Learning Elimination Ordering for Tree Decomposition Problem

Taras Khakhulin, Roman Schutski, Ivan Oseledets
Skolkovo Institute of Science and Technology, Moscow
{taras.khakhulin, r.schutski, i.oseledets}@skoltech.ru

Abstract

We propose a Reinforcement Learning-based approach to approximately solve the Tree Decomposition problem. Recently, it was shown that learned heuristics could successfully solve combinatorial problems. We establish that our approach successfully generalizes from small graphs, where an optimal Tree Decomposition can be found by exact algorithms, to large instances of practical interest, while still having very low time-to-solution. On the other hand, the agent-based approach surpasses all classical greedy heuristics by the quality of the solution.

1 Introduction

At the core of many practical tasks, such as probabilistic inference, decision making, planning, and other problems, lies a combinatorial (NP-complete) optimization problem. The solution of large NP-problems is often possible only with the help of heuristics. These heuristics are designed manually, which is a complicated and time-consuming process. The resulting algorithm is also typically domain-specific and can not be reused. Recently, Reinforcement Learning (RL) application to design heuristics gained significant attention [2, 19, 15]. RL is a natural framework for the automatic design of approximation algorithms for problems with an inherent cost function and large search space, which is the essence of combinatorial optimization. In this work, we consider the Tree Decomposition (TD) [6] problem. The TD is central to the analysis of the complexity and the topological structure of graphs. An integer parameter treewidth characterizes the solution of the TD problem. The treewidth quantifies the complexity of many NP-problems; the computational cost of solving these problems is exponential in the treewidth, but only polynomial in the problem's graph size. Tree Decomposition emerges as a core step in various contexts, such as probabilistic inference [14] or shortest path search [9]. The TD problem is usually solved on non-Euclidean graphs, as opposed to the traveling salesman problem (TSP), which is the most common target of recent studies of trainable heuristics [19, 2].

Several exact [13, 22, 7] and approximate [5] algorithms exist to solve the TD problem. We propose a Reinforcement Learning-based approach for TD. In this work, we demonstrate that the resulting policy can successfully *generalize* to problems with different graph structures and sizes. Our findings show that the agent can be trained even on a *single graph*. The quality of the solution of our agent-based procedure is superior compared to all simple greedy heuristics, and the time-to-solution is much lower compared to advanced algorithms

2 Graph Convolutional Policy

We start with a formulation of Tree Decomposition as a linear ordering problem of a graph's vertices. Then we shortly define a graph agent, which finds an ordering to calculate a TD.

During tree decomposition, an arbitrary input graph is mapped to a tree graph. A full definition of Tree Decomposition can be found in the original work [20] or in a classical review [6]. Informally,

1st Workshop on Learning Meets Combinatorial Algorithms @ NeurIPS 2020, Vancouver, Canada.

the Tree Decomposition measures how close a given graph resembles a tree. We provide a more formal definition in Appendix A. The quality of TD is measured by *treewidth*, which should be minimized across all possible trees. It can be shown that finding a tree with the lowest treewidth is NP-hard [3]. Instead of building the tree directly, in this work, we will search for TD by using its relation to the ordering of vertices [4]. The procedure to build a final tree given a permutation of vertices is described in Appendix A.

A permutation $\pi(u) : u \in U \to [1 \dots |U|]$ of vertices of a graph $G = (U, E)$ is called an *elimination* order. Consequently, $\pi^{-1}(t) : t \in [1 \dots |U|] \to U$ is an inverse function of the order. Given a number $\pi^{-1}(t)$ returns a node. The elimination order of the graph can be used in the following procedure for a graph G:

1. For $t \in [1 \dots |U|]$, take the t-th node $u = \pi^{-1}(t)$.
2. Connect all neighbors of u into a clique (fully connected subgraph) and record the size of the resulting clique. Remove u. This results in a new graph G_t.
3. Repeat the procedure until G_t is empty.

