Graph Neural Networks are Dynamic Programmers

Anonymous Author(s)
Affiliation
Address
email

Abstract

Recent advances in neural algorithmic reasoning with graph neural networks (GNNs) are propped up by the notion of algorithmic alignment. Broadly, a neural network will be better at learning to execute a reasoning task (in terms of sample complexity) if its individual components align well with the target algorithm. Specifically, GNNs are claimed to align with dynamic programming (DP), a general problem-solving strategy which expresses many polynomial-time algorithms. However, has this alignment truly been demonstrated and theoretically quantified? Here we show, using methods from category theory and abstract algebra, that there exists an intricate connection between GNNs and DP, going well beyond the initial observations over individual algorithms such as Bellman-Ford. Exposing this connection, we easily verify several prior findings in the literature, produce better-grounded GNN architectures for edge-centric tasks, and demonstrate empirical results on the CLRS algorithmic reasoning benchmark. We hope our exposition will serve as a foundation for building stronger algorithmically aligned GNNs.

1 Introduction

One of the principal pillars of neural algorithmic reasoning [24] is training neural networks that *execute* algorithmic computation in a high-dimensional latent space. While this process is in itself insightful, and can lead to stronger combinatorial optimisation systems [19], it is valuable in terms of expanding the applicability of classical algorithms. Evidence of this value are emerging, with pre-trained algorithmic reasoners utilised in implicit planning [11] and self-supervised learning [25].

A fundamental question in this space is: which *architecture* should be used to learn a particular algorithm (or collection of algorithms [33])? Naturally, we seek architectures that have low *sample complexity*, as they will allow us to create models that generalise better with fewer training examples.

The key theoretical advance towards achieving this aim has been made by [34]. Therein, the authors formalise the notion of *algorithmic alignment*, which states that we should favour architectures that align better to the algorithm, in the sense that we can separate them into modules, which individually correspond to the computations of the target algorithm's subroutines. It can be proved that architectures with higher algorithmic alignment will have lower sample complexity in the NTK regime [18]. Further, the theory of [34] predicts that graph neural networks (GNNs) algorithmically align with dynamic programming [3, DP]. The authors demonstrate this by forming an analogy to the Bellman-Ford algorithm [2].

Since DP is a very general class of problem-solving techniques that can be used to express many classical algorithms, this finding has placed GNNs as the central methodology for neural algorithmic execution [7]. However, it quickly became apparent that it is not enough to just train any GNN—for many algorithmic tasks, careful attention is required. Several papers illustrated special cases of GNNs that align with sequential algorithms [28], linearithmic sequence processing [15], physics simulations [21], iterative algorihtms [23], data structures [26] or auxiliary memory [22]. Some explanations for

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

this lack of easy generalisation have arisen—we now have both geometric [35] and causal [4] views into how better generalisation can be achieved.

We believe that the fundamental reason why so many isolated efforts needed to look into learning specific classes of algorithms is the fact the GNN-DP connection *has not been sufficiently explored*. Indeed, the original work of [34] merely mentions in passing that the formulation of DP algorithms seems to align with GNNs, and demonstrates one example (Bellman-Ford). Our thorough investigation of the literature yielded no concrete follow-up to this initial claim. But DP algorithms are very rich and diverse, often requiring a broad spectrum of computations. Hence what we really need is a *framework* that could allow us to identify GNNs that could align particularly well with certain *classes* of DP, rather than assuming a "one-size-fits-all" GNN architecture will exist.

As a first step towards this, in this paper we interpret the operations of *both* DP and GNNs from the lens of category theory and abstract algebra. We elucidate the GNN-DP connection by observing a diagrammatic abstraction of their computations, recasting algorithmic alignment to aligning the diagrams of (G)NNs to ones of the target algorithm class. In doing so, several previously shown results will naturally arise as corollaries, and we propose novel GNN variants that empirically align better to edge-centric algorithms. We hope our work opens up the door to a broader unification between algorithmic reasoning and the geometric deep learning blueprint [5].

2 GNNs, dynamic programming, and the categorical connection

Before diving into the theory behind our connection, we provide a quick recap on the methods being connected: graph neural networks and dynamic programming. Further, we cite related work to outline why it is sufficient to interpret DP from the lens of graph algorithms.

We will use the definition of GNNs based on [5]. 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:

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

where $\psi : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}^k$ is a *message function*, $\phi : \mathbb{R}^k \to \mathbb{R}^k$ is a *readout function*, and $\bigoplus$ is a permutation-invariant *aggregation function* (such as $\sum$ or $\max$). Both ψ and ϕ can be realised as MLPs, but many special cases exist, giving rise to, e.g., attentional GNNs [27].

Dynamic programming is defined as a process that solves problems in a *divide et impera* fashion: imagine that we want to solve a problem instance x. DP proceeds to identify a set of *subproblems*, $\eta(x)$, such that solving them first, and recombining the answers, can directly lead to the solution for x: $f(x) = \rho(\{f(y) \mid y \in \eta(x)\})$. Eventually, we decompose the problem enough until we arrive at an instance for which the solution is trivially given (i.e. $f(y)$ which is known upfront). From these "base cases", we can gradually build up the solution for the problem instance we initially care for in a bottom-up fashion. This rule is often expressed programmatically:

$$\texttt{dp[x]} \leftarrow \texttt{recombine(score(dp[y], dp[x]) for y in expand(x))} \tag{2}$$

To initiate our discussion on why DP can be connected with GNNs, it is a worthwhile exercise to show how Equation 2 induces a *graph* structure. To see this, we leverage a categorical analysis of dynamic programming first proposed by [10]. Therein, dynamic programming algorithms are reasoned about as a composition of *three* components (presented here on a high level):

$$\text{dp} = \underbrace{\rho}_{\texttt{recombine}} \circ \underbrace{\sigma}_{\texttt{score}} \circ \underbrace{\eta}_{\texttt{expand}} \tag{3}$$

Expansion selects the relevant subproblems; scoring computes the quality of each individual subproblem's solution w.r.t. the current problem, and recombining combines these solutions into a solution for the original problem (e.g. by taking the max, or average).

