Training Graph Neural Networks with Policy Gradients to Perform Tree Search

Matthew Macfarlane
University of Amsterdam
`m.v.macfarlane@uva.nl`

Diederik Roijers
Vrije Universiteit Brussel
City of Amsterdam
(Urban Innovation & R&D)
`diederik.roijers@vub.be`

Herke van Hoof
University of Amsterdam
`h.c.vanhoof@uva.nl`

Abstract

Monte Carlo Tree Search has been shown to be a well-performing approach for decision problems such as board games and Atari games, but relies on heuristic design decisions that are non-adaptive and not necessarily optimal for all problems. Learned policies and value functions can augment MCTS by leveraging the state information at the nodes in the search tree. However, these learned functions do not take the search tree structure into account and can be sensitive to value estimation errors. In this paper[1], we propose a new method that, using Reinforcement Learning, learns how to expand the search tree and make decisions using Graph Neural Networks. This enables the policy to fully leverage the search tree and learn how to search based on the specific problem. Firstly, we show in an environment where state information is limited that the policy is able to leverage information from the search tree. Concluding, we find that the method outperforms popular baselines on two diverse and problems known to require planning: Sokoban and the Travelling salesman problem.

1 Introduction

Planning in complex decision problems with many states is an important and challenging area of research, with applications to areas such as routing [16] and game playing [28][26]. Without planning, model free methods can require very large training budgets to learn a policy (mapping states to distributions over actions) that leads to high reward, particularly when the problem is non-Markovian.

Planning can make policies a lot more effective by explicitly looking ahead of the current state. This can improve performance as there is more information to support decision making (future states and rewards). Theoretically the current state should be sufficient for acting optimally, however this can practically be a difficult function to learn. The downside to planning is that it costs time, often when we need it the most (when the policy is being executed). Thus, methods that are parsimonious with the amount of planning performed are desirable.

A popular method of planning is Monte Carlo Tree Search. This method consists of forming a search tree containing future states and rewards, while also iteratively updating expected rewards for all node-action pairs in the tree to guide expansions. MCTS only makes use of the reward information and visit counts in the tree. With appropriate reward scaling (UCB requires action values to be between 0 and 1), MCTS performs well on many decision problems with otherwise no domain knowledge. However, since it does not generalise between states, it tends to be inefficient for larger problems and requires tree search with a high budget. This prevents its use in domains where compute at execution time is limited compared to the problem size.

[1]Accepted to the Deep Reinforcement Learning Workshop NeurIPS 2022

MCTS has been made more efficient through the use of policy and value functions [3] [28]. By leveraging the state information at nodes in the search tree, these functions generalize to unseen states without needing to expand the tree as extensively. However, adding learnt components to MCTS in a heuristic manner adds some additional problems. Firstly, it can be difficult to balance trading off the various components of the algorithm; the optimal hyperparameters controlling this trade-off are game dependent. Secondly, using the value function to evaluate off-policy states leads to bias and needs correction [11]. Such methods are usually designed in a way that they act as a policy improvement operator for the model free policy. However, this is usually suboptimal when we care about maximising performance for a given search budget. Other methods such as performing policy gradient updates at test time [16] likewise are not optimized for a particular budget and usually require long rollouts.

Instead of evaluating search trees in a heuristic manner, we design a policy capable of mapping search trees directly to a distribution over actions. This policy can then be trained using Reinforcement Learning to maximize reward for a given search budget. Some advantages of this method include that it can be trained with any search budget. It also overcomes the problem of evaluating off-policy states, by training directly how to use these states, even though we are unlikely to ever reach this state outside of searching. We first demonstrate a parametrised Graph Network policy can be trained in a stable manner using Reinforcement Learning to plan, on a problem designed for planning to be necessary. In particular, we show this policy can generalise to budgets not seen at training and continues to scale in performance for budgets larger than the maximum budget trained on. Then we show on two diverse problems known to require planning, Sokoban and the Travelling Salesman Problem, the parameterized Graph Network can outperform popular baselines with similar search budgets.

