Local Causal Discovery for Estimating Causal Effects

Shantanu Gupta, Zachary C. Lipton, David Childers

Carnegie Mellon University
{shantang,zlipton,dchilders}@cmu.edu

Abstract

Even when the causal graph underlying our data is unknown, we can nevertheless use observational data to narrow down the possible values that an average treatment effect (ATE) can take by (1) identifying the graph up to a Markov equivalence class; and (2) estimating that ATE for each graph in the class. While the PC algorithm can identify this class under strong faithfulness assumptions, it can be computationally prohibitive. Fortunately, only the local graph structure around the treatment is required to identify an ATE, a fact exploited by local discovery algorithms to identify the possible values for an ATE more efficiently. In this paper, we introduce Local Discovery using Eager Collider Checks (LDECC), a new local discovery algorithm that leverages unshielded colliders to orient the treatment's parents differently from existing methods. We show that there exist graphs where LDECC exponentially outperforms existing local discovery algorithms and vice versa. Moreover, we show that LDECC and existing algorithms rely on different faithfulness assumptions, leveraging this insight to test for and recover from certain faithfulness violations.

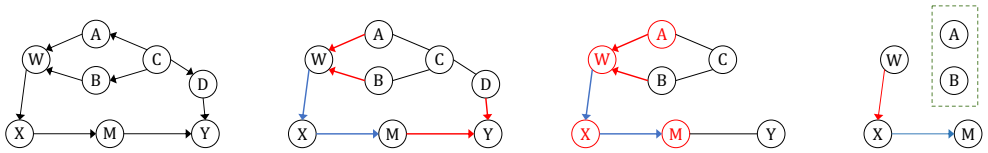
1 Introduction

Estimating an average treatment effect (ATE) from observational data typically requires structural knowledge, which can be represented in the form of a causal graph. While a rich literature offers methods for identifying and estimating causal effects given a *known* causal graph [13, 8, 3], many applications require that we investigate the values that an ATE could possibly take when the causal graph is unknown. Under certain faithfulness assumptions, we can (i) perform *causal discovery* using observational data to identify the graph up to a Markov equivalence class (MEC); and (ii) estimate the ATE of a given treatment on a given outcome for every graph in the MEC, thus identifying the set of possible ATE values. We denote this (unknown) set of identified ATE values by Θ^*.

Causal discovery has been investigated under a variety of assumptions [12, 2]. Under causal sufficiency (i.e., no unobserved variables) and faithfulness, the PC algorithm [12, Sec. 5.4.2] can consistently learn the MEC of the true graph from observational data (and thus Θ^*). However, fully characterizing the MEC is not always necessary and can be prohibitively expensive. Maathuis et al. [5] proved that the local structure around the treatment node is sufficient for estimating Θ^*. Existing local causal discovery algorithms [16, 19, 15, 1] leverage this insight, discovering just enough of the graph to identify any parents and children of the treatment that PC would have discovered. These methods sequentially discover the local structure around the treatment, its neighbors, and so on, terminating whenever all neighbors of the treatment are oriented (or no remaining neighbors can be oriented). These algorithms typically differ in how they discover the local structure at each step. We instantiate a simple version of existing local causal discovery algorithms, which we call Sequential Discovery (SD), that runs the PC algorithm locally to discover the local structure around the nodes.

In this work, we introduce Local Discovery with Eager Collider Checks (LDECC) (Sec. 3), a new local discovery algorithm that uses a different method to orient the parents of a treatment X. Initially,

NeurIPS 2022Workshop on Causal Machine Learning for Real-World Impact (CML4Impact 2022).



(a) True DAG $\mathcal{G}^*$. (b) CPDAG from PC. (c) SD's output (d) LDECC

Figure 1: Demonstration of PC, SD, and LDECC for the graph in (a).

LDECC runs the same set of Conditional Independence (CI) tests as SD to identify the neighbors of X. Subsequently, LDECC chooses the same CI tests as PC would choose given the state of the graph, with one crucial exception: Whenever two nodes A and B are identified such that (i) $A \perp\!\!\!\perp B|\mathbf{S}$ for some set $\mathbf{S}$; and (ii) $X \notin \mathbf{S}$, LDECC then checks whether they become dependent when X is added to the conditioning set. If the test reveals dependence $A \not\perp\!\!\!\perp B|\mathbf{S} \cup \{X\}$, then LDECC orients the smallest subset of X's neighbors that d-separate it from $\{A, B\}$ as parents. We prove that, under faithfulness and with access to a CI oracle, the estimated ATE set Θ_{LDECC} is equal to Θ^*.

We analyze LDECC and compare it to SD, highlighting complementary strengths, both in terms of their computational and the faithfulness requirements. We present classes of causal graphs where LDECC performs exponentially fewer CI tests than SD, and vice versa. Thus, the methods can be combined profitably (by running LDECC and SD in parallel and terminating when either algorithm terminates), avoiding exponential runtimes whenever either algorithm's runtime is subexponential. We also find that LDECC and SD rely on a different set of faithfulness assumptions (Sec. 3.2). Thus, there are classes of faithfulness violations where one algorithm will correctly estimate the ATE set while the other, in general, will not. Under the assumption that one of the sets of faithfulness assumptions is correct, we propose a hybrid procedure which recovers a conservative bound on the ATE set which can in some cases can be made sharp. We empirically test LDECC on synthetic as well as semi-synthetic graphs (Sec. 4) and show that it performs comparably to SD (and PC) and typically runs fewer CI tests than SD.

2 Preliminaries

In this work, we assume that the causal structure of the observational data can be encoded using a DAG $\mathcal{G}^*(\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ and $\mathbf{E}$ are the set of nodes and edges, respectively. Each edge $A \to B \in \mathbf{E}$ indicates that A is a direct cause of B. Throughout this paper, we denote the treatment node by X and the outcome node by Y. For a node $V \in \mathbf{V}$, we denote its neighbors, parents, children, and descendants by $\text{Ne}(V), \text{Pa}(V), \text{Ch}(V)$, and $\text{Desc}(V)$, respectively. Let $\text{Ne}^+(V) = \text{Ne}(V) \cup \{V\}$. An unshielded collider (UC) is a triple $P \to R \leftarrow Q$ such that P—$Q \notin \mathbf{E}$. For a UC $\alpha = (P \to R \leftarrow Q)$, let $\text{sep}(\alpha) = \min\{|\mathbf{S}| : \mathbf{S} \subseteq \mathbf{V} \setminus \{P, Q\}$ and $P \perp\!\!\!\perp Q|\mathbf{S}\}$ denote the size of a smallest subset $\mathbf{S}$ that d-separates P and Q.

A DAG entails a set of CIs via *d-separation* [8, Sec. 1.2.3]. DAGs that entail the same set of CIs form an MEC, which can be characterized by a *completed partially directed acyclic graph* (CPDAG). For the true DAG $\mathcal{G}^*$, we denote by Θ^* the set of ATE values (of X on Y) in each DAG in the MEC corresponding to $\mathcal{G}^*$. The causal faithfulness assumption (CFA) holds iff all CIs satisfied by $\mathbb{P}(\mathbf{V})$ are entailed by $\mathcal{G}^*$. Throughout this work, we focus on causally sufficient graphs and unless stated otherwise, assume that the CFA holds. Under the CFA, the PC algorithm recovers this MEC roughly as follows (demonstrated for the DAG in Fig. 1a): (i) estimate the skeleton by running CI tests; (ii) find UCs in this skeleton (Fig. 1b red edges); and (iii) orient additional edges using Meek's rules [7] (Fig. 1b blue edges) to get a CPDAG. Full details on the PC algorithm are in Appendix A.

