POLICY-BASED SELF-COMPETITION FOR PLANNING PROBLEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

AlphaZero-type algorithms may stop improving on single-player tasks in case the value network guiding the tree search is unable to approximate the exact outcome of an episode sufficiently well. One technique to address this problem is transforming the single-player task through self-competition. The main idea is to compute a scalar baseline from the agent's historical performances and to reshape an episode's reward into a binary output, indicating whether the baseline has been exceeded or not. However, this baseline only carries limited information for the agent about strategies how to improve. We leverage the idea of self-competition and directly incorporate a historical policy into the planning process instead of its scalar performance. Based on the recently introduced Gumbel AlphaZero (GAZ), we propose our algorithm GAZ 'Play-to-Plan' (GAZ PTP), in which the agent learns to find strong trajectories by planning against possible strategies of its past self. We show the effectiveness of our approach in two well-known combinatorial optimization problems, the Traveling Salesman Problem and the Job-Shop Scheduling Problem. With only half of the simulation budget for search, GAZ PTP consistently outperforms all selected single-player variants of GAZ.

1 INTRODUCTION

One of the reasons for the success of AlphaZero (Silver et al., 2017) is the use of a policy and value network to guide the Monte Carlo tree search (MCTS) to decrease the search tree's width and depth. Trained on state-outcome pairs, the value network develops an 'intuition' to tell from single game positions which player might win. By normalizing values in the tree to handle changing reward scales (Schadd et al.; Schrittwieser et al., 2020), AlphaZero's mechanisms can be applied to single-agent (or single-player) tasks. Although powerful, the MCTS relies on value approximations which can be hard to predict (van Hasselt et al., 2016; Pohlen et al., 2018). Furthermore, without proper normalization, it can be difficult for value function approximators to adapt to small improvements in later stages of training. In recent years there has been an increasing interest in learning sequential solutions for combinatorial optimization problems (COPs) from zero knowledge via deep reinforcement learning. Particularly strong results have been achieved with policy gradient methods by using variants of *self-critical training* (Rennie et al., 2017; Kool et al., 2018; Kwon et al., 2020) where it is avoided to learn a value function at all. By baselining the gradient estimate with the outcome of rolling out a current or historical policy, actions are reinforced by how much better (or worse) an episode is compared to the rollouts. Something similar is achieved in MCTS-based algorithms for single-player tasks by computing a scalar baseline from the agent's historical performance. The reward of the original task is reshaped to a binary ± 1 outcome indicating whether an episode has exceeded this baseline or not (Laterre et al., 2018; Schmidt et al., 2019; Mandhane et al., 2022). This *self-competition* brings the original single-player task closer to a two-player game and bypasses the need for in-tree value scaling during training. The scalar baseline against which the agent is planning must be carefully chosen as it should neither be too difficult nor too easy to outperform. Additionally, in complex problems, a single scalar value holds limited information about the instance at hand and the agent's strategies for reaching the threshold performance.

In this paper, we follow the idea of self-competition in AlphaZero-style algorithms for deterministic single-player sequential planning problems. Inspired by the original powerful 'intuition' of AlphaZero to evaluate board positions, we propose to evaluate states in the value function not by comparing them against a scalar threshold but directly against states at similar timesteps coming

from a historical version of the agent. The agent learns by reasoning about potential strategies of its past self. We summarize our contributions as follows:

(i) We assume that in a self-competitive framework, the scalar outcome of a trajectory ζ under some baseline policy is less informative for tree-based planning than ζ's intermediate states. Our aim is to put the flexibility of rollouts in self-critical training into a self-competitive framework while maintaining the policy's information in intermediate states of the rollout. We motivate this setup from the viewpoint of advantage baselines, show that policy improvements are preserved, and arrive at a simple instance of gamification where two players start from the same initial state, take actions in turn and aim to find a better trajectory than the opponent.

(ii) We propose the algorithm GAZ Play-to-Plan (GAZ PTP) based on Gumbel AlphaZero (GAZ), the latest addition to the AlphaZero family, introduced by Danihelka et al. (2022). An agent plays the above game against a historical version of itself to improve a policy for the original single-player problem. The idea is to allow only one player in the game to employ MCTS and use the opponent's policy for stochastical branching. Policy improvements obtained through GAZ's tree search propagate from the game to the original task, yielding a uniform framework.

We show the superiority of GAZ PTP over single-player variants of GAZ on two COP classes, the Traveling Salesman Problem and the standard Job-Shop Scheduling Problem. We compare GAZ PTP with different single-player variants of GAZ, with and without self-competition. We consistently outperform all competitors even when granting GAZ PTP only half of the simulation budget for search. In addition, we reach competitive results for both problem classes compared to state-of-the-art benchmarks in the literature.

2 Related work

Gumbel AlphaZero In GAZ, Danihelka et al. (2022) redesigned the action selection mechanisms of AlphaZero and MuZero (Silver et al., 2017; Schrittwieser et al., 2020). At the root node, actions to explore are sampled without replacement using the Gumbel-Top-k trick (Vieira; Kool et al., 2019b). Sequential halving (Karnin et al., 2013) is used to distribute the search simulation budget among the sampled actions. The singular action remaining from the halving procedure is selected as the action to be taken in the environment. At non-root nodes, action values of unvisited nodes are completed by a value interpolation, yielding an updated policy. An action is then selected deterministically by matching this updated policy to the visit count distribution (Grill et al., 2020). This procedure theoretically guarantees a policy improvement for correctly estimated action values, both for the root action selection and the updated policy at non-root nodes. Consequently, the principled search of GAZ works well even for a small number of simulations, as opposed to AlphaZero, which might perform poorly if not all actions are visited at the root node. Similarly to MuZero, GAZ normalizes action values with a min-max normalization based on the values found during the tree search to handle changing and unbounded reward scales. However, if for example a node value is overestimated, the probability of a good action might be reduced even when all simulations reach the end of the episode. Additionally, value function approximations might be challenging if the magnitude of rewards changes over time or must be approximated over long time horizons (van Hasselt et al., 2016; Pohlen et al., 2018).