2 Background

In this section, we first cover Markov Decision Processes and Search within them. We note that search can be represented using graphs and then introduce Graph Networks [5], a framework for representing functions on graphs. With these elements in place, we discuss how popular search methods can be represented with Graph Networks.

2.1 Markov Decision Processes

The problems tackled in this work can be approximated as Markov Decision Processes (MDPs). An MDP can be represented by the tuple (S, A, ρ, r, γ), where S is the set of available states, A is the set of available actions, ρ is the transition function, r is the reward function and γ is the discount factor.

A stochastic policy that can be used to take actions in an MDP can be defined as $\pi : S \times A \rightarrow [0, 1]$. An associated function is the expected discounted sum of rewards of a policy when acting in a particular MDP.

$$n(\pi) = \mathbb{E}_{s_0,a_0,s_1\ldots} \sum_{t=0}^{t=n} \gamma^t r(s_t). \tag{1}$$

Our focus is on finding a policy π^* from the set of realizable policies Π such that the associated $n(\pi^*)$ is maximized:

$$\pi^* = \arg\max_{\pi\in\Pi} n(\pi). \tag{2}$$

This policy can then be used to make decisions in the MDP in order to maximise the reward function of interest.

2.2 Search in MDPs

Tree Search is an approach where before taking an irreversible action in an MDP, a tree containing future actions, states and rewards is constructed. This tree provides more information which can be utilized for selecting the most promising action. In some cases the true MDP model is available, and therefore the tree (containing future states and rewards given certain actions) has no error and can be relied upon [28][16]. In other cases this tree can be estimated using a learnt MDP model [26] but will likely contain errors, making planning more difficult. In this paper, we focus on the case where the model is given in order to isolate the impact of how we leverage tree information. We define an associated maximum budget with search that is the number of MDP transitions used to expand the search tree.

2.3 Graph Networks

Search trees formed by planning in MDPs can be represented using graphs. Planning algorithms are then functions that process this graph in some way. Graph Networks [5] is a framework that accommodates many popular functions for processing graphs. A Graph Network block defines a set of functions which describe the computations for processing a graph and outputting a new graph.

2.3.1 Graphs

We can represent labelled directed graphs with the tuple $G = (u, V, E)$. V represents the set of nodes in the graph, each node can contain node specific features. E is a set of directional edges with associated edge features. Lastly, u represents any graph level features.

2.3.2 Graph Network Block

A Graph Network block (GN block) is composed of six functions: three update functions and three aggregation functions. Not all common graph network functions utilize all of these functions. The first three equations (ϕ^e,ϕ^v,ϕ^u) show the update functions for edges, nodes, graph respectively. The last three equations contain the aggregation functions for local edges, global edges and global nodes.

$$e'_k = \phi^e(e_k, v_{r_k}, v_{s_k}, u), \qquad v'_i = \phi^v(\overline{e}'_i, v_i, u),$$
$$u' = \phi^u(\overline{e}', \overline{v}', u), \qquad \overline{e}'_i = \rho^{e\rightarrow v}(E'_i),$$
$$\overline{e}' = \rho^{e\rightarrow u}(E'), \qquad \overline{v}' = \rho^{v\rightarrow u}(V').$$

Nodes are notated as v_i where i refers to the node id. Edges are notated as e_k where k refers to the edge id. Subscripts r_k and s_k refer to the node ids for the node receiving the message from e_k and the node sending the message, respectively.

2.3.3 GN Block Computation

For the GN block to act as a graph to graph function, each of the 6 functions is in turn triggered, progressively updating node, edge and graph embeddings. This starts with all edge embeddings being updated, the local edge aggregation function being determined for each node, followed by each node being updated using the aggregation of incoming messages. Then the remaining two aggregation functions are used to update 2 graph level variables that represent nodes and edges ($\overline{v}', \overline{e}'$). The full algorithm for updating a graph representation using a GN block is included in Appendix A.

