Neural Algorithmic Reasoning with Causal Regularisation

Beatrice Bevilacqua [1]   **Kyriacos Nikiforou** [* 2]   **Borja Ibarz** [* 2]   **Ioana Bica** [2]   **Michela Paganini** [2]   **Charles Blundell** [2]
Jovana Mitrovic [† 2]   **Petar Veličković** [† 2]

Abstract

Recent work on neural algorithmic reasoning has investigated the reasoning capabilities of neural networks, effectively demonstrating they can learn to execute classical algorithms on unseen data coming from the train distribution. However, the performance of existing neural reasoners significantly degrades on out-of-distribution (OOD) test data, where inputs have larger sizes. In this work, we make an important observation: there are many *different* inputs for which an algorithm will perform certain intermediate computations *identically*. This insight allows us to develop data augmentation procedures that, given an algorithm's intermediate trajectory, produce inputs for which the target algorithm would have *exactly* the same next trajectory step. We ensure invariance in the next-step prediction across such inputs, by employing a self-supervised objective derived by our observation, formalised in a causal graph. We prove that the resulting method, which we call Hint-ReLIC, improves the OOD generalisation capabilities of the reasoner. We evaluate our method on the CLRS algorithmic reasoning benchmark, where we show up to $3\times$ improvements on the OOD test data.

1. Introduction

Recent works advocate for building neural networks that can reason (Xu et al., 2020; 2021; Veličković & Blundell, 2021; Veličković et al., 2022a). Therein, it is posited that combining the robustness of algorithms with the flexibility of neural networks can help us accelerate progress towards models that can tackle a wide range of tasks with real world impact (Davies et al., 2021; Deac et al., 2021; Veličković

et al., 2022b; Bansal et al., 2022; Beurer-Kellner et al., 2022). The rationale is that, if a model learns how to reason, or learns to execute an algorithm, it should be able to apply that reasoning, or algorithm, to a completely novel problem, even in a different domain. Specifically, if a model has learnt an algorithm, it should be gracefully applicable on out-of-distribution (OOD) examples, which are substantially different from the examples in the training set, and return correct outputs for them. This is because an algorithm—and reasoning in general—is a sequential, step-by-step process, where a simple decision is made in each step based on outputs of the previous computation.

Prior work (Diao & Loynd, 2022; Dudzik & Veličković, 2022; Ibarz et al., 2022; Mahdavi et al., 2022) has explored this setup, using the CLRS-30 benchmark (Veličković et al., 2022a), and showed that while many algorithmic tasks can be learned by Graph Neural Network (GNN) processors in a way that generalises to larger problem instances, there are still several algorithms where this could not be achieved.

Importantly, CLRS-30 also provides ground-truth *hints* for every algorithm. Hints correspond to the state of different variables employed to solve the algorithm (e.g. positions, pointers, colouring of nodes) along its trace. Such hints can optionally be used during training, but are not available during evaluation. In previous work, they have mainly been used as auxiliary targets together with the algorithm output. The prevailing hypothesis is that gradients coming from predicting these additional relevant signals will help constrain the representations in the neural algorithmic executor and prevent overfitting. Predicted hints can also be optionally fed back into the model to provide additional context and aid their prediction at the next step.

In practice, while utilising hints in this way does lead to models that *follow* the algorithmic trajectory better, they have had a less substantial impact on the accuracy of the predicted *final output*. This is likely due to the advent of powerful strategies such as recall (Bansal et al., 2022), wherein the input is fed back to the model at every intermediate step, constantly "reminding" the model of the problem that needs to be solved. The positive effect of recall on the final output accuracy has been observed on many occasions (Mahdavi et al., 2022), and outweighs the contribution from directly

---
[*]Equal contribution [†]Equal Advising [1]Purdue University [2]DeepMind. Work done while Beatrice Bevilacqua was at DeepMind. Correspondence to: Beatrice Bevilacqua <bbevilac@purdue.edu>.
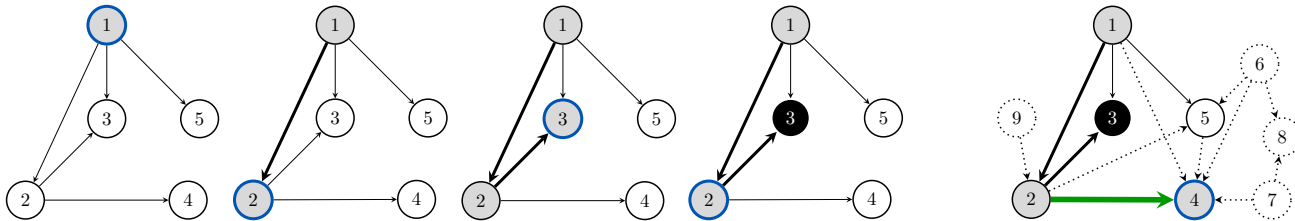
Figure 1. An illustration of the key observation of our work, on the depth-first search (DFS) algorithm as implemented in CLRS-30 (Veličković et al., 2022a). On the left, the first four steps of DFS are visualised. At each step, DFS explores the unvisited neighbour with the smallest index, and backtracks if no unexplored neighbours exist. The next computational step—*assigning* 2 *as the parent of* 4—is bound to happen, even under many *transformations* of this graph. For example, if we were to insert new (dashed) nodes and edges into the graph, this step would still proceed as expected. Capturing this computational invariance property is the essence of our paper.

predicting hints and feeding them back.

In this work, we propose a method, namely Hint-ReLIC, that decisively demonstrates an advantage to using hints. We base our work on the observation that there are many different inputs for which an algorithm will make *identical* computations at a certain step (Figure 1). For example, applying the bubble sort algorithm from the left on $[2, 1, 3]$ or $[2, 1, \underline{5}, 3]$ will result in the same first step computation: a comparison of 2 and 1, followed by swapping them. Conversely, the first step of execution would be different for inputs $[2, 1, 3]$ and $[2, \underline{5}, 1, 3]$; the latter input would trigger a comparison of 2 and 5 without swapping them. This observation allows us to move beyond the conventional way of using hints, i.e. autoregressively predicting them (Veličković et al., 2022a). Instead, we design a novel way that learns more informative representations that enable the networks to more faithfully execute algorithms. Specifically, we learn representations that are similar for inputs that result in identical intermediate computation. First, we design a causal graph in order to formally model an algorithmic execution trajectory. Based on this, we derive a self-supervised objective for learning hint representations that are invariant across inputs having the same computational step. Moreover, we prove that this procedure will result in stronger causally-invariant representations.

Contributions. Our three key contributions are as follows:

1. We design a *causal graph* capturing the observation that the execution of an algorithm at a certain step is determined only by a subset of the input;

2. Motivated by our causal graph, we present a *self-supervised objective* to learn representations that are *provably* invariant to changes in the input subset that does not affect the computational step;

3. We test our model, dubbed Hint-ReLIC, on the CLRS-30 algorithmic reasoning benchmark (Veličković et al.,

2022a), demonstrating a *significant improvement* in out-of-distribution generalisation over the recently published state-of-the-art (Ibarz et al., 2022).

2. Related Work

GNNs and invariance to size shifts. Graph Neural Networks (GNNs) constitute a popular class of methods for learning representations of graph data, and they have been successfully applied to solve a variety of problems. We refer the reader to Bronstein et al. (2021); Jegelka (2022) for a thorough understanding of GNN concepts. While GNNs are designed to work on graphs of any size, recent work has empirically shown poor size-generalisation capabilities of standard methods, mainly in the context of molecular modeling (Gasteiger et al., 2022), graph property prediction (Corso et al., 2020), and in executing specific graph algorithms (Veličković et al., 2020; Joshi et al., 2020). A theoretical study of failure cases has been recently provided in Xu et al. (2021), with a focus on a geometrical interpretation of OOD generalisation. In order to learn models performing equally well in- and out-of-distribution, Bevilacqua et al. (2021); Chen et al. (2022); Zhou et al. (2022) designed ad-hoc solutions satisfying assumed causal assumptions. However, these models are not applicable to our setting, as the assumptions on our data generation process are significantly different. With the same motivation, Buffelli et al. (2022) introduced a regularisation strategy to improve generalisation to larger sizes, while Yehudai et al. (2021) proposed a semi-supervised and a self-supervised objective that assume access to the test distribution. However, these models are not designed to work on algorithmic data, where OOD generalisation is still underexplored.

Neural Algorithmic Reasoning. In order to learn to execute algorithmic tasks, a neural network must include a *recurrent* component simulating the individual algorithmic steps. This component is applied a variable number of times, as required by the size of the input and the problem at hand.

The recurrent component can be an LSTM (Gers & Schmidhuber, 2001), possibly augmented with a memory as in Neural Turing Machines (Graves et al., 2014; 2016); it could exploit spatial invariances in the algorithmic task through a convolutional architecture (Bansal et al., 2022); it could be based on the transformer self-attentional architecture, as in the Universal Transformer (Dehghani et al., 2019); or it could be a Graph Neural Network (GNN). GNNs are particularly well suited for algorithmic execution (Veličković et al., 2020; Xu et al., 2020), and they have been applied to algorithmic problems before with a focus on extrapolation capabilities (Palm et al., 2017; Selsam et al., 2019; Joshi et al., 2020; Tang et al., 2020). Recently, Veličković & Blundell (2021) have proposed a general framework for algorithmic learning with GNNs. To reconcile different data encodings and provide a unified evaluation procedure, Veličković et al. (2022a) have presented a benchmark of algorithmic tasks covering a variety of areas. This benchmark, namely the CLRS algorithmic benchmark, represents data as graphs, showing that the graph formulation is general enough to include several algorithms, and not just the graph-based ones. On the CLRS benchmark, Ibarz et al. (2022) has recently presented several improvements in the architecture and learning procedure in order to obtain better performances. However, even the latest state-of-the-art models suffer from performance drops in certain algorithms when going out-of-distribution, an aspect we wish to improve upon here.

