Neural Algorithmic Reasoning with Multiple Correct Solutions

Zeno Kujawa*
Northeastern University
Boston, Massachusetts, USA
kujawa.z@northeastern.edu

John Poole*
University of Cambridge
Cambridge, United Kingdom
jmp251@cam.ac.uk

Dobrik Georgiev
University of Cambridge
Cambridge, United Kingdom
dgg30@cam.ac.uk

Danilo Numeroso
Università di Pisa
Pisa, Italy
danilo.numeroso@phd.unipi.it

Henry Fleischmann
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
hfleisch@andrew.cmu.edu

Pietro Liò
University of Cambridge
Cambridge, United Kingdom
Pietro.Lio@cam.ac.uk

Abstract

Neural Algorithmic Reasoning (NAR) extends classical algorithms to higher-dimensional data. However, canonical implementations of NAR train neural networks to return only a single solution, even when there are multiple correct solutions to a problem, as is the case with finding single-source shortest paths in a graph. For some applications, it is desirable to recover more than one correct solution. To that end, we give the first method for NAR with multiple solutions. We demonstrate our method on two classical algorithms: Bellman-Ford (BF) and Depth-First Search (DFS), favouring deeper insight into two algorithms over a broader survey of algorithms. This method involves generating appropriate training data as well as sampling and validating solutions from model output. Each step of our method, which can serve as a framework for neural algorithmic reasoning beyond the tasks presented in this paper, might be of independent interest to the field and our results represent the first attempt at this task in the NAR literature.

CCS Concepts

• **Computing methodologies → Machine learning**.

Keywords

Neural Algorithmic Reasoning, Graph Neural Networks

ACM Reference Format:
Zeno Kujawa, John Poole, Dobrik Georgiev, Danilo Numeroso, Henry Fleischmann, and Pietro Liò. 2025. Neural Algorithmic Reasoning with Multiple Correct Solutions. In *Workshop on Machine Learning on Graphs in the Era of Generative Artificial Intelligence at KDD '25 (MLoG-GenAI Workshop at KDD '25).* ACM, New York, NY, USA, 14 pages.

1 Introduction

1.0.1 The problem with algorithms. Classical algorithms like Merge Sort guarantee correctness and generalize perfectly on abstract inputs. Unfortunately, real problems rarely have abstract inputs. Consider pathfinding: to give traffic directions via a shortest paths algorithm, one must compress physical distance and congestion data for each street into single-number edge labels, thereby losing information. Bad edge labels lead to bad directions, even despite a good algorithm.

1.0.2 Why NAR?. Neural Networks (NNs) avoid information loss by directly operating on high-dimensional data. Instead of compressing distance and congestion data to a single path-length edge label, an NN can operate directly on distance and congestion data. However, the chief hurdle to NN performance is failure to generalize: common architectures are fragile to changes in input size and structure — exactly where algorithms succeed. Thus, it is natural to wonder whether an NN might learn to generalize from an algorithm. To that end, Neural Algorithmic Reasoning is the field that aims to train NNs to mimic the running of classical algorithms [9]. A benefit of this approach is staying high-dimensional: no information is lost when projecting high-dimensional real data into a single abstract output for a classical algorithm [9].

1.0.3 Multiple Solutions. Thus far, NNs have been trained to mimic algorithms which recover *one* solution from the space of possible correct solutions [1–3, 5, 8, 10, 11]. When recovering *one* solution, an NN trained to perform DFS produces an array, considered a distribution of probabilities of different solutions from which the likeliest solution is returned as the model output. To recover *multiple* solutions, we adapt the NN training so the NN output distribution better reflects multiple correct solutions, and present methods of retrieving multiple solutions from the NN output distribution.[1]

1.0.4 Why multiple solutions? For applications like cyberphysical systems, the flexibility of multiple solutions is essential. A single solution, for reasons of cost, safety, or compliance, may not be amenable to a final task. Furthermore, using only one solution in NAR may be vulnerable to local minima in the model training. By diversifying the space of solutions returned by NAR, we may decrease the risk of solutions originating from a local minimum.

1.1 Contributions

We give the first analysis of Neural Algorithmic Reasoners which return more than one solution. For training examples, we create distributions of solutions from multiple runs of classical algorithms with randomized tiebreaking. We train NNs to predict the generated

*Equal contribution

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MLoG-GenAI Workshop at KDD '25, Toronto, Canada
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

[1]The code used for the experimenets in this paper can be found here.

distribution of solutions; and we leverage randomness in algorithm-specific ways to extract distinct solutions from the NN-predicted distributions. We also discuss methods of evaluating the solutions our methods yield. Thus, we contribute methods that might each be of independent interest, methods to:

(1) train a neural network on a distribution of correct solutions
(2) extract multiple solutions from the NN output distribution
(3) evaluate the diversity and correctness of many NN solutions

Together, these contributions can serve as a framework for further research into Neural Algorithmic Reasoning with multiple solutions for a single problem instance.

2 Methodology

2.0.1 Generate distribution, train, extract many solutions. To produce multiple solutions for a single graph, we train a NN to predict a distribution of solutions for each individual graph (solutions being individual DFS trees or BF paths, their distribution represented as distributions over child-parent pairs, as discussed in Appendices A.1.1 and A.2.1). Ordinarily, the CLRS benchmark [10] generates training data by running classical algorithms with deterministic tiebreaking. For example, the input graph in Figure 1 has two valid orders of depth-first traversal, but classical DFS will only return one traversal based on which edge it explores first. To create our training examples, we run a classical algorithm (e.g. DFS) multiple times with randomized tiebreaking, and average the outputs (e.g. individual DFS trees) into a distribution of solutions. As we want the NN to predict a distribution, we train to minimize a distribution divergence (Kullback-Leibler: KL) [6] between the input distribution of solution and that given by the NN. Finally, our goal is to recover multiple solutions for an input graph; so we develop algorithm-specific stochastic sampling methods.

2.0.2 Stochasticity gives multiple solutions. There are two sources of randomness in our process: changing training labels from a deterministic algorithm result to a probability distribution over predecessor-successor relationships, and extracting solutions from predecessor arrays that encode the model output distribution. Stochasticity of sampling methods means samples are likely to have multiple different solutions. For DFS, we sample the immediate parents of vertices until we reach the root vertex (which can be seen as executing DFS in reverse on a graph); our two DFS extractors, which we call *Upwards* and *AltUpwards*, differ in the manner in which they discard potential solutions (Appendix A.1.2). For BF, *Beam* samples a number of parents for non-source vertices according to parent-probabilities and chooses the lowest cost path arising (Appendix A.2.2). Also for BF, *Greedy* samples parents for each non-source vertex and chooses the parent with the lowest-cost outgoing edge for each non-source vertex (Appendix A.2.2).

2.1 DFS

2.1.1 Problem setting. We consider a generalization of the deterministic DFS setting implemented in the CLRS benchmark [10]. In our setting, edge-exploration is random: from a given vertex, any of the outgoing edges may be explored first. To decrease the computational burden of verifying solutions, we require that once DFS backtracks past the root of a connected component, it restarts

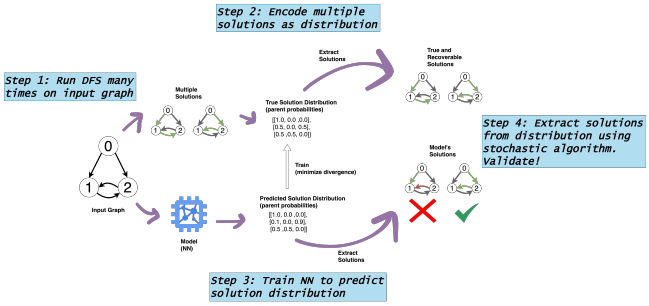


Figure 1: Blueprint for NAR with multiple correct solutions. For Step 2 see Figure 3. For Step 4 see Figure 4 and Appendices A.1.2 A.2.2. Best viewed on a screen.

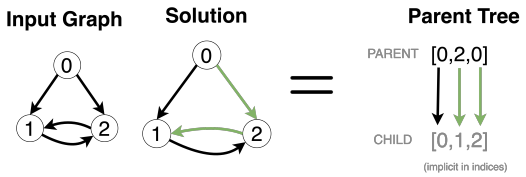


Figure 2: Parent Tree encoding of a single DFS solution. The start node, 0, is its own parent, represented by the value 0 at index 0. The value 2 at index 1 indicates that vertex 2 is the parent of vertex 1. Similarly, the value 0 at index 2 indicates that vertex 0 is the parent of vertex 2.

