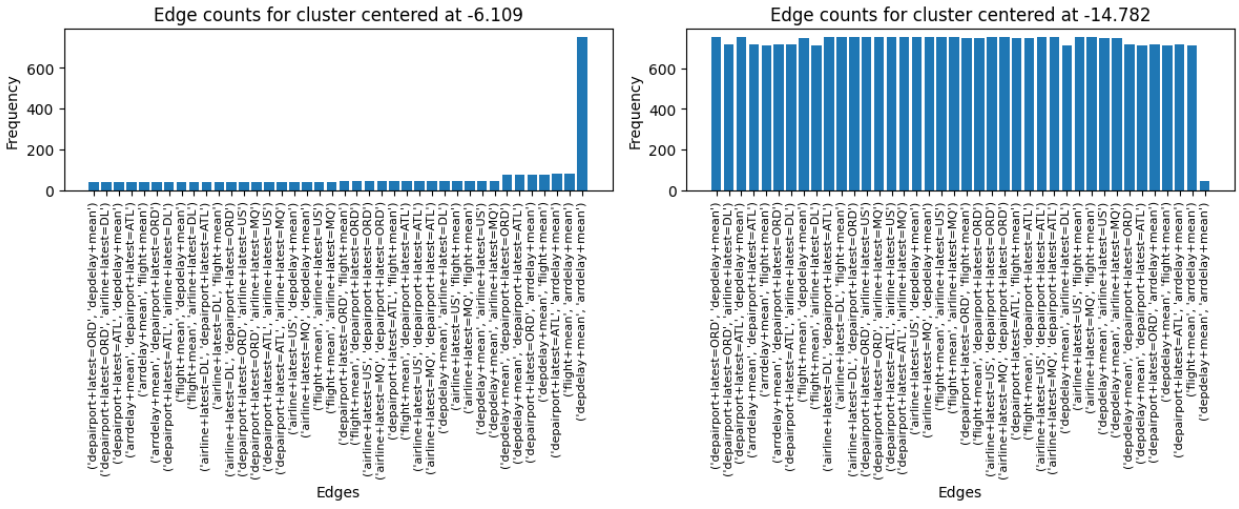


Figure 4.13: Edge counts for each cluster in flights experiment

to be true in 10446 out of the 11313 scenarios we test. However, in the 867 cases where the ATE is actually closer to -6, the edge that is most highly represented is `(depdelay+mean, arrdelay+mean)`. This means that the user should highly consider whether or not `depdelay+mean` has a causal effect on `arrdelay+mean` or not. A common-sense user may recognize that a flight's departure delay does in fact impact the arrival delay. With this context in mind, we would conclude that that flying Delta does **not** in fact result in arriving 14 minutes early, but only 6. Upon further investigation, Sawmill actually reveals that `airline+latest=DL` has an ATE of -9 on `depdelay+mean`. Given these results, a user who would have switched to Delta for the 14 minute speedup may actually choose not to, since they do not want wish to risk missing the flight if it departs 9 minutes early!

Chapter 5

Future Work

In this work, we extended the existing system Sawmill to make it more robust in answering causal questions. Specifically, we are able to provide key edges that significantly impact the answer to: "What is the ATE of T on Y?". These key edges represent assumptions that the user should carefully consider and accept/reject. We do so by choosing variables that have the highest impact on ATE themselves, then enumerating all possible causal graphs encompassing those variables. We then cluster the resulting ATEs and their corresponding DAGs, and identify key edges that differentiate DAGs in different clusters.

We showed how we successfully decreased the performance complexity without compromising on accuracy through our method of pruning variables, and how this is an improvement from other methods. There remains, however, a couple of interesting directions that future work can explore.

5.1 Accuracy Scoring

While the accuracy scoring scheme described in Definition 9 captures the merit of the edges our system suggests, there are graph similarity algorithms that we can leverage which may provide a better metric. This is well studied in the field of spectral theory [23]–[25], calculating graph distance via edit distance and various matrices. For each of these, we assume that the user created graph G' is constructed by the user accepting/rejecting edge suggestions to the best of their ability. Namely:

- User accepts edge (u, v) if it exists in the ground truth causal graph
- User accepts edge (u, v) with probability 0.5^{n-1} if a path from u to v exists in the ground truth causal graph, where n is the length of the shortest path
- User rejects edge (u, v) in all other scenarios

The true graph G that we are comparing against consists of all edges on backdoor paths as determined by the backdoor criterion in the ground truth causal graph. Since both G and G' will be limited in size, there is a less restrictive upper bound on the complexity of the algorithms we can use to calculate graph similarity.