Self-critical training Rennie et al. (2017) introduce self-critical training, a policy gradient method that baselines the REINFORCE (Williams, 1992) gradient estimator with the reward obtained by rolling out the current policy greedily. As a result, trajectories outperforming the greedy policy are given positive weight while inferior ones are suppressed. Self-critical training eliminates the need for learning a value function approximator (and thus all innate training challenges) and reduces the variance of the gradient estimates. The agent is further provided with an automatically controlled curriculum to keep improving. Self-critical methods have shown great success in neural combinatorial optimization. Kool et al. (2018) use a greedy rollout of a periodically updated best-so-far policy as a baseline for routing problems. Kwon et al. (2020) and Kool et al. (2019a) bundle the return of multiple sampled trajectories to baseline the estimator applied to various CPOs. The idea of using rollouts to control the agent's learning curriculum is hard to transfer one-to-one to MCTS-based algorithms, as introducing a value network to avoid full Monte Carlo rollouts in the tree search was exactly one of the great strengths in AlphaZero (Silver et al., 2016).

Self-competition Literature is abundant on learning by creating competitive environments for (practical) single-player problems, e.g. Bansal et al. (2017); Sukhbaatar et al. (2018); Zhong et al. (2019); Xu & Lieberherr (2020); Wang et al. (2020); Göttl et al. (2022). Reward mechanisms based on *self-competition* for AlphaZero-type algorithms are different but comparable to self-critical training in policy gradient methods. A scalar baseline is computed from the agent's historical performance against which the current agent needs to compete. A reward of ± 1 is returned at the end of an episode, depending on whether the trajectory performed better than the baseline, eliminating the need for value scaling and normalization. An iterative process of improvement is created by continuously or periodically updating the baseline according to the agent's performance. As in self-critical training, the baseline provides the agent automatically with the right curriculum to improve. We use the term *self-competition* in the context of competing against historical performance to distinguish from *self-play* as in board games, where usually the latest agent version is used. Ranked Reward (Laterre et al., 2018) stores historical rewards in a buffer which is used to calculate a threshold value based on a predetermined percentile. Mandhane et al. (2022) use an exponential moving average-based scheme for the threshold and apply self-competition to maximize a constrained objective. Beloborodov et al. (2020) use a similar approach as Ranked Reward, but also allow non-binary rewards in $[-1, 1]$. The described self-competition based on combining historical performances cannot differentiate between problem instances. If an instance's difficulty is higher than usual, in extreme cases the agent might obtain a reward of -1 even though the found solution is optimal, thus adding significant noise to the reshaped reward (Bello et al., 2016; Laterre et al., 2018).

3 Preliminaries

3.1 Problem Formulation

We consider undiscounted Markov decision processes (MDPs) of finite problem horizon T with state space $\mathcal{S}$, a finite action space $\mathcal{A}$, a reward function $r\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and an initial state distribution ρ_0. The goal is to find a state-dependent policy π which maximizes the expected total reward

$$G^\pi := \mathbb{E}_{s_0 \sim \rho_0} \mathbb{E}_{\zeta_0 \sim \eta^\pi(\cdot|s_0)} \left[\sum_{t=0}^{T-1} r(s_t, a_t) \right],$$

where $\eta^\pi(\cdot|s_t)$ is the distribution over possible trajectories $\zeta_t = (s_t, a_t, \ldots, s_{T-1}, a_{T-1}, s_T)$ obtained by rolling out policy π from state s_t at timestep t up to the finite horizon T. We use the subscript t in actions and states to indicate the timestep index. For an action $a_t \sim \pi(\cdot|s_t)$ in the trajectory, the state transitions deterministically to $s_{t+1} = F(s_t, a_t)$ according to some known deterministic state transition function $F\colon \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. We shortly write $a_t s_t := F(s_t, a_t)$ for a state transition in the following. With abuse of notation, we write $r(\zeta_t) := \sum_{l=0}^{T-1-t} r(s_{t+l}, a_{t+l})$ for the accumulated return of a trajectory. As usual, we denote by

$$V^\pi(s_t) := \mathbb{E}_{\zeta_t \sim \eta^\pi(\cdot|s_t)} \left[r(\zeta_t) \right] \qquad Q^\pi(s_t, a_t) := r(s_t, a_t) + \mathbb{E}_{\zeta_{t+1} \sim \eta^\pi(\cdot|a_t s_t)} \left[r(\zeta_{t+1}) \right]$$
$$A^\pi(s_t, a_t) := Q^\pi(s_t, a_t) - V^\pi(s_t)$$

the state-value function, action-value function and advantage function w.r.t. the policy π. Furthermore, for two policies π and μ, we define

$$V^{\pi,\mu}(s_t, s'_l) := \mathbb{E}_{\substack{\zeta_t \sim \eta^\pi(\cdot|s_t) \\ \zeta'_l \sim \eta^\mu(\cdot|s'_l)}} \left[r(\zeta_t) - r(\zeta'_l) \right]$$

as the expected difference in accumulated rewards taken over the joint distribution of $\eta^\pi(\cdot|s_t)$ and $\eta^\mu(\cdot|s'_l)$. We define $Q^{\pi,\mu}(s_t, s'_l; a_t) := r(s_t, a_t) + V^{\pi,\mu}(a_t s_t, s'_l)$ and $A^{\pi,\mu}(s_t, s'_l; a_t) := Q^{\pi,\mu}(s_t, s'_l; a_t) - V^{\pi,\mu}(s_t, s'_l)$ analogously.

