SAB: Scalable Analysis of Bayesian Networks

Osayamen Jonathan Aimuyo
oja7@cornell.edu
Cornell University
Ithaca, NY, USA

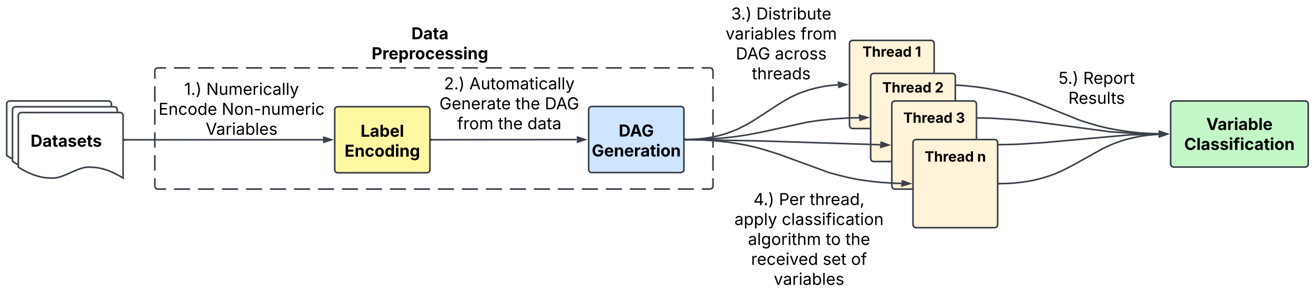
Datasets
Data Preprocessing
1.) Numerically Encode Non-numeric Variables
Label Encoding
2.) Automatically Generate the DAG from the data
DAG Generation
3.) Distribute variables from DAG across threads
Thread 1
Thread 2
Thread 3
Thread n
4.) Per thread, apply classification algorithm to the received set of variables
5.) Report Results
Variable Classification

Figure 1: *SAB Processing Pipeline.* Note NOTEARS is due to Zheng et al. [32]

Abstract

In this work, we explore automated and scalable analysis of bayesian networks. We consider this work to be a blend of an application and algorithmic project. Specifically, we propose a parallel algorithm and implement it within a data processing pipeline to target the problem of *automatically tagging nodes* as members of the common causal topologies: collider, chain, and Fork. We motivate this work, contextualize with existing software libraries, and show evaluation results benchmarking our implementation on causal graphs generated from real-world datasets. We provide an instructive, self-contained Jupyter notebook at https://github.com/osayamenja/sab.

Keywords

Bayesian Networks, Causal Inference, Information Retrieval

ACM Reference Format:
Osayamen Jonathan Aimuyo. 2025. SAB: Scalable Analysis of Bayesian Networks. In *Proceedings of (CS 6386: Data to Decisions)*. ACM, New York, NY, USA, 9 pages.

1 Introduction

Computational processing of Bayesian networks is a challenging task to scale to massive datasets [29]. On the other hand, causal analysis is still a manual endeavor today that requires expert-in-the-loop intervention to ensure the validity and reliability of automatically generated causal graphs [1]. For example, we show in Figures 3-5, the three *primitive* topologies, chain, collider and Fork, that are easily identifiable by manual inspection. However, such a process becomes laborious when applied to more complex Directed Acyclic Graphs (DAGs) such as those that underlie real-world, observational datasets. We depict an example of such a DAG in Figure 6; observe that this DAG exhibits a *multiplicity of nodes participating in multiple of these primitive relationships*. This type of *structural analysis* is important as (1) it allows for correct adjustment of confounding variables during causal analysis which prevents problems like Simpson's paradox and (2) it enables clearer interpretability analysis of causal models [17, 22]. Real-world datasets often have thousands to millions of observations [9] with many variables, and as such we argue a *scalable solution* is needed. Thus, we attack this problem with *SAB*, an automated solution for structural analysis of Bayesian networks. The remainder of this report gives an (1) interesting motivational example, (2) formally describes the problem statement, lays out the landscape of related work, (3) outlines our method, (4) gives performance results from evaluations and (5) details future work.

2 Motivational Example

Table 1: Kidney Stone Treatment Study [5, 13, 30]

Stone Size \ Treatments	Treatment A Success Rate	Treatment B Success Rate
Small Stones	**93**% (81/87)	87% (234/270)
Large Stones	**73**% (192/263)	69% (55/80)
Total	78% (273/350)	**83**% (289/350)

Table 1 presents the results of a medical study [5] investigating two treatments for kidney stones. More abstractly, the results of such studies serve as suggestive guidelines for practitioners, Kidney Specialists in this scenario, who interpret the data and apply their conclusion to solve a real-world problem, e.g., intervention for kidney stones. Let's apply this *inference recipe* but only to the last row of the table. Doing so would lead us to prefer Treatment B overall. However, further analysis of the first two rows yields a contradictory outcome: *Treatment A is to be preferred for treating small and large stones.* This inversion of conclusions is an example of the well-known Simpson's Paradox [26] where the exclusion of confounding variables results in incorrect conclusions drawn from data. Figure 2 illustrates the causal relationships between the variables

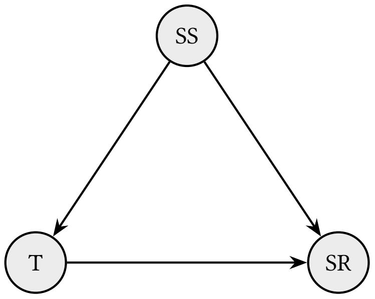

Figure 2: *Causal DAG of Table 1.* SS, T, and SR denote Kidney Stone Size, Treatment, and Sucess Rate, respectively.

