KNARsack: Teaching Neural Algorithmic Reasoners to Solve Pseudo-Polynomial Problems

Stjepan Požgaj[1] **Dobrik Georgiev**[2] **Marin Šilić**[1] **Petar Veličković**[3]
[1]University of Zagreb, Faculty of Electrical Engineering and Computing
[2]Graphcore [3]Google DeepMind
{stjepan.pozgaj, marin.silic}@fer.unizg.hr
dobrikg@graphcore.ai petarv@google.com

Abstract

Neural algorithmic reasoning (NAR) is a growing field that aims to embed algorithmic logic into neural networks by imitating classical algorithms. In this extended abstract, we detail our attempt to build a neural algorithmic reasoner that can solve Knapsack, a pseudo-polynomial problem bridging classical algorithms and combinatorial optimisation, but omitted in standard NAR benchmarks. Our neural algorithmic reasoner is designed to closely follow the two-phase pipeline for the Knapsack problem, which involves first constructing the dynamic programming table and then reconstructing the solution from it. The approach, which models intermediate states through dynamic programming supervision, achieves better generalization to larger problem instances than a direct-prediction baseline that attempts to select the optimal subset only from the problem inputs.

1 Introduction

Neural algorithmic reasoning [1, NAR] aims to connect the flexibility of machine learning with the structure and reliability of classical algorithms. Rather than learning purely from input-output pairs, as in most standard machine learning pipelines, NAR models are usually trained under supervision at the algorithm's intermediate computation steps, enabling them to better generalize and reason algorithmically. The CLRS-30 benchmark [2] was introduced to support this paradigm, covering 30 polynomial-time algorithms ranging from sorting to graph algorithms and dynamic programming. However, many important combinatorial optimization problems – including the Knapsack problem [3] – fall outside its scope.

Knapsack is NP-hard. [4] CLRS-30 omits this class of problems, due to the impossibility of generating sufficiently many samples for them (unless NP=co-NP) [5]. However, for Knapsack, there exists a dynamic programming solution whose runtime depends on the numeric value of the capacity rather than its bit-length. This makes Knapsack a pseudo-polynomial problem[1], lifting the implications of Yehuda et al. [5], *provided we do not consider extreme capacities*. Despite its theoretical and practical significance, the problem has not been explored within the NAR framework. Thus, we develop a neural algorithmic reasoner for Knapsack following a two-phase pipeline: dynamic programming table construction and solution reconstruction, as shown in Figure 1. This approach is transferable to other pseudo-polynomial problems, as further discussed in Appendix B.

In general, the contributions of our work can be summarised as follows: 1) we introduce a **two-step construction-reconstruction** NAR approach for pseudo-polynomial problems; 2) we outline several key design choices that enabled generalization to larger problem instances, namely **edge length encoding** and explicitly removing unnecessary input; 3) through results and appendices we provide extensive benchmarking and ablations of our approach giving insight into our architectural decisions.

[1]Informal term meaning the problem admits a pseudo-polynomial algorithm; formally, it is weakly NP-hard.

Požgaj et al., KNARsack: Teaching Neural Algorithmic Reasoners to Solve Pseudo-Polynomial Problems (Extended Abstract). Presented at the Fourth Learning on Graphs Conference (LoG 2025), Hybrid Event, December 10–12, 2025.

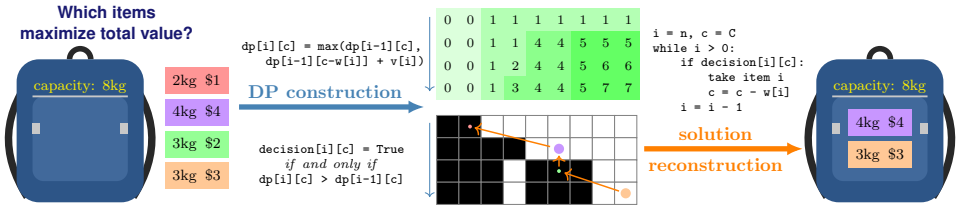


Figure 1: Visualization of the Knapsack problem and its dynamic programming solution. Left: input items with their weights and values. Middle: dynamic programming table and decision table. Right: optimal solution with selected items. `dp[i][c]` – optimal value for the first i items with capacity c.

2 NAR for Knapsack

2.1 Problem Representation

Given n items, each with a weight w_i and value v_i, and a capacity C, we aim to select a subset of the items such that the total weight does not exceed C, whilst maximizing value. The problem is pseudo-polynomial if at least one of the weights or values are integers [6] – we chose the weights to be integers. As current NAR models struggle with unbounded integers, we represent weights with a one-hot encoded input feature, limiting the weight to w_{max}. This is equivalent to lowering the capacity to $C := \min(n \cdot w_{max}, C)$, bringing us in the "non-extreme capacity" case discussed in §1.

For the problem representation, we use the CLRS-30 framework [7]. In it, any algorithm A operates on graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and the execution of A is defined by probes, representing the set of node/edge/-graph inputs ($\mathbb{I}_\mathcal{V}$/$\mathbb{I}_\mathcal{E}$/$\mathbb{I}_\mathcal{G}$), hints ($\mathbb{H}_\mathcal{V}^{(t)}$/$\mathbb{H}_\mathcal{E}^{(t)}$/$\mathbb{H}_\mathcal{G}^{(t)}$), or outputs. The latent node/edge/graph features at timestep t – $\mathbf{X}^{(t)} \in \mathbb{R}^{|\mathcal{V}|\times d}$, $\mathbf{E}^{(t)} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|\times d}$, and $\boldsymbol{g}^{(t)} \in \mathbb{R}^d$, where $d = 128$ – are obtained by encoding hints and inputs using encoders (various functions f below) based on their specific types:

$$\mathbf{X}^{(t)} = f_n^A(\mathbb{I}_\mathcal{V}) + \tilde{f}_n^A(\mathbb{H}_\mathcal{V}^{(t)}) \qquad \mathbf{E}^{(t)} = f_e^A(\mathbb{I}_\mathcal{E}) + \tilde{f}_e^A(\mathbb{H}_\mathcal{E}^{(t)}) \qquad \boldsymbol{g}^{(t)} = f_g^A(\mathbb{I}_\mathcal{G}) + \tilde{f}_g^A(\mathbb{H}_\mathcal{G}^{(t)})$$

After encoding, the features are passed through a processor network and next step hints ($\mathbb{H}_\mathcal{V}^{(t+1)}$/$\mathbb{H}_\mathcal{E}^{(t+1)}$/$\mathbb{H}_\mathcal{G}^{(t+1)}$) are predicted from the latent embeddings. At inference the model uses its own *soft* hint predictions [7, p.5]. For processor's architecture and losses we follow the CLRS-30 framework, unless otherwise stated, but we deviate on how we predict final outputs – see below.

2.2 Pseudo-Polynomial Modeling

We split our training into training the model that constructs the DP table, and training the model to reconstruct the solution based on the input and the DP table generated by the first. We initially attempted a unified approach but found it too unstable to train, resulting in poor inference performance.

Dynamic Programming Table Construction. The dynamic programming state $\mathrm{dp}_{i,c}$ represents the maximum value achievable with the first i items and a knapsack capacity of c. A $\mathrm{decision}_{i,c}$ variable indicates whether the i-th item is included in the optimal solution $\mathrm{dp}_{i,c}$, as shown in Figure 1. As the DP table is constructed row by row, we chose $\mathcal{V} = \{u_0, \dots u_C\}$ for each $t \in \{1, \dots, n\}$. At timestep t we set $\mathbb{I}_\mathcal{G} = \{w_t, v_t\}$. Then the model predicts $\mathbb{H}_\mathcal{V}^{(t)} = \{\mathrm{dp}_{t,:}, \mathrm{decision}_{t,:}\}$. The decision probability table is obtained by stacking the decision row predictions $\mathrm{decision}_{t,:}$ at each step.