The following lemma follows directly from the definitions. It tells us that we can work with the policy μ as with any other scalar baseline, and that policy improvements are preserved. A proof is given in Appendix A.1.

Lemma 1 *Let $\pi, \tilde{\pi}$, and μ be state-dependent policies. For any states $s_t, s'_l \in \mathcal{S}$ and action $a_t \in \mathcal{A}$, we have*

$$A^\pi(s_t, a_t) = A^{\pi,\mu}(s_t, s'_l; a_t), \text{ and} \tag{1}$$

$$\Big[\sum_{a_t} \tilde{\pi}(a_t|s_t) Q^\pi(s_t, a_t) \Big] - V^\pi(s_t) = \Big[\sum_{a_t} \tilde{\pi}(a_t|s_t) Q^{\pi,\mu}(s_t, s'_l; a_t) \Big] - V^{\pi,\mu}(s_t, s'_l). \tag{2}$$

3.2 MOTIVATION FOR THE TWO-PLAYER GAME

We now consider MDPs with episodic rewards, i.e., we assume $r(s_t, a_t) = 0$ for $t < T - 1$. The policy target and action selection within the search tree in GAZ is based on the following policy update: At some state s, given logits of the form $\text{logit}^\pi(a)$ for an action a predicted by a policy π, an updated policy π' is obtained by setting

$$\text{logit}^{\pi'}(a) := \text{logit}^\pi(a) + \sigma(\hat{A}^\pi(s, a)). \tag{3}$$

Here, σ is some monotonically increasing linear function and $\hat{A}^\pi(s, a) := \hat{Q}^\pi(s, a) - \hat{V}^\pi(s)$ is an advantage estimation, where $\hat{V}^\pi(s)$ is a value approximation based on the output of a value network and $\hat{Q}^\pi(s, a)$ is a Q-value approximation coming from the tree search statistics (and is set to $\hat{V}^\pi(s)$ for unvisited actions). Note that (3) differs from the presentation in Danihelka et al. (2022) due to the additional assumption that σ is linear, which matches its practical choice (see Appendix A.2 for a derivation). The update (3) is proven to provide a policy improvement but relies on the correctness of the value approximations $\hat{V}^\pi$.

Our aim is to couple the ideas of self-critical training and self-competition using a historical policy. Firstly, the baseline should adapt to the difficulty of an instance via rollouts as in self-critical training. Secondly, we want to avoid losing information about intermediate states of the rollout as when condensing it to a scalar value.

Consider some initial state $s_0 \sim \rho_0$, policy π, and a historical version μ of π. Let $\zeta_0^{\text{greedy}} = (s_0, a'_0, \ldots, s'_{T-1}, a'_{T-1}, s'_T)$ be the trajectory obtained by rolling out μ greedily. We propose to plan against ζ_0^{greedy} timestep by timestep instead of baselining the episode with $r(\zeta_0^{\text{greedy}})$, i.e., we want to approximate $V^{\pi,\mu}(s_t, s'_t)$ for any state s_t. By Lemma 1, we have for an updated policy π'

$$\sum_{a_t} \pi'(a_t|s_t) Q^\pi(s_t, a_t) \geq V^\pi(s_t) \iff \sum_{a_t} \pi'(a_t|s_t) Q^{\pi,\mu}(s_t, s'_t; a_t) \geq V^{\pi,\mu}(s_t, s'_t). \tag{4}$$

So if π' improves policy π baselined by μ, then π' improves π in the original MDP. Furthermore, $\hat{A}^\pi(s_t, a_t)$ in (3) can be swapped with an approximation of $\hat{A}^{\pi,\mu}(s_t, s'_t; a_t)$ for the update in GAZ. Note that $V^{\pi,\mu}(s_t, s'_t) = \mathbb{E}_{\zeta_t, \zeta'_t}\left[r(\zeta_t) - r(\zeta'_t)\right] = \mathbb{E}_{\zeta_t, \zeta'_t}\left[s(\zeta_t, \zeta'_t) \cdot |r(\zeta_t) - r(\zeta'_t)|\right]$, where $s(\zeta_t, \zeta'_t) = \text{sgn}(r(\zeta_t) - r(\zeta'_t))$. Especially in later stages of training, improvements might be small and it can be hard to approximate the expected reward difference $|r(\zeta_t) - r(\zeta'_t)|$. Thus, we switch to a binary reward as in self-competitive methods and arrive at

$$V^{\pi,\mu}_{\text{sgn}}(s_t, s'_t) := \mathbb{E}_{\substack{\zeta_t \sim \eta^\pi(\cdot|s_t) \\ \zeta'_t \sim \eta^\mu(\cdot|s'_t)}}\left[s(\zeta_t, \zeta'_t)\right], \tag{5}$$

the final target output of the value network in the self-competitive formulation.

4 GAZ PLAY-TO-PLAN

4.1 GAME MECHANICS

Given the original MDP, the above discussion generalizes to a *two-player zero-sum perfect information game* with separated states: two players start with identical copies s_0^1 and s_0^{-1}, respectively, of an initial state $s_0 \sim \rho_0$. The superscripts 1 and -1 indicate the first (max-) and second (min-) player, respectively. Player 1 observes both states (s_0^1, s_0^{-1}), chooses an action a_0^1 and transitions to the next state $s_1^1 = a_0^1 s_0^1$. Afterwards, player -1 observes (s_1^1, s_0^{-1}), chooses action a_0^{-1}, and transitions to state $s_1^{-1} = a_0^{-1} s_0^{-1}$. Then, player 1 observes (s_1^1, s_1^{-1}), chooses action a_1^1, and in this manner both players take turns until they arrive at terminal states s_T^1 and s_T^{-1}. Given the resulting trajectory ζ_0^p for player $p \in \{1, -1\}$, the outcome z of the game is set to $z = 1$ if $r(\zeta_0^1) \geq r(\zeta_0^{-1})$ (player 1 wins) and $z = -1$ otherwise (player 1 loses). In case of equality, player 1 wins to discourage player -1 from simply copying moves.