Self-supervised learning. Recently, many self-supervised representation learning methods that achieve good performance on a wide range of downstream vision tasks without access to labels have been proposed. One of the most popular approaches relies on contrastive objectives that make use of data augmentations to solve the instance discrimination task (Wu et al., 2018; Chen et al., 2020; He et al., 2020; Mitrovic et al., 2021). Other approaches that rely on target networks and clustering have also been explored (Grill et al., 2020; Caron et al., 2020). Our work is similar in spirit to Mitrovic et al. (2021), which examines representation learning through the lens of causality and employs techniques from invariant prediction to make better use of data augmentations. This approach has been demonstrated to be extremely successful on vision tasks (Tomasev et al., 2022). In the context of graphs, You et al. (2020); Suresh et al. (2021); You et al. (2022) have studied how to learn contrastive representations, with particular attention paid to data augmentations. Moreover, Veličković et al. (2019); Zhu et al. (2020) proposed novel objectives based on mutual information maximization in the graph domain to learn representations. Several other self-supervised methods (e.g. Thakoor et al. (2022)) have also been studied, and we refer the reader to Xie et al. (2022) for a review of existing literature on self-supervision with GNNs.
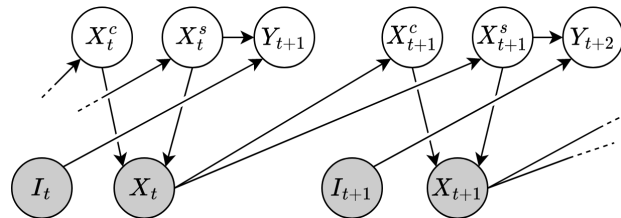

Figure 2. The causal graph formalising our assumption about the outcome of a step depends only on a subset X_t^s of the snapshot X_t, while the remainder X_t^c of the snapshot can be arbitrarily different.

3. Causal Model for Algorithmic Trajectories

An algorithm's execution trajectory is described in terms of the *inputs*, *outputs* and *hints*, which represent intermediate steps in the execution. We consider a graph-oriented way of representing this data (Veličković et al., 2022a): inputs and outputs are presented as data on nodes and edges of a graph, and hints are encoded as node, edge or graph features changing over time steps.

To better understand the data at hand, we propose to formalise the data generation process for an algorithmic trajectory using a *causal graph*. In such a causal graph, nodes represent random variables, and incoming arrows indicate that the node is a function of its parents (Pearl, 2009). The causal graph we use can be found in Figure 2. Note that this graph does not represent input data for the model, but a way of describing how any such data is generated.

Let us consider the execution trajectory of a certain algorithm of interest, at a particular time step t. Assume X_1 to be the observed input, and let X_t be the random variable denoting the "snapshot" at step t of the algorithm execution on the input. For example, in bubble sort, X_1 will be the initial (unsorted) array, and X_t the array after t steps of the sorting procedure (thus a partially-sorted array).

The *key contribution* of our causal graph is modelling the assumption that outcomes of a particular execution step depend only on a subset of the current snapshot, while the remainder of the snapshot can be arbitrarily different.

Accordingly, we assume the snapshot X_t to be generated from *two* random variables, X_t^c and X_t^s, with X_t^c representing the part of the snapshot that does not influence the current execution step (what can be changed without affecting the execution), while X_t^s the one that determines it (what needs to be stable).

Let us now revisit our bubble sort example from this perspective (see Figure 3). At each execution step, bubble sort compares two adjacent elements of the input list, and swaps them if they are not correctly ordered. Hence, in this par-
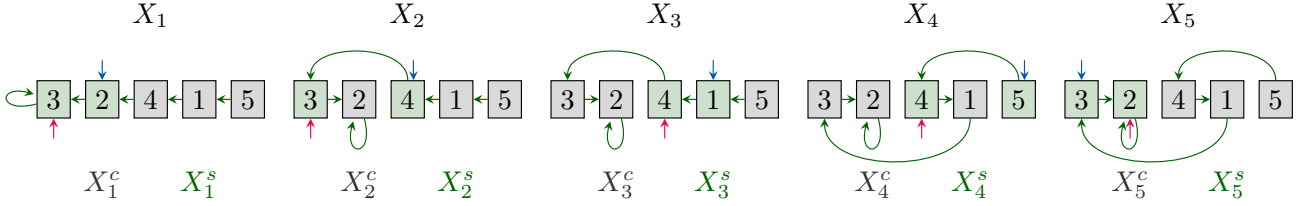
Figure 3. Example of values of X_t^c and X_t^s on an input array in the execution of the bubble sort algorithm. At every step of computation, bubble sort compares and possibly swaps exactly *two* nodes—those nodes are the only ones determining the outcome of the current step, and hence they constitute X_t^s. All other nodes are part of X_t^c.

ticular example, X_t^s constitutes these two elements being compared at step t, while the remaining elements—which do not affect whether or not a swap is going to happen at time t—form X_t^c. By definition this implies that the next algorithm state is a function of *only* X_t^s.

The data encoding used by Veličković et al. (2022a) prescribes that hints have values provided in *all* relevant parts of the graph. That is, in a graph of n nodes, an m-dimensional node hint has shape $\mathbb{R}^{n \times m}$, and an m-dimensional edge hint has shape $\mathbb{R}^{n \times n \times m}$. However, in order to keep our causal model simple, we choose to track the next-step hint in *only one* of those values, using an *index*, I_t, to decide which. Specifically, $I_t \in \{1, 2, \ldots, n\}$ are possible indices for node-level hints, and $I_t \in \{(1, 1), (1, 2), \ldots, (1, n), (2, 1), \ldots, (2, n), \ldots, (n, n)\}$ are possible indices for edge-level hints. For the indexed node/edge only, our causal graph then tracks the next-step value of the hint (either no change from the previous step or the new value), which we denote by Y_{t+1}.

Returning once again to our bubble sort example: one specific hint being tracked by the algorithm is which two nodes in the input list are currently considered for a swap. If $I_2 = 4$, then Y_3 will track whether node 4 is being considered for a swap, immediately after two steps of the bubble sort algorithm have been executed.

Once step t of the algorithm has been executed, a new snapshot X_{t+1} is produced, and it can be decomposed into X_{t+1}^c and X_{t+1}^s, just as before. Note that the execution in CLRS-30 is assumed *Markovian* (Veličković et al., 2022a): the snapshot at step t contains all the information to determine the snapshot at the next step. Finally, the execution terminates after T steps, and the final output is produced. We can then represent the output in a particular node/edge—indexed by I_T, just as before—by $Y_{T+1}^o := g(X_T^s, I_T)$, with g being the function producing the algorithm output.

As can be seen in Figure 2, X_t^s has all the necessary information to predict Y_{t+1}, since our causal model encodes the conditional independence assumption $Y_{t+1} \perp X_t^c \mid X_t^s$. More importantly, using the independence of mechanisms (Peters

et al., 2017) we can conclude that under this causal model, performing interventions on X_t^c by changing its value, does not change the conditional distribution $P(Y_{t+1} \mid X_t^s)$. Note that this is exactly the formalisation of our initial intuition: *the output of a particular step of the algorithm (i.e., Y_{t+1}) depends only on a subset of the current snapshot (i.e., X_t^s), and thus it is not affected by the addition of input items that do not interfere with it* (which we formalise as an **intervention** on X_t^c).[1] Therefore, given a step $t \in [1 \ldots T]$, for all $x, x' \in \mathcal{X}_t^c$, where $\mathcal{X}_t^c$ denotes the domain of X_t^c, we have that X_t^s is an *invariant* predictor of Y_{t+1} under interventions on X_t^c:

$$p^{\mathrm{do}(X_t^c)=x}(Y_{t+1}|X_t^s) = p^{\mathrm{do}(X_t^c)=x'}(Y_{t+1}|X_t^s), \quad (1)$$

where $p^{\mathrm{do}(X_t^c)=x}$ denotes the distribution obtained from assigning X_t^c the value of x, i.e. the interventional distribution.

Note, however, that Equation (1) does not give us a practical way of ensuring that our neural algorithmic reasoner respects these causal invariances, because it only has access to the entirety of the current snapshot X_t, without knowing its specific subsets X_t^c and X_t^s. More precisely, it is generally not known *which* input elements constitute X_t^s. For this reason, X_t^c and X_t^s are represented as unobserved random variables (white nodes) in Figure 2. In the next section, we will describe how to ensure invariant predictions for our reasoner, leveraging only X_t.

4. Size-Invariance through Self-Supervision in Neural Algorithmic Reasoning

Given a step t, to ensure invariant predictions of Y_{t+1} without access to X_t^s, we construct a *refinement* task Y_{t+1}^R and learn a representation $f(X_t, I_t)$ to predict Y_{t+1}^R, as originally proposed for images in Mitrovic et al. (2021). A refinement for a task (Chalupka et al., 2014) represents a more fine-grained version of the initial task.

More formally, given two tasks $R : \mathcal{A} \to \mathcal{B}$ and $T : \mathcal{A} \to$

---

[1]In bubble sort, adding sorted keys at the end of the array does not affect whether we are swapping the current entries.
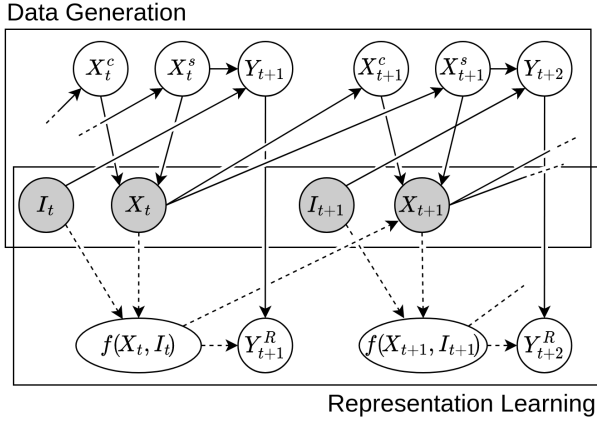
Data Generation

Representation Learning

Figure 4. Our causal graph with the inclusion of the representation learning components as in Mitrovic et al. (2021). Solid arrows represent the causal relationships. Dashed arrows represent what is used to learn (in the case of $f(X_t, I_t)$) or predict (in the case of Y_{t+1}^R) the corresponding random variables.