of the medical study, and it is clear that the stone size exhibits a direct influencing relationship on the treatment choice and success rate. With this causal information **we can avoid the paradox by identifying the confounding variable (SS) and controlling for it before performing causal inference**. However, applying this intervention to massive, real-world observational datasets raises two challenges:

(1) How do we get a causal DAG reasonably faithful to the observational dataset?
(2) How do we systematically perform *scalable* structural analysis of causal DAGs to identify confounders? [This Work]

Challenge 1 can be solved manually and precisely by leveraging the knowledge of a domain expert. This is expensive, tedious, and not amenable to automation. There exists a cost-accuracy tradeoff through crowdsourcing platforms, such as Amazon Mechanical Turk or Large Language Models (LLMs) [28], where the cost of domain knowledge aggregation is more affordable but at the expense of lower quality data labeling. On the other hand, recent work has tackled this problem from the algorithmic standpoint through conditional independence [27], causal discovery [3, 23, 24], and more recently continuous optimizations [32]. Yet to the best of our knowledge, Challenge 2 remains open for research. In this work, we seek to not only address this challenge but to present a fast, scalable, end-to-end solution.

3 Related Works

Causal Inference. Existing Causal Analysis software libraries, namely CausalNex[1] automatically generate the causal graph from observational data and allow probabilistic predictions on said graph. However, to the best of our knowledge, there does not exist work that precisely addresses Challenge 2 from §2.
Network Analysis. NetworkX [10] and cuGRAPH [20] are defacto solutions for large-scale network analysis over graphs. This work studies structural analysis of causal DAGs, which, at the time of writing, neither of these existing libraries supports. We argue this is because these methods provide functionality for generic analysis of graphs rather than domain-specific investigation which we address in this work.
Causal Discovery. DoWhy [23] is a causal inference library that provides extensive support for causal learning. They provide implementations of different causal graph discovery algorithms such as LiNGAM [11, 24, 25]. Yet within this software, we did not find any features for structural analysis, specifically addressing node tagging.

4 Preliminaries

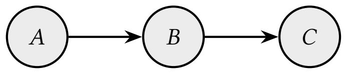

Figure 3: Chain structure: $A \longrightarrow B \longrightarrow C$ Node B is the chain participant.

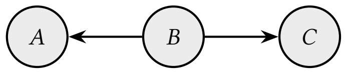

Figure 4: Fork structure: $B \longrightarrow A$, $B \longrightarrow C$. Node B is the fork parent (common cause).

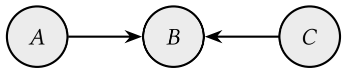

Figure 5: Collider structure: $A \longrightarrow B \longleftarrow C$. Node B is the collider.

We begin with a series of definitions that capture the key terminology to be used throughout this report.

Definition 4.1 (Directed Acyclic Graph (DAG)). A *directed acyclic graph* (DAG) is a pair $G = (V, E)$, where V is a finite set of nodes and $E \subseteq V \times V$ is a set of directed edges such that G contains no cycles, i.e., there does not exist a sequence of edges forming a path $v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_n \longrightarrow v_1$.

Definition 4.2 (Bayesian Network). A *Bayesian network* is a probabilistic graphical model defined by a DAG $G = (V, E)$, where each node $v \in V$ represents a random variable and each directed edge $u \longrightarrow v \in E$ encodes a direct probabilistic dependence.

Definition 4.3 (Chain). A node $v \in V$ is said to be part of a *chain structure* if there exist nodes $u, w \in V$ such that $u \longrightarrow v \longrightarrow w \in E$. In this structure, v acts as an intermediate link in a causal path.

Definition 4.4 (Fork). A node $v \in V$ is the *parent of a fork* if it has at least two distinct children $c_1, c_2 \in V$, i.e., $(v \longrightarrow c_1) \in E$ and $(v \longrightarrow c_2) \in E$ with $c_1 \neq c_2$. This structure represents a common cause of c_1 and c_2.

Definition 4.5 (Collider). A node $v \in V$ is a *collider* if it has at least two distinct parents $p_1, p_2 \in V$, such that $p_1 \longrightarrow v \in E$ and $p_2 \longrightarrow v \in E$ with $p_1 \neq p_2$. In this configuration, information may become dependent when conditioning on v or its descendants.

We also list out some key graph theoretical terms that are useful for the latter section when we describe our method.

Definition 4.6 (In-degree and Out-degree). Let $G = (V, E)$ be a DAG. For a node $v \in V$,

- The *in-degree* of v, denoted $\deg^-(v)$, is the number of edges directed into v:
$$\deg^-(v) = |\{u \in V : (u \to v) \in E\}|.$$
- The *out-degree* of v, denoted $\deg^+(v)$, is the number of edges directed out of v:
$$\deg^+(v) = |\{w \in V : (v \to w) \in E\}|.$$

5 Problem Statement

With the above preliminaries, we care about *node tagging* which we describe formally as follows.

