Asynchronous Algorithmic Alignment with Cocycles

Andrew Dudzik
Google DeepMind
adudzik@google.com

Tamara von Glehn
Google DeepMind
tamaravg@google.com

Razvan Pascanu
Google DeepMind
razp@google.com

Petar Veličković
Google DeepMind
petarv@google.com

Abstract

State-of-the-art neural algorithmic reasoners make use of message passing in graph neural networks (GNNs). But typical GNNs blur the distinction between the definition and invocation of the message function, forcing a node to send messages to its neighbours at every layer, synchronously. When applying GNNs to learn to execute dynamic programming algorithms, however, on most steps only a handful of the nodes would have meaningful updates to send. One, hence, runs the risk of inefficiencies by sending too much irrelevant data across the graph. But more importantly, many intermediate GNN steps have to learn the identity functions, which is a non-trivial learning problem. In this work, we explicitly separate the concepts of node state update and message function invocation. With this separation, we obtain a mathematical formulation that allows us to reason about asynchronous computation in both algorithms and neural networks. Our analysis yields several practical implementations of synchronous scalable GNN layers that are provably invariant under various forms of asynchrony.

1 Introduction

The message passing primitive—performing computation by aggregating information sent between neighbouring entities [1, 2]—is known to be remarkably powerful. Message passing is the core primitive in *graph neural networks* [3, GNNs], a prominent family of deep learning models. Owing to the ubiquity of graphs as an abstraction for describing the *structure* of systems, GNNs have enjoyed immense popularity across both scientific [4] and industrial applications, including novel drug screening [5, 6], designing next-generation machine learning chips [7], serving travel-time estimates [8], particle physics [9], and settling long-standing problems in pure mathematics [10–12].

Another active area of research for GNNs is *neural algorithmic reasoning* [13, NAR]. NAR seeks to design neural network architectures that capture *classical computation*, largely by learning to execute it [14]. This is an important problem in the light of today's large-scale models, as they tend to struggle in performing exactly the kinds of computations that classical algorithms can trivially capture [15].

The use of GNNs in NAR is largely due to *algorithmic alignment* [16]: the observation that, as we increase the structural similarity between a neural network and an algorithm, it will be able to learn to execute this algorithm with improved sample complexity. Here, we make novel contributions to the theory of algorithmic alignment. Our approach "zooms in" on the theoretical analysis in [17], which analysed computations—of both algorithms and GNNs—globally. Instead, we center our discussion on a *node-centric*[1] view: analysing computations around individual nodes in the graph, in isolation.

This view allows us to study message passing under various synchronisation regimes and can help us identify choices of message functions that better align with target algorithms, in a manner that was not possible under previous frameworks—indeed, it allows us to theoretically justify the unreasonable effectiveness of architectures such as PathGNNs [20]. We refer to our new framework as *asynchronous algorithmic alignment*, and formalise it using tools of category theory, monoid actions, and cocycles. To visualise what executing an asynchronous GNN might look like, refer to Figure 1.

[1] As we will expand on later in the work, *node* is a misnomer for what we precisely mean, but to improve the exposition we will rely on this term for now.

A. Dudzik et al., Asynchronous Algorithmic Alignment with Cocycles. *Proceedings of the Second Learning on Graphs Conference (LoG 2023)*, PMLR 231, Virtual Event, November 27–30, 2023.

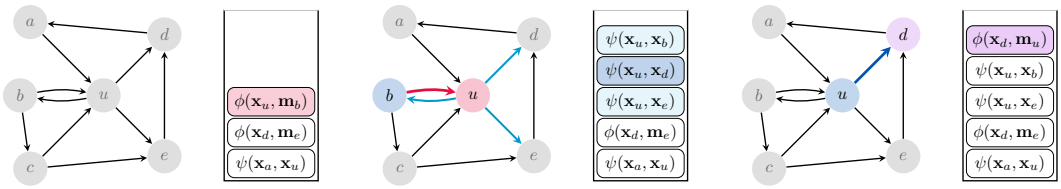


Figure 1: A possible execution trace of an asynchronous GNN. While traditional GNNs send and receive all messages synchronously, under our framework, at any step the GNN may choose to execute any number of possible operations (depicted here with a collection on the right side of the graph). Note that we do *not* aim to *implement* an asynchronous GNN—a feat concurrently explored by AMP [18] and Co-GNN [19]—rather, we seek to build *synchronous* GNNs that are *invariant*—i.e., will yield identical output node embeddings—under various forms of asynchronous execution.

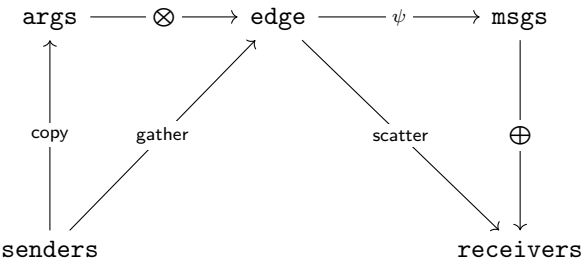
2 Message passing and (asynchronous) algorithms: Addition with carry

To better understand the concept of algorithmic alignment, and in particular what we mean by a *node-centric* perspective, we will utilise a simple illustrative example—namely *addition with carry*—and present it in the framework of message passing that GNNs rely on.

Message passing framework. We will use the definition of GNNs based on Bronstein et al. [21]. Let a graph be a tuple of *nodes* and *edges*, $G = (V, E)$, with one-hop neighbourhoods defined as $\mathcal{N}_u = \{v \in V \mid (v, u) \in E\}$. Further, a node feature matrix $\mathbf{X} \in \mathbb{R}^{|V| \times k}$ gives the features of node u as $\mathbf{x}_u$; we omit edge- and graph-level features for clarity. A *(message passing)* GNN over this graph is then executed as:

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

To put this equation in more abstract terms, we review the diagram of the message-passing framework of Dudzik and Veličković [17], with the addition of the message function ψ to emphasise symmetry:



First, sender features (`senders`) are duplicated along outgoing edges to form the arguments (`args`) to a message function ψ. These arguments are collected into a list using the $\bigotimes$ operator—which is traditionally a concatenation, though as per Dudzik and Veličković [17], it can be any operator with a monoidal structure. This list of arguments now lives on a new, transient, `edge` datatype. These two steps constitute a *gather* operation. In Equation 1, this corresponds to copying the node features in $\mathbf{X}$, considered as a V-indexed tensor, to obtain feature pairs $(\mathbf{x}_u, \mathbf{x}_v)$, as an E-indexed tensor.

Note that the `senders` do not necessarily correspond to the more established notion of "sender nodes" in graph representation learning [22]—rather, we consider a node to be a sender if its features are *necessary* to compute the message function ψ. Hence, in Equation 1, we consider both the features of nodes u and v to be "senders", rather than assuming that only v is a sender. This allows us to easily extend this idea to messages spanning arbitrary numbers of inputs living on various places in the graph. For example, if edge features are used for ψ, we can include them as part of `senders` also.

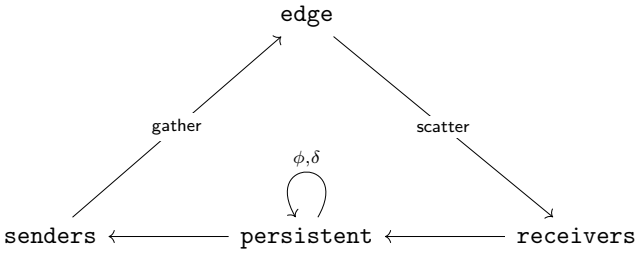
Next, we perform a similar operation, a *scatter*, by first applying the message function ψ to the arguments, which computes the messages to be sent (`msgs`). Then, messages are copied to suitable receivers, which aggregate along their incoming edges to form the final set of receiver node features (`receivers`). In Equation 1, this refers to the application of the message function ψ, and the aggregation $\bigoplus$. This general gather-scatter paradigm can be seen throughout the literature on message passing, for example in the sheaf Laplacian of Bodnar et al. [23].