Edge length encoding was a crucial ingredient for successful training of the NAR constructor:

$$\mathbb{I}_\mathcal{E} = \mathbb{I}_\mathcal{E} \cup \{\mathrm{EL}(\mathcal{E})\} \qquad \mathrm{EL}(\mathcal{E})_{ijm} = \delta_{m,min(|i-j|,M-1)} \qquad \mathrm{EL} : \mathcal{P}(\mathcal{E}) \to \{0,1\}^{|\mathcal{V}|\times|\mathcal{V}|\times M}$$

where δ is the Kronecker delta function and $M = 10$ is the cutoff. Such categorical encoding gives the model the ability to choose the correct past states that influence the current one. Edge length encoding is illustrated in Figure 2. Figure 3 shows the impact of EL on an in-distribution problem instance in comparison to standard random node positional encoding. For more details, see Appendix F.

We additionally observe that the CLRS-30's processors struggle to handle the Knapsack DP scalar values larger than those seen in training. We propose to mitigate this issue by adopting a *homogeneous*

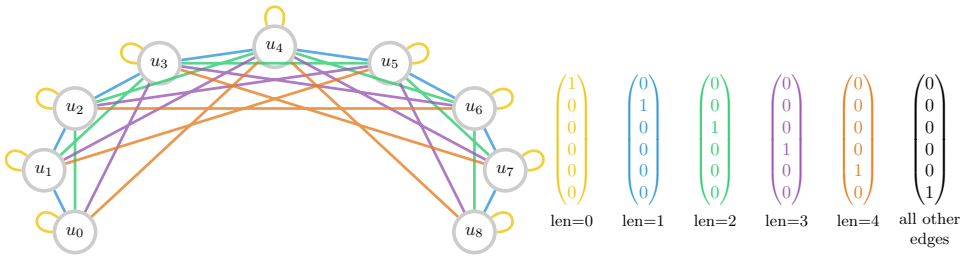
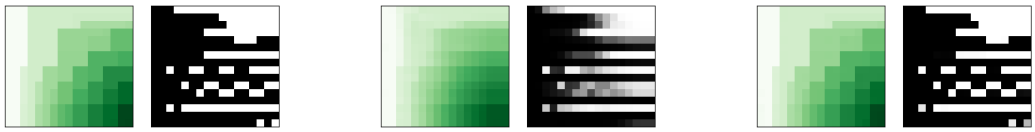


Figure 2: Categorical edge length encoding for the NAR construction model. Although we use $M = 10$ in the implementation, here, due to visualization constraints, the case with $M = 6$ is shown.



(a) True DP tables **(b)** No edge length encoding **(c)** With edge length encoding

Figure 3: Comparison of constructed DP tables for $n = 16$, $C = 16$ (in-distribution). Using categorical edge length encoding sharpens the predicted tables and enables reasoning.

processor [8], which enforces the model to be invariant to the scale of the item values. This substantially improves the value generalization performance of the model on instances larger than those seen in training. Appendix G provides more results and a discussion.

Solution Reconstruction. Based on problem input and the predicted decision table, the solution is reconstructed by iteratively selecting the items that are part of the optimal set, as illustrated in Figure 1. Unlike in the construction phase, in order to encode information about all items and the entire decision table, $\mathcal{V} = \{u_0, \ldots, u_C\} \cup \{u_{C+1}, \ldots C+n\}$ where the second set are the item nodes.

$$\mathbb{I}_{\mathcal{V}} = \{\boldsymbol{p}, \boldsymbol{w}, [\underbrace{0, \ldots, 0}_{C+1}, \underbrace{1, \ldots, 1}_{n}]\} \qquad \mathbb{I}_{\mathcal{E}} = \bigcup_{(c,i)\in\mathcal{E}} \{\text{decision}_{ic}\} \cup \bigcup_{(c,c')\in\mathcal{E}} \{EL(\mathcal{E})_{cc'}\}$$

where $\boldsymbol{p}$ and $\boldsymbol{w}$ are the vectors of positional encodings and weights (capacity nodes are padded with 0), $0 \leq c, c' \leq C$ and $C + 1 \leq i \leq C + n$ and $\mathbb{I}_{\mathcal{G}} = \emptyset$. To model the reconstruction process, the hints provide information on the item currently under consideration, the total remaining capacity, and the set of items selected up to that point. For each item, the model outputs the probability that it belongs to the solution. Due to the space constraints, details can be found in Appendix E.

To reconstruct the solution from the decision table, it is sufficient to have information about the item weights. During reconstruction, as shown in Figure 4, *having item values leads to worse generalization*, as the model is trying to find a shortcut to the solution, without actually learning to navigate the decision table. We hypothesise that this is the reason why the joint model approach did not work. We also note that *having two alternating steps*, one for decision and one for moving the pointer works better than combining the two. Further details are provided in Figure 8, Appendix E.

Deterministic Reconstruction. An alternative would be to perform reconstruction based on the decision probability table, as it allows for end-to-end differentiability. In Appendix H, we describe in more detail such a differentiable deterministic reconstruction. Sadly, our initial experiments in training an NAR construction, coupled with deterministic reconstruction, were unstable. Nevertheless, in the evaluation, we include as a baseline a model that, during the inference phase, performs deterministic reconstruction based on the decision table generated by NAR construction model.

3 Evaluation

Experimental Setup. Our experiments are implemented on top of CLRS-30 [7] (3 seeds for std). Hyperparameters (learning rate, optimiser, etc.) are inherited, including the use of Triplet-GMPNN

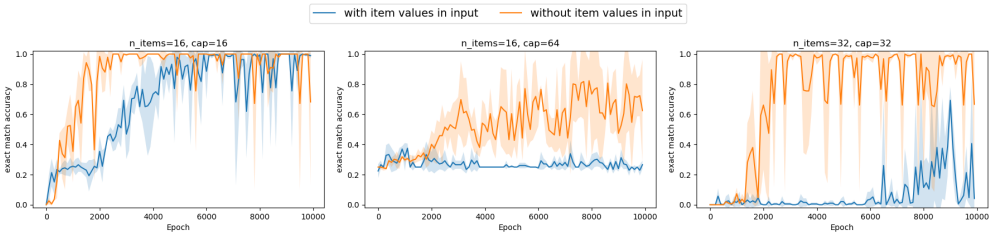


Figure 4: Comparison of NAR reconstruction performance with and without item values in the input, given the true decision table during both training and inference. Having the items' values in the input prevents the NAR reconstruction model from generalizing to larger instances.

Table 1: In- and out- of distribution performance for different models. Best results are **boldfaced**, but we separate models using an oracle in their execution. Best viewed on screen.

n	C	no-hint baseline		reg. c. + reg. r.		homo. c. + reg. r.		homo. c. + det. r.		Tropical Attention [11]	
		micro-F1	exact	micro-F1	exact	micro-F1	exact	micro-F1	exact	micro-F1	exact
16	16	$0.889_{\pm 0.017}$	$0.385_{\pm 0.101}$	$\mathbf{0.989}_{\pm 0.004}$	$\mathbf{0.906}_{\pm 0.031}$	$0.985_{\pm 0.007}$	$0.901_{\pm 0.033}$	$0.975_{\pm 0.004}$	$0.891_{\pm 0.013}$	$0.909_{\pm 0.013}$	$0.381_{\pm 0.005}$
16	64	$\mathbf{0.943}_{\pm 0.000}$	$0.250_{\pm 0.000}$	$0.785_{\pm 0.063}$	$0.438_{\pm 0.095}$	$0.917_{\pm 0.017}$	$\mathbf{0.495}_{\pm 0.115}$	$\mathbf{0.957}_{\pm 0.012}$	$\mathbf{0.578}_{\pm 0.109}$	$0.892_{\pm 0.011}$	$0.493_{\pm 0.027}$
32	32	$0.656_{\pm 0.122}$	$0.000_{\pm 0.000}$	$0.854_{\pm 0.015}$	$0.177_{\pm 0.039}$	$\mathbf{0.925}_{\pm 0.040}$	$\mathbf{0.510}_{\pm 0.173}$	$\mathbf{0.933}_{\pm 0.034}$	$\mathbf{0.521}_{\pm 0.128}$	$0.923_{\pm 0.012}$	$0.186_{\pm 0.078}$
64	16	$0.835_{\pm 0.030}$	$0.031_{\pm 0.031}$	$0.824_{\pm 0.055}$	$0.135_{\pm 0.127}$	$\mathbf{0.939}_{\pm 0.011}$	$\mathbf{0.615}_{\pm 0.110}$	$\mathbf{0.949}_{\pm 0.015}$	$\mathbf{0.635}_{\pm 0.102}$	$0.822_{\pm 0.024}$	$0.043_{\pm 0.047}$
64	64	$0.543_{\pm 0.000}$	$0.000_{\pm 0.000}$	$0.448_{\pm 0.084}$	$0.000_{\pm 0.000}$	$\mathbf{0.668}_{\pm 0.087}$	$0.000_{\pm 0.000}$	$\mathbf{0.770}_{\pm 0.115}$	$0.000_{\pm 0.000}$	$0.374_{\pm 0.000}$	$0.000_{\pm 0.000}$

processor [9] as well as the standard node positional encoding. Differently from CLRS-30, we use the model from the final epoch for testing, as it was recommended in [10]. Furthermore, during the construction training phase, we perform 30,000 epochs, as we observed that the model does not stabilize within the first 10,000 epochs.