4.2 ALGORITHM

Algorithm 1 and Figure 1 summarize our approach for improving the agent's performance. In each episode, a 'learning actor' improves its policy by playing against a 'greedy actor', which is a histor-

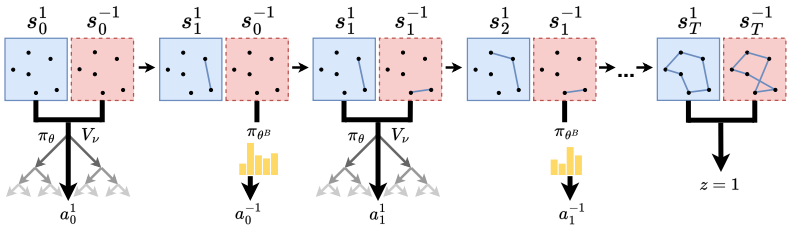

Figure 1: Schematic view of the proposed game in an example instance of the Traveling Salesman Problem: The learning actor is player 1 (solid outline, blue), the greedy actor is player -1 (dashed outline, red). The learning actor chooses moves via MCTS, taking into account the states of both players, whereas the greedy actor moves greedily with respect to only its own state.

Algorithm 1: GAZ Play-to-Plan (GAZ PTP) Training

Input: ρ_0: initial state distribution; $\mathcal{J}_{\text{arena}}$: set of initial states sampled from ρ_0
Input: $0 \leq \gamma < 1$: self-play parameter
Init policy replay buffer $\mathcal{M}_\pi \leftarrow \emptyset$ and value replay buffer $\mathcal{M}_V \leftarrow \emptyset$
Init parameters θ, ν for policy net $\pi_\theta \colon \mathcal{S} \to \Delta\mathcal{A}$ and value net $V_\nu \colon \mathcal{S} \times \mathcal{S} \to [-1, 1]$
Init 'best' parameters $\theta^B \leftarrow \theta$
foreach episode **do**
  Sample initial state $s_0 \sim \rho_0$ and set $s_0^p \leftarrow s_0$ for $p = 1, -1$
  Assign learning actor to player position: $l \leftarrow \text{random}(\{1, -1\})$
  Set greedy actor's policy $\mu \leftarrow \begin{cases} \pi_\theta & \text{with probability } \gamma, \\ \pi_{\theta^B} & \text{with probability } 1-\gamma \end{cases}$
  for $t = 0, \ldots, T-1$ **do**
    for player $p = 1, -1$ **do**
      if player $p \neq l$ **then**
        Take greedy action a_t^p according to policy $\mu(s_t^p)$ and receive new state s_{t+1}^p
      else
        Perform policy improvement $\mathcal{I}$ with MCTS
          using $V_\nu(\cdot, \cdot)$ and policy $\pi_\theta(\cdot)$ for player p
          where in tree, player $-p$ samples actions from μ
        Receive improved policy $\mathcal{I}\pi(s_t^p)$, action a_t^p and new state s_{t+1}^p
        Store $(s_t^p, \mathcal{I}\pi(s_t^p))$ in $\mathcal{M}_\pi$
  Have trajectories $\zeta^p \leftarrow (s_0^p, a_0^p, \ldots, s_{T-1}^p, a_{T-1}^p, s_T^p)$ for players $p \in \{1, -1\}$
  $z \leftarrow \begin{cases} 1 & \text{if } r(\zeta^1) \geq r(\zeta^{-1}), \\ -1 & \text{else} \end{cases}$ ▷ game outcome from perspective of player 1
  Store tuples (s_t^1, s_t^{-1}, z) and $(s_t^{-1}, s_{t+1}^1, -z)$ in $\mathcal{M}_V$ for all timesteps t
  Periodically update $\theta^B \leftarrow \theta$ if $\sum_{s_0 \in \mathcal{J}_{\text{arena}}} \left(r(\zeta_{0,\pi_\theta}^{\text{greedy}}) - r(\zeta_{0,\pi_{\theta^B}}^{\text{greedy}}) \right) > 0$

ical greedy version of itself. We elaborate on the main parts in the following (more details can be found in Appendix B).

Network and training An agent maintains a state-dependent policy network $\pi_\theta \colon \mathcal{S} \to \Delta\mathcal{A}$ and value network $V_\nu \colon \mathcal{S} \times \mathcal{S} \to [-1, 1]$ parameterized by θ and ν, respectively, where $\Delta\mathcal{A}$ is the probability simplex over actions and V_ν serves as a function approximator of (5). The agent generates experience in each episode for training the networks by playing the game in Section 4.1 against a frozen historically best policy version π_{θ^B} of itself. Initially, θ and θ^B are equal. The networks are trained from experience replay as in AlphaZero, where we store value targets from the perspective of both players and only policy targets coming from the learning actor.

Choice of players The learning actor and greedy actor are randomly assigned to player positions at the beginning of each episode. The greedy actor simply chooses actions greedily according to its policy π_{θ^B}. In contrast, the learning actor chooses actions using the search and action selection mechanism of GAZ to improve its policy and dominate over π_{θ^B}. Using MCTS is a huge advantage for the learning actor. If the player positions would not be reassigned between episodes, the value network might just learn to extract and output the player position of the learning actor by counting the remaining moves. In addition, we found the policy to become stronger when the learning actor makes its moves sometimes after and sometimes before the greedy actor. At test time, we fix the learning actor to the position of player 1.