Definition 5.1 (Node Tagging Problem). Given a Dataset $\mathcal{D}$, let $G = (V, E)$ be its underlying causal DAG. Denote the below as a set of topology tags
$$\mathcal{T} = \{\text{chain, collider, fork, untagged}\}$$
and $\rho : \{v, t, G\} \to \{0, 1\}$ as a function indicating whether a vertex $v \in V$ is involved (**1**) in a topology t contained in G or not (**0**). Define **node tagging** as the following function
$$\mathcal{F} : G \to V_T, \quad \text{where } V_T = \{(v, t) \in V \times T \mid \rho(v, t, G) = 1\}$$
Put simply, we seek to get a collection of tuples V_T, where each tuple $v \in V_T$ maps a node to a topology tag $t \in \mathcal{T}$. As an example, Figure 2 has the following node tagging.
$$V_T = \{(SS,\ fork), (T,\ chain), (SR,\ collider)\}$$

6 Method

We solve the problem laid out in §5 with an algorithm-system co-design methodology. Specifically, our key insight is that *local structural analysis unlocks parallelism.* We elaborate on the intuition and simplicity of our method using the succeeding example. Consider the graph in Figure 6. We can take each vertex (variable) as a local structure and ask a series of questions to determine its tag based on the structural definitions outlined in §4. Applied per vertex, this technique is independent and requires no global aggregate information; thus it yields an embarrassingly parallel implementation. We incorporate these conditional questions as structural rules which we encode in Table 2. We explicate our algorithm in the following section.

Tag	Rule	Example
Chain	One parent, ≥ 1 child	$A \to B \to C$ (tag B)
Fork	≥ 2 children	$B \to C, D$ (tag B)
Collider	≥ 2 parents	$A \to B \leftarrow C$ (tag B)
Untagged	None of the above	$A \to B$ (tag none)

Table 2: Tagging rules based on local structure in a DAG.

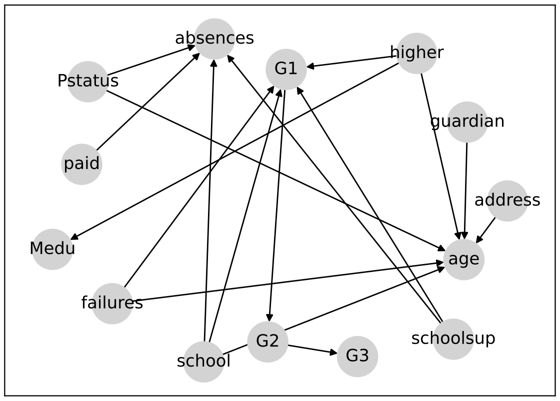

Figure 6: Causal DAG Generated for the UC Irvine Student Performance Dataset [6]. See Table 6 for a description of the vertices.

Algorithm 1: *SAB* Node Tagging

```
Input: $\mathcal{D}$: Observational Dataset
Result: $V_T$: Tagged nodes of Causal DAG of $\mathcal{D}$
begin
    $V_T \leftarrow [\ ]$
    $\mathcal{D}' \leftarrow$ Transform($\mathcal{D}$)
    $\mathcal{G} \leftarrow$ NOTEARS($\mathcal{D}'$)
    do in parallel
        for $v \in \mathcal{G}.V$ do
            $\varepsilon \leftarrow \deg^-(v)$
            $\omega \leftarrow \deg^+(v)$
            if $\varepsilon > 1$ then
                $V_T(v,\ collider) \leftarrow 1$
            else if $\varepsilon == 1$ and $\omega \geq 1$ then
                $V_T(v,\ chain) \leftarrow 1$
            if $\omega > 1$ then
                $V_T(v,\ fork) \leftarrow 1$
            end if
        end for
    end
    return $V_T$
end
```

6.1 Algorithm

Algorithm 1 encodes the intuition of our method. More explicitly, we outline the data processing pipeline (Figure 1) SAB implements as follows:

- SAB receives as input an observational dataset $\mathcal{D}$.
- We preprocess $\mathcal{D}$ by encoding categorical variables as numerical values.

- We generate the causal DAG $G = (V, E)$ underlying $\mathcal{D}$ using the `NOTEARS` algorithm by Zheng et al. [32].
- In parallel, for each vertex $v \in V$, we apply the rules from Table 2 to solve node tagging.
- We return the result set V_T.

6.2 Computational Complexity

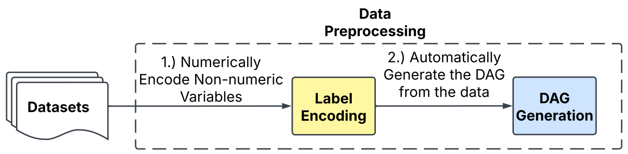
Data Preprocessing
Datasets
1.) Numerically Encode Non-numeric Variables
Label Encoding
2.) Automatically Generate the DAG from the data
DAG Generation

Figure 7: *SAB Preprocessing Pipeline*

Theorem 6.1. *Let P be the number of threads executing SAB. For a dataset $\mathcal{D} \in \mathbb{R}^{n\times d}$ whose causal graph is $G = (V, E)$, SAB's data processing pipeline completes in $O\left(d \cdot (n \log n) + nd^2 + d^3 + |E| + \left\lceil \frac{|V|}{P} \right\rceil\right)$ time, where $|V| \le d$.*

Proof. We begin by decomposing the runtime into three terms, C: data transformation, D: DAG generation and L: node tagging. Thus, we have the total time as $T = C + D + L$.
Data Transformation. We implement label encoding using a simple sort-hash technique where we sort and deduplicate categorical labels and hash to numerical values. Thus per column of $\mathcal{D}$, this encoding costs $O(n \log n)$, given there are n rows. In total, there are d columns, therefore $C = O(d \cdot (n \log n))$.
DAG Generation. For learning G from $\mathcal{D}$, we use the `NOTEARS` algorithm, whose runtime is dominated by loss computation $O(nd^2)$ and evaluation of the acyclicity constraint via matrix exponentiation $O(d^3)$. In addition, it takes $|E|$ to precompute the indegree and outdegree of all nodes; consequently, $D = O(nd^2 + d^3 + |E|)$.
Node Tagging. It is clear from Algorithm 1 that each thread processes $\left\lceil \frac{|V|}{P} \right\rceil$ vertices. Processing each vertex takes $O(1)$ time, therefore, $L = O\left(\left\lceil \frac{|V|}{P} \right\rceil\right)$.
Combining all three terms C, D, L, yields the below as desired.