5.1.1 Eigenvalue Distribution

One idea is to compare the similarity of two graph's eigenvalue distributions in order to measure how similar the graphs themselves are, as presented by Crawford et al [23]. Specifically, given graphs G and G', we would calculate the eigenvalues of the **Laplacian** matrix and perform the Kolmogorov-Smirnov (KS) Test:

$$KSTest(\Lambda_{L(G)}, \Lambda_{L(G')})$$

The study chose to use the Laplacian matrix eigenvalues since Fan Chung's work [26] suggests that this distribution is more closely linked to the structure of the graph than the adjacency matrix eigenvalues. Then, the KS test's p-value, which is calculated for the null hypothesis $\Lambda_{L(G)} = \Lambda_{L(G')}$ describes how similar the two graphs' structures are. While this was applied to graphs representing networks, the fact that eigenvalues representing the algebraic connectivity of a graph [27] suggests that it can still be of use to us.

5.1.2 Graph Edit Distance (GED)

As described by Gao et al [24], a graph can be transformed to another one by a finite sequence of graph edits, and the GED is defined as the least-cost edit operation sequence. The paper defines multiple algorithms [28], [29] for non-attributed graphs like ours, which usually include converting the graph to strings and computing the edit distance for those strings [30]–[32].

5.2 Pruning Edges

In this work, we focused on decreasing the number of graphs we need to consider by pruning variables. If V is the number of nodes, the number of graphs is $O(3^{V^2})$, so each variable we are able to prune results in an exponential decrease in complexity. However, it is possible that pruning edges can be more efficient. If E is the number of edges, the number of graphs is $O(3^E)$. While each edge that is pruned results in less of a decrease in complexity than with variables, $ATE(T, Y, G)$ does not consider any part of the graph that is not connected to T or Y. For example, if we had a graph as shown in Figure 5.1, the existence of edge (A, B) has no impact on the ATE calculation. Additionally, consider the graph shown in Figure 5.2. With our current method, we would find and keep variable A. Then in the DAG enumeration phase, we would still consider the edge (A, T). However, if we were able to prune edges, we would discard (A, T) and only keep (A, Y), resulting less graphs to consider.

The challenge remains as described in Section 3.1.1, which is that an edge may be important even if on its own, it doesn't greatly impact the ATE. Further work could explore how to work around this challenge, such as exploring longer paths or certain edges in aggregate.

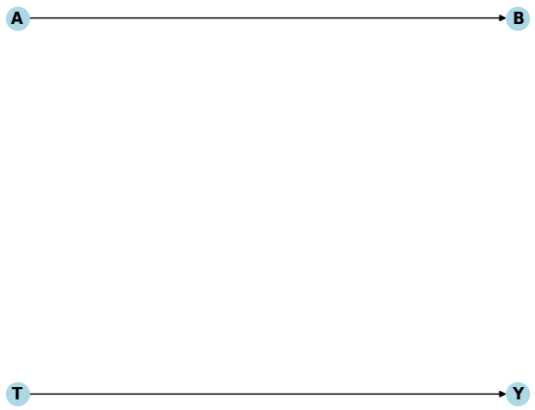


Figure 5.1: Example graph in which edge (A, B) does not matter for $ATE(T, Y, G)$