Adding graph- and edge-level features complicates our exposition, also potentially creating some confusion within the diagrams above. In this work we will argue that, in order to simplify our theoretical treatment of the computations carried by GNNs and algorithms, *it is useful to lift any graph component that has a **persistent** state—e.g. a feature vector updated at every layer—to the status of **node**, while **edges** are **transient** features created as part of the model's information flow*. For any graph, is trivial to construct a corresponding graph that has the above property. From now on, we assume that anything with a persistent state is a node in the graph, while edges are always transient.

Asynchronicity and constructing arguments. Now we discuss the *addition with carry* algorithm on a high level—we will focus on its details in Section 3. Intuitively, digits would represent nodes with persistent state, while edges show how computation needs to be carried, i.e. which digits need to be added. Because of the carry, we quickly notice that there is a misalignment between the algorithm and its current graph representation. On one hand, the algorithm requires a certain level of asynchrony, where digits need to be added in sequence, in order to be able to correctly accumulate the carry. The parallel computation imposed by having all nodes sending messages within their neighbourhoods, synchronously, can be problematic, requiring many nodes to learn the identity function on most steps. Fitting such sparse-update computation with (G)NNs has led to solutions that were either very brittle [24, 25] or requiring vast amounts of strong supervision [26, 27]. This problem gets even further exacerbated when GNNs are used to fit multiple algorithms at once [28, 29].

A second distinct issue in terms of executing the algorithm is: how will the carry be handled? Indeed, assuming the carry can not be added to the state of the node (which is meant to be only a single digit), the model is unable to properly represent the required computation.

To resolve this issue we can focus on how messages and the persistent state of the node get transformed into new states and *arguments* for the message function ψ. In particular we augment the computation of the node in order to capture not only how the new state is constructed but also how the node constructs *new arguments* for the next computational step, allowing us to chain these computations. We capture the stateful nature of this computation in the following complementary diagram:



We will make one key assumption: the sender and receiver features should each have the structure of a *monoid*—we have included an introduction to the theory of monoids in Appendix A. This implies that we can think of both arguments and messages as being sent in chunks, which are assembled in order. We can think of these monoids as *instruction queues*, where the monoid operation corresponds to instruction composition. Our main definitions will not assume commutativity, though it holds in many examples—we may reorder instructions arbitrarily if so.

3 Node-centric view on algorithmic alignment

For this section, we focus our attention on a single node, and explore the relationship between the message monoid, which we write as $(M, \cdot, 1)$ using multiplicative notation, and the argument monoid, which we write as $(A, +, 0)$ using additive notation.

Suppose that the internal state takes values in a set S. The process by which a received message updates the state and produces an argument is described by a function $M \times S \to S \times A$. This is equivalent, by currying, to a Kleisli arrow $M \to [S, S \times A] = \mathtt{state}_S(A)$ for the state monad.

Given a pair (m, s), we denote the image under this function by $(m \bullet s, \delta_m(s))$, where $\bullet : M \times S \to S$ is written as a binary operation and $\delta : M \times S \to A$ is described by some argument function δ. First, we look into the properties of $\bullet$.

Each incoming message (an element of M) transforms the state (an element of S) in some way. We assume that the multiplication of M corresponds to a composition of these transformations.

Specifically, we ask that $\bullet$ satisfies the unit and associativity axioms:

$$\begin{aligned} 1 \bullet s &= s \\ (n \cdot m) \bullet s &= n \bullet (m \bullet s) \end{aligned} \tag{2}$$

Next, we interpret Equation 2 in terms of the argument function δ. In the first equation, the action $1 \bullet s$ generates an argument $\delta_1(s)$. But on the right-hand side there is no action, so no argument is produced. In order to process both sides consistently, $\delta_1(s)$ must be the zero argument.

Similarly, in the second equation, the left-hand side produces one argument $\delta_{n \cdot m}(s)$, while the right-hand side produces two, $\delta_m(s)$ and then $\delta_n(m \bullet s)$. Setting these equal, we have the following:

$$\begin{aligned} \delta_1(s) &= 0 \\ \delta_{n \cdot m}(s) &= \delta_n(m \bullet s) + \delta_m(s) \end{aligned} \tag{3}$$

Equivalently, we could arrive at these equations by extending the action of M on S to one of M on $S \times A$, as follows. Given a pair (s, a) of a state s together with an outgoing argument a, we act on the state, while generating a new argument that gets added to the old one: $m \star (s, a) := (m \bullet s, \delta_m(s) + a)$. One can show that Equation 3 is exactly the unit and associativity axioms for the operator $\star$.

Cocycles. Equation 3 can be rewritten in a more elegant form, but this requires a few preliminaries.

First, consider the set $F = [S, A]$ of *readout functions*; functions mapping states to corresponding arguments. F inherits structure from both S and A. First, F is a monoid because A is; we define a zero function $0(s) := 0$ and function addition $(f + g)(s) := f(s) + g(s)$.

Second, F has an action of M, given by $(f \bullet m)(s) := f(m \bullet s)$. This is a *right* action rather than a left action; the associativity axiom $f \bullet (m \cdot n) = (f \bullet m) \bullet n$ holds, but the reversed axiom may not.

With these definitions, if we write δ in its curried form $D : M \to [S, A]$, we can rewrite Equation 3:

$$\begin{aligned} D(1) &= 0 \\ D(n \cdot m) &= D(n) \bullet m + D(m) \end{aligned} \tag{4}$$

Equation 4 specifies that D is a *1-cocycle*, otherwise known as a *derivation*. To understand the latter term, consider the more general situation where F also has a left action of M. Then we may write $D(n \cdot m) = D(n) \bullet m + n \bullet D(m)$, which is just the Leibniz rule for the derivative. Our equation describes the special case where this left action is trivial.

We summarise the above chain of deductions as follows:

Proposition 3.1. *A node equipped with a rule D for generating arguments is invariant to asynchrony, i.e. its output does not depend on the grouping of incoming messages, if and only if D is a 1-cocycle.*

Edges: homomorphisms and multimorphisms. 1-cocycles also appear in the literature under the name *crossed homomorphisms*. Indeed, if the right action of M on $[S, A]$ is trivial, the above equations are $D(1) = 0$ and $D(n \cdot m) = D(n) + D(m)$, i.e. D is a homomorphism of monoids.

We can use this observation to describe edges—at least, edges with only one input—as stateless nodes. We simply reverse the roles of argument and message monoids: for asynchronous communication over edges, we require that the message function ψ satisfies $\psi(0) = 1$ (null arguments generate null messages) and $\psi(a + b) = \psi(a) \cdot \psi(b)$ (aggregating before ψ gives same results as aggregating after). In other words:

Proposition 3.2. *An edge with a single-input message function ψ supports asynchronous invocation if and only if ψ is a homomorphism of monoids.*

How to extend to edges that take k inputs, for example, edge features or receiver features? Dudzik and Veličković [17] proposed repeated multiplication in a semiring, but for asynchrony, we only require the much weaker requirement that ψ be a *multimorphism*, i.e. given any fixed values for $k - 1$ of the variables, ψ is a homomorphism in the remaining variables.[2]

With this in mind, if $M_1, \cdots, M_k$ are commutative monoids, we simply treat a stateful node with multiple message inputs $M_1, \cdots, M_k$ as a stateful node with a single message input given by the tensor product of commutative monoids $M_1 \otimes \cdots, \otimes M_k$[3]. In the stateless case, this is equivalent, by

[2] This is the analogue of multilinearity for maps of vector spaces.

[3] Commutativity is needed here, otherwise the tensor product may not exist.

the universal property of the tensor product, to the above condition that ψ is a multimorphism, so this is a convenient mathematical way to describe asynchrony with respect to multiple arguments.

Example: The natural numbers and addition with carry. If $M = (\mathbb{N}, +, 0)$ is the monoid of natural numbers under addition,[4] then an action of M on a set S is equivalently an endofunction $\pi : S \to S$, with $m \cdot s := \pi^m(s)$. We now characterise all possible cocycles $M \to [S, A]$.