2.4 Planning Methods as Graph Networks

2.4.1 Search Tree Graph Representation

When performing search in a state we can represent the current search state using a graph as shown in Figure 1. The initial state where search begins takes root node placement, and then any actions from this state induce an edge between this state and the successor state. Nodes in this graph have a state feature which corresponds to the raw state representation v_{i_s}. Edges are directional from successor node to predecessor node, since the value of a state is determined by its children nodes (Bellman equation). Edges have the following features, the action taken $e_{j_a} = a$, the reward received $e_{j_r} = r$ and optionally we could add a terminal feature $e_{j_t} = t$ indicating if the environment terminates following this action. Note, even if the state representation is the same for two nodes, both nodes are separately added to a tree. In deterministic environments, each node is connected to a maximum of one child node through an edge feature of a particular action.

2.4.2 GN Block Sequential Computation

In trees, since no cycles can occur, instead of updating all the nodes at each iteration, we can instead start with the lowest nodes in the tree, working up to the highest. The ordering in which nodes are

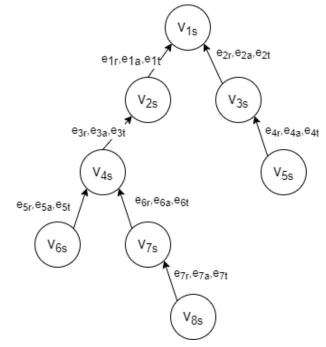


Figure 1: Search state as a graph

required to be updated is specified by a topological sort [22] where all dependencies of a node appear before it in the list. TreeLSTM [29] is an example of a model that processes graphs in this way. See Appendix A for the sequential GN block computation algorithm for trees.

2.4.3 Planning Algorithms

Popular Tree Search algorithms such as MCTS, Alpha Zero [28] and Policy Gradient Search [2] [16] can be represented using Graph Networks with the GN block computation performed sequentially (see Appendix A). This is then followed by a readout upon a subset of the graph nodes. Several possible choices of readout functions are possible [8].

3 Parameterized Graph Network for Planning

Current planning algorithms are heuristic Graph Networks. By fully parameterizing a Graph Network we can in principle train it to outperform these heuristics and tailor planning for specific problems.

In this section, we detail our method for parametrizing a Graph Network based policy that can be trained to make decisions on search trees. We discuss important details regarding how this policy is trained and implemented.

We parameterize our Graph Network with neural networks, forming a Graph Neural Network (GNN) [34]. This GNN is used to update the graph such that node embeddings contain information about the subtrees of states following them. This information can then be leveraged by a policy and value readout function.

3.1 Architecture

3.1.1 Parameterized Graph Network

In the Background section, we detailed the general equations that form a GN block. Here, we specify the parameterized equations we use in this paper for updating search trees. Specifically, we define neural versions of the edge update function, edge aggregation function and the node update function as

$$\phi^e(e_k, v_{r_k}, v_{s_k}, u) = \mathrm{GRU}(f_m(f_e(v_{s_{k_s}}), e_{k_r}, e_{k_t}), v_{s_k}), \tag{3}$$

$$c(e_k) = \mathrm{LeakyReLU}(f_a(e_k)), \tag{4}$$

$$\rho^{e\to v}(E'_i) = \sum_{k: r_k = i} e'_k \frac{e^{c(e'_k)}}{\sum_{k: r_k = i} e^{c(e'_k)}}, \tag{5}$$

$$\phi^v(\bar{e}_i^{'}, v_i, u) = \bar{e}'_i, \tag{6}$$

where f_m, f_e, f_a all represent neural network layers (see Appendix B). Note that $v_{s_{k_s}}$ refers to the state feature of the sender node and e_{k_r},e_{k_t} are the reward and terminal features of e_k respectively.

For edge aggregation, an attention mechanism was used motivated by the Bellman optimality equation, suggesting it is useful to ignore certain edge information when predicting the value of a node using a subtree. A Gated Recurrent Unit [9] was used for the message passing function to reduce the vanishing gradient problem which can happen when propagating messages from lower parts of the tree to the root node [20].