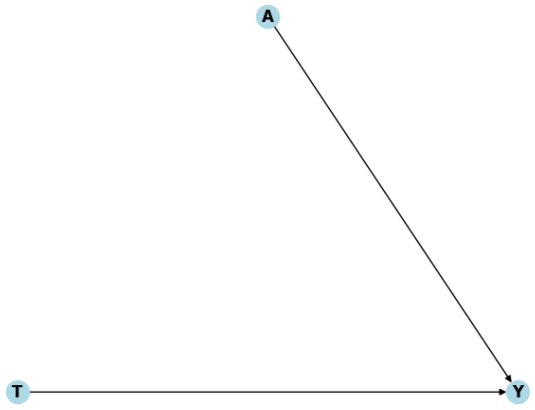
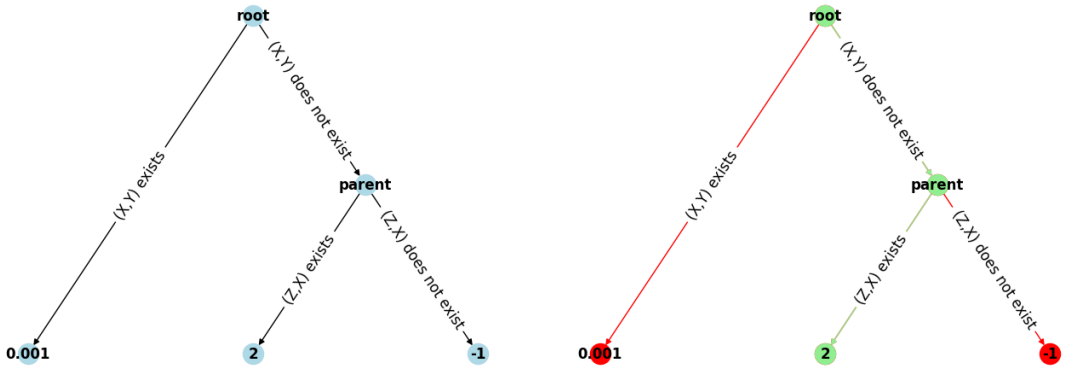


Figure 5.2: Example graph in which pruning variables we keep A, but pruning edges would only keep edge (A, Y)



(a) UI example for displaying clusters and key edges

(b) UI example after user accepts/rejects some edges

Figure 5.3: Example graphs for potential user interface

5.3 A Better User Interface

While we were able to build up the algorithms to identify key edges and present them to the user, there is work to be done in integrating this functionality in with Sawmill, and allowing users to proceed how to choose in a more intuitive way. One idea is to present the hierarchical clustering as a tree, with the key edges identified annotated on the edges between clusters, as shown in Figure 5.3a. Here, each leaf represents a cluster, with the average ATE within that cluster displayed as the node name. The system could further make each leaf node expandable, allowing the user to examine more fine-grained clusters and the key edges differentiating them if they wanted.

A user could then accept a branch (and thus accepting/rejecting the key edges on it) by interacting with this tree. An updated tree after the user decides to reject edge (X, Y) and accept edge (Z, X) might then look like Figure 5.3b

5.4 Graph Discovery

In this algorithm, we were able to leverage the fact that we are given the treatment T and outcome Y by the user to limit the ATEs we consider. However, we can use a similar line of thinking with impact of variables and edges on ATE to aid a user in discovering the graph when nothing is known to them. One can imagine how this would be helpful to a system manager presented with a large log and simply wants to learn more about the variables and causal relationships in them. We briefly explored a sampling algorithm which may prove useful in futue work on graph discovery.

Algorithm 5.4.1 VariableToLayer: Sampling algorithm to assign variables to layers in the DAG

```
function VariableToLayer(variables, num_layers, num_samples)
    ates ←[[0]*num_layers]*len(variables)
    for k, v in enum(variables) do
        for i in num_layers do
            mean_ate ← 0
            for j in num_samples do
                G ← randomly assign variables\v to layers with random edges
                // Calculate the average impact this variable has on its out-neighbors
                ate ← mean(normalized(ATE(v, out_node, G) for all out_nodes)
                mean_ate += ate
            end for
            mean_ate \= num_samples
            ates[k][i] = mean_ate
        end for
    end for
    best_layers ← {}
    for k, v in enum(variables) do
        best_layers[v] ← idxmax(abs(ates[k]))
    end for
    return best_layers
```

5.4.1 Assign Variables to Layers

The key idea here follows with Assumption 2, in which most ground truth causal graphs are sparse and with limited height. Similarly, we know that most causal graphs are DAGs, although there exists cyclical ones [33]. As such, we can try to assign variables to a certain layer (height) in the DAG. The algorithm shown in 5.4.1 attempts to find, for each variable, the layer in the DAG in which it has the largest impact on other variables. It does so by considering each variable in each layer, and generating many random graphs. It then calculates the average ATE of the fixed variable in the fixed layer on its out-neighboring nodes. Note that this ATE must be normalized somehow, since pure magnitude of ATE does not necessarily reflect the strength of the impact. Consider $x = 100z$ vs $w = 50y$. The first will yield an ATE two times that of the second, but both edges are equally true. Once the sampling process is done for each variable in each layer, the algorithm returns the layer in which the average impact was largest for each variable.

