Asynchrony Invariance Loss Functions for Graph Neural Networks

Pablo Monteagudo-Lago [*1] **Arielle Rosinski** [*1] **Andrew Dudzik** [2] **Petar Veličković** [1 2]

Editors: *S. Vadgama, E.J. Bekkers, A. Pouplin, S.O. Kaba, H. Lawrence, R. Walters, T. Emerson, H. Kvinge, J.M. Tomczak, S. Jegelka*

Abstract

A ubiquitous class of graph neural networks (GNNs) operates according to the message-passing paradigm, such that nodes systematically broadcast and listen to their neighbourhood. Yet, these synchronous computations have been deemed potentially sub-optimal as they could result in irrelevant information sent across the graph, thus interfering with efficient representation learning. In this work, we devise self-supervised loss functions biasing learning of synchronous GNN-based neural algorithmic reasoners towards representations that are invariant to asynchronous execution. Asynchrony invariance could successfully be learned, as revealed by analyses exploring the evolution of the self-supervised losses as well as their effect on the learned latent embeddings. Our approach to enforce asynchrony invariance constitutes a novel, potentially valuable tool for graph representation learning, which is increasingly prevalent in multiple real-world contexts.

1. Introduction

Graph neural networks (GNNs) constitute deep learning models for representation learning from graph-structured data. Because graph-based abstractions apply to a wide range of real-world domains, GNNs have proven powerful in multiple scientific and industrial settings (Lam et al., 2023; Wang et al., 2023; Veličković, 2023; Veličković et al., 2020). A large class of GNNs operates according to the message-passing framework (Gilmer et al., 2017), where the representation vector of a node is updated based on recursive aggregation and transformation of the neighbouring node vectors.

Traditionally, these GNN operations are performed in a synchronous fashion, that is, nodes necessarily broadcast and listen to their neighbours at every layer, which has been deemed sub-optimal (Faber & Wattenhofer, 2022; Dudzik et al., 2023). Indeed, systematic neighbourhood aggregation can induce over-smoothing of the neighborhood features (i.e., progressively more indistinguishable node representations) as the number of GNN layers increases (Li et al., 2018). Moreover, for ubiquitous tree-based graphs, listening to the complete neighbourhood can also imply that the node receptive field grows exponentially with the number of layers. Yet information becomes compressed into a fixed-size vector, incidentally resulting in information loss referred to as over-squashing (Di Giovanni et al., 2024; Alon & Yahav, 2020). Finally, the representational capacity of GNNs is upper bounded by the Weisfeiler-Lehman (WL; Weisfeiler & Leman 1968) graph isomorphism test (Xu et al., 2018; Morris et al., 2019), which has also been attributed to the uniform treatment of nodes inherent to standard message-passing (Faber & Wattenhofer, 2022). Importantly, these GNN-associated limitations were mitigated by facilitating asynchronous communication between nodes (Faber & Wattenhofer, 2022; Finkelshtein et al., 2023).

In addition to potential expressivity gains, promoting asynchrony in GNNs could be beneficial from an algorithmic alignment perspective. Xu et al. (2019) proposed that neural network architectures whose sub-parts, or modules, align with the sub-routines of the correct reasoning process for a given task will generalise better. Specifically, these authors established a parallel between the computational structure of GNNs and that of the Bellman-Ford algorithm, which could account for better in- and out-of-distribution (OOD) generalisation performance on shortest path-based tasks for GNNs (Dudzik & Veličković, 2022; Xu et al., 2019). Yet, an important caveat remains. For algorithmic tasks such as Bellman-Ford, only a handful of updates would need to be sent at each step. Dynamic programming-aligned reasoning

[*]Equal contribution [1]University of Cambridge [2]Google DeepMind. Correspondence to: Pablo Monteagudo-Lago <pm797@cam.ac.uk>, Arielle Rosinski <ar2217@cam.ac.uk>.

Proceedings of the Geometry-grounded Representation Learning and Generative Modeling Workshop (GRaM) at the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 251, 2024. Copyright 2024 by the author(s).

might thus still be hindered by the synchronous nature of GNN computations.

This possibility was explored by Dudzik et al. (2023), who devised synchronous GNNs whose layers were provably invariant to asynchrony (i.e., asynchronous execution would have produced analogous latent node embeddings). The authors distinguished between increasingly strict levels of asynchrony, which progressively improved OOD performance on the CLRS-30 Algorithmic Reasoning Benchmark (Veličković et al., 2022a).

However, Dudzik et al. (2023) implemented asynchrony invariance axiomatically, deriving conditions that could be met through specific GNN architectural choices. The present study aims at leveraging the asynchrony invariance-associated benefits in Dudzik et al. (2023), while permitting to increase the design space, and hence the expressivity of possible GNN architectures. To this end, we evaluate whether asynchrony invariance can be enforced through self-supervised loss functions.

2. Neural Algorithmic Reasoning

This work focuses on asynchrony invariance-related benefits in a neural algorithmic reasoning (NAR) context (though a broader range of applications is possible). NAR aims at neuralising classical algorithms. Because deep learning networks are more malleable, novel, potentially optimised versions of the initial algorithms can be learned (Veličković & Blundell, 2021; Xu et al., 2019). NAR can also be viewed as being concerned with designing deep learning architectures capable of solving classical algorithmic tasks, thereby enhancing the generalisability and tractability of neural networks.

Veličković & Blundell (2021) proposed a blueprint for NAR based on the encode-process-decode paradigm (Hamrick et al., 2018). An encoder function f maps inputs to high-dimensional latent embeddings. These are then read out by a processor neural network P, which learns to execute the algorithmic task, typically by relying on supervised signals promoting reproduction of the algorithmic output and potentially of the intermediate algorithmic trajectory. P generates output embeddings that are subsequently decoded using a function g. Importantly, while the inputs would typically be artificial during learning, f and g could later be altered (while freezing parameters in P), such that the model would be capable of coping with natural inputs (Veličković et al., 2022b).

Notably, NAR can be probed and compared across distinct GNN-based models in a systematic fashion using the CLRS-30 Algorithmic Reasoning Benchmark introduced by Veličković et al. (2022a). Using the encode-process-decode approach, CLRS-30 implements a collection of algorithmic tasks of differing nature (incl. dynamic programming, sorting, greedy, searching). This benchmark was leveraged here, using settings matching Dudzik et al. (2023) (see Section 4 for architectural details).

3. Self-supervised Asynchrony Invariance

Dudzik et al. (2023) define different forms of asynchrony invariance and identify increasingly tight sufficient conditions for the message passing functions to enforce them. However, the authors acknowledge that these are not necessary conditions. Therefore, the potential richness of the space of functions implementing asynchrony invariance is not fully exploited, which might lead to sub-optimal performance. Moreover, although the node latent representations lie in a high-dimensional space, it is possible that they are distributed in the vicinity of a lower-dimensional manifold (Fefferman et al., 2016). Consequently, enforcing asynchrony invariance in the entire high-dimensional space might not be required, as long as the message passing functions satisfy asynchrony invariance in the subspace where the data lives.