The model is trained on samples where $n \leq 16$ and $C \leq 16$ (in-distribution). For out-of-distribution experiments, we consider different values of n and C (up to $n = 64$ and $C = 64$). Each configuration consists of 64 samples. Item weights are uniformly sampled from $\{1, 2, \ldots, 8\}$, and item values are uniformly sampled from the interval $[0, 1]$.

Following CLRS-30, during testing, we report the micro-F1 score. We also add *exact-match accuracy* – the fraction of samples whose discretized solution, obtained by greedily selecting the highest-probability item until capacity is reached, exactly matches the ground truth.

Models and Baselines. Alongside the models with regular processors (**reg. c. + reg. r.**), and a homogeneous processor during construction (**homo. c. + reg. r.**), we evaluate a **no-hint baseline**, which directly predicts the optimal subset from the input. It is implemented using the *no-hint* mode in CLRS-30 with $2n$ message-passing steps, and includes scalar features for item weights and capacity. We additionally zero-shot evaluate the homogeneous construction processor with the deterministic reconstruction baseline (**homo. c. + det. r.**), despite the fact they were not trained together. We also compare our models to Tropical Attention [11], a recent direct-prediction model for combinatorial optimization problems based on tropical projective geometry and polyhedral geometric inductive bias. To make the comparison, we adjusted the sampler and the training and test sizes to match ours, see Appendix C. For more information on this and other relevant works, see Appendix A.

3.1 Results

Results are presented in Table 1. Our first observation is that the combination of two regular processors does not obtain higher F1-scores than the baseline and even falls short on the most extreme OOD (64, 64) test set. Despite that, exact-match accuracy for (**reg. c. + reg. r.**) is often significantly higher than the baseline. The results suggest that: 1) the construction-reconstruction pipeline works differently than the baseline and 2) node-level metrics should often be combined with instance-level ones [10].

The second result we want to highlight is the increase in performance when using a homogeneous NAR constructor. In the larger OOD regimes (**homo. c. + reg. r.**) significantly outperforms the baseline F1 scores and achieves best exact-match accuracy from the fully-neural models. According to our observations, even if the input value scales remain fixed across test distributions, homogeneity

Table 2: Proportions of DP table entries satisfying three fundamental properties under ε=0.01: (1) item-wise monotonicity: $\mathrm{dp}_{i,c} \geq \mathrm{dp}_{i-1,c} - \varepsilon$; (2) capacity-wise monotonicity: $\mathrm{dp}_{i,c} \geq \mathrm{dp}_{i,c-1} - \varepsilon$; (3) optimal substructure: $\mathrm{dp}_{i,c} \geq \max(\mathrm{dp}_{i-1,c}, \mathrm{dp}_{i-1,c-w_i} + v_i) - \varepsilon$.

n	C	item-wise mon.		capacity-wise mon.		optimal substructure	
		reg. c.	homo. c.	reg. c.	homo. c.	reg. c.	homo. c.
16	16	$0.893_{\pm 0.032}$	$\mathbf{0.974}_{\pm 0.009}$	$\mathbf{0.951}_{\pm 0.027}$	$0.945_{\pm 0.036}$	$0.700_{\pm 0.057}$	$\mathbf{0.948}_{\pm 0.015}$
16	64	$0.915_{\pm 0.049}$	$\mathbf{0.988}_{\pm 0.004}$	$\mathbf{0.892}_{\pm 0.035}$	$0.799_{\pm 0.062}$	$0.377_{\pm 0.070}$	$\mathbf{0.677}_{\pm 0.043}$
64	16	$0.711_{\pm 0.014}$	$\mathbf{0.978}_{\pm 0.003}$	$0.975_{\pm 0.010}$	$\mathbf{0.981}_{\pm 0.014}$	$0.548_{\pm 0.031}$	$\mathbf{0.967}_{\pm 0.006}$
32	32	$0.773_{\pm 0.028}$	$\mathbf{0.983}_{\pm 0.004}$	$\mathbf{0.967}_{\pm 0.009}$	$0.967_{\pm 0.018}$	$0.447_{\pm 0.060}$	$\mathbf{0.901}_{\pm 0.028}$
64	64	$0.636_{\pm 0.022}$	$\mathbf{0.981}_{\pm 0.005}$	$\mathbf{0.860}_{\pm 0.031}$	$0.841_{\pm 0.064}$	$0.241_{\pm 0.032}$	$\mathbf{0.715}_{\pm 0.036}$

could be one of the requirements when modelling pseudo-polynomial DP problems, since the magnitude of the solution increases with size (Appendix G).

The best results were achieved with the (**homo. c. + det. r.**) combination. Although we did observe improvements over (**homo. c. + reg. r.**) in both metrics, results were comparable. Our conclusions are: 1) **perfect** DP construction execution might be necessary and 2) our neural reconstruction behaves similarly to the deterministic one.

Our best fully neural model (**homo. c. + reg. r.**) achieves better performance than Tropical Attention in all OOD settings, but it is important to interpret these results in the context of the differences between approaches. We note that Tropical Attention's primary aim, like ours, is not to compete with traditional or neural Knapsack solvers, but rather to demonstrate the effectiveness of their proposed architecture. We further hypothesize that combining Tropical Attention's insights with ours – which are largely orthogonal – could yield even stronger performance.

3.2 Properties of Generated Scalar DP Tables

In Table 5, Appendix G, we report qualities of the generated decision tables, but – aside from a few visual examples – we do not quantify the quality of the predicted scalar DP tables. Because mean squared error is not very interpretable for this purpose, we perform an analysis based on three statistics that correspond to the fundamental properties of the Knapsack DP table. To avoid issues with numerical precision of predicted values, we use $\varepsilon = 0.01$, which is sufficiently precise for gaining insights into the constructed DP tables, given that item values are sampled from $[0, 1]$. These statistics measure the proportion of DP table cells for which the specified constraints hold.

The results in Table 2 show that the homogeneous processor model produces significantly better tables in terms of item-wise monotonicity and optimal substructure property, while capacity-wise monotonicity results are comparable. The results suggest that capacity generalization presents a greater challenge than item count generalization when predicting DP scalars. While the optimal substructure property was violated 30% of the time for our largest OOD experiment, the 46% improvement was sufficient to improve our downstream predictions. Note that there exist other alternative approaches [12], that provably generalise to *any* input size, but those mimic the algorithms too closely, trading the flexibility of neural networks, which is one of the reasons NAR exists.

Limitations and Future Work. In spite of the solid results, there are limitations remaining to be addressed: 1) **Oracle instability** – we were unable to train (**homo. c. + det. r.**), due to exploding/vanishing gradients. We hypothesise this is due to the algorithm chaining multiplications of probabilities and/or operating in scalar space (Appendix H). 2) **Separated trainings** – currently, the training of the construction model is completely decoupled from the training of the reconstruction one. This is not common for NAR models and is a topic of our future interests. Looking ahead, we also plan to investigate how the observations made in this work, such as the use of edge length encoding, could be applied to problems from CLRS-30, shedding light on their potential beyond the current setting.

Acknowledgements

The authors would like to thank Federico Barbero (Google DeepMind), Simon Osindero (Google DeepMind), and Ndidi Elue (Google DeepMind) for their valuable comments and suggestions on this work. S.P. thanks Pietro Liò for hosting him in his research group at University of Cambridge during part of this work.