5.4.2 Exploration

Once we know which layer of the DAG each variable likely belongs to, the number of possible causal graphs decreases drastically. From here, we could implement something similar to the algorithm described in Section 3.3 in order to use ATEs and clusters to find likely graphs to present to the user. It is probable that even if the user started with no knowledge of the graph, they will have enough knowledge of the system generating the log in order to

differentiate between nonsensical and realistic causal graphs.

Appendix A

Code listing

```
"""
Gets the more user-friendly tagged name for a Sawmill variable
"""
def get_readable_name(name_to_tag, var):
    readable = name_to_tag.get(var.split('+')[0], var.split('+')[0])
    if len(var.split('+'))==2:
        readable += "+" + var.split('+')[1]
    return readable
```

```
"""
Recursively builds a tree by reading the output of
   scipy.cluster.hierarchy.linkage and the leaves of a dendrogram
"""
def build_tree(linked, leaves):
    root = Node()
    curr = root
    if len(leaves) == 1:
        return Node(cluster_id=leaves[0]), -1
    i = len(linked)-1
    while i > -1:
        c1, c2, _, _ = linked[i]
        if c1 not in leaves and c2 not in leaves:
            curr.left, i = build_tree(linked[:i], leaves)
            curr.right, i = build_tree(linked[:i], leaves)
            break
        if c1 in leaves:
            curr.left = Node(leaves.index(c1))
            curr.right = Node()
            curr = curr.right
        if c2 in leaves:
            curr.right = Node(leaves.index(c2))
            break
        i -=1
    root = cleanup_tree(root)
```

```
    return root, i
```

```
"""
Generates a sequence of length n consisting of values {-1, 1, None}
   where there can be at most k non-None values. Used to find all
   possible causal graphs with at most k edges.
"""
def generate_sequences(n, k=3):
    # Initialize a list to store valid sequences
    valid_sequences = []
    stack = [([], 0, 0)]  # (sequence, non_none_count, index)

    while stack:
        sequence, count, index = stack.pop()
        # If the sequence is complete, add it to valid_sequences
        if index == n:
            valid_sequences.append(tuple(sequence))
            continue
        stack.append((sequence + [None], count, index + 1))
        if count < k:
            stack.append((sequence + [-1], count + 1, index + 1))
        if count < k:
            stack.append((sequence + [1], count + 1, index + 1))
    return valid_sequences
```

```
"""
Ranks nodes by impact on ATE(self.treatment, self.outcome) and
   returns the top_n
"""
def prune_triangle(self, nodes, top_n):
    print("Starting to prune using triangle method")
    max_diffs = {}
    base_ate = self.sawmill.get_effect(self.treatment, self.outcome,
       nx.DiGraph([(self.treatment, self.outcome)]), calculate_error=
       False)[0]
    nodes = [node for node in nodes if node != self.treatment and
       node != self.outcome]
    for var in tqdm(nodes, "Processing triangle dags"):
        graphs = []
        graphs.append(nx.DiGraph([(self.treatment, self.outcome), (
           var, self.outcome)]))
        graphs.append(nx.DiGraph([(self.treatment, self.outcome), (
           var, self.treatment), (var, self.outcome)]))
        graphs.append(nx.DiGraph([(self.treatment, self.outcome), (
           self.treatment, var), (var, self.outcome)]))
        graphs.append(nx.DiGraph([(self.treatment, var), (var,
           self.outcome)]))
        ates = [base_ate]
```

```
        for G in graphs:
            ates.append(self.sawmill.get_effect(self.treatment,
                self.outcome, G, calculate_error=False)[0])
        max_diffs[var] = max(ates) - min(ates)
    max_diffs = max_diffs
    df = pd.DataFrame.from_dict(max_diffs, orient='index', columns=['
        max_diff'])
    df = df.sort_values(by='max_diff', ascending=False)
    nodes = list(df.index[:top_n].values)
    print(f'Found {len(nodes)} nodes: {[get_readable_name(
        self.sawmill.name_to_tag, node) for node in nodes]}')
    return nodes
```

```
"""
Hierarchically clusters ates (saved in self.effects) using the ward
    distance and a threshold of one-half the max distance two clusters
    is merged at
"""
def cluster(self, num_clusters=None):
    preprocessed = [val[0] for val in self.effects.values()]
    data = np.array(preprocessed).reshape(-1, 1)
    linked = linkage(data, method='ward')
    if not num_clusters: # try to find optimal
        distances = linked[:,2]
        threshold_multiplier = 0.5
        optimal_clusters = fcluster(linked, t=threshold_multiplier *
            np.max(distances), criterion='distance')
    else:
        optimal_clusters = fcluster(linked, t=num_clusters, criterion
            ='maxclust')
    # change from 1-indexed to 0-indexed
    optimal_clusters = np.array(optimal_clusters - 1)

    # Save cluster mappings
    cluster_mapping = {dag: cluster for dag, cluster in zip(
        self.effects.keys(), optimal_clusters)}
    self.num_clusters = int(np.max(optimal_clusters)+1)
    self.cluster_data = {i: (np.mean(data[optimal_clusters==i]), len(
        data[optimal_clusters==i])) for i in range(self.num_clusters)}
    print(f"Found {max(optimal_clusters)+1} clusters")
    # Create the full dendrogram with a maximum number of clusters

    R = dendrogram(linked, orientation='top', labels=None,
                   show_leaf_counts=False, truncate_mode='lastp', p=
                       self.num_clusters, leaf_rotation=90, no_plot=
                       True)

    label_mapping = {R['leaves'][i]: (np.mean(data[optimal_clusters==
```

```
