A. Background

This section supplements §3.1 with more background on algorithmic information theory and standard causality. For a more thorough treatment on the foundational mathematics and formalism, please refer to Li et al. (2008) for algorithmic information theory, to Pearl (2009) for standard causality, and to Janzing & Schölkopf (2010); Peters et al. (2017) for algorithmic causality.

A.1. Notation

We denote with bolded uppercase monospace a computation graph at a single level of abstraction (e.g. the model of execution $\mathtt{\mathbf{G}}$, the model of credit assignment $\mathtt{\mathbf{C}}$). We denote with blackboard bold (e.g. the algorithmic causal model of learning $\mathbb{L}$) a computation graph that represents multiple levels of abstraction.

We denote binary strings that represent the data nodes in $\mathbb{L}$ with lower case (e.g. x or $\mathtt{f}$), where script (x) is used to emphasize that the string typically represents a variable and monospace ($\mathtt{f}$) is used to emphasize that the string typically represents a function. We use bolded lower case (e.g. $\mathbf{x}$, $\boldsymbol{\delta}$, $\mathtt{\mathbf{f}}$) to indicate a group of binary strings. We denote the function nodes in $\mathbb{L}$ (e.g., $\mathtt{APPLY}$, $\mathtt{UPDATE}$) with uppercase.

We write $\mathtt{f}(x) \to y$ to mean "a program $\mathtt{f}$ that takes a string x as input and produces a string y as output."

A.2. Background on algorithmic causality

The formalism of algorithmic causality derives from Janzing & Schölkopf (2010); Peters et al. (2017), which builds upon algorithmic statistics (Gács et al., 2001). Here we directly restate or paraphrase additional relevant definitions, postulates, and theorems from Janzing & Schölkopf (2010) and Gács et al. (2001).

A.2.1. Algorithmic information theory

Kolmogorov complexity Kolmogorov complexity (Solomonoff, 1960; 1964; Kolmogorov, 1965; Chaitin, 1966; 1975; Li et al., 2008) is a function $K : \{0,1\}^* \to \mathbb{N}$ from the binary strings $\{0,1\}^*$ to the natural numbers $\mathbb{N}$ that represents the amount of information contained in an object (represented by a binary string).

Definition 5 (Kolmogorov-complexity). *Given a universal Turing machine and universal programming language as reference, the* ***Kolmogorov complexity*** *$K(s)$ is the length of the shortest program that generates s. The* ***conditional Kolmogorov complexity*** *$K(y \mid x)$ of a string y given another string x is the length of the shortest program that generates y given x as input. Let the shortest program for string x be denoted as x^*. The* ***joint Kolomogorov complexity*** *$K(x, y)$ is defined as:*

$$K(x, y) \overset{+}{=} K(x) + K(x \mid y^*) \overset{+}{=} K(y) + K(y \mid x^*).$$

The **invariance theorem** (Kolmogorov, 1965) states that the Kolmogorov complexities of two strings written in two different universal languages differ only up to an additive constant. Therefore, we can assume any reference universal language for defining K (e.g. Python) and work with equalities ($\overset{+}{=}$) and inequalities ($\overset{+}{\geq}$, $\overset{+}{\leq}$) up to an additive constant.

Algorithmic mutual information Algorithmic mutual information I measures the amount of information two objects have in common:

Definition 6 (algorithmic mutual information). *The* ***algorithmic mutual information*** *of two binary strings x, y is*

$$I(x : y) \overset{+}{=} K(x) + K(y) - K(x, y).$$

The ***conditional algorithmic mutual information*** *of strings x, y given string z is*

$$I(x : y \mid z) \overset{+}{=} K(x \mid z) + K(y \mid z) - K(x, y \mid z)$$

We can intuitively think of $I(x : y \mid z)$ as, given z, the number of bits that can be saved when describing y knowing the shortest program that generated x. Then algorithmic independence is the property of two strings that says that the description of one cannot be further compressed given knowledge of the other.

