The CLRS Algorithmic Reasoning Benchmark

Petar Veličković [1] **Adrià Puigdomènech Badia** [1] **David Budden** [1]
Razvan Pascanu [1] **Andrea Banino** [1] **Misha Dashevskiy** [1] **Raia Hadsell** [1] **Charles Blundell** [1]

Abstract

Learning representations of algorithms is an emerging area of machine learning, seeking to bridge concepts from neural networks with classical algorithms. Several important works have investigated whether neural networks can effectively reason like algorithms, typically by learning to execute them. The common trend in the area, however, is to generate targeted kinds of algorithmic data to evaluate specific hypotheses, making results hard to transfer across publications, and increasing the barrier of entry. To consolidate progress and work towards unified evaluation, we propose the CLRS Algorithmic Reasoning Benchmark, covering classical algorithms from the Introduction to Algorithms textbook. Our benchmark spans a variety of algorithmic reasoning procedures, including sorting, searching, dynamic programming, graph algorithms, string algorithms and geometric algorithms. We perform extensive experiments to demonstrate how several popular algorithmic reasoning baselines perform on these tasks, and consequently, highlight links to several open challenges. Our library is readily available at `https://github.com/deepmind/clrs`.

1. Introduction

Neural networks and classical algorithms are two techniques that operate on diametrically opposite (and complementary) sides of problem-solving: neural networks can adapt and generalise to raw inputs, automatically extracting appropriate features and a single neural network setup is often applicable to many separate tasks (Zamir et al., 2018). However, they are hard to interpret, notoriously unreliable when extrapolating outside of the dataset they have been trained on, and rely on massive quantities of training data. On the other hand, algorithms trivially strongly generalise to inputs of arbitrary sizes, and can be verified or proven to be correct, with interpretable step-wise operations. Their shortcoming is that inputs must be made to conform to a particular algorithm specification, and looking at a separate task often requires coming up with an entirely new algorithm (Veličković & Blundell, 2021).

Bringing the two sides closer together can therefore yield the kinds of improvements to performance, generalisation and interpretability that are unlikely to occur through architectural gains alone. Accordingly, algorithmic modelling as a domain for testing neural networks has been gaining popularity over the last few years (Zaremba & Sutskever, 2014; Kaiser & Sutskever, 2015; Trask et al., 2018; Vinyals et al., 2015; Kool et al., 2018; Freivalds et al., 2019; Dwivedi et al., 2020; Chen et al., 2020; Tang et al., 2020; Veličković et al., 2019; Yan et al., 2020; Deac et al., 2020) due to its ability to highlight various reasoning limitations of existing architectures.

Earlier work (Zaremba & Sutskever, 2014; Kaiser & Sutskever, 2015) focused on the need of long-term memory capabilities when executing algorithms, which offered a good test-bed for various recurrent and memory architectures. Recently, algorithmic tasks have been used to highlight the efficiency of graph neural networks (Dwivedi et al., 2020; Chen et al., 2020; Veličković et al., 2019; Yan et al., 2020; Corso et al., 2020; Tang et al., 2020; Georgiev & Lió, 2020; Veličković et al., 2020) and to distinguish between different variations of them, typically through the lens of *algorithmic alignment*—architectures that align better with the underlying algorithm can be proven to have better sample complexity (Xu et al., 2019). Unfortunately, many of these works remain disconnected in terms of the algorithms they target, how the data is presented to the model or through the training and testing protocols they use, making direct comparison somewhat difficult.

To make a first step towards a unified benchmark for algorithmic reasoning tasks, we propose a comprehensive dataset which we will refer to as *The CLRS Algorithmic Reasoning Benchmark*, in homage to the *Introduction to Algorithms* textbook by Cormen, Leiserson, Rivest and Stein (Cormen et al., 2009).

[1] DeepMind. Correspondence to: Petar Veličković <petarv@deepmind.com>.

Proceedings of the 39^{th} International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022. Copyright 2022 by the author(s).

Within this benchmark, we propose and evaluate on **CLRS-30**: a dataset containing trajectories—a trajectory is formed of inputs, the corresponding outputs and optional intermediary targets—of 30 classical algorithms covering various forms of reasoning, including sorting, searching, dynamic programming, geometry, graphs and strings. Some of these algorithms are depicted in Figure 1. The appeal and motivation for such a benchmark goes beyond unifying or providing a common ground for previous works, as we will describe. We believe that CLRS-30 is well positioned to explore *out-of-distribution* (OOD) generalization and transfer (as potentially part of a meta-learning setting) given the explicit and known relationship between different algorithms (e.g. what subroutines are shared and so forth).

2. Motivation

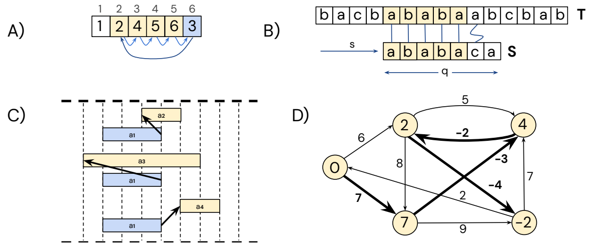


Figure 1. Example of four algorithms within CLRS-30. A) insertion sort; B) string matching; C) greedy task scheduling; D) shortest paths.

Timely posed benchmarks have led to a significant progress in the field, from the impact of ImageNet (Russakovsky et al., 2015) on the vision community, to that of Wikipedia and Penn Treebank in popularizing neural networks for language modelling (Merity et al., 2016; Mikolov et al., 2011) or Atari-2600 for deep reinforcement learning (Bellemare et al., 2013). The prevalence of recent works focusing on algorithmic reasoning[1], as well as a history of disparate work on a variety of bespoke benchmarks (Graves et al., 2014; Zaremba & Sutskever, 2014; Kaiser & Sutskever, 2015; Trask et al., 2018), suggests significant utility in a benchmark covering a wide-range of classical CS algorithms.

Learning to mimic an algorithm also provides an opportunity to extensively test the limitations of architectures both in terms of their representation capacity and processing. This can then be related back directly onto underlying operations and qualities of the well-studied CS algorithms being mimicked as we are aware of both the process used to generate the inputs and the specifics of the underlying function producing the corresponding outputs. Hence, benchmarking in this area can be used to better understand the limitations of current architectures and the optimisation schemes used. This benchmarking can come in many forms:

Data can be easily generated, allowing the neural network behaviour to be probed under different regimes: from few-shot learning all the way to infinite-data.

Algorithms can be used to understand the efficiency of different inductive biases and neural components. For example, a recent study (Tang et al., 2020) has demonstrated the direct benefits of choosing inductive biases that align well with iterative algorithms. Algorithms have also been used to highlight the importance of attention mechanisms (Graves et al., 2014) or to disambiguate various message passing mechanisms for graph neural networks (Richter & Wattenhofer, 2020; Joshi et al., 2020; Veličković et al., 2019).

Algorithms can require repeated computation, recursion, or performing very different forms of computations conditioned on the input, providing an excellent test-bed for evaluating compositionality; i.e. whether an algorithm executor can effectively exploit these repeated computations.

One can control the amount of memory required to solve a problem instance, hence test the memorization ability of neural networks. Moreover, one can build a curriculum of tasks of increasing memory requirements (Zaremba & Sutskever, 2014).

Control over the difficulty of problem instances also allows the behaviour of a trained model to be tested on OOD samples. While neural networks are highly efficient on solving complex perceptual tasks, current theoretical understanding suggests that their power relies on their ability to *interpolate* (Liu et al., 2020; Belkin et al., 2019; Jacot et al., 2018), limiting them to *in-distribution* generalisation. General reasoning systems, however, need to be able to expand beyond this type of generalization. *OOD generalization* (Li et al., 2020) is paramount, as generally one can not control the distribution a model will face over time when deployed.

Understanding how algorithms operate on corner cases is a standard approach for analysing their correctness. Similarly, understanding the behaviour of a trained model on larger instances of the problem, or instances that expose such corner cases that were not covered in the training set, can elucidate to what degree the model has truly learned the *algorithm* (as opposed to overfitting to specific statistics of the training data). Particularly, we can control how far from the training distribution a test instance is, potentially allowing us to understand to what extent the model generalizes OOD, and under which circumstances. In turn, this can offer insight into the effectiveness of different inductive biases, highlighting what kinds of inductive biases are useful for mimicking reasoning processes.

[1] Concurrent works published at the same venue include: (Xu et al., 2019; Veličković et al., 2019) at ICLR'20 and (Veličković et al., 2020; Corso et al., 2020; Tang et al., 2020) at NeurIPS'20.

One would also expect a general reasoning system to be able to *reuse* parts of learned computations when learning a new task, and to *compose* learnt computational subroutines (Lake, 2019; Griffiths et al., 2019; Alet et al., 2018). These forms of generalization have been the aim of several learning paradigms from transfer learning to meta-learning and continual learning or domain adaptation. However, many of these paradigms rely on the concept of a *task*, and measuring or understanding the ability of a learned system to *reuse* or *compose* requires the ability to decompose a task into sub-tasks and to be able to relate tasks among themselves. In many scenarios, such decompositions are ambiguous. Without a clear segmentation into sub-tasks, there can be no clearly defined distance metric between tasks (Du et al., 2018). Conversely, algorithms are built based on subroutines that tend to be extensively shared, providing a good playground for formalizing and measuring *reuse* and *composition*, making an algorithmic reasoning benchmark potentially attractive to meta-learning practitioners.