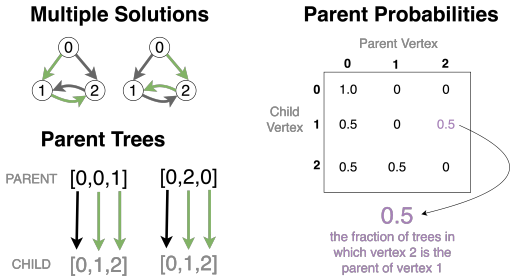


Figure 3: Parent Probability Distribution encoding of multiple DFS solutions. The value 0.5 in row 1 column 2 indicates that vertex 2 is a parent of vertex 1 in 50% of correct parent trees. We call the parent probabilities obtained by repeatedly running an algorithm the *true parent distribution* $\mathcal{P}$. The model outputs, which we aim to be as close as possible to $\mathcal{P}$, are denoted as $\tilde{\mathcal{P}}$.

its search at the lowest-indexed unvisited vertex. We believe this generalization is natural, modelling a setting where one conducts DFS according to a chosen sequence of starting points, without assumptions about which edges to pursue first.

2.1.2 Randomness in training. We generate an empirical parent distribution $\mathcal{P}$ for each graph in the test set (Figure 3) by running our randomized version of DFS 20 times (we conclude from our results in Appendix B that 20 runs provide a good approximation

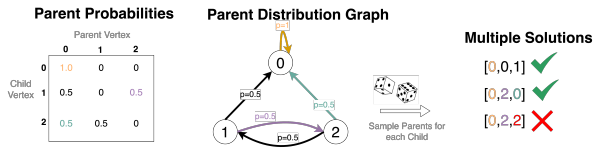


Figure 4: DFS tree extraction from a parent probability matrix: $\tilde{\mathcal{P}}$ or $\mathcal{P}$. The parent tree [0,2,2] is incorrect because there is no solution in which 2 is its own parent. [0,0,0] is incorrect, despite there being solutions in which 0 is the parent of each vertex, because there is no solution in which 0 is the parent of every vertex: if 0 is the parent of 1, 1 must be the parent of 2. Best viewed on a screen.

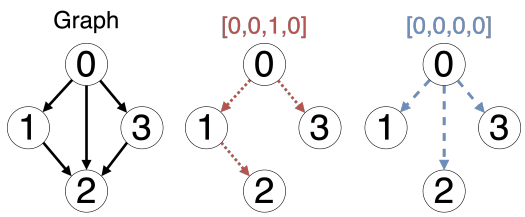


Figure 5: A graph with multiple correct DFS solutions. Graph edges are drawn in black. The red tree explores vertices in increasing order, while the blue tree begins with vertex 0 discovering vertex 2. Note also how the DFS-tree is encoded as a predecessor array π (where the entry at position i is the predecessor of the ith vertex).

of the target distribution for each graph). We then train the model to minimize the KL-divergence between its predicted distribution $\tilde{\mathcal{P}}$ and $\mathcal{P}$ (Appendix A.1.1).

2.1.3 Randomness in sampling. We compare 4 methods, which we refer to as *Upwards altUpwards, Argmax*, and *Random*, for extracting DFS solutions from a parent distribution, $\tilde{\mathcal{P}}$. Our two main methods sample a parent for each vertex in a way that strives to emulate the structure of a DFS traversal while taking into account the given probability distribution. We provide detailed descriptions of our methods in Appendix A.1.2.

2.2 Bellman-Ford

The Bellman-Ford algorithm computes the minimum-cost paths from a single source vertex s to each other vertex v in the graph. As is the case for DFS, the solutions are represented as a predecessor array π.

2.2.1 Randomness in training. Similarly to randomising DFS, our implementation of a randomised Bellman-Ford algorithm relies on randomising the order in which paths are considered by randomising the tiebreaking between minimum-length paths. As with DFS, we obtain a distribution of parent nodes by re-running the algorithm 20 times and computing the frequency of each parent for each node (Appendix A.2.1).

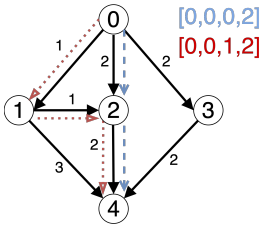


Figure 6: A graph with multiple possible minimum-cost $(source, v)$-paths. As previously, the black arrows represent the edges, with the red and blue edges showing different minimum-cost paths corresponding to their respective predecessor arrays.

2.2.2 Randomness in sampling. We use a sampling algorithm we call *BF Beamsearch* to sample predecessor arrays π representing the minimum-cost (s, v)-paths for each vertex in the graph. This algorithm closely resembles standard Beam Search, but it differs as it samples according to the distribution given by the model (A.2.2). To sample predecessor arrays more easily from $\mathcal{P}$ and $\tilde{\mathcal{P}}$, we also use a greedy randomised algorithm (A.2.2).

2.3 Permuting Inputs

We compare our method for generating multiple solutions by predicting solution distributions and stochastically extracting solutions, with another method: permuting the inputs to the standard CLRS benchmark model [10]. Each input graph may be encoded many times isomorphically by permuting node labels. We test whether permuting node labels will lead to a neural network extracting a new solution.

3 Experiments

3.0.1 Parameters. We run our experiments using the CLRS benchmark [10], which handles data generation, model training, and model evaluation in a unified way. The CLRS benchmark generates graphs of size 4, 7, 11, 13, and 16 to train our models on. In keeping with established methodology [5, 10], we test our models on graphs within and outside the range of our training sizes. For within-sample testing, we choose $n = |V(G)| = 5$ and $n = 16$ in order to cover the range of our training data graph sizes. For out-of-distribution evaluation, we use the default CLRS benchmark testing size $n = 64$. For each graph size, we test 32 Erdős-Rényi random graphs constructed with edge probability 1/2. The models are trained with default CLRS benchmark model parameters (Appendix D) [10].

3.0.2 Metrics. Addressing the lack of evaluation methods for NAR with multiple solutions, we propose metrics to evaluate the validity and diversity of solutions. We regard a solution graph G' to be valid for a randomized algorithm A and a graph G if G' is in the set of possible outputs of $A(G)$. We assess the diversity of solutions by considering the degree to which repeatedly sampling from the same distribution produces different solutions. For a collection of graphs and corresponding solutions, we call the fraction of correct solutions produced by a given method the *accuracy* of the method (referred to as *graph accuracy in [7]*). For a single graph,

our NN predicts one solution distribution. From this distribution, we sample 5 solutions and count the number of distinct and valid solutions. For Bellman-Ford, a solution is valid if and only if it gives all shortest paths from the source vertex. For DFS, no straightforward evaluation method is known and we thus present a novel verification algorithm in Appendix A.1.3. Figures 3 and 4 illustrate data generation and solution extraction on a graph. We investigate our coverage of the solution space in Appendix C.

4 Results

Note validity in Table 1 and accuracy in Tables 2 and 3 report graph accuracy, which is much stricter than the typically reported node accuracy [5, 10] (Appendix E). On an $n = 64$ graph, a method with node accuracy of 92% would be expected to score only 1% graph accuracy.

4.1 Results for Bellman-Ford

4.1.1 Small graphs are easy for BF. On small graphs, ($n = 5$), our sampling methods extract a single correct minimum-cost path with perfect accuracy from the NN-predicted distributions (Table 1, Table 2). To validate sampling methods, we also extract solutions from the empirical distributions and conclude that our sampling methods are successful across graph sizes. Similar results for empirical and predicted distributions suggest the NN predicts a good distribution for $n = 5$ graphs. The lack of variety in solutions, with sampling methods always extracting the same solution, is expected for small graphs as there is likely only one minimum-cost path from the source to each target. Diversity increases for $n = 16$ (Table 1), with the sampling methods producing a correct solution in all cases when sampling from the empirical distributions and in more than 90% of cases when sampling from the model output distributions. In the case of $n = 64$ graphs sampling a predecessor array five times gives a new solution in nearly all cases, though not always correctly (Table 1). In Appendix C.1, we examine the diversity of the returned solution in comparison to running the randomized version of Bellman-Ford.