$\mathcal{B}'$, task R is more (or equally) fine-grained than task T if, for any two elements $a, a' \in \mathcal{A}$, $R(a) = R(a') \implies T(a) = T(a')$.

We will use this concept to show that a representation learned on the refinement task can be effectively used in the original task. Note that, as for Y_{t+1}, we assume $f(X_t, I_t)$ to be the representation learned from X_t of a predefined hint value—indexed by I_t—for example, the representation of the predecessor of a specific element of the input list.

Given a step t, let Y_{t+1}^R be a refinement of Y_{t+1}, and let $f(X_t, I_t)$ be a representation learned from X_t, used for the prediction of the refinement (see Figure 4). As we will formally prove, a representation that is invariant in the prediction of the refinement task across changes in X_t^c is also invariant in the prediction of the algorithmic step under these changes. Therefore, optimising $f(X_t, I_t)$ to be an invariant predictor for the refinement task Y_{t+1}^R represents a *sufficient* condition for the invariance in the prediction of the next algorithmic state, Y_{t+1}.

In the next subsection we present how to learn $f(X_t, I_t)$ in order to be an invariant predictor of Y_{t+1}^R under changes in X_t^c. Then, we show that this represents a sufficient condition for $f(X_t, I_t)$ to be an invariant predictor of Y_{t+1} across changes in X_t^c.

4.1. Learning an invariant predictor of the refinement

We consider Y_{t+1}^R to be the most-fine-grained refinement task, which corresponds to classifying each (hint) instance individually, that is, a contrastive learning objective where

we want to distinguish each hint from all others. This represents the *most-fine-grained refinement*, because $Y_{t+1}^R(a) = Y_{t+1}^R(a') \iff a = a'$, by definition. Our goal is to learn $f(X_t, I_t)$ to be an invariant predictor of Y_{t+1}^R under changes (interventions) of X_t^c. Thus, given a step $t \in [1 \dots T]$, for all $x, x' \in \mathcal{X}_t^c$, we want $f(X_t, I_t)$ such that

$$p^{\mathrm{do}(X_t^c)=x}(Y_{t+1}^R | f(X_t, I_t)) = p^{\mathrm{do}(X_t^c)=x'}(Y_{t+1}^R | f(X_t, I_t)), \tag{2}$$

where $p^{\mathrm{do}(X_t^c)}$ is the interventional distribution and $\mathcal{X}_t^c$ denotes the domain of X_t^c. Since we do not have access to X_t^c, as it is unobserved (it is a white node in Figures 2 and 4), we cannot explicitly intervene on it. Thus, we simulate interventions on X_t^c through data augmentation.

As we are interested in being invariant to appropriate size changes, we design a data augmentation procedure tailored for neural algorithmic reasoning, which mimics interventions changing the size of the input. Given a current snapshot of the algorithm on a given input, the data augmentation procedure should produce an augmented input which is larger, but on which the execution of the current step is going to proceed identically.

For example, a valid augmentation in bubble sort at a certain step consists of adding new elements to the tail of the input list, since the currently-considered swap will occur (or not) regardless of any elements added there. Thus, the valid augmentations for the bubble sort algorithm at a given step are all possible ways to add items in such a way that ensures that the one-step execution is unaffected by this addition.

To learn an encoder $f(X_t, I_t)$ that satisfies Equation (2), we propose to explicitly enforce invariance under valid augmentations. Such augmentations, as discussed, provide us with diverse inputs with an identical intermediate execution step.

Specifically, we use the ReLIC objective (Mitrovic et al., 2021) as a regularisation term, which we adapt to our causal graph as follows. Consider a time step, t, and let $\mathcal{D}_t$ be the dataset containing the snapshots at time t for all the inputs. Let $i_t, j_t \in I_t$ be two indices, and denote by $a_{lk} = (a_l, a_k) \in \mathcal{A}_{x_t} \times \mathcal{A}_{x_t}$ a pair of augmentations, with $\mathcal{A}_{x_t}$ the set of all possible valid augmentations at t for x_t (which simulate the interventions on X_t^c).

The objective function to optimise becomes:

$$\mathcal{L}_t =$$

$$- \sum_{x_t \in \mathcal{D}_t} \Big(\sum_{i_t} \sum_{a_{lk}} \log \frac{\exp\left(\phi(f(x_t^{a_l}, i_t), f(x_t^{a_k}, i_t))\right)}{\sum_{j_t \neq i_t} \exp\left(\phi(f(x_t^{a_l}, i_t), f(x_t^{a_k}, j_t))\right)}$$

$$- \alpha \sum_{a_{lk}, a_{qm}} \mathrm{KL}(p^{\mathrm{do}(a_{lk})}, p^{\mathrm{do}(a_{qm})}) \Big) \tag{3}$$

with x_t^a the data augmented with augmentation a, and α a weighting of the KL divergence penalty. The first term
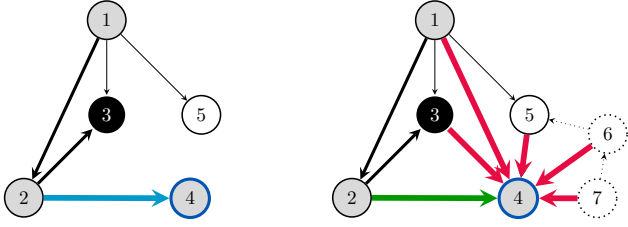
Figure 5. Example of applying our data augmentation and contrastive loss, following the example in Figure 1. An input graph (left) is augmented by adding nodes and edges (right), such that the next step—making 2 the parent of 4, i.e. $\pi_4 = 2$—remains the same. The representation of the pair $(4, 2)$ is hence contrasted against all other representations of pairs $(4, u)$ in the augmented graph. In other words, the green edge is the *positive* pair to the blue edge, with other edges (in red) being *negative* pairs to it.

represents a contrastive objective where we compare a hint representation in $x_t^{a_l}$, namely $f(x_t^{a_l}, i_t)$, with all the possible representations in $x_t^{a_k}$, $f(x_t^{a_k}, j_t)$. Note that this is different from standard contrastive objectives, where negative examples are taken from the batch. Due to space constraints, we expand on the derivation of Equation (3) in Appendix A.

In practice, we consider only one augmentation per graph, which is equivalent to setting a_l to the identity transformation. Consequently, the hint representation in the original graph $f(x_t^{a_l}, i_t)$ is regularised to be similar to the hint representation in the augmentation $f(x_t^{a_k}, i_t)$ and dissimilar to all other possible representations in the augmentation $f(x_t^{a_k}, j_t)$, $j_t \neq i_t$. Similarly, the hint representation in the augmentation $f(x_t^{a_k}, i_t)$ is regularised to be similar to the hint representation in the original graph $f(x_t^{a_l}, i_t)$ and dissimilar to all other possible representations in the original graph $f(x_t^{a_l}, j_t)$, $j_t \neq i_t$.

We follow the standard setup in contrastive learning and implement $\phi(f(x_t^{a_l}, i_t), f(x_t^{a_k}, i_t)) = \langle h(f(x_t^{a_l}), i_t), h(f(x_t^{a_k}), i_t) \rangle / \tau$ with h a fully-connected neural network and τ a temperature parameter. Finally, we use a KL penalty to ensure invariance in the probability distribution across augmentations. This is a requirement for satisfying the assumptions of our key theoretical result.

Example. To better understand Equation (3), we provide an example illustrated in Figure 5. We will consider one of the algorithms in CLRS-30—Kosaraju's strongly connected component (SCC) algorithm (Aho et al., 1974)—which consists of two invocations of depth-first search (DFS).

Let $G = (V, E)$ be an input graph to the SCC algorithm. Further, assume that at step t, the algorithm is visiting a node $v \in V$. We will focus on the prediction of the *parent*

of v: the node from which we have reached v in the current DFS invocation. Note that, in practice, this is a classification task where node v decides which of the other nodes is its parent. Accordingly, given a particular node v, our model computes a representation for every other node $u \in V$. This representation is which is then passed through a final classifier, outputting the (unnormalised) probability of u being the parent of v.

Now, consider any augmentation of G's nodes and edges that does not disrupt the current step of the search algorithm, denoted by $G^a = (V^a, E^a)$. For example, as the DFS implementation in CLRS-30 prefers nodes with a smaller `id` value, a valid augmentation can be obtained by adding nodes with a larger `id` than v to V^a, and adding edges from them to v in E^a (dashed nodes and edges in Figure 5). Note that this augmentation does not change the predicted parent of v. We can enforce that our representations respect this constraint by using our regularisation loss in Equation (3).

Given a node $v \in V$, we denote the representation of its parent node, $\pi_v \in V$ by $f(G, (v, \pi_v))$. This representation is contrasted to *all* other representations of nodes $w \in V^a$ in the augmented graph, that is $f(G^a, (v, w))$.[2]

More precisely, the most similar representation of $f(G, (v, \pi_v))$ is the representation *in the augmentation* of the parent of v, $f(G^a, (v, \pi_v))$, while the representations associated to all other nodes (including the added ones) represent the negative examples $f(G^a, (v, w))$, for $w \neq \pi_v$.

Figure 5 illustrates the prediction of the parent of node $v = 4$. In this case, i_t in Equation (3) indexes the true parent of node 4, namely $\pi_4 = 2$, and therefore $i_t = (4, 2)$, while j_t iterates over all other possible indices of the form $(4, u)$, $u \in V^a$, indeed representing *all other possible parents* of 4. The objective of Equation (3) is to make the true parent representation in the original graph $f(G, (4, 2))$ similar to the true parent representation in the augmentation $f(G^a, (4, 2))$, and dissimilar to the representations of the other possible parents in the augmentation $f(G^a, (4, u))$, $u \in V^a$. The same process applies to the augmentation.