Lastly and fundamentally, computer scientists rely on a relatively small[2] number of algorithms to address an extremely vast set of problems. They can be seen as a very powerful basis that spans most forms of reasoning processes. On one hand, this means that any generic reasoning system likely has to be able to reproduce all such kinds of procedures, hence, building a system that properly learns all of them is a major stepping stone towards generic reasoning. On the other hand, this means that they can be used to discover inductive biases that will enable tackling more complex problems. This is either because these complex problems can be seen as a combination of several algorithms, or because learning certain algorithms can provide a reliable way for the model to learn how to access its own memory or how to attend to its input or other such internal mechanisms. So by first training on algorithms—potentially controlling the difficulty of training instances—one can pre-train for tasks where full trajectories may not be available (Veličković et al., 2021). One such example is discovering novel polynomial-time heuristics for combinatorial optimisation (Bengio et al., 2020; Cappart et al., 2021; Khalil et al., 2017) or reinforcement learning (Deac et al., 2021). Note that our focus with this benchmark lies in learning the basic algorithms themselves only–this in itself proves sufficiently challenging for neural networks, and is itself a useful outcome for the reasons highlighted above. However, we speculate that once a neural network can learn not only individual algorithms but novel combinations of multiple algorithms or even discover new algorithms, such networks will be useful in a wide variety of problems from scientific problems such as protein folding and genomics to simulated environments such as those used by reinforcement learning and control–much as classic CS algorithms already make in-roads into these domains but lack the ability to learn from data.

Guided by these observations, we regard CLRS-30 as a first step towards a pragmatic setting to test many of these different aspects of current architectures. While we do not directly target all of the scenarios outlined above, the benchmark was built with ease of expansion in mind; enabling for extensive tweaking of training/testing setups, kinds of information captured in algorithm trajectories, as well as including additional algorithms, which we aim to do consistently over time.

3. CLRS Algorithmic Reasoning Benchmark

Owing to its name, CLRS-30 consists only of algorithms which may be encountered in the CLRS textbook (Cormen et al., 2009). Further, all algorithm trajectories and relevant variables have been designed to match the pseudocode in the textbook as closely as possible. We begin by describing the selection criteria we applied when determining which algorithms to include in CLRS-30.

Our initial survey of the textbook yielded 94 algorithms and data structures of interest. From this point, we set out to filter this set to algorithms suitable for inclusion in the initial version of our benchmark. The criteria we applied, with justification and remarks, are as follows:

We want to be able to reliably generate *ground-truth* outputs for large inputs. As such, NP-hard tasks (and approximation algorithms thereof) have been excluded. Our decision is backed up by theoretical work suggesting impossibility of accurately modelling NP-hard problems using polynomial-time samplers, unless NP=co-NP (Yehuda et al., 2020).

Tasks requiring *numerical outputs* have been excluded. Evaluating their performance is ambiguous, and may be dependent on the way architectures choose to represent numbers. For example, Yan et al. (2020) (which represents numbers in binary) and Veličković et al. (2019) (which represents them in floating-point) report different metrics on predicting shortest-path lengths. This excludes most number-theoretic algorithms, linear programming, and max-flow[3]. It does *not* exclude shortest-path algorithms: we can treat them as tasks of finding edges belonging to the shortest path, as was done in Veličković et al. (2019); Tang et al. (2020). The numerical values of path lengths are then treated as intermediate parts of the trajectory, and not directly evaluated on.

Standalone *data structures* do not directly represent a task[4].

[2] The entire Introduction to Algorithms textbook (Cormen et al., 2009) proposes and discusses ∼100 algorithms in total.

[3] It should be noted that, by the max-flow min-cut theorem (Ford Jr & Fulkerson, 2015), any max-flow problem can be cast as finding the minimum cut containing the source vertex. This is a discrete decision problem over input vertices, which hence doesn't violate our constraints, and could be included in future iterations.

[4] In programming language terms, their algorithms tend to be

Rather, their target is appropriately updating the internal state of the data structure. Hence, we don't include their operations, unless they appear as components of algorithms. We, of course, look forward to including them in subsequent versions of the dataset, as they can provide useful building blocks for learning complex algorithms.

Lastly, there are representational issues associated with dynamically allocated memory—it may be unclear what is the best way to represent the internal memory storage and its usage in algorithm trajectories. One example of the ambiguity is in asking whether the algorithm executor should start with a "scratch space" defined by the space complexity of the problem that gets filled up, or dynamically generate such space[5] (Strathmann et al., 2021). As such, we for now exclude all algorithms that require allocating memory which cannot be directly attached to the set of objects provided at input time. This excludes algorithms like merge sort, Hierholzer's algorithm for finding Euler tours (Hierholzer & Wiener, 1873), or string matching using finite automata.

All of the above applied, we arrive at the 30 algorithms that are selected into CLRS-30, which we categorize as follows:

Sorting: Insertion sort, bubble sort, heapsort (Williams, 1964), quicksort (Hoare, 1962).

Searching: Minimum, binary search, quickselect (Hoare, 1961).

Divide and Conquer (D&C): Maximum subarray (Kadane's variant (Bentley, 1984)).

Greedy: Activity selection (Gavril, 1972), task scheduling (Lawler, 1985).

Dynamic Programming: Matrix chain multiplication, longest common subsequence, optimal binary search tree (Aho et al., 1974).

Graphs: Depth-first and breadth-first search (Moore, 1959), topological sorting (Knuth, 1973), articulation points, bridges, Kosaraju's strongly-connected components algorithm (Aho et al., 1974), Kruskal's and Prim's algorithms for minimum spanning trees (Kruskal, 1956; Prim, 1957), Bellman-Ford and Dijkstra's algorithms for single-source shortest paths (Bellman, 1958; Dijkstra et al., 1959) (+ directed acyclic graphs version), Floyd-Warshall algorithm for all-pairs shortest paths (Floyd, 1962).

Strings: Naïve string matching, Knuth-Morris-Pratt (KMP) string matcher (Knuth et al., 1977).

Geometry: Segment intersection, Convex hull algorithms: Graham scan (Graham, 1972), Jarvis' march (Jarvis, 1973).

The chosen algorithms span a wide variety of reasoning procedures, and hence can serve as a good basis for algorithmic reasoning evaluation, as well as extrapolation to more challenging problems.

3.1. Implementation, probes and representation

We have implemented the selected 30 algorithms in an idiomatic way, which aligns as closely as possible to the original pseudocode from Cormen et al. (2009). This allows us to automatically generate input/output pairs for all of them, enabling full control over the input data distribution, so long as it conforms to the preconditions of the algorithm. Further, we capture the intermediate algorithm trajectory in the form of **"hints"** (detailed in section 3.2), which allow insight into the inner workings of the algorithm. Such trajectories have already been extensively used in related work (Veličković et al., 2019; 2020; Georgiev & Lió, 2020; Deac et al., 2020) and are typically crucial for OOD generalisation.

In the most generic sense, algorithms can be seen as manipulating sets of objects, along with any relations between them (which can themselves be decomposed into binary relations). If the sets are (partially) ordered (e.g. arrays or rooted trees), this can be imposed by including predecessor links. Therefore, algorithms generally operate over *graphs*. Motivated by existing theoretical results showing that graph neural networks align well with dynamic programming-style computations (Xu et al., 2019; Dudzik & Veličković, 2022), we propose a graph-oriented way to encode the data.

Generally, our data is represented as a set of n vertices[6], where n is a hyperparameter that is provided as part of the dataset generation process.

When the semantics of these nodes are not immediately clear from the task (e.g. graph algorithms naturally operate over a graph of n nodes), we make an appropriate modification to derive nodes. For example, in sorting algorithms, we treat every input list element as a separate node, and in string matching, we treat each character of the two input strings as a separate node.

All information over these graphs falls under the following categorisation:

Stage: Every feature, i.e. observation in the trajectory, is either part of the *input*, *output*, or the *hints*. As we do not cover algorithms that perform on-line querying, for all 30 algorithms there will be exactly one snapshot of the input and output values, whereas hints will be a time-series of intermediate algorithm states.

Location: Every feature is either present within the *nodes*, *edges* (pairs of nodes) or the *graph*[7].

of the `void` type.

[5] Akin to `malloc`-like calls in C++.

[6] Edges are only present to represent the predecessor vertex if the input is a partially ordered.

[7] This also determines shapes of each feature, e.g. node features

Type: Every feature can be of five possible types, which can determine the appropriate method for encoding/decoding it, and the appropriate loss function to use when learning to predict it:

- `scalar`: Floating-point scalar[8] feature. This would typically be fit using mean-squared error.
- `categorical`: Categorical feature over K possible classes. The type corresponds typically to cross-entropy loss over the classes.
- `mask`: Categorical feature over two classes. This can be fit using binary cross-entropy.
- `mask_one`: Categorical feature over two classes, where exactly one node is active ("one-hot"). One would generally optimise this argmax operation using categorical cross-entropy.
- `pointer`: Categorical feature over the n nodes. To predict "similarity" score against every node, and typically optimised using categorical cross entropy (as introduced in Pointer Graph Networks (PGN) (Veličković et al., 2020)).

Specifying a feature's stage, location and type fully determines its role in the dataflow. A tuple `(stage, loc, type, values)` is referred to as a *probe*. Each of the 30 algorithms has a static (w.r.t. stage, location and type) set of probes, which are considered to be a *spec* for the algorithm. We will later describe how these specs may be used to construct baseline architectures for the benchmark.

Every node is always endowed with a *position* scalar input probe, which uniquely indexes it—the values are linearly spaced between 0 and 1 along the node index. This allows not only representing the data sequentially (when this is appropriate), but also serves as a useful tie-breaker when algorithms could make an arbitrary choice on which node to explore next—we force the algorithms to favour nodes with *smaller* position values.

To illustrate these concepts further, at the end of this section we will describe the probes in detail for a popular algorithm (insertion sort).

Note that, while we format the data in a way that clearly favours graph neural network executors, it can be easily adapted for different types of neural architectures; for example, sequence to sequence models (Sutskever et al., 2014). Overall, CLRS-30 requires $\sim 1\text{h}$ to generate, and occupies $\sim 4.5\text{GB}$ when uncompressed, across all 30 tasks.