Proposition 3.3. *Given any function $\omega : S \to A$, there is a unique 1-cocycle δ with $\delta_1 = \omega$.*

Proof. If δ is a cocycle and $\omega = \delta_1$, the cocycle condition gives us an inductive definition of δ: $\delta_{n+1}(s) = \delta_n(s) + \delta_1(s + n)$. On the other hand, given ω we can define: $\delta_n(s) := \sum_{i=0}^{n-1} \omega(s + i)$.

We note that $\delta_{m+n} = \sum_{i=0}^{m-1} \omega(s+i) + \sum_{j=m}^{m+n-1} \omega(s+j) = \sum_{i=0}^{m-1} \omega(s+i) + \sum_{j=0}^{n-1} \omega(m+s+j) = \delta_m(s) + \delta_n(m + s)$ so δ is a 1-cocycle. □

Now, we consider the example of digit arithmetic with carry. Suppose that $S = \{0, \cdots, 9\}$ is the set of digits in base 10, and our action is given by the permutation $\pi(s) = \overline{s + 1}$, where the bar denotes reduction modulo 10. If $M = (\mathbb{N}, +)$, we can define an argument function based on the number of carries produced when 1 is added to s, m times: $\delta_m(s) := \left\lfloor \frac{m+s}{10} \right\rfloor$.

Proposition 3.4. *δ is a 1-cocycle $M \to [S, A]$.*

Proof. We directly verify that, for all $m, n \in \mathbb{N}$, $s \in \{0, \ldots, 9\}$: $\left\lfloor \frac{m+n+s}{10} \right\rfloor = \left\lfloor \frac{m+\overline{n+s}}{10} \right\rfloor + \left\lfloor \frac{n+s}{10} \right\rfloor$. This follows quickly by induction on m: for $m = 0$ the two sides are equal, and as m increments by 1, the LHS and RHS are both incremented if and only if $m + n + s + 1 \equiv 0 \pmod{10}$. □

4 (A)synchrony and idempotence in GNNs

Having discussed the general mathematical situation, we can now leverage the cocycle conditions to more rigorously discuss the synchronisation of GNN operations (such as gathers, scatters, and applications of ψ or ϕ). In the process of our discussion, we will show how our theory elegantly re-derives and extends a well-known neural algorithmic reasoning model.

In the study of algorithms, we are especially interested in a certain property of monoids: *idempotence*. We start by proving a highly useful fact about the relationship between idempotence and cocycles.

4.1 Idempotent monoids

We say that A is *idempotent* if $a + a = a$ for all $a \in A$.

Proposition 4.1. *Suppose that $S = A$. Define:*

$$\delta_m(s) := \begin{cases} 0 & \text{if } m = 1 \\ m \bullet s & \text{otherwise} \end{cases}$$

If δ is a 1-cocycle, then A is idempotent. If $M = S = A$ and $\cdot = +$, the converse holds.

Proof. In this case, the second equation of Equation 3 becomes $(n \cdot m) \bullet s = n \bullet (m \bullet s) + m \cdot s$. Setting $n = m = 1$ gives $s = s + s$.

If $\cdot = +$ and A is idempotent, then the equation is $n + m + s = (n + m + s) + (m + s)$, which holds since $(n + m + s) + (m + s) = n + ((m + s) + (m + s)) = n + m + s$. □

4.2 Making ϕ a cocycle enables asynchrony

Now we can explicitly formalise the residual map ϕ in Equation 1: it is just another description of what we have called an "action". That is, we have $\phi(s, m) = m \bullet s$ for all node features s and (aggregated) non-null messages m.

[4] We are using additive notation here, so the identity element is denoted with 0 instead of 1.

Message aggregation asynchrony. $\bigoplus$ is usually taken, axiomatically, to be the operation of a commutative monoid [30]. This is to emphasise the importance of message aggregation that does not depend on the order in which the messages are received. That is, letting $\bigoplus$ define a commutative monoid operation already allows us to support a certain form of asynchrony: we can aggregate messages online as we receive them, rather than waiting for all of them before triggering $\bigoplus$.

Node update asynchrony. Similarly, the axiom that ϕ defines an associative operation, as in Equation 2, corresponds to another type of asynchrony. When ϕ satisfies the associativity equation:

$$\phi(s, m \bigoplus n) = \phi(\phi(s, m), n) \tag{5}$$

this tells us that ϕ itself can be applied asynchronously. Put differently, after each message arrives into the receiver node, we can use it to update the node features by triggering ϕ, without waiting for the messages to be fully aggregated.

One common way to enforce associativity is to take $\phi = \bigoplus$, in which case the associativity of ϕ follows from the assumed associativity of $\bigoplus$.

Argument generation asynchrony. While these two conditions allow us to reason about the asynchrony in how incoming messages are processed, and the asynchrony in how the node's features are updated, what does the cocycle condition (Equation 3) mean for a GNN?

Recall, the cocycle condition concerns the argument function δ, which determines which arguments are produced after a node update. Specifically, the cocycle condition allows us to express the arguments produced by receiving two messages together ($\delta_{n \cdot m}$) as a combination of the arguments produced by receiving them in isolation (δ_n and δ_m). Therefore, it gives us a mechanism that allows for each sender node to prepare their arguments to the message function, ψ, asynchronously, rather than waiting for all the relevant node updates to complete first.

Note that Equation 1 does not distinguish between node features and sent arguments. In other words, it implicitly defines the argument function $\delta_m(s) = m \bullet s = \phi(s, m)$. Knowing this, we can leverage the converse direction of Proposition 4.1, to provide some conditions under which the update function ϕ will satisfy the cocycle condition:

First, we require the state update (which we previously set to $\phi = \bigoplus$) to be idempotent. Not all commutative monoids are idempotent; $\bigoplus = \max$ is idempotent, while aggregators like sum are not. Note that this aligns with the known utility of the $\max$ aggregation in algorithmic tasks [14, 17, 31].

Second, we require $M = S = A$, which means that the messages sent must come from the same set of values as the node features and the arguments to the message function. This means that the dimensionality of the node features, arguments and messages should be the same—or alternately, that invoking the message or update functions should not change this dimensionality. This can be related to the encode-process-decode paradigm [32]: advocating for the use of a *processor module*, repeatedly applied to its inputs for a certain number of steps.

Note that this is only one possible way to enforce the cocycle condition in ϕ; we remark that there might be more approaches to achieving this, including approximating the cocycle condition by optimising relevant loss functions.

4.3 The rich asynchrony of $\max$-$\max$ GNNs: Rediscovering PathGNNs

It is worthwhile to take a brief pause and collect all of the constraints we have gathered so far: (1) $\bigoplus$ is a commutative monoid; (2) $\phi = \bigoplus$; (3) $\bigoplus$ must be idempotent, e.g., $\bigoplus = \max$; (4) The message function should output messages of the same dimensionality as node features.

We can observe that Equation 1 now looks as follows:

$$\mathbf{x}'_u = \max\left(\mathbf{x}_u, \max_{v \in \mathcal{N}_u} \psi(\mathbf{x}_u, \mathbf{x}_v)\right) \tag{6}$$

Such a $\max$-$\max$ GNN variant allows for a high level of *asynchrony*: messages can be sent, received and processed in an arbitrary order, arguments can be prepared in an online fashion, and it is mathematically guaranteed that the outcome will be identical as if we fully synchronise all of these steps, as is the case in typical GNN implementations. We also remark that Equation 6 is almost exactly equal to the hard variant of the PathGNN model from Tang et al. [20]—the only missing aspect is to remove the dependence of ψ on the receiver node (i.e., to remove u from the `senders`).

The only remaining point of synchronisation is the invocation of the message function, ψ; messages can only be generated once all of the arguments for the message function are fully prepared (i.e., no invocations of ϕ are left to perform for the relevant sender nodes).

4.4 Extending PathGNNs with multimorphisms: Message generation asynchrony

PathGNNs give one example of *isotropic* message passing, where the message function for each edge, ψ_e, is a function of a single variable, the sender argument, and produces a single output, a receiver message. That is, we have a function $\psi_e : (A_{s(e)}, +, 0) \to (M_{t(e)}, \cdot, 1)$.

As we elaborated in Section 3, the condition needed for argument asynchrony and message asynchrony to be compatible is that ψ is a monoid homomorphism. This means that ψ can be called—and messages generated—even before its arguments are fully ready, so long as it is called again each time the arguments are updated.