Definition 7 (**algorithmic conditional independence**). *Given three strings x, y, z, x is **algorithmically conditionally independent** of y given z, denoted by $x \perp\!\!\!\perp y \mid z$, if the additional knowledge of y does not allow for stronger compression of x, given z. That is:*

$$x \perp\!\!\!\perp y \mid z \;\Leftrightarrow\; I\left(x : y \mid z\right) \overset{+}{=} 0.$$

We further define the joint conditional independence of strings $x_1, ..., x_n$ given strings $y_1, ..., y_m$ analogously:

Definition 8 (**algorithmic joint conditional independence**). *Strings $x_1, ..., x_n$ are **algorithmically jointly conditionally independent** given strings $y_1, ..., y_m$ if*

$$I\left(x_1, ..., x_n \mid y_1, ..., y_m\right) \overset{+}{=} 0, \tag{5}$$

which means that

$$K\left(x_1, ..., x_n \mid y_1, ..., y_m\right) \overset{+}{=} \sum_{i=1}^{n} K\left(x_i \mid y_1, ..., y_m\right), \tag{6}$$

meaning that, conditioned on knowing $y_1, ..., y_m$, the length of the joint description of $x_1, ..., x_n$ cannot be further compressed than sum of the lengths of the descriptions of the individual strings x_i. A proof of the equivalence between Eqs. 5 and 6 is given by the proof of Theorem 3 in Janzing & Schölkopf (2010).

We state as a lemma the following result from Gács et al. (2001, Corollary II.8) that states the mutual information of strings x and y cannot be increased by separately processing by functions f and g.

Lemma 6 (**information non-increase**). *Let f and g be computable programs. Then*

$$I\left(\mathtt{f}(x) : \mathtt{g}(y)\right) \overset{+}{\leq} I\left(x : y\right) + K(\mathtt{f}) + K(\mathtt{g}). \tag{7}$$

This intuitively makes sense: if $K(\mathtt{f})$ is constant with respect to x and $K(\mathtt{g})$ is constant with respect to y (i.e. $K(\mathtt{f}) \overset{+}{=} 0$ and $K(\mathtt{g}) \overset{+}{=} 0$), then mutual information cannot increase between x and y separately with $\mathtt{f}$ and $\mathtt{g}$. In particular, if x and y were independent to begin with (i.e. $I(x : y) \overset{+}{=} 0$), then $I\left(\mathtt{f}(x) : \mathtt{g}(y)\right) \overset{+}{=} 0$.

The following lemma states that the mutual information between two strings is constant if we condition on one of the strings.

Lemma 7 (**self-conditioning**). *For two strings x and y,*

$$I(x : y \mid y) \overset{+}{=} 0.$$

Proof.

$$\begin{aligned} I(x : y \mid y) &\overset{+}{=} K(x \mid y) + K(y \mid y) - K(x, y \mid y) \\ &\overset{+}{=} K(x \mid y) + K(y \mid y) - [K(x \mid y) + K(y \mid x^*, y)] \\ &\overset{+}{=} K(x \mid y) + 0 - [K(x \mid y) + 0] \\ &\overset{+}{=} K(x \mid y) - K(x \mid y) \\ &\overset{+}{=} 0. \end{aligned}$$

□

Terminology In this paper, we regard the following statements about a program $\mathtt{f}(x) \to y$ as equivalent:

- "$K(\mathtt{f}) \overset{+}{=} 0$."
- "$\mathtt{f}$ is an $O(1)$-length program."
- "$K(\mathtt{f})$ is constant with respect to x."

Note that $K(\mathtt{f}) \overset{+}{=} 0$ implies that $\mathtt{f}$ and x are algorithmically independent (i.e. $I(\mathtt{f} : x) \overset{+}{=} 0$) because

$$0 \overset{+}{\leq} I\left(x : \mathtt{f}\right) \overset{+}{=} K\left(\mathtt{f}\right) - K\left(\mathtt{f} \mid x^*\right) \overset{+}{\leq} K\left(\mathtt{f}\right) \overset{+}{=} 0.$$

A.2.2. Causality

Before we review algorithmic causality, we first review some key concepts in standard causality: structural causal models and d-separation.

The following definition defines standard causal models over random variables as Bayesian networks represented as directed acyclic graphs (DAG), analogous to the algorithmic version we presented in Def. 1.