        i]), len(data[optimal_clusters==i])) for i in range(len(R['
        leaves']))}
    self.tree, _ = build_tree(linked, R['leaves'])
    self.tree.annotate(cluster_mapping)
    return DendrogramRenderer(linked, self.num_clusters,
                              lambda x: "Mean: {:.3f}\n Count: {}"
                                 .format(label_mapping[x][0],
                                 label_mapping[x][1], no_plot=True)
    )
```

```
from collections import defaultdict
import numpy as np

"""
All tree functions
"""
class Node:
    def __init__(self, cluster_id=None):
        self.cluster_id = cluster_id
        self.left = None
        self.right = None

    """
    Assign each DAG to the node it belongs to within the key
    cluster_mapping maps DAGs to cluster id's
    """
    def annotate(self, cluster_mapping):
        self.num_dags = 0
        if self.cluster_id is not None:
            self.dags = [key for key in cluster_mapping.keys() if
                cluster_mapping[key] == self.cluster_id]
            self.num_dags = len(self.dags)
        if self.left:
            self.left.annotate(cluster_mapping)
            self.num_dags += self.left.num_dags
        if self.right:
            self.right.annotate(cluster_mapping)
            self.num_dags += self.right.num_dags

    """
    Recursively count the number of times an edge occurs
    amongst the DAGs assigned to all the children of this
    node, omitting the edge from treatment -> outcome,
    since this always exists
    """
    def count_edges(self, treatment, outcome, dag):
        self.edge_counts = defaultdict(int)
        if self.cluster_id is not None:
```

```
38              for graph in self.dags:
39                  for edge in graph.edges:
40                      if edge != (treatment, outcome) and (not dag or (
                            dag and edge not in dag.edges)):
41                          self.edge_counts[edge] += 1
42
43          if self.left:
44              self.left.count_edges(treatment, outcome, dag)
45              for key in self.left.edge_counts.keys():
46                  self.edge_counts[key] += self.left.edge_counts[key]
47          if self.right:
48              self.right.count_edges(treatment, outcome, dag)
49              for key in self.right.edge_counts.keys():
50                  self.edge_counts[key] += self.right.edge_counts[key]
51
52          freq_counts = list(self.edge_counts.values())
53          if len(freq_counts)==0:
54              self.mean = None
55              self.std_dev = None
56          else:
57              self.mean = np.mean(freq_counts)
58              self.std_dev = np.std(freq_counts)
59
60      """
61      For each edge at each node, calculate what percent over or under
62      expectancy the edge is at
63      """
64      def count_expectancy(self, totals=None):
65          if totals is None:
66              # at root node, calculate expectancy
67              totals = (self.num_dags, self.edge_counts)
68          total_dag, total_edges = totals
69          self.percent_expectancy = defaultdict(float)
70          for edge in self.edge_counts.keys():
71              expected = self.num_dags/total_dag*total_edges[edge]
72              self.percent_expectancy[edge] = (self.edge_counts[edge] -
                    expected)/expected
73          if self.left:
74              self.left.count_expectancy((self.num_dags,
                   self.edge_counts))
75          if self.right:
76              self.right.count_expectancy((self.num_dags,
                   self.edge_counts))
77
78      """
79      Define an outlier as an edge that is below expectancy
80      on one side of the tree, and above on the other side, and
```

```
    optionally, over some threshold on both sides
    """
    def find_outliers(self, threshold=0):
        if self.left and self.right:
            self.left.outliers = {}
            self.right.outliers = {}
            edges = set(self.left.edge_counts.keys()).union(set(
               self.right.edge_counts.keys()))
            for edge in edges:
                if np.sign(self.left.percent_expectancy[edge]) !=
                   np.sign(self.right.percent_expectancy[edge]) and
                   abs(self.left.percent_expectancy[edge]) >
                   threshold and abs(self.right.percent_expectancy[
                   edge]) > threshold:
                    self.left.outliers[edge] =
                       self.left.percent_expectancy[edge]
                    self.right.outliers[edge] =
                       self.right.percent_expectancy[edge]
        if self.left:
            self.left.find_outliers(threshold)
        if self.right:
            self.right.find_outliers(threshold)
```

```
"""
Accuracy score function, comparing edges saved in self.outliers() to
   edges in graph as ground-truth DAG to score against
"""
def score(self, graph, display_df=True):
    edges = set()
    for _, outliers in self.outliers.items():
        edges = edges.union(set(outliers.keys()))
    scores ={}
    for edge in edges:
        edge = (get_readable_name(self.sawmill.name_to_tag, edge[0]),
            get_readable_name(self.sawmill.name_to_tag, edge[1]))
        if edge in graph.edges:
            scores[edge] = 1
        elif nx.has_path(graph, edge[0], edge[1]):
            scores[edge] = 0.5
        elif nx.has_path(graph, edge[1], edge[0]):
            scores[edge] = 0.25
    if display_df:
        df = pd.DataFrame.from_dict({'Edge': list(scores.keys()), '
           Score': list(scores.values())}).sort_values(by='Score',
           ascending=False)
        display(df)
    return sum(scores.values())
```