Note that PathGNNs, in their default formulation, do not always satisfy this constraint. We have, therefore, used our analysis to find a way to extend PathGNNs to a *fully asynchronous* model. One way to obtain such a GNN—assuming it is already isotropic—is to make ψ be a *tropical linear* transformation. That is, ψ would be parametrised by a $k \times k$ matrix, which is multiplied with $\mathbf{x}_v$, but replacing "$+$" with $\max$ and "$\times$" with $+$.

In the non-isotropic case [33], where ψ may take multiple arguments, different properties of ψ may correspond to different forms of asynchrony. Since various DP algorithms can be parallelised in many different ways, we consider this case a promising avenue for future exploration. In Section 5 we give empirical results for multimorphisms as asynchronous message functions in anisotropic GNNs.

4.5 Example: Semilattices and Bellman-Ford

Originally, PathGNN was designed to align with the Bellman-Ford algorithm [34], due to its claimed structural similarity to the algorithm's operation—though this claim was not rigorously established. Now, we can rigorously conclude where this alignment comes from: the choice of aggregator ($\max$) and making ψ only dependent on one sender node is fully aligned with the Bellman-Ford algorithm (as in Xu et al. [31]), *and* both PathGNNs and Bellman-Ford can be implemented asynchronously without any errors in the final output (a novel conclusion enabled by our mathematical framework). We have already showed that PathGNNs satisfy the cocycle condition; now we will prove the same statement about Bellman-Ford, in order to complete our argument.

Let P be any join-semilattice, i.e. a poset with all finite joins, including the empty join, which we denote 0 as it is a lower bound for P. A standard result says that P is equivalently a commutative, idempotent monoid $(P, \vee, 0)$. When P is totally ordered, we usually write $\vee = \sup$ or $\max$.

In Bellman-Ford, P will generally be either the set of all path lengths, (taken with a negative sign) or the set of all relevant paths ending at the current vertex, equipped with a total order that disambiguates between paths of equal length.

We set $M = S = A = P$, with action of M on S given by $m \cdot s := m \vee s$.

Since we want to inform our neighbors when our state improves, i.e. gets smaller, we could likewise define our argument function δ by $\delta_m(s) := m \vee s$, as this in fact satisfies the cocycle equation due to Proposition 4.1. However, this will lead to an algorithm that never terminates, as new redundant arguments will continue to be generated at each step.

So we define the argument function a bit more delicately:

$$\delta_m(s) := \begin{cases} 0 & \text{if } m \leq s \\ m \vee s & \text{otherwise} \end{cases}$$

Proposition 4.2. *δ is a 1-cocycle $M \to [S, A]$.*

Proof. Pick $m, n \in M$, $s \in S$. We wish to show that $\delta_{m \vee n}(s) = \delta_m(n \vee s) \vee \delta_n(s)$.

If $m, n \leq s$ then both sides equal 0. If $m \leq s$ but $n \not\leq s$ then the LHS is $m \vee n \vee s = n \vee s$ while the RHS is $0 \vee (n \vee s)$. If $n \leq s$ but $m \not\leq s$ then the LHS is $m \vee n \vee s = m \vee s$ while the RHS is $(m \vee n \vee s) \vee 0 = m \vee s$. □

Now, we can see that the definition of δ prevents new arguments from being generated once the optimal value has been obtained, if we take the convention that zero arguments are ignored. In particular, if P is well-ordered, only finitely many arguments can be generated regardless of the input.

Lastly, note that Bellman-Ford is also a perfect example of message generation asynchrony; its ψ_e, for each edge, simply adds the edge length of e to the sender node distance. $+$ is easily seen to distribute over $\max$, making ψ_e a monoid homomorphism. All conditions combined, Bellman-Ford can be indeed made fully asynchronous, without sacrificing the fidelity of the final output. This solidifies the algorithmic alignment of our PathGNN variant with Bellman-Ford.

5 Evaluating asynchrony-invariant GNNs

In our paper, we introduced three[5] distinct *levels* of asynchrony-invariant GNNs:

- **L1** A GNN with a *commutative monoid* aggregator [30], $\bigoplus$, is invariant to message receiving order;
- **L2** A GNN which, additionally, has an associative and idempotent update function [20], ϕ, is invariant to repeated calls to the update function.
- **L3** A GNN which, additionally, has a message function, ψ, which is a *monoid multimorphism*, is invariant to repeated calls to the message function.

To supplement our theory, and illustrate how progressing up these three levels practically results in a more robust algorithmic executor, we perform comparative experiments on the CLRS-30 benchmark [35]. Specifically, we closely follow the single-task CLRS-30 experimental setup of Ibarz et al. [29], incorporating inverted pointer features as discovered by Bevilacqua et al. [36]; we defer to these papers for concrete details on the train/test data pipeline incorporated.

In order to avoid confounding factors, we further modify the CLRS-30 baselines to focus on linear message/update functions, and do not deploy triplet messages [17]. Within this framework, we focus on the following three hyperparameter settings, corresponding to the three levels:

- **L1** Let $\bigoplus = \sum$, ψ a linear layer, and ϕ a linear layer with ReLU activation.
- **L2** Let $\bigoplus = \phi = \max$, and ψ a linear layer.
- **L3** Let $\bigoplus = \phi = \max$, and ψ is a $\log$-semiring bilinear layer, potentially preceded by an additional linear layer (we treat this architectural choice as a per-task hyperparameter).

A $\log$-semiring bilinear layer is a matrix multiplication of the form $\psi(\mathbf{x}, \mathbf{y}) = \mathbf{A}_{\log}\mathbf{x} \times \mathbf{B}_{\log}\mathbf{y}$ where we "re-interpret" "$+$" to be the *logsumexp* operator (i.e. $x +_{lse} y := \log(\exp(x) + \exp(y))$), and "$\times$" to be *addition* (i.e. $x \times_{lse} y := x + y$). We implemented this bilinear layer using SynJax [37].

We use the $\log$-semiring as a smooth approximation of the previously described tropical linear (where "$+$" would be $\max$, and "$\times$" would be $+$). This is a choice made due to the large sparsity in gradients for a tropical linear layer, which makes learning more challenging in practice—a phenomenon concurrently empirically observed for aggregation functions by Mirjanić et al. [38].

The comparative results across the algorithmic execution tasks in CLRS-30, in the out-of-distribution regime, are provided in Figure 2. The overall ranking of the methods in terms of their "performance profile" across the thirty tasks closely matches the three levels, with the overall best results achieved by the $\log$-semiring message function, and the second-best results achieved by $\max$-$\max$ GNNs [20]. It is very useful to comment on two "opposite" classes of algorithms in light of these results.

We first discuss the CLRS-30 algorithms that are "embarrassingly parallel", in the sense that each node can continuously broadcast its state until convergence is reached, and meaningful updates may happen anywhere in the graph. BFS [39], Bellman-Ford [34] and Floyd-Warshall [40] are all standard examples. For all of the above algorithms, GNNs satisfying the cocycle condition (levels 2 and 3) generalise substantially more favourably than the basic, level-1, expressive GNN.

Conversely, we have algorithms that are "purely sequential", in that at every step, exactly one node is broadcasting meaningful state, and exactly one of its neighbours may get a meaningful update. In

[5] Strictly speaking, our analysis predicts *four* levels, but as in typical GNNs (of Equation 1) we do not assign special semantics to the argument-generating function δ, so we merge the second (ϕ associative) and third (ϕ idempotent) level for the purpose of this analysis, leaving exploration of the extra expressivity of δ to later work.