Definition 9 (structural-causal-model). *A* ***structural causal model*** *(SCM) (Pearl, 1995; 2009) represents the assignment of random variable X as the output of a function, denoted by lowercase monospace (e.g. $\mathtt{f}$), that takes as input an independent noise variable N_X and the random variables $\{PA_X\}$ that represent the parents of X in a DAG:*

$$X := \mathtt{f}(\{PA_X\}, N_X). \tag{8}$$

Given the noise distributions $\mathbb{P}(N_X)$ for all variables X in the DAG, SCM entails a joint distribution $\mathbb{P}$ over all the variables in the DAG (Peters et al., 2017).

The graph-theoretic concept of d-separation is used for determining conditional independencies induced by a directed acyclic graph (see point 3 in Thm. 8):

Definition 10 (d-separation). *A path p in a DAG is said to be d-separated (or blocked) by a set of nodes Z if and only if*

1. *p contains a chain $i \to m \to j$ or fork $i \leftarrow m \to j$ such that the middle node m is in Z, or*
2. *p contains an inverted fork (or collider) $i \to m \leftarrow j$ such that the middle node m is not in Z and such that no descendant of m is in Z.*

*A set of nodes Z d-****separates*** *a set of nodes X from a set of nodes Y if and only if Z blocks every (possibly undirected) path from a node in X to a node in Y.*

A.2.3. Algorithmic causality

For convenience, we re-state the technical content from §3.1 here.

Definition 1 (computational graph). *Define a* ***computational graph*** *$\boldsymbol{G} = (\boldsymbol{x}, \boldsymbol{\mathtt{f}})$ as a directed acyclic factor graph (DAG) of variable nodes $\boldsymbol{x} = x_1, ..., x_N$ and function nodes $\boldsymbol{\mathtt{f}} = \mathtt{f}^1, ..., \mathtt{f}^N$. Let each x_j be computed by a program $\mathtt{f}^j$ with length $O(1)$ from its parents $\{pa_j\}$ and possibly an additional noise input n_j. Assume the noise n_j are jointly independent: $n_j \perp\!\!\!\perp \{n_{\neq j}\}$. Formally, $x_j := \mathtt{f}^j(\{pa_j\}, n_j)$, meaning that the Turing machine computes x_j from the input $\{pa_j\}, n_j$ using the additional program $\mathtt{f}^j$ and halts.*

Theorem 1 (algorithmic causal Markov condition). *Let $\{pa_j\}$ and $\{nd_j\}$ respectively represent concatenation of the parents and non-descendants (except itself) of x_j in a computational graph. Then $\forall x_j$, $x_j \perp\!\!\!\perp \{nd_j\} \mid \{pa_j\}$.*

Postulate 2 (faithfulness). *Given sets S, T, R of nodes in a computational graph, $I(S : T|R) \overset{+}{=} 0$ implies R d-separates S and T.*

The following theorem Janzing & Schölkopf (2010, Thm. 3) establishes the connection between the graph-theoretic concept of d-separation with condition algorithmic independence of the nodes of the graph.

Theorem 8 (equivalence of algorithmic Markov conditions). *Given the strings $x_1, ..., x_n$ and a computational graph, the following conditions are equivalent:*

1. ***Recursive form:*** *the joint complexity is given by the sum of complexities of each node x_j, given the optimal compression of its parents $\{pa_j\}$:*
$$K(x_1, ..., x_n) \overset{+}{=} \sum_{j=1}^{n} K(x_j|\{pa_j\}^*).$$
2. ***Local Markov Condition:*** *Every node x_j is independent of its non-descendants $\{nd_j\}$, given the optimal compression of its parents $\{pa_j\}$:*
$$I(x_j : nd_j|\{pa_j\}^*) \overset{+}{=} 0.$$

3. ***Global Markov Condition:*** *Given three sets S, T, R of nodes*

$$I(S : T|R^*) \overset{\pm}{=} 0$$

if R d-separates S and T.

Together, Thm. 1, Post. 2, and Thm. 8 imply that variable nodes in a computational graph are algorithmically independent if and only if they are d-separated in the computational graph.

B. Assumptions

This section states our assumptions for the results that we prove in §D. This paper analyzes learning algorithms from the perspective of algorithmic information theory, specifically algorithmic causality. To perform this analysis, we assume the following, and state our justifications for such assumptions:

1. The learning algorithm is implemented in on a universal Turing machine with a universal programming language.

   Justification: *This is a standard assumption in machine learning research that the machine learning algorithm can be implemented on a machine.*

2. Each initial parameter of the learnable functions $\boldsymbol{\mathtt{f}}$ is jointly algorithmically independent of the other initial parameters.