Therefore, we can actually identify every subproblem as a *node* in a *graph*. Let V be the space of all subproblems, and R an appropriate value space (e.g. the real numbers). Then, expansion is defined

as $\eta : V \to \mathcal{P}(V)$, giving the set of all subproblems relevant for a given problem. Note that this also induces a set of *edges* between subproblems, E; namely, $(x, y) \in E$ if $x \in \eta(y)$. Each subproblem is scored by using a function $\sigma : \mathcal{P}(V) \to \mathcal{P}(R)$. Finally, the individual scores are recombined using the recombination function, $\rho : \mathcal{P}(R) \to R$. The final dynamic programming primitive therefore computes a function $\mathrm{dp} : V \to R$ in each of the subproblems of interest.

Therefore, dynamic programming algorithms can be seen as performing computations over a *graph of subproblems*, which can usually be precomputed for the task at hand (since the outputs of η are assumed known upfront for every subproblem). One specific popular example is the Bellman-Ford algorithm [2], which computes single-source shortest paths from a given source node, s, in a graph $G = (V, E)$. In this case, the set of subproblems is exactly the set of nodes, V, and the expansion $\eta(u)$ is exactly the set of one-hop neighbours of u in the graph. The algorithm maintains *distances* of every node to the source, d_u. The rule for iteratively recombining these distances is as follows:

$$d_u \quad \leftarrow \quad \min\left(d_u, \min_{v \in \mathcal{N}_u} d_v + w_{v \to u}\right) \tag{4}$$

where $w_{v \to u}$ is the distance between nodes v and u. The algorithm's base cases are $d_s = 0$ for the source node, $d_u = +\infty$ otherwise. Note that more general forms of Bellman-Ford pathfinding exist, for appropriate definitions of + and $\min$ (in general known as a *semiring*). Several recent research papers such as NBFNet [36] explicitly call on this alignment in their motivation.

3 The difficulty of connecting GNNs and DP

The basic technical obstacle to establishing a rigorous correspondence between neural networks and DP is the vastly different character of the computations they perform. Neural networks are built from linear algebra over the familiar real numbers, while DP, which is often a generalisation of path-finding problems, typically takes place over "tropical" objects like $(\mathbb{N} \cup \{\infty\}, \min, +)$[1], which are usually studied in mathematics as "degenerations" of Euclidean space. The two worlds cannot clearly be reconciled, directly, with simple equations.

However, if we define an arbitrary "latent space" R and make as few assumptions as possible, we can observe that many of the behaviors we care about, *for both GNNs and DP*, arise from looking at functions $S \to R$, where S is a finite set. R can be seen as the set of real-valued vectors in the case of GNNs, and the tropical numbers in the case of DP.

So our principal object of study is the category of finite sets, and "R-valued quantities" on it. By "category" here we mean a collection of *objects* (all finite sets) together with a notion of composable *arrows* (functions between finite sets).

To draw our GNN-DP connection, we need to devise an abstract object which can capture both the GNN's message passing/aggregation stages (Equation 1) and the DP's scoring/recombination stages (Equation 2). It may seem quite intuitive that these two concepts can and should be relatable, and category theory is a very attractive tool for *"making the obvious even more obvious"* [14]. Indeed, recently concepts from category theory have enabled the construction of powerful GNN architectures beyond permutation equivariance [9]. Here, we propose **integral transforms** as such an object.

We will construct the integral transform by composing transformations over our input features in a way that will depend minimally on the specific choice of R. In doing so, we will build a computational diagram that will be applicable for both GNNs and DP (and their own choices of R), and hence allowing for focusing on making components of those diagrams as aligned as possible.

4 The integral transform

An integral transform can be encoded in a diagram of this form, which we call a *polynomial span*:

[1] Here we require the addition of a special "∞" placeholder object to denote the vertices the DP expansion hasn't reached so far.

$$
\begin{tikzcd}
X \arrow[r, "p"] \arrow[d, "i"'] & Y \arrow[d, "o"] \\
W & Z
\end{tikzcd}
$$

where W, X, Y and Z are finite sets. The arrows i, p, o stand, respectively, for "input", "process", and "output". In context, the sets will have the following informal meaning: W represents the set over which we define our *inputs*, Z the set over which we define *outputs*. X and Y are, respectively, carrier sets for the *arguments*, and the *messages*[2]—we will clarify their meaning shortly.

Before proceeding, it is worthy to note the special case of $X = Y = E$, with p being the identity map. Such a diagram is commonly known as a *span*. A span that additionally has $W = Z = V$ is equivalent to a representation of a directed graph with vertex set Z and edge set Y ($V \leftarrow E \rightarrow V$); in this case $i(e)$ and $o(e)$ are the functions identifying the source and target nodes of each edge.

The key question is: given input data f on W, assigning features $f(w)$ to each $w \in W$, how to transform it, via the polynomial span, into data on Z? If we can do this, we will be able to characterise both the process of sending messages between nodes in GNNs *and* scoring subproblems in DP.

For us, data on a carrier set S consists of an element of $[S, R] := \{f : S \rightarrow R\}$, where R is a "set of possible values". For now, we will think of R as an arbitrary (usually infinite) set, though we will see later that it should possess some algebraic structure; it should be a semiring.

The transform proceeds in three steps, following the edges of the polynomial span:

$$
\begin{tikzcd}
{[X, R]} \arrow[r, "p_\otimes"] & {[Y, R]} \arrow[d, "o_\oplus"] \\
{[W, R]} \arrow[u, "i^*"] & {[Z, R]}
\end{tikzcd}
\tag{5}
$$

We call the three arrows $i^*, p_\otimes, o_\oplus$ the *pullback*, the *argument pushfoward*, and the *message pushforward*. Taken together, they form an *integral transform*—and we conjecture that this transform can be described as a polynomial functor, where $p_\otimes$ and $o_\oplus$ correspond to the *dependent product* and *dependent sum* from type theory (cf. Appendix C for details).

The pullback i^* is the easiest to define. Since we have a function $i : X \rightarrow W$ (part of the polynomial span) and a function $f : W \rightarrow R$ (our input data), we can produce data on X, that is, a function in $X \rightarrow R$, by composition. We hence define $i^* f = f \circ i$.

Unfortunately, the other two arrows of the polynomial span point in the wrong direction for naïve composition. For the moment, we will focus on how to define $o_\oplus$ and leave $p_\otimes$ for later.