We instantiate existing local discovery algorithms using SD (Fig. 2). SD sequentially finds the neighbors of nodes (*FindNeighbors* in Fig. 8a) starting from X, then its neighbors, and so on (Lines 3–7). After each such local discovery step, SD orient nodes in the subgraph discovered until that point using UCs and Meek's rules like PC (Line 8). It terminates if all neighbors of X get oriented. The ATE set is estimated by applying the backdoor adjustment [8, Thm. 3.3.2] with every *locally valid* parent set of X, i.e., one that does not create a new UC at X (see Lines 11–16 and *isLocallyValid* in Fig. 2). Consider the DAG in Fig. 1a. Here, SD will discover the neighbors of X, W, M and A. After this, the UC $A \to W \leftarrow B$ will be detected and after propagating orientations, both W and

```
def OrientChildren(Ne(X), mns_X):
    children = ∅;
    for C ∈ Ne(X) do
        for V ∈ V s.t. C ∉ mns_X(V) do
            if C ⫫̸ V|mns_X(V) then
                children.add(C) ;
        end
    end
    return children;
def isLocallyValid(S):
    for every A, B ∈ S do
        if isNonCollider(A—X—B)
          then return False;
    end
    return True;
```

```
def SD_algorithm(Treatment X):
    queue = [X], done = ∅;
    while queue ≠ ∅ do
        V = queue.pop();
        FindNeighbors(V);
        done.add(V);
        queue.add(Ne(V) \ (done ∪
          queue));
        parents, children, unoriented =
          OrientNodesInSubgraph(done);
        if unoriented = ∅ then break ;
    end
    Θ_SD = ∅;
    for S ⊆ unoriented do
        if isLocallyValid(S) then
            R = S ∪ parents;
            Θ_SD.add(β_{Y|X,R});
    end
    return Θ_SD;
```

Figure 2: The SD algorithm and additional subroutines.

```
Input: Treatment X.
Ne(X), mns_X = FindNbrsAndMNS(X);
parents = ∅;
children = OrientChildren(Ne(X), mns_X);
unoriented = Ne(X) \ children;
for (A ⫫ B|S) ∈ PC test (A, B ≠ X) do
    if A, B ∈ Ne(X) and X ∉ S then
        parents.add({A, B});
        parents.add(S ∩ Ne(X));
    else if A, B ∈ Ne(X) and X ∈ S then
        MarkNonCollider(A—X—B);
        for V ∈ Ne(X) \ (S ∪ {A, B}) do
            if A ⫫̸ B|{S, V} then
              children.add(V) ;
        end
    else if X ∉ S and A ⫫̸ B|{S, X} then
        for V ∈ {A, B} do
            if V ∈ Ne(X) then
              parents.add(V);
            else parents.add(mns_X(V));
        end
    end
    for P ∈ parents, C ∈ unoriented do
        if isNonCollider(P—X—C) then
          children.add(C) ;
    end
    unoriented.remove(parents ∪ children);
    if unoriented = ∅ then break ;
end
Θ_LDECC = ∅;
for S ⊆ unoriented do
    if isLocallyValid(S) then
        R = S ∪ parents;
        Θ_LDECC.add(β_{Y|X,R});
end
Output: Θ_LDECC
```

Figure 3: The LDECC algorithm.

M will be oriented and SD terminates (Fig. 1c). Full details on the PC and SD algorithms are in Appendix A.

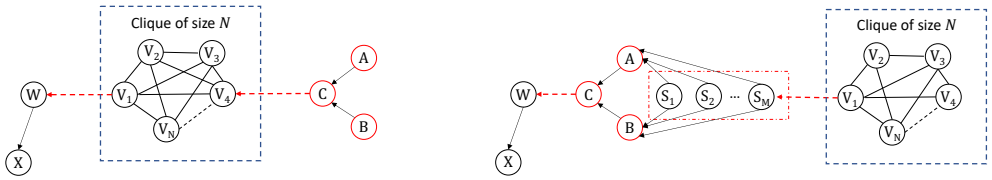
3 Local Discovery using Eager Collider Checks (LDECC)

In this section, we propose LDECC, a local causal discovery algorithm that orients the parents of X by leveraging UCs differently from existing methods (Prop. 3). We prove its correctness under the CFA (Thm. 1) and then show its complementary nature relative to SD in terms of computational (Sec. 3.1) and faithfulness (Sec. 3.2) requirements. We defer the proofs to Appendix B.2.

We first define a *Minimal Neighbor Separator* (MNS) which plays a key role in LDECC.

Definition 1 (Minimal Neighbor Separator (MNS))**.** *For a DAG $\mathcal{G}(\mathbf{V}, \mathbf{E})$ and nodes X and $A \notin Ne^+(X)$; $mns_X(A) \subseteq Ne(X)$ is the unique set (see Prop. 2) of nodes such that (i) (d-separation) $A \perp\!\!\!\perp X|mns_X(A)$, and (ii) (minimality) for any $\mathbf{S} \subset mns_X(A)$, $A \not\perp\!\!\!\perp X|\mathbf{S}$.*

For the graph in Fig. 1a, we have $\text{mns}_X(A) = \text{mns}_X(B) = \text{mns}_X(C) = \{W\}$ and $\text{mns}_X(Y) = \{W, M\}$. While an MNS need not exist for every node (see Example 3), $\text{mns}_X(V)$ always exists $\forall V \notin \text{Desc}(X)$ (which is sufficient for correctness of LDECC):



(a) LDECC outperforms SD.

(b) SD outperforms LDECC.

Figure 4: Classes of graphs where LDECC and SD perform a different number of CI tests.

Proposition 1. *For any node $V \notin (Desc(X) \cup Ne^+(X))$, $mns_X(V)$ exists and $mns_X(V) \subseteq Pa(X)$.*

Proposition 2 (Uniqueness of MNS)**.** *For every node V such that $mns_X(V)$ exists, it is unique.*

Proposition 3 (Eager Collider Check)**.** *For nodes $A, B \in \mathbf{V} \setminus Ne^+(X)$ and $\mathbf{S} \subseteq \mathbf{V} \setminus \{A, B, X\}$, if (i) $A \perp\!\!\!\perp B|\mathbf{S}$; and (ii) $A \not\perp\!\!\!\perp B|\mathbf{S} \cup \{X\}$; then $A, B \notin Desc(X)$ and $mns_X(A), mns_X(B) \subseteq Pa(X)$.*

Prop. 3 suggests a different strategy for orienting parents of X: if two nodes that are d-separated by $\mathbf{S}$ become d-connected by $\mathbf{S} \cup \{X\}$, the MNS of such nodes exists and contains the parents of X.

The LDECC algorithm (Fig. 3), first finds the neighbors of X and MNS (Line 1, *FindNeighborsAndMNS* in Fig. 8b) by running PC locally like SD. In Line 1, $\text{mns}_X = \{\text{mns}_X(V) : V \in \mathbf{V} \setminus \text{Ne}^+(X)\}$). Next, using mns_X, we orient the children C of X that are part of UCs of the form $X \to C \leftarrow V$ (Line 3). LDECC then starts running CI tests in the same way as the PC algorithm would (excluding tests for X since we already know $\text{Ne}(X)$) (the for-loop in Line 5). Every time we detect a CI $A \perp\!\!\!\perp B|\mathbf{S}$, we check the following cases: (i) if $A, B \in \text{Ne}(X)$ and $X \notin \mathbf{S}$, then there must be a UC $A \to X \leftarrow B$ and so we mark A and B as parents (Lines 6, 7); (ii) if $A, B \in \text{Ne}(X)$ and $X \in \mathbf{S}$, then we mark A—X—B as a non-collider (Lines 9, 10); and (iii) **Eager Collider Check (ECC):** if $X \notin \mathbf{S}$ and $A \not\perp\!\!\!\perp B|\{\mathbf{S}, X\}$, then, by leveraging Prop. 3, we mark $\text{mns}_X(A)$ and $\text{mns}_X(B)$ as parents (Lines 14–17). Next, for each oriented parent, we use the non-colliders detected in Case (ii) to mark children (Line 20, 21). LDECC terminates if there are no unoriented neighbors. Lines 11–13 and Line 8 are needed to account for Meek's rule 3 and 4. The key departure from PC is Case (iii): with an ECC, unlike SD and PC, the skeleton is not used to orient parents.

Consider the running example of the DAG in Fig. 1a. We first find the local structure around X. Then we start running CI tests the same way as PC. After we run $A \perp\!\!\!\perp B|C$, we do an ECC and find $A \not\perp\!\!\!\perp B|\{C, X\}$. Since $\text{mns}_X(A) = \text{mns}_X(B) = \{W\}$, we mark W as a parent of X. Next, we find that $W \perp\!\!\!\perp M|X$ and mark W—X—M as a non-collider which lets us mark M as a child (because W is a parent). LDECC terminates as all neighbors of X are oriented (Fig. 1d) and the ATE set is computed by using $\{W\}$ as the backdoor adjustment set: $\Theta_{\text{LDECC}} = \{\beta_{Y|X.W}\} = \Theta^*$.

