8 Appendix

8.1 A Graph Networks

We can represent popular tree search algorithms such as MCTS and Alpha Zero as Graph Networks. The following details how these search algorithms can be represented within the Graph Network framework, along with a set of readout functions (which readout information about updated nodes).

8.1.1 MCTS

Nodes are initialized with 2 dimensional embedding features. The first dimension is used to store the total reward experienced from this state, and the second dimension stores the total visits. Together, these statistics can be used to calculate the mean reward from a state. These can then be used to choose between actions by reading the mean reward for the resulting node of each action (accounting for the immediate reward for moving to this state).

8.1.2 Graph Network

$$\phi^e(e_k, v_{r_k}, v_{s_k}, u) = v_{s_k} + [e_{kr} \cdot v_{s_k}[2], 0]^T \tag{11}$$

$$\rho^{e\to v}(E_i') = \sum_{k:r_k=i} (e_k') \tag{12}$$

$$\phi^v(\overline{e}_i', v_i, u) = [V_{rollout}(v_i), 1]^T + \overline{e}_i' \tag{13}$$

8.1.3 Readout Functions

$$s_{exp}(a|v_i) = \frac{v_{n(a,v_i)}[1]}{v_{n(a,v_i)}[2]} + e_{k(a,v_i)_r} + c \cdot \sqrt{\frac{\ln \sum_{a\in A}(v_{n(a,v_i)}[2])}{v_{n(a,v_i)}[2]}} \tag{14}$$

$$s(a|v_i) = \frac{v_{n(a,v_i)}[2]}{\sum_{a\in A}(v_{n(a,v_i)}[2])} \tag{15}$$

$$\pi_{exp} = \arg\max(s_{exp}(a|v_i)) \tag{16}$$

$$\pi = \arg\max(s(a|v_i)) \tag{17}$$

8.1.4 Details

We use $[j]$ to define the dimension j of an embedding. We define $v_{n(a,v_i)}$ to be the node ID reached when taking action a from the node v_i. Likewise, $k(a, v_i)$ defines the edge ID for the edge representing the action taken from the node v_i.

Note, for the exploration term to be able to compute estimated action values we need at least one visit for each action, so all actions are taken at least once, before this score function which trades off between exploration and exploitation is used. Note that this is one form of MCTS where the final action chosen is based on the most visited action, however others exist, such as using a readout of the highest action value.

Since UCB, upon which MCTS is based is designed to trade off exploitation and exploration when action values are between 0 and 1, in order to use this for general environments where this is not the case, scaling needs to be used. We leave the scaling of action values out of the equations for simplicity but a common method used in MuZero [26] is to scale rewards based on the maximum and minimum action values seen in the current search tree so far.

8.1.5 Alpha Zero

The Graph Network and readout functions for Alpha Zero are very similar to MCTS with a few small but important differences. Firstly, instead of using a rollout function to estimate the value of a node, a value function is used (based on the state information at that node). The exploration function is also biased with a model free policy (mapping states to actions directly). Lastly, a slightly different exploration function is often used [26]. Nodes are initialised with three dimensional embedding features. The last dimension we use for the storage of the model free policy for a state.

8.1.6 Graph Network

$$\phi^e(e_k, v_{r_k}, v_{s_k}, u) = [v_{s_k}[1], v_{s_k}[2], 0] + [e_{kr} \cdot v_{s_k}[2], 0, 0]^T \tag{18}$$

$$\rho^{e \to v}(E_i') = \sum_{k:r_k=i} (e_k') \tag{19}$$

$$\phi^v(\overline{e}_i', v_i, u) = [V_{NN}(v_{i_s}), 1, \pi_{mf}(a|v_{i_s})]^T + \overline{e}_i' \tag{20}$$