4.2. Implications of the invariance

In the previous subsection, we have presented a self-supervised objective, justified by our assumed causal graph, in order to learn invariant predictors for a refinement task Y_{t+1}^R under changes of X_t^c. However, our initial goal was to ensure invariance in the prediction of algorithmic hints Y_{t+1} across X_t^c. Now we will bridge these two aims.

In the following, we show how learning a representation that is an invariant predictor of Y_{t+1}^R under changes of X_t^c

---

[2]Note that, in this case, I_t is a two-dimensional index, choosing two nodes—i.e., an edge—at once.

represents a *sufficient* condition for this representation to be invariant to X_t^c when predicting Y_{t+1}.

Theorem 4.1. *Consider an algorithm and let $t \in [1 \ldots T]$ be one of its steps. Let Y_{t+1} be the task representing a prediction of the algorithm step and let Y_{t+1}^R be a refinement of such task. If $f(X_t, I_t)$ is an invariant representation for Y_{t+1}^R under changes in X_t^c, then $f(X_t, I_t)$ is an invariant representation for Y_{t+1} under changes in X_t^c, that is, for all $x, x' \in \mathcal{X}_t^c$, the following holds:*

$$p^{do(X_t^c)=x}(Y_{t+1}^R | f(X_t, I_t)) = p^{do(X_t^c)=x'}(Y_{t+1}^R | f(X_t, I_t))$$
$$\implies$$
$$p^{do(X_t^c)=x}(Y_{t+1} | f(X_t, I_t)) = p^{do(X_t^c)=x'}(Y_{t+1} | f(X_t, I_t)).$$

We prove Theorem 4.1 in Appendix D. Note that this justifies our self-supervised objective: by learning invariant representations though a refinement task, we can also guarantee invariance in the hint prediction. In other words, we can *provably* ensure that the prediction of an algorithm step is not affected by changes in the input that do not interfere with the current execution step. Since we can express these changes in the form of *addition* of input nodes, we are ensuring that the hint prediction is the same on two inputs of different sizes, but identical current algorithmic step.

5. Experiments

We conducted an extensive set of experiments to answer the following main questions:

1. *Can our model, Hint-ReLIC, which relies on the addition of our causality-inspired self-supervised objective, outperform the corresponding base model in practice?*

2. *What is the importance of such objective when compared to other changes made with respect to the previous state-of-the-art model?*

3. *How does Hint-ReLIC compare to a model which does not leverage hints at all, directly predicting the output from the input? Are hints necessary?*

Model. As a base model, we use the Triplet-GMPNN architecture proposed by Ibarz et al. (2022), which consists of a fully-connected MPNN (Gilmer et al., 2017) where the input graph is encoded in the edge features, augmented with gating and triplet reasoning (Dudzik & Veličković, 2022).

We replace the loss for predicting the next-step hint in the base model with our regularisation objective (Equation (3)), which aims at learning hint representations that are *invariant to size changes that are irrelevant to the current step* via constrastive and KL losses.

We make an additional change with respect to the base model, consisting of including the *reversal* of hints of `pointer` type. More specifically, given an input graph, if a node A points to another node B in the graph, we include an additional (edge-based) hint representing the pointer from B to A. This change (which we refer to as **reversal** in the results) consists simply in the inclusion of these additional hints, and we study the impact of this addition in Section 5.1. The resulting model is what we call Hint-ReLIC.

Data augmentations. To simulate interventions on X_t^c and learn invariant representations, we design augmentation procedures which construct augmented data given an input and an algorithm step, such that the step of the algorithm is the same on the original input and on the augmented data.

We consider simple augmentations, which we describe in detail in Appendix E. To reduce the computational overhead, given an input graph, instead of sampling an augmentation at each algorithm step, we sample a single step, $\tilde{t} \sim \mathcal{U}\{1, T\}$, and construct an augmentation only for the sampled step. Then, we use the (same) constructed augmentation in all the steps *until* the sampled one, $t \leq \tilde{t}$. This follows from the consideration that, if augmentations are carefully constructed, the execution of the algorithm is the same not only in the next step but in all steps leading up to that.

Whenever possible, we relax the requirement of having the augmentation with *exactly* the same execution, and we allow for approximate augmentations, in order to avoid over-engineering the methodology and obtain a more robust model. This results in more general and simpler augmentations, though we expect more tailored ones to perform better. We refer the reader to Appendix E for more details.

We end this paragraph by stressing that we *never* run the target algorithm on the augmented inputs: rather, we directly construct them to have the same next execution step as the corresponding inputs. As a result, our method does not require direct access to the algorithm used to generate the inputs. Furthermore, the number of nodes in our augmentations is at most one more than the number of nodes in the largest training input example. This means that, in all of our experiments, we still never significantly cross the intended test size distribution shift during training.

Datasets. We run our method on a diverse subset of the algorithms present in the CLRS benchmark consisting of: 1. *DFS-based algorithms* (Articulation Points, Bridges, Strongly Connected Components (Aho et al., 1974), Topological Sort (Knuth, 1973)); 2. *Other graph-based algorithms* (Bellman-Ford (Bellman, 1958), BFS (Moore, 1959), DAG Shortest Paths, Dijkstra (Dijkstra et al., 1959), Floyd-Warshall (Floyd, 1962), MST-Kruskal (Kruskal, 1956), MST-Prim (Prim, 1957)); 3. *Sorting algorithms* (Bubble
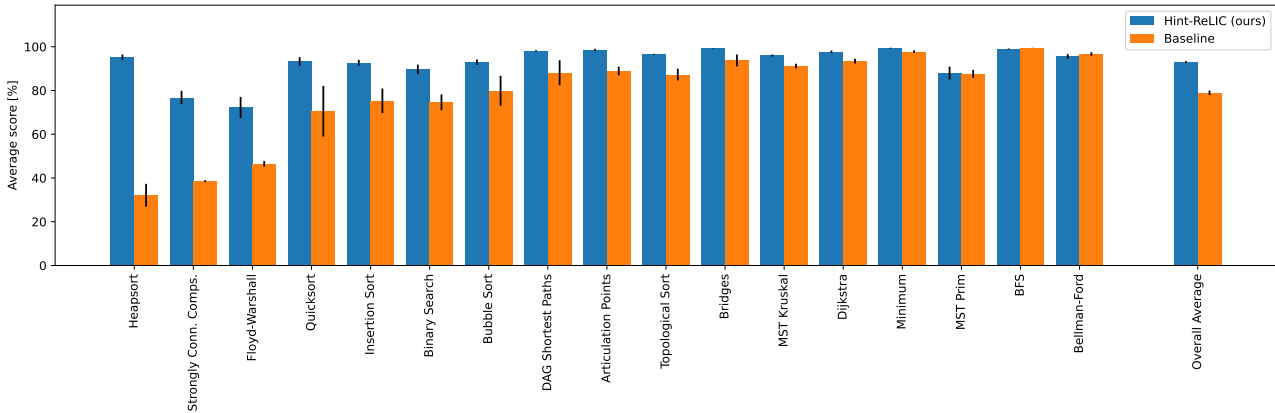
Figure 6. Per-algorithm comparison of the Triplet-GMPNN baseline (Ibarz et al., 2022) and our Hint-ReLIC. Error bars represent the standard error of the mean across three random seeds. The final column shows the average and standard error of the mean performances across the different algorithms.

Sort, Heapsort (Williams, 1964), Insertion Sort, Quicksort (Hoare, 1962)); 4. *Searching algorithms* (Binary-search, Minimum). This subset is chosen as it contains most algorithms suffering from out-of-distribution performance drops in current state-of-the-art; see Ibarz et al. (2022, Table 2).

Results. Figure 6 compares the out-of-distribution (OOD) performances of the Triplet-GMPNN baseline, which we have re-trained and evaluated in our experiments, to our model Hint-ReLIC, as described above. Hint-ReLIC performs better or comparable to the existing state-of-the-art baseline, showcasing how the proposed procedure appears to be beneficial not only theoretically, but also in practice. The most significant improvements can be found in the sorting algorithms, where we obtain up to $3\times$ increased performance.

5.1. Ablation study

In this section we study the contribution and importance of two main components of our methodology. First, we consider the impact of the change we made with respect to the original baseline proposed in Ibarz et al. (2022), namely the inclusion of the reversal of hint pointers. Second, as we propose a novel way to leverage hints through our self-supervised objective, which is different from the direct supervision in the baseline, one may wonder whether completely removing hints can achieve even better scores. Thus, we also study the performance when completely disregarding hints and directly going from input to output. Finally, we refer the reader to Appendix F.1 for additional ablation experiments, including the removal of the KL component in Equation (3)—which is necessary for the theoretical results but may not always be needed in practice.

Table 1. Effect of the inclusion of pointers' reversal on each algorithm. The table shows mean and stderr of the OOD micro-F_1 score after 10,000 training steps, across different seeds.