3.2. Hints

Hints are an important component of our benchmark, which we find fundamental in order to make progress on algorithmic reasoning. As we previously argued, the advantage of algorithms as a task is our understanding of their behaviour, and our ability to decompose them into useful subroutines that can be shared or repeatedly applied.

While, implicitly, we hope that such a decomposition would happen in any learned system, even when trained just using inputs and outputs (as studied in Xu et al. (2019)), the degree to which we can measure or encourage this is limited in the typical end-to-end learning process, and often most of the generalisation happens only in-distribution (as observed by Veličković et al. (2019); Xu et al. (2020); Bevilacqua et al. (2021)). The underlying algorithm may not be statistically identifiable from a small set of input/output pairs.

Conversely, a perfect decomposition of a task into small subtasks can be generated for algorithmic problems. Then, individual models for each subtask may be trained and recomposed into a solution. Such an approach will, by construction, provide strong decompositional benefits: as studied by Yan et al. (2020), perfect OOD generalisation can be observed with such models, and they can even generalise zero-shot to test algorithms that reuse their modules. However, the downstream applicability of this is potentially limited; when faced with a novel task which cannot be easily decomposed into subtasks, it can be hard to decide how to reuse the learnt modules.

We believe hints to lie in-between these two approaches. On one hand, they represent intermediate targets which the network should be able to predict if it performs reasoning similar[9] to the ground truth algorithm it is supposed to mimic. Indeed, several lines of recent work (Veličković et al., 2019; Georgiev & Lió, 2020; Veličković et al., 2020; Deac et al., 2020) make favourable conclusions about using them, when it comes to achieving stronger OOD generalisation. Furthermore, models leveraging hints are still end-to-end models; when faced with a novel task at test-time, we don't need explicit knowledge of that task's hints in order to re-use the weights learnt on a task which had them.

Algorithms specify *one* way of attacking a problem, that is explicitly detailed through the hints. In this sense, insertion sort (to be presented shortly) is one way of implementing

are of shape $n \times f$; edge features are of shape $n \times n \times f$; graph features are of shape f, where f is the dimension of this feature (excluding batch axis).

[8] Given our current restriction on numerical predictions, scalar types will never be given in the output stage.

[9] Note that architectures supervised in this way usually don't model the hints perfectly, and will deviate from the target algorithm in subtle ways—Veličković et al. (2020) perform a qualitative study which shows GPU-specialised data structures could emerge as a result of such setups.

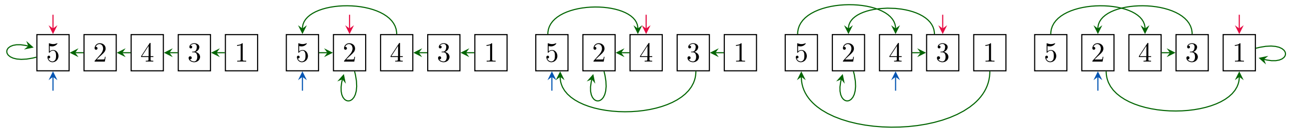


Figure 2. A sequence of hints for insertion sorting a list $[5, 2, 4, 3, 1]$. **Green** pointers correspond to the predecessor pointers (specifying the list's state throughout the algorithm's execution. Note how the head of the list always points to itself, by convention. Further, note how, at every step, the list is rewired such that the node selected by the **blue** pointer (slot) will point to the current iterator (pointed in **red**).

a *sorting* function: **all** sorting algorithms model sorting functions, and will hence have identical outputs for identical inputs. The aspects that set the different sorting algorithms apart are exposed through their hints.

Being mindful of the fact that neural networks commonly run on parallelisable architectures, we have made efforts to "compress" the hints as much as possible. For example, if a single `for` loop is used to sweep the data and detect the node which optimises a certain quantity (without doing any order-sensitive computations), that `for` loop can typically be entirely "skipped" when recording hints: as parallel architectures may typically examine all the nodes at once. Further, we make every effort possible that the hint at step $t + 1$ will be predictable from the hints at step t by using only a single step of message passing.

3.3. Worked example: insertion sort

To illustrate all of the concepts outlined above, we observe the trajectories extracted by our data collection procedure on an example: *insertion sorting* the array $[5, 2, 4, 3, 1]$.

Insertion sort uses one pointer (j) to scan through the array, and then another pointer (i) to slot the j-th item into the correct place within $[0..j]$. This ascertains the invariant that, after k steps, the subarray of the first k elements is completely sorted. Hence the trajectory (with i and j marked) is: $[5_{i,j}, 2, 4, 3, 1] \rightarrow [2_i, 5_j, 4, 3, 1] \rightarrow [2, 4_i, 5_j, 3, 1] \rightarrow [2, 3_i, 4, 5_j, 1] \rightarrow [1_i, 2, 3, 4, 5_j]$. Here, at each step, j scans along the array, and i indicates the correct place for the element that was j-th at the start of each iteration.

Converting this trajectory into a graph representation requires some considerations. Requiring the model to perform explicit swapping of node values would, ultimately, require numerical predictions. To avoid it, we ask the model to predict the *predecessor pointer* of each node (by convention, the head of the array points to itself). Hence the actual recorded trajectory can be realised as depicted in Figure 2. In this figure, **green** pointers correspond to the predecessor pointers, **red** ones point to j, and **blue** ones point to i. i and j are realised as type `mask_one`, whereas predecessors are of type `pointer`—and all three are stored in the nodes. The **red** and **blue** pointers represent the "hints" for this task.

Finally, note that the original insertion sort pseudocode mandates that, at each iteration, i starts at position j and shifts backward until the right position is found. However, this procedure can be performed in one step by a GNN, as it can locate the correct position by examining all relevant positions, and we can omit all of those intermediate steps.

In order to further illustrate how these hints are collected, we also provide an informal pseudocode for collecting hints for insertion sort in Algorithm 1:

```
Algorithm 1 Hint updates for Insertion Sort
Input : Input array val, Positions pos
Hints : Predecessors pred, Iterator iter, swap slot slot

$\texttt{pred}[\texttt{i}] \leftarrow \begin{cases} 0 & \texttt{i} = 0 \\ \texttt{i} - 1 & \texttt{i} > 0 \end{cases}$ ;    // Initialise list
$\texttt{slot} \leftarrow 0, \texttt{iter} \leftarrow 0$
while $\texttt{iter} < n$ do
    $\texttt{iter} \leftarrow \texttt{iter} + 1$
    $\texttt{max\_node} \leftarrow \operatorname*{argmax}_{\texttt{j}\,:\,\texttt{pos}[\texttt{j}] < \texttt{pos}[\texttt{iter}]} \texttt{val}[\texttt{j}]$
    if $\texttt{val}[\texttt{max\_node}] < \texttt{val}[\texttt{iter}]$ then
        $\texttt{slot} \leftarrow \texttt{max\_node}$
        $\texttt{pred}[\texttt{i}] \leftarrow \begin{cases} \texttt{slot} & \texttt{i} = \texttt{iter} \\ \texttt{pred}[\texttt{i}] & \text{otherwise} \end{cases}$
    else
        $\texttt{slot} \leftarrow \operatorname*{argmin}_{\texttt{j}\,:\,\texttt{pos}[\texttt{j}] < \texttt{pos}[\texttt{iter}], \texttt{val}[\texttt{j}] \geq \texttt{val}[\texttt{iter}]} \texttt{val}[\texttt{j}]$
        $\texttt{pred}[\texttt{i}] \leftarrow \begin{cases} \texttt{iter} & \texttt{i} = \texttt{slot} \\ \texttt{iter} & \texttt{i}{=}\texttt{iter} \wedge \texttt{pred}[\texttt{slot}]{=}\texttt{slot} \\ \texttt{pred}[\texttt{slot}] & \texttt{i}{=}\texttt{iter} \wedge \texttt{pred}[\texttt{slot}]{\neq}\texttt{slot} \\ \texttt{max\_node} & \texttt{pred}[\texttt{i}] = \texttt{iter} \\ \texttt{pred}[\texttt{i}] & \text{otherwise} \end{cases}$
    end
end
return pred ;                                  // Return final list
```

In the interest of illustrating the hint structures further, we provide worked examples of trajectories for three more algorithms (dynamic programming, path-finding and string matching) in Appendix B. It should be remarked that we directly expose all of the hint collection routines as Python code inside the CLRS library, allowing for direct inspection.

4. Empirical evaluation

Having surveyed the specifics of CLRS-30, we now present experimental results on it for several proposed algorithmic reasoning models. We primarily investigate whether a natural *ladder* of model performance will emerge when extrapolating to larger inputs. Beyond this, we believe the benchmark will be useful for empirically examining many other properties of algorithmic models, such as evaluating generalisation across different graph types, task types, or various multi-task (Xhonneux et al., 2021) or continual learning setups. We make available complete implementations of our data generating, probing and model training subroutines, which should make evaluating on such settings simple to deploy[10]. We survey several key ways of interacting with the benchmark (e.g. implementing baselines, modifying datasets, adding new algorithms) in Appendix A.

4.1. Baseline models

Encode-process-decode For our experimental validation, we adopt the encode-process-decode paradigm of Hamrick et al. (2018), which is a common direction for several hint-based architectures (Veličković et al., 2019; Georgiev & Lió, 2020; Veličković et al., 2020; Deac et al., 2020).