4.1.2 Large Graphs are harder, but doable for BF. For larger graphs, Table 2 shows that the sampling methods successfully extract correct minimum-cost paths from the empirical distributions (denoted as $\mathcal{P}$). However, extraction is often inaccurate from the distributions of solutions predicted by our models (denoted $\tilde{\mathcal{P}}$). These results permit us to conclude that our models perform well on small graphs, but have difficulty scaling outside of their training distribution. Comparing our approach with the deterministic algorithmic frameworks of [5, 10] is disadvantageous to our method due to its broader scope. Furthermore in [5, 10], node accuracy is used, meaning that a given solution for a given graph and algorithm scores 50% if half of its parent-successor relationships are the same as in the (single) correct solution for the graph. There is no such optimal solution that we could compare a given solution against in our case and we thus use graph accuracy (see [7] and Appendix E for further discussion of the metrics). It is important to note that graph accuracy is much stricter than node accuracy: a method with node accuracy of 92% would be expected to score only 1% graph accuracy on a $n = 64$ graph. Therefore, it is difficult to compare our results to previous work in NAR. However, our results are encouraging for future work, as we have demonstrated the feasibility of training neural networks to emulate a version of Bellman-Ford that is more powerful than its deterministic variant.

Table 1: Mean proportions of unique and valid solutions when sampling 5 solutions from empirical ($\mathcal{P}$) and model output ($\tilde{\mathcal{P}}$) distributions for each graph in 5 training runs. Rounded figures.

	Bellman-Ford			
	Uniques ($\mathcal{P}$)	**V**alids ($\mathcal{P}$)	**U** ($\tilde{\mathcal{P}}$)	**V** ($\tilde{\mathcal{P}}$)
Greedy (n=5)	0.20 ± 0.0	1.00 ± 0.0	0.20 ± 0.0	1.00 ± 0.0
Beam (n=5)	0.20 ± 0.0	1.00 ± 0.0	0.20 ± 0.0	1.00 ± 0.0
Greedy (n=16)	0.35 ± 0.0	1.00 ± 0.0	0.40 ± 0.1	0.92 ± 0.0
Beam (n=16)	0.34 ± 0.0	1.00 ± 0.0	0.39 ± 0.0	0.90 ± 0.0
Greedy (n=64)	1.00 ± 0.0	1.00 ± 0.0	1.00 ± 0.0	0.40 ± 0.2
Beam (n=64)	1.00 ± 0.0	1.00 ± 0.0	1.00 ± 0.0	0.54 ± 0.2
	Depth-First Search			
	Uniques ($\mathcal{P}$)	**V**alids ($\mathcal{P}$)	**U** ($\tilde{\mathcal{P}}$)	**V** ($\tilde{\mathcal{P}}$)
Upwards (n=5)	0.73 ± 0.0	0.36 ± 0.0	0.45 ± 0.0	0.19 ± 0.0
AltUpwards (n=5)	0.60 ± 0.0	0.80 ± 0.0	0.66 ± 0.0	0.77 ± 0.0
Upwards (n=16)	1.00 ± 0.0	0.10 ± 0.0	1.00 ± 0.0	0.01 ± 0.0
AltUpwards (n=16)	1.00 ± 0.0	0.18 ± 0.0	1.00 ± 0.0	0.14 ± 0.0
Upwards (n=64)	1.00 ± 0.0	0.02 ± 0.0	1.00 ± 0.0	0.01 ± 0.0
AltUpwards (n=64)	1.00 ± 0.0	0.05 ± 0.0	1.00 ± 0.0	0.03 ± 0.0

4.2 Results for Depth-First Search

4.2.1 DFS is hard, consistent with literature. Our results indicate that even on small graphs, our methods do not achieve a good accuracy (Table 2). Without modifications such as hint pointer reversal [1], DFS is a difficult NAR problem [5, 10], and our results point towards the additional difficulty of learning to execute DFS when considering multiple solutions. As in the discussion of our results for Bellman-Ford, we point out that no graph accuracy comparison exists with previous work. In [5], it is indicated the node-level micro-F1 score for DFS does not exceed 60% for $n = 64$ without hint pointer reversal [1] (these values being an improvement over the previous state of the art), making graph-level accuracies above 0% unlikely. Thus, while our reported accuracies are low, they are likely to be in line with previous work in NAR, despite the additional difficulty of retrieving multiple solutions.

4.2.2 Many incorrect solutions make DFS harder. Considering the accuracy results and the fact that even on the solution distributions, our sampling methods do not produce correct solutions, we hypothesize that the encoding of multiple solutions into a probability distribution, as presented above, leads to a lack of differentiation between different correct solutions and thus to difficulties retrieving correct solutions. Further evidence for this hypothesis is the observation that there are fewer correct solutions to the single-source shortest path problem on one of the random weighted graphs we sample than there are valid depth-first traversals of that same graph. This is the case because for a given unweighted graph G, every

path from the root could be included in a depth-first traversal, but only the shortest paths from the root to the vertices can be part of the output of the randomized Bellman-Ford algorithm on a given weighted version of G.

Table 2: Mean accuracies (0 to 1) for all sampling methods, with standard deviations over 5 runs. Rounded figures.

	Bellman-Ford			
	Argmax	Beam	Greedy	Random
True (n=5)	1.00 ± 0.0	1.00 ± 0.0	1.00 ± 0.0	0.00 ± 0.0
Model (n=5)	1.00 ± 0.0	1.00 ± 0.0	1.00 ± 0.0	0.00 ± 0.0
True (n=16)	1.00 ± 0.0	1.00 ± 0.0	1.00 ± 0.0	0.00 ± 0.0
Model (n=16)	0.98 ± 0.0	0.91 ± 0.0	0.92 ± 0.0	0.00 ± 0.0
True (n=64)	1.00 ± 0.0	1.00 ± 0.0	1.00 ± 0.0	0.00 ± 0.0
Model (n=64)	0.50 ± 0.3	0.56 ± 0.2	0.38 ± 0.2	0.00 ± 0.0

	Depth-First Search			
	Argmax	AltUpwards	Upwards	Random
True (n=5)	0.80 ± 0.1	0.90 ± 0.1	0.34 ± 0.2	0.00 ± 0.0
Model (n=5)	0.61 ± 0.1	0.50 ± 0.1	0.16 ± 0.0	0.00 ± 0.0
True (n=16)	0.00 ± 0.0	0.00 ± 0.0	0.08 ± 0.0	0.00 ± 0.0
Model (n=16)	0.00 ± 0.0	0.00 ± 0.0	0.00 ± 0.0	0.00 ± 0.0
True (n=64)	0.00 ± 0.0	0.00 ± 0.0	0.01 ± 0.0	0.00 ± 0.0
Model (n=64)	0.00 ± 0.0	0.00 ± 0.0	0.00 ± 0.0	0.00 ± 0.0

4.3 Results for Permuting Inputs

Permuting node indices either fails to produce new solutions, or else incurs significant accuracy penalties (Table 3). When permuting indices produces a correct solution, we observe it to be isomorphic to the original correct solution. When permuting indices produces a new solution, it is incorrect. For DFS with ordered restarts, some failure is explained because a correct solution may become incorrect after a permutation. For example, the ordered restarts requirement means any tree becomes incorrect if its start node is relabeled from 0 to anything else. However, for the Bellman-Ford algorithm, a correct solution ought to remain correct after permuting the vertex indices: shortest paths remain shortest paths, despite relabeling. Yet even for BF, the NN does not appear to be capable of generating any solutions that are correct and not isomorphic to a single solution.

5 Conclusion

We outline a two-part approach to NAR with multiple solutions: we train models to predict distributions of solutions and stochastically extract solutions from those distributions. In the case of using a NAR to find multiple single-source shortest path solutions in a weighted graph we demonstrate that this is feasible, particularly on graphs with sizes that lie within the range of sizes we train our models on. For DFS, we see that this task is significantly more challenging, as indicated by prior NAR research. We attribute this result to a number of factors. First, previous work indicates that emulating DFS with NAR is a difficult task. Second, training a model to fit a distribution of solutions does not directly optimize for correct solutions. Solutions must still be extracted from the distribution via stochastic methods, to which the loss metric does not extend (Figure 10). Improving methods for extracting solutions remains an important algorithmic task. Still, with the promise of sampling multiple solutions from a distribution demonstrated, we believe our modifications of the CLRS benchmark act as proof-of-concept for further work. Such work could immediately apply our method to other CLRS benchmark algorithms (e.g. Minimum Spanning Tree, Bipartite Matching, and Task Scheduling). Beyond CLRS, we hope developing NAR with multiple solutions will add essential flexibility to applications such as cyberphysical systems, giving safety and explainability.

Table 3: Mean accuracies and proportions of distinct solutions (0 to 1) when permuting inputs 5 times, with standard deviations over 5 runs.