References

[1] B. Speich, N. Schur, D. Gryaznov, *et al.*, "Resource use, costs, and approval times for planning and preparing a randomized clinical trial before and after the implementation of the new swiss human research legislation," *PLOS ONE*, vol. 14, no. 1, pp. 1–16, Jan. 2019. DOI: 10.1371/journal.pone.0210669. [Online]. Available: https://doi.org/10.1371/journal.pone.0210669.

[2] B. Schölkopf, F. Locatello, S. Bauer, N. R. Ke, N. Kalchbrenner, A. Goyal, and Y. Bengio, "Towards causal representation learning," *CoRR*, vol. abs/2102.11107, 2021. arXiv: 2102.11107. [Online]. Available: https://arxiv.org/abs/2102.11107.

[3] H. Guo, S. Yuan, and X. Wu, *Logbert: Log anomaly detection via bert*, 2021. arXiv: 2103.04475 [cs.CR].

[4] V.-H. Le and H. Zhang, *Log-based anomaly detection without log parsing*, 2021. arXiv: 2108.01955 [cs.SE].

[5] Y. Guo, Y. Wen, C. Jiang, Y. Lian, and Y. Wan, *Detecting log anomalies with multi-head attention (lama)*, 2021. arXiv: 2101.02392 [cs.LG].

[6] P. Sedgwick, "Pearson's correlation coefficient," *BMJ*, vol. 345, e4483–e4483, Jul. 2012. DOI: 10.1136/bmj.e4483.

[7] D. B. Rubin and L. John, "Rubin causal model," in *International Encyclopedia of Statistical Science*, M. Lovric, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1263–1265, ISBN: 978-3-642-04898-2. DOI: 10.1007/978-3-642-04898-2_64. [Online]. Available: https://doi.org/10.1007/978-3-642-04898-2_64.

[8] J. Pearl, *Causality: Models, reasoning, and inference*, 1980.

[9] D. B. Rubin, "Estimating causal effects of treatments in randomized and nonrandomized studies.," *Journal of educational Psychology*, vol. 66, no. 5, p. 688, 1974.

[10] J. Pearl, *Causality: Models, Reasoning and Inference*. Cambridge University Press, 2009.