8.1.7 Readout Functions

$$exp(a|v_i) = \frac{\sqrt{\sum_{a \in A}(v_{n(a,v_i)}[2])}}{1 + v_{n(a,v_i)}[2]} \cdot (c_1 + \ln(\frac{\sum_{a \in A}(v_{n(a,v_i)}[2]) + c_2 + 1}{c_2})) \tag{21}$$

$$s_{exp}(a|v_i) = \frac{v_{n(a,v_i)}[1]}{max(1, v_{n(a,v_i)}[2])} + v_i[3] \cdot exp(a|v_i) \tag{22}$$

$$s(a|v_i) = \frac{v_{n(a,v_i)}[2]}{\sum_{a \in A}(v_{n(a,v_i)}[2])} \tag{23}$$

$$\pi_{exp} = \arg\max(s_{exp}(a|v_i)) \tag{24}$$

$$\pi(a|v_i) = \frac{s(a|v_i)^{1/T}}{\sum_{a \in A} s(a|v_i)^{1/T}} \tag{25}$$

8.1.8 Details

T is a temperature parameter that controls the level of greediness of the policy and is often adjusted during training. Like MCTS action values here also require scaling. However, Alpha Zero implementations usually do not require that all actions must be taken to ascertain an action value estimate, and instead actions that are not taken are initialised to 0. Therefore, if the model free policy determines an action is very unlikely, it may not even be explored once.

8.2 B Model Architectures

Here, we expand on the details of the GNN block and readout functions that make up the GNN policy.

8.2.1 Graph Network Block

Three neural network architectures were used as part of the GN block (f_e, f_m, f_s).

For the state embedding Network f_e , in Sokoban and Cartpole the same architecture was used. A 3 layer feedforward neural network with ReLu activation and the following layer architecture was used: [64,64,32]. In the Travelling Salesman problem, the encoder architecture from Kool et al[19] was used. We remove the decoder architecture, as we only require an embedding of the state such that the value can be predicted. Firstly, the graph is encoded using a GNN (we reduce the number of layers to 1 for computational reasons). A global graph embedding called the context h_c is calculated (mean of all the graph embeddings), which is updated using Multi-head attention following Kool et al. This

embedding then is used for the state embedding. All state embedding functions embed states to a dimension = 32. For the Message Network f_m a single layer (size 32) feedforward neural network was used with ReLu activation. For the Attention Network f_a a single layer feedforward neural network (size 32) with no bias and no activation was used. The GRUCell architecture implemented in pytorch was used with input size = 32 and hidden size = 32.

8.2.2 Readout Functions

There are 3 neural network functions that make up the readout equations (f_v, f_{r_1}, f_{r_2}). For the value readout f_v we use a budget conditional value function. First we scale the budget between the maximum and minimum budget used during training and project it to 32 dimensions using a single layer neural network with ReLu activations. Concatenating with the tree encoding [budget embedding, tree encoding] we apply a single layer neural network to output a value prediction. For the first readout function f_{r_1} a three layer neural network with architecture [32,32,1] for the first two layers ReLu activations were used. For the second readout function f_{r_2} ,this corresponds to the part of the policy that directly maps states to action logits. For Cartpole and Sokoban here we embed states with the same embedding function as in the GN block followed by a single layer neural network mapping the embeddings to a dimension equal to the number of actions. For the travelling salesman problem we utilised the full encoder-decoder architecture in Kool et al. with a single layer GNN in the encoder function.

8.3 C Experiments

8.3.1 C.1 Environments

8.3.2 Cartpole with Noise

The Cartpole implementation by OpenAI was used, with the specific version being 'CartPole-v1'. This implementation was then modified to add noise to each state in the environment. Within one phase of planning, the noise stays constant for each state. To each state variable, Gaussian noise with $\mu = 0$ and $\sigma = 0, 2, 10$ was added. This makes utilising state information more difficult and policies have to rely on planning.

8.3.3 Sokoban

The publicly available implementation of Sokoban [25] was used in this paper. Environment specifications used to generate problems to solve were

- room dimension = (6,6)
- maximum environment steps = 20
- number of boxes = 2.

These specifications were chosen as they are the hardest level of problems where a model free policy trained using PPO out of the box can learn to improve performance in a computationally feasible number of environment transitions. To solve harder levels problem specific architectures or Reinforcement Learning methods are required, which was not the focus of this paper.

8.3.4 Travelling Salesman Problem

We implemented the Travelling Salesman Problem using the OpenAI gym environment. Each state consists of the coordinates of the cities in the problem defined on a unit square with coordinates $x, y, \in [0, 1]$, the current location of the agent, the location the agent needs to return to and the remaining cities to visit. At each step, the agent receives a reward equal to -distance travelled. At each action step, the agent has to choose a new city to visit. In this paper, we focus on solving problems of size 10.