$$T = O\left(d \cdot (n \log n) + nd^2 + d^3 + |E| + \left\lceil \frac{|V|}{P} \right\rceil\right) \qquad \square$$

6.3 Space Complexity

Theorem 6.2. *SAB's data processing pipeline exhibits space complexity $O(\max\{n, d^2\})$.*

Proof. Note that we can implement **Data Transformation** as a streaming algorithm where we only require a working set of size n entries. **DAG Generation** requires an adjacency matrix of size $O(d^2)$, while **Node Tagging** requires storage for the indegrees, outdegrees and the result vector V_T all of which have space complexity $O(|V|)$. Given $|V| \le d$, the total space complexity is therefore $O(\max\{n, d^2\})$. $\square$

7 Implementation Details

We implemented the entire SAB pipeline using Python in 100 lines of code. Below, we elaborate on existing work we leveraged to implement different layers of the pipeline.

Data Transformation. SAB uses scikit-learn[18] to preprocess the data, leveraging its off-the-shelf LabelEncoder for converting categorical features into numerical ones.

DAG Generation. SAB relies heavily on the Structural Model abstraction exposed by CausalNex [1] and networkx [10]. CausalNex provides an off-the-self implementation of the NOTEARS algorithm, which SAB uses for DAG generation.

Node Tagging. To realize parallel execution, SAB uses the *concurrent.futures* module from the Python standard library. This module provides a coherent API for asynchronously launching and destroying a pool of threads for executing the parallel subroutine in Algorithm 1.

7.1 Work-Conserving Scheduling

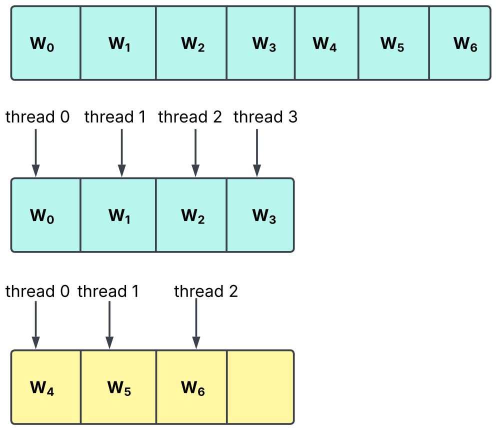

Figure 8: *Stride Work Partitioning.* The topmost array identifies seven work items that must be partitioned across four threads. The figure illustrates the strided work partitioning scheme where $thread_0$ gets W_0 in iteration 1 and W_4 in iteration 2 and so on.

In SAB, to ensure a fair distribution of work across threads, we use **Stride Work Partitioning** (SWP). This is a popular scheme in parallel computing [15] where contiguous threads are assigned to work items contiguously. This abstracts the linear list of work items to a grid G_w whose dimensions are $\left(\left\lceil \frac{W}{P} \right\rceil, P\right)$, where W is the number of work items and P the number of threads. For example, Figure 8 exemplifies how this scheme works when $W = 7$ and $P = 4$. We show Python code implementing this scheduling policy in Figure 9.

Note that SWP is *work-conserving* as it is never the case that in any iteration there are pending tasks and idle threads. An intuitive but suboptimal scheme would be **Blocked Partitioning** (BP), where each thread is allocated a contiguous slice of $\left\lceil \frac{W}{P} \right\rceil$ work

```python
# executed per thread
def sab_threaded(dag: DiGraph,
                result : Dict[str, TopologyLabel],
                tid: int,
                n_threads: int) -> None:
    idx = tid
    n_nodes = len(dag.nodes)
    nodes = list(dag.nodes)
    while idx < n_nodes:
        v = nodes[idx]
        i_d = dag.in_degree(v)
        o_d = dag.out_degree(v)
        if i_d > 1:
            result[v].collider = True
        elif i_d == 1 and o_d >= 1:
            result[v].chain = True
        if o_d > 1:
            result[v].fork = True
        idx += n_threads
```

Figure 9: *Node Tagging Scheduling and Task Loop.* This subroutine is executed per thread using SWP.

items. Consider $W = 5$ and $P = 4$. With BP, $thread_0$ and $thread_1$ get 2 work items, whereas $thread_2$ gets one and $thread_3$ gets none. This schedule is clearly not work-conserving as in iteration one W_1 would be pending while $thread_3$ is idle. On the other hand, SWP would yield a work-conserving placement by dispatching $W0, W1, W2, W3, W4$ to $thread_0, thread_1, thread_2, thread_3, thread_0$, respectively.

7.2 Asynchronous Task Dispatch

```python
def sab_core(dag: DiGraph, parallel=False) -> Dict[str, TopologyLabel]:
    result : Dict[str, TopologyLabel] = {}
    if not parallel:
        sab_threaded(dag, result, 0, 1)
    else:
        num_threads = min(8, len(dag.nodes))
        with concurrent.futures.ThreadPoolExecutor(max_workers=num_threads) as sab_executor:
            for tid in range(num_threads):
                sab_executor.submit(sab_threaded, dag, result, tid, num_threads)
```

Figure 10: *SAB Asynchronous Executor*

The *concurrent.futures* library dispatches callables *asynchronously* rather than sequentially, such that the main thread does not block within the loop in Figure 10. The launched callable therein is implemented in Figure 9. Finally, all of these are consolidated into a simple API which SAB exposes, that consumes a CSV dataset and generates its node tagging results. All code is available at https://github.com/osayamenja/sab.

```python
def sab(dataset: str, parallel=False) -> Dict[str, TopologyLabel]:
    assert is_csv_file(dataset) and os.path.exists(dataset)
    dag = preprocessor(dataset)
    result = sab_core(dag, parallel)
    return result
```

Figure 11: *SAB External API.* SAB presents a simple interface requiring only the input dataset path and a flag requesting a parallel or sequential backend.