Alg.	Baseline	Baseline + reversal	Hint-ReLIC (ours)
Articulation points	$88.93\% \pm 1.92$	$91.04\% \pm 0.92$	$\mathbf{98.45}\% \pm 0.60$
Bridges	$93.75\% \pm 2.73$	$97.70\% \pm 0.34$	$\mathbf{99.32}\% \pm 0.09$
SCC	$38.53\% \pm 0.45$	$31.40\% \pm 8.80$	$\mathbf{76.79}\% \pm 3.04$
Topological sort	$87.27\% \pm 2.67$	$88.83\% \pm 7.29$	$\mathbf{96.59}\% \pm 0.20$
Bellman-Ford	$\mathbf{96.67}\% \pm 0.81$	$95.02\% \pm 0.49$	$95.54\% \pm 1.06$
BFS	$99.64\% \pm 0.05$	$\mathbf{99.93}\% \pm 0.03$	$99.00\% \pm 0.21$
DAG Shortest Paths	$88.12\% \pm 5.70$	$96.61\% \pm 0.61$	$\mathbf{98.17}\% \pm 0.26$
Dijkstra	$93.41\% \pm 1.08$	$91.50\% \pm 1.85$	$\mathbf{97.74}\% \pm 0.50$
Floyd-Warshall	$46.51\% \pm 1.30$	$46.28\% \pm 0.80$	$\mathbf{72.23}\% \pm 4.84$
MST-Kruskal	$91.18\% \pm 1.05$	$89.93\% \pm 0.43$	$\mathbf{96.01}\% \pm 0.45$
MST-Prim	$87.64\% \pm 1.79$	$86.95\% \pm 2.34$	$\mathbf{87.97}\% \pm 2.94$
Insertion sort	$75.28\% \pm 5.62$	$87.21\% \pm 2.80$	$\mathbf{92.70}\% \pm 1.29$
Bubble sort	$79.87\% \pm 6.85$	$80.51\% \pm 9.10$	$\mathbf{92.94}\% \pm 1.23$
Quicksort	$70.53\% \pm 11.59$	$85.69\% \pm 4.53$	$\mathbf{93.30}\% \pm 1.96$
Heapsort	$32.12\% \pm 5.20$	$49.13\% \pm 10.35$	$\mathbf{95.16}\% \pm 1.27$
Binary Search	$74.60\% \pm 3.61$	$50.42\% \pm 8.45$	$\mathbf{89.68}\% \pm 2.13$
Minimum	$97.78\% \pm 0.63$	$98.43\% \pm 0.01$	$\mathbf{99.37}\% \pm 0.20$

The effect of the inclusion of pointers' reversal. As discussed above, pointers' reversal simply consists of adding an additional hint for each hint of `pointer` type (if any), such that a node not only has the information representing which other node it points to, but also from which nodes it is pointed by. We study the impact of this inclusion by running the baseline with these additional hints, and evaluate its performance against both the baseline and our Hint-ReLIC. Table 1 shows that this addition, which we refer to as **Baseline + reversal**, indeed leads to improved results for certain algorithms, but does not obtain the predictive performances we reached with our regularisation objective.

The removal of hints. While previous works directly included the supervision on the hint predictions, we argue in

8

Table 2. Importance of hint usage in the final performance. The table shows mean and stderr of the OOD micro-F_1 score after 10,000 training steps, across different seeds.

Alg.	No Hints	Hint-ReLIC (ours)
Articulation points	$81.97\% \pm 5.08$	$\mathbf{98.45}\% \pm 0.60$
Bridges	$95.62\% \pm 1.03$	$\mathbf{99.32}\% \pm 0.09$
SCC	$57.63\% \pm 0.68$	$\mathbf{76.79}\% \pm 3.04$
Topological sort	$84.29\% \pm 1.16$	$\mathbf{96.59}\% \pm 0.20$
Bellman-Ford	$93.26\% \pm 0.04$	$\mathbf{95.54}\% \pm 1.06$
BFS	$\mathbf{99.89}\% \pm 0.03$	$99.00\% \pm 0.21$
DAG Shortest Paths	$97.62\% \pm 0.62$	$\mathbf{98.17}\% \pm 0.26$
Dijkstra	$95.01\% \pm 1.14$	$\mathbf{97.74}\% \pm 0.50$
Floyd-Warshall	$40.80\% \pm 2.90$	$\mathbf{72.23}\% \pm 4.84$
MST-Kruskal	$92.28\% \pm 0.82$	$\mathbf{96.01}\% \pm 0.45$
MST-Prim	$85.33\% \pm 1.21$	$\mathbf{87.97}\% \pm 2.94$
Insertion sort	$77.29\% \pm 7.42$	$\mathbf{92.70}\% \pm 1.29$
Bubble sort	$81.32\% \pm 6.50$	$\mathbf{92.94}\% \pm 1.23$
Quicksort	$71.60\% \pm 2.22$	$\mathbf{93.30}\% \pm 1.96$
Heapsort	$68.50\% \pm 2.81$	$\mathbf{95.16}\% \pm 1.27$
Binary Search	$\mathbf{93.21}\% \pm 1.10$	$89.68\% \pm 2.13$
Minimum	$99.24\% \pm 0.21$	$\mathbf{99.37}\% \pm 0.20$

favour of a novel way of leveraging hints. We use hints first to construct the augmentations representing the same algorithm step, and then we employ their representations in the self-supervised objective. An additional valid model might consist of a model that directly goes from input to output and completely ignores hints. In Table 2 we show that this **No Hints** model can achieve very good performances, but it is still generally outperformed by Hint-ReLIC.

6. Conclusions

In this work we propose a self-supervised learning objective that employs augmentations derived from available hints, which represent intermediate steps of an algorithm, as a way to better ground the execution of GNN-based algorithmic reasoners on the computation that the target algorithm performs. Our Hint-ReLIC model, based on such self-supervised objective, leads to algorithmic reasoners that produce more robust outputs of the target algorithms, especially compared to autoregressive hint prediction. In conclusion, hints can take you a long way, if used in the right way.

Acknowledgements

References

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. The design and analysis of computer algorithms. *Reading*, 1974.

Alet, F., Doblar, D., Zhou, A., Tenenbaum, J., Kawaguchi, K., and Finn, C. Noether networks: meta-learning useful conserved quantities. *Advances in Neural Information Processing Systems*, 34:16384–16397, 2021.

Bansal, A., Schwarzschild, A., Borgnia, E., Emam, Z., Huang, F., Goldblum, M., and Goldstein, T. End-to-end algorithm synthesis with recurrent networks: Logical extrapolation without overthinking. *arXiv preprint arXiv:2202.05826*, 2022.

Bellman, R. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

Beurer-Kellner, L., Vechev, M., Vanbever, L., and Veličković, P. Learning to configure computer networks with neural algorithmic reasoning. In *Advances in Neural Information Processing Systems*, 2022.

Bevilacqua, B., Zhou, Y., and Ribeiro, B. Size-invariant graph representations for graph classification extrapolations. In *International Conference on Machine Learning*, pp. 837–851. PMLR, 2021.

Bronstein, M. M., Bruna, J., Cohen, T., and Veličković, P. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.

Buffelli, D., Liò, P., and Vandin, F. Sizeshiftreg: a regularization method for improving size-generalization in graph neural networks. In *Advances in Neural Information Processing Systems*, 2022.

Caron, M., Misra, I., Mairal, J., Goyal, P., Bojanowski, P., and Joulin, A. Unsupervised learning of visual features by contrasting cluster assignments. *Advances in Neural Information Processing Systems*, 33:9912–9924, 2020.

Chalupka, K., Perona, P., and Eberhardt, F. Visual causal feature learning. *arXiv preprint arXiv:1412.2309*, 2014.

Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pp. 1597–1607. PMLR, 2020.

Chen, Y., Zhang, Y., Bian, Y., Yang, H., KAILI, M., Xie, B., Liu, T., Han, B., and Cheng, J. Learning causally invariant representations for out-of-distribution generalization on graphs. In *Advances in Neural Information Processing Systems*, 2022.

Corso, G., Cavalleri, L., Beaini, D., Liò, P., and Veličković, P. Principal neighbourhood aggregation for graph nets. *Advances in Neural Information Processing Systems*, 33: 13260–13271, 2020.

Davies, A., Veličković, P., Buesing, L., Blackwell, S., Zheng, D., Tomašev, N., Tanburn, R., Battaglia, P., Blundell, C., Juhász, A., et al. Advancing mathematics by guiding human intuition with ai. *Nature*, 600(7887):70–74, 2021.

Deac, A.-I., Veličković, P., Milinkovic, O., Bacon, P.-L., Tang, J., and Nikolic, M. Neural algorithmic reasoners are implicit planners. *Advances in Neural Information Processing Systems*, 34:15529–15542, 2021.

Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., and Kaiser, L. Universal transformers. In *International Conference on Learning Representations*, 2019.

Diao, C. and Loynd, R. Relational attention: Generalizing transformers for graph-structured tasks. *arXiv preprint arXiv:2210.05062*, 2022.

Dijkstra, E. W. et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

Dudzik, A. J. and Veličković, P. Graph neural networks are dynamic programmers. In *Advances in Neural Information Processing Systems*, 2022.

Floyd, R. W. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

Gasteiger, J., Shuaibi, M., Sriram, A., Günnemann, S., Ulissi, Z. W., Zitnick, C. L., and Das, A. How do graph networks generalize to large and diverse molecular systems? *ArXiv*, abs/2204.02782, 2022.

Gers, F. A. and Schmidhuber, J. Lstm recurrent networks learn simple context-free and context-sensitive languages. *IEEE transactions on neural networks*, 12 6:1333–40, 2001.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning*, ICML'17, pp. 1263–1272. JMLR.org, 2017.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *ArXiv*, abs/1410.5401, 2014.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., Badia, A. P., Hermann, K. M., Zwols, Y., Ostrovski, G., Cain, A., King, H., Summerfield, C., Blunsom, P., Kavukcuoglu, K., and Hassabis, D. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538 (7626):471–476, October 2016. ISSN 00280836.

Grill, J.-B., Strub, F., Altché, F., Tallec, C., Richemond, P., Buchatskaya, E., Doersch, C., Avila Pires, B., Guo, Z., Gheshlaghi Azar, M., et al. Bootstrap your own latent-a new approach to self-supervised learning. *Advances in neural information processing systems*, 33:21271–21284, 2020.

He, K., Fan, H., Wu, Y., Xie, S., and Girshick, R. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 9729–9738, 2020.

Hoare, C. A. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

Ibarz, B., Kurin, V., Papamakarios, G., Nikiforou, K., Bennani, M., Csordás, R., Dudzik, A. J., Bošnjak, M., Vitvitskyi, A., Rubanova, Y., Deac, A., Bevilacqua, B., Ganin, Y., Blundell, C., and Veličković, P. A generalist neural algorithmic learner. In *The First Learning on Graphs Conference*, 2022.

Jegelka, S. Theory of graph neural networks: Representation and learning. *arXiv preprint arXiv:2204.07697*, 2022.

Joshi, C. K., Cappart, Q., Rousseau, L.-M., and Laurent, T. Learning the travelling salesperson problem requires rethinking generalization. *Constraints*, 27:70 – 98, 2020.