3.1.2 Readout Equations

Once the Graph Network is used to update the node embeddings in the graph, to evaluate the actions at the root node and to predict the value of the current state we use the following readout functions.

$$V_t(v_i) = f_v(\mathrm{GRU}(f_{m_v}(f_{e_v}(v_{i_s}), 0, 0), v_i)), \tag{7}$$

$$c(a, i) = m(a) f_{r_1}(e_{n(a,v_i)}) + (1 - m(a)) f_{r_2}(v_{i_s}, a), \tag{8}$$

$$\pi(a|v_i) = \frac{e^{c(a,i)}}{\sum_{a \in A} e^{c(a,i)}}, \tag{9}$$

with $f_{m_v}, f_{e_v}, f_v, f_{r_1}, f_{r_2}$ all representing neural network layers (See Appendix B). m is an indicator for each action, determining whether there is an edge from this action to a future state (has it been expanded). For actions without a message, we use the logits from an alternative policy operating directly on states. $n(a, v_i)$ refers to the edge id corresponding to taking a from node v_i.

The value function is used for advantage estimation when conducting PPO. It does not necessarily need to also be a function of the entire tree, however, it is likely to be beneficial for estimating value, so we leverage it. In this paper, we consider a value function and policy with separate parameters to ensure stable learning, they could be in principle combined for efficiency.

3.2 Expansion

To expand the search tree, we follow the method used in MCTS and Alpha Zero. An exploration policy is used to take actions, starting from the root node, continuing until the action taken results in a new node being added to the search tree.

We can interpret Alpha Zero's exploration policy as combining two different policies, one which can make decisions on small trees (just the root state) and one which can make decisions on large search trees based on value estimation. The problem here is that a trade-off needs to be determined for how to switch from one to another. This is very hard to predict for each problem and leads to parameter tuning, which can be inefficient for Reinforcement Learning, which is already computationally heavy.

We leverage the fact that the tree policy can make decisions on any size of tree and use this as part of the expansion policy. Expanding by only following the policy limits exploration. We expand using a heuristic that combines the tree policy with the exploration term from MCTS

$$\pi_{exp}(a|v_i) = \pi_r(a|v_i) + c\frac{\sqrt{\sum_{a \in A} N_i}}{N_i + 1} \tag{10}$$

in order to make expansion decisions. N_i is a statistic referring to the number of times that action i has been taken from node v. While we could utilize a method that also learns the best way to search such as in [15] [17] in this paper we keep the expansion heuristic and focus on learning to process the final tree in the best way possible.

In MCTSnets [15], Guez et al. learn to search using supervised learning, but find that using the trained model free policy to make expansions instead of the policy trained to search has little effect on the performance of the final policy. This suggests that with a reasonable expansion policy, most gains come from having a good tree function, therefore learning this is the sole focus of this paper. Note that trees can be reused between consecutive planning steps for efficiency. However, in order to keep the tree distribution the same ensuring stable learning (early states will have trees with fewer nodes) here we do not make use of previous search trees although it would be desirable for efficiency reasons.

3.3 Enforcing Generalization

While in principle the GNN block and readout functions can be used to make decisions on any size of tree, if we train it on one size of tree there is no guarantee that there will be any generalization to different sizes or shapes of trees. Generalization is important firstly as it means users can select the budget they want at test time. Secondly, the tree policy can be used as part of the expansion policy, which by definition is making decisions on smaller trees and so needs to generalize in order to make good expansion decisions. We consider two methods of enforcing generalisation during training. The first refered to as GNN Dropout utilises dropout on the message passing framework, with the idea of training the policy to not over rely on messages and to learn to make good predictions with less information. This does however result in off-policy updates, which can be unstable if the dropout policy is significantly different from the full GNN policy. Secondly, for the method we refer to as GNN, we simply consider training the tree based model on varying budgets. Each episode is generated using a fixed budget, this is to reduce variance added by the uncertainty of what budgets were used later in the episode, which could have a large impact on expected reward. We then introduce a budget conditional value function, which is important as the model likely will have different expected rewards when using different budgets.