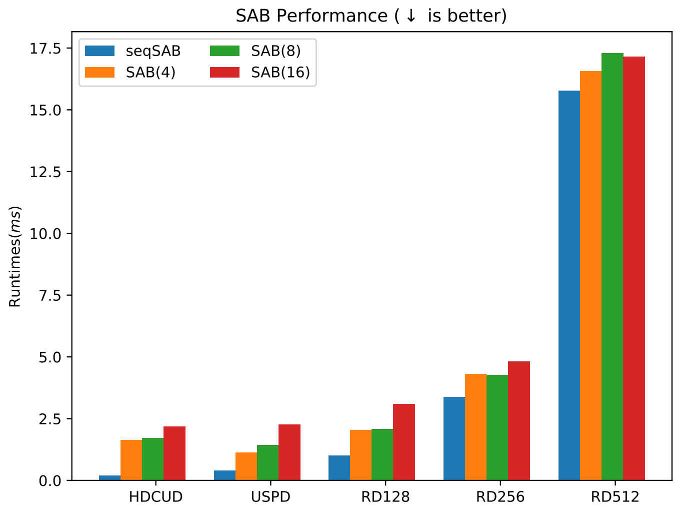

Figure 12: *SAB's Node Tagging Runtime.* On the x-axis, *HDCUD* denotes Heart Disease Cleveland UCI Dataset, while *USPD* expands to UCI Student Performance Dataset. Note the depicted runtime does not include *Transform* or *DAG Generation.* SAB(x) means SAB implemented with x Python threads.

8 Evaluation

For this section, we seek to answer the following questions:

(1) **Performance** How do SAB's runtimes compare with baselines?
(2) **Overhead** What is the memory and subsystem overhead incurred by SAB?
(3) **Robustness** How well does SAB scale to massive datasets?

8.1 Experiment Setup

Hardware. We run all our experiments on an AMD EPYC 7513 32-Core Processor with (1) 2 sockets, (2) 32 cores per socket, and (3) 2 threads per core. Therefore, it supports hardware concurrency of up till 128. We choose this hardware infrastructure as it allows for measuring performance gains at very high levels of concurrency.

Baselines. Given no prior work, we use a sequential version of SAB: *seqSAB*, as a baseline for comparing against the concurrent SAB. This baseline is intellectually interesting, as the results in Figure 12 would show, because it allows for characterizing the critical region where the synchronization overhead of using concurrency outweighs its benefits. seqSAB is the default configuration from the external API depicted in Figure 11 where `parallel = False`. Toggling the flag yields parallel SAB.

8.2 Datasets

We use Kaggle to obtain high-quality datasets. We define "high-quality" as datasets with **usability score** $u >= 8$. This threshold ensures the dataset is descriptively complete, has a public data card, and parsable, downloadable CSV files. We selected datasets with higher number of variables (d) as that is the bottleneck for our algorithm (see §6.2).

Table 3: Summary of Selected Kaggle Datasets

Dataset	# Variables	# Rows	Kaggle Usability Score
Heart Disease Cleveland UCI [12]	14	303	8.82
UCI Student Performance [6]	33	649	10

Table 4: Summary of Dense Synthetic DAGs

Synthetic DAG	Variables
RD128	128
RD256	256
RD512	512

8.3 Note on Accuracy

For any input DAG G, our deterministic algorithm yields a 100% accurate solution, so we instead investigate performance metrics, overhead and do a robustness analysis in the succeeding sections.

8.4 Node Tagging Runtime

We show our empirical performance results in Figure 12. Interestingly, seqSAB performs significantly better, up to 11.1x, than parallel SAB and this is due to the **G**lobal **I**nterpreter **L**ock (GIL) used by the Python [2, 19]. The GIL allows only Python thread to execute bytecode at any instant. However, I/O-bound workloads see speedup when using python multiprocessing due to context-switching [2]. However, SAB is not I/O-bound thus, we see no speedup only further slowdown due to the cost of creating, synchronizing, and destroying threads.

Key Takeaway 1

To realize concurrent gains, SAB must be implemented in a programming language that provides access to hardware parallelism, such as C++ or CUDA. Otherwise, a single-threaded implementation is best for sequential languages like Python.

We leave the implementation of SAB in CUDA or C++ as exciting future work.

8.5 Preprocessing Overhead

We evaluate the preprocessing overhead of SAB by isolating the runtime contributions of the data transformation and DAG generation phases (see Figures 13 and 14). These results are measured only for the real-world Kaggle datasets, as synthetic DAGs were directly generated and thus bypass preprocessing. Our findings reveal that preprocessing accounts for the majority of SAB's end-to-end runtime, consuming between 85% and 97% of total system time. This is consistent across both sequential and parallel variants. In particular, label encoding and NOTEARS-based [32] DAG generation are the dominant operations, with transformation time increasing sharply with the number of categorical variables, and DAG generation scaling cubically with the number of columns (d^3).

Key Takeaway 2

SAB's preprocessing pipeline occupies between *85% to 97%* of the end-to-end system runtime.