   Justification: *This is a standard assumption in machine learning research that the noise from the random number generator is given background knowledge (Janzing & Schölkopf, 2010, §2.3), thus allowing us to ignore possible dependencies among the parameters induced by the random number generator in developing our algorithms.*

3. The function nodes of ACML – `APPLY`, `UPDATE`, and the internal function nodes of Π – are $O(1)$-length programs.

   Justification: *Assuming `APPLY` (which encompasses the transition and reward functions of the MDP, see §6.1) is an $O(1)$-length program is reasonable because it is a standard assumption that they are fixed with respect to the activations and parameters of the learning algorithm. Assuming `UPDATE` is an $O(1)$-length program is a standard assumption in machine learning research that the source code that computes the update rule (e.g. a gradient descent step) is agnostic to the feedback signals (e.g. gradients) it takes as input. Assuming the internal function nodes of Π are $O(1)$-length programs is reasonable because these function nodes are operations in the programming language like addition, multiplication, etc that are agnostic to the activations and parameters of the learning algorithm.*

4. $\mathbb{L}$ is faithful. That is, any conditional independence among the data nodes ($\mathtt{f}$, x, δ, and the internal variable nodes of Π) in $\mathbb{L}$ is due to the causal structure of $\mathbb{L}$ rather than a non-generic setting of these data nodes.

   Justification: *Faithfulness has been justified for standard causal models (Meek, 1995). Deriving an algorithmic analog has been the subject of ongoing work (Lemeire & Janzing, 2013; Lemeire, 2016). For our work, a violation of faithfulness means that two nodes x, y in the computational graph have $I(x : y) \overset{\pm}{=} 0$ but are not d-separated in the computational graph. This would happen if x and y were tuned in such a way that makes one compressible given the other. Given assumption (2) above, the source of a violation of faithfulness must be the data experienced by the learning algorithm. Indeed, the data could be such that after learning certain parameters within $\boldsymbol{\mathtt{f}}$ may be conditionally independent given the training history, as suggested by Csordás et al. (2020); Filan et al. (2021); Watanabe (2019). However, as our focus is on theoretical results that hold regardless of the data distribution the learning algorithm is trained on, we consider the specific instances where the data does induce such faithfulness violations as "non-generic" and thus out of scope of the paper.*

C. Additional Theoretical Results

All modular credit assignment mechanisms must be factorized in the following way:

Theorem 9 (modular factorization). *The credit assignment mechanism $\Pi(\boldsymbol{\tau}, \boldsymbol{\mathtt{f}}) \rightarrow \boldsymbol{\delta}$ is modular if and only if*

$$K(\boldsymbol{\delta} \mid \boldsymbol{x}, \boldsymbol{\mathtt{f}}) \overset{\pm}{=} \sum_{t=1}^{T} K(\delta_t \mid \boldsymbol{x}, \boldsymbol{\mathtt{f}}) . \tag{9}$$

For modular credit assignment mechanisms, the complexity of computing feedback for the entire system is minimal because all redundant information among the gradients has been "squeezed out." This connection between simplicity and modularity is another way of understanding why if a credit assignment mechanisms were not modular it would be impossible for Π to modify a function without simultaneously modifying another, other than due to non-generic instances when δ_t has a simple description, i.e. $\delta_t = 0$, which, unless imposed, are not likely to hold over all iterations of learning.

D. Proofs

Given the assumptions stated in §B, we now provide the proofs for our theoretical results. We will prove Lemma 4 first. Together with the faithfulness postulate (Post. 2) and the equivalence of algorithmic Markov conditions (Thm. 8) we can prove algorithmic independence by inspecting the graph of $\mathbb{L}$ for d-separation.