References

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

[2] Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. The clrs algorithmic reasoning benchmark. In *International Conference on Machine Learning*, pages 22084–22102. PMLR, 2022. 1, 8, 9, 13

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition, 2009. ISBN 978-0-262-03384-8. 1

[4] Richard M Karp. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*, pages 219–241. Springer, 2009. 1

[5] Gal Yehuda, Moshe Gabel, and A. Schuster. It's not what machines can learn, it's what we cannot teach. *International Conference on Machine Learning*, 2020. 1

[6] Dominik Wojtczak. On strong np-completeness of rational problems. In *International Computer Science Symposium in Russia*, pages 308–320. Springer, 2018. 2

[7] Borja Ibarz, Vitaly Kurin, George Papamakarios, Kyriacos Nikiforou, Mehdi Bennani, Róbert Csordás, Andrew Joseph Dudzik, Matko Bošnjak, Alex Vitvitskyi, Yulia Rubanova, et al. A generalist neural algorithmic learner. In *Learning on graphs conference*, pages 2–1. PMLR, 2022. 2, 3, 14

[8] Hao Tang, Zhiao Huang, Jiayuan Gu, Bao-Liang Lu, and Hao Su. Towards scale-invariant graph-related problem solving by iterative homogeneous gnns. *Advances in Neural Information Processing Systems*, 33:15811–15822, 2020. 3, 8, 14

[9] Andrew Joseph Dudzik and Petar Veličković. Graph neural networks are dynamic programmers. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=wu1Za9dY1GY. 4, 8

[10] Sadegh Mahdavi, Kevin Swersky, Thomas Kipf, Milad Hashemi, Christos Thrampoulidis, and Renjie Liao. Towards better out-of-distribution generalization of neural algorithmic reasoning tasks. *arXiv preprint arXiv:2211.00692*, 2022. 4, 8

[11] Baran Hashemi, Kurt Pasque, Chris Teska, and Ruriko Yoshida. Tropical attention: Neural algorithmic reasoning for combinatorial algorithms. *arXiv preprint arXiv:2505.17190*, 2025. 4, 8

[12] Gleb Rodionov and Liudmila Prokhorenkova. Discrete neural algorithmic reasoning. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=Inrv8EXylW. 5

[13] Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SkgKO0EtvS. 8

[14] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? *arXiv preprint arXiv:1905.13211*, 2019. 8

[15] Keyulu Xu, Mozhi Zhang, Jingling Li, Simon Shaolei Du, Ken-Ichi Kawarabayashi, and Stefanie Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=UH-cmocLJC. 8

[16] Julian Minder, Florian Grötschla, Joël Mathys, and Roger Wattenhofer. SALSA-CLRS: A sparse and scalable benchmark for algorithmic reasoning. In *The Second Learning on Graphs Conference*, 2023. URL https://openreview.net/forum?id=PRapGjDGFQ. 8

[17] Heiko Strathmann, Mohammadamin Barekatain, Charles Blundell, and Petar Veličković. Persistent message passing. In *ICLR 2021 Workshop on Geometrical and Topological Representation Learning*, 2021. URL https://openreview.net/forum?id=HhOJZT--N23. 8

[18] Cameron Diao and Ricky Loynd. Relational attention: Generalizing transformers for graph-structured tasks. *arXiv preprint arXiv:2210.05062*, 2022. 8

[19] Artur Back de Luca, George Giapitzakis, Shenghao Yang, Petar Veličković, and Kimon Fountoulakis. Positional attention: Expressivity and learnability of algorithmic computation. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=0IJQD8zRXT. 8

[20] Kaijia Xu and Petar Veličković. Recurrent aggregators in neural algorithmic reasoning. In *The Third Learning on Graphs Conference*, 2024. URL https://openreview.net/forum?id=TcQZGWWEeg. 8

[21] Jonas Jürß, Dulhan Hansaja Jayalath, and Petar Veličković. Recursive algorithmic reasoning. In *Learning on Graphs Conference*, pages 5–1. PMLR, 2024. 8

[22] Rishabh Jain, Petar Veličković, and Pietro Liò. Neural priority queues for graph neural networks. *arXiv preprint arXiv:2307.09660*, 2023. 8

[23] Gleb Rodionov and Liudmila Prokhorenkova. Neural algorithmic reasoning without intermediate supervision. *Advances in Neural Information Processing Systems*, 36:51663–51674, 2023. 8

[24] Dobrik Georgiev, Joseph Wilson, Davide Buffelli, and Pietro Liò. Deep equilibrium algorithmic reasoning. *Advances in Neural Information Processing Systems*, 37:33638–33667, 2024.

[25] Beatrice Bevilacqua, Kyriacos Nikiforou, Borja Ibarz, Ioana Bica, Michela Paganini, Charles Blundell, Jovana Mitrovic, and Petar Veličković. Neural algorithmic reasoning with causal regularisation. In *International Conference on Machine Learning*, pages 2272–2288. PMLR, 2023.

[26] Hefei Li, Peng Chao, Chenyang Xu, and Zhengfeng Yang. Open-book neural algorithmic reasoning. *Advances in Neural Information Processing Systems*, 37:9819–9836, 2024. 8

[27] Petar Veličković. Everything is connected: Graph neural networks. *Current Opinion in Structural Biology*, 79:102538, 2023. ISSN 0959-440X. doi: https://doi.org/10.1016/j.sbi.2023.102538. URL https://www.sciencedirect.com/science/article/pii/S0959440X2300012X. 8

[28] Dobrik Georgiev Georgiev, Danilo Numeroso, Davide Bacciu, and Pietro Liò. Neural algorithmic reasoning for combinatorial optimisation. In *Learning on Graphs Conference*, pages 28–1. PMLR, 2024. 8

[29] Yu He and Ellen Vitercik. Primal-dual neural algorithmic reasoning. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=iBpkzB5LEr. 8

[30] Christoph Hertrich and Martin Skutella. Provably good solutions to the knapsack problem via neural networks of bounded size. *INFORMS journal on computing*, 35(5):1079–1097, 2023. 8

[31] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016. 8

[32] Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in Neural Information Processing Systems*, 33:21188–21198, 2020. 8

[33] Reza Refaei Afshar, Yingqian Zhang, Murat Firat, and Uzay Kaymak. A state aggregation approach for solving knapsack problem with deep reinforcement learning. In *Asian Conference on Machine Learning*, pages 81–96. PMLR, 2020. 8

[34] Zhenfu Zhang, Haiyan Yin, Liudong Zuo, and Pan Lai. Reinforcement learning for solving the knapsack problem. *Computers, Materials & Continua*, 84(1), 2025. 8

[35] Alex Schutz, Victor-Alexandru Darvariu, Efimia Panagiotaki, Bruno Lacerda, and Nick Hawes. Tackling gnarly problems: Graph neural algorithmic reasoning reimagined through reinforcement learning. *arXiv preprint arXiv:2509.18930*, 2025. 8

[36] Darko Drakulic, Sofia Michel, Florian Mai, Arnaud Sors, and Jean-Marc Andreoli. Bq-nco: Bisimulation quotienting for efficient neural combinatorial optimization. *Advances in Neural Information Processing Systems*, 36:77416–77429, 2023. 8

[37] Hazem AA Nomer, Khalid Abdulaziz Alnowibet, Ashraf Elsayed, and Ali Wagdy Mohamed. Neural knapsack: A neural network based solver for the knapsack problem. *IEEE access*, 8: 224200–224210, 2020. 8

[38] Vladimir V Mirjanic, Razvan Pascanu, and Petar Veličković. Latent space representations of neural algorithmic reasoners. In *The Second Learning on Graphs Conference*, 2023. 14

[39] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. URL https://arxiv.org/abs/1607.06450. 14

A Related Work

Neural algorithmic reasoning aligns learned computation with classical algorithms, with the aim of obtaining trained neural models that can execute the target algorithm in the out-of-distribution regime. Intermediate-step supervision [2, 13], also utilised here, has been the initial approach to achieve such alignment. However, this alone is not sufficient and architectural modifications were necessary. Early analyses studied what neural networks can reason about and how they extrapolate [14, 15] – on an informal level their findings state that neural networks extrapolate well if the algorithm can be separated into subfunctions and each of them is easily learnable by a corresponding neural submodule. This, combined with the rise of standardised NAR benchmarks (CLRS-30 [2] and SALSA-CLRS [16]) gave rise to many subsequent works, proposing architectural alignment strategies such as homogeneity [8], persistent message passing [17], relational/positional attention [18, 19], and recurrent aggregators [20]. Related efforts explore structured memory for recursion and data structures [21, 22], while others investigate reducing or re-structuring hints [10, 23–26].