This result suggests that efforts to optimize SAB should focus primarily on reducing preprocessing costs, rather than the node tagging algorithm itself. Potential improvements include caching transformation outputs, accelerating NOTEARS with GPU support [20], or replacing it with linear-time DAG approximators [11, 31] when structure fidelity is non-critical.

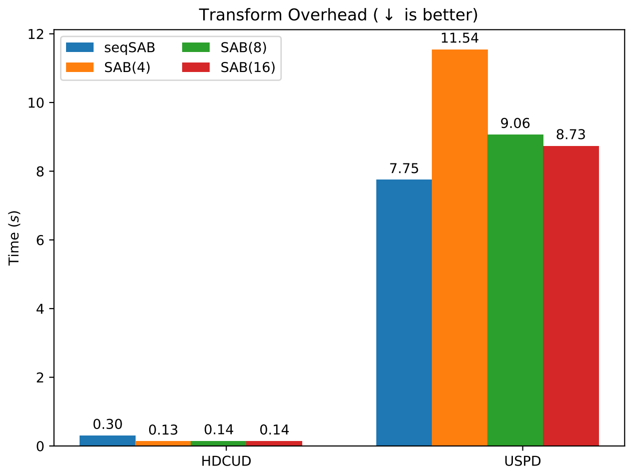

Figure 13: ***SAB's Transform Overhead.***

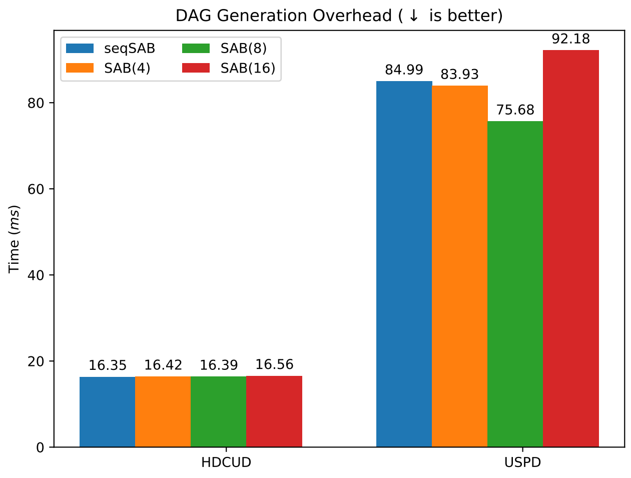

Figure 14: ***SAB's DAG Generation Overhead.***

8.6 Memory Consumption

We report the memory overhead of SAB in Figure 15 for both the Heart Disease Cleveland UCI (HDCUD) and UCI Student Performance Dataset (USPD). Our measurement entails using `tracemalloc` [8] from the Python standard library to measure peak memory usage

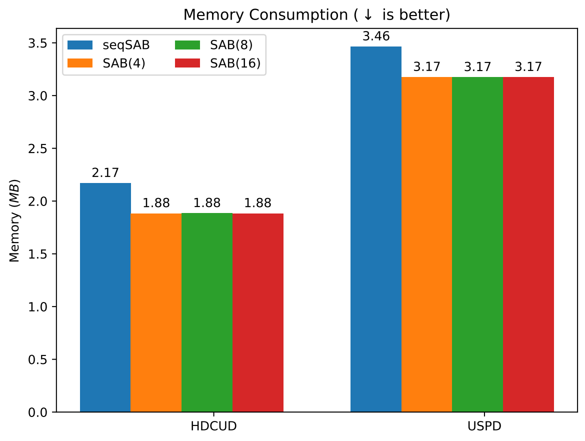

Figure 15: *SAB's Memory Overhead.*

during the execution of SAB. The results show that across all concurrency configurations, SAB incurs negligible memory overhead, with memory usage ranging from 1.88MB to 3.46MB. The sequential variant (seqSAB) exhibits the lowest footprint, while parallel variants demonstrate marginal increases due to potentially thread-local buffers.

Key Takeaway 3

SAB's memory footprint remains minimal across all configurations, making it suitable for deployment in memory-constrained environments.

This minimal footprint is a direct consequence of the embarrassingly parallel nature of our tagging algorithm, which only requires in-degree and out-degree information per node.

9 Future Work

While SAB demonstrates strong potential as a scalable, automated tool for structural analysis of Bayesian networks, several avenues remain for future exploration:

- **Language-Level Optimization:** As evidenced by runtime performance, Python's Global Interpreter Lock (GIL) limits the gains from multithreading. Re-implementing SAB in C++ or CUDA could unlock significant concurrency improvements by enabling true parallelism.
- **Distributed Execution:** For massive graphs beyond single-machine memory limits, SAB could be extended using a distributed graph processing framework such as Dask [21] or Ray [14]. This would allow efficient analysis of web-scale DAGs, for example.
- **Graph Compression and Indexing:** As preprocessing dominates runtime, we propose integrating lightweight graph compression [4, 7] or dimensionality reduction [16] techniques to accelerate DAG generation and transformation stages.

10 Conclusion

This work introduces SAB, a parallel system for the automated structural analysis of Bayesian networks. We define and solve the node tagging problem by leveraging the local nature of causal topology detection, enabling an embarrassingly parallel implementation. SAB is evaluated across real-world and synthetic datasets, showing minimal memory overhead and strong scalability properties.

Though Python's runtime constraints limit concurrent speedup, SAB sets the stage for deeper investigation into high-performance causal inference tooling. Our findings emphasize the significance of preprocessing as the dominant bottleneck and highlight opportunities for accelerating this step. Overall, SAB contributes a practical and extensible tool for scalable causal structure analysis.