We start with message data $m : Y \rightarrow R$. It may be attractive to *invert* the output arrow o in order to define a composition with o^{-1}, as was done in the case of the pullback. However, unless o is bijective, the preimage $o^{-1} : Z \rightarrow \mathcal{P}(Y)$ takes values in the *power set* of Y. There is an additional technicality: if the composition $m \circ o^{-1}$ takes values in $\mathcal{P}(R)$, it will fail to detect multiplicities; we are unable to tell from a subset of R whether multiple messages had the same value.

So instead, our pushforward takes values in $\mathtt{bag}(R)$, the set of finite multisets (or bags) of R, which we describe in more detail in the next section. For the moment, it is enough to know that a bag is equivalent to a formal sum, and we define an intermediate message pushforward $(\overline{o_\oplus} m)(u) := \sigma_{e \in t^{-1}(u) m(e)} \in [Z, \mathtt{bag}(R)]$.

[2]For technical reasons, we ask that for each $y \in Y$, the preimage $p^{-1}(y) = \{x \in X \mid p(x) = y\}$ should have a total order. This is to properly support functions with non-commuting arguments, though this detail is unnecessary for our key examples, where arguments commute.

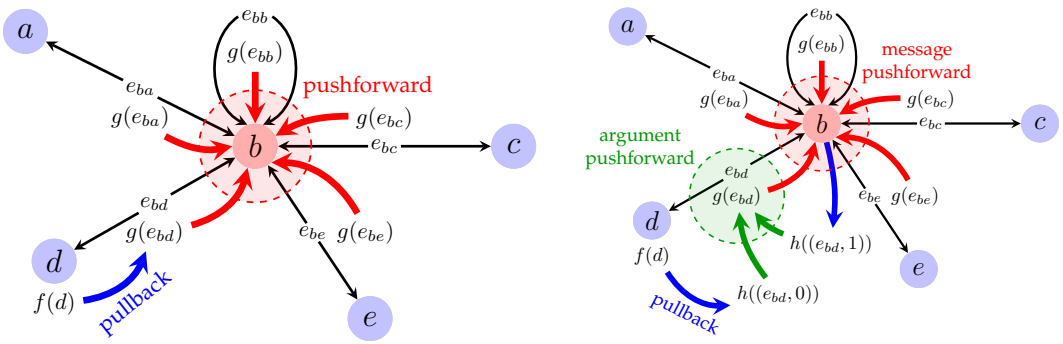

Figure 1: The illustration of how pullback and pushforward combine to form the *integral transform*, for two specific cases. **Left:** Polynomial span $V \leftarrow E \rightarrow E \rightarrow V$ with trivial argument pushforward (identity). Each edge e_{uv} is connected to its sender and receiver nodes (u, v) via the *span* (black arrows). The *pullback* then "pulls" the node features $f(u)$ along the span, which the *argument pushforward* folds into edge features $g(e_{vu}) = f(u)$. Once all sender features are pulled back to their edges, the *message pushforward* then "collects" all of the edge features that send to a particular receiver, by pushing them along the span. **Right:** Polynomial span $V \leftarrow E + E \rightarrow E \rightarrow V$, a situation more commonly found in GNNs. In this case, the pullback pulls sender and receiver node features into the argument function, h. The *argument pushforward* then computes, from these arguments, the edge messages, g, which are sent to receivers via the *message pushforward*, as before.

All that is missing to complete our definition of $o_\oplus$ is an *aggregator* $\bigoplus : \texttt{bag}(R) \rightarrow R$. As we will see later, specifying a well-behaved aggregator is the same as imposing a *commutative monoid* structure on R. With such an aggregator on R, we can define $(o_\oplus m)(u) := \bigoplus(\overline{o_\oplus} m)(u)$.

We return to $p_\otimes$, which is constructed very similarly. The only difference is that, while we deliberately regard the collection of messages as unordered, the collection of arguments used to compute a message has an *ordering* we wish to respect. So instead of the type $\texttt{bag}(R)$, we use the type $\texttt{list}(R)$ of finite lists of elements of R, and our aggregator $\bigotimes : \texttt{list}(R) \rightarrow R$ is now akin to a *fold* operator.

We illustrate the use of these two aggregators in a decomposed diagram:

$$
\begin{tikzcd}
 & {[Y, \texttt{list}(R)]} \arrow[dr, "\bigotimes"] & & \\
{[X, R]} \arrow[ur, "\overline{p_\otimes}"] \arrow[rr, dashed, "p_\otimes"] & & {[Y, R]} \arrow[d, dashed, "o_\oplus"] \arrow[dr, "\overline{o_\oplus}"] & \\
{[W, R]} \arrow[u, "i^*"] & & {[Z, R]} & {[Z, \texttt{bag}(R)]} \arrow[l, "\bigoplus"]
\end{tikzcd}
$$

Note that any semiring $(R, \otimes, \oplus)$ comes equipped with binary operators $\otimes, \oplus$ that allow aggregators $\bigotimes, \bigoplus$ to be defined inductively. In fact, the converse—that every set with two such aggregators is a semiring—is also true, if we assume some reasonable conditions on the aggregators, which we can explain in terms of one of the most utilised concepts in category theory and functional programming—*monads* [30]. Due to space constraints, we refer the interested reader to Appendices A and B for a full exposition of how we can use monads over lists and bags to constrain the latent space R to respect a semiring structure.

For now, it's enough to know that our key examples of the real numbers (with multiplication and addition, for GNNs) and the tropical natural numbers (with addition and minimum, for DP) both allow for natural interpretations of $\bigotimes$ and $\bigoplus$ in the integral transform.

We are now ready to show how the integral transform can be used to instantiate popular examples of algorithms and GNNs. We start with the Bellman-Ford algorithm [2] (Equation 4) that was traditionally used to demonstrate the concept of algorithmic alignment.

5 Bellman-Ford

Let $R = (\mathbb{N} \cup \{\infty\}, +, \min)$ be the "min-plus" semiring of extended natural numbers, with $\otimes = +$ and $\oplus = \min$. This is the coefficient semiring over which the Bellman-Ford algorithm takes place.

Let (V, E) be a weighted graph with source and target maps $s, t : E \to V$ and edge weights $w : E \to R$. For purely technical reasons, we also need to explicitly materialise a *bias function* $b : V \to R$, which is, in practice, a constant-zero function ($b(v) = 0$ for all $v \in V$) but will prove necessary for defining the argument pushforward.