Graph neural networks [27], the underlying architecture behind the NAR processors, are theoretically aligned to dynamic programming algorithms [9]. However, when NAR is benchmarked, most studies address polynomial-time DPs, and the 0–1 Knapsack algorithm, being pseudo-polynomial (weakly NP-hard), was deliberately omitted from CLRS-30. While some works have addressed NP-hard problems [28, 29], none have focused specifically on pseudo-polynomial problems. Outside of the field of NAR, there exist DP-inspired neural networks [30] that provably solve Knapsack, but their weights are hardcoded, rather than learned from data, as is the case in our work. Due to the nature of our problem, classical neural combinatorial optimisation (NCO) approaches could also be applied. Some possible approaches are Bello et al. [31], Kwon et al. [32], Afshar et al. [33], Zhang et al. [34], but these are fundamentally different from our approach. Moreover, these approaches represent item values and weights as real numbers and use fixed capacity. Additionally, they do not consider out-of-distribution settings at all – the instance size is fixed and ranges up to several hundred items. It is worth noting that the idea of combining NAR and RL is very recent [35], and there are no approaches that would tackle pseudo-polynomial algorithms in this manner.

From the NCO approaches, our work is most conceptually similar to supervised methods that solve Knapsack by learning the input-output mapping directly, without following the algorithm's intermediate steps [11, 36, 37]. One supervised approach [36], which uses the same sampler as Kwon et al. [32], does consider OOD, but trains on sizes of 100 and tests on 200, 500, and 1000, which is not comparable to our setting. Another supervised learning approach for solving Knapsack [37] does not have available source code and reproducibility is not at the required level. The only work we found that is comparable to ours in terms of the training and testing scales is Tropical Attention [11]. Tropical Attention [11] introduces a solution based on tropical projective geometry and polyhedral geometric inductive bias. The Tropical Transformers enhance OOD performance in both length and value generalization on several combinatorial optimization problems. In the Knapsack experiments, the model is trained for $n = 8$ and tested for length generalization to $n = 64$. Capacity generalization is not performed – capacity is integer from 10 to 20, in both cases. The model has much shorter training time, primarily due to the lightweight implementation and the fact that it does not use hints, but rather performs supervised direct-prediction of outputs based on inputs. The source code for Tropical Attention is available, so we adapted the sampler and training and test sizes to compare it with our approach.

B Other Pseudo-Polynomial Problems

Table 3: In- and out- of distribution performance for the Subset Sum problem.

n_{numbers}	target	no-hint baseline		homo. c. + reg. r.	
		micro-F1	exact	micro-F1	exact
16	16	$0.770_{\pm 0.014}$	$0.354_{\pm 0.024}$	$\mathbf{1.000}_{\pm 0.000}$	$\mathbf{1.000}_{\pm 0.000}$
16	64	$\mathbf{0.886}_{\pm 0.000}$	$\mathbf{0.266}_{\pm 0.000}$	$0.621_{\pm 0.121}$	$0.255_{\pm 0.018}$
32	32	$0.406_{\pm 0.094}$	$0.365_{\pm 0.036}$	$\mathbf{0.854}_{\pm 0.135}$	$\mathbf{0.542}_{\pm 0.423}$
64	16	$0.666_{\pm 0.056}$	$0.380_{\pm 0.055}$	$\mathbf{0.993}_{\pm 0.008}$	$\mathbf{0.990}_{\pm 0.009}$
64	64	$0.353_{\pm 0.000}$	$\mathbf{0.141}_{\pm 0.000}$	$\mathbf{0.530}_{\pm 0.106}$	$0.094_{\pm 0.041}$

Table 4: In- and out- of distribution performance for the Partition problem.

n_{numbers}	sum	no-hint baseline		homo. c. + reg. r.	
		micro-F1	exact	micro-F1	exact
8	32	$0.813_{\pm 0.007}$	$0.406_{\pm 0.027}$	$\mathbf{0.996}_{\pm 0.007}$	$\mathbf{0.990}_{\pm 0.018}$
16	64	$0.093_{\pm 0.154}$	$\mathbf{0.063}_{\pm 0.000}$	$\mathbf{0.698}_{\pm 0.027}$	$0.042_{\pm 0.036}$
24	96	$0.037_{\pm 0.064}$	$0.000_{\pm 0.000}$	$\mathbf{0.578}_{\pm 0.088}$	$0.000_{\pm 0.000}$
32	128	$0.030_{\pm 0.052}$	$0.000_{\pm 0.000}$	$\mathbf{0.572}_{\pm 0.123}$	$0.000_{\pm 0.000}$

While this work focuses exclusively on the Knapsack problem, the underlying approach is readily transferable to other pseudo-polynomial problems. Both the Subset Sum and Partition problem can be directly reduced to the Knapsack formulation, which allowed us to apply our method with only minimal modifications to the sampler. The results for these problems are reported in Table 3 and Table 4. Even without any problem-specific tuning, our model tended to generalize better than the baseline in most cases. Extending the approach to other pseudo-polynomial problems – such as Minimum Coin Exchange or Rod Cutting – would require adjustments to the problem representation. Nevertheless, the core methodology can be applied without substantial changes.

C Additional Details on the Experimental Setup

Training and Testing Scales. Training on sizes up to 16 with OOD testing on size 64 is the standard for the CLRS-30 benchmark [2] and is therefore followed by the vast majority of NAR works. Since we ultimately aim to include pseudo-polynomial problems in the benchmark, this emerged as the most natural choice. Given that in our reconstruction model the number of graph nodes equals $N + C + 1$, training on $(n = 32, C = 32)$ is already not memory-feasible within the CLRS-30 framework. If we consider node accuracy alone, $4\times$ generalization is gradually becoming obsolete as OOD test, e.g., for the Bellman-Ford algorithm. However, from our baselines (the default models in CLRS-30), we can see that this is not the case for Knapsack. Additionally, in our case we have two parameters that determine the problem size, while all CLRS-30 algorithms have only one parameter. The size of our DP table is $N \times (C + 1)$, so with our OOD experiments on parameters $(n = 16, C = 64)$, $(n = 64, C = 16)$, and $(n = 32, C = 32)$, we effectively test $4\times$ generalization as in CLRS-30, as the number of total elements in the table grows fourfold, and with the additional $(n = 64, C = 64)$ we test up to $16\times$ generalization.

Runtime and Memory Requirements. Runtime and memory requirements are comparable to the ones reported for CLRS-30 benchmark algorithms. Apart from the homogeneity, we do not modify the computational structure of the GNN layers, so we do not incur any slowdown from the GNN architecture. The only slowdown incurred is for reconstruction, where we do twice the amount of steps a standard model would do (see Figure 8). All experiments were conducted on a single NVIDIA A100 GPU with 40 GB of memory. The longest single training run for any model presented in this work was 44 minutes and 56 seconds. All models consumed less than 10 GB of memory, which allowed us to train 4 different models simultaneously. As a result, we were able to conduct

all experiments presented in the paper (across all 3 seeds) in less than 10 hours. Inference time is negligible, measuring less than one second per instance across all configurations.

Adapting Tropical Attention. To enable comparison with our model, we made modifications to the Tropical Attention repository[2] (commit `534ac67`). In `dataloaders.py`, the Knapsack dataset generation was modified to use uniform random values in the range $[0, 1]$ for item values instead of integer values from `value_range` (line 264), and the type signature of `set_knapsack_01` was updated to accept `float` values (line 177). A new `greedy_decoding` method was added to `experiment.py` to convert the model's probability predictions into valid Knapsack solutions. The `_eval_one_epoch` method (`experiment.py`, line 205) was extended to integrate greedy decoding for exact match calculation. Exact match ratio was added as an evaluation metric. The model was trained on samples where $n = 16$ and $C \leq 16$, and tested on the same size combinations as in our experiment.

