Graph Neural Networks are Dynamic Programmers

Andrew Dudzik*
DeepMind
adudzik@deepmind.com

Petar Veličković*
DeepMind
petarv@deepmind.com

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, and hope it will serve as a foundation for building stronger algorithmically aligned GNNs.

1 Introduction

One of the principal pillars of neural algorithmic reasoning (Veličković & Blundell, 2021) is training neural networks that *execute* algorithmic computation in a high-dimensional latent space. A fundamental question in this space is: which *architecture* should be used to learn a particular algorithm? We seek architectures with low *sample complexity*: generalising better with fewer training examples.

The key theoretical advance towards this aim was made by Xu et al. (2019). Therein, the notion of *algorithmic alignment* is formalised, favouring architectures that align better to the algorithm, in the sense that we can separate them into modules, which correspond to the computations of the target algorithm's subroutines. The theory predicts that graph neural networks (GNNs) algorithmically align with dynamic programming (DP). However, it quickly became apparent that it is not enough to just train any GNN—many recent works propose specialised GNNs that align with linearithmic (Freivalds et al., 2019), iterative (Tang et al., 2020) or structured algorithms (Veličković et al., 2020).

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 Xu et al. (2019) 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.

Here, we make a first step towards 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. We interpret the operations of *both* DP and GNNs from the lens of category theory and abstract algebra. In doing so, several previously shown results will naturally arise as corollaries, and we hope it opens up the door to a broader unification between algorithmic reasoning and the geometric deep learning blueprint (Bronstein et al., 2021).

2 GNNs, dynamic programming, and the categorical connection

We will use the definition of GNNs based on Bronstein et al. (2021). Let a graph be a tuple of *nodes* and *edges*, $G = (V, E)$, with one-hop neighbourhoods defined as $\mathcal{N}_u = \{v \in V \mid (v, u) \in E\}$.

*Equal contribution.

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:

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

where $\psi : \mathbb{R}^k \times \mathbb{R}^k \rightarrow \mathbb{R}^k$ is a *message function*, $\phi : \mathbb{R}^k \rightarrow \mathbb{R}^k$ 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 (Veličković et al., 2017).

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 a 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:

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

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 De Moor (1994). Therein, dynamic programming algorithms are reasoned about as a composition of *three* components (presented here on a high level):

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

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). Let V be the space of all subproblems, and R an appropriate value space (e.g. the real numbers). Then, expansion is defined as $\eta : V \rightarrow \mathcal{P}(V)$, 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 $\sigma : \mathcal{P}(V) \rightarrow \mathcal{P}(R)$. Finally, the individual scores are recombined using the recombination function, $\rho : \mathcal{P}(R) \rightarrow R$. The final dynamic programming primitive therefore computes a function $\mathrm{dp} : V \rightarrow R$ in each of the subproblems of interest.

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 $S \rightarrow R$, 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

[1]Here we add the "∞" object to denote the vertices the DP expansion hasn't reached so far.

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"* (Fong & Spivak, 2019). Indeed, recently concepts from category theory have enabled the construction of powerful GNN architectures beyond permutation equivariance (de Haan et al., 2020).

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 Four Arrows of the Integral Transform

A directed graph (V, E) is defined by two functions $s, t : E \to V$, defining the sender and receiver nodes for each edge. We express this in the following diagram:

$$\begin{tikzcd} V & E \arrow[l, "s"'] \arrow[r, "t"] & V \end{tikzcd}$$

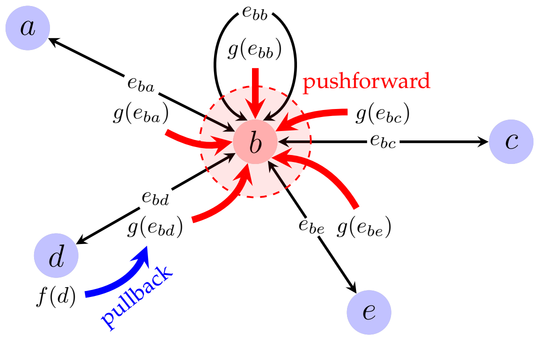

Figure 1: The illustration of how pullback and pushforward combine to form the *integral transform*. Each edge e_{uv} is connected to its sender and receiver nodes (u, v) via the *span* (black arrows). The *pullback* then "pulls" the node features $f(u)$ along the span, into edge features $g(e_{vu}) = f(u)$. Once all sender features are pulled back to their edges, the *pushforward* then "collects" all of the edge features that send to a particular receiver, by pushing them along the span.