Namely, we consider a setup with inputs $\mathbf{x}_i$ in nodes, $\mathbf{e}_{ij}$ in edges, and $\mathbf{g}$ in the graph. We first encode each of these using linear layers f_n, f_e, f_g, to obtain encodings

$$\mathbf{h}_i = f_n(\mathbf{x}_i) \qquad \mathbf{h}_{ij} = f_e(\mathbf{e}_{ij}) \qquad \mathbf{h}_g = f_g(\mathbf{g}) \tag{1}$$

We then feed these latents through a *processor network* to perform one step of computation. As we are focusing on graph representation learning in the current data format, most of our processors will be realised as graph neural networks (Gilmer et al., 2017). Most generally, along every edge (i, j), a *message* from node i to node j, $\mathbf{m}_{ij}$ is computed (using a message function f_m), and these messages are then aggregated across all neighbouring nodes using a permutation-invariant aggregation function, $\bigoplus$. Finally, a *readout network* f_r transforms these aggregated messages and the node encodings into processed node encodings:

$$\mathbf{m}_{ij} = f_m(\mathbf{h}_i, \mathbf{h}_j, \mathbf{h}_{ij}, \mathbf{h}_g) \tag{2}$$

$$\mathbf{m}_i = \bigoplus_{i \in \mathcal{N}_j} \mathbf{m}_{ji} \qquad \mathbf{h}'_i = f_r(\mathbf{h}_i, \mathbf{m}_i) \tag{3}$$

Once node encodings are updated, we can *decode* them to make various predictions for this step of reasoning, depending on the type of the prediction required (using relevant decoder functions $g_\cdot$), as prescribed in Section 3.1. Further, we keep track of previous-step node encodings $\mathbf{h}_i^{(t-1)}$, to explicitly use in a recurrent cell update (exactly as done by Veličković et al. (2019)). We opt to provide this recurrent update in order to provide long-range capacity to the model.

Lastly, we need to decide in what capacity will hints be used. We provide results for the option where hints are both decoded (used for computing the loss function) and encoded (considered as part of $\mathbf{x}$, $\mathbf{e}_{ij}$ and $\mathbf{g}$). At testing time, the encoded hint is equal to the hints decoded by the previous step, whereas we can stabilise these trajectories at training time by performing *noisy teacher forcing*—inspired by Noisy Nodes (Godwin et al., 2021), at each step we feed back ground-truth hints with probability 0.5. The quantity of hints is still used to determine the number of processor steps to perform at evaluation time. This requirement of knowing the hint-size can be lifted by, e.g., using termination networks (Veličković et al., 2019; Banino et al., 2021) or aligning to iterative algorithms (Tang et al., 2020).

Processor networks The only remaining component to specify is the *processor network* used by our models. As this component carries the most computational load, it is also the most obvious module to sweep over. We provide all implementations and hyperparameters within our codebase.

Unless otherwise specified, we assume *fully-connected graphs*, i.e. $\mathcal{N}_i = \{1, 2, \dots, n\}$, hence every node is connected to every other node. We consider the following baseline processor networks:

Deep Sets (Zaheer et al., 2017); where each node is only connected to itself: $\mathcal{N}_i = \{i\}$ (i.e., choice of $\bigoplus$ is irrelevant). Such a model is popular for summary statistic tasks.

Graph Attention Networks (Veličković et al., 2017), where the aggregation function $\bigoplus$ is self-attention (Vaswani et al., 2017), and the message function f_m merely extracts the sender features: $f_m(\mathbf{h}_i, \mathbf{h}_j, \mathbf{h}_{ij}, \mathbf{h}_g) = \mathbf{W}\mathbf{h}_i$. We report the best performance across GAT (Veličković et al., 2017) and GATv2 (Brody et al., 2021) attention mechanisms.

Message-passing Neural Networks (Gilmer et al., 2017), which correspond exactly to the formulation in Equation 2, with $\bigoplus = \max$, as prescribed by previous work (Veličković et al., 2019). As a sanity check, we also attempted $\bigoplus = \sum$ finding it underperformed on all tasks compared to $\max$.

Pointer Graph Networks (Veličković et al., 2020), which use only graph neighbourhoods $\mathcal{N}_i$ specified by a union of all node `pointer` and edge `mask` hints, and $\bigoplus = \max$. This restricts the model to only reason over the edges deemed important by the inputs and hints.

Memory Networks (Sukhbaatar et al., 2015) have been

[10] https://github.com/deepmind/clrs

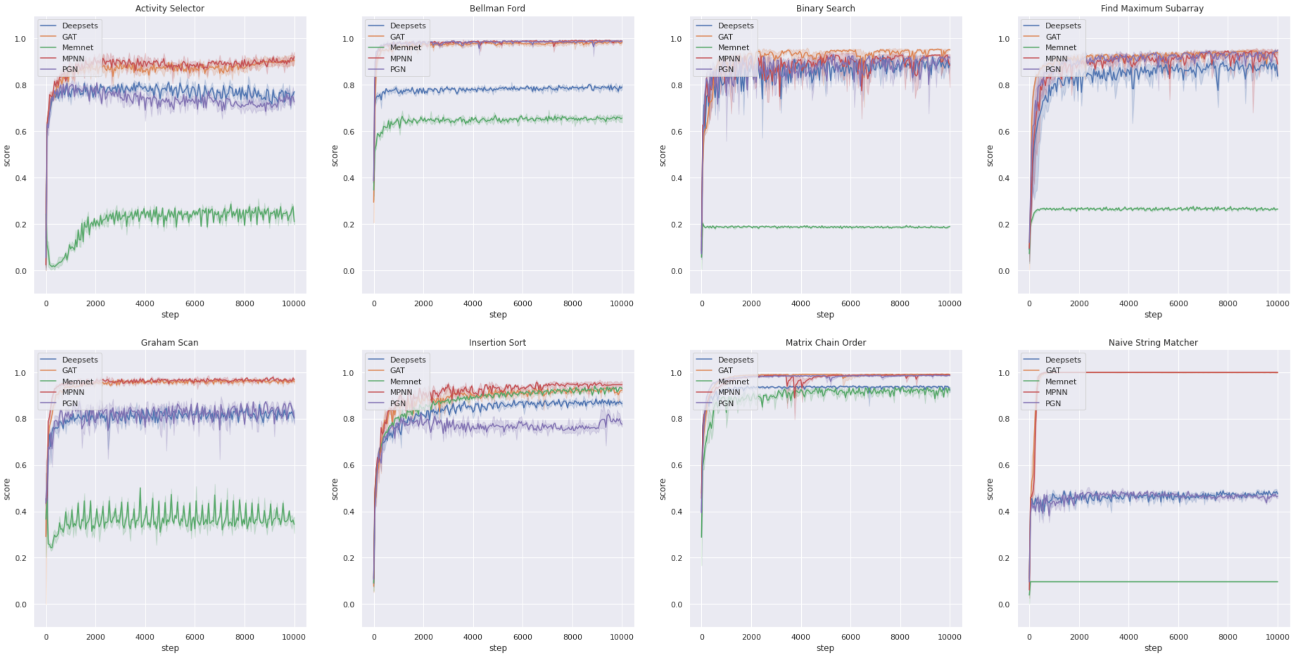


Figure 3. Validation results on eight representative algorithms in CLRS-30 (activity selector, Bellman-Ford, binary search, find maximum subarray, Graham scan, insertion sort, matrix chain order, naïve string matcher), averaged over three seeds. In all cases the y-axis is between $[0, 100]\%$. Legend: MPNN **red**, PGN **purple**, Deep Sets **blue**, GAT **orange**, Memory Networks **green**. Validation results for all 30 individual algorithms can be found in Appendix D.

used in the past as baseline for investigating reasoning in neural networks (e.g. Banino et al., 2020), as they provide an alternative way to use structural dependencies in a graph by treating edges as memories and nodes as queries. Here we used latents representing node features $\mathbf{h}_i$ as queries and latents representing edge features $\mathbf{h}_{ij}$ (where there is a connecting edge and $\mathbf{0}$ otherwise) as memory inputs.

4.2. Dataset statistics

For each algorithm in CLRS-30, we provide a canonical set of training, validation and test trajectories for benchmarking in- and out-of-distribution generalisation. We obtain these trajectories by running the algorithms on randomly sampled inputs that conform to their input specification. This implies, e.g., that the inputs to most graph algorithms are Erdős-Rényi graphs (Erdös & Rényi, 2011) with a certain edge probability. All scalar inputs are sampled from $U(0, 1)$.

For validation, our aim is to measure in-distribution generalisation. Hence we sample inputs of 16 nodes for both, and generate 1,000 trajectories for training and 32 for validation. For testing, we measure out-of-distribution generalisation, and sample 32 trajectories for inputs of 64 nodes. For algorithms where the output is on the graph stage (rather than node/edge), we generate 64× more trajectories, in order to equalise the number of targets across tasks.

We optimise our models on the training trajectories in a teacher-forced fashion, with a batch size of 32, using the Adam optimiser (Kingma & Ba, 2014) with an initial learning rate of $\eta = 0.001$. We train for $10,000$ steps, early stopping on the validation performance. Our models are trained on one V100 Volta GPU, requiring roughly between 1h and 30h to train, depending on the algorithm's time complexity. For example, linear-time algorithms have significantly fewer hints—hence message passing steps—than cubic-time ones.

4.3. Validation (in-distribution) performance

We provide the in-distribution performance throughout training in Figure 3, for eight representative tasks in CLRS-30 (one per each algorithm type); see Appendix D for the full results on all 30 algorithms. In this regime, the MPNN appears to dominate for most tasks: achieving over 90% F_1 score for nearly all of them.

While this might seem like strong evidence in favour of the fully-connected MPNNs, their added degrees of freedom may also make MPNNs more prone to overfitting to specifics of the input (e.g. the input graphs' sizes), rather than truly learning the underlying reasoning rule. We present the out-of-distribution results next, in order to make this distinction clear.

Table 1. Average test micro-F_1 score of all models on all algorithm classes. The full test results for all 30 algorithms, along with a breakdown of the "win/tie/loss" metric, are given in Appendix C.