Knuth, D. E. Fundamental algorithms. 1973.

Kruskal, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

Mahdavi, S., Swersky, K., Kipf, T., Hashemi, M., Thrampoulidis, C., and Liao, R. Towards better out-of-distribution generalization of neural algorithmic reasoning tasks. *arXiv preprint arXiv:2211.00692*, 2022.

Mitrovic, J., McWilliams, B., Walker, J. C., Buesing, L. H., and Blundell, C. Representation learning via invariant causal mechanisms. In *International Conference on Learning Representations*, 2021.

Moore, E. F. The shortest path through a maze. In *Proc. Int. Symp. Switching Theory, 1959*, pp. 285–292, 1959.

Palm, R. B., Paquet, U., and Winther, O. Recurrent relational networks. In *Neural Information Processing Systems*, 2017.

Pearl, J. *Causality*. Cambridge university press, 2009.

Peters, J., Janzing, D., and Schölkopf, B. *Elements of causal inference: foundations and learning algorithms*. The MIT Press, 2017.

Prim, R. C. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6): 1389–1401, 1957.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. Learning a SAT solver from single-bit supervision. In *International Conference on Learning Representations*, 2019.

Suresh, S., Li, P., Hao, C., and Neville, J. Adversarial graph augmentation to improve graph contrastive learning. In *Advances in Neural Information Processing Systems*, 2021.

Tang, H., Huang, Z., Gu, J., Lu, B.-L., and Su, H. Towards scale-invariant graph-related problem solving by iterative homogeneous gnns. *Advances in Neural Information Processing Systems*, 33:15811–15822, 2020.

Thakoor, S., Tallec, C., Azar, M. G., Azabou, M., Dyer, E. L., Munos, R., Veličković, P., and Valko, M. Large-scale representation learning on graphs via bootstrapping. In *International Conference on Learning Representations*, 2022.

Tomasev, N., Bica, I., McWilliams, B., Buesing, L., Pascanu, R., Blundell, C., and Mitrovic, J. Pushing the limits of self-supervised resnets: Can we outperform supervised learning without labels on imagenet? *arXiv preprint arXiv:2201.05119*, 2022.

Veličković, P. and Blundell, C. Neural algorithmic reasoning. *Patterns*, 2(7):100273, 2021.

Veličković, P., Badia, A. P., Budden, D., Pascanu, R., Banino, A., Dashevskiy, M., Hadsell, R., and Blundell, C. The CLRS algorithmic reasoning benchmark. In *International Conference on Machine Learning*, 2022a.

Veličković, P., Bošnjak, M., Kipf, T., Lerchner, A., Hadsell, R., Pascanu, R., and Blundell, C. Reasoning-modulated representations. In *The First Learning on Graphs Conference*, 2022b.

Veličković, P., Fedus, W., Hamilton, W. L., Liò, P., Bengio, Y., and Hjelm, R. D. Deep graph infomax. In *International Conference on Learning Representations*, 2019.

Veličković, P., Ying, R., Padovano, M., Hadsell, R., and Blundell, C. Neural execution of graph algorithms. In *International Conference on Learning Representations*, 2020.

Williams, J. W. J. Algorithm 232: heapsort. *Commun. ACM*, 7:347–348, 1964.

Wu, Z., Xiong, Y., Yu, S. X., and Lin, D. Unsupervised feature learning via non-parametric instance discrimination. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3733–3742, 2018.

Xie, Y., Xu, Z., Zhang, J., Wang, Z., and Ji, S. Self-supervised learning of graph neural networks: A unified review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

Xu, K., Li, J., Zhang, M., Du, S. S., ichi Kawarabayashi, K., and Jegelka, S. What can neural networks reason about? In *International Conference on Learning Representations*, 2020.

Xu, K., Zhang, M., Li, J., Du, S. S., Kawarabayashi, K.-I., and Jegelka, S. How neural networks extrapolate: From feedforward to graph neural networks. In *International Conference on Learning Representations*, 2021.

Yehudai, G., Fetaya, E., Meirom, E., Chechik, G., and Maron, H. From local structures to size generalization in graph neural networks. In *International Conference on Machine Learning*, pp. 11975–11986. PMLR, 2021.

You, Y., Chen, T., Sui, Y., Chen, T., Wang, Z., and Shen, Y. Graph contrastive learning with augmentations. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 5812–5823. Curran Associates, Inc., 2020.

You, Y., Chen, T., Wang, Z., and Shen, Y. Bringing your own view: Graph contrastive learning without prefabricated data augmentations. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*, pp. 1300–1309, 2022.

Zhou, Y., Kutyniok, G., and Ribeiro, B. OOD link prediction generalization capabilities of message-passing GNNs in larger test graphs. In *Advances in Neural Information Processing Systems*, 2022.

Zhu, Y., Xu, Y., Yu, F., Liu, Q., Wu, S., and Wang, L. Deep Graph Contrastive Representation Learning. In *ICML Workshop on Graph Representation Learning and Beyond*, 2020.

A. Derivation of our Self-Supervised Objective

Equation (3) represents our objective function to optimise, which we derived by adapting the ReLIC objective (Mitrovic et al., 2021) to our causal graph. Note that we propose a unique and novel way of employing a contrastive-learning based objective, which is also different from Mitrovic et al. (2021), as we consider positive and negative examples for each hint representation in an input graph to be hint representations in valid augmentations of the graph. To better understand the equation, in this section we expand the derivation of our objective.

Recall that our goal is to learn $f(X_t, I_t)$ to be an invariant predictor of Y_{t+1}^R under changes (interventions) of X_t^c. As we do not have access to X_t^c, because we do not know which subset of the input forms X_t^c, we simulate interventions on X_t^c through data augmentations. Therefore, our goal becomes to learn $f(X_t, I_t)$ to be an invariant predictor of Y_{t+1}^R under (valid) augmentations, that is:

$$p^{\mathrm{do}(a_i)}(Y_{t+1}^R | f(X_t, I_t)) = p^{\mathrm{do}(a_j)}(Y_{t+1}^R | f(X_t, I_t)), \quad \forall a_i, a_j \in \mathcal{A}_{x_t}$$

where $\mathcal{A}_{x_t}$ contains all possible valid augmentations at t for x_t, and $p^{\mathrm{do}(a)}$ represents the simulation of the intervention on X_t^c through data augmentation a. Following Mitrovic et al. (2021), we enforce this invariance through a regularisation objective, which for every time step t has the following form:

$$\mathbb{E}_{\substack{X_t}} \mathbb{E}_{\substack{a_{lk}, a_{qm} \\ \sim \mathcal{A}_{x_t} \times \mathcal{A}_{x_t}}} \sum_{i_t} \sum_{b \in \{a_{lk}, a_{qm}\}} \hat{\mathcal{L}}_b(f(X_t, i_t), Y_{t+1}^R = i_t) \quad \text{s.t.} \quad \mathrm{KL}(p^{\mathrm{do}(a_{lk})}(Y_{t+1}^R | f(X_t, i_t)), p^{\mathrm{do}(a_{qm})}(Y_{t+1}^R | f(X_t, i_t))) \leq \rho,$$

for some small number ρ. Since we consider $\hat{\mathcal{L}}$ to be a contrastive learning objective, we take pairs of hint representations, indexed by i_t and j_t, to compute similarity scores and use pairs of augmentations $a_{lk} = (a_l, a_k) \in \mathcal{A}_{x_t} \times \mathcal{A}_{x_t}$, that is

$$p^{\mathrm{do}(a_{lk})}(Y_{t+1}^R = j_t | f(x_t, i_t)) \propto \exp\left(\phi(f(x_t^{a_l}, i_t), f(x_t^{a_k}, j_t))\right),$$

where f is a neural network and ϕ is a (learnable) function to compute the similarity between two representations. Note that, in words, we are computing the similarity of two hint representations (indexed by i_t and j_t, respectively) in two data augmentations (obtained from a_l and a_k). Now, note that we want the representations of the same hint to be similar in the two augmentations. This means that the hint representation indexed by i_t in $x_t^{a_l}$ must be similar to the hint representation indexed by i_t in $x_t^{a_k}$. Obviously, the same must be true when considering j_t instead of i_t. Furthermore, we want representations of different hints to be dissimilar in the two augmentations. Putting all together, our objective function at time t can be rewritten as

$$\mathcal{L}_t = - \sum_{x_t \in \mathcal{D}_t} \left(\sum_{a_{lk}} \sum_{i_t} \log \frac{\exp\left(\phi(f(x_t^{a_l}, i_t), f(x_t^{a_k}, i_t))\right)}{\sum_{j_t \neq i_t} \exp\left(\phi(f(x_t^{a_l}, i_t), f(x_t^{a_k}, j_t))\right)} - \alpha \sum_{a_{lk}, a_{qm}} \mathrm{KL}(p^{\mathrm{do}(a_{lk})}, p^{\mathrm{do}(a_{qm})}) \right),$$

where $\mathcal{D}_t$ is the dataset containing the snapshots at time t for all the inputs, $i_t, j_t \in I_t$ are two indices, $a_{lk} = (a_l, a_k) \in \mathcal{A}_{x_t} \times \mathcal{A}_{x_t}$ is a pair of augmentations, with $\mathcal{A}_{x_t}$ the set of all possible valid augmentations at t for x_t (which simulate the interventions on X_t^c). Finally, α is the weighting of the KL divergence penalty and $p^{\mathrm{do}(a_{lk})}$ is a shorthand for $p^{\mathrm{do}(a_{lk})}(Y_{t+1}^R | f(x_t, i_t))$.

We note here that j_t is such that the hint representations of i_t and j_t are actually different. In the SCC example in the main text (visualised in Figure 5), the sum over j_t is a sum over all other possible parents of the node v under consideration. In a DFS's execution, when the hint under consideration is the colour of a node in an input graph, its representation is regularised to be similar to the hint representation of the same node in the augmentation, and dissimilar to all other hint representations in the augmentation corresponding to *different* colours.