References

[1] Paul Beaumont, Ben Horsburgh, Philip Pilgerstorfer, Angel Droth, Richard Oentaryo, Steven Ler, Hiep Nguyen, Gabriel Azevedo Ferreira, Zain Patel, and Wesley Leong. 2021. *CausalNex*. https://github.com/quantumblacklabs/causalnex
[2] David Beazley. 2010. Understanding the Python GIL. PyCon 2010. https://www.dabeaz.com/python/UnderstandingGIL.pdf Accessed: 2025-05-06.
[3] Patrick Blöbaum, Peter Götz, Kailash Budhathoki, Atalanti A. Mastakouri, and Dominik Janzing. 2024. DoWhy-GCM: An Extension of DoWhy for Causal Inference in Graphical Causal Models. *Journal of Machine Learning Research* 25, 147 (2024), 1–7. http://jmlr.org/papers/v25/22-1258.html
[4] Giorgos Bouritsas, Andreas Loukas, Nikolaos Karalias, and Michael Bronstein. 2021. Partition and Code: learning how to compress graphs. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 18603–18619. https://proceedings.neurips.cc/paper_files/paper/2021/file/9a4d6e8685bd057e4f68930bd7c8ecc0-Paper.pdf
[5] C R Charig, D R Webb, S R Payne, and J E Wickham. 1986. Comparison of treatment of renal calculi by open surgery, percutaneous nephrolithotomy, and extracorporeal shockwave lithotripsy. *BMJ* 292, 6524 (1986), 879–882. doi:10.1136/bmj.292.6524.879 arXiv:https://www.bmj.com/content/292/6524/879.full.pdf
[6] Paulo Cortez. 2008. Student Performance. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5TG7T.
[7] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing Graphs and Indexes with Recursive Graph Bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 1535–1544. doi:10.1145/2939672.2939862
[8] Python Software Foundation. 2025. tracemalloc — Trace memory allocations — docs.python.org. https://docs.python.org/3/library/tracemalloc.html. [Accessed 07-05-2025].
[9] Tomas Geffner, Javier Antoran, Adam Foster, Wenbo Gong, Chao Ma, Emre Kiciman, Amit Sharma, Angus Lamb, Martin Kukla, Nick Pawlowski, Miltiadis Allamanis, and Cheng Zhang. 2022. Deep End-to-end Causal Inference. In *NeurIPS 2022 Workshop on Causality for Real-world Impact*. https://openreview.net/forum?id=6DPVXzjnbDK
[10] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). Pasadena, CA USA, 11 – 15.
[11] Aapo Hyvärinen and Stephen M. Smith. 2013. Pairwise likelihood ratios for estimation of non-Gaussian structural equation models. *J. Mach. Learn. Res.* 14, 1 (Jan. 2013), 111–152.
[12] Andras Janosi, William Steinbrunn, Matthias Pfisterer, and Robert Detrano. 1989. Heart Disease. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C52P4X.
[13] Steven A Julious and Mark A Mullee. 1994. Confounding and Simpson's paradox. *BMJ* 309, 6967 (1994), 1480–1481. doi:10.1136/bmj.309.6967.1480 arXiv:https://www.bmj.com/content
[14] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: a distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 561–577.
[15] NVIDIA. 2025. *CUB: cub::WarpScan, CUDA Core Compute Libraries (CCCL)*. https://nvidia.github.io/cccl/cub/api/classcub_1_1WarpScan.html#

[16] Fernando Paulovich, Alessio Arleo, and Stef van den Elzen. 2024. When Dimensionality Reduction Meets Graph (Drawing) Theory: Introducing a Common Framework, Challenges and Opportunities. arXiv:2412.06555 [cs.LG] https://arxiv.org/abs/2412.06555

[17] Judea Pearl. 2009. *Causality: Models, Reasoning and Inference* (2nd ed.). Cambridge University Press, USA.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[19] Python Software Foundation. 2022. Global Interpreter Lock. https://wiki.python.org/moin/GlobalInterpreterLock. Accessed: 2025-05-06.

[20] Brad Rees, Alex Fender, Chuck Hastings, Seunghwa Kang, Iroy30, Rick Ratzel, gpuCI, James Wyles, Kumar Aatish, Ray Douglass, Xavier Cadet, Andrei Schaffer, Hugo Linsenmaier, Joseph Nke, Alex Barghi, AJ Schmidt, Paul Taylor, Dillon Cullinan, Erik Welch, Oded Green, Ralph Liu, Bradley Dice, Don Acosta, Vyas Ramasubramani, Vibhu Jawa, GALI PREM SAGAR, Naim, James Lamb, Jake Awe, and Mike Wendt. 2025. *rapidsai/cugraph.* https://github.com/rapidsai/cugraph

[21] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra (Eds.). 130 – 136.

[22] Donald B Rubin. 2005. Causal Inference Using Potential Outcomes. *J. Amer. Statist. Assoc.* 100, 469 (2005), 322–331. doi:10.1198/016214504000001880 arXiv:https://doi.org/10.1198/016214504000001880

[23] Amit Sharma and Emre Kiciman. 2020. DoWhy: An End-to-End Library for Causal Inference. *arXiv preprint arXiv:2011.04216* (2020).

[24] Shohei Shimizu, Patrik O. Hoyer, Aapo Hyvärinen, and Antti Kerminen. 2006. A Linear Non-Gaussian Acyclic Model for Causal Discovery. *J. Mach. Learn. Res.* 7 (Dec. 2006), 2003–2030.