We interpret Bellman-Ford as the following polynomial span:

$$\begin{tikzcd} (V+E)+(V+E) \arrow[r, "p"] \arrow[d, "i"] & V+E \arrow[d, "o"] \\ V+(V+E) & V \end{tikzcd} \tag{6}$$

Here "+" is the disjoint union of sets, defined as $A + B = \{(a, 1) \mid a \in A\} \cup \{(b, 2) \mid b \in B\}$. Note that $[S + T, R] \cong [S, R] \times [T, R]$, i.e. specifying data on a disjoint union is equivalent to specifying data on each component separately.

Initially, we describe each of the four sets of the polynomial span, making their role clear:

- **Input:** $W = V + (V + E)$. Our input to Bellman-Ford includes: the current estimate of node distances (d_u; a function in $[V, R]$), edge weights (w; a function in $[E, R]$), and the previously discussed bias b, a function in $[V, R]$. Hence our overall inputs are members of $[V, R] \times [E, R] \times [V, R] \cong [V + (V + E), R]$, justifying our choice of input space.
- **Arguments:** $X = (V + E) + (V + E)$. Here we collect the ingredients necessary to compute Bellman-Ford's subproblem solutions coming from neighbouring nodes. To do this, we need to combine data in the nodes with data living on edges—those are the *arguments* to the function. And since they meet in the edges, we "lift" our node distances $[V, R]$ to edges they are sending from, giving us an additional function in $[E, R]$. Hence our argument carrier space is now $(V + E) + (V + E)$ (the remaining three inputs remain unchanged).
- **Message:** $Y = V + E$. Once the arguments are combined to compute messages, we are left with signal in each edge (containing the sum of corresponding d_u and w_{uv}), and each node (containing just d_u, for the purposes of access to the previous optimal solution). Hence our messages are members of $[V, R] \times [E, R]$, justifying our choice of message space.
- **Output:** $Z = V$. Lastly, the output of one step of Bellman-Ford are updated values d'_u, which we can interpret as just (output) data living on V.

We now describe how to propagate data along each arrow of the diagram in turn, beginning with inputs (f, b, w) of node features $f : V \to R$, a bias $b : V \to R$, and edge weights $w : E \to R$:

- **Pullback, i^*:** First, we can note the input function $i : (V + E) + (V + E) \to V + (V + E)$ decomposes as the sum of two arrows. $i_1 : V + E \to V$ is the identity function on V and the source function on E, and $i_2 : V + E \to V + E$ is just the identity. So we calculate the pullback $i^*(f, b, w) = (f, f \circ s, b, w)$, giving us the arguments to compute messages.
- **Argument pushforward, $p_\otimes$:** Next, the process function p simply identifies the two copies of $V + E$, and sums their values. So the argument pushforward is $p_\otimes(f, f \circ s, b, w) =$

$(f, f \circ s) \otimes (b, w) = (f + b, (f \circ s) + w)$. This also allows us to interpret the bias function, b, as a "self-edge" in the graph with weight 0.

- **Message pushforward, $o_\oplus$:** The output function $o : V + E \to V$ is the identity function on V and the target function on E. So the message pushforward gives us $(o_\oplus(f + b, (f \circ s) + w))(u) = (f(u) + b(u)) \oplus \bigoplus_{t(e)=u}(f \circ s)(e) = \min(f(u) + b(u), \min_{v \to u} f(v) + w_{v \to u})$.

Letting $b(u) = 0$, we can see that this is *exactly* Equation 4. So we have produced the formula for the Bellman-Ford algorithm directly from the polynomial span in Diagram 6.

Note that $p_\oplus$ is aligned with using max aggregation in neural networks—directly explaining several previous proposals, such as [28]. But additionally, $p_\otimes$, as defined, is aligned with concatenating all message arguments together and passing them through a linear function, which is how such a step is implemented in GNNs' message functions. We now direct our polynomial span analysis at GNNs.

6 GNNs

We study the popular *message passing neural network* (MPNN) model [17], which can be interpreted using the following polynomial span diagram:

$$\begin{tikzcd}
E + (E + E) + E \arrow[r, "p"] \arrow[d, "i"] & E \arrow[d, "o"] \\
1 + V + E & V
\end{tikzcd}$$

Here the set 1 refers to a singleton set—sometimes also called $()$, or *unit*—which is used as a carrier for graph-level features. This implies the graph features will be specified as $[1, R] \cong R$, as expected.

Given all these features, how would we compute messages? The natural way is to combine the features of the sender and receiver node of each edge, features of said edge, and graph-level features—these will form our arguments, and they need to all "meet" in the edges. This motivates our argument space as $E + (E + E) + E$: all of the above four, accordingly broadcast into their respective edge(s).

The input map, i, is then the unique map to the singleton, the sender and receiver functions on the two middle copies of E, and the identity on the last copy of E, i.e. $i(a, b, c, d) = \{(), s(b), t(c), d\}$. The process map, p, collapses the four copies of E into just one, to hold the computed message. Lastly, the output map, o, is the target function, identifying the node to which the message will be delivered.

The actual computation performed by the network (over real values in $\mathbb{R}$, which can support various semirings of interest) is exactly an integral transform, with an extra MLP processing step on messages:

$$\begin{tikzcd}
{[E + (E + E) + E, \mathbb{R}]} \arrow[r, "p_\otimes"] & {[E, \mathbb{R}]} \arrow[loop right, "MLP"] \arrow[d, "p_\oplus"] \\
{[1 + V + E, \mathbb{R}]} \arrow[u, "i^*"] & {[V, \mathbb{R}]}
\end{tikzcd}$$

It is useful to take a moment to discuss what was just achieved: with a single abstract template (the polynomial span), we have successfully explained both a dynamic programming algorithm, and a GNN update rule—merely by choosing the correct support sets and latent space.

7 Improving GNNs with edge updates, with experimental evaluation

From now on, we will set $E = V^2$, as all our baseline GNNs will use fully connected graphs, and it will accentuate the *polynomial* nature of our construction.