Theorem 1 (Correctness)**.** *Assuming the CFA holds and access to a CI oracle, we have $\Theta_{LDECC} \overset{set}{=} \Theta^*$.*

3.1 Comparison of computation requirements

In this section, we compare LDECC, SD, and PC based on the number of CI tests they perform. We show that neither of LDECC or SD uniformly dominates the other and characterize their performance in terms of properties of the true DAG.

To begin, we prove that, in the worst case, LDECC performs a polynomial (in $|\mathbf{V}|$) number of extra tests compared to PC when $|\text{Ne}(X)|$ is bounded.

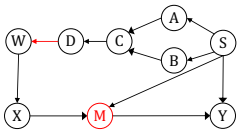
Proposition 4 (Number of tests: PC vs LDECC)**.** *Assume access to a CI oracle. Let the number of tests performed by PC and LDECC be T_{PC} and T_{LDECC}, respectively. Then we have*

$$T_{LDECC} \leq T_{PC} + \mathcal{O}(|\mathbf{V}|^2) + \mathcal{O}\left(|\mathbf{V}|^{|Ne(X)|}\right).$$

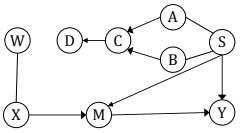
The $\mathcal{O}(|\mathbf{V}|^2)$ term accounts for the extra ECCs we might perform and the $\mathcal{O}\left(|\mathbf{V}|^{|\text{Ne}(X)|}\right)$ term accounts for discovering $\text{Ne}(X)$. In practice, we observe that the upper bound is quite loose: Even when $|\text{Ne}(X)|$ is large, PC also has to perform a large number of tests for X.

Proposition 5 (LDECC exponentially better)**.** *Assume access to a CI oracle. There exist graphs s.t.*

$$T_{LDECC} + \Theta(2^{|\mathbf{V}|}) \leq T_{PC}, \text{ and } T_{LDECC} + \Theta(2^{|\mathbf{V}|}) \leq T_{SD}.$$



(a) DAG with FF violations.



(b) CPDAG discovered by PC under FF violation.

Proof. Consider the class of graphs shown in Figure 4a where there is a clique of size N on the path from the UC $A \to C \leftarrow B$ to W. Both PC and SD will perform $\Theta(2^N)$ CI tests for nodes inside this clique. Thus, when $N \asymp |\mathbf{V}|$, SD will have exponential complexity. By contrast, LDECC unshields the UC and orients W as a parent via an ECC in $\mathcal{O}(|\mathbf{V}|^2)$ CI tests. □

Qualitatively, the result shows that if there is a dense region between a UC and the parent it orients, SD might perform poorly because it has to wade through this dense region to get to the UC. By contrast, LDECC can avoid this problem as it does not rely on the skeleton to orient the parents. However, LDECC does *not* uniformly dominate SD in terms of the number of tests. There are classes of causal graphs where LDECC performs an exponentially greater number of CI tests than SD:

Proposition 6 (SD exponentially better)**.** *Assume access to a CI oracle. Let* $\mathbf{U}$ *be the set of UCs in the graph and* $M = \max_{U \in \mathbf{U}} sep(U)$*. There exist graphs such that*

$$T_{SD} + \Theta(2^{M-1}) \leq T_{PC}, \; T_{SD} + \Theta(2^{M-1}) \leq T_{LDECC}.$$

Proof. Consider the class of graphs shown in Figure 4b with a UC with a separating set of size M and a clique of size N upstream of that UC. Since LDECC runs CI tests in the same order as PC, if $N \geq M$, LDECC performs $\Theta(2^{M-1})$ CI tests for nodes inside the clique (all tests of the form $V_i \perp\!\!\!\perp V_j | \mathbf{S}$ s.t. $\mathbf{S} \subseteq \{V_1, \ldots, V_N\} \setminus \{V_i, V_j\}$ and $|\mathbf{S}| < M$ will be run) before the UC is unshielded via the test $A \perp\!\!\!\perp B | \{S_1, \ldots, S_M\}$. By contrast, SD terminates before even getting to the clique, thus avoiding these tests. So if $M \asymp |\mathbf{V}|$, LDECC will perform exponentially more tests than SD. □

Qualitatively, this shows that if UCs have large separating sets and there are dense regions upstream of such UCs, LDECC might perform poorly whereas SD can avoid tests in these regions since it would not reach these dense regions.

Hybrid algorithm. In practice, we can implement a hybrid algorithm that runs both SD and LDECC in parallel and terminates when either algorithm terminates. The number of CI tests performed will be $T_{\text{hybrid}} \leq 2 \min \{T_{\text{SD}}, T_{\text{LDECC}}\}$. Thus this procedure broadens the class of graphs where local discovery can be performed efficiently because it will be subexponential if at least one of the algorithms is subexponential (whereas PC might be exponential).

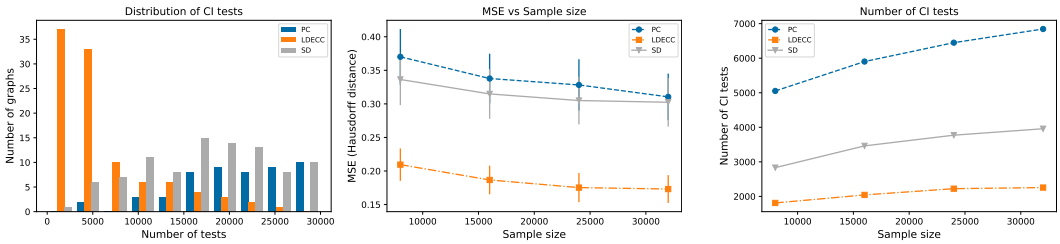
3.2 Comparison of faithfulness requirements

In this section, we show that SD and LDECC are robust to different faithfulness (FF) violations. The next two examples demonstrate that some FF violations affect SD but not LDECC and vice versa.

Example 1 (SD incorrect.)**.** *Consider the DAG in Fig. 5a and the following FF violation: Let's say that* $W \perp\!\!\!\perp D | M$ *(i.e., conditioning on the collider cancels out the* $W \leftarrow D$ *edge) and thus the edge* $W \leftarrow D$ *is incorrectly removed. Thus, both SD and PC will not mark* W *as a parent since the orientations will not be propagated (Fig. 5b). However, this violation does not affect LDECC since it uses ECCs to orient parents and so* W *will be oriented correctly.*

Example 2 (LDECC incorrect.)**.** *Consider the DAG in Fig. 5a and the following FF violation: Let's say that* $A \perp\!\!\!\perp X | M$ *and* $B \perp\!\!\!\perp X | M$*. Thus we might incorrectly detect* $mns_X(A) = mns_X(B) = \{M\}$ *causing the ECC to orient* M *as a parent. However, this FF violation does not impact SD as the skeleton and UCs are still detected correctly.*

The CFA is a controversial assumption [14] and many prior works attempt to test and recover from FF violations (usually by running additional CI tests) [9, 17, 11, 18, 6]. If LDECC and SD give different ATE set estimates, one strategy is to output a conservative ATE set. Consider the following procedure: (i) run both SD and LDECC to get Θ_{SD} and Θ_{LDECC}; (ii) If $\Theta_{\text{SD}} \neq \Theta_{\text{LDECC}}$ (i.e., there



(a) Number of tests with CI oracle. (b) MSE vs sample size. (c) Number of tests (finite samples).

Figure 6: Comparison of PC, LDECC, and SD on synthetic graphs.

is a contradiction), output the set $\Theta_{\text{union}} = \Theta_{\text{LDECC}} \cup \Theta_{\text{SD}}$. Then Θ_{union} will contain the true ATE value if either of the two algorithms is correct. Another strategy would be to check (via additional CI tests) if the assumptions for one of the algorithms is violated. As an illustration, consider the case in Example 2. We can run extra CI tests of the form $V \perp\!\!\!\perp X|\mathbf{S}$ for $V \in \{A, B\}$ and every possible subset $\mathbf{S} \subseteq \text{Ne}(X)$. We would find that $(A, B) \perp\!\!\!\perp X|W$. Thus $\{W\}$ would also be a valid MNS thereby violating uniqueness of MNS. In this case, we can output the ATE set from SD which would be correct if the FF requirements for SD hold. This also shows the complementary nature of the two algorithms: If there are detectable FF violations for one algorithm, we have an alternative ATE set from the other algorithm that we can output.