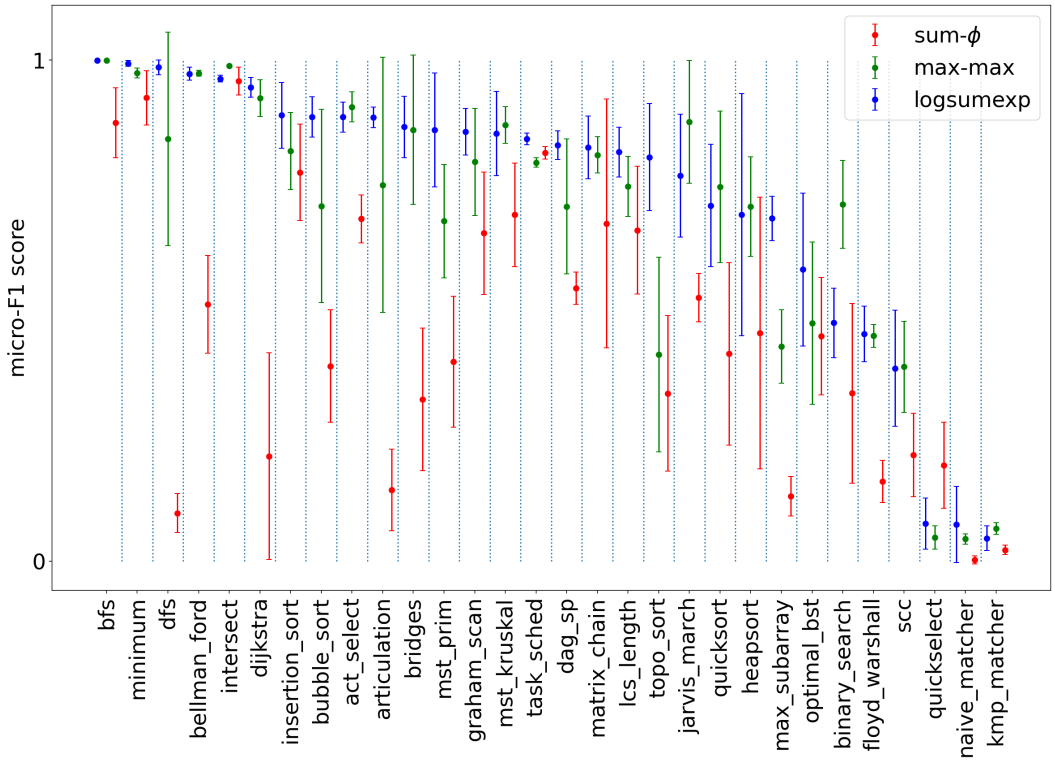


Figure 2: Test (out-of-distribution) results across all tasks in CLRS-30, for the three models described in Section 5, averaged across six seeds.

this setting, nearly all messages are redundant! Therefore, having a GNN that is invariant to partial invocations of the message functions—as is the case with the level-3 $\log$-semiring architecture—may be particularly relevant, as it will reduce the variance caused by long trajectories of redundant messages. And indeed, in many representative algorithms of this class, such as DFS [39], articulation points, bridges and topological sorting [41], the level-3 architecture significantly reduces variance, or further improves generalisation, compared to the level-2 architecture.

We find these results, taken all together, to offer significant evidence to the utility of our theory, and we hope they can serve as an inspiration for follow-up studies and variations! In general, the level of asynchrony required by a task may not always be easily anticipated, and we foresee future work that constructs better-aligned architectures through the lens of various *schedulers*.

6 Conclusions

In this work we have taken a *node-centric* perspective on the computation of the message passing mechanism. We note that this allows us to reason about the asynchrony of the updates on the node *persistent* state, where the cocycle conditions are necessary in order to support asynchronous updates. We show that these conditions are respected for the Bellman-Ford algorithm as well as an extension of the recently-proposed PathGNN [20], and can be used as a principle to discover other GNN formulations that allow complete asynchronous execution.

Through our analysis we provide a complementary perspective on formally describing computations carried by both algorithms and graph neural networks, which we believe to be a step forward towards further formalising the concept of algorithmic alignment that is widely relied on by approaches for neural algorithmic reasoning. Note that our approach does not compel actual GNNs to sacrifice their scalability by being asynchronous—rather, it imposes mathematical constraints on the GNNs' building blocks, such that, *if we were to execute it in a particular asynchronous regime, it would behave identically to the target algorithm*. Such a correspondence, naturally, improves the level of algorithmic alignment enjoyed by the GNN—as is evident from our presented empirical results.

References

[1] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017. 1

[2] Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, and Koray Kavukcuoglu. Interaction networks for learning about objects, relations and physics. *NeurIPS*, abs/1612.00222, 2016. 1

[3] Petar Veličković. Everything is connected: Graph neural networks. *Current Opinion in Structural Biology*, 79:102538, 2023. 1

[4] Hanchen Wang, Tianfan Fu, Yuanqi Du, Wenhao Gao, Kexin Huang, Ziming Liu, Payal Chandak, Shengchao Liu, Peter Van Katwyk, Andreea Deac, et al. Scientific discovery in the age of artificial intelligence. *Nature*, 620(7972):47–60, 2023. 1

[5] Jonathan M Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M Donghia, Craig R MacNair, Shawn French, Lindsey A Carfrae, Zohar Bloom-Ackermann, et al. A deep learning approach to antibiotic discovery. *Cell*, 180(4):688–702, 2020. 1

[6] Gary Liu, Denise B. Catacutan, Khushi Rathod, Kyle Swanson, Wengong Jin, Jody C. Mohammed, Anush Chiappino-Pepe, Saad A. Syed, Meghan Fragis, Kenneth Rachwalski, Jakob Magolan, Michael G. Surette, Brian K. Coombes, Tommi Jaakkola, Regina Barzilay, James J. Collins, and Jonathan M. Stokes. Deep learning-guided discovery of an antibiotic targeting acinetobacter baumannii. *Nature Chemical Biology*, May 2023. ISSN 1552-4469. doi: 10.1038/s41589-023-01349-8. URL https://doi.org/10.1038/s41589-023-01349-8. 1

[7] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021. 1

[8] Austin Derrow-Pinion, Jennifer She, David Wong, Oliver Lange, Todd Hester, Luis Perez, Marc Nunkesser, Seongjae Lee, Xueying Guo, Brett Wiltshire, et al. Eta prediction with graph neural networks in google maps. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 3767–3776, 2021. 1

[9] Gage DeZoort, Peter W Battaglia, Catherine Biscarat, and Jean-Roch Vlimant. Graph neural networks at the large hadron collider. *Nature Reviews Physics*, pages 1–23, 2023. 1

[10] Alex Davies, Petar Veličković, Lars Buesing, Sam Blackwell, Daniel Zheng, Nenad Tomašev, Richard Tanburn, Peter Battaglia, Charles Blundell, András Juhász, et al. Advancing mathematics by guiding human intuition with AI. *Nature*, 600(7887):70–74, 2021. 1

[11] Charles Blundell, Lars Buesing, Alex Davies, Petar Veličković, and Geordie Williamson. Towards combinatorial invariance for kazhdan-lusztig polynomials. *arXiv preprint arXiv:2111.15161*, 2021.

[12] Geordie Williamson. Is deep learning a useful tool for the pure mathematician? *arXiv preprint arXiv:2304.12602*, 2023. 1

[13] Petar Veličković and Charles Blundell. Neural algorithmic reasoning. *Patterns*, 2(7):100273, 2021. 1

[14] Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. *arXiv preprint arXiv:1910.10593*, 2019. 1, 6

[15] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *arXiv preprint arXiv:2206.14858*, 2022. 1

[16] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? *arXiv preprint arXiv:1905.13211*, 2019. 1

[17] Andrew Dudzik and Petar Veličković. Graph neural networks are dynamic programmers, 2022. 1, 2, 4, 6, 8

[18] Lukas Faber and Roger Wattenhofer. Asynchronous message passing: A new framework for learning in graphs. 2022. 2

[19] Ben Finkelshtein, Xingyue Huang, Michael Bronstein, and İsmail İlkan Ceylan. Cooperative graph neural networks. *arXiv preprint arXiv:2310.01267*, 2023. 2

[20] Hao Tang, Zhiao Huang, Jiayuan Gu, Bao-Liang Lu, and Hao Su. Towards scale-invariant graph-related problem solving by iterative homogeneous gnns. *Advances in Neural Information Processing Systems*, 33:15811–15822, 2020. 1, 6, 8, 9