8.3.5 C.2 Training

In this paper, we implement a standard version of Proximal Policy Iteration consisting of two phases. Trajectory collection and model training. For the trajectory collection phase the most recently updated

policy is used to take actions in the environment using a specific search budget or range of budgets specified. This is performed by 20 separate cpu workers, each collecting $\frac{5000}{20}$ transitions. Once 5000 transitions are collected we switch to model updating. For each tree (generalisation from state), action pair, an advantage estimation $\hat{A}_t$ is calculated using Generalized Advantage estimation. The policy and value losses are defined by the following equations. Note that states are replaced with tree variables T_t to indicate that policies and value functions make decisions on trees instead of states (s_t).

$$r_t(\theta) = \frac{\pi_\theta(a_t|T_t)}{\pi_{\theta_{old}}(a_t|T_t)} \tag{26}$$

$$Loss_\pi(\theta) = E_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{27}$$

The value function loss is as follows, with the value targets being the reward to go.

$$Loss_v(\theta_v) = (V_{\theta_v}(T_t) - V_t^{targ})^2 \tag{28}$$

The policy and value function are then updated for 40 iterations.

When training according to GNN Dropout the policy is executed with a dropout variable d indicating a probability for messages to be deleted. In this work, we only consider using dropout of $d = 1$. With this dropout policy, we can then define the loss function.

$$d_t(\theta, d) = \frac{\pi_\theta(a_t|T_t, d)}{\pi_{\theta_{old}}(a_t|T_t)} \tag{29}$$

$$Loss_{dropout}(\theta, d) = E_t[min(d_t(\theta, d)\hat{A}_t, clip(d_t(\theta, d), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{30}$$

In standard PPO at the first iteration the policy update is on policy, with future updates progressively becoming more and more off-policy. This dropout update is a more significant off-policy update compared to the standard PPO update since at the first update the dropout policy is not necessary close to the full policy. For Cartpole with Noise the difference between the performance in these two policies lead to unstable updates, however in the two practical problems (Sokoban and TSP) this problem was not as severe.

Overall the GNN policy is trained with the following loss

$$Loss = Loss_\pi + Loss_v \tag{31}$$

The GNN Dropout policy is trained with the following Loss

$$Loss = Loss_\pi + Loss_v + Loss_{dropout}(\theta, d) \tag{32}$$

The following hyperparameters were used for PPO

- steps per epoch = 5000
- gamma = 0.9
- clip ratio = 0.2
- policy learning rate = 1e-3
- value function learning rate = 5e-3
- update iterations per epoch = 40
- lambda = 0.97
- max episode length = 1000

After each epoch the GNN policies and baselines were evaluated on a fixed number of games to measure performance during learning. The number of games used for evaluation were 10,40,100 for Cartpole, Sokoban and Travelling Salesman Problem respectively.

8.3.6 C.3 Baselines

8.3.7 Model Free

The model free policy is just a specific instance of the GNN policy with $budget = 0$.

8.3.8 MCTS

MCTS was performed using the Graph Network functions specified in section A and was implemented using Deep Graph Library. At each leaf node, a rollout is performed using a random policy in order to estimate the value of the node. In order for the exploration explotation trade specified by UCB to be utilized the action values need to be scaled between 0 and 1. We used the method of scaling from MuZero [26].

$$\overline{Q}(s^{k-1}, a^k) = \frac{Q(s^{k-1}, a^k) - min_{s,a \in Tree} Q(s,a)}{max_{s,a \in Tree} Q(s,a) - min_{s,a \in Tree} Q(s,a)} \tag{33}$$

Where k is the current time step in the environment. We set the exploration term c at the theoretical motivated value of $c = \sqrt{2}$. Note that we count random rollout environment transitions as part of the search budget.

8.3.9 Alpha Zero

The architecture for processing search trees by Alpha Zero follows the method in Schrittwieser et al. [26] , detailed in Appendix A. There are slight differences to how we train Alpha Zero, which often involes the use of complex replay buffer strategies. We train Alpha Zero using the same framework as Proximal Policy iteration. A period of trajectory collection followed by a model training. The only difference in the model training compared to GNN is that Alpha Zero uses a different policy loss function. It only updates a model free policy rather than the overall search policy directly.