Algorithm	**Deep Sets**	**GAT**	**Memnet**	**MPNN**	**PGN**
Divide & Conquer	12.48% ± 0.67	24.43% ± 0.74	13.05% ± 0.00	20.30% ± 0.85	**65.23**% ± 4.44
Dynamic Prog.	66.05% ± 7.79	67.19% ± 5.33	67.94% ± 7.75	65.10% ± 6.44	**70.58**% ± 6.48
Geometry	64.08% ± 6.60	**73.27**% ± 11.18	45.14% ± 11.65	73.11% ± 17.19	61.19% ± 7.01
Graphs	37.65% ± 8.09	46.80% ± 8.66	24.12% ± 5.20	**62.79**% ± 8.75	60.25% ± 8.42
Greedy	75.47% ± 6.81	78.96% ± 4.59	53.42% ± 20.73	**82.39**% ± 3.01	75.84% ± 6.59
Search	43.79% ± 18.29	37.35% ± 19.81	34.35% ± 21.67	41.20% ± 19.87	**56.11**% ± 21.56
Sorting	39.60% ± 7.19	14.35% ± 4.64	**71.53**% ± 1.09	11.83% ± 2.78	15.45% ± 8.46
Strings	2.64% ± 0.68	3.02% ± 1.08	1.51% ± 0.21	**3.21**% ± 0.94	2.04% ± 0.20
Overall average	42.72%	43.17%	38.88%	44.99%	**50.84**%
Win/Tie/Loss counts	0/3/27	1/5/24	4/2/24	8/3/19	**8/6/16**

4.4. Test (out-of-distribution) performance

The averaged out-of-distribution performance (using the early-stopped model on validation) across each of the eight algorithm types is provided in Table 1; see Appendix C for the full results on all 30 algorithms. MPNNs are unable to transfer their impressive gains to graphs that are four times larger: in fact, the PGN takes over as the most performant model when averaged across task types—this aligns well with prior research (Veličković et al., 2020). The outperformance is also observed when we count how frequently each model is among the best-performing models for a given algorithm, as per our "win/tie/loss" metric, which we explain in Appendix C. GNN models, additionally, outperform models like Deep Sets and Memory Nets, reinforcing that GNNs are a useful primitive for algorithmic reasoning (Xu et al., 2019; Dudzik & Veličković, 2022).

Aside from all of the above, we note that the OOD version of the CLRS-30 benchmark is highly challenging and far from solved for most tasks, making it a meaningful informant of future progress in the area. In particular, PGNs struggled on tasks requiring long-range rollouts (such as DFS), or recursive reasoning (such as Quicksort and Quickselect). This invites further research in algorithmic reasoners that can support such computation. It is further revealed that more specialised inductive biases and training regimes may be required to deal with string matching algorithms (such as KMP), and that the processor studied here tended to perform the best on tasks which were of favourable (sublinear) complexity in terms of hint counts (such as BFS, Bellman-Ford, and task scheduling).

The specific results we obtain with our baselines validate several bits of prior research in the area, but also demonstrate we still have a long way to go, with even simple OOD scenarios only being fit to about 50% micro-F_1 performance.

5. Conclusion

We introduce CLRS-30, a dataset that contains trajectories from 30 classical algorithms. This benchmark constitutes an effective way to test out-of-distribution generalization and transfer, and brings a means to evaluate algorithmic reasoning learnt by neural network models. The dataset provides input/output pairs for all algorithms, as well as intermediate trajectory information ("hints").

It is our hope that CLRS-30 will be a useful tool to shepherd future research in algorithmic reasoning, as prior art in the area largely generated their own datasets, making progress tracking challenging. Further, we hope that CLRS-30 will make algorithmic reasoning a more accessible area: one does not need a background in theoretical computer science to generate the dataset, and can focus on the modelling.

If we convinced you to try out our library, please consult Appendix A for detailed instructions on most common ways to interact with our platform. CLRS is in constant development, and we welcome any and all feedback.

Acknowledgements

CLRS-30 was developed over a long time-frame, with many useful contributions, which we kindly acknowledge here.

We would like to particularly thank Borja Ibarz for numerous fixes and additions, and laying foundation for future iterations. Additionally, we warmly thank Jonathan Godwin, Sadegh Mahdavi, Euan Ong, MohamedElfatih Salah, Ahmed Elhag, Andreea Deac, Frederik Nijweide, Andrew Dudzik, Thomas Kipf, Amin Barekatain and Dobrik Georgiev for their support, and identifying numerous bugs during development. Finally, we thank Kim Stachenfeld, Nate Kushman and Daan Wierstra for reviewing the paper prior to submission, and anonymous reviewers for their careful feedback, strengthening the paper significantly.

References

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

Alet, F., Lozano-Perez, T., and Kaelbling, L. P. Modular meta-learning. volume 87 of *Proceedings of Machine Learning Research*. PMLR, 2018.

Banino, A., Badia, A. P., Köster, R., Chadwick, M. J., Zambaldi, V., Hassabis, D., Barry, C., Botvinick, M., Kumaran, D., and Blundell, C. Memo: A deep network for flexible combination of episodic memories. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rJxlc0EtDr.

Banino, A., Balaguer, J., and Blundell, C. Pondernet: Learning to ponder. *arXiv preprint arXiv:2107.05407*, 2021.

Belkin, M., Hsu, D., and Xu, J. Two models of double descent for weak features. *arXiv preprint arXiv:1903.07571*, 2019.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

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

Bengio, Y., Lodi, A., and Prouvost, A. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2020.

Bentley, J. Programming pearls: algorithm design techniques. *Communications of the ACM*, 27(9):865–873, 1984.

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

Brody, S., Alon, U., and Yahav, E. How attentive are graph attention networks? *arXiv preprint arXiv:2105.14491*, 2021.

Cappart, Q., Chételat, D., Khalil, E., Lodi, A., Morris, C., and Veličković, P. Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint arXiv:2102.09544*, 2021.

Chen, Z., Chen, L., Villar, S., and Bruna, J. Can graph neural networks count substructures? *arXiv preprint arXiv:2002.04025*, 2020.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*. MIT press, 2009.

Corso, G., Cavalleri, L., Beaini, D., Liò, P., and Veličković, P. Principal neighbourhood aggregation for graph nets. *arXiv preprint arXiv:2004.05718*, 2020.

Deac, A., Bacon, P.-L., and Tang, J. Graph neural induction of value iteration. *arXiv preprint arXiv:2009.12604*, 2020.

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

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

Du, Y., Czarnecki, W. M., Jayakumar, S. M., Pascanu, R., and Lakshminarayanan, B. Adapting auxiliary losses using gradient similarity, 2018.

Dudzik, A. and Veličković, P. Graph neural networks are dynamic programmers. *arXiv preprint arXiv:2203.15544*, 2022.

Dwivedi, V. P., Joshi, C. K., Laurent, T., Bengio, Y., and Bresson, X. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.

Erdös, P. and Rényi, A. On the evolution of random graphs. In *The structure and dynamics of networks*, pp. 38–82. Princeton University Press, 2011.

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

Ford Jr, L. R. and Fulkerson, D. R. *Flows in networks*. Princeton university press, 2015.

Freivalds, K., Ozoliņš, E., and Šostaks, A. Neural shuffle-exchange networks-sequence processing in o (n log n) time. In *Advances in Neural Information Processing Systems*, pp. 6630–6641, 2019.

Gavril, F. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

Georgiev, D. and Lió, P. Neural bipartite matching. *arXiv preprint arXiv:2005.11304*, 2020.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

Godwin, J., Schaarschmidt, M., Gaunt, A. L., Sanchez-Gonzalez, A., Rubanova, Y., Veličković, P., Kirkpatrick, J., and Battaglia, P. Simple gnn regularisation for 3d molecular property prediction and beyond. In *International Conference on Learning Representations*, 2021.

Graham, R. L. An efficient algorithm for determining the convex hull of a finite planar set. *Info. Pro. Lett.*, 1: 132–133, 1972.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Griffiths, T., Callaway, F., Chang, M., Grant, E., Krueger, P., and Lieder, F. Doing more with less: meta-reasoning and meta-learning in humans and machines. *Current Opinion in Behavioral Sciences*, October 2019.

Hamrick, J. B., Allen, K. R., Bapst, V., Zhu, T., McKee, K. R., Tenenbaum, J. B., and Battaglia, P. W. Relational inductive bias for physical construction in humans and machines. *arXiv preprint arXiv:1806.01203*, 2018.

Hennigan, T., Cai, T., Norman, T., and Babuschkin, I. Haiku: Sonnet for JAX, 2020. URL http://github.com/deepmind/dm-haiku.

Hierholzer, C. and Wiener, C. Über die möglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. *Mathematische Annalen*, 6(1):30–32, 1873.

Hoare, C. A. Algorithm 65: find. *Communications of the ACM*, 4(7):321–322, 1961.

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

Jacot, A., Gabriel, F., and Hongler, C. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in Neural Information Processing Systems 31*. 2018.

Jarvis, R. A. On the identification of the convex hull of a finite set of points in the plane. *Information processing letters*, 2(1):18–21, 1973.

Joshi, C. K., Cappart, Q., Rousseau, L.-M., Laurent, T., and Bresson, X. Learning tsp requires rethinking generalization. *arXiv preprint arXiv:2006.07054*, 2020.

Kaiser, Ł. and Sutskever, I. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Khalil, E., Dai, H., Zhang, Y., Dilkina, B., and Song, L. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pp. 6348–6358, 2017.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Knuth, D. E. Fundamental algorithms. 1973.

Knuth, D. E., Morris, Jr, J. H., and Pratt, V. R. Fast pattern matching in strings. *SIAM journal on computing*, 6(2): 323–350, 1977.

Kool, W., van Hoof, H., and Welling, M. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

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

Lake, B. M. Compositional generalization through meta sequence-to-sequence learning. In *Advances in Neural Information Processing Systems 32*, pp. 9791–9801. 2019.

Lawler, E. L. The traveling salesman problem: a guided tour of combinatorial optimization. *Wiley-Interscience Series in Discrete Mathematics*, 1985.