We now show how our polynomial span view can be used to directly propose better-aligned GNN architectures for certain algorithmic tasks. Since the MPNN diagram above outputs only node features, to improve predictive performance on edge-centric algorithms, it is a natural augmentation to also update edge features, by adding edges to the output carrier (as done by, e.g., [1]):

$$\begin{tikzcd} V^2 + (V^2 + V^2) + V^2 \arrow[r, "p"] \arrow[d, "i"] & V^2 \arrow[d, "o"] \\ 1 + V + V^2 & V + V^2 \end{tikzcd} \tag{7}$$

But notice that there is a problem with the output arrow. Since we are using each message twice, o is no longer a function—it'd have to send each edge message to *two* different objects! To resolve this, we need to appropriately augment the messages and the arguments. This is equivalent to specifying a new polynomial span with output V^2, which we can then recombine with Diagram 7:

$$\begin{tikzcd} ? \arrow[r, "p"] \arrow[d, "i"] & ? \arrow[d, "o"] \\ 1 + V + V^2 & V^2 \end{tikzcd} \tag{8}$$

Most edge-centric algorithms of interest (such as the Floyd-Warshall algorithm for all-pairs shortest paths [13]), compute edge-level outputs by *reducing* over a choice of "intermediate" node. Hence, it would be beneficial to produce messages with shape V^3, which would then reduce to features over V^2. There are three possible ways to broadcast both node and edge features into V^3, so we propose the following polynomial span, which materialises each of those arguments:

$$\begin{tikzcd} V^3 + (V^3 + V^3 + V^3) + (V^3 + V^3 + V^3) \arrow[r, "p"] \arrow[d, "i"] & V^3 \arrow[d, "o"] \\ 1 + V + V^2 & V^2 \end{tikzcd} \tag{9}$$

Finally, inserting this into Diagram 7 gives us a corrected polynomial span with output $V + V^2$:

$$\begin{tikzcd} 4V^2 + 7V^3 \arrow[r, "p"] \arrow[d, "i"] & V^2 + V^3 \arrow[d, "o"] \\ 1 + V + V^2 & V + V^2 \end{tikzcd} \tag{10}$$

Here we have collapsed the copies of V^2 and V^3 in the argument position for compactness.

While Diagram 7 doesn't make sense as a polynomial diagram of sets, we can clearly still implement it as an architecture [1], since nothing stops us from sending the same tensor to two places. We want to investigate whether our proposed modification of Diagram 10, which materialises order-3 messages, leads to improved algorithmic alignment on edge-centric algorithms. To support this evaluation, we use six edge-centric tasks from the recently proposed CLRS Algorithmic Reasoning Benchmark [29], which evaluates how well various (G)NNs align to classical algorithms, both in- and out-of-distribution.

We implemented each of these options using a two-layer MLP with embedding dimension 24, and hidden layers of size 8 and 16. Our test results (out-of-distribution) are summarised in Table 1.

For convenience, we also illustrate the in-distribution performance of our models via plots given in Appendix D.

We found that the V^3 architecture was equivalent to, or outperformed, the non-polynomial (V^2) one in all cases. In particular, the size 8 polynomial architecture has fewer parameters than the size 16 non-polynomial one, but shows substantial improvements in several algorithms that benefit from aligned edge features. Therefore, this result directly validates our theory's predictions, in the context of presenting a better-aligned GNN for edge-centric algorithmic targets.

Table 1: Test (out-of-distribution) results of our models on all models on the six algorithms studied. V^2 corresponds to the baseline model offered by Diagram 7, while V^3 corresponds to our proposal in Diagram 10, which respects the polynomial span.

Algorithm	V^2**-large**	V^3**-large**	V^2**-small**	V^3**-small**
Dijkstra	$59.58\% \pm 2.82$	$\mathbf{68.53\% \pm 2.40}$	$56.10\% \pm 3.25$	$60.32\% \pm 2.70$
Find Maximum Subarray	$8.33\% \pm 0.50$	$\mathbf{9.06\% \pm 0.65}$	$8.46\% \pm 0.55$	$7.89\% \pm 0.64$
Floyd-Warshall	$7.46\% \pm 0.63$	$\mathbf{9.00\% \pm 0.81}$	$6.66\% \pm 0.62$	$8.23\% \pm 0.62$
Insertion Sort	$15.39\% \pm 1.27$	$\mathbf{24.67\% \pm 2.44}$	$14.69\% \pm 1.32$	$20.23\% \pm 2.21$
Matrix Chain Order	$67.64\% \pm 1.23$	$\mathbf{70.79\% \pm 1.54}$	$68.85\% \pm 2.26$	$68.76\% \pm 1.21$
Optimal BST	$53.03\% \pm 2.80$	$\mathbf{54.56\% \pm 4.34}$	$46.65\% \pm 3.82$	$51.94\% \pm 4.60$
Overall average	35.24%	$\mathbf{39.43\%}$	33.57%	36.23%

8 Conclusions

In this paper, we describe the use of category theory and abstract algebra to explicitly expand on the GNN-DP connection, which was previously largely handwaved on specific examples. We derived a generic diagram of an *integral transform* (based on standard categorical concepts like pullback, pushforward and commutative monoids), and argued why it is general enough to support both GNN and DP computations. With this diagram materialised, we were able to immediately unify large quantities of prior work as simply manipulating one arrow or element in the integral transform. We also provided empirical evidence of the utility of polynomial spans for analysing GNN architectures, especially in terms of algorithmic alignment. It is our hope that our findings inspire future research into better-aligned neural algorithmic reasoners, especially focusing on generalising or diving into several aspects of this diagram.

Lastly, it is not at all unlikely that analyses similar to ours have already been used to describe other fields of science—beyond algorithmic reasoners. The principal ideas of *span* and *integral transform* are central to defining Fourier series [32], and appear in the analysis of Yang-Mills equations in particle physics [12]. Properly understanding the common ground behind all of these definitions may, in the very least, lead to interesting connections, and a shared understanding between the various fields they span.

References

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

[3] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[4] Beatrice Bevilacqua, Yangze Zhou, and Bruno Ribeiro. Size-invariant graph representations for graph classification extrapolations. In *International Conference on Machine Learning*, pages 837–851. PMLR, 2021.

[5] Michael M Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.

[6] Francesca Cagliari and Sandra Mantovani. Cartesianness: topological spaces, uniform spaces, and affine schemes. *Topology and its Applications*, 41:263–272, 1991.

[7] Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint arXiv:2102.09544*, 2021.