	Accuracy	
	DFS	Bellman-Ford
(n=5)	0.10 ± 0.03	0.30 ± 0.07
(n=16)	0.01 ± 0.02	0.00 ± 0.00
(n=64)	0.00 ± 0.00	0.00 ± 0.00

	Variety	
	DFS	BF
(n=5)	0.90 ± 0.09	0.62 ± 0.08
(n=16)	1.00 ± 0.00	0.99 ± 0.01
(n=64)	1.00 ± 0.00	1.00 ± 0.00

References

[1] Beatrice Bevilacqua, Kyriacos Nikiforou, Borja Ibarz, Ioana Bica, Michela Paganini, Charles Blundell, Jovana Mitrovic, and Petar Veličković. 2023. Neural Algorithmic Reasoning with Causal Regularisation. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 2272–2288. https://proceedings.mlr.press/v202/bevilacqua23a.html

[2] Montgomery Bohde, Meng Liu, Alexandra Saxton, and Shuiwang Ji. 2024. On the Markov Property of Neural Algorithmic Reasoning: Analyses and Methods. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=Kn7tWhuetn

[3] Dobrik Georgiev, Danilo Numeroso, Davide Bacciu, and Pietro Liò. 2024. Neural Algorithmic Reasoning for Combinatorial Optimisation. arXiv:2306.06064 [cs.NE]

[4] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 1263–1272. https://proceedings.mlr.press/v70/gilmer17a.html

[5] Borja Ibarz, Vitaly Kurin, George Papamakarios, Kyriacos Nikiforou, Mehdi Bennani, Róbert Csordás, Andrew Dudzik, Matko Bošnjak, Alex Vitvitskyi, Yulia Rubanova, Andreea Deac, Beatrice Bevilacqua, Yaroslav Ganin, Charles Blundell, and Petar Veličković. 2022. A Generalist Neural Algorithmic Learner. arXiv:2209.11142 [cs.LG]

[6] S. Kullback and R. A. Leibler. 1951. On Information and Sufficiency. *The Annals of Mathematical Statistics* 22, 1 (1951), 79–86. http://www.jstor.org/stable/2236703

[7] Julian Minder, Florian Grötschla, Joël Mathys, and Roger Wattenhofer. 2023. SALSA-CLRS: A Sparse and Scalable Benchmark for Algorithmic Reasoning. arXiv:2309.12253 [cs.LG] https://arxiv.org/abs/2309.12253

[8] Gleb Rodionov and Liudmila Prokhorenkova. 2024. Neural algorithmic reasoning without intermediate supervision. *Advances in Neural Information Processing Systems* 36 (2024).

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

[10] Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. 2022. The CLRS Algorithmic Reasoning Benchmark. *arXiv preprint arXiv:2205.15659* (2022).
[11] Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. 2020. Neural Execution of Graph Algorithms. arXiv:1910.10593 [stat.ML] https://arxiv.org/abs/1910.10593

A Methodology

Sections A.1.1, A.2.1 discuss our methods for randomising the deterministic (canonical) versions of Depth-First Search and the Bellman-Ford algorithm in order to use the resulting distributions over solutions as inputs to the CLRS baseline. Sections A.1.2 A.2.2 present our methods of extracting candidate solutions from model outputs. Sections A.1.3 A.2.3 give approaches verifying whether candidate solutions are correct solutions. Figures 7, 8, and 9 repeat from the main body for a comprehensive overview, illustrate the steps taken from encoding graphs as distributions to obtaining solutions from NN output.

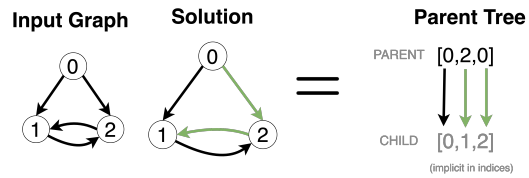


Figure 7: Parent Tree encoding of a single DFS solution. The start node, 0, is its own parent, represented by the value 0 at index 0. The value 2 at index 1 indicates that vertex 2 is the parent of vertex 1. Similarly, the value 0 at index 2 indicates that vertex 0 is the parent of vertex 2.

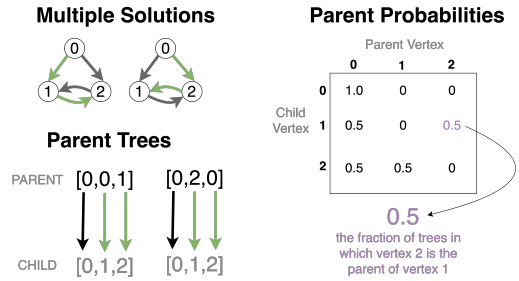


Figure 8: Parent Probability Distribution encoding of multiple DFS solutions. The value 0.5 in row 1 column 2 indicates that vertex 2 is a parent of vertex 1 in 50% of correct parent trees.

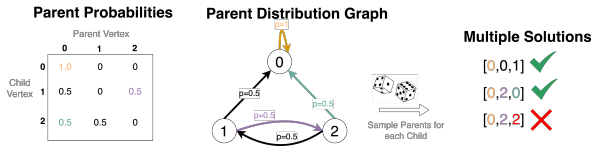


Figure 9: DFS tree extraction from a parent probability matrix: $\tilde{\mathcal{P}}$ or $\mathcal{P}$. The parent tree [0,2,2] is incorrect because there is no solution in which 2 is its own parent. [0,0,0] is incorrect, despite there being solutions in which 0 is the parent of each vertex, because there is no solution in which 0 is the parent of every vertex: if 0 is the parent of 1, 1 must be the parent of 2.

A.1 DFS

We provide a more in-depth discussion of our methodology for DFS described in the main body of the paper.

A.1.1 Randomness in training. Algorithm 1 gives detailed description of the adapted randomized DFS algorithm used in this paper, with the line that randomizes edge-exploration highlighted in red. After running Algorithm 1 20 times, we compute the fraction of solutions in which node s is the parent of node t, which we say is the parent probability, as shown in Figure 3.

Algorithm 1 Randomised Depth-First Search

```
Input: Adjacency Matrix A, vertices V in linear order
Output: Parent Tree π, encoded as list: π[i] := parent of i
Initialize color ← [0] * |V|
Initialize startNode ← V[0]
rootOrdering ← V                                  ▷ ordered restarts
tiebreakOrder ← shuffle(V \ V[0])                 ▷ random edge-exploration
for root in rootOrder do
    if color[root] == 0 then                      ▷ vertex not yet explored
        color[root] ← 1
        π[root] ← root
        current ← root                            ▷ pointer to node being explored
        s_last ← root                             ▷ pointer to node most-recently found
        s_prev[s_last] ← root                     ▷ pointer to node before the node most-recently found
        while True do
            for potentialChild in tiebreakOrder do
                if A[root][potentialChild] != 0 then
                    if color[potentialChild == 0] then
                        color[potentialChild] ← 1
                        π[potentialChild] ← current
                        s_last ← potentialChild
                        break                     ▷ child found, exit for-loop, continue while
                    end if
                end if
            end for
            if current == s_last then             ▷ No outgoing edge found
                if s_prev[s_last] == s_last then  ▷ Done: backtracked source and no new edges
                    break                         ▷ Exit while loop, try further with next root in rootOrder
                end if
                s_last ← s_prev[current]          ▷ Prepare to Backtrack
            end if
            current ← s_last                      ▷ go-deeper if new node found, else backtrack
        end while
    end if
end for
```

A.1.2 Randomness in sampling. We compare 4 methods for extracting valid DFS solutions from a parent distribution, $\tilde{\mathcal{P}}$. First, argmax sampling assumes the parent of each node x to be the node u such that

$$u = \max_{v \in V(G)} \tilde{\mathcal{P}}_{v,u}, \tag{1}$$

where $\tilde{\mathcal{P}}_{v,u}$ is the probability of an edge (v, u) according to $\tilde{\mathcal{P}}$. Argmax sampling is deterministic and can return only one solution. We include it in our discussion as a good baseline which corresponds to the internal mechanism of the CLRS-30 benchmark [10]. To extract multiple solutions for DFS, we introduce stochasticity according to *Upwards Sampling*, which is described in Algorithm 2.