Li, Y., Gimeno, F., Kohli, P., and Vinyals, O. Strong generalization and efficiency in neural programs. *arXiv preprint arXiv:2007.03629*, 2020.

Liu, C., Zhu, L., and Belkin, M. Toward a theory of optimization for over-parameterized systems of non-linear equations: the lessons of deep learning. *CoRR*, abs/2003.00307, 2020.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Mikolov, T., Deoras, A., Kombrink, S., Burget, L., and Cernocký, J. Empirical evaluation and combination of advanced language modeling techniques. In *INTERSPEECH*, pp. 605–608, 2011.

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

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

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

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

Strathmann, H., Barekatain, M., Blundell, C., and Veličković, P. Persistent message passing. *arXiv preprint arXiv:2103.01043*, 2021.

Sukhbaatar, S., Szlam, A., Weston, J., and Fergus, R. End-to-end memory networks. *arXiv preprint arXiv:1503.08895*, 2015.

Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.

Tang, H., Huang, Z., Gu, J., Lu, B., and Su, H. Towards scale-invariant graph-related problem solving by iterative homogeneous gnns. *the 34th Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

Trask, A., Hill, F., Reed, S. E., Rae, J., Dyer, C., and Blunsom, P. Neural arithmetic logic units. In *Advances in Neural Information Processing Systems*, pp. 8035–8044, 2018.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Veličković, P. and Blundell, C. Neural algorithmic reasoning. *arXiv preprint arXiv:2105.02761*, 2021.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Veličković, P., Ying, R., Padovano, M., Hadsell, R., and Blundell, C. Neural execution of graph algorithms. *arXiv preprint arXiv:1910.10593*, 2019.

Veličković, P., Buesing, L., Overlan, M. C., Pascanu, R., Vinyals, O., and Blundell, C. Pointer graph networks. *arXiv preprint arXiv:2006.06380*, 2020.

Veličković, P., Bošnjak, M., Kipf, T., Lerchner, A., Hadsell, R., Pascanu, R., and Blundell, C. Reasoning-modulated representations. *arXiv preprint arXiv:2107.08881*, 2021.

Vinyals, O., Fortunato, M., and Jaitly, N. Pointer networks. In *Advances in Neural Information Processing Systems*, pp. 2692–2700, 2015.

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

Xhonneux, L.-P., Deac, A.-I., Veličković, P., and Tang, J. How to transfer algorithmic reasoning knowledge to learn new algorithms? *Advances in Neural Information Processing Systems*, 34, 2021.

Xu, K., Li, J., Zhang, M., Du, S. S., Kawarabayashi, K.-i., and Jegelka, S. What can neural networks reason about? *arXiv preprint arXiv:1905.13211*, 2019.

Xu, K., Li, J., Zhang, M., Du, S. S., ichi Kawarabayashi, K., and Jegelka, S. How neural networks extrapolate: From feedforward to graph neural networks. *arXiv preprint arXiv:2009.11848*, 2020.

Yan, Y., Swersky, K., Koutra, D., Ranganathan, P., and Heshemi, M. Neural execution engines: Learning to execute subroutines. *arXiv preprint arXiv:2006.08084*, 2020.

Yehuda, G., Gabel, M., and Schuster, A. It's not what machines can learn, it's what we cannot teach. *arXiv preprint arXiv:2002.09398*, 2020.

Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R. R., and Smola, A. J. Deep sets. In *Advances in neural information processing systems*, pp. 3391–3401, 2017.

Zamir, A. R., Sax, A., Shen, W., Guibas, L. J., Malik, J., and Savarese, S. Taskonomy: Disentangling task transfer learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3712–3722, 2018.

Zaremba, W. and Sutskever, I. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

A. Interfacing with the CLRS benchmark

The CLRS benchmark is publicly hosted on GitHub: `https://github.com/deepmind/clrs`. All code and artifacts are released under an **Apache 2.0** license, which is highly permissive.

Within `clrs/examples/run.py`, we demonstrate an extensively configurable example script that evaluates a specific baseline on CLRS-30.

Our baselines are provided in JAX and Haiku (Hennigan et al., 2020), but the dataset is generated using NumPy, making it possible to create learning pipelines in virtually any framework, including PyTorch and TensorFlow.

We will now highlight three key ways in which researchers can interface with the library.

A.1. Evaluating a new baseline on CLRS-30

To support a new baseline, the recommended path depends on how fundamentally different the baseline is to an encode-process-decode GNN.

In most cases, we anticipate that only the processor network needs changing, and the remainder of the architecture can match our baselines. In this case, it is only necessary to implement the new processor network within `clrs/_src/processors.py` and appropriately set `self.mpnn` within the `_construct_processor` method in `clrs/_src/baselines.py`.

For more fundamentally different baselines, it is necessary to create a new class that extends the `Model` API (as found within `clrs/_src/model.py`). `clrs/_src/baselines.py` provides one example of how this can be done efficiently, for the case of our baselines.

A.2. Modifying the data distribution of CLRS-30

If users want to train and/or evaluate the models on different versions of the tasks given in CLRS-30, the key routines to modify are located in `clrs/_src/samplers.py`.

The easiest modification concerns the graph sizes and/or numbers of trajectories. They can be directly changed by modifying the `CLRS30` dictionary near the top of the file.

For more elaborate modifications, e.g. to the specific data sampling distributions, the users would need to modify and/or extend the relevant sampler class. As a guiding example, we provide a `SortingSampler` class which is convenient for generating inputs for sorting algorithms. The specific sampler used for each task is provided in the `SAMPLERS` dictionary towards the end of the file.

A.3. Adding new algorithms to CLRS

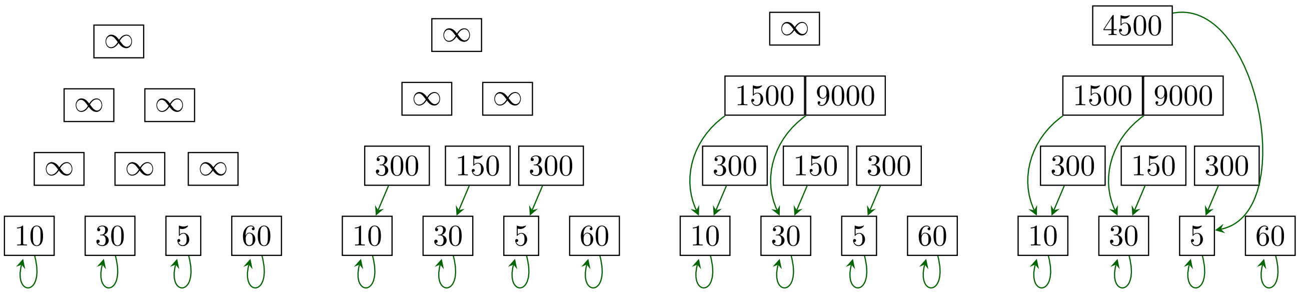
As the most elaborate of the three workflows, adding a new algorithm to the task suite requires following several steps, which are potentially comprehensive, depending on the complexity of the algorithm. However, the CLRS benchmark code still provides may helper routines for probing and batching that facilitate inclusion of novel algorithms. The steps are as follows:

1. First, determine the input/hint/output specification of your algorithm, and include it within the `SPECS` dictionary of `clrs/_src/specs.py`.
2. Implement the desired algorithm in an abstractified form. Examples of this can be found throughout the `clrs/_src/algorithms/` folder.
3. Next, choose appropriate moments within the algorithm's execution to create probes that capture the inputs, outputs and all intermediate state (using the `probing.push` function).
4. Once generated, probes can be prepared using the `probing.finalize` method, and should be returned together with the algorithm output.
5. Lastly, implement an appropriate input data sampler for your algorithm, and include it within the `SAMPLERS` dictionary within `clrs/_src/samplers.py`.

B. Additional worked examples of algorithm trajectories

Matrix Chain Order As a representative dynamic programming algorithm, we visualise the steps of the procedure for optimising the order of multiplications in a chain of matrices, for multiplying matrices of size $(10 \times 30)(30 \times 5)(5 \times 60)$, assuming a $O(n^3)$-time multiplication algorithm.

The algorithm proceeds by filling up an "upper-triangular" part of a dynamic programming matrix, where cell $[i, j]$ corresponds to the optimal number of operations when multiplying all the matrices between the ith and jth. Such an algorithm may also be represented in a "pyramidal" form as below:

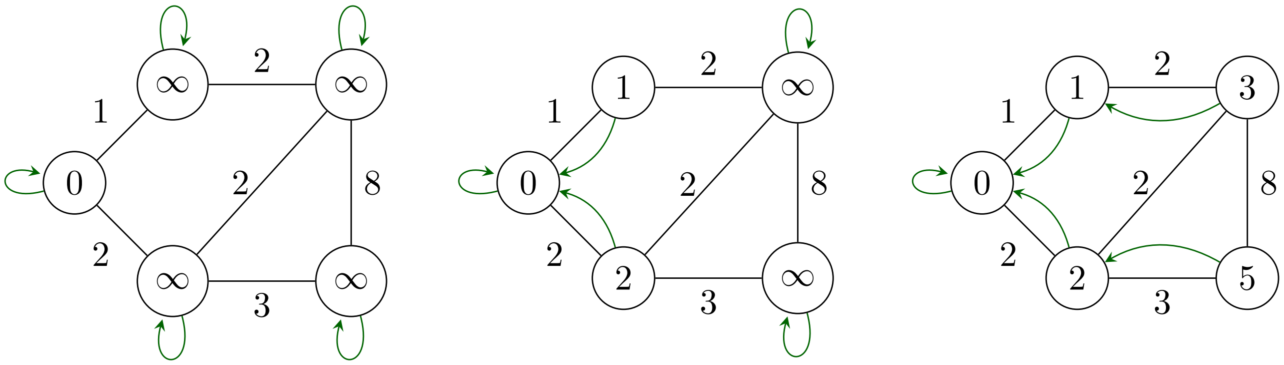


Additionally, the algorithm maintains (and returns) the optimal way to recursively divide each subsequence into two (by storing the optimal dividing point, in green). Here, it is optimal to first multiply $(10 \times 30)(30 \times 5)$ (yielding $1,500$ operations), then multiply the remaning matrices as $(10 \times 5)(5 \times 60)$ (yielding $3,000$ operations; $4,500$ in total).