[21] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges, 2021. 2, 12

[22] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018. 2

[23] Cristian Bodnar, Francesco Di Giovanni, Benjamin Paul Chamberlain, Pietro Liò, and Michael M. Bronstein. Neural sheaf diffusion: A topological perspective on heterophily and oversmoothing in gnns, 2023. 2

[24] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014. 3

[25] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538 (7626):471–476, 2016. 3

[26] Petar Veličković, Lars Buesing, Matthew Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. *Advances in Neural Information Processing Systems*, 33: 2232–2244, 2020. 3

[27] Heiko Strathmann, Mohammadamin Barekatain, Charles Blundell, and Petar Veličković. Persistent message passing. *arXiv preprint arXiv:2103.01043*, 2021. 3

[28] Louis-Pascal Xhonneux, Andreea-Ioana Deac, Petar Veličković, and Jian Tang. How to transfer algorithmic reasoning knowledge to learn new algorithms? *Advances in Neural Information Processing Systems*, 34:19500–19512, 2021. 3

[29] Borja Ibarz, Vitaly Kurin, George Papamakarios, Kyriacos Nikiforou, Mehdi Bennani, Róbert Csordás, Andrew Joseph Dudzik, Matko Bošnjak, Alex Vitvitskyi, Yulia Rubanova, et al. A generalist neural algorithmic learner. In *Learning on Graphs Conference*, pages 2–1. PMLR, 2022. 3, 8

[30] Euan Ong and Petar Veličković. Learnable commutative monoids for graph neural networks. *arXiv preprint arXiv:2212.08541*, 2022. 6, 8

[31] Keyulu Xu, Mozhi Zhang, Jingling Li, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. *arXiv preprint arXiv:2009.11848*, 2020. 6, 7

[32] Jessica B Hamrick, Kelsey R Allen, Victor Bapst, Tina Zhu, Kevin R McKee, Joshua B Tenenbaum, and Peter W Battaglia. Relational inductive bias for physical construction in humans and machines. *arXiv preprint arXiv:1806.01203*, 2018. 6

[33] Xavier Bresson and Thomas Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017. 7

[34] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958. 7, 8

[35] Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. The clrs algorithmic reasoning benchmark. In *International Conference on Machine Learning*, pages 22084–22102. PMLR, 2022. 8

[36] Beatrice Bevilacqua, Kyriacos Nikiforou, Borja Ibarz, Ioana Bica, Michela Paganini, Charles Blundell, Jovana Mitrovic, and Petar Veličković. Neural algorithmic reasoning with causal regularisation. *arXiv preprint arXiv:2302.10258*, 2023. 8

[37] Miloš Stanojević and Laurent Sartran. Synjax: Structured probability distributions for jax. *arXiv preprint arXiv:2308.03291*, 2023. 8

[38] Vladimir V Mirjanić, Razvan Pascanu, and Petar Veličković. Latent space representations of neural algorithmic reasoners. *arXiv preprint arXiv:2307.08874*, 2023. 8

[39] Edward F Moore. The shortest path through a maze. In *Proc. of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959. 8, 9

[40] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962. 8

[41] Donald Ervin Knuth. Fundamental algorithms. *The art of computer programming*, 1:51–78, 1973. 9

[42] Bartosz Milewski. *Category theory for programmers*. Blurb, 2018. 12

[43] Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International conference on machine learning*, pages 2990–2999. PMLR, 2016. 17

A Introduction to monoids

The central arguments of our paper rest on the construct of a *monoid*. As we will unpack throughout this Appendix, monoids are an excellent abstraction for studying repeated computation in algorithms. We start with a preliminary overview of monoids, guided by the example of processing data in lists.

Consider a few common functions $f : \texttt{list}(A) \to B$ whose input is a list type:

1. $f(L) = \#L$ (the length function)
2. $f(L) = \begin{cases} 1 & \text{if "hello"} \in L \\ 0 & \text{otherwise} \end{cases}$, where $A = \texttt{String}$ (the *any* function)
3. $f(L) = \prod_{a \in L} a$, where $A = \mathbb{R}$ (the product function on reals)

In these cases (and many others), we are performing a *fold*, that is, a repeated application of a binary operation over the list's elements. (possibly after a map) Specifically, for the above functions, this operation is:

1. The operation $+$ on the set $\mathbb{N}$ of natural numbers.
2. The operation $\texttt{or}$ on the set $\{0, 1\}$ of Boolean values.
3. The operation $\times$ on the set $\mathbb{R}$ of real numbers.

The fact that these functions arise from such an operation has deep consequences for the parallelisation of the computation, as well as its semantics. Further, the binary operations considered above all have properties of interest. In all three cases, we may arbitrarily split the list into sublists, perform the computation in parallel on each sublist, and then compute the aggregate result—and this will not affect the final result of the fold operation.

With this is mind, we define a *monoid* to be a triple $(M, \cdot, 1)$ consisting of a set M, a binary operation $\cdot : M \times M \to M$, and a "neutral" element $1 \in M$, satisfying the following axioms for all $a, b, c, x \in M$:

$$\begin{aligned} 1 \cdot x = x \cdot 1 = x \\ a \cdot (b \cdot c) = (a \cdot b) \cdot c \end{aligned} \tag{7}$$

The reader may have encountered these axioms when studying *groups*, particularly in the context of geometric deep learning [21]. Groups have an additional axiom of *invertibility*; therein, every element $x \in M$ must have an inverse element $x^{-1} \in M$ such that $x \cdot x^{-1} = x^{-1} \cdot x = 1$. That being said, we note that none of the three binary operations considered above are invertible, hence none of these structures are groups. Since programming is usually focused on non-invertible operations, groups arise quite rarely in algorithmic computation, while monoids are extremely common.

We also note a commonality with *category theory*: a monoid is exactly the same thing as a category with only one object. We will not explore this perspective further here, but the reader can consult Milewski [42] for more information about the categorical approach to monoids.

When defining a mathematical structure, we should also say how to define *arrows*, or structure-preserving maps, between them. A *monoid homomorphism* $f : (M, \cdot_M, 1_M) \to (M', \cdot_{M'}, 1_{M'})$ is

defined to be a function $f : M \to M'$ satisfying the following axioms.

$$\begin{aligned} f(1_M) &= 1_{M'} \\ f(a \cdot_M b) &= f(a) \cdot_{M'} f(b) \end{aligned} \tag{8}$$

Such a function preserves the structure contained in M when mapping its elements into M'. If a homomorphism is bijective then its inverse is automatically a homomorphism; we call such a homomorphism an *isomorphism*.

Just as with groups, we are typically not only interested in monoids themselves, but how they *act* on data. A set S is said to be equipped with a *left M-action* when we have a binary operation $\bullet : M \times S \to S$ such that the following axioms hold for all $m, n \in M$ and $s \in S$:

$$\begin{aligned} 1 \bullet s &= s \\ n \bullet (m \bullet s) &= (n \cdot m) \bullet s \end{aligned} \tag{9}$$

These axioms are equivalent to saying that we have a map $\rho : M \to [S, S]$ from M to endofunctions on S, satisfying $\rho(1) = \mathrm{id}_S$ and $\rho(n \cdot m) = \rho(n) \circ \rho(m)$, where id_S is the *identity function* on S[6], and $\circ$ is function composition. The map ρ is sometimes also referred to as a *representation* of the monoid.

We will also make use of the notion of *right M-action*, which is the same as the above but with all orders of operation reversed. Whenever not specified, actions are assumed to be on the left.

All monoids act on themselves. Specifically, if we define $m \bullet s := m \cdot s$ for $m, s \in M$, we can see by comparing Equations 7 and 9 that $\bullet$ satisfies the axioms of a left action. This is called the *left regular representation* of M. Analogously, we can define a *right regular representation*. This gives a large class of examples of monoid actions.

A.1 Monoid actions as state machines

Now, we justify why monoids are excellent for representing repeated computation, by relating monoid actions to the foundational computer science concept of *state machines*[7].

Recall that we use the notation $[A, B]$ to denote the set of all possible functions mapping A to B.