In this work, we study the possibility of weakly enforcing the different forms of asynchrony invariance. To this end, a self-supervised loss is added to the optimisation objective to guide the training procedure towards networks exhibiting asynchrony invariance. Our approach is similar to (Cohen-Karlik et al., 2020), who include a loss term penalising violations of subset-permutation invariance in a recurrent network model, which does not exhibit this property by design. Similarly, Ong & Veličković (2022) introduce a regularisation term to enforce commutativity for a learnable aggregator.

To define our augmented optimisation objective, we rely on the notation and the mathematical foundations in Dudzik et al. (2023). Their theory builds on the notion of *monoid*, which corresponds to a set equipped with a non-invertible operation. This mathematical structure allows to formally describe how the messages *act* on the node states. Additionally, the authors rely on 1-cocycles to formalise enforcement of invariance for argument generation. These can be understood intuitively as mappings satisfying a condition analogous to the Leibniz derivative rule, i.e. $D(fg) = D(f) \cdot g + f \circ D(g)$, where $\cdot$ and $\circ$ define a right and a left action, respectively, over a given monoid. For conciseness purposes, we omit the formal presentation of these notions, and refer the reader to Dudzik et al. (2023, Appendix A).

Equation 1 defines the transformation process of the node embeddings $\{\mathbf{x}_u\}_{u\in\mathcal{V}} \subseteq S$ at each message passing step

$$\mathbf{x}'_u = \phi\left(\mathbf{x}_u, \bigoplus_{v\in N(u)} \psi(\mathbf{x}_u, \mathbf{x}_v)\right) \tag{1}$$

The function $\phi : S \times M \longrightarrow S$ represents the node update, and takes as input the node embeddings at a given point of execution, as well as the aggregated messages from the neighbours of the node. If the set of messages has a monoidal structure $(M, \oplus, 1)$, the node update ϕ can be described in terms of a left action $\bullet : M \times S \longrightarrow S$, so $\phi(s, m) := m \bullet s$. With this notation, two types of asynchrony invariance arise:

1. **Message aggregation asynchrony**. If M is a commutative monoid, the aggregated message does not depend on the order in which the messages are received, as illustrated in Figure 1. In Dudzik et al. (2023), this level of asynchrony is enforced axiomatically, relying on an aggregator such as *sum*, *max* or *mean* (though a learnable commutative aggregator could be used; see Ong & Veličković 2022).
2. **Node update asynchrony**. If ϕ is a left action, it verifies the associativity axiom i.e. $\phi(s, n \oplus m) = \phi(\phi(s, m), n)$. Therefore, upon message reception, the node can be updated asynchronously without requiring to aggregate all of the messages beforehand (Figure 2).

(a)

$$\begin{tikzcd}
m_{ba} \arrow[dr, "\oplus"] & \\
m_{bb} \arrow[r] & m_b \\
m_{bc} \arrow[ur] &
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
m_{ba} \arrow[dr, "\oplus"] & & \\
m_{bb} \arrow[r] & m_{b\{a,b\}} \arrow[r, "\oplus"] & m_b \\
& m_{bc} \arrow[ur] &
\end{tikzcd}$$

Figure 1. (a) Messages from nodes $\{a, b, c\}$ to node b are aggregated synchronously to obtain the message m_b that updates the embeddings of node b. (b) For a commutative aggregator $\oplus$, the resulting message is the same as when the messages from nodes $\{a, b\}$ are first aggregated to obtain $m_{b\{a,b\}}$, and then aggregated with m_{bc}.

In order to enforce node update asynchrony invariance, one possibility is to set $\phi = \oplus$, such that associativity follows from that of $\oplus$ (Dudzik et al., 2023). However, this choice can potentially limit the expressivity of the GNN. It is therefore desirable to guide the learnable node update towards asynchrony invariance during training rather than enforcing associativity axiomatically. Specifically, the violation of the associativity constraint can be measured in terms of the l^2 distance between the node embeddings resulting from updating after having processed all messages, and those obtained after successive updates with the individual messages

(a)

$$\begin{tikzcd}
m_{ba} \arrow[dr, "\oplus"] & & x_b \arrow[dr, "\phi"] & \\
& m_b \arrow[rr] & & x'_b \\
m_{bb} \arrow[ur] & & &
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
& x_b \arrow[dr, "\phi"] & & \\
m_{ba} \arrow[rr, "\oplus"] & & x'_{b\{a\}} \arrow[dr, "\phi"] & \\
m_{bb} \arrow[rrr] & & & x'_b
\end{tikzcd}$$

Figure 2. (a) Node b embeddings are updated upon reception of the message obtained after synchronously aggregating the messages from its neighbours. (b) If ϕ is a left-action, this transformation yields the same embeddings as updating b with the message m_{ba} and then with m_{bb}.

$$R_{\mathrm{L2}} = \mathbb{E}\left[\left\| \, |\phi(s, n \oplus m) - \phi(\phi(s, m), n)| \, \right\|_2 \right] \quad (2)$$

where the expectation is taken with respect to the messages $m, n \in M$ and the node embeddings $s \in S$. Therefore, it can be estimated from samples during training in a Monte-Carlo fashion. The pseudocode enforcing R_{L2} can be found in Appendix D (computations for the upcoming, additional penalty terms are analogous).

If the node update implements a left action, R_{L2} trivially vanishes. Yet, the converse only holds in the support of the message and node embeddings distributions, which is a strength of this method. Indeed, enforcing associativity for all elements of the sets S and M is not strictly required, as it is only necessary for this property to hold in the subset in which the data lives.

Thus far, the discussion on how the messages are generated has been intentionally omitted. From Equation 1, the arguments to the message function are the node embeddings themselves, obtained after processing all of the messages from the previous message-passing step. Still, for promoting a higher level of asynchrony, each node should be able to prepare its arguments for the message function without requiring to process all of its incoming messages beforehand. In this regard, Dudzik et al. (2023) define an argument generation function $\delta : M \times S \longrightarrow A$, where $(A, +, 0)$ denotes the argument monoid, thus explicitly distinguishing the argument generation from the node update. With this notation, a further level of asynchrony can be defined:

3. **Argument generation asynchrony**. If δ verifies that $\delta_1(s) = 0$ and $\delta_{n\oplus m}(s) = \delta_n(m \bullet s) + \delta_m(s)$, then δ supports argument generation asynchrony, as it is illustrated in Figure 3. This is equivalent to enforcing the 1-cocycle condition for the function $D : m \in M \longrightarrow D(m) := \delta_m(\cdot) \in [S, A]$, as proved by Dudzik et al. (2023).

(a)

$$\begin{tikzcd}
m_{ba} \arrow[dr, "\oplus"] & & x_b \arrow[d, "\delta"] \\
& m_b \arrow[r] & a'_b \\
m_{bb} \arrow[ur] & &
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
m_{ba} \arrow[r] & x'_{b\{a\}}, a'_{b\{a\}} \arrow[d, "\phi{,}\delta"'] \arrow[dr, "+"] & x_b \arrow[l, "\phi{,}\delta"'] \\
m_{bb} \arrow[r] & x'_b, a'_{b\{ab\}} \arrow[r] & a'_b
\end{tikzcd}$$

Figure 3. (a) Node update and argument generation at node b upon reception of the message obtained after synchronously aggregating the messages from its neighbours. (b) The 1-cocycle condition for δ ensures that the aggregated partial arguments, generated after processing m_{ba} and then m_{bb}, are identical to those generated by the invocation of the argument generation function with the aggregated messages m_b.