GAZ tree search for learning actor The learning actor uses MCTS to improve its policy and dominate over the greedy behavior of π_{θ^B}. We use the AlphaZero variant (as opposed to MuZero) in Danihelka et al. (2022) for the search, i.e., we do not learn models for the reward and transition dynamics. The search tree is built through several simulations, as usual, each consisting of the phases *selection*, *expansion*, and *backpropagation*. Each node in the tree is of the form $\mathcal{N} = (s^1, s^{-1}, k)$, consisting of both players' states, where $k \in \{1, -1\}$ indicates which player's turn it is. Each edge $(\mathcal{N}, a)$ is a node paired with a certain action a. We apply the tree search of GAZ similarly to a two-player board game, with two major modifications:

(i) Choice of actions: For the sake of explanation, let us assume that $l = 1$, i.e., the learning actor takes the position of player 1. In the *selection* phase, an action is chosen at any node as follows: If the learning actor is to move, we regularly choose an action according to GAZ's tree search rules which are based on the completed Q-values. However, if it is the greedy actor's turn, we *sample* an action from π_{θ^B}. This is repeated until a leaf edge $(\mathcal{N}, a)$ with $\mathcal{N} = (s^1, s^{-1}, k)$ is reached. By sampling a move from the greedy actor's policy π_{θ^B}, the learning actor is forced to plan against not only one (possibly suboptimal) greedy trajectory but also other potential trajectories. This encourages exploration and yields a better estimate of the true value $V_{\text{sgn}}^{\pi_\theta, \pi_{\theta^B}}$ as defined in (5).

(ii) Backpropagation: When a leaf edge $(\mathcal{N}, a)$ as above is *expanded* and $k = -l = -1$, the learning actor is to move in the new node $\tilde{\mathcal{N}} = (s^1, as^{-1}, 1)$. We proceed by querying the policy and value network to obtain $\pi_\theta(s^1)$ and $v := V_\nu(s^1, as^{-1})$. The new node $\tilde{\mathcal{N}}$ is added to the tree and the value approximation v is *backpropagated* up the search path. If $k = 1$, i.e. the turn of the greedy actor in $\tilde{\mathcal{N}} = (as^1, s^{-1}, -1)$, we only query the policy network $\pi_{\theta^B}(s^{-1})$, sample an action $\tilde{a}$ from it, and *directly* expand the edge $(\tilde{N}, \tilde{a})$. In particular, we do not backpropagate a value approximation of state pairs where the greedy actor is to move. Nodes, where it's the greedy actor's turn, are similar to *afterstates* (Sutton & Barto, 2018), which represent chance transitions in stochastic environments. We further illustrate the procedure in Appendix B.2.

Arena mode To ensure we always play against the strongest greedy actor, we fix at the beginning of the training a large enough 'arena' set of initial states $\mathcal{J}_{\text{arena}} \sim \rho_0$ on which periodically π_θ and π_{θ^B} are pitted against each other. For each initial state $s_0 \in \mathcal{J}_{\text{arena}}$, the policies π_θ and π_{θ^B} are unrolled greedily to obtain the trajectories $\zeta^{\text{greedy}}_{0,\pi_\theta}$ and $\zeta^{\text{greedy}}_{0,\pi_{\theta^B}}$, respectively. We replace θ^B with θ if $\sum_{s_0 \in \mathcal{J}_{\text{arena}}} \left(r(\zeta^{\text{greedy}}_{0,\pi_\theta}) - r(\zeta^{\text{greedy}}_{0,\pi_{\theta^B}}) \right) > 0$.

Self-play parameter The parameters θ, θ^B might be initialized disadvantageously (worse than uniformly random play), so that the learning actor generates no useful experience while maintaining a greedy policy worse than π_{θ^B}. We especially experienced this behavior when using a small number of simulations for the tree search. To mitigate this and to stabilize training, we switch to *self-play* in an episode with a small nonzero probability γ, i.e., the greedy actor uses the non-stationary policy π_θ instead of π_{θ^B}.

5 EXPERIMENTS

We evaluate our algorithm on the Traveling Salesman Problem (TSP) and the standard Job-Shop Scheduling Problem (JSSP). For this purpose, we train the agent on randomly generated instances to learn heuristics and generalize beyond presented instances.

Experimental goal There is a plethora of deep reinforcement learning approaches for training heuristic solvers for COPs, including Bello et al. (2016); Deudon et al. (2018); Kool et al. (2018); Xing et al. (2020); da Costa et al. (2020); Kool et al. (2022) for the TSP and Zhang et al. (2020); Park et al. (2021a;b); Tassel et al. (2021) for the JSSP. Most of them are problem-specific and especially concentrate on how to optimize (graph) neural network-based architectures, state representations, and reward shaping. Additionally, learned solvers often focus on short evaluation times while accepting a high number of training instances. Our main goal is to show the performance of GAZ PTP compared with applying GAZ in a single-player way. Nevertheless, we also present results of recent learned solvers to put our approach into perspective. An in-depth comparison is difficult for several reasons. (i) Using MCTS is a strong advantage. However, its runtime is highly sensitive to its implementation, and running MCTS is *always* slower than a single (possibly batch-wise) rollout of a policy network. For this purpose, we do not report inference times. (ii) The network architecture is kept simple to be adjustable for both problem classes. (iii) We give only episodic rewards, even if there would be (canonical) intermediate rewards. Furthermore, we do not include heuristics in state representations. (iv) Eventually, the agent starts from zero knowledge and runs for a comparably lower number of episodes: 100k for TSP (for comparison, Kool et al. (2018) trains on 128M trajectories), 20k for JSSP (40k trajectories in Zhang et al. (2020)).