4 Experiments

We empirically compare PC, SD, and LDECC on random synthetic linear graphs with Gaussian errors. We show that LDECC leads to some MSE gains on synthetic graphs while performing fewer tests than SD on average. We also compare these algorithms on both linear and discrete semi-synthetic graphs from *bnlearn* [10] and observe that LDECC performs comparably to SD (see Appendix C.2).

We generate synthetic linear graphs with Gaussian errors, N_c covariates—non-descendants of X and Y with paths to both X and Y—and N_m mediators—nodes on some causal paths from X to Y. We generate edges between the different types of nodes with varying probabilities (see Appendix C.1 for the precise procedure). The ground-truth parameter is set identified, so we use Hausdorff distance as our evaluation metric:

$$\text{MSE}_{\text{Hausdorff}}(\{\widehat{\Theta}\}_{i=1}^{N}, \Theta^*) = \frac{1}{N}\sum_{i=1}^{N} \max\left\{ \sup_{u \in \widehat{\Theta}^{(i)}} \inf_{v \in \Theta^*} (u-v)^2, \sup_{v \in \Theta^*} \inf_{u \in \widehat{\Theta}} (u-v)^2 \right\},$$

where $\{\widehat{\Theta}\}_{i=1}^{N}$ is the estimated ATE set over N graphs and Θ^* is the ground-truth ATE set.

Results on synthetic data. We first compare the three algorithms based on the number of CI tests performed with access to a CI oracle. We generate 100 synthetic graphs with $N_c = 25$ and $N_m = 3$ and plot the distribution of CI tests (Fig. 6a). We see that LDECC performs substantially fewer tests than both SD and PC for ~ 65 graphs and has comparable performance on the other graphs. Next, we evaluate the methods on 250 synthetic graphs with $N_c = 20$ and $N_m = 3$, and for each graph, estimate Θ by resampling the data 5 times, at four different sample sizes. We use the Fisher-z's CI test and OLS for estimation. LDECC outperforms both SD and PC in terms of MSE at all sample sizes (Fig. 6b) while still performing fewer tests (Fig. 6c).

5 Conclusion

Broadening the landscape of local causal discovery, we propose a new algorithm that uses ECCs to orient parents. We show that it complements existing methods in terms computational and faithfulness requirements. Thus LDECC can be fruitfully combined with the existing class of local causal discovery algorithms that operate sequentially. In future work, we hope to come up with procedures to test and recover from a larger class of faithfulness violations. Extending local causal discovery methods to handle causal graphs with latent variables is also a promising direction.

References

[1] T. Gao and Q. Ji. Local causal discovery of direct causes and effects. *Advances in Neural Information Processing Systems*, 2015.

[2] C. Glymour, K. Zhang, and P. Spirtes. Review of causal discovery methods based on graphical models. *Frontiers in genetics*, 2019.

[3] A. Jaber, J. Zhang, and E. Bareinboim. Causal identification under markov equivalence: Completeness results. In *International Conference on Machine Learning*. PMLR, 2019.

[4] M. H. Maathuis and D. Colombo. A generalized back-door criterion. *The Annals of Statistics*, 2015.

[5] M. H. Maathuis, M. Kalisch, and P. Bühlmann. Estimating high-dimensional intervention effects from observational data. *The Annals of Statistics*, 2009.

[6] A. Marx, A. Gretton, and J. M. Mooij. A weaker faithfulness assumption based on triple interactions. In *Uncertainty in Artificial Intelligence*. PMLR, 2021.

[7] C. Meek. Causal inference and causal explanation with background knowledge. *arXiv preprint arXiv:1302.4972*, 2013.

[8] J. Pearl. *Causality*. Cambridge university press, 2009.

[9] J. Ramsey, J. Zhang, and P. L. Spirtes. Adjacency-faithfulness and conservative causal inference. *arXiv preprint arXiv:1206.6843*, 2012.

[10] M. Scutari. Learning bayesian networks with the bnlearn r package. *arXiv preprint arXiv:0908.3817*, 2009.

[11] P. Spirtes and J. Zhang. A uniformly consistent estimator of causal effects under the k-triangle-faithfulness assumption. *Statistical Science*, 2014.

[12] P. Spirtes, C. N. Glymour, R. Scheines, and D. Heckerman. *Causation, prediction, and search*. MIT press, 2000.

[13] J. Tian and J. Pearl. *A general identification condition for causal effects*. eScholarship, University of California, 2002.

[14] C. Uhler, G. Raskutti, P. Bühlmann, and B. Yu. Geometry of the faithfulness assumption in causal inference. *The Annals of Statistics*, 2013.

[15] C. Wang, Y. Zhou, Q. Zhao, and Z. Geng. Discovering and orienting the edges connected to a target variable in a dag via a sequential local learning approach. *Computational Statistics & Data Analysis*, 2014.

[16] J. Yin, Y. Zhou, C. Wang, P. He, C. Zheng, and Z. Geng. Partial orientation and local structural learning of causal networks for prediction. In *Causation and Prediction Challenge*. PMLR, 2008.

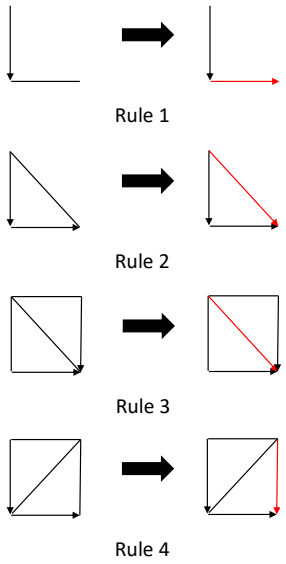
[17] J. Zhang and P. Spirtes. Detection of unfaithfulness and robust causal inference. *Minds and Machines*, 2008.

[18] J. Zhang and P. Spirtes. The three faces of faithfulness. *Synthese*, 2016.

[19] Y. Zhou, C. Wang, J. Yin, and Z. Geng. Discover local causal network around a target to a given depth. In *Causality: Objectives and Assessment*. PMLR, 2010.

Completely connected undirected graph $\mathcal{U}(\mathbf{V}, \mathbf{E})$;
$\forall A, B \in \mathbf{V}$ SepSet$[A][B] = \text{null}$;
$s = 0$;
while $\exists (A—B) \in \mathbf{E}$ *s.t.* $|Ne(A) \setminus \{B\}| \geq s$ **do**
  for $\mathbf{S} \subseteq Ne(A) \setminus \{B\}$ *s.t.* $|\mathbf{S}| = s$ **do**
    if $A \perp\!\!\!\perp B | \mathbf{S}$ **then**
      $\mathcal{U}$.removeEdge($A—B$);
      SepSet$[A][B] = \mathbf{S}$;
      break;
  end
  $s \leftarrow s + 1$;
end
for *every unshielded* $A—C—B$ **do**
  if $C \notin$ *SepSet*$[A][B]$ **then**
    Orient $A \rightarrow C \leftarrow B$;
end
ApplyMeekRules($\mathcal{U}$);
Output: $\mathcal{U}$, SepSet

(a) The PC algorithm [12, Sec. 5.4.2]



(b) Meek's orientation rules.

Figure 7: The PC algorithm and Meek's rules.

A Additional details on the PC and SD algorithms.