D NAR Construction Implementation

Figure 5: Visualization of the NAR construction process for the Knapsack problem. At each step, the next row of the DP value table and the decision table is predicted from the current latent embeddings. Unlike the standard CLRS-30 approach, where the output is predicted separately using additional node/edge/graph features, our model accumulates the predicted rows of the decision table, which are then passed to the NAR reconstruction.

```python
# Specification:
'NAR_construction': {
  'pos': (Stage.INPUT, Location.NODE, Type.SCALAR),
  'edge_length_encoding': (Stage.INPUT, Location.EDGE, Type.CATEGORICAL),
  'dp_h': (Stage.HINT, Location.NODE, Type.SCALAR),
  'decision_h': (Stage.HINT, Location.NODE, Type.MASK),
  'input_categorical_weight': (Stage.HINT, Location.GRAPH, Type.CATEGORICAL), # "RNN input"
  'input_value': (Stage.HINT, Location.GRAPH, Type.SCALAR),                   # "RNN input"
  'dummy_output': (Stage.OUTPUT, Location.GRAPH, Type.SCALAR),
},

# Implementation:
def NAR_consturction(weights, values, capacity):
  probes = probing.initialize(specs.SPECS['NAR_construction'])
  # Edge length encoding construction is omitted for brevity, see Appendix F.
  probing.push(
    probes,
    specs.Stage.INPUT,
    next_probe={
      'pos': np.arange(capacity + 1) * 1.0 / (capacity + 1),
      'edge_length_encoding': edge_length_encoding,
    })

  # Capacity is implicitly defined through the number of nodes in the graph.
  dp = np.zeros(capacity + 1)
  decision_h = np.zeros(capacity + 1)

```

[2] https://github.com/Baran-phys/Tropical-Attention/

```
  for i in range(num_items + 1):
    probing.push(
      probes,
      specs.Stage.HINT,
      next_probe={
        'dp_h': dp.copy(),
        'decision_h': decision_h.copy(),
        # The following two are representetd as hints for implementation purposes,
        # but they are handled as inputs, because the true values are given in each
        # step and no prediction is done for them.
        'input_categorical_weight': to_categorical(weights[i] if i < num_items else 0),
        'input_value': values[i] if i < num_items else 0,
      })
    decision_h = np.zeros(capacity + 1)
    if i < num_items:
      for j in range(capacity, weights[i] - 1, -1):
        if dp[j] < dp[j - weights[i]] + values[i]:
          decision_h[j] = 1
          dp[j] = dp[j - weights[i]] + values[i]

  # Dummy output omitted for brevity.

  probing.finalize(probes)
  return 0, probes
```

Listing 1: Python-style pseudocode for NAR construction for Knapsack. In line with CLRS-30, adding a new algorithm involves specifying its specification, implementation, and sampler. Based on this, examples are generated and the neural model is trained. Our only deviation from CLRS-30 is that at each step we explicitly provide the current item as input. For simplicity, this is implemented via hints (marked with "`RNN input`"), which are ignored during decoding and in the loss function.

E NAR Reconstruction Implementation

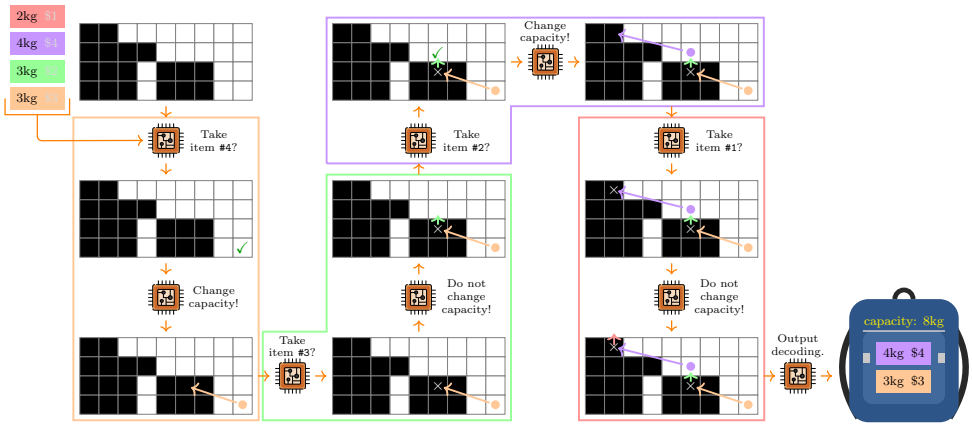


Figure 6: Visualization of the NAR reconstruction process for the Knapsack problem. Using the predicted decision table and item weights (see Figure 7), the model simulates item selection and traversal of the table, i.e. 2 steps per item ($2n$ in total). After the table is traversed, the probability of each item being part of the solution is predicted. While item selection and capacity updates could be merged into a single step, keeping them separate has shown better generalization (see Figure 8).

```
# Specification:
'NAR_reconstruction': {
  'pos': (Stage.INPUT, Location.NODE, Type.SCALAR),
  'node_type': (Stage.INPUT, Location.NODE, Type.MASK),
  'categorical_weight': (Stage.INPUT, Location.NODE, Type.CATEGORICAL),
  'edge_length_encoding': (Stage.INPUT, Location.EDGE, Type.CATEGORICAL),
  'predicted_decision_table': (Stage.INPUT, Location.EDGE, Type.MASK), # Soft MASK
  'node_idx': (Stage.HINT, Location.NODE, Type.MASK_ONE),             # The item currently being processed.
  'capacity_pointer': (Stage.HINT, Location.NODE, Type.MASK_ONE),     # The remaining capacity corresponds to one
                                                                      # of our capacity nodes.
  'alternation_type': (Stage.HINT, Location.GRAPH, Type.MASK),        # Item taking step or capacity decrease step?
                                                                      # Check alternation_type.
  'take_curr_item': (Stage.HINT, Location.GRAPH, Type.MASK),          # Whether the current item is part of the final solution.
  'selected_h': (Stage.HINT, Location.NODE, Type.MASK),               # The selected items so far.
  'selected': (Stage.OUTPUT, Location.NODE, Type.MASK),               # The final solution.
}


# Implementation:
def NAR_reconsturction(weights, capacity, predicted_decision_table):
  # Predicted_decision_table are probabilities calucluated by the NAR construction model.
  # These probabilities are assigned to the edges connecting the item and capacity nodes.
  # The other edges are assigned a probability of 0.

```

```python
num_items = len(weights)

dp = np.zeros((num_items, capacity + 1))
decision = np.zeros((num_items, capacity + 1))
# Calculate true dp and decision tables as in NAR_construction, omitted for brevity.

# Edge length encoding construction is omitted for brevity, see Appendix F.
# Capacity is implicitly defined through the number of the capacity nodes in the graph.
probing.push(
    probes,
    specs.Stage.INPUT,
    next_probe={
      'node_type': np.concatenate([np.zeros(num_items), np.ones(capacity + 1)]),
      'pos': np.concatenate([np.arange(num_items) * 1.0 / num_items, np.arange(capacity + 1) * 1.0 / (capacity + 1)]),
      'categorical_weight': to_categorical(np.concatenate([np.array(weights), np.zeros(capacity + 1)])),
      'edge_length_encoding': edge_length_encoding,
      'predicted_decision_table': predicted_decision_table,
    })

selected_items = np.zeros(num_items)
curr_capacity_pointer = capacity

for i in range(num_items, -1, -1):  # -1 = initialization
  for phase in (["init"] if i == num_items else ["taking_item", "moving_capacity_pointer"]):
    if phase == "init":
      node_idx, alternation_type, take_curr_item = 0, 0, 0
    elif phase == "taking_item":
      node_idx, alternation_type, take_curr_item = i, 1, decision[i, curr_capacity_pointer]
    else:
      node_idx, alternation_type, take_curr_item = i, 0, 0

    probing.push(
      probes,
      specs.Stage.HINT,
      next_probe={
        'node_idx': probing.mask_one(node_idx, num_items + capacity + 1),
        'capacity_pointer': probing.mask_one(num_items + curr_capacity_pointer, num_items + capacity + 1),
        'alternation_type': alternation_type,
        'take_curr_item': take_curr_item,
        'selected_h': np.concatenate([selected_items, -1 * np.ones(capacity + 1)]),
      })

    if phase == "taking_item" and take_curr_item == 1:
      curr_position -= weights[i] # Move capacity pointer.
      selected_items[i] = 1       # Add item to selected items.


probing.push(
    probes,
    specs.Stage.OUTPUT,
    next_probe={
      'selected': np.concatenate([selected_items, -1 * np.ones(capacity + 1)]),
    })

probing.finalize(probes)
return selected, probes
```

Listing 2: Python-style pseudocode for NAR reconstruction for Knapsack. The hints discussed in the main text (§2, p. 3) are presented in L7-L14. Out of those, the only less standard is `alteration_type`. It is similar to the `phase` hint found in the other algorithms implemented in CLRS-30, with the exception that `alteration_type` is *always* a sequence of alternating zeros and ones. Contrary to construction, where output was obtained from stacked hints, here it is decoded from the final latent embeddings, as standard for CLRS-30.