3.4 Training

We use Proximal Policy Optimization (PPO) [27] to train the tree policy. Each epoch 5000 tree action pairs are collected, the tree policy and tree value function are then updated. Two versions of the GNN policy were tested, GNN and GNN dropout (See Appendix C.2 for full training details).

3.5 Implementation

This project was implemented using Pytorch [24] along with Open AI Gym [7] for each environment model. Deep Graph Library [32] was used to store expanded search trees and implement Graph Networks. Experiments were run on a machine with an NVIDIA GeForce 1080Ti GPU and Intel Xeon E5-2630 v3 CPU.

4 Experiments

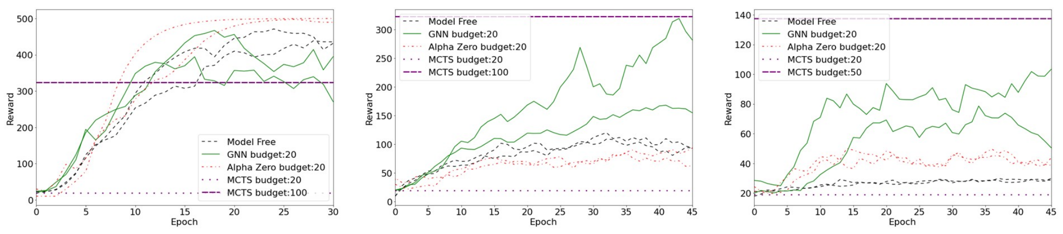
We test the performance of our GNN policy on three diverse environments. The parameters for training with PPO are kept constant throughout each of the experiments. We compare our method to two heuristic planning algorithms, MCTS and an Alpha Zero style method (see Appendix C.3 for implementation details), along with a model free policy that performs no planning. All methods are trained for two seeds 1001 and 1002 with the average performance on a test set of problems evaluated every epoch. For MCTS since low budgets perform very poorly we show performance for a budget of 20 and the performance for a budget that gets close to the best method on that environment. See Appendix C.2 for details regarding training of the GNN policy and specific hyperparameters.

4.1 Cartpole

In order to clearly demonstrate the ability for the GNN to leverage search trees, we adapt a simple problem (Cartpole) such that information at each state is limited (using noise). This then encourages planning since many states do not contain enough information in order to reliably make the optimal action. By adding noise we are simulating how in complex planning problems, learning on states can be difficult, particularly without the right inductive biases. This problem guarantees that planning is important, and therefore is a useful initial problem to understand the capabilities of the GNN and the stability of training it using Reinforcement Learning. We investigate 3 levels of Gaussian noise for our algorithm and baselines, with $\sigma = 0$ being easy to learn a model free policy, $\sigma = 10$ being practically difficult for a model free policy to learn anything and $\sigma = 2$ representing a mid-point between these.

4.2 Sokoban

Sokoban [25] is a problem which planning is known to be useful and has frequently been used as a domain for comparing planning methods. It consists of an agent in a grid world with the goal of pushing boxes onto goal squares. Planning is often required as moves can have long term consequences, such as making the problem unsolvable. We test on small problems in 7x7 grids with 2 boxes (See Appendix C.1).



(a) Cartpole $\sigma = 0$ (b) Cartpole $\sigma = 2$ (c) Cartpole $\sigma = 10$

Figure 2: Cartpole Experiments, lines in the same color show each model trained with different random seeds

4.3 Travelling Salesman Problem

Lastly, we test methods on instances of the Travelling Salesman problem to demonstrate the applicability of our model combinatorial problems. We demonstrate our method on TSP of size 10 to keep training budgets feasible. Without inductive biases, it is very difficult for state based policies to learn anything, and so we train using an encoder-decoder architecture [19] (See Appendix C.1).

5 Results

For all experiments, we train all models twice with 2 different seeds. Graphs then show two lines for each model, each referring to a different seed.