Definition 2 (CPDAG [4, Pg. 5])**.** *A set of DAGs that entail the same set of CIs form an MEC. This MEC can be uniquely represented using a CPDAG. A CPDAG is a graph with the same skeleton as each DAG in the MEC and contains both directed (→) and undirected (—) edges. A directed edge $A \rightarrow B$ means that the $A \rightarrow B$ is present in every DAG in the MEC. An undirected edge $A—B$ means that there is at least one DAG in the MEC with an $A \rightarrow B$ edge and at least one DAG with the $B \rightarrow A$ edge.*

The complete PC algorithm is given in Fig. 7a. PC starts with a fully connected skeleton and runs CI tests to remove edges. For each pair of nodes (A, B) that are adjacent in the skeleton, check CIs of size s—starting with $s = 0$ and then increasing it by one in each subsequent iteration—until the edge is removed or the number of nodes adjacent to both A and B is less than s. Once the skeleton is found, UCs are detected and then additional edges are oriented by repeatedly applying Meek's rules (Fig. 7b) until no additional edges can be oriented. With access to a CI oracle, the output of PC is a CPDAG encoding the MEC of the true DAG $\mathcal{G}^*$.

The *FindNeighbors* function used in the SD algorithm is given in Fig. 8a. For a given target node, *FindNeighbors* essentially runs the CI tests like the PC algorithm, but locally around the target node.

B Additional details for Section 3

B.1 Additional functions used by LDECC

The *FindNeighborsAndMNS* function used by the LDECC algorithm is given in Fig. 8b. We first begin by calling *FindNeighbors* (Fig. 8a) to get the neighbors of X. Then in Lines 3–11, we find the MNS for each node. In Lines 12–15, we verify that the MNS is valid. The following proposition shows that the MNS returned by this algorithm is correct.

Proposition 7 (MNS is correct)**.** *Assume access to a CI oracle. Then $mns_X(V)$ returned by the* FindNeighborsAndMNS *function (Fig. 8b) is correct.*

Proof. First, consider a node V for which there is no valid MNS. For all such nodes, Lines 12–15 will set the MNS to *invalid*. So going forward, we will focus on nodes with a valid MNS.

Input: Target node V.
1 Completely connected undirected graph $\mathcal{U}(\mathbf{V}, \mathbf{E})$;
2 $\forall A, B \in \mathbf{V},\ \text{SepSet}[A][B] = \text{null}$;
3 $s = 0$;
4 **while** $\exists (V—B) \in \mathbf{E}$ *s.t.* $|Ne(V) \setminus \{B\}| \geq s$ **do**
5 **for** $\mathbf{S} \subseteq Ne(V) \setminus \{B\}$ *s.t.* $|\mathbf{S}| = s$ **do**
6 **if** $A \perp\!\!\!\perp B | \mathbf{S}$ **then**
7 $\mathcal{U}$.removeEdge($A—B$);
8 $\text{SepSet}[V][B] = \mathbf{S}$;
9 **break**;
10 **end**
11 $s \leftarrow s + 1$;
12 **end**
Output: $\mathcal{U}, \text{SepSet}, \text{Ne}_{\mathcal{U}}(V)$;

(a) The *FindNeighbors* function.

Input: Treatment node X.
1 $\mathcal{U}, \text{SepSet}, \text{Ne}(X) = \text{FindNeighbors}(X)$;
2 $\forall V,\ \text{mns}_X(V) = \text{invalid}$;
3 **for** $V \in \mathbf{V} \setminus Ne^+(X)$ **do**
4 $\text{mns}_X(V) = \text{Ne}(X) \cap \text{SepSet}[X][V]$;
5 $\mathbf{R} = \text{SepSet}[X][V] \setminus \text{Ne}(X)$;
6 **while** $\mathbf{R} \neq \emptyset$ **do**
7 $A = \text{GetAndRemoveFirstItem}(\mathbf{R})$;
8 $\text{mns}_X(V) \leftarrow \text{mns}_X(V) \cup (\text{SepSet}[X][A] \cap \text{Ne}(X))$;
9 $\mathbf{R} \leftarrow \mathbf{R} \cup \text{SepSet}[X][A] \setminus \text{Ne}(X)$;
10 **end**
11 **end**
12 **for** $V \in \mathbf{V} \setminus Ne^+(X)$ **do**
13 **if** $V \not\perp\!\!\!\perp X | mns_X(V)$ **then**
14 $\text{mns}_X(V) = \text{invalid}$;
15 **end**
Output: $\text{Ne}(X), \text{mns}_X$;

(b) The *FindNeighborsAndMNS* function.

Figure 8: Functions used by SD and LDECC.

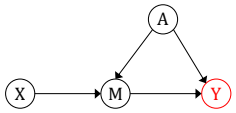


Figure 9: mns_X does not exist for node Y.

First, we start with the base case of a node V such that $\text{SepSet}[X][V] \subseteq \text{Ne}(X)$. Since the MNS is unique (Prop. 2), Line 4 will correctly set the MNS for this node and the *while* loop in Line 6 will not be entered.

Now consider a node V such that $\text{SepSet}[X][V] \setminus \text{Ne}(X) \neq \emptyset$. We first prove that $\text{SepSet}[X][V] \cap \text{Ne}(X) \subseteq \text{mns}_X(V)$. Consider any node $N \in \text{SepSet}[X][V] \cap \text{Ne}(X)$. Then there must be some path $X—N— \ldots —V$ and thus $N \in \text{mns}_X(V)$. If N was not part of $\text{mns}_X(V)$, then $\text{SepSet}[X][V] \setminus \{N\}$ would also have d-separated V from X. This cannot happen because PC runs CI tests in increasing order of size, $\text{SepSet}[X][V]$ is also minimal in that there is no subset $\mathbf{S}' \subset \text{SepSet}[X][V]$ that would d-separate X and V. This ensures the correctness of Line 4.

Let $\mathbf{R} = \text{SepSet}[X][V] \setminus \text{Ne}(X)$. Now we need to ensure that all paths from V to X through $\mathbf{R}$ get blocked. The while-loop in Line 6 ensures this. Consider the path $X—B— \ldots —A— \ldots —V$ such that $A \in \mathbf{R}$ and $B \notin \text{SepSet}[X][V]$. Here there are two possibilities: (i) $B \in \text{SepSet}[X][A]$ in which case it will be added to $\text{mns}_X(V)$ in Line 8; or (ii) $B \notin \text{SepSet}[X][A]$ in which case some other intermediate node on the path from A to X will be added to $\mathbf{R}$ in Line 9 and eventually in some subsequent iteration of the while-loop, B will be added to $\text{mns}_X(V)$.

Finally, we show that minimality is still maintained, i.e., that every node added by Line 8 must be a part of $\text{mns}_X(V)$. This has to be the case because the only way a node B can be added in Line 8 is if there is some path from V to X via B. Thus B has to be a part of $\text{mns}_X(V)$ to block this path. □

Example 3 (MNS does not exist.)**.** *Consider the graph in Fig. 9. For node $Y \in Desc(X)$, mns_X does not exist: There is no subset of $Ne(X)$ that d-separates Y from X.*

B.2 Omitted Proofs.

Proposition 1. *For any node $V \notin (Desc(X) \cup Ne^+(X))$, there exists a MNS and $mns_X(V) \subseteq Pa(X)$.*

Proof. Let $\mathbf{Q} = \text{Desc}(X) \cup \text{Ne}^+(X)$. Since $\text{Pa}(X)$ blocks all backdoor paths from X, for every $V \notin \mathbf{Q}$, we have $V \perp\!\!\!\perp X|\text{Pa}(X)$. Therefore, for every $V \notin \mathbf{Q}$, there exists some subset $\mathbf{S} \subseteq \text{Pa}(X)$ such that $V \perp\!\!\!\perp X|\mathbf{S}$. □

Proposition 2 (Uniqueness of MNS.)**.** *For nodes V s.t. $mns_X(V)$ exists, it is unique.*

Proof. We will prove this by contradiction. Consider a node V with two MNSs: $\mathbf{S}_1 \subseteq \text{Ne}(X)$ and $\mathbf{S}_2 \subseteq \text{Ne}(X)$ with $\mathbf{S}_1 \neq \mathbf{S}_2$.

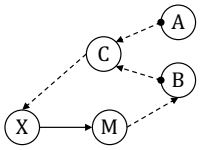
If $\mathbf{S}_1 \subset \mathbf{S}_2$ or $\mathbf{S}_2 \subset \mathbf{S}_1$, then minimality is violated. Hence, going forward we will only consider the case where $\mathbf{S}_1 \setminus \mathbf{S}_2 \neq \emptyset$ and $\mathbf{S}_2 \setminus \mathbf{S}_1 \neq \emptyset$.