Assuming $S = A$ and $\delta = \phi$, if the 1-cocycle condition is satisfied, A is an idempotent monoid (Dudzik et al., 2023), which limits the choices for the argument aggregation. Alternatively, the constraint $\delta = \phi$ can be lifted and the cocycle condition weakly enforced by including an additional regularisation term in the loss, as defined by Equation 3

$$R^{\text{CO}}_{\text{L3}} = \mathbb{E}\left[\left\|\delta_{n\oplus m}(s) - (\delta_n(\phi(s, m)) +_A \delta_m(s))\right\|_2\right] \tag{3}$$

A final, related, level of asynchrony posits that the message resulting from successive processing of partial arguments is equivalent to that resulting from aggregated arguments:

4. **Message generation asynchrony**. If $\psi : A \times A \longrightarrow M$ is a monoid multimorphism, i.e. a monoid homormorphism in each individual component, the message function can be invoked even when its arguments are not fully ready, as shown in Figure 4. Therefore, $\psi(a_{u_1} + a_{u_2}, a_v) = \psi(a_{u_1}, a_v) \oplus \psi(a_{u_2}, a_v)$, and $\psi(a_u, a_{v_1} + a_{v_2}) = \psi(a_u, a_{v_1}) \oplus \psi(a_u, a_{v_2})$ need to be satisfied, so that ψ is bi-linear.

(a)

$$\begin{tikzcd}
a'_{b\{a\}} \arrow[dr, "+"] & a'_c \arrow[dr, "\psi"] & \\
a'_{b\{a,b\}} \arrow[r] & a'_b \arrow[r] & m'_{bc}
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
a'_{b\{a\}} \arrow[r] & m'_{bc\{a\}} \arrow[ddr, "\oplus"] & \\
a'_c \arrow[ur, "\psi"] \arrow[dr, "\psi"'] & & \\
a'_{b\{a,b\}} \arrow[r] & m'_{bc\{a,b\}} \arrow[r] & m'_{bc}
\end{tikzcd}$$

Figure 4. Comparison of the message generation procedure when the arguments are aggregated before the invocation of the message function (a), with the generation of the message from the partial arguments $a'_{b\{a,b\}}$ and $a'_{b\{a\}}$ (b). For a monoid multimorphism ψ, both operations yield the same message m'_{bc}.

Message generation asynchrony can be enforced axiomatically, for instance by implementing ψ as a log-semiring bilinear layer (Dudzik et al., 2023). Alternatively, the expectation shown in Equation 4 can be used as an additional regularisation term to ensure that ψ is linear in both variables

$$\begin{aligned}
R^{\text{MULT}}_{\text{L3}} = \; & \mathbb{E}\Big[\Big\|\psi(\delta_{n_u}(\phi(s_u, m_u)), s_v) \oplus \psi(\delta_{m_u}(s_u), s_v) \\
& - \psi(\delta_{n_u}(\phi(s_u, m_u)) +_A \delta_{m_u}(s_u), s_v)\Big\|_2\Big] \\
& + \mathbb{E}\Big[\Big\|\psi(s_u, \delta_{n_v}(\phi(s_v, m_v))) \oplus \psi(s_u, \delta_{m_v}(s_v)) \\
& - \psi(s_u, \delta_{n_v}(\phi(s_v, m_v)) +_A \delta_{m_v}(s_v))\Big\|_2\Big]
\end{aligned} \tag{4}$$

The regularisation terms in Equations 2, 3, and 4 can be used to augment the optimisation objective, as shown in Equation 5

$$\mathcal{L}_{\text{AUG}}(w) = \mathcal{L}(w) + \lambda_{\text{L2}} R_{\text{L2}} + \lambda_{\text{L3}} \underbrace{\left(R^{\text{CO}}_{\text{L3}} + R^{\text{MULT}}_{\text{L3}}\right)}_{R_{\text{L3}}} \tag{5}$$

where $\mathcal{L}(w)$ represents the quality loss, and λ_{L2}, λ_{L3} are hyperparameters that define the weight given to each regularisation component in the augmented loss.

4. Architectures

Based on the definitions in Section 3, Dudzik et al. (2023) define three levels of asynchrony invariance for a GNN:

- **Level 1**: The GNN satisfies **message aggregation asynchrony**. As in Dudzik et al. (2023), this level of asynchrony is enforced axiomatically by relying on a commutative aggregator such as *max* or *sum*.
- **Level 2**: The GNN satisfies **Level 1**, as well as **node update asynchrony**. Dudzik et al. (2023) enforce this level by setting $\phi = \oplus = \max$.
- **Level 3**: The GNN satisfies **Level 2**, along with **argument and message generation asynchrony**. To enforce this level axiomatically, Dudzik & Veličković (2022) set $\phi = \oplus = \max$ and use a log-semiring bilinear layer for the message function ψ.

These increasingly strict levels of asynchrony invariance can be weakly enforced using the augmented loss in Equation 5. The present work examines whether a GNN can robustly learn to implement asynchrony invariance through self-supervision.

Moreover, we build upon Dudzik et al. (2023), explicitly distinguishing the argument generation function δ and the node update ϕ, thereby promoting better exploration of the model design space. Additionally, we allow the argument monoid operation $+$ to be different from that of the message monoid $\oplus$. Specific architectures evaluated in the following sections are summarised in Table 1 (see Table B.1 for complete version).

Table 1. GNN architectures. The prefix in the architecture name indicates the level of asynchrony invariance that the network implements by design. In the main text, max-based architectures (e.g., L1-max) are analogous except for $\oplus$ and, if applicable, $+$.

	L1-SUM	L2	L1-δ-SUM	L3
$\oplus$	SUM	MAX	SUM	MAX
ϕ	LINEAR + RELU	MAX	LINEAR + RELU	MAX
δ	ϕ	ϕ	LINEAR + RELU	ϕ
$+$	-	-	SUM	MAX
ψ	LINEAR	LINEAR	LINEAR	LOG-SEMIRING BILINEAR

5. Results

5.1. Quantitative Analyses

Table 2 displays test performance for GNN architectures described in Section 4 (see Table C.3 for full range of architectures). In CLRS-30, training and validation performance, which assess generalisation inside the support of the training data, are distinguished from test performance, which evaluates OOD extrapolation by varying the problem size. That is, input graphs of 16 nodes are presented during training and validation, while 64 nodes are used at test time. All models generally exhibited perfect or near-perfect validation performance (see Table C.2). Yet, this was not the case for OOD generalisation.