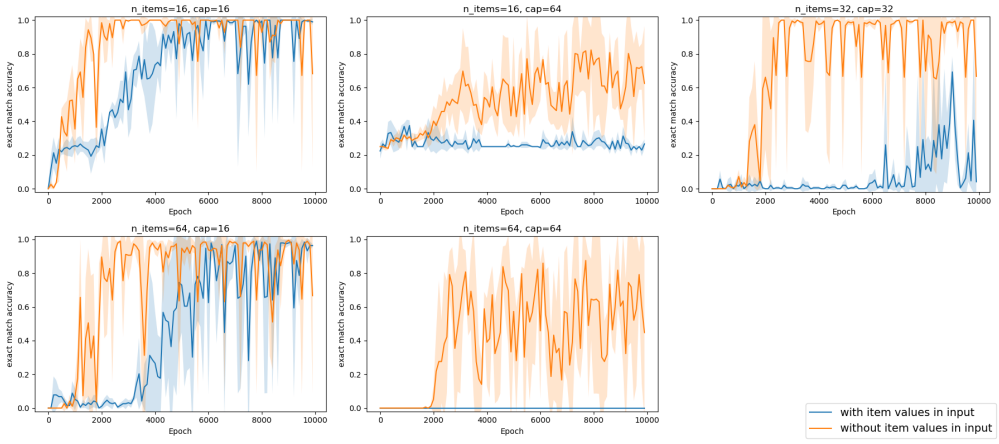


Figure 7: Comparison of NAR reconstruction performance with and without item values in the input, given the true decision table during both training and inference. Having the items' values in the input prevents the NAR reconstruction model from generalizing to larger instances.

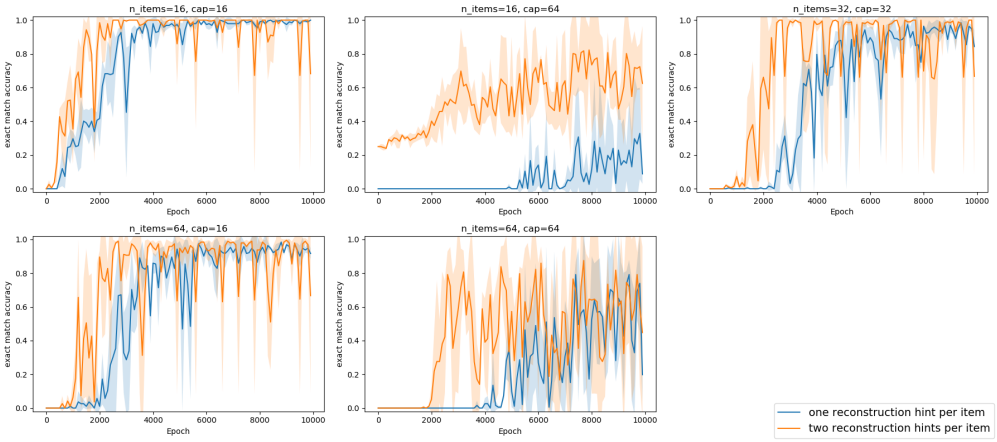


Figure 8: Comparison of the joint-step and split-step NAR reconstruction performances, given the true decision table during both training and inference. Splitting the processing of a single item into two steps enables better generalization across the configurations we consider, suggesting that a single message-passing step is insufficient for both item selection and capacity updates.

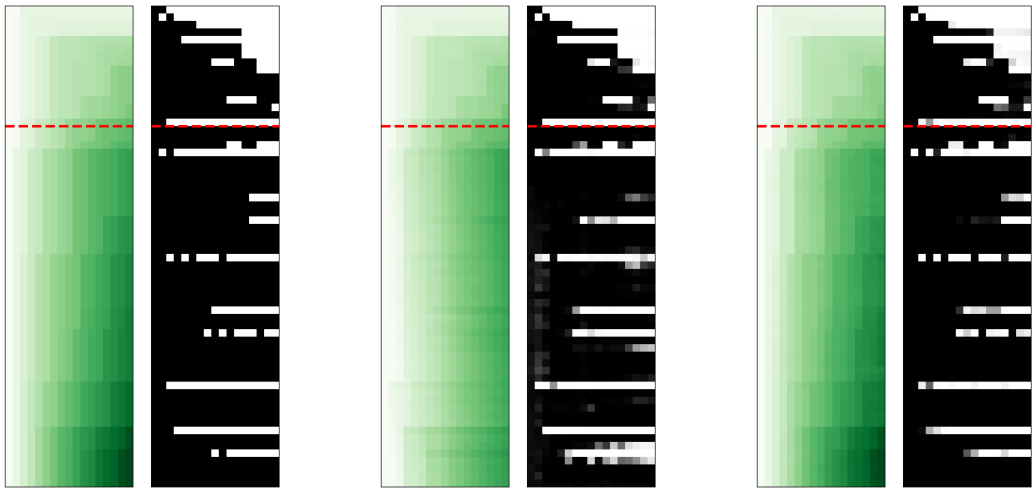
F Edge Length Encoding

As shown in Figure 2, edge length encoding for the NAR construction model assigns to each graph edge $e_{ij} \in \mathcal{E}$ a categorical feature of size $M = 10$ that represents the absolute difference between the capacities associated with vertices u_i and u_j, specifically $|i - j|$. Since these are categorical features, all differences greater than or equal to $M - 1$ are treated as equivalent. For the NAR reconstruction model, edge length encoding follows a similar approach, but only considers edges between the $C + 1$ capacity vertices. The remaining $(C + 1 + n)^2 - (C + 1)^2$ edges are assigned the same categorical label as distances greater than or equal to $M - 1$.

We further elaborate on this empirically important inductive bias by explaining its motivation. As an example, we use the construction model, which has $C + 1$ nodes corresponding to capacities $0, 1, \ldots, C$. A similar argument can be straightforwardly applied to the reconstruction model. Recall the DP formula $\mathrm{dp}_{i,c} = \max(\mathrm{dp}_{i-1,c}, \mathrm{dp}_{i-1,c-w_i} + v_i)$. Observe that at the i-th step, which corresponds to item i, the following holds for every node c: the DP scalar feature associated with node c must either remain unchanged or be updated based on the value of the DP scalar feature associated with node $c' = c - w_i$, where $c \geq w_i$. We see that $c - c' = w_i$, meaning that for all nodes c, the distance to their corresponding candidate node c' is equal to the item weight w_i. NAR processors are based on GNNs where, along every edge (i, j), a message from node i to node j, m_{ij}, is computed based on the edge embedding and the embeddings of i and j, and these messages are then aggregated across all neighboring nodes [2]. As it is currently implemented, the CLRS-30 employs a positional encoding scheme with *scalar real* values in $[0, 1]$. As a result, the model cannot infer if either of the two nodes is a candidate for the other. The introduction of edge length encoding, which assigns categorical features to edges corresponding to their length (i.e., the absolute difference in capacities), means that, to a large extent, computing the message m_{ij} boils down to checking whether the corresponding distance equals w_i or not. Note that our use of the absolute difference $|c - c'|$ in the definition of edge length encoding reduces the number of required categories, which is possible because candidates $c' < c$ are easily identified based on the existing scalar node positional encoding. The cutoff M is used because it is necessary when employing categorical features, and its value is adjusted to the maximum considered item weight. It is important to note that this argumentation is not limited to the Knapsack problem but can be applied to all pseudo-polynomial DP algorithms. Specifically, the same alignment between edge length encoding and the structure of the pseudo-polynomial DP relation holds for other problems: in Subset Sum and Partition problems, the alignment corresponds to differences of possible sums, while in Minimum Coin Exchange, it corresponds to differences of possible target amounts, to name a few.