D.1. Dynamic Modularity and the Algorithmic Causal Model of Learning

Lemma 4 (algorithmic causal model of learning). *Given a model of execution* $\boldsymbol{\mathtt{E}}$ *and of credit assignment* $\boldsymbol{\mathtt{C}}$*, define the **algorithmic causal model of learning** (ACML) as a dynamic computational graph* $\mathbb{L}$ *of the learning process. We assume* Π *has its own internal causal structure with internal variable and function nodes. The function nodes of* $\mathbb{L}$ *are* `APPLY`*,* `UPDATE`*, and the internal function nodes of* Π*, all with length* $O(1)$*. The variable nodes of* $\mathbb{L}$ *are* x*,* $\mathtt{f}$*,* δ*, and internal variable nodes of* Π*. Then these variable nodes satisfy the algorithmic causal Markov condition with respect to* $\mathbb{L}$ *for all steps of credit assignment.*

Proof. $\mathbb{L}$ is a well-defined computational graph and so by Thm. 1 it satisfies the algorithmic causal Markov condition. □

Remark. By Lemma 6, if a set of data nodes in $\mathbb{L}$ are independent, then processing them separately with factor nodes of $\mathbb{L}$ will maintain this independence. For example, given that the `UPDATE` operation is applied in separately for each pair $(\mathtt{f}^k, \sum_t \delta_t^k)$ to produce a corresponding $\mathtt{f}^{k\prime}$, then if $(\mathtt{f}^k, \sum_t \delta_t^k)$ were independent of $(\mathtt{f}^j, \sum_t \delta_t^j)$ before applying `UPDATE`, then $\mathtt{f}^{k\prime}$ would be independent of $\mathtt{f}^{j\prime}$ after applying `UPDATE`.

Theorem 3 (modular credit assignment). *Dynamic modularity is enforced at learning iteration* i *if and only if static modularity holds at iteration* $i = 0$ *and the credit assignment mechanism satisfies the modularity constraint.*

Proof. We will prove by induction on i. The inductive step will make use of the equivalence between d-separation and conditional independence.

Base case: $i = 1$. There is no training history, so static modularity is equivalent to dynamic modularity.

Inductive hypothesis: Assuming that dynamic modularity holds if and only if static modularity holds at $i = 0$ and modular credit assignment holds for learning iteration $i - 1$, dynamic modularity holds if and only if static modularity and modular credit assignment hold for learning iteration i.

Inductive step: The modularity constraint states

$$I\left(\delta_1, ..., \delta_M \mid \mathbf{x}_i, \boldsymbol{\mathtt{f}}_i\right) \overset{+}{=} 0.$$

Dynamic modularity at iteration $i - 1$ states that

$$\forall k \neq j, \quad I\left(\mathtt{f}^{k,i} : \mathtt{f}^{j,i} \;\middle|\; \mathbf{x}_{i-1}, \boldsymbol{\mathtt{f}}_{i-1}\right) \overset{+}{=} 0.$$

These two above statements correspond to the computational graph in Fig. 8. Note that by Def. 8, disjoint subsets of $\delta_1, ..., \delta_T$ also have have zero mutual information up to an additive constant. Letting these subsets be $\sum_t \delta_t^k$ where k is the index of function $\mathtt{f}^k$ in $\boldsymbol{\mathtt{f}}$, then

$$I\left(\sum_t \delta_t^1, ..., \sum_t \delta_t^N \;\middle|\; \mathbf{x}_i, \boldsymbol{\mathtt{f}}_i\right) \overset{+}{=} 0. \tag{10}$$

Then, as we can see by direct inspection in Fig. 8, $\mathtt{f}^{k_i}$ and $\mathtt{f}^{j_i}$ are d-separated by $(\mathbf{x}_i, \boldsymbol{\mathtt{f}}_i)$, which is equivalent to saying that dynamic modularity holds for iteration i. □

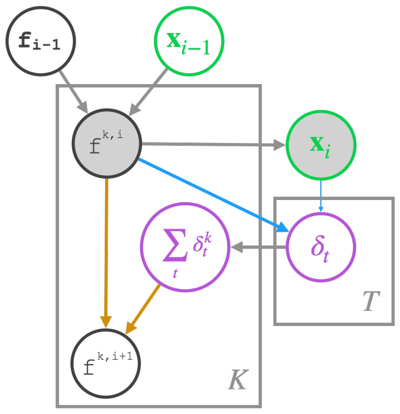

Figure 8: This figure shows the computation graph of $\mathbb{L}$ across one credit assignment update. Inputs to the credit assignment mechanism are shaded. A modular credit assignment mechanism (shown with blue edges) is equivalent to showing the gradients δ_t as conditionally independent, as shown by the plate notation labeled with T. Dynamic modularity at iteration $i-1$ is equivalent to showing that the functions $\mathtt{f}^{k,i}$ are inside the plate labeled with N. Then because the `UPDATE` operation, shown with yellow edges, operates only within the plate labeled with N, the updated functions $\mathtt{f}^{k,i+1}$ are also conditionally independent given $(\mathbf{x}, \boldsymbol{\mathtt{f}})$.