[11] J. Zeng, M. F. Gensheimer, D. L. Rubin, S. Athey, and R. D. Shachter, "Uncovering interpretable potential confounders in electronic medical records," *Nat Commun*, vol. 13, 2022. DOI: 10.1038/s41467-022-28546-8. [Online]. Available: https://www.nature.com/articles/s41467-022-28546-8.

[12] J. Pearl, "Graphs, causality, and structural equation models," *Sociological Methods & Research*, vol. 22, no. 3, pp. 364–404, 1993.

[13] J. Pearl, *Causality: Models, Reasoning, and Inference*, 2nd. Cambridge University Press, 2000.

[14] P. Spirtes, C. Glymour, and R. Scheines, *Causation, Prediction, and Search*. MIT Press, 2000.

[15] M. A. Hernán and J. M. Robins, *Causal Inference: What If*. Chapman and Hall/CRC, 2020.

[16] A. Sharma and E. Kiciman, *Dowhy: An end-to-end library for causal inference*, 2020. arXiv: 2011.04216 [stat.ME].

[17] J. Pearl, *Causal inference in statistics*, 2016.

[18] D. Lang, "Using sec," *USENIX ;login: Magazine*, vol. 38, 2013. [Online]. Available: https://www.usenix.org/system/files/login/articles/09_lang-online.pdf.

[19] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An Online Log Parsing Approach with Fixed Depth Tree," in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 33–40. DOI: 10.1109/ICWS.2017.13.

[20] D. B. Rubin, "Discussion of 'randomization analysis of experimental data in the fisher randomization test'by basu.," *Journal of the American Statistical Association*, vol. 75, no. 371, pp. 591–93, 1980.

[21] P. Spirtes and C. Glymour, "An algorithm for fast recovery of sparse causal graphs," *Social Science Computer Review*, vol. 9, no. 1, pp. 62–72, 1991. DOI: 10.1177/089443939100900106. eprint: https://doi.org/10.1177/089443939100900106. [Online]. Available: https://doi.org/10.1177/089443939100900106.

[22] D. of Transportation. "2015 flight delays and cancellations," Department of Transportation. (2016), [Online]. Available: https://www.kaggle.com/datasets/usdot/flight-delays?select=flights.csv.

[23] B. Crawford, R. Gera, J. House, T. Knuth, and R. Miller, "Graph structure similarity using spectral graph theory," in *Complex Networks & Their Applications V*, H. Cherifi, S. Gaito, W. Quattrociocchi, and A. Sala, Eds., Cham: Springer International Publishing, 2017, pp. 209–221, ISBN: 978-3-319-50901-3.

[24] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Analysis and Applications*, vol. 13, pp. 113–129, 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:11336685.

[25] T. Gervens and M. Grohe, *Graph similarity based on matrix norms*, 2022. arXiv: 2207.00090 [cs.DM].

[26] F. R. K. Chung, *Spectral Graph Theory*. American Mathematical Society, 1997, vol. 92.

[27] P. Frankl and V. Rödl, "Forbidden intersections," *Transactions of the American Mathematical Society*, vol. 300, no. 1, pp. 259–286, 1987.

[28] A. Robles-Kelly and E. R. Hancock, "Graph edit distance from spectral seriation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 3, pp. 365–378, 2005.

[29] A. Robles-Kelly and E. R. Hancock, "String edit distance, random walks, and graph matching," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 18, no. 3, pp. 315–327, 2004.

[30] H. Yu and E. R. Hancock, "String kernels for matching seriated graphs," in *Proceedings of IEEE International Conference on Pattern Recognition*, Hong Kong, 2006, pp. 224–228.

[31] A. Robles-Kelly and E. R. Hancock, "Edit distance from graph spectra," in *Proceedings of IEEE International Conference on Computer Vision*, Nice, 2003, pp. 234–241.

[32] H. Qiu and E. R. Hancock, "Graph matching and clustering using spectral partitions," *Pattern Recognition*, vol. 39, no. 1, pp. 22–34, 2006.

[33] G. Lacerda, P. Spirtes, J. D. Ramsey, and P. O. Hoyer, "Discovering cyclic causal models by independent components analysis," *CoRR*, vol. abs/1206.3273, 2012. arXiv: 1206.3273. [Online]. Available: http://arxiv.org/abs/1206.3273.