G Homogeneous NAR Construction

In Figure 9, we observe that for the regular processor the bottom-right scalar predictions appear lighter than expected, indicating for out-of-distribution the model is predicting lower DP scalar values than the ground-truth ones. A similar phenomenon was reported by Mirjanic et al. [38] in the context of the Bellman-Ford algorithm, where it was suggested to scale the embeddings by a constant factor $c < 1$ at every message-passing step. Differently from them, we address this problem by *constraining our processor to be homogeneous* [8], which enforces the model to be invariant with respect to the scale ($f(\alpha x) = \alpha f(x), \alpha > 0$) of the item values. Concretely, this entails removing bias terms from the network, as well as eliminating layer normalisations [39] and gating mechanisms [7, p.5]. Additionally, in each step, it is necessary to encode only homogeneous $\mathrm{dp}_{i,:}$ hints, while applying the *decode-only* hint mode for other hints. As shown in Figure 10, these modifications improve the correlation between true and predicted DP values, and lead to better generalization of the NAR construction model (Table 5), which enables significantly better overall NAR model performance compared to the no-hint baseline.

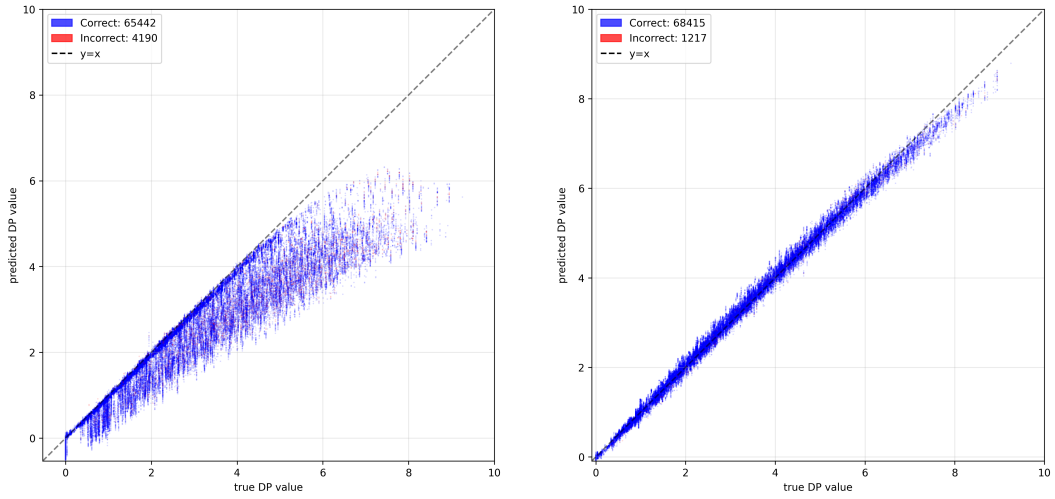


(a) True DP value and decision table

(b) Tables predicted by regular NAR construction

(c) Tables predicted by homogeneous NAR construction

Figure 9: Comparison of DP value and decision tables for $n = 64$, $C = 16$, examining the effect of the NAR construction model homogeneity. The lighter color in the bottom-right of the regular processor's scalar predictions indicates under-prediction of out-of-distribution values, which is corrected in the homogeneous model.



(a) Regular NAR construction

(b) Homogeneous NAR construction

Figure 10: Correlation between true and predicted DP values for (a) regular and (b) homogeneous NAR construction models, considering all examples for $n = 64$, $C = 16$. Blue dots represent elements of the DP table whose corresponding decision is correctly determined, while red dots represent those where it is not.

Table 5: In- and out- of distribution performance comparison of regular vs. homogeneous NAR construction models, tested on original and $10\times$ scaled item values. From the $10\times$ results we observe that the homogeneous model is indeed invariant to item value scaling, while the poor performance of the regular model in this case further confirms its inability to generalize to larger scalar values.

mult.	n	C	micro-F1	
			reg. c.	homo. c.
1	16	16	$\mathbf{0.991}_{\pm 0.001}$	$0.989_{\pm 0.002}$
	16	64	$\mathbf{0.978}_{\pm 0.005}$	$0.976_{\pm 0.013}$
	32	32	$0.958_{\pm 0.008}$	$\mathbf{0.972}_{\pm 0.013}$
	64	16	$0.909_{\pm 0.030}$	$\mathbf{0.972}_{\pm 0.011}$
	64	64	$0.847_{\pm 0.033}$	$\mathbf{0.861}_{\pm 0.105}$
10	16	16	$0.460_{\pm 0.182}$	$\mathbf{0.989}_{\pm 0.002}$
	16	64	$0.646_{\pm 0.255}$	$\mathbf{0.976}_{\pm 0.013}$
	32	32	$0.436_{\pm 0.157}$	$\mathbf{0.971}_{\pm 0.014}$
	64	16	$0.263_{\pm 0.066}$	$\mathbf{0.972}_{\pm 0.010}$
	64	64	$0.423_{\pm 0.121}$	$\mathbf{0.861}_{\pm 0.104}$

H Deterministic Reconstruction Implementation

```python
def deterministic_reconstruction(weights, capacity, decision):
    n = len(weights)
    soft_selected = numpy.zeros(n)
    dp_prob = numpy.zeros((n, capacity + 1))
    for i in range(n - 1, -1, -1):
        for j in range(capacity, 0, -1):
            if i == n - 1:
                if j == capacity:
                    dp_prob[i, j] = 1
                continue
            dp_prob[i, j] = (1 - decision[i + 1, j]) * dp_prob[i + 1, j]
            if j + weights[i + 1] <= capacity:
                dp_prob[i, j] += decision[i + 1, j + weights[i + 1]] * dp_prob[i + 1, j + weights[i + 1]]
        for j in range(weights[i], capacity + 1):
            soft_selected[i] += dp_prob[i, j] * decision[i, j]
    return soft_selected
```

Listing 3: Python implementation of the deterministic reconstruction algorithm.

Note that in our two-phase NAR approach, the reconstruction model is always trained on the true discrete steps of classical backtracking, regardless of the probabilistic decision table predicted by the NAR construction model. The classical reconstruction procedure relies on *discrete* dynamic programming: at each state (i, j) one either includes or excludes item i, producing a single path and a final 0–1 solution vector. This process is not differentiable, as it involves the discrete $\arg\max$ operator.

We introduce a *deterministic relaxation* of this process (see Listing 3). Instead of hard choices, we employ a table $\texttt{decision}[i, j] \in [0, 1]$, which represents the "soft" probability of including item i given remaining capacity j. Using these probabilities, we recursively compute a distribution of probability mass $\texttt{dp\_prob}[i, j]$, describing the probability of reaching state (i, j) during reconstruction. At each branching step, the mass is split proportionally according to $\texttt{decision}$ (for inclusion) and $1 - \texttt{decision}$ (for exclusion). In this way, the algorithm maintains a superposition of all feasible backtracking paths, rather than committing to a single one. Finally, the expected inclusion of item i is obtained as

$$\texttt{soft\_selected}[i] \;=\; \sum_{j=0}^{C} \texttt{dp\_prob}[i, j] \cdot \texttt{decision}[i, j].$$

This produces values in $[0, 1]$, which can be interpreted as a continuous relaxation of binary inclusion decisions. The procedure is differentiable since the recursion involves only additions and multiplications of continuous variables. No discrete operators (such as $\arg\max$ or indicator functions) appear, ensuring that gradients can propagate through the entire reconstruction phase.