Theorem 5 (modularity criterion). *If $\mathbb{L}$ is faithful, the modularity constraint holds if and only if for all i, outputs δ_t and $\delta_{\neq t}$ of Π are d-separated by its inputs $\boldsymbol{x}$ and $\boldsymbol{\mathtt{f}}$.*

Proof. The forward direction holds by the equivalence of algorithmic causal Markov conditions (Thm. 8), and the backward direction holds by the faithfulness assumption. □

Theorem 9 (modular factorization). *The credit assignment mechanism $\Pi(\boldsymbol{\tau}, \boldsymbol{\mathtt{f}}) \to \boldsymbol{\delta}$ is modular if and only if*

$$K\left(\boldsymbol{\delta} \mid \boldsymbol{x}, \boldsymbol{\mathtt{f}}\right) \overset{+}{=} \sum_{t=1}^{T} K\left(\delta_t \mid \boldsymbol{x}, \boldsymbol{\mathtt{f}}\right). \tag{11}$$

Proof. The proof comes from the definition of algorithmic mutual information.

$$K\left(\boldsymbol{\delta} \mid \mathbf{x}, \boldsymbol{\mathtt{f}}\right) \overset{+}{=} \sum_{t=1}^{M} K\left(\delta_t \mid \mathbf{x}, \boldsymbol{\mathtt{f}}\right) \tag{12}$$

$$\sum_{t=1}^{M} K\left(\delta_t \mid \mathbf{x}, \boldsymbol{\mathtt{f}}\right) - K\left(\boldsymbol{\delta} \mid \mathbf{x}, \boldsymbol{\mathtt{f}}\right) \overset{+}{=} 0 \tag{13}$$

$$I\left(\delta_1, ..., \delta_M \mid \mathbf{x}, \boldsymbol{\mathtt{f}}\right) \overset{+}{=} 0. \tag{14}$$

□

D.2. Modularity in Reinforcement Learning

In the following, we sometimes use b_t instead of b_s to reduce clutter.

Corollary 5.1 (policy gradient). *All policy gradient methods do not satisfy the modularity criterion.*

Proof. It suffices to identify a single shared hidden variable that renders $\delta_1, ..., \delta_T$ not d-separated. Computing the policy gradient includes the log probability of the policy as one of its terms. Computing this log probability for any action involves the same normalization constant $\sum_k b^k$. This normalization constant is a hidden variable that renders $\delta_1, ..., \delta_T$ not d-separated, as shown in Fig. 9. □

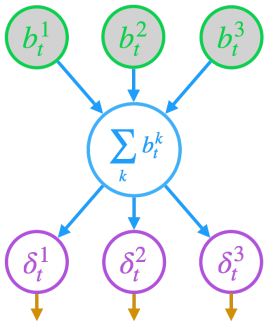

Figure 9: This figure shows part of the computational graph within Π for policy gradient methods. Conditioning on $\mathbf{x}$ implies we condition on the lightly shaded nodes. $\sum_k b_t^k$ is the shared hidden variable that renders $\delta_1, ..., \delta_T$ not d-separated.

Corollary 5.2 (n-step TD). *All TD($n > 1$) methods do not satisfy the modularity criterion.*

Proof. It suffices to identify a single shared hidden variable that renders $\delta_1, ..., \delta_T$ not d-separated. TD($n > 1$) methods include a sum of estimated returns or advantages at different steps of the decision sequence that is shared among multiple δ_t's. This sum is the hidden variable that renders $\delta_1, ..., \delta_T$ not d-separated, as shown in Fig. 10. □

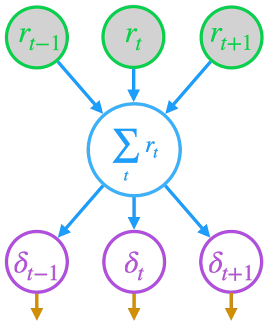

Figure 10: This figure shows part of the computational graph within Π for TD($n > 1$) methods. Conditioning on $\mathbf{x}$ implies we condition on the lightly shaded nodes. $\sum_t r_t$ is the shared hidden variable that renders $\delta_1, ..., \delta_T$ not d-separated.