Algorithm 2 Upwards Sampling

```
1:  Input: P̃
2:  Output: A sampled predecessor array π
3:  Initialize π
4:  leaves ← sort(Σ_{v∈V} P̃_{u,v} for all u ∈ V)   ▷ Sort vertices by their probability of being leafs
5:  while leaves ≠ ∅ do
6:      leaf ← leaves[0]  ▷ Take the vertex most likely to be a leaf
7:      π[leaf] ← SamplePredecessor(P̃, leaf)      ▷ Sample π[leaf] according to P̃_leaf
8:      leaves ← leaves.remove(leaf)  ▷ Prevent leaf from being considered as a child again
9:      for v ∈ V do
10:         P̃_{leaf,v} ← 0   ▷ Prevent leaf from being sampled as a parent
11:     end for
12:     leaf ← π[leaf]                            ▷ Make predecessor the next leaf
13:     while π[leaf] is undefined do ▷ Sample the predecessor's predecessor
14:         π[leaf] ← SamplePredecessor(P̃, leaf)
15:         leaves ← leaves.remove(leaf)
16:         for v ∈ V do
17:             P̃_{leaf,v} ← 0
18:         end for
19:         leaf ← π[leaf]
20:     end while
21: end while
```

In Upwards Sampling (Algorithm 2), we first determine how likely each node is to be a parent node of any other node. We do so by summing the probabilities of each vertex $u \in V$ to be the parent of each vertex $v \in V$ and then sorting this list in ascending order, as done in Line 4. In each iteration of the outer loop (Line 5), we begin by taking the vertex with the lowest probability of being a parent and sampling its parent according to $\tilde{\mathcal{P}}_{leaf}$ (Line 7). After having done so, we remove the leaf from future consideration as a leaf and as a parent (as a leaf by definition will not be the parent of another node). Next, we sample the parent of $\pi[leaf]$ until we encounter a node that already has a parent (Line 5). This ensures we do not overwrite previously sampled parents. We have chosen this sampling method as it works similarly to DFS while respecting the inherent stochasticity resulting from returning $\tilde{\mathcal{P}}$ rather than one unambigous solution, as has been done in [10] and [5]. During testing, we encountered the situation that the removal of leaves from consideration as parents leads to no potential parents being available for a node due to having been removed. To address

this problem, we give a node a random parent in this case. We further implement a method without this removal, which we call *AltUpwards Sampling*. While less faithful to graph properties, it addresses the limitations caused by the difficulty of computing the likelihood of being a leaf for each node while operating closer to DFS. It can be seen in Section 4.2 that this method outperforms Algorithm 2.

A.1.3 Checking Validity. We verify whether a given forest F is the result of some depth-first traversal of the graph with ordered restarts (RDFSO). By RDFSO, we mean DFS with randomised tiebreaking and ordered restarts. Randomised tiebreaking means that given two successors v_1, v_2 of a node $v \in G$, the algorithm chooses randomly from which node to continue the depth-first traversal. Ordered restarts means that when all descendants of a given root have been visited by the search, the algorithm continues DFS from the lowest-indexed unvisited node. Since we are not aware of a solution to this problem in the literature, we define it as the randomised DFS recognition problem (RDFSR) and present an algorithm to solve it.

Theorem A.1. *Algorithm 3 accepts if and only if a forest F is the output of some run of randomised DFS with ordered restarts (RDFSO) on G.*

Idea. The algorithm attempts to build a valid DFS traversal of G, using F to inform the traversal order. We derive the traversal order by comparing the descendants of each node in the graph G and forest F. The key fact is that any depth-first traversal from node v visits all unvisited nodes reachable from v before backtracking to v. As a result, **all unvisited nodes reachable from v in graph G are still reachable from v in DFS-forest F**. We use this to rebuild a DFS traversal for F one vertex at a time, visiting next whichever vertex has the same unvisited descendants in G that it does in F.

We present Algorithm 3, which solves DFS verification by rebuilding F by traversing G. Algorithm 3 rebuilds F one vertex at a time, removing each vertex after considering it. Algorithm 3 discovers the next vertex in traversal by calling Algorithm 4.

Algorithm 4, the subroutine called by our main algorithm, finds the next vertex after v in the traversal order by looking for the adjacent vertex to v in F that discovers all its descendants in G. If our most recently visited node m has two children v_1, v_2 in F, Algorithm 4 determines whether traversal can continue first to v_1 or v_2 and still produce F.

Line 2 checks whether the vertex v is indeed a valid next vertex for the traversal. We write $D_G(v)$ to mean the descendants of node v in graph G. Line 2 checks whether $D_G(v) = D_F(v)$.

The loop in Lines 7-18 of Algorithm 4 finds the first vertex v_1 adjacent to v for which $D_G(v_1) = D_F(v_1)$ and recursively repeats this procedure to retrace the run of RDFSO until it reaches the maximum depth in G, removing the vertices discovered during the recursive calls in G and F, thus rendering them unreachable in G from other vertices. It then continues the loop in Lines 7-18 with the remaining descendants of v.

Say there is a vertex v_2 adjacent to v in F from which the depth-first traversal has continued after having terminated its traversal from v_1. After the vertices discovered by v_1 have been removed from G, the sets of vertices reachable from v_2 in F and G must now be the same. This is the case because if $D_G(v_1) \cap D_G(v_2) \neq \emptyset$ before the traversal, then all vertices in $D_G(v_1) \cap D_G(v_2)$ have been removed during the call from v_1 (or during an earlier call) and are now unreachable from v_2 in G. Our algorithm now picks v_2 as the next vertex from which to continue the traversal. Therefore, if Algorithm 4 has not rejected previously, then the set of vertices reachable from v_2 in F and G at a given point could have only been discovered in a traversal from v_2. We thus continue the recursive traversal from v_2, et cetera.

If the algorithm encounters a situation where none of the remaining descendants of a given vertex can reach the same set of reachable vertices in F and G, the algorithm rejects. This is equivalent to the idea that there is no vertex the traversal can continue towards and still result in F. Therefore, the algorithm rejects if there is ever no vertex that discovers all its descendants in G during the run of RDFSO and accepts only if such a vertex exists at every depth of the recursion.

Algorithm 3 RDFSO Verifier

```
Input: G, F
for each vertex i in G do
    if not DVC(G, F, i) then
        return False
    end if
end for
return True
```

Algorithm 4 Descendant Validity Check (DVC)

```
1:  Input: G, F, vertex v
2:  if D_G(v) ≠ D_F(v) then
3:      return False
4:  end if
5:  descendants ← descendants of v in F
6:  Delete v from G and F
7:  for each descendant in descendants do
8:      Compute pd[descendant]          ▷ compute the possible descendants in G
9:      Compute ad[descendant]          ▷ compute the actual descendants in F
10:     if pd[descendant] = ad[descendant] then   ▷ next vertex in traversal found
11:         if not DVC(G, F, descendant) then
12:             return False
13:         end if
14:         restart line 7, with updated descendants ▷ vertices may have been deleted
15:     else
16:         Check next descendant
17:     end if
18: end for
19: return True if descendants is empty. Else return False.
```

A.2 Bellman-Ford

We expand on the presentation of our method as it relates to the Bellman-Ford algorithm.

A.2.1 Randomness in training. We adapt the Bellman-Ford algorithm by randomising the tiebreaking between paths of equal length in the graph. Figure 6 illustrates a situation in which multiple valid Bellman-Ford predecessor arrays are possible. However, the final output of the canonical Bellman-Ford algorithm will only be influenced by such a randomisation procedure if there are multiple minimal-weight paths from the source node s to at least one $v \in V \backslash \{s\}$. As the CLRS-30 benchmark samples Erdős-Rényi graphs with real-valued edge weights, the probability of two paths (s, v) for any $v \in V \backslash \{s\}$ having the same cost is vanishingly low. To this end, we use a smaller set of possible weights such that for each weight $w_{ij} \in \{1, 2, 3\}$, which we normalise for training purposes. As for DFS, we obtain a distribution of parent nodes by re-running the algorithm 20 times and computing the frequency of each parent for each node.

A.2.2 Randomness in sampling. For each $v \in V(G)$, *BF Beamsearch* considers paths from the source to v of length at most n by first sampling candidate predecessors of v and then recursively sampling predecessors of predecessors until the path has length n. At each stage, only the 3 lowest-cost paths are kept and used for the next sampling. As soon as the maximum recursion depth is reached, the predecessor that is at the end of the lowest-cost path from the source is chosen as $\pi[v]$. This mimics the operation of the deterministic Bellman-Ford algorithm while respecting randomness and is thus a suitable, albeit involved, way of sampling π. To sample predecessor arrays more easily from $\mathcal{P}$ and $\tilde{\mathcal{P}}$, we use a greedy randomised algorithm. This algorithm samples a set of parents for each vertex v except the source node (which we assume to be its own parent) according to $\tilde{\mathcal{P}}_v$. In the next step, it checks whether any of the sampled parents are plausible (i.e., if there exists an edge $(p, v) \in E(G)$ for the parent p). It resamples if there are no plausible parents. If any of the sampled parents are plausible, or if the maximum number of resamplings is reached, it chooses the plausible parent p with the lowest weight $w_{p,v}$ as the predecessor $\pi[v]$. Furthermore, we also use the methods of deterministically choosing the likeliest parent of v according to $\tilde{\mathcal{P}}_v$ and randomly uniformly choosing a parent as was done with DFS in order to obtain some baseline results.