If G has treewidth at most k, then there is an elimination order π of G, such that each vertex has at most $k + 1$ neighbors in the elimination procedure with respect to π [1]. We define the maximal number of neighbors associated with a permutation π as c_π. The treewidth is a minimum of $c_\pi - 1$ across all possible permutations, i.e. $tw(G) = \min_\pi c_\pi - 1$. If the treewidth of a graph is small, then it is tree-like. In particular, a tree has treewidth 1. We define our problem as follows: given an undirected graph G, find an elimination order, i.e., a permutation of the vertices, such that the number of neighbors in the elimination procedure along π is minimal across all permutations. The example of the elimination procedure is shown in Appendix A.

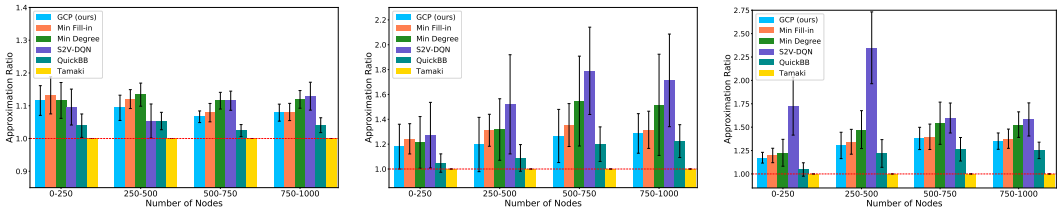
Graph Agent. At every step t of the elimination process we represent the graph G_t with n nodes as the pair $(\mathbf{A^t}, \mathbf{H^t})$, where $\mathbf{A^t}$ is the adjacency matrix and $\mathbf{H^t} \in \mathbb{R}^{n \times d}$ is the node feature matrix, assuming that the graph has d features. Input features H_0^t is a constant matrix initialized with ones. H_0^t is passed through a graph neural network, which parameterizes our *Graph Convolution Policy* (GCP), and is transformed into $\mathbf{H^t}$. We experiment with different graph neural architectures, which consist of three types of functions: message passing, aggregation, and update function. Our approach to feature extraction is similar to the one used in the recent work [25]. The details of the architecture are listed in Appendix B.1. To formulate the TD problem in the RL framework, we should define a Markov decision process (MDP) for the agent. At each time step t, the agent selects a node of the graph G_t based on the observed information, represented by the state features. The node is then eliminated from G_t, and the graph G_{t+1} is produced. We define the MDP as a tuple $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$, where the $\mathcal{A}$ is a set of actions, $\mathcal{S}$ is a set of states and r is the cost for the sequential combinatorial problem. $\mathcal{A}$ consists of nodes $u \in U_t \subset U$, which have not been eliminated at the current step. For the TD problem, r is the size of a maximal clique in the graph obtained during the elimination process. Each state $s_t \in \mathcal{S}$ directly defined as the embedding matrix $\mathbf{H}^{(l)}$ and the structure of the graph G_t. We use original GCN [17] to extract features on every step, and conduct experiments with its extensions [26, 24]. Solving an MDP means finding an optimal policy Π, a mapping which outputs a distribution of actions. We apply an Actor-Critic [18] algorithm to solve this problem.

3 Experiments

In this section, we demonstrate that the agent can successfully learn a heuristic. We compare our results with common human-designed heuristics and show that our neural heuristic trained on a single graph can generalize on the graphs with many vertices.

3.1 Data

We use three datasets with different structures for the experiments. The first dataset consists of random Erdős–Rényi (ER) graphs [11] with edge probability $5/n$, where n is the number of nodes. We experimentally found that this choice of the edge probability leads to hard instances of TD problem (the ER graphs contain many cycles and are not too dense). For the validation, we use a fixed set $\mathcal{D}$ of 100 Erdős–Rényi graphs with 10 to 1000 nodes and the set $\mathcal{D}_{\text{small}}$ as first 50 graphs from $\mathcal{D}$. The second dataset is taken from the PACE2017 competition [10] on designing TD algorithms. The



(a) Erdős–Rényi (b) PACE2017 (c) Circuits

Figure 1: Approximation Ratio (less is better) of different methods averaged over all graphs in the respective dataset. The red line shows the AR compared to Tamaki solver with 30 minutes time bound. The result of the agent (trained on a single graph) is slightly better than all greedy human-designed heuristics.

third dataset comprises graphs that emerge during the simulation of random quantum circuits [8], a common framework in the study of quantum computing supremacy. The main reason for using the ER graphs is to simplify the reproducibility of the experiments. Our main results are obtained on the PACE2017 dataset, explicitly created to test TD algorithms. The third dataset is selected as a practical TD application example.