The L1-sum GNN was largely outperformed by the L2 model (Table 2), thereby replicating Dudzik et al. (2023). In these cases, high validation performance for L1-sum thus appears to have partly been due to overfitting to the statistics of the training graphs, as opposed to truly having learned the algorithm. Notably, this large performance difference was reduced when using the L1-max architecture (Table 2), which complements previous findings that the max aggregator improves OOD generalisation relative to summation (Veličković et al., 2020; Veličković et al., 2022a; Xu et al., 2020). This result can be understood in terms of algorithmic alignment, since max-aggregation is better aligned to the discrete decisions over neighbourhoods inherent to various algorithms such as Bellman-Ford (Veličković et al., 2020). Further focusing on Bellman-Ford, it was noted that, while the ReLU MLP modules of a GNN can demonstrate learning of nonlinear functions when interpolating (Cybenko, 1989), their predictions become linear outside the support of the training data (Xu et al., 2020). This does not fit well with sum-aggregation, where a nonlinear function must be learned to simulate the algorithm. Conversely, max-aggregation restricts learning to a linear function (Xu et al., 2020), thus promoting better generalisation.

While performance was largely determined by structural architectural properties (e.g., form of $\oplus$), in some cases (e.g., Dijkstra, activity selector, MST Prim), enforcing L2 asynchrony through self-supervision (i.e., $\lambda_{\text{L2}} > 0$) improved OOD performance for the L1-sum architecture (Table 2). Additional analyses enforcing L3 asynchrony (i.e., L1-based architecture with $\lambda_{\text{L}\{2,3\}} > 0$; see Table C.3) also revealed improvements.

Importantly, even in situations where the effects of asynchrony invariance were difficult to infer from OOD performance, this does not imply that it was not learned. This was verified by the qualitative analyses in Sections 5.2 and 5.3.

5.2. Loss Analyses

To gain further insight into the results in Section 5.1, we examined whether R_{L2} and R_{L3} effectively decreased when $\lambda_{\text{L}\{2,3\}} > 0$, which would be indicative of learned asynchrony invariance. These analyses focused on the Bellman-Ford (Bellman, 1958) algorithm, which computes shortest paths from a source node by relaxation. That is, at each step the distances from each node to the source are updated

Table 2. Test (OOD) results for CLRS-30 algorithmic tasks across different GNN architectures. Mean scores across 3 random seeds are displayed, with standard deviations indicated. $\lambda_{L2} > 0$ denotes regularised architecture ($= 0$ otherwise). For regularised models, the hyperparameter settings yielding the highest performance are displayed.

Algorithm	L1-sum	L1-max	L1-sum $\lambda_{L2} > 0$	L2
Bellman Ford	0.65 ± 0.00	0.98 ± 0.00	0.67 ± 0.04	0.97 ± 0.01
Dijkstra	0.41 ± 0.17	0.77 ± 0.16	0.53 ± 0.07	0.96 ± 0.01
Task Scheduling	0.82 ± 0.01	0.68 ± 0.24	0.81 ± 0.02	0.92 ± 0.03
MST Prim	0.37 ± 0.10	0.69 ± 0.07	0.45 ± 0.08	0.71 ± 0.04
BFS	0.92 ± 0.04	1.00 ± 0.00	0.87 ± 0.09	1.00 ± 0.00
Activity Selector	0.56 ± 0.06	0.85 ± 0.04	0.77 ± 0.00	0.69 ± 0.08
Topological Sort	0.26 ± 0.14	0.68 ± 0.12	0.31 ± 0.11	0.47 ± 0.03

with better approximations until convergence. Focusing on Bellman-Ford was motivated by its alignment with the computation structure of GNNs (see Xu et al. 2019).

The evolution of the quality loss (i.e., $\mathcal{L}(w)$) as well as of the regularised losses are displayed in Figures 5 (for R_{L2}), and 6 (for R_{L3}). For illustration purposes, the results shown are restricted to comparing non-regularised with regularised versions of the L1-δ-sum architecture, since this model incorporated both R_{L2} and R_{L3}. Yet, analogous loss patterns were observed for the broader range of GNNs described in this work.

Notably, without self-supervision, R_{L2} and R_{L3} do not decrease throughout training. This supports the absence of an inductive bias for asynchrony invariance in the GNN architecture, thereby suggesting that potential asynchrony-related effects would arise from the self-supervised procedure. This is reinforced by the decrease in R_{L2} and R_{L3} when enforcing asynchrony invariance. Crucially, the network appeared to leverage asynchrony invariant transformations in both in- and out-of-distribution settings. Indeed, R_{L2} and R_{L3} were also associated with low (i.e., $< 10^{-1}$) values at test time. Therefore, overall, these findings indicate that Level 2 (Figure 5) and Level 3 (Figure 6) asynchrony invariance (Dudzik et al., 2023) can be learned in a self-supervised fashion.

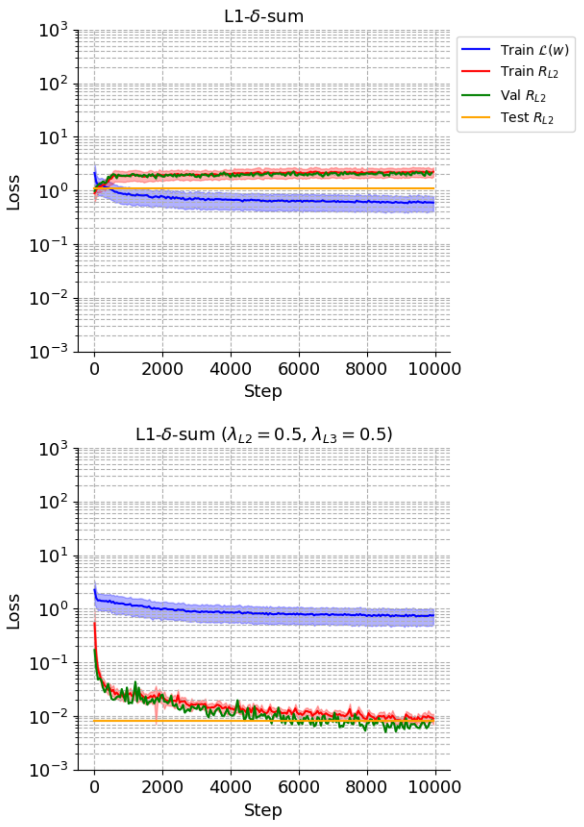

Figure 5. Loss evolution for $\mathcal{L}(w)$ (training) as well as for R_{L2} at training, validation and test time for L1-δ-sum without (upper) and with (lower) self-supervised asynchrony invariance enforcement ($\lambda_{L2} = \lambda_{L3} = 0.5$). For illustration purposes, training losses elements have been grouped in 50-iteration bins (i.e., means are displayed, with standard deviations indicated by the fill). Similarly, 2-iteration bins are displayed for the validation loss. The mean loss is shown for test.

5.3. Latent Space Representations

To further probe the putative influence of learned asynchrony invariance, we examined the latent space representations of the neural processor at test time. Specifically, we leveraged the fact that the three regularisation terms in Section 3 are expressed in terms of the distance between embedding pairs. For instance, the regularisation term R_{L2} in Equation 2 involves the difference between the node embeddings after the update with the aggregated messages $\phi(s, n \oplus m)$ and the embeddings after subsequent updates $\phi(\phi(s, m), n)$. Therefore, we retrieved the trajectories of these embedding pairs across message passing steps, and jointly represented their evolution by applying PCA and keeping the two most dominant directions at each computation step. We filtered the graphs terminating at the same step T, and aggregated the trajectories across the node dimensions using max (see Mirjanić et al. 2023 for a similar approach). This yielded embeddings of shape $N \times D \times T$, where N is the number of sampled execution graphs and D is the dimensionality of the latent representations (set to $D = 128$).