Consider any node $A \in \mathbf{S}_1$ s.t. $A \notin \mathbf{S}_2$. For $\mathbf{S}_2$ to be a valid MNS, some nodes in $\mathbf{S}_2 \setminus \mathbf{S}_1$ must block all paths from V to X that contain A (this is because if this path were to be only be blocked by some nodes in $\mathbf{S}_1$, then minimality of $\mathbf{S}_1$ will be violated as $\mathbf{S}_1 \setminus \{A\}$ would also have been a valid MNS). This means that there is a path from V to X through some nodes in $\mathbf{S}_2 \setminus \mathbf{S}_1$ that cannot be blocked by $\mathbf{S}_1$ (else these nodes in $\mathbf{S}_1$ would have blocked the paths from V to A violating minimality of $\mathbf{S}_1$). This contradicts the fact that $\mathbf{S}_1$ is a valid MNS. Therefore, we must have $\mathbf{S}_1 = \mathbf{S}_2$. □

Proposition 3 (Eager Collider Check)**.** *For nodes $A, B \in \mathbf{V} \setminus Ne^+(X)$, any $\mathbf{S} \subset \mathbf{V} \setminus \{A, B, X\}$, if (i) $A \perp\!\!\!\perp B|\mathbf{S}$; and (ii) $A \not\perp\!\!\!\perp B|\mathbf{S} \cup \{X\}$; then $A, B \notin Desc(X)$ and $mns_X(A), mns_X(B) \subseteq Pa(X)$.*

Proof. We prove this by contradiction. Let's say there is a child M of X such that $M \in \text{mns}_X(B)$ or $M \in \text{mns}_X(A)$.

First, note that if Conditions (i, ii) hold, then there is a path of the form $A\bullet \to C$ and $B\bullet \to C$ and $C \to \ldots X$, where $\bullet$ means that there can be either an arrowhead or tail (i.e., there can be either a directed path $A \to \ldots \to C$ or a backdoor path $A \leftarrow \ldots \to C$ and likewise for B) with $C \notin \mathbf{S}$. W.l.o.g., let's say that $M \in \text{mns}_X(B)$ (the argument for node A follows similarly). Then there is a directed path from X to B through M (i.e., $X \to M \to \ldots \to B$). There cannot be a path $B \to \ldots \to M$ because then M will be a collider and therefore we will have $M \notin \text{mns}_X(B)$. These components are illustrated in the figure below:



We now show that $B \notin \text{Desc}(X)$ (the argument for node A is the same). There cannot be a directed path $B \to \ldots \to C$ because otherwise a cycle $B \to \ldots \to C \to \ldots \to X \to M \to \ldots \to B$ gets created.

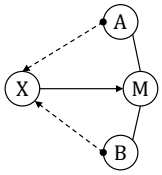
Thus the path from B to C must be of the form $B \leftarrow \ldots \to C$. Note that there is an active path between A and B through X ($A\bullet \to \ldots \to C \to \ldots \to X \to M \to \ldots \to B$). Since $A \perp\!\!\!\perp B|\mathbf{S}$, there are two possibilities: (i) $\mathbf{S}$ contains X to block this path which contradicts the definition of $\mathbf{S}$ (where $X \notin \mathbf{S}$); or (ii) $\mathbf{S}$ blocks all paths between A and X or between B and X in which case A and B cannot become dependent when additionally conditioned on X thereby violating Condition (ii).

Therefore, we have that $A, B \notin \text{Desc}(X)$ and by Prop. 1, $\text{mns}_X(A)$ and $\text{mns}_X(B)$ will be valid and only contain parents of X. □

Proof of correctness of LDECC under the CFA

Lemma 1. *Consider a DAG $\mathcal{G}(\mathbf{V}, \mathbf{E})$ with a node $X \in \mathbf{V}$. Let $A, B \in \mathbf{V}$ be two nodes such that A—$B \notin \mathbf{E}$ and $A, B \notin Desc(X)$ (i.e., both A, B are non-descendants of X). Then, for any $\mathbf{S} \subseteq \mathbf{V} \setminus \{A, B\}$ such that $A \perp\!\!\!\perp B|\mathbf{S}$ and $A \not\perp\!\!\!\perp B|\mathbf{S} \cup \{X\}$, we have $Ch(X) \cap \mathbf{S} = \emptyset$.*

Proof. We prove this by contradiction. Let's say that there is child $M \in \mathbf{S}$ (the relevant component of the graph is shown in the figure below).



Since $M \in \mathbf{S}$, there is a path A—$\ldots$—M—$\ldots$—B such that M is a non-collider on this path. Thus there has to be an arrowhead into at least one of A or B from M: i.e., there is at least one of $M \to \ldots \to B$ or $M \to \ldots \to A$. W.l.o.g., let's say there is an arrowhead into B. Then we would have $B \in \text{Desc}(X)$ (because of the path $X \to M \to \ldots \to B$) leading to a contradiction. □

Theorem 1 (Correctness)**.** *Assuming the CFA holds and access to a CI oracle, we have* $\Theta_{LDECC} \overset{set}{=} \Theta^*$.

Proof. We will prove the correctness of LDECC by showing that (i) every orientable neighbor of the treatment X will get oriented correctly by LDECC; and (ii) every unorientable neighbor of the treatment will remain unoriented.

Note that the function *FindNbrAndMNS* returns mns_X correctly (Prop. 7).

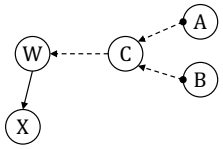
Parents are oriented correctly. In PC, edges get oriented using UCs and then propagating additional orientations via application of Meek's rules (Figure 7b).

The simplest case is where two parents form a UC at X. Consider parents W_1 and W_2 that get oriented because they form a UC $W_1 \to X \leftarrow W_2$. Lines 6,7 will mark W_1 and W_2 as parents.

We will now consider parents that get oriented due to each of the four Meek rules and show that LDECC orients parents for each of the four cases.

Meek Rule 1*:*

Consider a parent W that gets oriented due to the application of Meek's rule 1. This can only happen due to some UC $A \to C \leftarrow B$ from which these orientations have been propagated (relevant components of the graph are illustrated in the figure below).

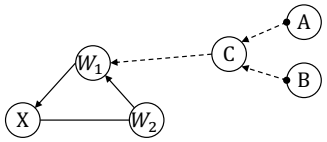


Thus there is a directed path $C \to \ldots \to W \to X$. This would mean that $W \in \text{mns}_X(A)$ and $W \in \text{mns}_X(B)$. Thus Line 17 will mark W as a parent.

Meek Rule 2*:*

Consider a parent W_2 that gets oriented due to the application of Meek's rule 2. In this case, we have an oriented path $W_2 \to W_1 \to X$ but the edge W_2—X is unoriented (and Meek Rule 2 must be applied to orient it).

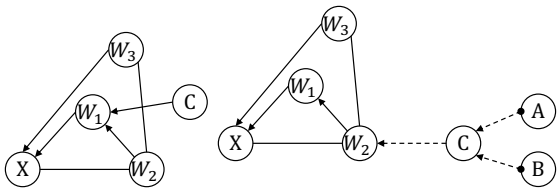
The first possibility is that the $X \leftarrow W_1$ was oriented due to some UC $A \to C \leftarrow B$ with a path $C \to \ldots \to W_1$ (relevant components of the graph are illustrated in the figure below):



In this case, there would be a collider at W_1: $C \to \ldots \to W_1 \leftarrow W_2$. Thus if $W_1 \in \text{mns}_X(A)$, then $W_2 \in \text{mns}_X(A)$ and thus LDECC will mark W_2 as a parent in Line 17.

The other possibility is that $X \leftarrow W_1$ was oriented due to a UC like $W_3 \to X \leftarrow W_1$ but there is an edge W_3—W_2 which causes the collider $W_2 \to X \leftarrow W_3$ to be shielded and due to this, the W_2—X remained unoriented. However, by definition of the Meek Rule, the edge $W_2 \to W_1$ is

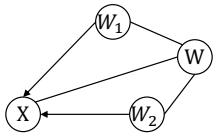
oriented. Thus, (i) either there is a UC of the form $W_2 \to W_1 \leftarrow C$; or (ii) there is a UC from which the $W_2 \to W_1$ orientation was propagated. The relevant components of the graph for these two cases are illustrated in the figures below.