5.1 Single-player variants

We compare our approach GAZ PTP with three single-player GAZ variants:

GAZ Single Vanilla We apply GAZ for single-agent domains to the original MDP. For the value target, we predict the final outcome of the episode, i.e., we train the value network $V_\nu \colon \mathcal{S} \to \mathbb{R}$ on tuples of the form $(s_t, r(\zeta_0))$ for a trajectory $\zeta_0 = (s_0, a_0, \ldots, a_{T-1}, s_T)$. We transform the Q-values with a min-max normalization based on the values found inside the tree search to cope with different reward scales in MCTS (Schrittwieser et al., 2020; Danihelka et al., 2022). For details, see Appendix C.1.3.

GAZ Single N-Step Same as 'GAZ Single Vanilla', except: we bootstrap N steps into the future for the value target, as used in MuZero (Schrittwieser et al., 2020) and its variants (Hubert et al., 2021; Antonoglou et al., 2022). As we do not assume intermediate rewards, this is equivalent to predicting the undiscounted root value of the search tree N steps into the future. We set $N = 20$ in all instances.

GAZ Greedy Scalar This is a self-competitive version with a scalar baseline, which can differentiate between instances as in a self-critical setting. We keep frozen parameters of a best-performing greedy policy. At the start of each episode, a full greedy rollout ζ_0^{greedy} from the initial state s_0 is performed with the frozen policy to obtain an instance-specific baseline $R := r(\zeta_0^{\text{greedy}}) \in \mathbb{R}$. At test time, $\max\{R, \text{learning actor's outcome}\}$ is taken as the result in the original MDP. A listing of the algorithm is provided in Appendix C.1.4.

5.2 General setup

In this section, we outline the common experimental setup for both TSP and JSSP. Full details and specifics can be found in Appendix C.1.

GAZ MCTS We closely follow the original work of Danihelka et al. (2022) for the choice of the function σ in (3) and the completion of Q-values. The number of simulations allowed from the search tree's root node is critical since a higher simulation budget generally yields more accurate value estimates. Although two edges are expanded in a single simulation in the modified tree search of GAZ PTP, the learning actor considers only one additional action in the expansion step because the greedy actor's moves are considered as environment transitions. Nevertheless, we grant the single-player variants twice as many simulations at the root node as GAZ PTP in all experiments.

Network architecture The policy and value network share a common encoding part $f \colon \mathcal{S} \to \mathbb{R}^d$ mapping states to some latent space $\mathbb{R}^d$. A policy head $g \colon \mathbb{R}^d \to \Delta\mathcal{A}$ and value head $h \colon \mathbb{R}^d \times \mathbb{R}^d \to$

Table 1: Results for TSP and JSSP. 'Num Sims' refers to the number of simulations starting from the root of the search tree.

	Method	Num Sims	Obj.	Gap	Obj.	Gap	Obj.	Gap
			n = 20		n = 50		n = 100	
TSP	Optimal (Concorde)	—	3.84	0.00%	5.70	0.00%	7.76	0.00%
TSP	Kool et al. (2018) (gr.)	—	3.85	0.34%	5.80	1.76%	8.12	4.53%
TSP	GAZ Single Vanilla	200	3.86	0.58%	6.06	6.36%	9.87	27.15%
TSP	GAZ Single N-Step	200	3.86	0.58%	5.96	4.72%	9.89	27.47%
TSP	GAZ Greedy Scalar	200	3.87	0.83%	6.02	5.74%	11.21	44.43%
TSP	GAZ PTP	100	3.84	0.19%	5.81	1.90%	8.16	5.11%
TSP	GAZ PTP	greedy[1]	3.86	0.74%	5.90	3.63%	8.35	7.60%
			15×15		20×20		30×20	
JSSP	Upper Bound	—	1228.9	0.0%	1617.3	0.0%	1921.3	0.0%
JSSP	Zhang et al. (2020)	—	1547.4	26.0%	2128.1	31.6%	2603.9	33.6%
JSSP	GAZ Single Vanilla	100	1585.8	29.0%	2062.4	27.5%	2579.1	34.4%
JSSP	GAZ Single N-Step	100	4090.8	232.8%	5749.1	255.6%	7500.9	290.4%
JSSP	GAZ Greedy Scalar	100	1447.9	17.8%	4721.5	191.9%	6184.3	222.1%
JSSP	GAZ PTP	50	1432.0	16.6%	1973.2	22.0%	2539.1	32.3%
JSSP	GAZ PTP	greedy[1]	1505.7	22.6%	1993.0	23.2%	2601.8	35.6%

$[-1, 1]$ are stacked on top of f such that $\pi_\theta(s) = g(f(s))$ and $V_\nu(s, s') = h(f(s), f(s'))$. For both problems, f is based on the Transformer architecture (Vaswani et al., 2017) and the policy head g uses a pointing mechanism as in (Bello et al., 2016) and (Kool et al., 2018). For GAZ PTP and all single-player variants, the architecture of f is identical, and g, h similar (see Appendix C.1.5).

5.3 Results

Results for TSP and JSSP are summarized in Table 1. In all experiments (except for the robustness experiments below), the random number generators were seeded with 42.

Traveling Salesman Problem The TSP is a fundamental routing problem that, given a graph, asks for a node permutation (a complete *tour*) with minimal edge weight. We focus on the two-dimensional Euclidean case in the unit square $[0, 1]^2 \subseteq \mathbb{R}^2$, and train on small- to medium-sized instances with $n = 20$, 50 and 100 nodes. Tours are constructed sequentially by choosing one node to visit after the other. We run the agent for 100k episodes (see Appendix C.2 for further details). We report the average performance of all GAZ variants on the 10,000 test instances of Kool et al. (2018). We compare with the greedy results of Kool et al. (2018), where a similar attention model is trained autoregressively using REINFORCE with a greedy rollout baseline. Optimality gaps are calculated using optimal solutions found by the Concorde TSP solver (Applegate et al., 2006). We consistently outperform all single-player variants, coming close to the greedy results of Kool et al. (2018), and obtain strong results even when simply unrolling the learned policy greedily (see 'Num Sims = greedy' in Table 1). While all methods perform well on $n = 20$, the single-player variants strongly underperform on $n = 50$ and $n = 100$. GAZ Greedy Scalar fails to improve very early in the training process for $n = 100$.