Note that every pointer points into one of the original n input nodes (at the lowest level), and how each cell of the pyramid corresponds to a pair of input nodes (specifying the corresponding range). Therefore, rather than creating $O(n^2)$ auxiliary nodes, we instead record all relevant values above as edge scalars and edge pointers, and store nodes only for the lowest level of the pyramid. Further, whether or not a particular edge has been populated yet (the "∞" indicator above) is stored as an additional binary flag.

Bellman-Ford As a representative graph algorithm, we visualise the steps of the Bellman-Ford algorithm for finding single-source shortest paths in a given graph.

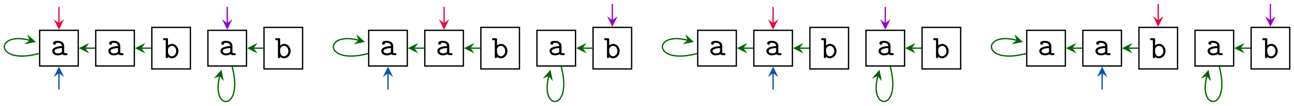
Initially, the source node is labelled with distance zero, and all other nodes with distance "∞" (which, once again, is represented as a binary node hint). The algorithm then iteratively relaxes all edges as follows, until convergence is achieved:



Besides updating the distance values, the algorithm also maintains, and returns, the predicted shortest path tree – for each node, a pointer to its predecessor along the optimal path from the source. By convention, the source node points to itself. These pointers are visualised in green.

Naïve String Matcher As a representative string algorithm, we visualise the steps of the naïve string matcher, for detecting string `"ab"` inside the string `"aab"`.

In this case, each character of the two strings is given a separate node, and three sets of indices are maintained: indicating the start of the current candidate match (in blue); and the current position being checked in both the haystack (red) and the needle (purple). The algorithm scans candidate positions left-to-right until a full match is detected for the first time.



Additionally, each character is tagged with its predecessor in the string (in green), and a binary flag indicating which of the two strings it belongs to (not shown here).

C. Test results for all algorithms

Test performance for all 30 algorithms in CLRS-30 may be found in Table 2. In addition, we provide a "win-tie-loss" metric as another way of differentiating model performance, which is less sensitive to outliers. The resulting counts are provided in Table 3, and are computed as follows:

- Let $\mu_A(\mathcal{M})$ and $\sigma_A(\mathcal{M})$ be the mean and standard deviation of model $\mathcal{M}$'s test performance on algorithm A (as in Table 2).
- We say that model $\mathcal{A}$ outperforms model $\mathcal{B}$ on algorithm A—denoted by $\mathcal{A} \succ_A \mathcal{B}$—if $\mu_A(\mathcal{A}) - \sigma_A(\mathcal{A}) > \mu_A(\mathcal{B})$.
- If $\forall \mathcal{X} \neq \mathcal{A}.\ \mathcal{A} \succ_A \mathcal{X}$, then model $\mathcal{A}$ **wins** on algorithm A.
- Otherwise, if $\exists \mathcal{X}.\ \mathcal{X} \succ_A \mathcal{A}$, then model $\mathcal{A}$ **loses** on algorithm A.
- Otherwise, model $\mathcal{A}$ is **tied** on algorithm A.

The win/tie/loss counts are then aggregated across all algorithms A to obtain a metric for each model. As already mentioned, the details of this on a per-algorithm level are given in Table 3.

D. Validation results individual plots

Validation performance for all 30 algorithms in CLRS-30 may be found in Figure 4. For convenience, we also report the early-stopped validation performance in Table 4.

Table 2. Test performance of all models on all algorithms.

Algorithm	Deep Sets	GAT	Memnet	MPNN	PGN
Activity Selector	66.09% ± 1.67	73.23% ± 1.37	24.10% ± 2.22	**80.66**% ± 3.16	66.80% ± 1.62
Articulation Points	39.06% ± 4.04	37.76% ± 1.62	1.50% ± 0.61	**50.91**% ± 2.18	49.53% ± 2.09
Bellman-Ford	51.33% ± 0.85	87.91% ± 1.19	40.04% ± 1.46	92.01% ± 0.28	**92.99**% ± 0.34
BFS	98.63% ± 0.38	99.04% ± 0.21	43.34% ± 0.04	**99.89**% ± 0.05	99.63% ± 0.29
Binary Search	47.97% ± 0.88	23.50% ± 3.12	14.37% ± 0.46	36.83% ± 0.26	**76.95**% ± 0.13
Bridges	32.43% ± 2.65	25.64% ± 6.60	30.26% ± 0.05	**72.69**% ± **4.78**	51.42% ± 7.82
Bubble Sort	50.73% ± 3.24	9.91% ± 1.77	**73.58**% ± 0.78	5.27% ± 0.60	6.01% ± 1.95
DAG Shortest Paths	73.21% ± 2.42	81.14% ± 1.37	66.15% ± 1.92	96.24% ± 0.56	**96.94**% ± 0.16
DFS	7.44% ± 0.73	11.78% ± 2.04	**13.36**% ± 1.61	6.54% ± 0.51	8.71% ± 0.24
Dijkstra	36.12% ± 3.10	58.01% ± 0.79	22.48% ± 2.39	**91.50**% ± 0.50	83.45% ± 1.75
Find Max. Subarray	12.48% ± 0.39	24.43% ± 0.43	13.05% ± 0.08	20.30% ± 0.49	**65.23**% ± 2.56
Floyd-Warshall	7.22% ± 0.90	16.66% ± 3.14	14.17% ± 0.13	26.74% ± 1.77	**28.76**% ± 0.51
Graham Scan	64.71% ± 2.75	77.89% ± 2.70	40.62% ± 2.31	**91.04**% ± 0.31	56.87% ± 1.61
Heapsort	28.94% ± 12.57	10.35% ± 1.83	**68.00**% ± 1.57	10.94% ± 0.84	5.27% ± 0.18
Insertion Sort	40.98% ± 4.65	29.52% ± 1.87	**71.42**% ± 0.86	19.81% ± 2.08	44.37% ± 2.43
Jarvis' March	50.25% ± 0.81	**51.51**% ± 10.25	22.99% ± 3.87	34.86% ± 12.39	49.19% ± 1.07
KMP Matcher	**3.22**% ± 0.54	3.03% ± 0.36	1.81% ± 0.00	2.49% ± 0.86	2.00% ± 0.12
LCS Length	50.10% ± 5.25	**57.88**% ± 1.02	49.84% ± 4.34	53.23% ± 0.36	56.82% ± 0.21
Matrix Chain Order	78.36% ± 3.58	78.19% ± 3.31	81.96% ± 1.03	79.84% ± 1.40	**83.91**% ± 0.49
Minimum	80.19% ± 2.08	84.20% ± 2.95	86.93% ± 0.11	85.34% ± 0.88	**87.71**% ± 0.52
MST-Kruskal	60.58% ± 4.71	65.72% ± 0.99	28.84% ± 0.61	**70.97**% ± 1.50	66.96% ± 1.36
MST-Prim	12.17% ± 5.47	38.20% ± 4.34	10.29% ± 3.77	**69.08**% ± 7.56	63.33% ± 0.98
Naïve String Match	2.05% ± 0.29	3.01% ± 1.20	1.22% ± 0.48	**3.92**% ± 0.30	2.08% ± 0.20
Optimal BST	69.71% ± 1.36	65.49% ± 1.75	**72.03**% ± 1.21	62.23% ± 0.44	71.01% ± 1.82
Quickselect	3.21% ± 1.33	**4.36**% ± 0.95	1.74% ± 0.03	1.43% ± 0.69	3.66% ± 0.42
Quicksort	37.74% ± 2.16	7.60% ± 0.98	**73.10**% ± 0.67	11.30% ± 0.10	6.17% ± 0.15
Segments Intersect	77.29% ± 0.60	90.41% ± 0.04	71.81% ± 0.90	**93.44**% ± 0.10	77.51% ± 0.75
SCC	17.81% ± 2.61	12.70% ± 3.12	16.32% ± 4.78	**24.37**% ± 4.88	20.80% ± 0.64
Task Scheduling	84.84% ± 0.70	84.69% ± 2.09	82.74% ± 0.04	84.11% ± 0.32	**84.89**% ± 0.91
Topological Sort	15.84% ± 3.57	27.03% ± 6.92	2.73% ± 0.11	52.60% ± 6.24	**60.45**% ± 2.69
Overall average	43.36%	44.69%	38.03%	51.02%	**52.31**%