For the L1-δ-sum architecture, if $\lambda_{L2} = 0$, the embeddings for each loss component grouped in clearly distinct clusters, as shown in Figure 7 (upper). This demonstrates that the node embeddings under asynchronous execution will necessarily diverge from those obtained by synchronously performing the message passing steps, thus possibly leading to a misalignment with the underlying algorithm. This was not the case with regularisation ($\lambda_{L2} > 0$), as evidenced by the node embeddings for synchronous and asynchronous

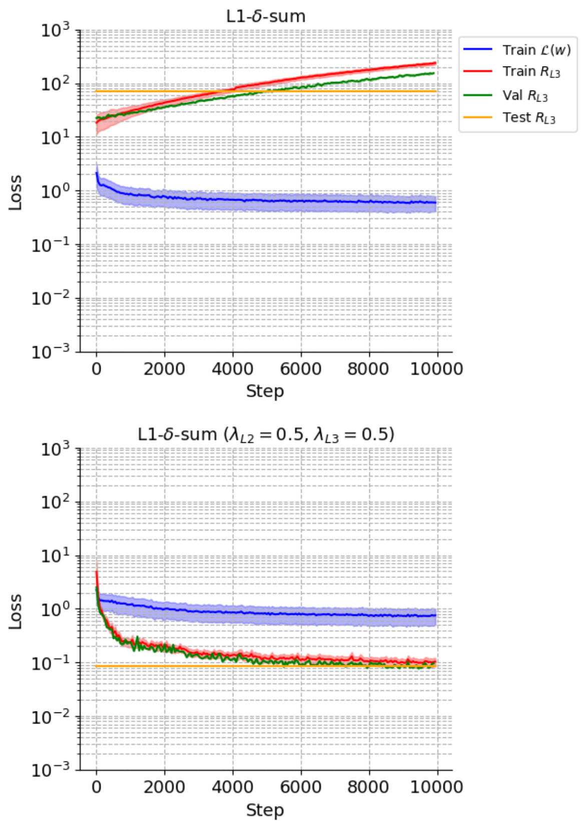

Figure 6. Same as Figure 5 but for R_{L3}.

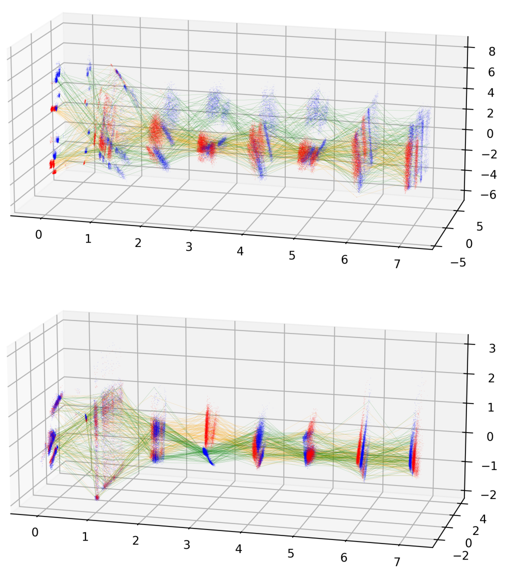

Figure 7. Step-wise PCA visualisations of the node embeddings $\phi(s, n \oplus m)$ (red) and $\phi(\phi(s, m), n)$ (blue) used to compute the associativity regularisation term R_{L2} (Equation 2) during Bellman-Ford execution at test time. The architecture L1-δ-sum was used, as defined in Table B.1 with no regularisation (upper) and $\lambda_{\text{L2}} = \lambda_{\text{L3}} = 0.5$ (lower).

execution clustering together (Figure 7; lower). Analogous results were obtained for the Level 3 asynchrony invariance regularisation terms, as shown in Figures 8 and 9. Indeed, for the regularised model ($\lambda_{\text{L3}} > 0$), the gap between synchronous and asynchronous trajectories was reduced relative to the non-regularised model.

Analogous analyses for the L1-δ-max architecture can be found in Appendix A. In that case, the two clusters grouped together almost perfectly at every step, as shown in Figures A.1, A.2 and A.3.

Regularisation towards asynchrony invariance therefore appeared to alter the structure of the processor computations, a phenomenon also observed through tracking the global execution trajectory of the node embeddings (see Figure E.4).

6. Conclusion & Future Work

This work has designed self-supervised, asynchrony invariance-enforcing loss functions for GNNs. Notably, asynchrony-related benefits had already been demonstrated. Indeed, altering the standard message-passing scheme to facilitate asynchronous communication enhanced GNN expressivity and mitigated GNN-associated limitations (Finkelshtein et al., 2023; Faber & Wattenhofer, 2022). Yet, asynchronous execution implied reduced scalability relative to synchronous alternatives (Dudzik et al., 2023). This was addressed in Dudzik et al. (2023), who built synchronous GNN architectures provably asynchrony invariant rather than asynchronous. Yet, while these authors implemented axiomatically-defined asynchrony invariance using specific architectural settings, GNNs under our proposed self-supervised approach span a broader design space.

It should still be noted that learning asynchrony invariant embeddings did not appear to systematically improve OOD performance on all of the algorithmic tasks evaluated here. In the future, greater OOD difficulty (e.g., problem size > 64) could be explored. This could increase the window for potential asynchrony-related performance improvements. Beyond problem size (Veličković et al., 2020), extrapolation could be investigated on graphs with unseen structure, edge weights, as well as node features (Xu et al., 2020), as different tasks could benefit from asynchrony to differing extents.

Still, overall, the present work yields novel insights regarding the ability of GNNs to converge towards asynchrony invariant embeddings. To our knowledge, this had not yet been achieved in a self-supervised fashion.

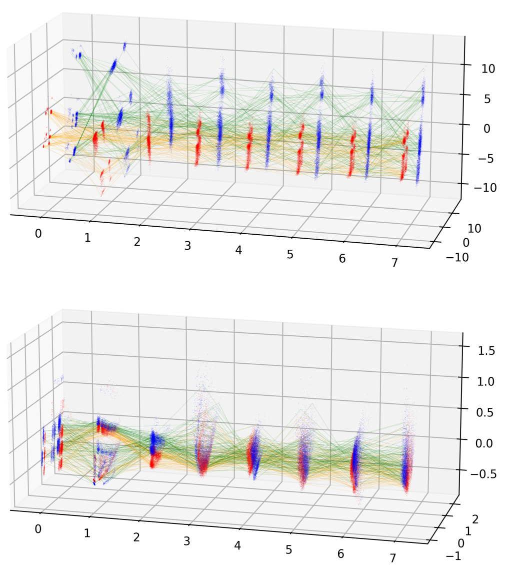

Figure 8. Same as Figure 7 but for the argument embeddings $\delta_{n \oplus m}(s)$ (red) and $\delta_n(m \bullet s) + \delta_m(s)$ (blue) used to compute the 1-coycle regularisation term $R_{\text{L3}}^{\text{CO}}$ (Equation 3).