B. Causal Graph and Representation Learning Components

In this section we expand on the definition of our causal graph and on the representation learning components, justifying design choices that were left implicit in the main paper due to space constraints. Specifically, we first explain thoroughly the causal relationships among the random variables and then stress how we use those variables in a learning setting.

Recall that our causal graph (Figure 2) describes the data generation process of an algorithmic trajectory. We denote by X_1 the random variable representing the input to our algorithm, and refer by X_t the snapshot at time step t of the algorithm execution on such input, for every time step in the trajectory $t \in [1 \dots T]$.

We assume X_t to be generated by two random variables, X_t^c and X_t^s, which we assume to be *distinct* parts (or splits) of X_t, which *together* form the whole X_t. We consider X_t^c to be the part of the snapshot that does not influence the current execution of the algorithm at time step t, and can therefore be arbitrarily different without affecting it. We instead denote by X_t^s the part of the snapshot that determines the current execution of the algorithm at time step t, and therefore should not be changed if we do not want to alter the current step execution.

The current execution is represented as hint values on all nodes and/or edges. We call Y_{t+1} the execution (or hint) *on a specific node or edge* chosen accordingly to an index I_t. Note that the current step hint Y_{t+1} is represented with an increment of the time step, following a convention we adopt to indicate that we first need an execution and only then (in the next time step) we materialise its results. Further, note that Y_{t+1} represents the algorithmic step in any node or edge, indicating whether or not it is involved in the current execution, thus either encoding that there is no change (and thus it is not involved in the current step) or representing what is its new hint value.

By definition of X_t^c and X_t^s, the current step of the algorithm on the node or edge indexed by I_t, namely Y_{t+1}, is determined by X_t^s only.

Assuming a Markovian execution, executing one algorithm step gives us X_{t+1}^c and X_{t+1}^s, which form the snapshot we observe X_{t+1}. Note that X_{t+1}^c and X_{t+1}^s are potentially different from X_t^c and X_t^s, because the current execution might now be determined by very different subsets. Finally, note that we do not need an arrow from Y_{t+1} to X_t^c or X_t^s, because Y_{t+1} is *deterministically* determined by X_t^s, and therefore all its information can be recovered from X_t^s.

Recall now that our goal is to learn an invariant predictor for the refinement task across changes of X_t^c, as this represents a sufficient condition for the invariance in the prediction of Y_{t+1} (see Theorem 4.1). We denote by Y_{t+1}^R the refinement task of the execution on a specific node or edge chosen according to I_t. We omit the arrow from I_t to Y_{t+1}^R as the dependency is already implicit through Y_{t+1}.

We denote by $f(X_t, I_t)$ the representation of the execution step on a particular node or edge indexed by I_t, which we learn to predict Y_{t+1}^R across changes of X_t^c. Finally, note that $f(X_t, I_t)$ is used by the network to determine the predicted next snapshot, which is determined by the next step prediction, and therefore it has a (dashed) arrow to X_{t+1}.

C. Assumptions on Prior Knowledge of the Unobserved X_t^c

Given a time step t, to ensure invariant predictions of Y_{t+1} we learn predictors of Y_{t+1}^R that are invariant across interventions on X_t^c, simulated through data augmentations. However, since X_t^c is an unobserved random variable, we must make assumptions about its properties to create valid data augmentations. In this section, we clarify our assumptions about prior knowledge of X_t^c and propose potential methods to eliminate this assumption in future work.

We start by remarking that our neural network is not assumed to have any prior knowledge about X_t^c. This knowledge is enforced into the network through our regularisation objective (Equation (3)), which is driven by appropriately chosen data augmentations. Indeed, those data augmentations do rely on priors that assume something about what X_t^c might look like. This is similar to how the choice of data augmentation in image CNNs governs which parts of the image we consider to be "content" and "style".

However, for most algorithms of interest, the required priors are conceptually very simple, and a single augmentation may be reused for many algorithms. As an example, for many graph algorithms, it is an entirely safe operation to add disconnected subgraphs – an augmentation we repeatedly employ. Similarly in several sorting tasks, adding elements to the tail end of the list represent a valid augmentation. We provide an exhaustive list of the augmentation for each algorithm in Appendix E.

Performing data augmentations without knowledge of X_t^c represents an interesting but challenging direction, that can be explored in future work. A simple, computationally-expensive, data-augmentation procedure that does not require any knowledge of X_t^c could consist in randomly augmenting the input graph, run the actual algorithm and consider the generated graph as a valid augmentation *only if* the next step execution of the algorithm remains unaltered. A more interesting approach would consist in learning valid augmentations of a given input, perhaps by meta-learning conserved quantities in the spirit of Noether Networks (Alet et al., 2021). Investigating these cases remains an important avenue for future research.

D. Theoretical Analysis

Theorem 4.1. *Consider an algorithm and let $t \in [1 \ldots T]$ be one of its steps. Let Y_{t+1} be the task representing a prediction of the algorithm step and let Y_{t+1}^R be a refinement of such task. If $f(X_t, I_t)$ is an invariant representation for Y_{t+1}^R under changes in X_t^c, then $f(X_t, I_t)$ is an invariant representation for Y_{t+1} under changes in X_t^c, that is, for all $x, x' \in \mathcal{X}_t^c$, the following holds:*

$$p^{do(X_t^c)=x}(Y_{t+1}^R | f(X_t, I_t)) = p^{do(X_t^c)=x'}(Y_{t+1}^R | f(X_t, I_t))$$
$$\implies$$
$$p^{do(X_t^c)=x}(Y_{t+1} | f(X_t, I_t)) = p^{do(X_t^c)=x'}(Y_{t+1} | f(X_t, I_t)).$$

Proof of Theorem 4.1.

$$p^{do(X_t^c)=x}(Y_{t+1} | f(X_t, I_t))$$
$$= \int p^{do(X_t^c)=x}(Y_{t+1} | Y_{t+1}^R) p^{do(X_t^c)=x}(Y_{t+1}^R | f(X_t, I_t)) dY_{t+1}^R$$
$$= \int p(Y_{t+1} | Y_{t+1}^R) p^{do(X_t^c)=x}(Y_{t+1}^R | f(X_t, I_t)) dY_{t+1}^R$$
$$= \int p(Y_{t+1} | Y_{t+1}^R) p^{do(X_t^c)=x'}(Y_{t+1}^R | f(X_t, I_t)) dY_{t+1}^R$$
$$= p^{do(X_t^c)=x'}(Y_{t+1} | f(X_t, I_t)).$$

The first equality is obtained by marginalising over Y_{t+1}^R and using the assumption of Y_{t+1}^R being a refinement of Y_{t+1}, which implies that Y_{t+1}^R has all the necessary information to predict Y_{t+1} (and thus we can drop the conditioning on $f(X_t, I_t)$). The second equality follows from the fact that the mechanism $Y_{t+1} | Y_{t+1}^R$ is independent of interventions on X_t^c under our assumptions. Finally, the third equality follows from the assumption that $f(X_t, I_t)$ is an invariant predictor of Y_{t+1}^R under changes in X_t^c. □

E. Data Augmentations

In this section we expand on our proposed augmentations, which simulate interventions on X_t^c, valid *until* step t. Further, we report which hints we use in our objective (see Equation (3)) using the naming convention in Veličković et al. (2022a).

DFS-based algorithms (Articulation Points, Bridges, Strongly Connected Components, Topological Sort). We construct exact augmentations for these kinds of problems. First, we sample a step by choosing uniformly at random amongst those where we enter a node for the first time (in case multiple DFSs are being executed for an input, we only consider the first one). Then, we construct a subgraph of nodes with larger node-ids, and we randomly determine connectivity between subgraph's nodes. Finally, we connect all the subgraph's nodes to the node we are entering in the sampled step. We contrast the following hints up to the sampled step (we mask out the contrastive loss on later steps): `pi_h` in Articulation Points and Bridges; `scc_id_h`, `color` and `s_prev` in Strongly Connected Components; and `topo_h`, `color`, and `s_prev` in Topological Sort.

Graph-based algorithms (Bellman-Ford, BFS, DAG Shortest Path, Dijkstra, Floyd-Warshall, MST-Kruskal, MST-Prim). We construct simple but exact augmentations consisting of adding a disconnected subgraph to each input's graph. The subgraph consists of nodes with larger nodes ids and whose connectivity is randomly generated. We contrast until the end of the input's trajectory the following hints: `pi_h` in Bellman-Ford, BFS, Dijkstra and MST-Prim; `pi_h`, `topo_h`, `color` in DAG shortest path; `Pi_h` in Floyd-Warshall; `pi` in MST-Kruskal.

Sorting algorithms (Insertion sort, Bubble Sort, Quicksort, Heapsort). We construct general augmented inputs obtained by simply adding items at the end of each input array. We consider as trajectories for those augmentations the ones of the corresponding inputs. We note that those do not correspond to exact augmentations for all sorting algorithms, but only for Insertion Sort. Indeed, running the executor of one of the other algorithms would yield potentially different trajectories than those we consider. However, since we use the inputs' trajectories, our regularisation aims at learning to be invariant to
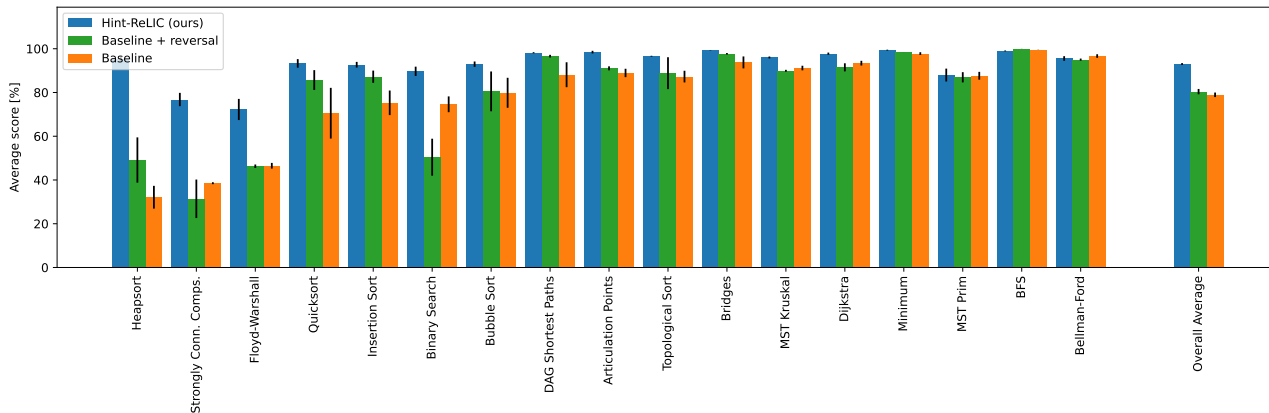
Figure 7. Per-algorithm comparison of the Triplet-GMPNN baseline (Ibarz et al., 2022), its augmented version which includes pointers' reversal and our Hint-ReLIC. Error bars represent the standard error of the mean across three random seeds. The final column shows the average and standard error of the mean performances across the different algorithms.

added nodes that do not contribute to each currently considered step. We contrast until the end of the input's trajectories the hints `pred_h`, and `parent` for Heapsort, and `pred_h` for all the other algorithms.