The general question is: given some data f on V assigning features $f(v)$ to each $v \in V$, how to transform it via this diagram into some other data on V? If we are able to do this, we will be able to characterise both the processes of sending messages between nodes in GNNs *and* scoring subproblems in DP.

Initially, we will focus on the cases where the "message" sent along edge $e \in E$ depends only on the sender and not the receiver. This is the case in many path-finding algorithms, such as Bellman-Ford. We provide details on how to generalise the messages to be receiver-conditioned in Appendix B.

We start by defining a *kernel* transformation $k : [E, R] \to [E, R]$ that performs some computation over edge features, e.g. in the case of Bellman-Ford, adding the sender distance $d_{s(e)}$ (previously pulled back to the edge via s^*) to the edge weight $w_{s(e)\to t(e)}$. Here we use $[E, R]$ as a shorthand notation for the set of functions $E \to R$. Using the kernel, we can complete the following diagram:

$$\begin{tikzcd} {[V,R]} \arrow[r, "s^*"] & {[E,R]} \arrow[r, "k"] & {[E,R]} \arrow[r, "t_*"] & {[V,\mathbb{N}[R]]} \arrow[r, "\bigoplus"] & {[V,R]} \end{tikzcd}$$

Here $\mathbb{N}[R]$ is the collection of of multisets in R, or "bags of R-values". This object is explained more rigorously in Appendix A.

These four arrows, taken together, form our integral transform: starting with node features, performing computations over edges, finally aggregating the edge messages in the receivers. In doing so, we have *updated* our initial node features (which are elements of $[V, R]$). We now explain each of the four arrows in turn.

s^* is the **pullback** arrow, which takes data defined on V and returns data defined on E. To do this, we precompose the node features with the source function. That is, $(s^* f)(v) := f(s(v))$. Then, the kernel is applied to the resulting edge features, integrating the sender's features with any provided edge features (e.g. edge weights).

After applying our kernel, we have edge messages $m : E \to R$ as a result. We now need to send these messages to the receiver nodes, for which we employ the **pushforward**. We define

$(t_*m)(v) = \sum_{e \in t^{-1}(v)} m(e)$, which we interpret as a formal sum in $\mathbb{N}[R]$. Intuitively, $(t_*m)(v)$ is the "bag of incoming values" at v. The reason why we require multisets is because there may be several edges in t's preimage that store the same value; this also aligns nicely with related work that studies GNN expressive power using multisets (Xu et al., 2018).

Finally, we apply an **aggregator** $\bigoplus$, pointwise for every receiver, to compute updated features for every node. To have such an aggregator, we need one property from R: that it have the structure of a commutative monoid. (see A for details)

Note the generality of our construction: nearly all of the operations at play here rely on the properties of the sets E and V. The only arrow which is constraining the choice of R is the aggregator $\bigoplus$; it necessitates that R be a commutative monoid, which is a sensible choice for both pathfinding algorithms and GNNs. Hence, we have defined a generic *blueprint* where arbitrary Rs can be inserted, hence amplifying the connection between GNNs (where R often includes real-valued vectors) and DP (where R may be, e.g., the "tropical" numbers).

5 Immediate corollaries of the integral transform view

Since we designed the aforementioned integral transform in a way that it can describe both GNNs (Equation 1) and DP (Equation 2), there exist some "embarrassingly obvious" ways of making the architecture of the GNN more algorithmically aligned to the target DP algorithm. Perhaps the clearest one is the final arrow, the aggregator ($\bigoplus$). If we make the GNN's choice of aggregator function match the one used by the target algorithm, this should lead to immediate gains to sample complexity and generalisation.

Indeed, this aligns well with one of the earliest lines of research in algorithmic reasoning: deploying GNNs with aggregators that align to the problem. Already in Veličković et al. (2019), it was demonstrated that using the $\max$ aggregation in GNNs yields superior performance on executing pathfinding algorithms, especially when compared to the more popular choice of $\sum$. Similar findings were observed by Tang et al. (2020); Richter & Wattenhofer (2020), who also clearly indicated environments where $\max$ is the superior aggregator. Lastly, theoretical works such as Xu et al. (2020); Li et al. (2021) have illustrated the additional useful corollaries to extrapolation and noisy-label training when using aligned aggregators.

The fact that already such a vast collection of research works can be unified as analysing one arrow in a categorical diagram is a statement to the potential of our proposed analysis. Any invested effort in analysing GNNs and DP in this manner could provide strong returns when organising existing research and proposing future work.

6 Ideas for future work

What other arrows can be "inspected" in this way? The most natural candidate is the kernel arrow, which corresponds to both the GNN's message function ψ (Equation 1) to the DP scoring function (Equation 2). Aligning the two more closely may require looking deeper into the kernel arrow to formalise entirely, and we leave such analyses for future work to maintain a focused contribution.