5.1 Cartpole

Figures 2a, 2b and 2c show the results for training on Cartpole with varying levels of noise. We see that for Cartpole with no noise, all methods converge fairly quickly to solving most instances. Alpha Zero is the most stable of the methods, with the GNN method performing slightly worse. Once noise is introduced in Figure 2b we see all baselines performance drop significantly, model free method understandably drops however we also see Alpha Zero significantly suffers. This is likely due to Alpha Zero equally weighting value estimates of nodes in subtrees, which isn't good when our confidence of value may change significantly depending on the state. MCTS does not change as it does not make use of state information. Figure 2c shows the robustness of the GNN policy, where a very large amount of noise makes state information difficult to leverage at all. This shows the GNN policy can leverage only rewards if need be. This demonstrates the GNN policy can adapt depending on the specific problem as to what information needs to be leveraged. We also see that the GNN policy consistently can achieve the same performance as MCTS with much lower budgets.

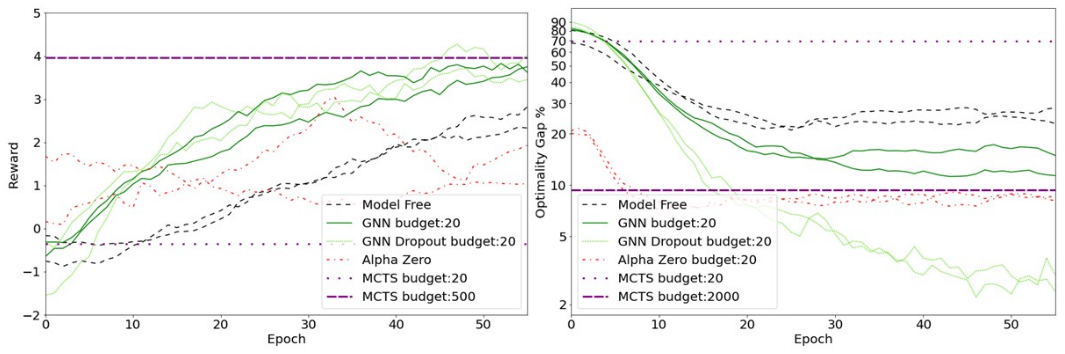
5.2 Practical Planning Problems

5.2.1 Sokoban

The results for the Sokoban experiments are shown in Figure 3a. We see that on an epoch basis, both GNN policies outperform other methods. However, the performance gap to model free is not particularly large. One of the reasons for this gap might be that the GNN performance with budget 0 is fairly weak (See Appendix D). Good expansions of the search tree are critical for constructing a useful tree, and this is likely limiting the performance of the GNN policy. This further highlights the importance of generalisation and suggests that for some problems more compute should perhaps be allocated to training the GNN on smaller budgets relative to larger budgets.

5.2.2 Travelling Salesman Problem

For the travelling salesman problem, we see that both GNN policies strongly outperform the model free policy. The GNN Dropout model also significantly outperforms Alpha Zero. In contrast to Sokoban this is likely down to the successful training of the GNN on budget 0 which also outperforms Alpha Zero (See Appendix D) showing the off-policy Reinforcement Learning updates can be very



(a) Sokoban (b) Travelling Salesman Problem

Figure 3: Experiments for Sokoban and the Travelling Salesman Problem, lines in the same color show each model trained with different random seeds

effective for enforcing generalisation in some problems. MCTS is very inefficient on the TSP, and it takes around a budget of 2000 to match Alpha Zero.