For Case (i), $W_2 \in \text{mns}_X(C)$ and for Case (ii), $W_2 \in \text{mns}_X(A)$ and $W_2 \in \text{mns}_X(B)$. In both cases, LDECC will mark W_2 as a parent in Lines 17,16.

Meek Rule 3*:*

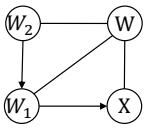
Consider a parent W that gets oriented due to the application of Meek's rule 3 (relevant component of the graph is shown in the figure below).



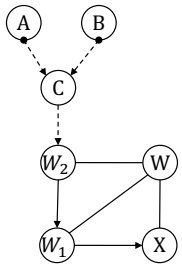
By definition of the Meek rule, W_1—W—W_2 is a non-collider (because if it were a collider, the edges would have been oriented since this triple is unshielded) and therefore for any $\mathbf{S} \subseteq \mathbf{V} \setminus \{W_1, W_2\}$ such that $W_1 \perp\!\!\!\perp W_2 | \mathbf{S}$, we have $W \in \mathbf{S}$. Thus Line 8 will mark W as a parent.

Meek Rule 4*:*

Consider a parent W that gets oriented due to the application of Meek's rule 4. The relevant component of the graph is shown in the figure below.

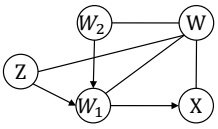


The first possibility is that the orientations $W_2 \to W_1 \to X$ were propagated from a UC $A \to C \leftarrow B$ with a path $C \to \ldots \to W_2$ (the relevant components of the graph are shown in the figure below).



In this case, due to the non-collider W_2—W—X (because if it were a collider, the edges would have been oriented since this triple is unshielded), we have $W \in \text{mns}_X(A)$ and thus W will be marked as a parent in Line 17.

The second possibility (similar to the Meek rule 2 case) is that $W_2 \to W_1$ was oriented due to a UC like $Z \to W_1 \leftarrow W_2$ but there is an edge Z—W which shields the $Z \to W_1$—W causing the W_1—W edge to remain unoriented (the relevant components of the graph are shown in the figure below).



In this case, we would have $W \in \text{mns}_X(Z)$ and thus W gets marked as a parent in Line 17.

Children are oriented correctly. We now similarly show that children of X get oriented correctly.

The simplest case is when there is a UC of the form $X \to M \leftarrow V$. We have $M \notin \text{mns}_X(V)$ and $M \not\perp\!\!\!\perp V | \text{mns}_X(V)$ and thus the function *OrientChildren* will mark M as a child.

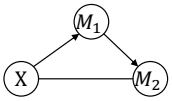
Now, we consider each Meek rule one at a time and show that LDECC will orient children for each rule.

Meek Rule 1*:*

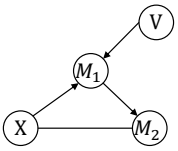
Consider a child M that gets oriented due to the application of Meek's rule 1. This can only happen if there is some parent W that gets oriented and W—X—M forms an unshielded non-collider. In this case, Line 21 will mark M as a child.

Meek Rule 2*:*

Consider a child M_2 that gets oriented due to the application of Meek's rule 2: there is an oriented path $X \to M_1 \to M_2$ but the X—M_2 edge is still unoriented (the relevant component of the graph is shown in the figure below).



One possibility is that there is UC of the form $V \to M_1 \to X$ which orients the $X \to M_1$ edge and V—M_1—M_2 is a non-collider which orients the $M_1 \to M_2$ edge (the relevant components of the graph are shown in the figure below). If the $X \to M_1$ was oriented but the X—M_2 was not, then there would have been a UC upstream of X that would have oriented $X \to M_1$ through the application of Meek rule 1. But this would also cause the $X \to M_2$ edge to be oriented.

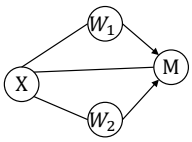


We have $M_2 \notin \text{mns}_X(V)$ and $M_2 \not\perp\!\!\!\perp V | \text{mns}_X(V)$ and thus the function *OrientChildren* will mark M as a child.

Similar to the Meek rule 2 case, there might be a UC of the form $M_1 \to M_2 \leftarrow Z$ that can orient $M_1 \to M_2$. However, for the $X \to M_1$ edge to remain unoriented, there must be an edge Z—X to shield the X—M_2—Z collider. If this happens, Meek rule 3 would apply (which we handle separately as shown next).

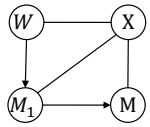
Meek Rule 3*:*

Consider a child M that gets oriented due to the application of Meek's rule 3 (the relevant component of the graph is shown in the figure below). By definition of the Meek rule, W_1—X—W_2 is a non-collider (because if it were a collider, the edges would have been oriented since this triple is unshielded) and since $W_1 \to M \leftarrow W_2$ forms a collider, we have $W_1 \not\perp\!\!\!\perp W_2 | \mathbf{S} \cup \{M\}$ for any $\mathbf{S}$ s.t. $W_1 \perp\!\!\!\perp W_2 | \mathbf{S}$. Thus Line 12 will mark M as a child.

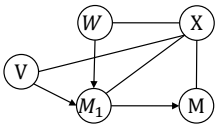


Meek Rule 4*:*

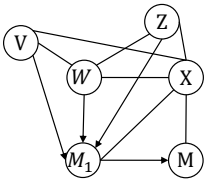
Consider a child M that gets oriented due to the application of Meek's rule 4 (the relevant component of the graph is shown in the figure below).



One possibility such that the $W \to M_1$ gets oriented leaving the edges W—X, X—M, and X—M_1 unoriented is if there is a UC of the form $V \to M_1 \leftarrow W$ where there is an edge V—X to shield the X—M_1 edge (the relevant component of the graph is shown in the figure below). Here the triple W—X—V must be a non-collider to keep the X—M edge unoriented (otherwise applying Meek rule 1 from the UC $W \to X \leftarrow V$ would orient $X \to M$). So for any $\mathbf{S}$ such that $V \perp\!\!\!\perp W|\mathbf{S}$, we must have $X \in \mathbf{S}$ and $V \not\perp\!\!\!\perp W|\mathbf{S} \cup \{M\}$. Therefore, Line 12 will mark M as a child.



The other possibility is that the $W \to M_1$ gets oriented due to Meek rule 3 (the relevant component of the graph is shown in the figure below). Here the Z—X and V—X must be present to keep the X—M_1 edge unoriented (because otherwise an unshielded collider would be created). Furthermore, the triple Z—X—V must be a non-collider in order to keep the X—M edge unoriented. (otherwise applying Meek rule 1 from the UC $Z \to X \leftarrow V$ would orient $X \to M$) So for any $\mathbf{S}$ such that $V \perp\!\!\!\perp Z|\mathbf{S}$, we must have $X \in \mathbf{S}$ and $V \not\perp\!\!\!\perp Z|\mathbf{S} \cup \{M\}$. Therefore, Line 12 will mark M as a child.

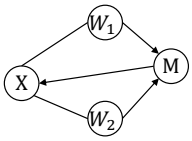


No spurious orientations. Now we prove that nodes are never oriented the wrong way by LDECC.

We show that *OrientChildren* will never orient a parent as a child. We prove this by contradiction. Let's say there is a parent W that is oriented as a child by *OrientChildren*. This can happen if there is a node V s.t. $W \notin \text{mns}_X(V)$ but $V \not\perp\!\!\!\perp W|\text{mns}_X(V)$. This can only happen if there is a path V—$\ldots$—W that is not blocked by $\text{mns}_X(V)$. In this case, there would be a path V—$\ldots$—$W \to X$ that is not blocked by $\text{mns}_X(V)$ and therefore we should have $W \in \text{mns}_X(V)$ which is a contradiction.