3.2 Baseline and Evaluation Details

Before starting to experiment with our model, we should introduce other algorithms used in this work. We use two greedy heuristics, two specialized TD solvers and adapt the S2V-DQN [15] method to the TD problem.

The greedy heuristics are minimal degree and minimal fill-in [6]. They work fast enough and are accurate in practice [23]. The minimal degree heuristic selects nodes with a minimal number of neighbors. The minimal fill-in algorithm selects the nodes such that the number of introduced edges at each step is minimized.

Two specialized solvers are based on very different approaches, and both will produce an exact solution if provided enough running time (exponential in the graph size). The first solver by Tamaki et al. [22] employs the connection of TD and vertex separators. This method searches for optimal TD directly in the space of tree graphs without referring to vertices' ordering. It is a winner algorithm on the PACE2017 competition. Another powerful solver is QuickBB [13], which is based on the branch and bound algorithm. We restrict both solvers' runtime to 30 minutes per graph instance as in the PACE2017 competition [10].

We also compare with another Reinforcement Learning approach called S2V-DQN [15]. This algorithm was previously used to solve sequential optimization problems on graphs, such as Minimum Vertex Cover, and we adapt it to our tasks. We train the S2V-DQN model on a single graph to have faster convergence.

To produce a solution using our RL-based heuristic, we sample 10 trajectories and take the one with the *lowest* treewidth. We compare the performance of different solvers with respect to the Tamaki solver [22]. As a performance metric an *Approximation Ratio* (AR), AR $= \frac{tw_{\text{method}}(G)}{tw_{\text{tamaki}}(G)}$ is used. The AR metric is the standard in the literature on approximation algorithms. The treewidth is calculated from the elimination order π produced by the solvers.

Training Graph. Surprisingly, we find that the agent's score does not significantly depend on the source of the training graph, despite the graphs which we use have quite different structures. Also, the training set's size does not significantly affect final accuracy but rather resulted in longer training times. More details are provided in Appendix B.2. This fact may be attributed to the agent's inefficiency or the uniform structure of the TD problem already for average-sized graphs (with 50 nodes). We admit that this intriguing fact may need additional investigation. To simplify the reproducibility of our experiments, we choose Erdős–Rényi graphs for further tests. As can be seen from Figure 4a in Appendix B.3, the accuracy of the GCN-agent depends on the number of nodes only slightly. For further experiments, we select a graph with 70 nodes.

Method	Approx. Ratio ↓	Ratio Max.	Avg. Time, s
	Exact solvers		
Tamaki	1.0	1.0	17.01
QuickBB	1.09 ± 0.11	1.41	1617.27
	Greedy heuristics		
Min-Fill	1.24 ± 0.22	1.78	153.52
Min-Degree	1.31 ± 0.23	2.13	0.04
S2V-DQN	1.45 ± 0.44	1.79	0.9
Random agent	1.84	-	-
	Ours		
GCP (sampling)	1.19 ± 0.16	1.44	30.93
GCP (greedy)	1.21 ± 0.13	1.41	3.564
GCP-GIN	1.34 ± 0.21	1.52	35.76
GCP-GAT	1.31 ± 0.14	1.42	32.12

Table 1: A summary of the agent's performance on the PACE2017 dataset. The values are averaged over all graphs with sizes from 10 to 1000 nodes. The GCP solver performs better than all greedy heuristics and has lower time-to-solution, especially when using a greedy algorithm.

3.3 Comparison with Other Solvers

In this section, we compare our agent's performance to other methods on the graphs of the different sizes and structures. The agent is trained on the ER graph with 70 nodes. The results of all experiments are summarized in Figure 1. The learned solver usually does not reach the accuracy of specialized exact algorithms but outperforms greedy heuristics. It is interesting to see that one can learn a relatively useful heuristic using only a single graph. The quality of the agent's solution deteriorates as the test graphs' size grows, but usually slower than the quality of the solution found by greedy heuristics.