Define a *state machine* to be a triple (S, Σ, τ), consisting of a set S of *states*, a set Σ of *symbols*, and a *transition function* $\tau : \Sigma \to [S, S]$ that interprets each symbol as a transition of states. In many cases, state machines also have distinguished starting and accepting states, but these are not important for the present discussion.

Given a monoid M and a set S equipped with an M-action ρ, we can quickly conclude that (S, M, ρ) fits the definition of a state machine. However, it turns out that we can go in the reverse direction as well: every state machine can be seen as specifying an action of a monoid.

To see this, suppose that (S, Σ, τ) is a state machine, and let Σ^* denote the collection of words drawn from Σ. That is, Σ^* comprises strings of the form $\sigma = \sigma_1 \cdots \sigma_n$ (where $\sigma_i \in \Sigma$), including the empty string, $\emptyset$. Note that, for each such word $\sigma \in \Sigma^*$, we have an associated composed transformation $\tau_\sigma := \tau(\sigma_1) \circ \cdots \circ \tau(\sigma_n)$. This function $\tau_\sigma \in [S, S]$ describes the final state reached after consuming the sequence of symbols in σ, for a given initial state.

Then, the set of all composed transitions, $M = \{\tau_\sigma \mid \sigma \in \Sigma^*\}$ forms a monoid $(M, \circ, \mathrm{id}_S)$ under the function composition operation $\circ$, with the identity transformation as the neutral element. Further, S can be equipped with an M-action, defined by $\tau_\sigma \bullet s := \tau_\sigma(s)$, i.e. $\rho = \tau$.

Note that some information contained within the state machine is lost when converting it to a monoid in this way. To recover the original state machine, we must specify the function $\Sigma \to M$ assigning each symbol to its transformation. However, since this amounts to labeling a particular subset of transitions[8], it does not affect the structure of the computations performed by the state machine.

[6] Defined as $\mathrm{id}_S(s) = s$ for all $s \in S$.

[7] Unlike the more standard computer science construct, the state machines we consider here may be infinite. Constraining the states and transitions in an appropriate way may allow us to recover standard hierarchies of computability, such as the Chomsky hierarchy.

[8] Specifically, the subset being labelled needs to be a *generating set*, meaning that every element of the monoid can be obtained from some composition of labelled elements.

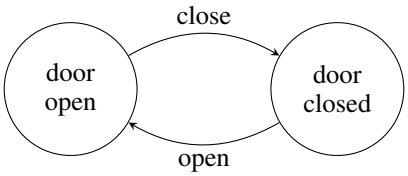
But information is also lost when passing from monoid actions to state machines, since we have neglected the monoid structure of M by discarding its binary operation and neutral element. The only structure that remains is contained in the monoid action, $\bullet$, but there are no guarantees that it will fully disambiguate different elements of the monoid.

To formalise this, we say that a monoid action, $\bullet : M \times S \to S$, is *faithful* if, for all $m, n \in M$, if m and n are different, they must act differently on some state in S using $\bullet$. Or, equivalently,

$$(\forall s \in S.\ m \bullet s = n \bullet s) \iff m = n \tag{10}$$

If a monoid action is not faithful, then we will have at least two *different* monoid elements mapping to exactly the same transitions in the state machine, and hence they will be indistinguishable. In general, state machines correspond to only the faithful actions. As we will see in Example A.2, not all monoid actions are faithful.

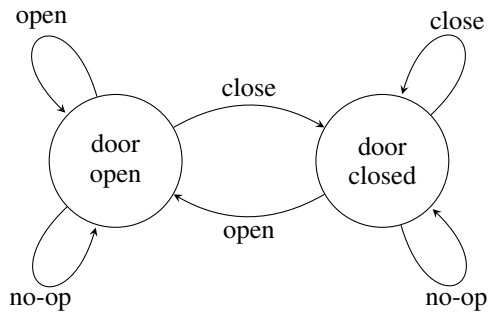
Example A.1. A standard example of a state machine consists of two states, $S = \{\text{door open}, \text{door closed}\}$, with $\Sigma = \{\text{open}, \text{close}\}$ performing the obvious transitions, with the convention that open does nothing to the open door, and close does nothing to the closed door.



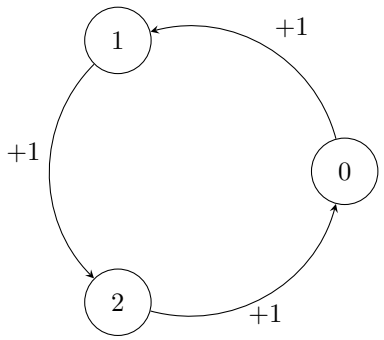
To see how to interpret this as a monoid action, we note the following composition laws:

$$\begin{aligned} \text{open} \circ \text{open} &= \text{open} \\ \text{open} \circ \text{close} &= \text{open} \\ \text{close} \circ \text{open} &= \text{close} \\ \text{close} \circ \text{close} &= \text{close} \end{aligned} \tag{11}$$

Such a structure already exhibits the required associativity properties. In order to obtain a monoid from it, note that we are missing a neutral transformation. One simple way to achieve this is to augment Σ with a single symbol, the identity no-op. So we have $M = \{\text{no-op}, \text{open}, \text{close}\}$. We can visualise the action of M on S by showing how these operations act on both states:

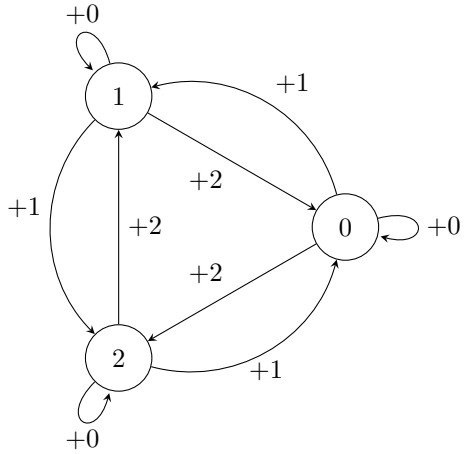


Example A.2. Let W_n be the set $\{0, 1, \ldots, n-1\}$, considered modulo n. We can make W_n into a (modular) counting state machine with $\Sigma = \{+1\}$. Here is a picture when $n = 3$:



Note that this state machine has a cyclic symmetry, so for example we get the same state machine if we relabel $0 \mapsto 1$, $1 \mapsto 2$, $2 \mapsto 0$. For this reason, we do not identify W_n with the set $\mathbb{Z}/n$, as the latter has an unambiguous additive unit element, 0.

As above, in order to get a monoid we need to augment with any remaining transitions associated with sequences of symbols: the no-op $(+0)$ corresponding to an empty string, as well as $+2 := (+1)\circ(+1)$ and more generally $+k := (+1)^{\circ k}$. The corresponding state machine for $n = 3$ now looks like this:



But we note that, because of the modular nature of the counting done by the state machine, we have $+n = +0$. This implies that, from the perspective of the state machine, all transitions $+k$ with $k \geq n$, are already described by the shorter ones. Accordingly, the monoid whose action corresponds to this state machine is the *cyclic group* of order n, denoted by $(\mathbb{Z}/n, +, 0)$.

However, it can be useful to think of W_n as having an action of the monoid spanning the entire set of natural numbers $(\mathbb{N}, +, 0)$[9], rather than just $(\mathbb{Z}/n, +, 0)$. If we define $k \bullet s := (+k)(s)$, we see that the axioms for an action are satisfied, even if $k \geq n$. But since, e.g., $0 \bullet s = n \bullet s$ for all $s \in W_n$, this is no longer a faithful action, and hence it does not arise from a state machine. We will discuss how to repair this defect in the following section.

A.2 Extension problems and cocycles

If $f : M \to M'$ is a morphism of monoids, we define the *kernel*, $\ker f := f^{-1}(1_{M'})$, as the set of all elements of M that map to $1_{M'}$. For general monoids, the kernel is not very informative. But when M and M' are groups, the kernel gives us quite a bit of information about f—for example, f is injective if and only if $\ker f = \{1_M\}$.