[8] Eugenia Cheng. Iterated distributive laws. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 150, pages 459–487. Cambridge University Press, 2011.

[9] Pim de Haan, Taco S Cohen, and Max Welling. Natural graph networks. *Advances in Neural Information Processing Systems*, 33:3636–3646, 2020.

[10] Oege De Moor. Categories, relations and dynamic programming. *Mathematical Structures in Computer Science*, 4(1):33–69, 1994.

[11] Andreea-Ioana Deac, Petar Veličković, Ognjen Milinkovic, Pierre-Luc Bacon, Jian Tang, and Mladen Nikolic. Neural algorithmic reasoners are implicit planners. *Advances in Neural Information Processing Systems*, 34, 2021.

[12] Michael G Eastwood, Roger Penrose, and RO Wells. Cohomology and massless fields. *Commun. Math. Phys*, 78(3):305–351, 1981.

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

[14] Brendan Fong and David I Spivak. *An invitation to applied category theory: seven sketches in compositionality*. Cambridge University Press, 2019.

[15] Karlis Freivalds, Emīls Ozoliņš, and Agris Šostaks. Neural shuffle-exchange networks-sequence processing in o (n log n) time. *Advances in Neural Information Processing Systems*, 32, 2019.

[16] Jeffrey Giansiracusa and Noah Giansiracusa. Equations of tropical varieties. *Duke Mathematical Journal*, 165(18):3379–3433, 2016.

[17] 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.

[18] Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. *Advances in neural information processing systems*, 31, 2018.

[19] Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020.

[20] Susan Niefield. Cartesianness: topological spaces, uniform spaces, and affine schemes. *J. Pure Appl. Alg.*, 23:147–163, 1982.

[21] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, pages 8459–8468. PMLR, 2020.

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

[23] 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.

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

[25] Petar Veličković, Matko Bošnjak, Thomas Kipf, Alexander Lerchner, Raia Hadsell, Razvan Pascanu, and Charles Blundell. Reasoning-modulated representations. *arXiv preprint arXiv:2107.08881*, 2021.

[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.

[27] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

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

[29] Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. The clrs algorithmic reasoning benchmark. 2021.

[30] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 61–78, 1990.

[31] Mark Weber. Polynomials in categories with pullbacks. *Theory and Applications of Categories*, 30(16):533–598, 2015.

[32] Simon Willerton. Integral transforms and the pull-push perspective, i. *The n-Category Café*, 2020.

[33] 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, 2021.

[34] 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.

[35] 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.

[36] Zhaocheng Zhu, Zuobai Zhang, Louis-Pascal Xhonneux, and Jian Tang. Neural bellman-ford networks: A general graph neural network framework for link prediction. *Advances in Neural Information Processing Systems*, 34, 2021.

Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] See Section **??**.
- Did you include the license to the code and datasets? [No] The code and the data are proprietary.
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...

(a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] We propose a novel approach to reinterpret both graph neural networks and dynamic programming, and show empirical gains from an architecture motivated by our blueprint.

(b) Did you describe the limitations of your work? [Yes]

(c) Did you discuss any potential negative societal impacts of your work? [N/A] Our work is of a theoretical nature.

(d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...

(a) Did you state the full set of assumptions of all theoretical results? [Yes]

(b) Did you include complete proofs of all theoretical results? [Yes] Appropriate references to proofs are provided in all areas where proofs are missing.

3. If you ran experiments...

(a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] We aim to release the code at a future point. The data is publicly available.

(b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] The data generation and base model implementation is publicly available within the CLRS benchmark. We detail the model extensions we made.

(c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]

(d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...

(a) If your work uses existing assets, did you cite the creators? [Yes] We cite the CLRS benchmark.

(b) Did you mention the license of the assets? [N/A]

(c) Did you include any new assets either in the supplemental material or as a URL? [N/A]

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] Our data is abstract and algorithmically generated.

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A The bag and list monads

Before we conclude, we turn back to the theory behind our polynomial spans, to more precisely determine the restrictions on our abstract latent space R. We found this investigation useful to include in the main paper, as it yields a strong connection to one of the most actively used concepts in theoretical computer science and functional programming.

Recall that the realisation of our pushforward operations required the existence of two aggregators: $\bigotimes$ (to fold lists) and $\bigoplus$ (to reduce bags). Previously, we mentioned only in passing how they can be recovered—now, we proceed to define $\bigoplus$ axiomatically.

Given a set S, we define $\texttt{bag}(S) := \{p : S \to \mathbb{N} \mid \#\{p(r) \neq 0\} < \infty\}$, the natural-valued functions of finite support on S. This has a clear correspondence to multisets over S: p sends each element of S to the amount of times it appears in the multiset. We can write its elements formally as $\sum_{s \in S} n_s s$, where all but finitely many of the n_s are nonzero.

Given a function $f : S \to T$ between sets, we can define a function $\texttt{bag}(f) : \texttt{bag}(S) \to \texttt{bag}(T)$, as follows: $\texttt{bag}(f)(\sum_{s \in S} n_s s) := \sum_{s \in S} n_s f(s)$, which we can write as $\sum_{t \in T} m_t t$, where $m_t = \sum_{f(s)=t} n_s$.

For each S, we can also define two special functions. The first is $\texttt{unit} : S \to \texttt{bag}(S)$, sending each element to its indicator function (i.e. an element $x \in S$ to the multiset $\{\!\{x\}\!\}$). The second is $\texttt{join} : \texttt{bag}(\texttt{bag}(S)) \to \texttt{bag}(S)$, which interprets a nested sum as a single sum.

These facts tell us that bag is a **monad**, a special kind of self-transformation of the category of sets. Monads are very general tools for computation, used heavily in functional programming languages (e.g. Haskell) to model the semantics of wrapped or enriched types. Monads provide a clean way for abstracting control flow, as well as gracefully handling functions with *side effects* [30].

It is well-known that the algebras for the monad bag are the *commutative monoids*, sets equipped with a commutative and associative binary operation and a unit element.