6 Related Work

6.1 Heuristic Search Methods with Learnt Components

Monte Carlo Tree Search (MCTS) [10], inspired by the UCB algorithm [4], is a widely applied method to Markov Decision Processes. All computation is done at test time by accumulating rewards received in the tree and using them to inform future expansions and ultimately a final output. More recently, MCTS has been adapted in [3] and in Alpha Zero [28] by integrating learnt components through a model free policy and a value function that makes search orders of magnitude more efficient. This method has been successfully applied to complex domains with high branching factors, such as Chess and Go. MuZero [26] extends Alpha Zero by not requiring a model of the environment to make expansions, by in parallel learning a model of the environment which is then used for planning. To avoid the need for a search tree to be stored at all, [2] [13] perform planning by updating a policy using policy gradient updates after sampling trajectories. TreeQN and ATreeC [12] introduce an inductive bias into a policy by training a model to expand a search tree in a breadth first like way and then enforce a structure on how the value function should behave. BCTS [11], like Alpha Zero, uses value functions to evaluate a tree but introduces a correction for the problem of poor value estimates for off policy trajectories. In Value Prediction network [21] a subset of actions are simulated to a specific depth, with the resulting tree evaluated by weighting the value function estimates of the nodes in the tree using a heuristic function. While able to deliver strong performance in some domains, each of these methods, due to their heuristic nature, are unlikely to work well in all domains. Methods that rely on using value functions to evaluate search trees suffer from problems such as off-policy node bias [11]. Evaluating subtrees using mean value [3] [28] can be susceptible to outlier errors. Lastly, in some domains (such as TSP) it appears that value functions can be difficult to learn [19] and so methods that rely on using this for policy improvement are unlikely to be stable. These downsides have led to a branch of work that focuses on learning to search instead of relying on crafted heuristics.

6.2 Learnt Search Methods

I2A [33] construct a very particular type of search tree by rolling out a single trajectory for a fixed depth for every action and using an LSTM to encode trajectories, training this using policy gradients. This can be viewed as a specific instance of our method with a very particular exploration function. While it is suitable for problems with small action spaces, this would be very inefficient for large action spaces. MCTSnets [15] train a parameterized graph network to process a more traditional tree distribution like that seen in MCTS where actions can have multiple expansions. MCTSnets is trained using supervised learning, showing in principle message passing architectures can learn to process trees. However, it is not practical due to the requirement of strong targets beforehand. IBP [23] makes use of imagined actions to build a history of future actions, which is then processed using an LSTM to make an improved final decision. The approach is trained using policy gradients and experiments shown an increase of performance when forward-looking budgets are increased and tree based search is used. Tasfi et al. [30] introduce Dynamic Planning Networks (DPN) which learn an environment model that is used for planning where the objective is to maximize information before making an action. Neither DPN nor IBP leverage the tree structure inherent in search trees in any way, opting to encode planning using a recurrent architecture encoding future state progressively.

6.3 Graph Neural Network Architectures

Graph Neural Networks [34] is a growing area of research which tackles learning functions on graphs. Applications of such architectures include predicting the chemical properties of molecules [14] and solving graph based combinatorial problems [19] [6]. Such architectures can also be applied to trees with small changes. TreeLSTMs [29] have been applied to semantic representations, while MCTSnets process search trees in Sokoban using message passing neural networks. I2A processes very specific trees where there are only single trajectories for each action, so no message aggregation is needed, and each trajectory can simply be encoded using an LSTM [33].

6.4 Combining Learning and Search in Combinatorial Optimization

Combinatorial Optimization is an example of a domain where search is important for achieving strong performance. There have been numerous approaches to applying Graph Neural networks to routing problems [31][18], Kool et al [19] show that significant performance gains can be made through adding very simple search methods to a GNN policy, such as stochastic rollouts and beam search to improve performance. Methods similar to Alpha Zero have also been applied to TSP problems [1]. Policy Gradient Search was used in [16] to improve the performance of a model free policy at test time for routing problems, making it efficient by only updating a small number of parameters at test time.

7 Conclusion

Firstly, we frame existing planning methods as Graph Networks. These Graph Networks are heuristically designed to make decisions on trees for planning problems. We introduce a parameterized Graph Network which we show can be trained using Reinforcement Learning to perform planning and to outperform existing Graph Network heuristics. We demonstrate that this model can be trained on a wide range of environments with the parameters. When compared to baselines for the same number of updates, the GNN policy outperforms other methods on Cartpole with Noise, Sokoban and the Travelling Salesman problem.

We also show that the Graph Neural Network policy demonstrates generalisation to other search budgets beyond what it was trained on. Performance is stable for budgets not seen during training, and even improves when given budgets larger than those seen during training, a strong indication that it has learnt a generalisable planning method and has not overfit to the large amount of information in trees.