A.2.3 Checking Validity. In contrast to DFS, the task of deciding whether a predecessor array sampled from a model output distribution $\tilde{\mathcal{P}}$ is a valid output of running the Bellman-Ford algorithm is straightforward. To this end, we implement a simple algorithm called *Check Valid BF Path*, which is given in Algorithm 5.

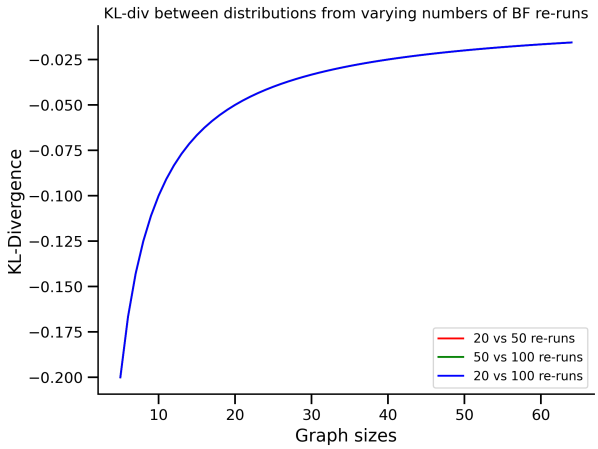
Algorithm 5 simply constructs a graph from the vertices and edges that are implicitly given in π and checks whether all edges correspond to edges in the graph G and whether the costs of the paths in π are the same of the costs when using the deterministic Bellman-Ford algorithm on (G, s).

Algorithm 5 Check Valid BF Path

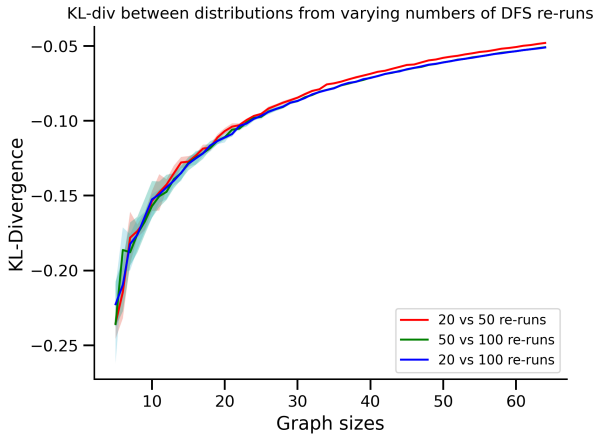
```
Input: Adjacency Matrix A, source vertex s, sampled predecessor array π
Output: Decide whether π is a correct Bellman-Ford solution
G' ← buildGraph(π)  ▷ Construct a graph given by the paths in π
if E(G') ⊈ E(G) then  ▷ Check whether G' contains any nonexistent edges
    Return False
else
    true_costs ← BellmanFord(A, s)  ▷ Compute the costs using the Bellman-Ford algorithm
    model_costs = PathCosts(π)  ▷ Compute costs of paths in π
    Return model_costs == true_costs
end if
```

B Using different numbers of algorithm re-runs

When creating our training data by generating distributions over parents for DFS and Bellman-Ford, we need to consider how often we run the randomised versions of the deterministic algorithms on a graph in order to obtain a good distribution over solutions. To this end, we consider the graph sizes $n \in \{5, \ldots, 64\}$ and generate 100 graphs for each size and run an algorithm (DFS or BF) 20, 50, and 100 times for each graph. For each graph, we then consider the KL-Divergence between each pair of distributions generated by a different number of algorithm re-runs (we compare the distribution generated from 20 runs to the distribution generated from 50 runs and to the one generated from 100 runs, et cetera). We plot the mean KL-Divergence along with its standard deviation for each pairwise comparison between re-run numbers for all considered sizes. In Figure 10, we see that while there are differences between the distributions generated from a varying number of algorithm re-runs, the distribution differences do not change markedly when going from 20 to 100 versus from 50 to 100 re-runs. Therefore, by increasing the number of re-runs, it appears that we do not gain enough information to justify the added computational cost of an increased re-run number. Therefore, we settle on 20 as a reasonable value for the number of algorithm re-runs.

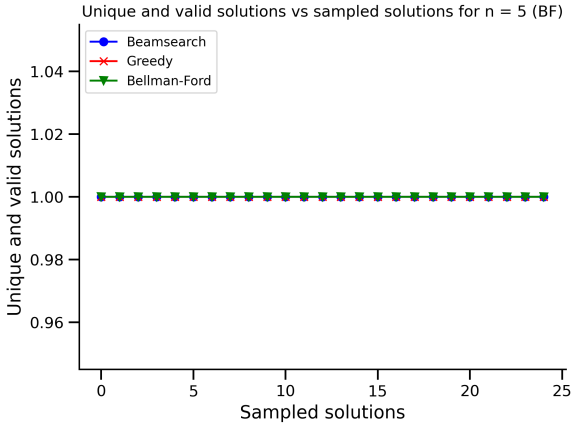


(a) Bellman-Ford

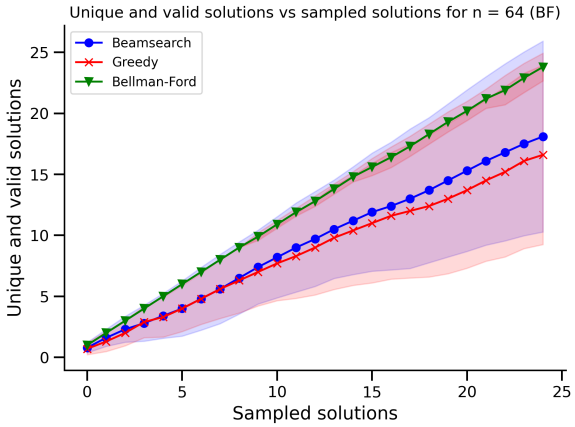


(b) Depth-first search

Figure 10: Pairwise mean distribution differences for DFS and BF for 20, 50, and 100 reruns for graphs of sizes 5 through 64.



(a) 5-vertex Graphs.



(b) 64-vertex Graphs.

Figure 11: The number of valid unique solutions found by our sampling methods on model output distributions and Bellman-Ford on the original graphs for 10 graphs and 25 samples.

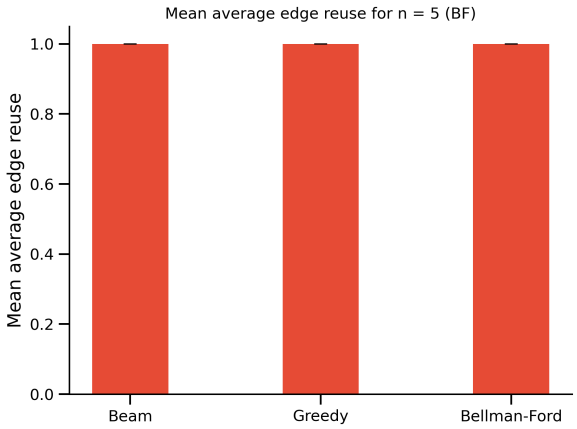
C Validating the model output distributions

Going beyond our analysis of the validity of solutions obtained from the method presented in this paper, we investigate the coverage of the space of all solutions (i.e. the space of possible solutions we can obtain from executing an algorithm on a given graph G) of our methods. Using DFS as an example, we compare the average number of solutions when using DFS on graphs of a given size to the average number of unique and valid solutions when using our sampling method on the outputs of our model on the test data generated from those same graphs. This affords insight into the proportion of the space of possible solutions our method is capable of covering, which is particularly relevant to applications of Neural Algorithmic Reasoning in which using many different solutions is important, such as in cyberphysical systems.