Corollary 5.3 (single-step TD). *TD(0) methods satisfy the modularity criterion for acyclic $\mathbf{x}$.*

Proof. If the decision mechanism $\mathtt{f}^k$ were selected (i.e. $w_t^k = 1$) at step i, TD(0) methods produce, for some function g, gradients as $\delta_t^k := g(b_t^k, s_t, s_{t+1}, r_t, \mathbf{f})$. Otherwise, $\delta_t^k := 0$. The only hidden variable is $[\max_j b^j_{s_{t+1}}]$, and for acyclic $\mathbf{x}$ there is only one state s_t in $\mathbf{x}$ that transitions into s_{t+1}. Therefore the hidden variable is unique to each of $\delta_1, ..., \delta_t$, so $\delta_1, ..., \delta_t$ remain d-separated, as shown in Fig. 11. □

Corollary 2.1 (tabular). *In the tabular setting, Thm. 3 holds for Q-learning, SARSA, and CVS.*

Proof. In the tabular setting, decision mechanisms are columns of the Q-table corresponding to each action. These columns do not share parameters, so static modularity holds. Then because Q-learning, SARSA, and CVS are TD(0) methods, by Corollary 5.3, their credit assignment mechanisms are modular. Therefore Thm. 3 holds. □

Corollary 2.2 (function approximation). *In the function approximation setting, Thm. 3 holds for TD(0) methods whose decision mechanisms do not share parameters.*

Proof. The decision mechanisms of CVS do not share weights, so static modularity holds. By Corollary 5.3 its credit assignment mechanism is modular. Therefore Thm. 3 holds. □

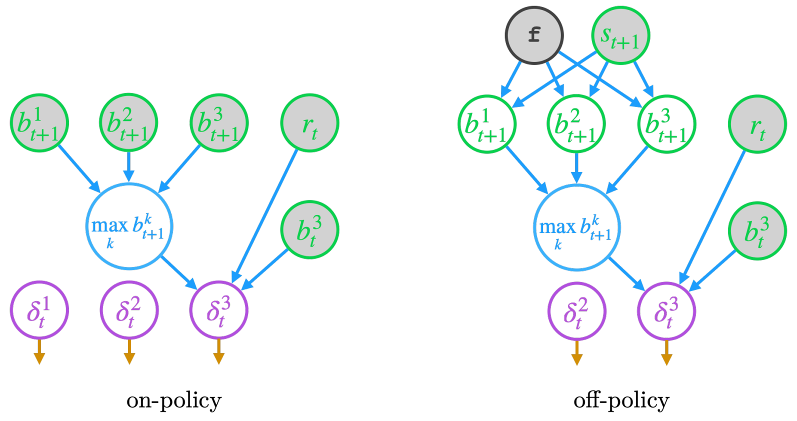

Figure 11: This figure shows part of the computational graph within Π for on-policy and off-policy TD(0) methods. Conditioning on $(\mathbf{x}, \mathbf{f})$ implies we condition on the lightly shaded nodes. For on-policy methods such as CVS and SARSA, the hidden variable would be $\max_k b^k_{t+1}$ for CVS and the bid corresponding to the decision mechanism that was sampled through ε-greedy for SARSA. The figure shows $\max_k b^k_{t+1}$ for concreteness. For off-policy methods such as Q-learning, the bids b_{t+1} are computed from s_{t+1} and $\mathbf{f}$, both of which we condition on. In both cases, the hidden variable is only parent to one of the δ_t's, and thus the $\delta_1, ..., \delta_T$ remain d-separated.

E. Simulation Details

We implemented our simulations using the PyTorch library (Paszke et al., 2019).