[25] Shohei Shimizu, Takanori Inazumi, Yasuhiro Sogawa, Aapo Hyvärinen, Yoshinobu Kawahara, Takashi Washio, Patrik O. Hoyer, and Kenneth Bollen. 2011. DirectLiNGAM: A Direct Method for Learning a Linear Non-Gaussian Structural Equation Model. *J. Mach. Learn. Res.* 12, null (July 2011), 1225–1248.

[26] E. H. Simpson. 2018. The Interpretation of Interaction in Contingency Tables. *Journal of the Royal Statistical Society: Series B (Methodological)* 13, 2 (12 2018), 238–241. doi:10.1111/j.2517-6161.1951.tb00088.x arXiv:https://academic.oup.com/jrsssb/article-pdf/13/2/238/49093972/jrsssb_13_2_238.pdf

[27] Eric V. Strobl, Kun Zhang, and Shyam Visweswaran. 2017. Approximate Kernel-based Conditional Independence Tests for Fast Non-Parametric Causal Discovery. arXiv:1702.03877 [stat.ME] https://arxiv.org/abs/1702.03877

[28] Zhen Tan, Dawei Li, Song Wang, Alimohammad Beigi, Bohan Jiang, Amrita Bhattacharjee, Mansooreh Karami, Jundong Li, Lu Cheng, and Huan Liu. 2024. Large Language Models for Data Annotation and Synthesis: A Survey. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 930–957. doi:10.18653/v1/2024.emnlp-main.54

[29] Jussi Viinikka, Antti Hyttinen, Johan Pensar, and Mikko Koivisto. 2020. Towards scalable bayesian learning of causal DAGs. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 552, 11 pages.

[30] Wikipedia contributors. 2025. Simpson's paradox — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Simpson%27s_paradox&oldid=1278179208. [Online; accessed 5-May-2025].

[31] Yue Yu, Jie Chen, Tian Gao, and Mo Yu. 2019. DAG-GNN: DAG Structure Learning with Graph Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 7154–7163. https://proceedings.mlr.press/v97/yu19a.html

[32] Xun Zheng, Bryon Aragam, Pradeep K Ravikumar, and Eric P Xing. 2018. DAGs with NO TEARS: Continuous Optimization for Structure Learning. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/e347c51419ffb23ca3fd5050202f9c3d-Paper.pdf

A Heart Disease UCI Dataset

Table 5: Attributes in the Heart Disease Cleveland UCI Dataset

Attribute	Type	Description
`age`	Numeric	Age of the patient
`sex`	Nominal	Gender of the patient
`cp`	Nominal	Type of chest pain experienced
`trestbps`	Numeric	Resting blood pressure
`chol`	Numeric	Serum cholesterol level
`fbs`	Nominal	Fasting blood sugar status
`restecg`	Nominal	Resting electrocardiographic results
`thalach`	Numeric	Maximum heart rate achieved
`exang`	Nominal	Exercise-induced angina
`oldpeak`	Numeric	ST depression induced by exercise
`slope`	Nominal	Slope of peak exercise ST segment
`ca`	Numeric	Number of major vessels colored by fluoroscopy
`thal`	Nominal	Thalassemia status
`target`	Nominal	Presence or absence of heart disease

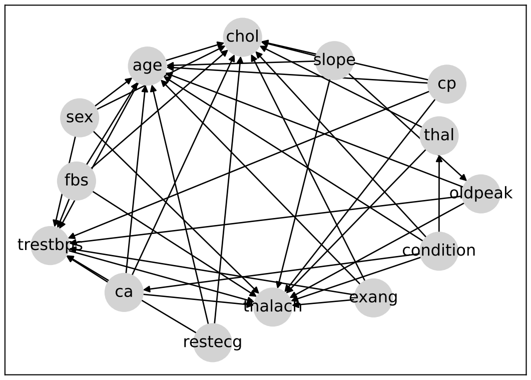

Figure 16: Causal DAG Generated for the UC Irvine Cleveland Heart Disease Dataset [6]. See Table 5 for a description of the vertices.

B UCI Student Performance Dataset

Table 6: Attributes in the UCI Student Performance Dataset

Attribute	Type	Description
`school`	Nominal	Student's school
`sex`	Nominal	Student's gender
`age`	Numeric	Student's age
`address`	Nominal	Home address type
`famsize`	Nominal	Family size
`Pstatus`	Nominal	Parent's cohabitation status
`Medu`	Numeric	Mother's education level
`Fedu`	Numeric	Father's education level
`Mjob`	Nominal	Mother's job type
`Fjob`	Nominal	Father's job type
`reason`	Nominal	Reason for choosing school
`guardian`	Nominal	Student's guardian
`traveltime`	Numeric	Travel time to school
`studytime`	Numeric	Weekly study time
`failures`	Numeric	Number of past class failures
`schoolsup`	Nominal	Extra educational support
`famsup`	Nominal	Family educational support
`paid`	Nominal	Extra paid classes
`activities`	Nominal	Extra-curricular activities
`nursery`	Nominal	Attended nursery school
`higher`	Nominal	Aspires to higher education
`internet`	Nominal	Internet access at home
`romantic`	Nominal	In a romantic relationship
`famrel`	Numeric	Quality of family relationships
`freetime`	Numeric	Free time after school
`goout`	Numeric	Social activity frequency
`Dalc`	Numeric	Weekday alcohol consumption
`Walc`	Numeric	Weekend alcohol consumption
`health`	Numeric	Current health status
`absences`	Numeric	Number of absences
`G1`	Numeric	First period grade
`G2`	Numeric	Second period grade
`G3`	Numeric	Final grade