Job-Shop Scheduling Problem The JSSP is an optimization problem in operations research, where we are given k jobs $j_1, \ldots, j_k$ consisting of individual operations, which need to be scheduled on m machines. We focus on the standard case, where each job j_i is a permutation $(o_{i,l})_{l=1}^m$ of the m machines. Here, $o_{i,l} \in \{1, \ldots, m\}$ indicates on which machine the l-th operation has to run, taking some processing time $p_{i,l} \in (0, 1]$. The operations of a job must run in order and a machine can only process one operation at a time. The objective is to find a *schedule* with a minimum *makespan*, i.e.,

[1]We greedily unroll the policy which was trained with 100 (resp. 50 for JSSP) simulations.

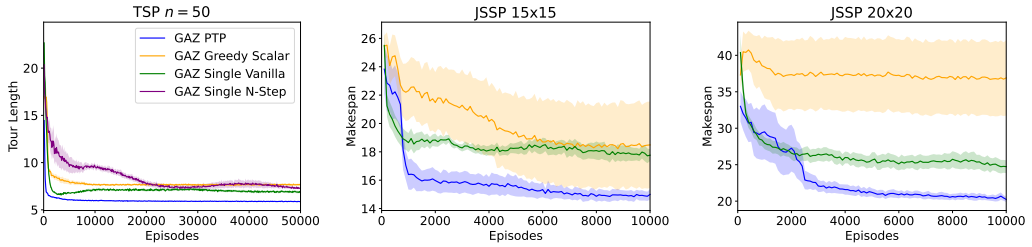

Figure 2: Mean training performance on TSP and JSSP, run with 4 distinct seeds $\in \{42, 43, 44, 45\}$. Shades denote standard errors. GAZ Single N-Step fails to learn on JSSP 15×15 and 20×20, so we omit the results for readability.

the time when all jobs are finished. The size of a problem instance is denoted by $k \times m$ (jobs $\times$ machines). To construct a schedule, we iteratively choose unfinished jobs of which to process the next operation. We run the agent for 20k episodes (see Appendix C.3). We report results on instances of medium size 15×15, 20×20, and 30×20 of the well-known Taillard benchmark set (Taillard, 1993), consisting of 10 instances for each problem size. Optimality gaps are calculated with respect to the best upper bounds found in the literature (see Appendix C.3.3). We compare with the L2D method of Zhang et al. (2020), which takes advantage of the disjunctive graph representation of JSSP and trains a Graph Isomorphism Network (Xu et al., 2018) using PPO (Schulman et al., 2017). As for TSP, our method outperforms all single-player variants of GAZ and even closes the optimality gap of L2D by 8.4% on instances of size 20×20, when unrolling the learned policy greedily. GAZ Greedy Scalar yields strong results only for 15×15 but fails to learn on larger instances, similar to GAZ Single N-Step.

Reproducibility Since GAZ Greedy Scalar fails to learn on multiple instances in Table 1, we further evaluate the robustness and plausibility on four different seeds for TSP $n = 50$ and JSSP 15×15 and 20×20 (Figure 2). To lower the computational cost, we reduce the number of episodes on TSP to 50k and JSSP to 10k, with a simulation budget of 100 for TSP (70 for JSSP) for single-player variants, and 50 for TSP (35 for JSSP) for GAZ PTP. The different seeding leads to only small variations in performance for TSP. JSSP is more challenging. Again, we can observe that GAZ PTP outperforms the single-player variants across all seeds. Especially, GAZ PTP escapes bad initializations early in the training process and becomes stable, as opposed to GAZ Greedy Scalar.

Limitations By planning against a historical policy π_{θ^B} in our proposed method, the problem's search space increases exponentially. We also need an additional network evaluation for $\pi_{\theta^B}(\cdot)$ in each MCTS simulation. Nevertheless, our empirical results show that our method performs well in general with a small number of simulations. Furthermore, even though GAZ's policy improvements propagate from the two-player setup to the original single-player problem by Lemma 1, an improved policy does not necessarily imply improved greedy behavior. Especially in later stages of training, when the performance gap between learning and greedy actor has shrunk significantly, it can take thousands of steps until the distribution of π_θ is sharp enough for updating the parameters θ^B.

6 Conclusion

We introduced GAZ PTP, a self-competitive method combining greedy rollouts as in self-critical training with the planning dynamics of two-player games. We demonstrated that the self-critical transformation of a deterministic single-player task does not alter the theoretical policy improvement guarantees obtained through the principled search of GAZ. Experiments on the TSP and JSSP confirm that our method learns strong policies in the original task with a low number of search simulations. We believe our algorithm is a powerful contribution to tackling the challenges of single-player planning problems.

7 REPRODUCIBILITY STATEMENT

We made versions in PyTorch (Paszke et al., 2017) of all our implementations, and will make them available in an anonymous repository shortly. Further, we will release the network weights for experiments in Section 5. We also reported random number generator seeds. We provide information about network architectures and training details in the appendix.

REFERENCES

Ioannis Antonoglou, Julian Schrittwieser, Sherjil Ozair, Thomas K. Hubert, and David Silver. Planning in stochastic environments with a learned model. *International Conference on Learning Representations*, 2022.