An important application that this work can be utilized in that was not explored in this paper was the use of imperfect environment models for planning. Such models are difficult to plan with as they can contain many mistakes and the optimal planning method is tied closely to how well the environment model has trained, which may be unstable between seeds. The results of this paper suggest that Graph Neural Networks may be a promising approach to learning how to leverage imperfect planning trees. With a model with no predictive power, the GNN could learn to utilize only the root state, but with better models it could learn to utilise the tree. Our experiments in noisy Cartpole demonstrate that the GNN policy is capable of adapting based on the available information in the search tree and so this could be an important application of this work. In conclusion, this paper highlights that parameterized graph networks are a very promising method for learning to plan that can learn to outperform heuristics at planning using Reinforcement Learning.

Acknowledgments and Disclosure of Funding

This research was partly funded by the funded by LIFT-project 019.011 which is partly financed by the Dutch Research Council (NWO).

This research was partly funded by Flemish Government under the "Onderzoeksprogramma Artifciële Intelligentie (AI) Vlaanderen".

References

[1] Kenshin Abe, Zijian Xu, Issei Sato, and Masashi Sugiyama. Solving NP-hard problems on graphs with extended Alphago Zero. *arXiv preprint arXiv:1905.11623*, 2019.

[2] Thomas Anthony, Robert Nishihara, Philipp Moritz, Tim Salimans, and John Schulman. Policy gradient search: Online planning and expert iteration without search trees. *arXiv preprint arXiv:1904.03646*, 2019.

[3] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, 2017.

[4] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[5] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[6] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *International Conference on Learning Represenations, Workshop Track*, 2017.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[8] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[9] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[10] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[11] Gal Dalal, Assaf Hallak, Steven Dalton, Shie Mannor, Gal Chechik, et al. Improve agents without retraining: Parallel tree search with off-policy correction. *Advances in Neural Information Processing Systems*, 34:5518–5530, 2021.

[12] Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. TreeQN and ATreeC: Differentiable tree-structured models for deep reinforcement learning. In *International Conference on Learning Representations*, 2018.

[13] Arnaud Fickinger, Hengyuan Hu, Brandon Amos, Stuart Russell, and Noam Brown. Scalable online planning via reinforcement learning fine-tuning. *Advances in Neural Information Processing Systems*, 34:16951–16963, 2021.

[14] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1263–1272, 2017.

[15] Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to search with MCTSNets. In *International conference on machine learning*, pages 1822–1831, 2018.

[16] André Hottung, Yeong-Dae Kwon, and Kevin Tierney. Efficient active search for combinatorial optimization problems. *arXiv preprint arXiv:2106.05126*, 2021.

[17] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Vime: Variational information maximizing exploration. *Advances in neural information processing systems*, 29, 2016.

[18] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017.

[19] Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

[20] Denis Lukovnikov, Jens Lehmann, and Asja Fischer. Improving the long-range performance of gated graph neural networks. *arXiv preprint arXiv:2007.09668*, 2020.

[21] Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. *Advances in neural information processing systems*, 30, 2017.

[22] Chaoyi Pang, Junhu Wang, Yu Cheng, Haolan Zhang, and Tongliang Li. Topological sorts on DAGs. *Information Processing Letters*, 115(2):298–301, 2015.

[23] Razvan Pascanu, Yujia Li, Oriol Vinyals, Nicolas Heess, Lars Buesing, Sebastien Racanière, David Reichert, Théophane Weber, Daan Wierstra, and Peter Battaglia. Learning model-based planning from scratch. *arXiv preprint arXiv:1707.06170*, 2017.

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[25] Max-Philipp B. Schrader. gym-sokoban. https://github.com/mpSchrader/gym-sokoban, 2018.

[26] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering Atari, Go, Chess and Shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[28] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[29] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing*, 2015.

[30] Norman Tasfi and Miriam Capretz. Dynamic planning networks. In *International Joint Conference on Neural Networks*, 2021.

[31] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

[32] Minjie Yu Wang. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*, 2019.

[33] Théophane Weber, Sébastien Racanière, David P Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. In *Advances in neural information processing systems*, 2017.

[34] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2020.