Searching algorithms (Binary Search, Minimum).   We construct general augmented inputs obtained by simply adding random numbers (different than the searched one) at the end of the input array. For those augmentations, we consider as trajectories the ones of the corresponding input arrays, whose hints are contrasted until the end of the trajectories themselves. We remark that running the searching algorithms on such augmentations could potentially lead to ground-truth trajectories different than those of the inputs. However, since we consider as trajectories for the augmentations the inputs' ones, the contrastive objective is still valid, and can be seen as pushing the hint representations to be invariant to messages coming from nodes that are not involved in the current computation. We run our model by allowing the network to predict the predecessor of every array's item, namely `pred_h`, at every time step and use its representation in our regularisation loss (in other words, we do not run with the static hint elimination of Ibarz et al. (2022)).

F. Experiments

F.1. Additional experiments

Table 3 contains a comprehensive set of experiments, including the performances of the **No Hints**, **Baseline**, **Baseline + reversal** models, as discussed in the main text. The column **Baseline + reversal + contr. + KL** represents our Hint-ReLIC model, which is obtained with the additional inclusion of the contrastive and KL losses (see Equation (3)). Additionally, we report performances of our model when *removing* the KL divergence loss (setting $\alpha = 0$ in Equation (3)), namely **Baseline + reversal + contr.**. By comparing **Baseline + reversal + contr.** to **Baseline + reversal + contr. + KL**, we can see that, even if the KL penalty produces some gain for certain algorithms, it does not represent the component leading to the most improvement. Finally, Table 3 also reports the scores obtained in the DFS algorithm, which appears to be solved by the inclusion of the pointers' reversal. We do not run our contrastive objective on such algorithm as there is no additional improvement to be made.

Finally, to further evaluate the impact of the pointers' reversal, we report the performances of Hint-ReLIC without the inclusion of such additional hints. As can be seen in Table 4, the pointers' reversal helps stabilise our model, especially in the sorting algorithms. We remark however how only including those pointers' reversal into a baseline model does not produce the performances of our model (see column **Baseline + reversal** in Table 3 and Figure 7).

Table 3. Comparison of performances for different models, with last column representing our proposed method Hint-ReLIC. Table shows mean and stderr of OOD micro-F_1 score after 10,000 training steps, across different seeds.

Alg.	No Hints	Baseline	Baseline + reversal	Baseline + reversal + contr.	Baseline + reversal + contr. + KL
Articulation points	$81.97\% \pm 5.08$	$88.93\% \pm 1.92$	$91.04\% \pm 0.92$	$\mathbf{98.91}\% \pm 0.34$	$98.45\% \pm 0.60$
Bridges	$95.62\% \pm 1.03$	$93.75\% \pm 2.73$	$97.70\% \pm 0.34$	$98.14\% \pm 2.00$	$\mathbf{99.32}\% \pm 0.09$
DFS	$33.94\% \pm 2.57$	$39.71\% \pm 1.34$	$\mathbf{100.00}\% \pm 0.00$	$-$	$-$
SCC	$57.63\% \pm 0.68$	$38.53\% \pm 0.45$	$31.40\% \pm 8.80$	$75.78\% \pm 1.25$	$\mathbf{76.79}\% \pm 3.04$
Topological sort	$84.29\% \pm 1.16$	$87.27\% \pm 2.67$	$88.83\% \pm 7.29$	$95.44\% \pm 0.52$	$\mathbf{96.59}\% \pm 0.20$
Bellman-Ford	$93.26\% \pm 0.04$	$\mathbf{96.67}\% \pm 0.81$	$95.02\% \pm 0.49$	$95.26\% \pm 0.92$	$95.54\% \pm 1.06$
BFS	$99.89\% \pm 0.03$	$99.64\% \pm 0.05$	$\mathbf{99.93}\% \pm 0.03$	$98.41\% \pm 0.39$	$99.00\% \pm 0.21$
DAG Shortest Paths	$97.62\% \pm 0.62$	$88.12\% \pm 5.70$	$96.61\% \pm 0.61$	$97.31\% \pm 0.51$	$\mathbf{98.17}\% \pm 0.26$
Dijkstra	$95.01\% \pm 1.14$	$93.41\% \pm 1.08$	$91.50\% \pm 1.85$	$97.22\% \pm 0.12$	$\mathbf{97.74}\% \pm 0.50$
Floyd-Warshall	$40.80\% \pm 2.90$	$46.51\% \pm 1.30$	$46.28\% \pm 0.80$	$71.43\% \pm 2.64$	$\mathbf{72.23}\% \pm 4.84$
MST-Kruskal	$92.28\% \pm 0.82$	$91.18\% \pm 1.05$	$89.93\% \pm 0.43$	$95.18\% \pm 1.29$	$\mathbf{96.01}\% \pm 0.45$
MST-Prim	$85.33\% \pm 1.21$	$87.64\% \pm 1.79$	$86.95\% \pm 2.34$	$\mathbf{89.23}\% \pm 1.23$	$87.97\% \pm 2.94$
Insertion sort	$77.29\% \pm 7.42$	$75.28\% \pm 5.62$	$87.21\% \pm 2.80$	$\mathbf{95.06}\% \pm 1.33$	$92.70\% \pm 1.29$
Bubble sort	$81.32\% \pm 6.50$	$79.87\% \pm 6.85$	$80.51\% \pm 9.10$	$\mathbf{94.09}\% \pm 0.80$	$92.94\% \pm 1.23$
Quicksort	$71.60\% \pm 2.22$	$70.53\% \pm 11.59$	$85.69\% \pm 4.53$	$90.54\% \pm 2.49$	$\mathbf{93.30}\% \pm 1.96$
Heapsort	$68.50\% \pm 2.81$	$32.12\% \pm 5.20$	$49.13\% \pm 10.35$	$89.41\% \pm 4.79$	$\mathbf{95.16}\% \pm 1.27$
Binary Search	$\mathbf{93.21}\% \pm 1.10$	$74.60\% \pm 3.61$	$50.42\% \pm 8.45$	$87.50\% \pm 3.62$	$89.68\% \pm 2.13$
Minimum	$99.24\% \pm 0.21$	$97.78\% \pm 0.63$	$98.43\% \pm 0.01$	$99.54\% \pm 0.05$	$\mathbf{99.37}\% \pm 0.20$

F.2. Implementation details

We use the best hyperparameters of the Triplet-GMPNN (Ibarz et al., 2022) base model, and we only reduce the batch size to 16. We set the temperature parameter τ to $1e - 1$ and the weight of the KL loss α to 1. We implement the similarity function as $\phi(f(x_t^{a_l}, i_t), f(x_t^{a_k}, i_t)) = \langle h(f(x_t^{a_l}), i_t), h(f(x_t^{a_k}), i_t) \rangle / \tau$ with h a two-layers MLP with hidden and output dimensions equal to the input one, and ReLU non-linearities.

Table 4. Importance of the inclusion of the pointers' reversal in our Hint-ReLIC. The table shows mean and stderr of the OOD micro-F_1 score after 10,000 training steps, across different seeds.

Alg.	Hint-ReLIC	Hint-ReLIC (no reversal)
Articulation points	**98.45**% $\pm$ 0.60	97.33% $\pm$ 1.32
Bridges	99.32% $\pm$ 0.09	**99.42**% $\pm$ 0.20
SCC	76.79% $\pm$ 3.04	**81.42**% $\pm$ 2.68
Topological sort	**96.59**% $\pm$ 0.20	80.25% $\pm$ 3.03
Bellman-Ford	**95.54**% $\pm$ 1.06	95.27% $\pm$ 0.97
BFS	**99.00**% $\pm$ 0.21	98.23% $\pm$ 0.17
DAG Shortest Paths	**98.17**% $\pm$ 0.26	89.23% $\pm$ 7.11
Dijkstra	**97.74**% $\pm$ 0.50	96.70% $\pm$ 0.92
Floyd-Warshall	**72.23**% $\pm$ 4.84	57.38% $\pm$ 1.75
MST-Kruskal	**96.01**% $\pm$ 0.45	94.53% $\pm$ 0.40
MST-Prim	**87.97**% $\pm$ 2.94	74.24% $\pm$ 10.85
Insertion sort	**92.70**% $\pm$ 1.29	67.80% $\pm$ 10.86
Bubble sort	**92.94**% $\pm$ 1.23	82.36% $\pm$ 6.88
Quicksort	**93.30**% $\pm$ 1.96	74.32% $\pm$ 10.12
Heapsort	**95.16**% $\pm$ 1.27	77.15% $\pm$ 4.73
Binary Search	**89.68**% $\pm$ 2.13	86.65% $\pm$ 2.38
Minimum	**99.37**% $\pm$ 0.20	98.91% $\pm$ 0.23