Concretely, a commutative monoid structure on a set R is *equivalent* to defining an *aggregator* function $\bigoplus : \texttt{bag}(R) \to R$ compatible with the unit and monad composition. Here, compatibility implies it should correctly handle sums of singletons and sums of sums, in the sense that the following two diagrams *commute*; that is, they yield the same result regardless of which path is taken:

$$\begin{tikzcd} R \arrow[r, "\texttt{unit}"] \arrow[dr, "\texttt{id}"'] & \texttt{bag}(R) \arrow[d, "\bigoplus"] \\ & R \end{tikzcd} \qquad \begin{tikzcd} \texttt{bag}(\texttt{bag}(R)) \arrow[r, "\texttt{bag}(\bigoplus)"] \arrow[d, "\texttt{join}"'] & \texttt{bag}(R) \arrow[d, "\bigoplus"] \\ \texttt{bag}(R) \arrow[r, "\bigoplus"] & R \end{tikzcd}$$

The first diagram explains that the outcome of aggregating a singleton multiset (i.e. the one produced by applying unit) with $\bigoplus$ is equivalent to the original value placed in the singleton. The second diagram indicates that the $\bigoplus$ operator yields the same results over a nested multiset, regardless of whether we choose to directly apply it twice (once on each level of nesting), or first perform the join function to collapse the nested multiset, then aggregate the collapsed multiset with $\bigoplus$.

So the structure of a commutative monoid on R is exactly what we need to complete our definition of the message pushforward $o_{\oplus}$. The story for the argument pushforward, $p_{\otimes}$, is remarkably similar.

Define $\texttt{list}(S) := \{(s_1, \ldots, s_n) \mid n \in \mathbb{N}, s_i \in S\}$, the set of all ordered lists of elements of S, including the empty list. Equivalently, $\texttt{list}(S) = \coprod_{n \geq 0} S^n$. We can also extend list to a functor: for a function $f : S \to T$, $\texttt{list}(f) : \texttt{list}(S) \to \texttt{list}(T)$ is just the well-known map operation: $\texttt{list}(f)(s_1, \ldots, s_n) := (f(s_1), \ldots, f(s_n))$.

list is also a monad, with $\texttt{unit} : S \to \texttt{list}(S)$ sending each $x \in S$ to the singleton list (x), and $\texttt{join} : \texttt{list}(\texttt{list}(S)) \to \texttt{list}(S)$ sending a list of lists to their concatenation.

The algebras for the list monads are *monoids*—not just commutative ones. So R needs a second monoid structure, possibly noncommutative, to support our definition of the argument pushforward. We detail how this can elegantly be done in our specific case in Appendix B.

B The monad for semirings

We have asked that R be an algebra for two monads: list and bag. But this is an unnatural condition without some compatibility between the two. It would more useful to find a single monad encapsulating both.

In general, the composition of two monads is not a monad. For example, the composite functor $\texttt{list} \circ \texttt{bag}$ does not support a monad structure.

However, the other composite $\texttt{bag} \circ \texttt{list}$ is actually a monad in a natural way, due to the existence of a *distributive law*, which is a natural transformation $\lambda : \texttt{list} \circ \texttt{bag} \to \texttt{bag} \circ \texttt{list}$ satisfying some axioms, see e.g. [8].[3]

It is easy to describe λ. Given any list of bags $(\sum_{i_1} a_{i_1}, \ldots, \sum_{i_n} a_{i_n})$, we have $\lambda(\sum_{i_1} a_{i_1}, \ldots, \sum_{i_n} a_{i_n}) = \sum_{i_1,\ldots,i_n}(a_{i_1}, \ldots, a_{i_n})$. In other words, λ takes a list of bags and returns the bag of all ordered selections from the list.

This is exactly how multiplication of sums works in a semiring. For example, if I think of a polynomial as a bag of monomials, and I want to compute a product of polynomials, I interpret this product as a list of polynomials, i.e. a list of bags. Then I expand it into a bag of lists (a sum of products), and finally perform the products to produce the resulting bag of monomials, i.e. polynomial.

So it shouldn't be a surprise that the algebras for the composite monad $\texttt{bag} \circ \texttt{list}$ are exactly semirings, i.e. sets R equipped with a commutative monoid structure $\bigoplus$, another monoid structure $\bigotimes$, and a "distributive law" $\bigotimes \bigoplus \to \bigoplus \bigotimes$, usually written as, e.g. $x(a+b)y = xay + xby$, and extended to arbitrary sums and products by induction.

Indeed, if R is an algebra for the monad $\texttt{bag} \circ \texttt{list}$, we have some "double aggregator" $ev : \texttt{bag}(\texttt{list}(R)) \to R$. We can recover $\otimes : \texttt{list}(R) \to R$ by packing our list into a singleton bag, and we can recover $\oplus : \texttt{bag}(R) \to R$ by packing our bag into a singleton list then applying λ.

C Polynomial functors

Polynomial spans are the starting point for our integral transform, but they are also the starting point for *polynomial functors*, which arise in dendent type theory. Let $\mathcal{C}$ be a locally cartesian closed category, and let $\mathcal{C}/A$ denote, for any object A of $\mathcal{C}$, the category of morphisms with target A. A polynomial functor starts with a polynomial span:

$$\begin{tikzcd} X \arrow[r, "p"] \arrow[d, "i"'] & Y \arrow[d, "o"'] \\ W & Z \end{tikzcd}$$

And it produces a composition of three functors:

$$\begin{tikzcd} \mathcal{C}/X \arrow[r, "\Pi_p"] & \mathcal{C}/Y \arrow[d, "\Sigma_o"'] \\ \mathcal{C}/W \arrow[u, "i^*"] & \mathcal{C}/Z \end{tikzcd} \tag{11}$$

Here Σ_o and Π_p are operations called the *dependent sum* and *dependent product* respectively.

Note that there is a direct correspondence between the three arrows in each of the diagrams 5 and 11. So it is very tempting to ask whether our integral transform is expressible as a polynomial functor. Can our results be rephrased in those terms?

We conjecture that the answer is yes. However, there are technical hurdles to overcome in comparing the two definitions. When computing with a polynomial functor, we start with a variable set E and a function $E \to W$, and produce a new set $\mathcal{F}(E)$ and a function $\mathcal{F}(E) \to Z$. Our integral transform starts with a function to a fixed set, and produces a function to the same set.