The value loss error is kept the same as

$$Loss_v(\theta_v) = (V_{\theta_v}(s_t) - V_t^{targ})^2. \tag{34}$$

However, the policy loss is the following

$$Loss(\theta) = KL(\pi(a|T_t), \pi_{mf}(a|s_t)). \tag{35}$$

This trains the model free policy in a supervised fashion towards the better tree policy. The tree policy gets implicitly updated by its components (π_{mf} and the value function) being updated, which improves the overall search policy.

In some implementations a replay buffer is used for Alpha Zero which we do not utilise for any of the implementations tested in this paper. This ensures a level playing field between GNN training and Alpha Zero training, reduces the overfitting required for each problem that would require different replay buffer parameters.

The hyperparameters c_1 and c_2 used as part of the exploration policy were set the same as commonly used values in the literature [26].

- $c_1 = 1.25$
- $c_2 = 19652$

Alpha Zero also suffers from problems where if the model free policy becomes too deterministic then search is limited and the most visited action also ends up being the model free output, leading to a vicious cycle that destroys learning. This is something that can make Alpha Zero hard to train in practice. However, we utilise the current popular method to mitigate this of adding Dirichlet noise to the model free policy for the root node.

$$\pi_{mf_{root}} = 0.75 * \pi_{mf} + 0.25 * Dirichlet(0.3) \tag{36}$$

Additionally, at test time, Alpha Zero is run greedily.

8.4 D Ablations

One important aspect discussed in the paper was the importance of generalisation for the GNN policy. We find that the ability to generalise to different sizes of search trees is different between problems and can have a severe effect on the GNN policy. As shown in Sokoban see Figure 4 the GNN policy slightly outperforms the GNN model free policy. We see that the performance of the GNN on budget 0 is actually fairly weak for this problem and this is likely holding back the GNN policy since rollouts will not be particularly high quality.

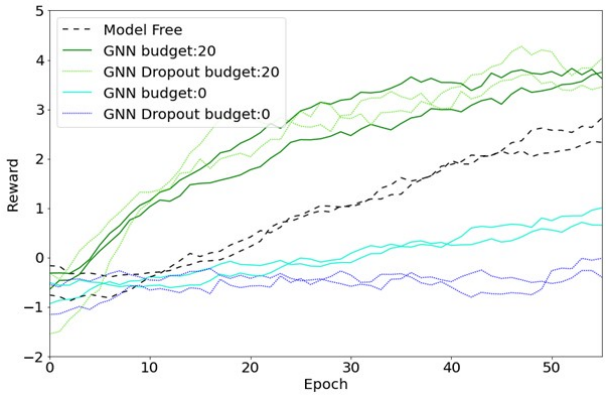

Figure 4: Results for Sokoban, highlighting performance of GNN on budget 0 during training

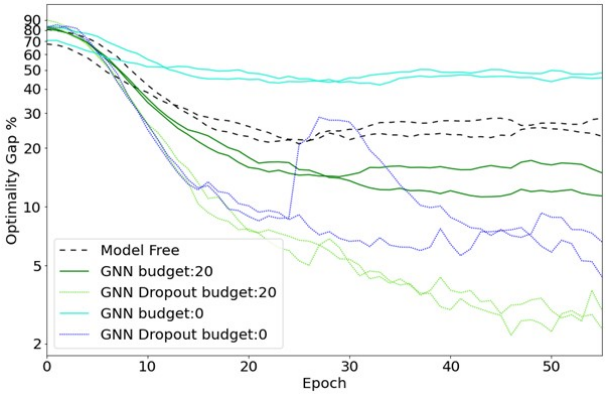

Figure 5: Results for Travelling Salesman Problem, highlighting performance of GNN on budget 0 during training

On the other hand, for the Travelling Salesman Problem, we see that the GNN with dropout learns a very strong policy on budget 0 and an even stronger GNN policy. This shows that for certain environments, the dropout method is very effective for ensuring generalisation. It may be counterintuitive that the GNN on budget 0 outperforms the model free policy, however this is possible as updating on budget 0 while acting with a higher budget has the benefit of being able to use the tree based value function which provide more accurate advantage estimations.