The PACE2017 dataset results, which we consider the most representative, are summarized in Table 1. We note that in addition to the high accuracy comparing to greedy heuristics (both on average and in the worst case), the RL-based heuristic has very competitive time-to-solution, which is essential for solving NP problems. Furthermore, this time can be significantly decreased, as sampling can be trivially performed in parallel. One explanation of why the agent works this way is the excellent ability of graph neural networks to approximate dynamic programming algorithms, which is demonstrated in [25].

4 Conclusion

This work formulates the elimination process for Tree Decomposition as a task for RL. We propose a model that can directly learn how to solve this combinatorial optimization problem using a *single graph* for training and simple GCN. The training procedure also can be performed using a large set of graphs, but this work aims to show the generalization ability of a simple agent trained on a single graph. We provide experiments that a learnable heuristic can surpass greedy, manually designed ones. Our method can generalize to large problem instances without significantly sacrificing the solution's quality and without a large increase in computational time. Our preliminary results suggest that this approach is a good starting point for learning heuristics for combinatorial problems on graphs. Outperforming specialized algorithms is still a challenge in this setting. The generalization ability of our agent-based approach could also be fruitful for algorithmic reasoning tasks. The perspective direction of research is a combination of RL-based heuristics with local-search methods and an extension of this method for tensor contraction task.

References

[1] Eyal Amir. Efficient approximation for triangulation of minimum treewidth. In *UAI*, 2001.

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

[3] Jeremias Berg and Matti Järvisalo. Sat-based approaches to treewidth computation: An evaluation. In *ICTAI*. IEEE Computer Society, 2014.

[4] Jean Blair and Barry Peyton. An introduction to chordal graphs and clique trees. In *Graph theory and sparse matrix computation*, 01 1991.

[5] Hans Bodlaender and Arie Koster. Treewidth computations. i: Upper bounds. *Information and Computation*, 208, 03 2010.

[6] Hans L Bodlaender. A tourist guide through treewidth. *Acta cybernetica*, 11(1-2):1, 1994.

[7] Hans L Bodlaender, Fedor V Fomin, Arie MCA Koster, Dieter Kratsch, and Dimitrios M Thilikos. On exact algorithms for treewidth. In *European Symposium on Algorithms*, pages 672–683. Springer, 2006.

[8] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, and Hartmut Neven. Simulation of low-depth quantum circuits as complex undirected graphical models. *arXiv preprint arXiv:1712.05384*, 2017.

[9] Krishnendu Chatterjee, Rasmus Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. Optimal reachability and a space-time tradeoff for distance queries in constant-treewidth graphs. In *ESA*, 2016.

[10] Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The pace 2017 parameterized algorithms and computational experiments challenge: The second iteration. In *IPEC*, 2017.

[11] P. Erdős and A Rényi. On the evolution of random graphs. In *Publ. Math. Inst. Hungary. Acad. Sci.*, pages 17–61, 1960.

[12] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on RLGM*, 2019.

[13] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. *arXiv preprint arXiv:1207.4109*, 2012.

[14] Kalev Kask, Andrew Gelfand, Lars Otten, and Rina Dechter. Pushing the power of stochastic greedy ordering schemes for inference in graphical models. In *AAAI*, 2011.

[15] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *NIPS*, 2017.

[16] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

[17] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2016.

[18] Vijaymohan Konda. *Actor-critic Algorithms*. PhD thesis, Cambridge, MA, Cambridge, MA, USA, 2002. AAI0804543.

[19] Wouter Kool, Herke van Hoof, and Max Welling. Attention, Learn to Solve Routing Problems! In *ICLR*, Mar 2018.

[20] Neil Robertson and P. D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309 – 322, 1986.

[21] Roman Schutski, Dmitry Kolmakov, Taras Khakhulin, and Ivan Oseledets. Simple heuristics for efficient parallel tensor contraction and quantum circuit simulation. *arXiv preprint arXiv:2004.10892*, 2020.

[22] Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. *Journal of Combinatorial Optimization*, 37(4):1283–1311, 2019.