A classic question in group theory, and a motivating example for the development of group cohomology, is this: Given two groups G and A, how can we describe all the groups G' with a surjective homomorphism $f : G' \to G$, such that $\ker f$ is isomorphic to A? In other words, we seek to describe all groups G' that can be mapped to G with a function $f : G' \to G$, in such a way that:

- All elements of G are mapped by some element in G', i.e., $\forall x \in G.\exists x' \in G'.f(x') = x$, and
- The structure of all elements of G' that map into the neutral element for G (1_G) is isomorphic to the structure of A.

Such groups G' are called *extensions of G by A*. For example, if $G = \mathbb{Z}/2$ and $A = \mathbb{Z}/3$, then one can easily define a "trivial" extension $\mathbb{Z}/6 \cong \mathbb{Z}/2 \times \mathbb{Z}/3$. This extension is valid because we can map elements of $\mathbb{Z}/6$ to all elements in G by using the function $x \mapsto x \pmod 2$. Then, all elements of $\mathbb{Z}/6$ mapping to 0 would be $\{0, 2, 4\}$, which has exactly the same structure as A.

Note that there also exists a nontrivial extension of $\mathbb{Z}/2$ by $\mathbb{Z}/3$: the symmetric group S_3, whose elements are permutations of three elements. Here the surjection $S_3 \to \mathbb{Z}/2$ is given by taking a permutation to its *sign*.

[9]Note that $(\mathbb{N}, +, 0)$ is not a group, as there are no inverses.

The analogous question of extensions for two monoids is much harder than for groups, due to severe technical challenges in defining cohomology, and we will not attempt to analyse it in detail here. But we will show how certain functions—which will correspond to the notion of "cocycle" we used in the body of our paper—may be used to construct certain extensions of monoids.

Within the state machine W_n given by our previous example, we were unable to distinguish between the instructions $+0$ and $+n$. We can recall from our early education why this is: in the transition from state $n-1$ to state 0, we should produce a *carry*, which would itself be applied to advance a different state machine. If this next machine also has the structure of W_n, this process can be iterated, giving us arithmetic in base n.

Focusing on just the ones digit, we can see that carrying is really describing $\mathbb{N}$ as an extension of $G = \mathbb{Z}/n$, the monoid associated to the state machine, by $A = n\mathbb{N} \cong \mathbb{N}$, the monoid for carries. That is, $n\mathbb{N}$ is the kernel of the natural map $\mathbb{N} \to \mathbb{Z}/n$.

Let's look in more detail at how this works, in terms of the state machine $S = W_n$ acted on by $M = \mathbb{N}$, and producing carries in $A = \mathbb{N}$. We describe carrying as a function $\delta : M \times S \to A$. Given an instruction m acting on a state s, $\delta_m(s)$ describes the carry produced during the transition $s \mapsto m \cdot s$.

For δ to be consistent with the equations for a monoid action, we need two things to be true. First, the identity transition $s \mapsto 1_M \cdot s = s$ must produce an identity carry. And second, given two instructions m and n, we must get the same total carry from the transition $s \mapsto (nm) \cdot s$ and the pair of transitions $s \mapsto m \cdot s \mapsto n \cdot (m \cdot s)$. We can summarise this in the following pair of equations, which we'll call the *carry equations*:

$$\begin{aligned} \delta_1(s) &= 0 \\ \delta_{n \cdot m}(s) &= \delta_n(m \bullet s) + \delta_m(s) \end{aligned} \tag{12}$$

Here we have written $(M, \cdot, 1)$ in multiplicative notation and $(A, +, 0)$ in additive notation, to avoid confusion between the two operations.

In the carry example, satisfying Equation 12 guarantees we can process and emit carries for multiple numbers to be added in any order. For example, if we want to make two actions, $(+2), (+3)$ on a state $s \in W_n$, we could either emit a carry for $s + 2$ and then combine it with another carry emitted for $(s + 2) + 3$, or only emit a single carry for $s + 5$—the final carry is guaranteed to be the same.

Now it is worthwhile to recall that, in the context of graph neural networks, this is exactly the abstraction we used to reason about the interplay between the incoming messages in a node (M) and the emitted node arguments (A) in response to those messages. In the GNN example, should its update function, ϕ, satisfy a condition as in Equation 12, when updating a node's state based on incoming messages $\mathbf{n}$ and $\mathbf{m}$, we can either first aggregate them and emit one update ($\phi(\mathbf{x}, \mathbf{n} \bigoplus \mathbf{m})$), or update based on the first one, then update based on the second one ($\phi(\phi(\mathbf{x}, \mathbf{n}), \mathbf{m})$).

To relate the carry equation back to extensions, we can verify the following by direct calculation:

Proposition A.3. *The rule $m \bullet (s, a) := (m \bullet s, \delta_m(s) + a)$ is a monoid action of M on $S \times A$ if and only if δ satisfies the carry equations.*

To bring this in line with more standard mathematical machinery, we note that the set of "readout functions" $[S, A]$ inherits an action of M from S: $(f \bullet m)(s) := f(m \bullet s)$. Note that this is a *right* action, while the action on S is a *left* action.

Writing δ in its curried form $D : M \to [S, A]$, we can rewrite the carry equation as follows:

$$\begin{aligned} D(1) &= 0 \\ D(n \cdot m) &= D(n) \bullet m + D(m) \end{aligned} \tag{13}$$

Functions D with these properties appear in the literature under three names: *crossed homomorphisms* (to emphasise that this specialises to Equation 8 when M acts trivially), *derivations* (to emphasise that this generalises to the Leibniz equation when M acts nontrivially on the left as well as the right), and 1-*cocycles* (to emphasise the possibly higher-dimensional situation of k-cocycles $M^k \to [S, A]$).

Note that it is more common to describe these objects in terms of left actions rather than right actions. But the two viewpoints are compatible, which we discuss in more detail in Appendix B.

B Star-monoids and group cocycles

We have related left actions of a monoid M to state machines. The reader may ask: does the right action on $[S, A]$ we describe above also relate to state machines?

Define M^{op}, the *opposite* of M, to be the monoid with the same underlying set as M, but reversed multiplication. Then it is easy to verify that a left action of M is equivalent to a right action of M^{op}, and likewise a left action of M^{op} is equivalent to a right action of M.

So $[S, A]$ can be interpreted as a state machine acted on by M^{op}. But with some additional structure, we will show that we can reinterpret this as a state machine acted on by M.

Define a *star-monoid* to be a monoid M equipped with an isomorphism $* : M \to M^{op}$ between M and its opposite. A homomorphism of star-monoids $f : M \to N$ is a homomorphism of the underlying monoids, satisfying $f(m^*) = f(m)^*$.

There is also a short categorical definition of star-monoids. Just as monoids can be identified with one-object categories, star-monoids can be identified with one-object *dagger categories*, a common tool in category theory. For example, the category of real vector spaces with a fixed basis has a dagger structure given by the transpose.

The category of commutative monoids has an embedding into the category of star-monoids: we simply let $*$ be the identity function. By commutativity, $*$ is an isomorphism $M \to M^{op}$.

But groups are also star-monoids, in a completely different way. In any group G, we have the equation $(gh)^{-1} = h^{-1}g^{-1}$, so $g^* = g^{-1}$ gives an isomorphism $G \to G^{op}$. So the category of groups embeds into the category of star monoids (beware: abelian groups can be treated as either commutative monoids or groups, and the two approaches give incompatible stars except in special cases).

This means that, if M has the structure of a star-monoid, we can rewrite Equation 13 as $D(n \cdot m) = m^* \bullet D(n) + D(m)$. So, if G and A are groups and A is abelian, we can take $M = G^{op}$ and our definition coincides with the standard definition of a group 1-cocycle for G with coefficients in $[S, A]$.[10]

In fact, it is a completely standard technique to define the left action of a group G on functions using the opposite group, i.e. $(gf)(s) := f(g^*s) = f(g^{-1}s)$, a definition which is heavily leveraged in group convolutional neural networks [43].

Since this technique of converting from right actions to left actions produces exactly the correct definitions for groups, we propose that star-monoids, and even more generally, dagger-categories, are an appropriate setting for generalising equivariant convolutions to the setting of monoids and categories—but we leave this to future work.

[10] We can even get non-abelian cocycles by replacing A with A^{op} as well.