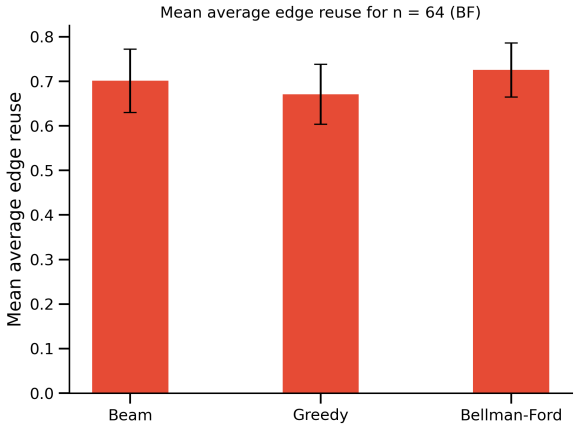
C.1 Validating model output distributions for Bellman-Ford

Figure 11a shows that our sampling methods simply find the one valid solutions that exists for graphs of size 5 in all cases, which validates both our model and the sampling methods. In Figure 11b, we can see that even for out-of-distribution graphs, the model output distributions are good enough for our sampling methods to return a high diversity of valid solutions, even within the margin of error of straightforwardly applying BF to the input graph. This coverage of the solution space is very encouraging for further research into NAR with multiple solutions, as it shows the potential advantage of using NAR to find all possible solutions, which is relevant to many applications, as discussed in the body of the paper.

Figures 12 and 13 show that for both considered graph sizes, the proportions of edges in common among sampled solutions is similar across sampling methods (referred to further as *mean edge reuse*),



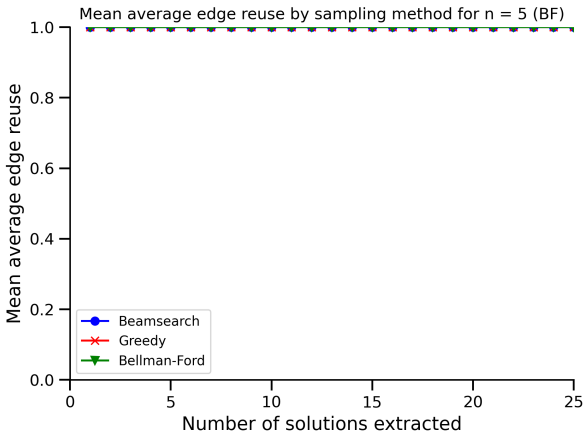
(a) 5-vertex Graphs.



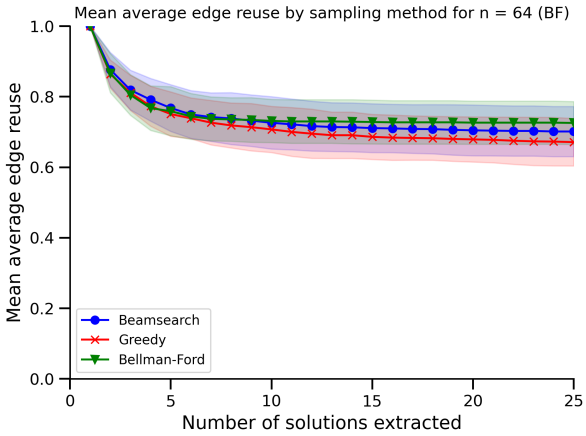
(b) 64-vertex Graphs.

Figure 12: Mean edge reuse for Bellman-Ford sampling.

which is unsurprising given the nature of single-source shortest paths, where only a certain subset of edges is relevant to finding the shortest paths.



(a) 5-vertex Graphs.



(b) 64-vertex Graphs.

Figure 13: Evolution of the mean edge reuse for different Bellman-Ford sampling methods.

C.2 Validating model output distributions for DFS

Figure 14 shows the marked advantage of *AltUpwards* over *Upwards*, as *AltUpwards* finds as many solutions as DFS on small graphs, where *Upwards* struggles to return any diversity of solutions. However, we see in Figure 15 that our methods do not generalise to larger graph sizes, possibly indicating that the chosen encoding of the multiple solutions as probability distributions over predecessors is not suitable for problems with a very diverse set of solutions for a given graph.

D Training Details

D.1 Model architecture and parameters

To give a good baseline, we use the default CLRS Benchmark NAR neural network [10]. Our processor network is an MPNN [4] with a two standard NAR adjustments: namely, gating and triplet reasoning [5]. Our encoder is standard: a linear layer initialised with

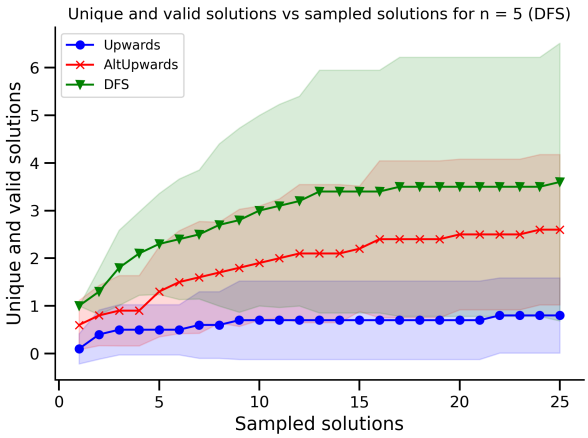


Figure 14: The number of valid unique solutions found by our sampling methods on model output distributions and DFS on the original graphs for 10 graphs and 25 samples.

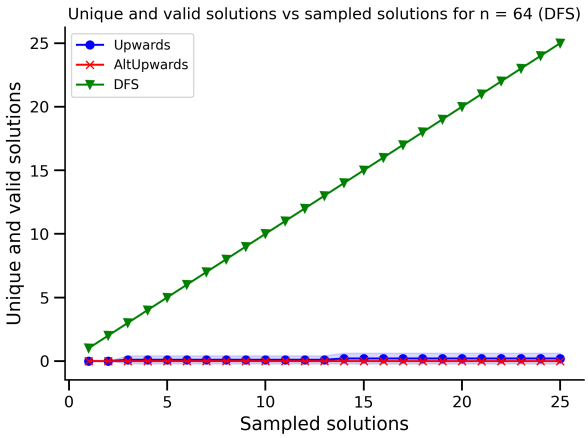


Figure 15: The number of valid unique solutions found by our sampling methods on model output distributions and DFS on the original graphs for 10 graphs and 25 samples.

xavier on scalars. Our decoder is 4 linear layers — three with hidden dimension (128) and a final layer with dimension one. The network is recurrent.

D.2 Time characteristics

We run our experiments using a publicly available compute provider that uses NVIDIA P100 GPUs. Including evaluation of our model every 50 steps, training the NARs for 10,000 steps is very time-inexpensive. We summarize the training times in Table 5.

E The Graph Accuracy Metric

With multiple solutions, node level accuracy is poorly defined. Ordinarily in a single-solution framework, a node is counted as accurately predicted if its prediction matches the true solution: If

Parameter	Value
Processor	triplet_gmpnn
Encoder	xavier_on_scalars
Decoder	4 linear layers
#Parameters	391225
#Hidden Units	128
Learning Rate	0.001
Batch Size	1
Training Graph Sizes	4, 7, 11, 13, 16
Training Set Size	1000
Testing Graph Sizes	5, 64
Testing Set Size	10

Table 4: Training parameters

	$n = 5$	$n = 64$
BF	240sec.	248sec.
DFS	531sec.	550sec.

Table 5: Training times for the conducted experiments in seconds

the true solution is $[0, 1]$ and the reasoner predicts $[1, 1]$, its scores 50% on node accuracy. Put another way, node accuracy scores the fraction of a candidate solution that matches the label solution. In a multiple solution framework, that metric becomes untenable. First, when there are multiple true solutions, producing all the solutions to give partial credit is computationally expensive. Second and more importantly, if $[0, 1]$ and $[1, 0]$ are both true solutions, node accuracy scores $[1, 1]$ at 100% even if it is incorrect. As a result, we use "graph" accuracy, as defined in [7]. If the model predicts $[1, 1]$, and it is not a correct solution, the solution is scored with 0% graph accuracy, regardless of whether $[0, 1]$ or $[1, 0]$ are true solutions. Formally, graph accuracy for multiple solutions scores a solution S as correct for a graph G and algorithm A if and only if $S \leftarrow A(G)$ for some run of A. To give a sense of relative strictness, we provide node level and graph level accuracies for the CLRS benchmark trained for 10,000 steps in Table 6. Interpreting node accuracy as the probability that a NNs prediction is correct for any node, we would expect $graph_{accuracy} = (node_{accuracy})^n$. We can see that the discrepancy between graph accuracy and node accuracy increases with graph size.