[23] Thomas van Dijk, Jan-Pieter van den Heuvel, and Wouter Slob. Computing treewidth with libtw. *Citeseer.*, 2006.

[24] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Y. Bengio. Graph attention networks. In *ICLR*, 2018.

[25] Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In *ICLR*, 2020.

[26] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *ICLR*, 2019.

APPENDIX

A Elimination Ordering

Here we provide examples of the elimination procedure and discuss its connection to tree decomposition. Consider two elimination procedures of the same graph in Fig. 2. In the first case (upper part), the maximal clique size is 3 (and hence the treewidth of the associated TD is 2). In the second case, the treewidth is 1, and the order is optimal (since the graph is a tree). Note that the optimal order may not be unique: nodes number 1 and 2 can be swapped, for example, but the same treewidth is achieved.

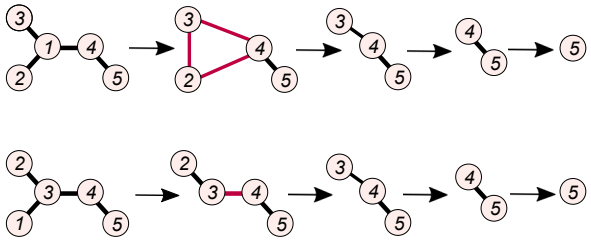


Figure 2: Elimination procedure for a given order of nodes π. The labels on nodes correspond to their order. In red is shown a maximal clique that emerges during the elimination procedure. The second order is optimal: the clique size is minimized

Let us formally introduce Tree Decomposition. A Tree Decomposition is a mapping of the initial graph $G = (U, E)$ into a tree graph $F = (B, T)$. Here U is the set of nodes, and E is the set of edges of the initial graph; B is the set of *bags* (each bag $b \in B$ is a subset of nodes of the graph G, $b \subset U$) and T is the set of the edges of the tree graph. A Tree Decomposition has to fulfill three criteria to be valid:

1. Every node of G is in some bag, i.e., $\cup_{b \in B} b = U$.
2. For every edge $(u, v) \in E$ there must be a bag such that both endpoints are in that bag, i.e., $\exists b : u \in b, v \in b$.
3. For every node u of G, the subgraph of the tree F, induced by all bags that contain u, is a connected tree.

Tree decomposition is inherently related to the elimination procedure and can be built provided some order of node eliminations is chosen. To build a TD, one has to add all cliques which emerge during the elimination procedure as the vertices B. A minimum spanning tree T over B, which is consistent with the criteria mentioned in the previous paragraph, is the TD. Pseudocode for building a TD given an elimination order can be found, for example, in [21]. The treewidth is the size of the maximal bag minus 1. Conversely, given a TD, multiple equivalent elimination orders (in the sense of the maximal clique/treewidth) can be found. This procedure can also be found in [21].

B Additional Experiments

This section contains details of additional experiments. All of them are aimed at choosing the agent training procedure and the analysis of generalization.

B.1 Training Details

Our code is based on the PyTorch Geometric [12]. We train all models with Adam [16] with parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$ and learning rate $lr = 0.008$. We also set the parameters of the RL algorithm as follows: the discount factor $\gamma = 0.999$, the GAE weight $\lambda = 0.85$, the weight of the value loss $\beta_{\text{value}} = 1.0$, and the multiplier of the entropy regularization $\beta_{\text{entropy}} = 0.001$. The GCN subnetwork contains three layers, and the hidden feature size is 64, as in [15]. A 2-layer perceptron parameterizes policy and value heads with a hidden dimension equal to 64. All hyperparameters are selected with grid search on the hold-out validation set. We train our model on the NVIDIA 1080ti with one thread for sampling.