While it might seem that the pullback and pushforward arrows are fairly static, this is primarily because both the *graph structure* (in the case of GNNs) and the *expansions* (in the case of DP) are assumed fixed known upfront. This is a simplifying assumption that need not always hold (for example, if our algorithm relies on any kind of *data structure*). Another promising avenue for future work could dive deeper into these two arrows as well, and specifically to develop GNNs that *automatically propose subproblems* within their computations.

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 (Willerton, 2020), and appear in the analysis of Yang-Mills equations in particle physics (Eastwood et al., 1981). 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

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.

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

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

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

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

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.

Jingling Li, Mozhi Zhang, Keyulu Xu, John Dickerson, and Jimmy Ba. How does a neural network's architecture impact its robustness to noisy labels? *Advances in Neural Information Processing Systems*, 34, 2021.

Oliver Richter and Roger Wattenhofer. Normalized attention without probability cage. *arXiv preprint arXiv:2005.09561*, 2020.

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.

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

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

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

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.

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

Simon Willerton. Integral transforms and the pull-push perspective, i. *The n-Category Café*, 2020. URL `https://golem.ph.utexas.edu/category/2010/11/integral_transforms_and_pullpu.html`.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

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.

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.

A THE MULTISET MONAD, COMMUTATIVE MONOIDS AND AGGREGATORS

Given a set S, we define $\mathbb{N}[S] := \{p : S \to \mathbb{N} \mid \#\{p(r) \neq 0\} < \infty\}$, 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 $\mathbb{N}[f] : \mathbb{N}[S] \to \mathbb{N}[T]$, as follows: $\mathbb{N}[f](\sum_{s\in S} n_s s) := \sum_{s \in S} n_s f(s)$, 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 $\texttt{unit} : S \to \mathbb{N}[S]$, sending each element to its indicator function (i.e. an element $x \in S$ to the multiset $\{\!\{x\}\!\}$). The second is $\texttt{join} : \mathbb{N}[\mathbb{N}[S]] \to \mathbb{N}[S]$, which interprets a nested sum as a single sum.

These facts tell us that $\mathbb{N}[-]$ 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* (Wadler, 1990).

It is well-known that the algebras for the monad $\mathbb{N}[-]$ 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 $\bigoplus : \mathbb{N}[R] \to R$ 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:

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

The first diagram explains that the outcome of aggregating a singleton multiset (i.e. the one produced by applying $\texttt{unit}$) 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 $\texttt{join}$ function to collapse the nested multiset, then aggregate the collapsed multiset with $\bigoplus$.

Conveniently, such a $\bigoplus$ is exactly what is required to appropriately connect the outcomes of our previously derived pullback (s^*) and pushforward (t_*) operations into a commutative diagram:

$$\begin{tikzcd}
V \arrow[d, "f"] & E \arrow[l, "s"'] \arrow[r, "t"] \arrow[d, dashed, "s^* f"] & W \arrow[d, dashed, "t_* s^* f"] \\
R & R \arrow[l, "\texttt{id}"] & \mathbb{N}[R] \arrow[l, "\bigoplus"]
\end{tikzcd}$$

As such, so long as R is endowed with a commutative monoid structure, we can use this construction to define a suitable $\bigoplus$ to serve as our aggregation function.

B RECEIVER-DEPENDENT KERNELS

The construction in Section 4 does not immediately support receiver-dependent features, as any feature attached to a non-sender node is lost in the initial pullback step. To get around this issue without losing the spirit of the integral transform, we assume that each $e \in E$ has an opposite $e^* \in E$ (s.t. $s(e) = t(e^*), s(e^*) = t(e)$).

While such a construction is natural for *undirected* graphs, it is possible to make it work while having directionality constraints. Specifically, the kernel can mask these extra edges as an implementation detail (e.g. attaching an edge weight of "$+\infty$" in the context of pathfinding), so this construction does not lose generality.

Now, we can express receiver dependence by factorising the kernel arrow as follows:

$$[E,R] \xrightarrow{\texttt{(id,*)}} [E,R]\times[E,R] \xrightarrow{k'} [E,R]$$

The first arrow sends $g(e)$ to $g(e, e^*)$. This gives our k' access to both sender and receiver features. Again, we can perform any necessary receiver masking within k' to avoid contributions from "virtual" edges.

In general, more complex kernels may require additionally enlarging E by a constant factor. For example, if E is given an additional symmetry $E \to E$ we could use this to express ternary kernels, and so on.