F Permuting Inputs

Permuting inputs to produce multiple solutions is a fundamentally different approach than predicting solution distributions. A graph has $n!$ ways of isomorphically labeling the nodes. One might hope that feeding a NAR the same graph with two different node labellings would produce distinct solutions. The results are largely negative. Regardless of what one permutes, NARs rarely predict

Table 6: Node and Graph Accuracy for Bellman-Ford and DFS with regular CLRS training. Network trained for 10,000 steps. Predictions on 64 graphs of each size. Rounded averages over 5 runs.

	Bellman Ford	
	Node	Graph
(n=4)	1.00 ± 0.00	1.00 ± 0.14
(n=16)	0.99 ± 0.00	0.91 ± 0.05
(n=64)	0.98 ± 0.00	0.21 ± 0.08
	DFS	
	Node	Graph
(n=4)	1.00 ± 0.00	1.00 ± 0.00
(n=16)	0.91 ± 0.02	0.41 ± 0.11
(n=64)	0.39 ± 0.01	0.00 ± 0.00

more than one solution, and when they do, the additional solution is usually incorrect.

F.1 Permuting All Inputs

The following experiments generate a permutation σ, and permute each item consistently. So, $\sigma(A_{ij}) = A_{\sigma(i),\sigma(j)}$ for all i, j.

F.1.1 Bellman Ford. For Bellman Ford, CLRS NARs predict on

- A, a weighted adjacency matrix
- s, a one-hot vector encoding the start node
- adj, a copy of A without self loops and with all edge weights set to 1.0
- pos, a vector of floats (useful for other algorithms, fairly irrelevant here)

We report several metrics of BF validity (Table 7). The variables 'ogA, ogS, ogP' stand for the original adjacency matrix, the original start node, and the NN's prediction on the original graph. We obtain 'ogP' by applying the inverse permutation to P, the NN's actual prediction. The variables 'A, S' represent the permuted adjacency matrix and the permuted start node, which are fed to the NN to produce prediction 'P'. Put succinctly, we have $P \leftarrow NN(A, S, adj, pos)$. Table 7 shows that Bellman Ford Prediction validity is the same on permuted and original graphs, as one might expect since shortest paths are invariant to node relabeling.

We also report several metrics of diversity. Diversity in this context refers to whether different solutions involve different node labels in each position: two isomorphic solutions may be counted as different. Second, "Distinctness" tests whether solutions are non-isomorphic, by comparing "ogP". Third, "VD" measures the fraction of valid solutions that are distinct. In Table 7, we see that the network only ever predicts one valid solution on 5 permutations of the same input data.

F.1.2 DFS. We report several metrics of DFS validity (Table 8). As with Bellman Ford, variables "ogA, ogS, ogP" stand for the original adjacency matrix, the original start node, and the NN's prediction on the original graph. The variable "A" represents the permuted adjacency matrix fed to the NN to predict "P". We thus have

$$P \leftarrow NN(A, adj, pos).$$

For DFS, the CLRS benchmark baseline models [10] predict solutions from

- A, a weighted adjacency matrix
- adj, a copy of A without self loops and with all edge weights set to 1.0
- pos, a vector of floats (useful for other algorithms, fairly irrelevant here)

We have two different algorithms for verification: "H" is algorithm 3, and determines whether P is producible by RDFSO. The second algorithm "J" is a relaxed version of "H", determining if P is producible by RDFS with restarts in any order. Table 8 shows that RDFSO prediction validity is hurt by permutation, as one might expect since a correct solution can become incorrect after relabeling (e.g. by starting at any node other than 0). Table 8 distinctness also shows that the NN predicts the same isomorphic solution each time, adapting it to the relabeling. Unfortunately, permuting all the inputs does not yield multiple correct solutions.

Table 7: Bellman Ford mean accuracies and proportions of distinct solutions (0 to 1) when permuting all inputs; 5 permutations, with standard deviations over 5 runs.

	Validity		
	valid(ogA, ogS, ogP)	valid(A,S,P)	valid(ogA,S,P)
(n=4)	1.00 ± 0.00	1.00 ± 0.00	0.32 ± 0.06
(n=16)	0.91 ± 0.05	0.91 ± 0.05	0.00 ± 0.00
(n=64)	0.21 ± 0.08	0.21 ± 0.08	0.00 ± 000
	Variety		
	Diversity	Distinctness	VD
(n=4)	0.61 ± 0.09	0.20 ± 0.00	0.21 ± 0.03
(n=16)	0.99 ± 0.01	0.20 ± 0.00	0.20 ± 0.00
(n=64)	1.00 ± 0.00	0.20 ± 0.00	0.00 ± 0.00

Table 8: DFS mean accuracies and proportions of distinct solutions (0 to 1) when permuting all inputs; 5 permutations, with standard deviations over 5 runs.

	Validity			
	H(ogA, ogP)	H(A,P)	H(ogA,P)	J(ogA,P)
(n=4)	1.00 ± 0.00	0.19 ± 0.14	0.08 ± 0.04	0.28 ± 0.05
(n=16)	0.48 ± 0.12	0.05 ± 0.05	0.01 ± 0.01	0.05 ± 0.04
(n=64)	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	Variety			
	Variety	Distinctness	VD	
(n=4)	0.93 ± 0.05	0.20 ± 0.00	0.38 ± 0.23	
(n=16)	1.00 ± 0.01	0.20 ± 0.00	0.27 ± 0.11	
(n=64)	1.00 ± 0.00	0.20 ± 0.00	0.00 ± 0.00	

Table 11: Node and Graph Accuracy for Bellman-Ford and DFS with CLRS training where algorithms tiebreak in *pos* order. Network trained for ten thousand steps. Predicting on 64 graphs of each size. Standard deviations over 5 runs

	Bellman-Ford	
	Node	Graph
(n=4)	1.00 ± 0.00	1.00 ± 0.00
(n=16)	0.99 ± 0.00	0.88 ± 0.03
(n=64)	0.97 ± 0.00	0.14 ± 0.06
	Depth-First Search	
	Node	Graph
(n=4)	0.58 ± 0.04	0.06 ± 0.01
(n=16)	0.13 ± 0.01	0.00 ± 0.00
(n=64)	0.03 ± 0.01	0.06 ± 0.00

Table 9: Bellman-Ford mean accuracies and proportions of distinct solutions (0 to 1) when permuting the *pos* input; 5 permutations, standard deviations over 5 runs.

	Validity		
	valid(ogA, ogS, ogP)	valid(A,S,P)	valid(ogA,S,P)
(n=4)	1.00 ± 0.00	1.00 ± 0.00	0.33 ± 0.10
(n=16)	0.86 ± 0.03	0.86 ± 0.03	0.00 ± 0.00
(n=64)	0.19 ± 0.03	0.19 ± 0.03	0.00 ± 000

	Variety	
	Distinctness	VD
(n=4)	0.20 ± 0.09	0.20 ± 0.00
(n=16)	0.24 ± 0.01	0.20 ± 0.00
(n=64)	0.75 ± 0.04	0.20 ± 0.00

Table 10: DFS mean accuracies and proportions of distinct solutions (0 to 1) when permuting the *pos* input; 5 permutations, standard deviations over 5 runs.

	Validity			
	H(ogA, ogP)	H(A,P)	H(ogA,P)	J(ogA,P)
(n=4)	0.06 ± 0.01	0.09 ± 0.04	0.01 ± 0.01	0.04 ± 0.01
(n=16)	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
(n=64)	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00

	Variety	
	Distinctness	VD
(n=4)	0.45 ± 0.05	0.28 ± 0.14
(n=16)	1.00 ± 0.00	0.00 ± 0.00
(n=64)	1.00 ± 0.00	0.00 ± 0.00

F.2 Only Permuting the Pos Array

Both Bellman-Ford and DFS networks take an input *pos*. One can think of *pos* as a proxy for node indices (the floats are randomly-spaced but always in ascending order: $pos[0] < pos[1] < ...pos[n]$. One might therefore wonder if training a network to tiebreak according to *pos* might produce better results. However, when we train and predict only permuting *pos*, the models still do not produce multiple correct solutions. For the Bellman-Ford algorithm, the network predicts only one solution (Table 9). For DFS, accuracy is too low to see one correct solution for n=16 graphs, let alone multiple for n=64 (Table 10). A possible reason for this failure is that situations where tiebreaking is necessary are rare, so tiebreaking according to *pos* may be more difficult to learn than tiebreaking according to index. This hypothesis is supported by the lower accuracies in Table 11 than Table 6, despite the same training for the same model architectures: the only difference is Table 11 labels tiebreak according to a permuted *pos*, whereas Table 6 labels tiebreak according to node index.

Received 10 May 2025; revised 14 July 2025; accepted 26 June 2025