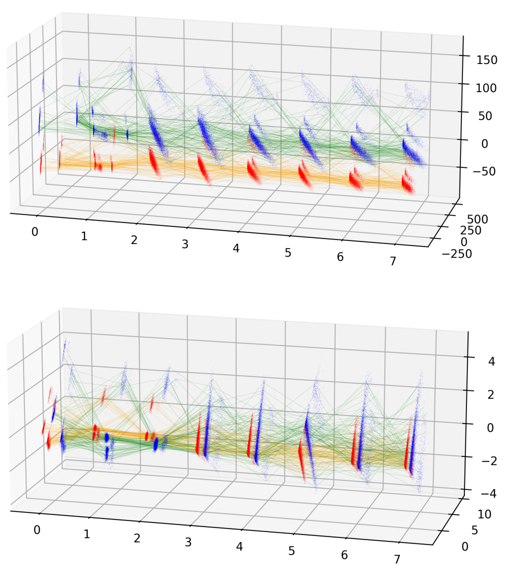

Figure 9. Same as Figure 7 but for the message embeddings $\psi(a_{u_1} + a_{u_2}, a_v)$ (red) and $\psi(a_{u_1}, a_v) \oplus \psi(a_{u_2}, a_v)$ (blue) used to compute the multimorphism regularisation term $R_{\text{L3}}^{\text{MULT}}$ (Equation 4).

References

Alon, U. and Yahav, E. On the bottleneck of graph neural networks and its practical implications. *arXiv*, 2020.

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

Cohen-Karlik, E., Ben David, A., and Globerson, A. Regularizing towards permutation invariance in recurrent models. *Advances in Neural Information Processing Systems*, 33:18364–18374, 2020.

Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Di Giovanni, F., Rusch, T. K., Bronstein, M. M., Deac, A., Lackenby, M., Mishra, S., and Veličković, P. How does over-squashing affect the power of gnns? *arXiv*, 2024.

Dudzik, A., von Glehn, T., Pascanu, R., and Veličković, P. Asynchronous algorithmic alignment with cocycles. *arXiv*, 2023.

Dudzik, A. J. and Veličković, P. Graph neural networks are dynamic programmers. *Advances in Neural Information Processing Systems*, 35:20635–20647, 2022.

Faber, L. and Wattenhofer, R. Asynchronous neural networks for learning in graphs. *arXiv*, 2022.

Fefferman, C., Mitter, S., and Narayanan, H. Testing the manifold hypothesis. *Journal of the American Mathematical Society*, 29(4):983–1049, 2016.

Finkelshtein, B., Huang, X., Bronstein, M., and Ceylan, İ. İ. Cooperative graph neural networks. *arXiv*, 2023.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pp. 1263–1272. PMLR, 2017.

Hamrick, J. B., Allen, K. R., Bapst, V., Zhu, T., McKee, K. R., Tenenbaum, J. B., and Battaglia, P. W. Relational inductive bias for physical construction in humans and machines. *arXiv*, 2018.

Lam, R., Sanchez-Gonzalez, A., Willson, M., Wirnsberger, P., Fortunato, M., Alet, F., Ravuri, S., Ewalds, T., Eaton-Rosen, Z., Hu, W., et al. Learning skillful medium-range global weather forecasting. *Science*, 382(6677):1416–1421, 2023.

Li, Q., Han, Z., and Wu, X.-M. Deeper insights into graph convolutional networks for semi-supervised learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Mirjanić, V. V., Pascanu, R., and Veličković, P. Latent space representations of neural algorithmic reasoners. 2023.

Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., and Grohe, M. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pp. 4602–4609, 2019.

Ong, E. and Veličković, P. Learnable commutative monoids for graph neural networks. In *Learning on Graphs Conference*, pp. 43–1. PMLR, 2022.

Veličković, P. Everything is connected: Graph neural networks. *Current Opinion in Structural Biology*, 79:102538, 2023.

Veličković, P. and Blundell, C. Neural algorithmic reasoning. *Patterns*, 2(7), 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*, pp. 22084–22102. PMLR, 2022a.

Veličković, P., Bošnjak, M., Kipf, T., Lerchner, A., Hadsell, R., Pascanu, R., and Blundell, C. Reasoning-modulated representations. In *Learning on Graphs Conference*, pp. 50–1. PMLR, 2022b.

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

Wang, H., Fu, T., Du, Y., Gao, W., Huang, K., Liu, Z., Chandak, P., Liu, S., Van Katwyk, P., Deac, A., et al. Scientific discovery in the age of artificial intelligence. *Nature*, 620(7972):47–60, 2023.

Weisfeiler, B. and Leman, A. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? *arXiv*, 2018.

Xu, K., Li, J., Zhang, M., Du, S. S., Kawarabayashi, K.-i., and Jegelka, S. What can neural networks reason about? *arXiv*, 2019.

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. *arXiv*, 2020.

A. Latent Space Representations (L1-δ-max)

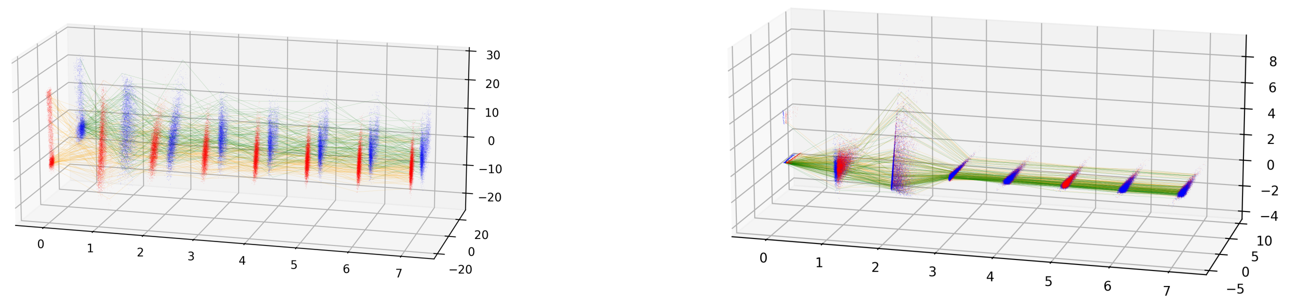

Figure A.1. Step-wise PCA visualisations of the node embeddings $\phi(s, n \oplus m)$ (red) and $\phi(\phi(s, m), n)$ (blue) used to compute the associativity regularisation term R_{L2} (Equation 2) during Bellman-Ford execution at test time. The architecture L1-δ-max was used, as defined in Table B.1 with no regularisation (left) and $\lambda_{\text{L2}} = \lambda_{\text{L3}} = 0.5$ (right).