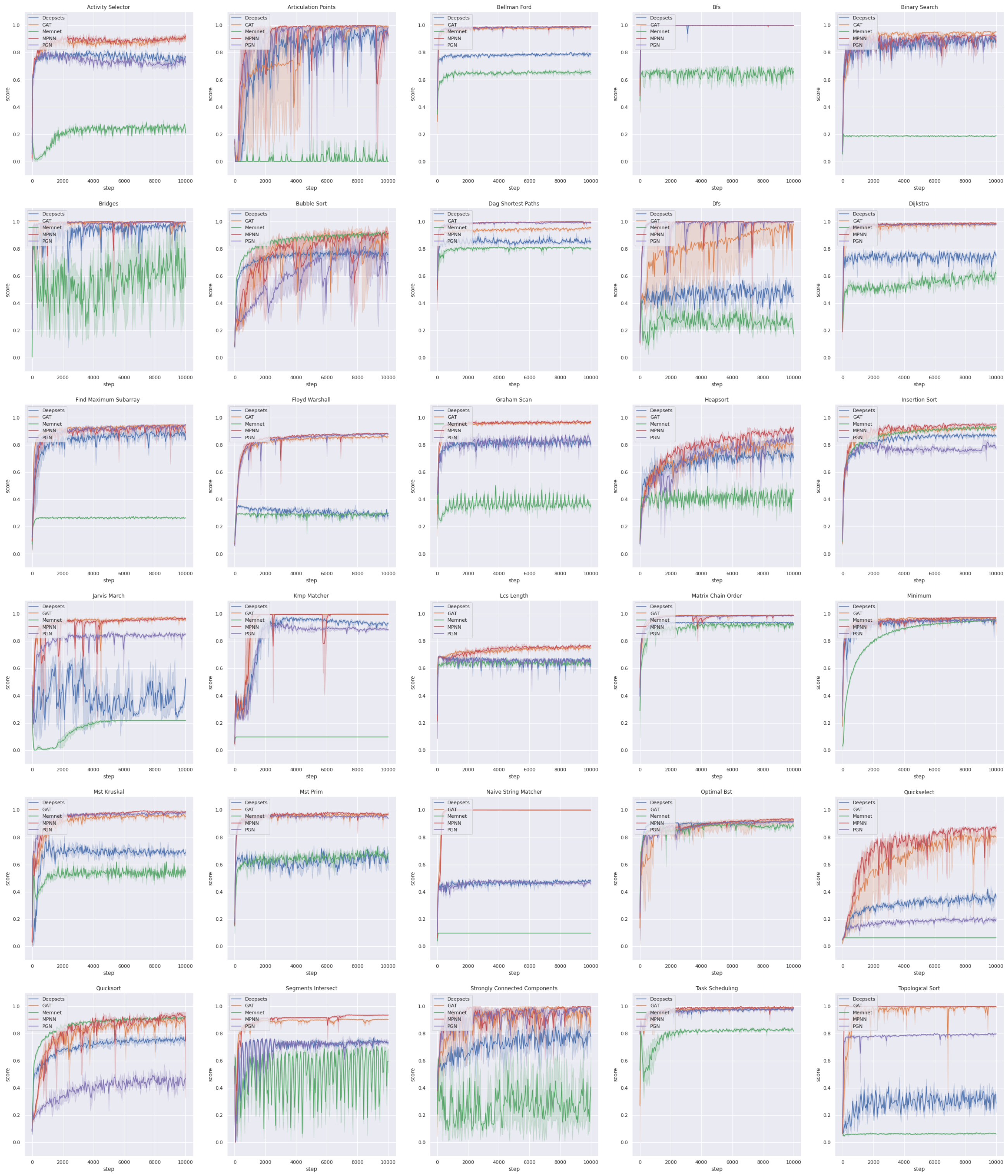


Figure 4. Validation results on all 30 algorithms in CLRS-30, averaged over three seeds.

Table 3. Win/Tie/Loss counts of all models on all algorithms. Legend: **W**: win, *T*: tie, L: loss.

Algorithm	**Deep Sets**	**GAT**	**Memnet**	**MPNN**	**PGN**
Activity Selector	L	L	L	**W**	L
Articulation Points	L	L	L	*T*	*T*
Bellman-Ford	L	L	L	L	**W**
BFS	L	L	L	**W**	L
Binary Search	L	L	L	L	**W**
Bridges	L	L	L	**W**	L
Bubble Sort	L	L	**W**	L	L
DAG Shortest Paths	L	L	L	L	**W**
DFS	L	*T*	*T*	L	L
Dijkstra	L	L	L	**W**	L
Find Max. Subarray	L	L	L	L	**W**
Floyd-Warshall	L	L	L	L	**W**
Graham Scan	L	L	L	**W**	L
Heapsort	L	L	**W**	L	L
Insertion Sort	L	L	**W**	L	L
Jarvis' March	*T*	*T*	L	L	L
KMP Matcher	*T*	*T*	L	L	L
LCS Length	L	**W**	L	L	L
Matrix Chain Order	L	L	L	L	**W**
Minimum	L	L	L	L	**W**
MST-Kruskal	L	L	L	**W**	L
MST-Prim	L	L	L	*T*	*T*
Naïve String Match	L	L	L	**W**	L
Optimal BST	L	L	*T*	L	*T*
Quickselect	L	*T*	L	L	*T*
Quicksort	L	L	**W**	L	L
Segments Intersect	L	L	L	**W**	L
SCC	L	L	L	*T*	*T*
Task Scheduling	*T*	*T*	L	L	*T*
Topological Sort	L	L	L	L	**W**
Overall counts	0/3/27	1/5/24	4/2/24	8/3/19	**8/6/16**

Table 4. Early-stopped validation results of all models on all algorithms.

Algorithm	**Deep Sets**	**GAT**	**Memnet**	**MPNN**	**PGN**
Activity Selector	83.50% ± 0.17	92.40% ± 0.50	34.59% ± 2.15	**93.89**% ± 0.39	82.26% ± 0.19
Articulation Points	99.63% ± 0.31	**100.00**% ± 0.00	16.84% ± 1.03	**100.00**% ± 0.00	**100.00**% ± 0.00
Bellman-Ford	81.12% ± 0.14	99.28% ± 0.14	68.75% ± 0.42	**99.48**% ± 0.05	99.35% ± 0.05
BFS	**100.00**% ± 0.00	**100.00**% ± 0.00	70.70% ± 0.09	**100.00**% ± 0.00	**100.00**% ± 0.00
Binary Search	93.34% ± 0.41	**95.72**% ± 0.17	20.33% ± 0.28	94.19% ± 0.12	94.17% ± 0.08
Bridges	99.36% ± 0.05	**100.00**% ± 0.00	96.46% ± 1.13	**100.00**% ± 0.00	**100.00**% ± 0.00
Bubble Sort	81.51% ± 1.02	**95.44**% ± 1.01	92.64% ± 0.14	94.53% ± 1.84	87.17% ± 5.46
DAG Shortest Paths	92.25% ± 0.28	96.81% ± 0.05	81.90% ± 0.05	**99.93**% ± 0.05	99.80% ± 0.00
DFS	62.76% ± 1.26	99.22% ± 0.64	47.72% ± 0.45	**100.00**% ± 0.00	**100.00**% ± 0.00
Dijkstra	80.34% ± 0.42	99.22% ± 0.40	67.38% ± 0.70	**99.67**% ± 0.14	99.28% ± 0.05
Find Max. Subarray	91.41% ± 0.22	95.00% ± 0.32	27.91% ± 0.08	95.13% ± 0.37	**95.30**% ± 0.16
Floyd-Warshall	35.79% ± 0.04	87.28% ± 0.09	31.29% ± 0.04	**89.14**% ± 0.03	88.70% ± 0.15
Graham Scan	87.66% ± 0.24	97.85% ± 0.11	53.53% ± 1.58	**98.45**% ± 0.15	89.06% ± 0.27
Heapsort	81.84% ± 0.33	87.24% ± 2.23	54.04% ± 0.28	**94.27**% ± 0.11	90.36% ± 0.67
Insertion Sort	89.58% ± 0.28	95.18% ± 0.58	94.40% ± 0.14	**96.74**% ± 0.19	84.57% ± 0.82
Jarvis' March	72.82% ± 0.42	**98.38**% ± 0.16	37.92% ± 6.61	97.94% ± 0.25	88.34% ± 0.36
KMP Matcher	98.03% ± 0.21	99.76% ± 0.08	9.67% ± 0.00	**99.87**% ± 0.05	94.14% ± 0.99
LCS Length	69.24% ± 0.36	77.00% ± 0.19	67.69% ± 0.24	**77.88**% ± 0.42	69.19% ± 0.04
Matrix Chain Order	94.46% ± 0.02	**99.37**% ± 0.03	93.91% ± 0.10	99.12% ± 0.04	99.21% ± 0.03
Minimum	97.59% ± 0.11	**97.74**% ± 0.21	95.56% ± 0.10	97.64% ± 0.05	97.07% ± 0.14
MST-Kruskal	83.79% ± 2.01	97.93% ± 0.25	64.65% ± 0.95	**99.71**% ± 0.17	99.12% ± 0.08
MST-Prim	74.61% ± 0.32	98.37% ± 0.14	74.09% ± 0.28	**99.02**% ± 0.09	97.79% ± 0.14
Naïve String Match	49.80% ± 0.15	**100.00**% ± 0.00	9.91% ± 0.20	**100.00**% ± 0.00	50.33% ± 0.08
Optimal BST	92.02% ± 0.14	93.30% ± 0.49	90.86% ± 0.40	**93.88**% ± 0.11	93.20% ± 0.27
Quickselect	42.30% ± 0.92	83.82% ± 1.86	6.56% ± 0.25	**88.74**% ± 0.78	54.02% ± 0.17
Quicksort	79.69% ± 1.12	92.97% ± 0.40	93.16% ± 0.24	**95.70**% ± 0.40	54.30% ± 1.42
Segments Intersect	77.49% ± 0.12	90.82% ± 0.16	71.57% ± 1.08	**93.84**% ± 0.20	78.32% ± 0.18
SCC	89.52% ± 1.23	**100.00**% ± 0.00	70.57% ± 1.43	**100.00**% ± 0.00	99.93% ± 0.05
Task Scheduling	99.16% ± 0.04	99.80% ± 0.04	84.80% ± 0.09	**100.00**% ± 0.00	99.06% ± 0.08
Topological Sort	47.23% ± 0.81	**100.00**% ± 0.00	8.30% ± 0.50	**100.00**% ± 0.00	**100.00**% ± 0.00
Overall average	80.93%	95.66%	57.92%	**96.63**%	89.47%