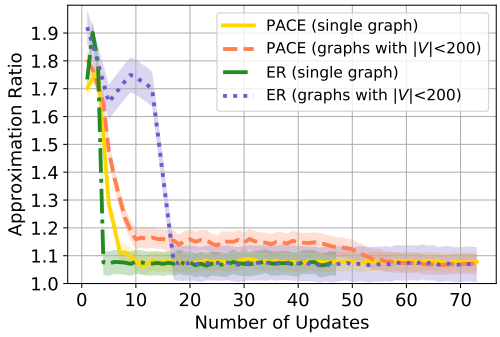
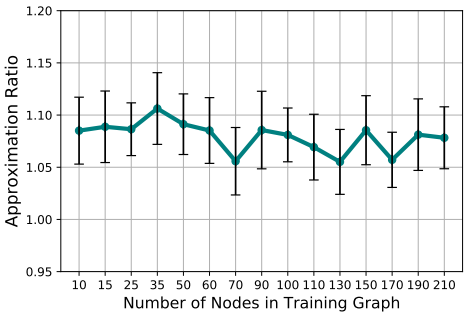


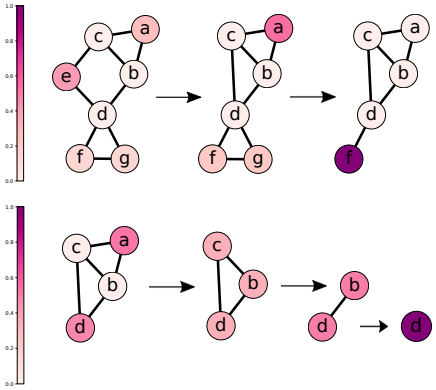
Figure 3: Learning curves showing the Approximation Ratio (less is better) and number of updates (mean and variance over 5 random seeds). The agents are trained on different training sets and compared on the same validation set $\mathcal{D}_{\text{small}}$. For training on a single graph, we choose the graph "he064" from the PACE dataset with $|V| = 68$, and the ER graph with the same number of nodes. For training on the graphs with $|V| < 200$, we sample a new graph at every update step. The agent trained on a single graph achieves the same accuracy with less training iterations.

B.2 Choice of the Training Graph

In this series of experiments, we would like to check how the choice of training dataset affects the resulting agent performance. In deep RL, it is common to train the agent on a huge number of problem instances and expect an increase in the model quality from a larger number of training examples. We check this intuition in the following experiment. First, we train the agent on a single graph. We choose one ER graph with 68 nodes and one graph from the PACE dataset ("he064"). Next, we train on different ER and PACE graphs with up to 200 nodes. We randomly select a graph for every training step. The performance of the agents is compared to the same validation set of Erdős–Rényi graphs. The results are shown in Figure 3. It turns out it is sufficient to train our agent for 10 epochs (around 20 minutes).



(a) Dependence of the Approximation Ratio (evaluated on the dataset $\mathcal{D}$) on the number of nodes in the training graph. The validation results slightly vary on the size of a training graph.



(b) Elimination procedure of an agent. The probabilities of actions are shown in color. At each step, the next node is sampled according to the probability. Starting from the fifth step, when a final clique is encountered, the distribution is uniform.

B.3 Training Graph size

In this set of experiments, we consider training on graphs of different sizes. It is known that if the action space is large, the RL methods may have problems with convergence due to the significant variance of the gradients. It is hence desirable to keep the training graph size small (for example, less than 100 nodes); however, training on larger instances may produce agents with better performance. To check the influence of the training graph size on the learned heuristic accuracy, we train separate agents on ER graphs with 10 to 210 nodes. The dependence of the AR is shown in Figure 4a.

It can be noted that better accuracy is obtained for the graphs with 70 and 130 nodes, which is approximately the GCN feature size or twice feature size. The final result on the validation set slightly depends on the number of nodes on the training graph.

B.4 Agent Decision Making

In this section, we analyze the agent's decision-making process in order to get insights into the structure of the TD problem.

As mentioned earlier, in all experiments, we use sampling to get candidate solutions and select one with the best score. We use a small sample of 10 trajectories, which is a stronger result compared to other works on neural-based heuristics, where samples of size 1000 are common [19]. The efficiency with a small sample size suggests that the agent learns a distribution close to the "true" distribution of solutions.

Another evidence in support of the high quality of the learned distribution is the behavior of the agent on complete graphs. A fully connected graph with n nodes has treewidth $n - 1$, and all elimination orders on it are equivalent. The policy learned by our agent produces a uniform distribution on the complete graphs (after they appeared during the elimination process), as shown in Figure 4b.