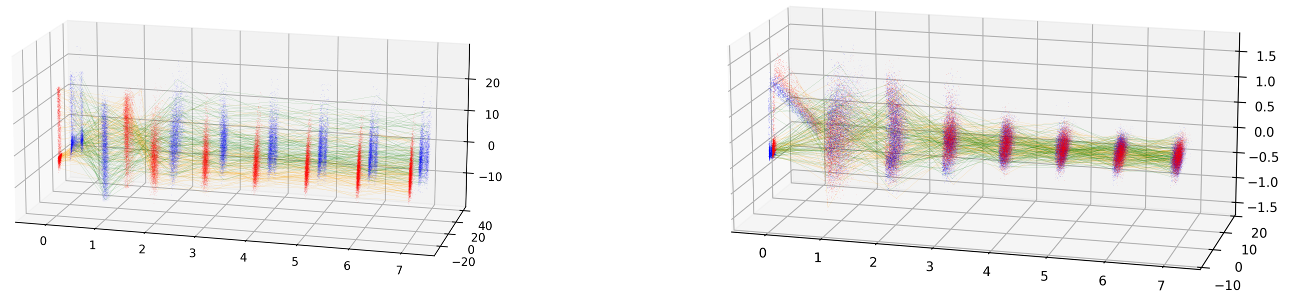

Figure A.2. Same as Figure A.1 but for the argument embeddings $\delta_{n\oplus m}(s)$ (red) and $\delta_n(m \bullet s) + \delta_m(s)$ (blue) used to compute the 1-coycle regularisation term $R_{\text{L3}}^{\text{CO}}$ (Equation 3).

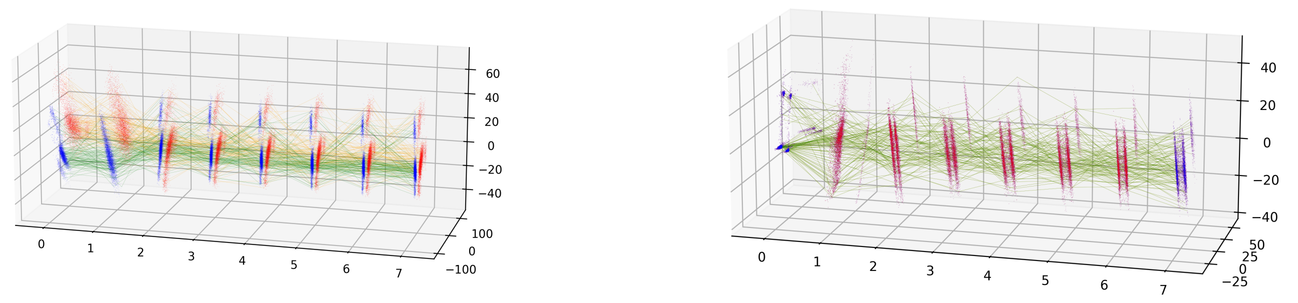

Figure A.3. Same as Figure A.1 but for the message embeddings $\psi(a_{u_1} + a_{u_2}, a_v)$ (red) and $\psi(a_{u_1}, a_v) \oplus \psi(a_{u_2}, a_v)$ (blue) used to compute the multimorphism regularisation term $R_{\text{L3}}^{\text{MULT}}$ (Equation 4).

B. Architectures

Table B.1. GNN architectures (incl. those in Table 1). The prefix in the architecture name indicates the level of asynchrony invariance that the network implements by design (Dudzik et al., 2023).

	L1-sum	L1-max	L2	L1-δ-sum	L1-δ-max	L2-δ-sum	L2-δ-max	L3
$\oplus$	sum	max	max	sum	max	max	max	max
ϕ	Linear + ReLU	Linear + ReLU	max	Linear + ReLU	Linear + ReLU	max	max	max
δ	ϕ	ϕ	ϕ	Linear + ReLU	Linear + ReLU	Linear + ReLU	Linear + ReLU	ϕ
$+$	-	-	-	sum	max	sum	max	max
ψ	Linear	Linear	Linear	Linear	Linear	Linear	Linear	Log-semiring bilinear (Dudzik et al., 2023)

C. Quantitative Results

C.1. Validation

Table C.2. Validation results for CLRS-30 algorithmic tasks across different GNN architectures. Mean scores across 3 random seeds are displayed, with standard deviations indicated. $\lambda_{\mathrm{L}\{2,3\}} > 0$ denotes regularised architectures ($= 0$ otherwise). For regularised models, the λ_{L2} and/or λ_{L3} hyperparameter settings yielding the highest test performance are displayed.

Algorithm	L1-sum	L1-max	L1-sum $\lambda_{\mathrm{L2}} > 0$	L1-max $\lambda_{\mathrm{L2}} > 0$	L2	L1-δ-sum $\lambda_{\mathrm{L}\{2,3\}} > 0$	L1-δ-max $\lambda_{\mathrm{L}\{2,3\}} > 0$	L2-δ-sum $\lambda_{\mathrm{L3}} > 0$	L2-δ-max $\lambda_{\mathrm{L3}} > 0$	L3
Bellman Ford	0.90 ± 0.02	0.99 ± 0.00	0.89 ± 0.02	0.98 ± 0.01	0.98 ± 0.00	0.87 ± 0.03	0.98 ± 0.01	0.98 ± 0.01	0.98 ± 0.00	0.95 ± 0.02
Dijkstra	0.95 ± 0.01	0.98 ± 0.01	0.96 ± 0.01	0.97 ± 0.01	0.98 ± 0.00	0.95 ± 0.00	0.97 ± 0.00	0.97 ± 0.00	0.97 ± 0.01	0.96 ± 0.00
Task Scheduling	0.99 ± 0.01	0.99 ± 0.01	0.98 ± 0.00	0.97 ± 0.01	0.98 ± 0.01	0.98 ± 0.01	1.00 ± 0.00	0.98 ± 0.01	0.95 ± 0.03	0.85 ± 0.03
MST Prim	0.92 ± 0.01	0.96 ± 0.00	0.92 ± 0.01	0.96 ± 0.01	0.93 ± 0.02	0.88 ± 0.02	0.95 ± 0.01	0.94 ± 0.01	0.94 ± 0.01	0.87 ± 0.01
BFS	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	0.99 ± 0.00
Activity Selector	0.93 ± 0.00	0.94 ± 0.02	0.93 ± 0.01	0.95 ± 0.03	0.89 ± 0.00	0.90 ± 0.03	0.91 ± 0.01	0.96 ± 0.01	0.94 ± 0.03	0.79 ± 0.06
Topological Sort	0.97 ± 0.03	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.83 ± 0.13	0.93 ± 0.02	0.98 ± 0.01	0.99 ± 0.00	0.95 ± 0.01	0.72 ± 0.11

C.2. Test

Table C.3. Test (OOD) results for CLRS-30 algorithmic tasks across different GNN architectures (incl. results from Table 2). Mean scores across 3 random seeds are displayed, with standard deviations indicated. $\lambda_{\mathrm{L}\{2,3\}} > 0$ denotes regularised architectures ($= 0$ otherwise). For regularised models, the λ_{L2} and/or λ_{L3} hyperparameter settings yielding the highest test performance are displayed (for L1-sum and L1-max: $\lambda_{\mathrm{L2}} \in \{0.05, 0.1, 0.5, 1\}$ was implemented, for L2-δ-sum and L2-δ-max: $\lambda_{\mathrm{L3}} \in \{0.05, 0.1, 0.5, 1\}$, and for L1-$\delta$-sum and L1-$\delta$-max: $\lambda_{\mathrm{L2}} \in \{0.05, 0.1, 0.5\}$ and $\lambda_{\mathrm{L3}} \in \{0.05, 0.1, 0.5\}$)