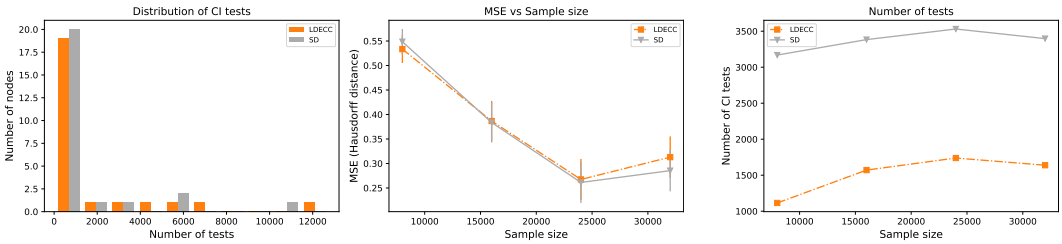
Line 7 can never mark a child as a parent since otherwise the CFA would be violated.

Line 12 will not mark a parent as a child. Consider a parent M that incorrectly gets marked as a child by Line 12. (the relevant component of the graph is shown in the figure below). For Line 12 to be reached, the if-condition in Line 9 must be *True*. This will happen if $W1$—X—$W2$ is a non-collider. Thus at least one of W_1 or W_2 is a child. W.l.o.g., let's assume that W_1 is a child. If that happens, a cycle gets created: $X \to W_1 \to M \to X$. Therefore M can never be oriented by Line 12.



Line 17 cannot mark a child as a parent because of the correctness of the ECC (Prop. 3).

Line 8 will not mark a child as a parent. Both A and B from Line 6 are parents of X. By Lemma 1, a child cannot d-separate two non-descendants nodes and thus the set $\mathbf{S}$ in Line 8 cannot contain a child.



(a) Number of tests with CI oracle. (b) MSE vs sample size. (c) Number of tests (finite samples).

Figure 10: Comparison of LDECC and SD on a semi-synthetic graph.

Line 21 will not add a child as a parent because otherwise the CFA would be violated. □

Proposition 4 (Number of tests: PC vs LDECC)**.** *Assume access to a CI oracle. Let the number of tests performed by PC and LDECC be T_{PC} and T_{LDECC}, respectively. Let the number of tests performed by LDECC in the* FindNbrsAndMNS *routine be $T^{(X)}_{LDECC}$. Then we have*

$$T_{LDECC} \leq T_{PC} + \mathcal{O}(|\mathbf{V}|^2) + T^{(X)}_{LDECC} \leq T_{PC} + \mathcal{O}(|\mathbf{V}|^2) + \mathcal{O}\left(|\mathbf{V}|^{|Ne(X)|}\right).$$

Proof. Compared to the PC algorithm, LDECC performs two main steps that are different: (i) It locally runs PC around X to find the neighbors of X and compute the MNS; and (ii) Run an ECC when an edge is removed.

The $\mathcal{O}\left(|\mathbf{V}|^{|\mathrm{Ne}(X)|}]\right)$ term upper bounds the complexity of step (i).

For any graph with $|\mathbf{V}|$ nodes, the number of edges is $|\mathbf{E}| \in \mathcal{O}(|\mathbf{V}|^2)$. The $\mathcal{O}(|\mathbf{V}|^2)$ term upper bounds the extra tests LDECC does for ECCs. □

C Experiments

C.1 More details on experiments with synthetic graphs

We generate synthetic linear graphs with Gaussian errors, N_c covariates—non-descendants of X and Y with paths to both X and Y—and N_m mediators—nodes on some causal paths from X to Y. We generate edges between the different types of nodes with varying probabilities: (i) We connect the covariates to the treatment with probability p_{cx}; (ii) We connect one covariate to another with probability p_{cc}; (iii) We connect the covariates to the outcome with probability p_{cy}; (iv) We connect the treatment to the mediators with probability p_{mx}; (v) We connect one mediator to another with probability p_{mm}; (vi) We connect the mediators to the outcome with probability p_{my}; (vi) We connect a mediators to a covariate with probability p_{cm}. For our experiments, we have used $p_{cx} = p_{cc} = p_{cy} = p_{mx} = p_{mm} = p_{my} = 0.1$ and $p_{cm} = 0.05$.

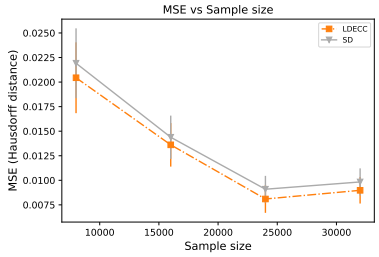
For each node V, we generate data using the following structural equation:

$$v := b_V^\top \mathrm{pa}(v) + \epsilon_V, \;\; \epsilon_V \sim \mathcal{N}(0, \sigma_V^2),$$

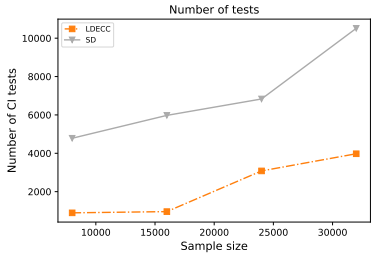
where v and $\mathrm{pa}(v)$ are the realized values of node V and its parents, respectively; $b_V^\top \in \mathbf{R}^{|\mathrm{Pa}(V)|}$ is the vector denoting the edge coefficients; and ϵ_V is an independently sampled noise term. Each element of b_V is sampled independently from a uniform distribution $U(0.1, 1)$ and σ_V^2 is sampled independently from a uniform distribution $U(0.3, 1)$.

C.2 Experiments with semi-synthetic graphs

We empirically compare PC, SD, and LDECC on both semi-synthetic linear and discrete graphs from *bnlearn*. We begin by comparing performance of PC, SD, and LDECC with a CI oracle on the *MAGIC-NIAB* linear graph. PC performs $\sim 1.472 \times 10^6$ tests. For SD and LDECC, we plot the distribution of CI tests on a subset of nodes: we only consider nodes where either algorithm performs < 20000 tests). Both algorithms perform comparably with SD doing slightly better (Fig. 10a). Next,

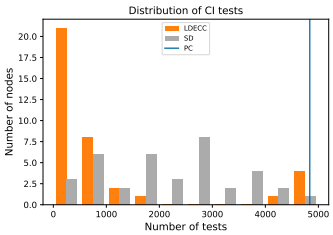


(a) MSE vs sample size.

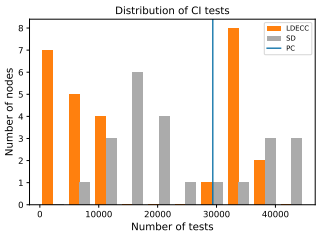


(b) Average number of tests (finite samples).

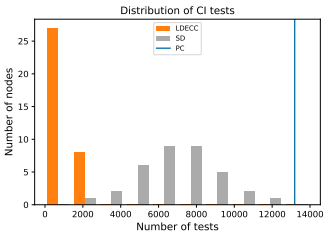
Figure 11: Comparison of LDECC and SD on the *MAGIC-IRRI* graph from *bnlearn*.



(a) *Alarm* graph.



(b) *Insurance* graph.



(c) *Mildew* graph.

Figure 12: Comparison of by LDECC and SD based on the number of CI tests (with a CI oracle) on discrete graphs from *bnlearn*.

we designate the nodes *G266* and *HT* as treatment and outcome, respectively. We see that in terms of MSE, which we compute over 200 runs, both are very similar at all four sample sizes (Fig. 10b) but LDECC performs fewer CI tests (Fig. 10c).

We also present results on the linear Gaussian *MAGIC-IRRI* graph from bnlearn (Fig. 11) for four different sample sizes. We designate the nodes *G2639* and *CHALK* as the treatment and outcome, respectively. For each of the four sample sizes, we sample data from the graph 200 times to compute the MSE and average number of tests. We see that in terms of MSE, both SD and LDECC perform comparably (Fig. 11a) and LDECC performs fewer CI tests (on average) than SD (Fig. 11b).

Finally, we compare PC, SD, and LDECC based on the number of CI tests (with access to a CI oracle) on three discrete graphs from *bnlearn*: *Alarm* (Fig. 12a), *Insurance* (Fig. 12b), and *Mildew* (Fig. 12c). We plot the distribution of tests for LDECC and SD by setting each node in the graph as the treatment. We can see that for both *Alarm* and *Insurance*, both SD and LDECC perform a comparable number of tests. On the *Mildew* graph, LDECC outperforms SD and performs strictly fewer CI tests.