To see how these two pictures are related, it is helpful to consider the example of finite sets. A function $\pi : E \to W$ of finite sets can be expressed, up to equivalence, as a function $f : W \to \mathbb{N}$

[3] Cheng [8] also explains the general problem of composing three or more monads and its relation to the Yang-Baxter equation, which provides further intuition about the unit axioms for semirings.

given by $f(w) = \#\pi^{-1}(w)$. In other words, we can describe E by describing the size of the fiber over each element of W. Indeed, the existence of such a "decategorification" for transforms over spans was an early inspiration for our present work.

To replace $\mathbb{N}$ with an arbitrary semiring R, we would need to find a way to interpret a function $f : W \to R$ as a classifying morphism for some kind of bundle $E \to W$ in a suitable category of geometric objects over R. For $R = \mathbb{N}^\infty$, one possibility is to define a category of R-schemes, which should be certain types of topological spaces equipped with sheaves of R-modules–we don't know of a place this theory is developed, but it is certainly related to tropical schemes, defined in [16]. For $R = \mathbb{R}$, we can consider the more familiar category of manifolds, or more generally the category of locally compact Hausdorff spaces.

But do polynomial functors work in categories like this? While polynomial functors were developed in type theory over locally cartesian closed categories–too strong of a condition for interesting topology to occur–[31] has shown that polynomial functors can be defined in any category with pullbacks, as long as the "processor" morphism $p : X \to Y$ satisfies an abstract condition called *exponentiability*. i and o can still be arbitrary morphisms.

For some intuition, we quote two results on exponentiability. [6] shows that the exponentiable morphisms in the category of compact Hausdorff spaces are the local homeomorphisms. And [20] shows that a morphism $R \to S$ of commutative rings gives rise to an exponentiable morphism of affine schemes exactly when S is dualizable as an R-module. So exponentiability seems to be strongly linked to covering spaces in classical topology, as well as descent theory in modern algebraic geometry.

Expanding on these ideas is far out of scope for the present work, but we hope it gives a glimpse into the possibilities on the horizon.

D Plots of in-distribution performance on CLRS

For plots that illustrate in-distribution performance of our proposed V^3 model, against the non-polynomial (V^2) model, please refer to Figure 2. Our findings largely mirror the ones from out-of-distribution—with V^3 either matching the performance of the baseline or significantly outperforming it (e.g. on Insertion Sort and Floyd-Warshall). We do note that sometimes, matched performance by the non-polynomial V^2 baseline in-distribution can be *misleading*, as it significantly loses out to V^3 out of distribution (cf. Table 1).

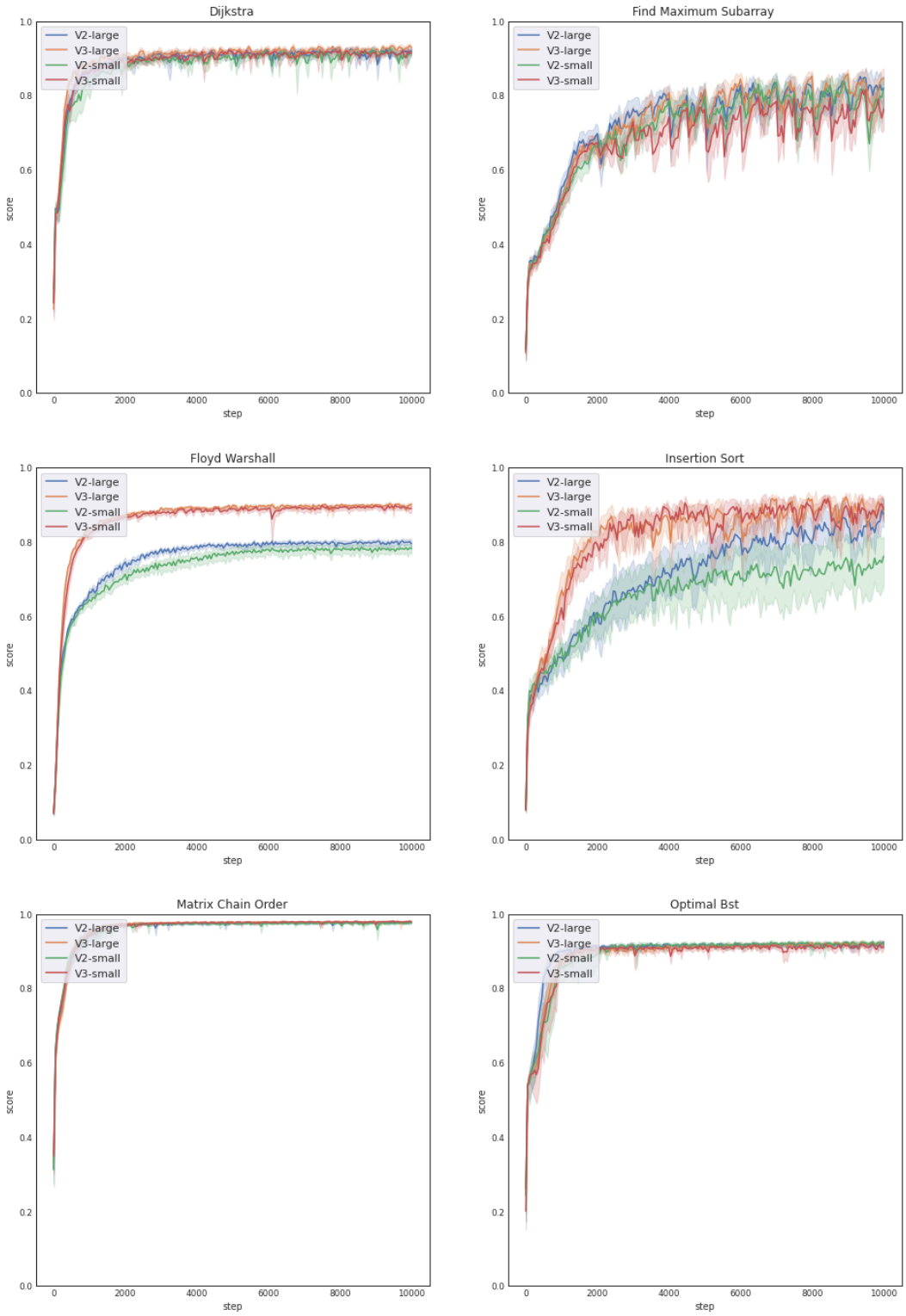

Figure 2: Validation curves of all models on all six algorithms studied, across three random seeds.