Algorithm	L1-sum	L1-max	L1-sum $\lambda_{\mathrm{L2}} > 0$	L1-max $\lambda_{\mathrm{L2}} > 0$	L2	L1-δ-sum $\lambda_{\mathrm{L}\{2,3\}} > 0$	L1-δ-max $\lambda_{\mathrm{L}\{2,3\}} > 0$	L2-δ-sum $\lambda_{\mathrm{L3}} > 0$	L2-δ-max $\lambda_{\mathrm{L3}} > 0$	L3
Bellman Ford	0.65 ± 0.00	0.98 ± 0.00	0.67 ± 0.04	0.97 ± 0.00	0.97 ± 0.01	0.56 ± 0.03	0.97 ± 0.01	0.97 ± 0.00	0.98 ± 0.00	0.90 ± 0.02
Dijkstra	0.41 ± 0.17	0.77 ± 0.16	0.53 ± 0.07	0.97 ± 0.01	0.96 ± 0.01	0.55 ± 0.04	0.95 ± 0.01	0.95 ± 0.01	0.94 ± 0.01	0.82 ± 0.01
Task Scheduling	0.82 ± 0.01	0.68 ± 0.24	0.81 ± 0.02	0.71 ± 0.25	0.92 ± 0.03	0.80 ± 0.00	0.84 ± 0.00	0.88 ± 0.00	0.92 ± 0.02	0.63 ± 0.24
MST Prim	0.37 ± 0.10	0.69 ± 0.07	0.45 ± 0.08	0.78 ± 0.01	0.71 ± 0.04	0.31 ± 0.07	0.80 ± 0.01	0.70 ± 0.01	0.75 ± 0.02	0.48 ± 0.11
BFS	0.92 ± 0.04	1.00 ± 0.00	0.87 ± 0.09	1.00 ± 0.00	1.00 ± 0.00	0.89 ± 0.05	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	0.99 ± 0.01
Activity Selector	0.56 ± 0.06	0.85 ± 0.04	0.77 ± 0.00	0.88 ± 0.04	0.69 ± 0.08	0.74 ± 0.03	0.91 ± 0.01	0.88 ± 0.01	0.86 ± 0.01	0.56 ± 0.07
Topological Sort	0.26 ± 0.14	0.68 ± 0.12	0.31 ± 0.11	0.64 ± 0.19	0.47 ± 0.03	0.35 ± 0.04	0.47 ± 0.18	0.48 ± 0.04	0.36 ± 0.06	0.64 ± 0.04

D. Pseudocode for $R_{\mathbf{L2}}$ penalty computation

The regularisation terms are expressed in terms of expectations, and thus they can approximated by uniformly sampling a subset of node embeddings and messages at each training step. Note that if the sampling was not uniform, weaker types of asynchrony invariance might be induced during training, e.g. the node update only behaving as a monoid action at a subset of steps in the algorithm execution.

Note that computational cost of estimating the regularisation is proportional to the number of messages, nodes and arguments sampled at each training step. For instance, here, asynchrony invariance was learned by sampling a small fraction of nodes (≈ 0.2) and messages (≈ 0.1) at each training step, though alternative settings might prove more optimal for other problem types.

Algorithm 1 R_{L2} penalty computation

```
Require: Number of steps in algorithm execution $T$.
Require: Graph $G = (V, E)$ with nodes $V$ and edges $E$.
Require: Node features $\mathbf{x}_u^{(t)}$ and messages $\mathbf{m}_{uv}^{(t)}$ at each point of execution.
 1: $R_{L2} = 0$
 2: for $t = 1$ to $T$ do
 3:     Sample a set of nodes $S \subseteq V$ uniformly from $V$
 4:     for each node $u \in S$ do
 5:         Sample a set of neighbours $N'(u) \subset N(u)$
 6:         Aggregate messages from subset of neighbours:
                $$\mathbf{m}_u^{(t)} = \bigoplus_{v \in N'(u)} \mathbf{m}_{uv}^{(t)}$$
 7:         Update node state with aggregated messages:
                $$\mathbf{x}_{u_1}^{(t)} = \phi\left(\mathbf{x}_u^{(t)}, \mathbf{m}_u^{(t)}\right)$$
 8:         Set $\mathbf{x}_{u_2}^{(t)} = \mathbf{x}_u^{(t)}$
 9:         for each neighbour $v \in N'(u)$ do
10:             Update partial node state with individual messages:
                $$\mathbf{x}_{u_2}^{(t)} = \phi\left(\mathbf{x}_{u_2}, \mathbf{m}_{uv}^{(t)}\right)$$
11:         end for
12:         Update $R_{L2}$ regularisation penalty:
                $$R_{L2} = R_{L2} + \left|\left|\mathbf{x}_{u_1}^{(t)} - \mathbf{x}_{u_2}^{(t)}\right|\right|_2$$
13:     end for
14: end for
15: return $R_{L2}$
```

E. Step-wise PCA

We replicated the step-wise PCA visualisations in Mirjanić et al. (2023) to investigate how regularisation towards asynchrony invariance modifies the global execution trajectory of the node embeddings. Again, this is shown for non-regularised and regularised versions of L1-δ-sum (Figure E.4). In both cases, clearly defined computation steps can be observed. Yet, without regularisation, the distances travelled by the node embeddings at each step tend to decrease, exhibiting convergence towards a particular region of the space, in line with Mirjanić et al. (2023). On the other hand, with regularisation, the decrease in the distances travelled at each computation step is not as dramatic. Additionally, the clusters at each computation are more compact. Therefore, regularisation towards invariance appears to significantly change the structure of the processor's computations.

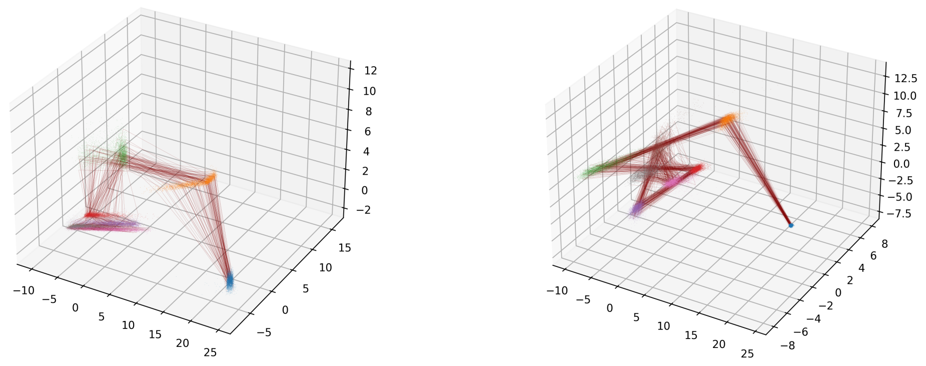

Figure E.4. Global step-wise PCA visualisations of the node embeddings for the architecture L1-δ-sum, as defined in Table B.1 with no regularisation (left) and $\lambda_{\text{L2}} = \lambda_{\text{L3}} = 0.5$ (right).