E.1. Implementation Details

The underlying PPO (Schulman et al., 2017) implementation used for CVS, PPO, and PPOF used a policy learning rate of 4×10^{-5}, a value function learning rate of 5×10^{-3}, a clipping ratio of 0.2, a GAE (Schulman et al., 2015) parameter of 0.95, a discount factor of 0.99, entropy coefficient of 0.1, and the Adam (Kingma & Ba, 2014) optimizer. For all algorithms, the policy and value functions for our algorithms were implemented as fully connected neural networks that used two hidden layers of dimension 20, with a ReLU activation. All algorithms performed a PPO update every 4096 samples with a minibatch size of 256.

E.2. Training Details

All learning curves are plotted from ten random seeds, with a different learning algorithm represented by a different hue. The dark line represents the mean over the seeds. The error bars represent one standard deviation above and below the mean.

Our protocol for transfer is as follows. A transfer problem is defined by a (training, transfer) task pair, where the initial network parameters for the transfer task are the network parameters learned the training task for H samples. In our simulations, we set H to 10^7 because that was about double the number of samples for all algorithms to visually converge on the training task for all seeds. To calculate the relative sample efficiency of CVS over PPO and PPOF (e.g. 13.9x and 6.1x respectively in the bottom-up right corner of Fig. 5), we set the criterion of convergence as the number of samples after which the return deviates by no more than $\varepsilon = 0.01$ from the optimal return for 30 epochs of training, where each epoch of training trains on 4096 samples.

E.3. Environment Details

The environment for our experiments shown in Figs. 4 and 5 are represented as discrete-state, discrete-action MDPs. Each state is represented by a binary-valued vector.

The structure of the MDP can best be explained via an analogy to a room navigation task, which we will explain in the context of the $A \to B \to C$ task in the *linear chain* topology. In this task, there are four rooms, room 0, room 1, room 2, and room 3. Room 0 has two doors, labeled A and F, that lead to room 1. Room 1 has two doors, labeled B and E, that lead to room 2. Room 2 has two doors, labeled C and D. Doors are unlocked by keys. The state representation is a concatenation of two one-hot vectors. The first one-hot vector is of length four; the "1" indicates the room id. The second one-hot vector is of length six; the "1" indicates the presence of a key for door A, B, C, D, E, or F. Only one key is present in a room at any given time. If the agent goes through the door corresponding to the key present in the room, then the agent transitions into the next room; otherwise the agent stays in the same room. In the last room, if the agent opens the door corresponding to the key that is present in the room, then the agent receives a reward of 1. All other actions in every other state receive a reward of 0. Therefore the agent only gets a positive reward if it opens the correct sequence of doors. For all of our experiments, the optimal policy is acyclic, but a suboptimal decision sequence could contain cycles.

Therefore, for the training task in the *linear chain* topology where the optimal solution is $A \to B \to C$, the optimal sequence of states are

```
[1, 0, 0, 0 ; 1, 0, 0, 0, 0, 0]  # room 0 with key for A
[0, 1, 0, 0 ; 0, 1, 0, 0, 0, 0]  # room 1 with key for B
[0, 0, 1, 0 ; 0, 0, 1, 0, 0, 0]  # room 2 with key for C.
```

For the transfer task whose optimal solution is $A \to B \to D$, the optimal sequence of states are

```
[1, 0, 0, 0 ; 1, 0, 0, 0, 0, 0]  # room 0 with key for A
[0, 1, 0, 0 ; 0, 1, 0, 0, 0, 0]  # room 1 with key for B
[0, 0, 1, 0 ; 0, 0, 0, 1, 0, 0]  # room 2 with key for D.
```

Whereas for the *linear chain* topology the length of the optimal solution is three actions, for the *common ancestor* and *common descendant* topologies this length is two actions. *Common ancestor* and *common descendant* are multi-task problems. As a concrete example, the training task for *common ancestor* is a mixture of two tasks, one whose optimal solution is $A \to B$ and one whose optimal solution is $A \to C$. Following the analogy to room navigation, this task is set up such that after having gone through door A, half the time there is a key for door B and half there is a key for door C.

E.4. Computing Details

For our experiments, we used the following machines:

- AWS: c5d.18xlarge instance
- Azure: Standard D64as_v4 (64 vcpus, 256 GiB memory), 50GiB Standard SD attached.

The average runtime training on 10^7 samples was three hours for PPO and PPOF and 6 hours for CVS.