David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006.

Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *arXiv:1710.03748*, 2017.

Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *International Conference on Learning Representations*, 2016.

Dmitrii Beloborodov, Alexander E. Ulanov, Jakob N. Foerster, Shimon Whiteson, and A.I. Lvovsky. Reinforcement learning enhanced quantum-inspired algorithm for combinatorial optimization. *Machine Learning: Science and Technology*, 2:025009, 2020.

Paulo da Costa, Jason Rhuggenaath, Yingqian Zhang, and Alp Akcay. Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning. *Asian Conference on Machine Learning*, 2020.

Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. Policy improvement by planning with gumbel. *International Conference on Learning Representations*, 2022.

Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning heuristics for the tsp by policy gradient. *International conference on the integration of constraint programming, artificial intelligence, and operations research*, 2018.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pp. 4171–4186. Association for Computational Linguistics, 2019.

Quirin Göttl, Dominik G. Grimm, and Jakob Burger. Automated synthesis of steady-state continuous processes using reinforcement learning. *Frontiers of Chemical Science and Engineering*, 16:288–302, 2022.

Jean-Bastien Grill, Florent Altché, Yunhao Tang, Thomas Hubert, Michal Valko, Ioannis Antonoglou, and Rémi Munos. Monte-carlo tree search as regularized policy optimization. *International Conference on Machine Learning*, 2020.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv:1606.08415*, 2016.

Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Mohammadamin Barekatain, Simon Schmitt, and David Silver. Learning and planning in complex action spaces. *International Conference on Machine Learning*, 2021.

Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. *International Conference on Machine Learning*, 2013.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *International Conference on Learning Representations*, 2018.

Wouter Kool, Herke van Hoof, and Max Welling. Buy 4 reinforce samples, get a baseline for free! *International Conference on Learning Representations*, 2019a.

Wouter Kool, Herke van Hoof, and Max Welling. Stochastic beams and where to find them: The gumbel-top-k trick for sampling sequences without replacement. *International Conference on Machine Learning*, 2019b.

Wouter Kool, Herke van Hoof, Joaquim Gromicho, and Max Welling. Deep policy dynamic programming for vehicle routing problems. *arXiv:2102.11756*, 2022.

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*, 2020.

Alexandre Laterre, Yunguan Fu, Mohamed K. Jabri, Alain-Sam Cohen, David Kas, Karl Hajjar, Torbjørn S Dahl, Amine Kerkeni, and Karim Beguir. Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization. *Advances in Neural Information Processing Systems 31*, 2018.

Kevin Leyton-Brown and Yoav Shoham. *Essentials of game theory: A concise multidisciplinary introduction (Synthesis lectures on artificial intelligence and machine learning)*, volume 1. Morgan & Claypool Publishers, 2008.

Amol Mandhane, Anton Zhernov, Maribeth Rauh, Chenjie Gu, Miaosen Wang, Flora Xue, Wendy Shang, Derek Pang, Rene Claus, Ching-Han Chiang, Cheng Chen, Jingning Han, Angie Chen, Daniel J. Mankowitz, Jackson Broshear, Julian Schrittwieser, Thomas Hubert, Oriol Vinyals, and Timothy Mann. Muzero with self-competition for rate control in vp9 video compression. *arXiv:2202.06626*, 2022.

Junyoung Park, Sanjar Bakhtiyar, and Jinkyoo Park. Schedulenet: Learn to solve multi-agent scheduling problems with reinforcement learning. *arXiv:2106.03051*, 2021a.

Junyoung Park, Jaehyeong Chun, Sang Hun Kim, Youngkook Kim, and Jinkyoo Park. Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning. *International Journal of Production Research*, 59:3360–3377, 2021b.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. *Conference on Neural Information Processing System*, 2017.

Tobias Pohlen, Bilal Piot, Todd Hester, Mohammad Gheshlaghi Azar, Dan Horgan, David Budden, Gabriel Barth-Maron, Hado Van Hasselt, John Quan, Mel Večerík, Matteo Hessel, Remi Munos, and Olivier Pietquin. Observe and look further: Achieving consistent performance on atari. *arXiv:1805.11593*, 2018.

Steven J. Rennie, Etienne Marcheret, Youssef Mroueh, Jerret Ross, and Vaibhava Goel. Self-critical sequence training for image captioning. *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

Maarten P.D. Schadd, Mark H.M. Winands, H.J. van den Herik, Guillaume M.J.-B. Chaslot, and Jos W.H.M. Uiterwijk. *International Conference on Computers and Games*.

Dan Schmidt, Nick Moran, Jonathan S. Rosenfeld, Jonathan Rosenthal, and Jonathan Yedidia. Self-play learning without a reward metric. *arXiv:1912.07557*, 2019.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588:604–609, 2020.

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

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. 2018.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*, volume 2. MIT press, 2018.

Eric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64:278–285, 1993.

Pierre Tassel, Martin Gebser, and Konstantin Schekotihin. A reinforcement learning environment for job-shop scheduling. *arXiv:2104.03760*, 2021.

Hado P van Hasselt, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. Learning values across many orders of magnitude. *Advances in Neural Information Processing Systems 29*, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems 30*, 2017.

Tim Vieira. *URL https://timvieira.github.io/blog/post/2014/08/01/gumbel-max-trick-andweighted-reservoir-sampling*.

Hui Wang, Mike Preuss, Michael Emmerich, and Aske Plaat. Tackling morpion solitaire with alphazero-like ranked reward reinforcement learning. *2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2020.

Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. Learning deep transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 1810–1822, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1176.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

Zhihao Xing, Shikui Tu, and Lei Xu. Solve traveling salesman problem by monte carlo tree search and deep neural network. *arXiv:2005.06879*, 2020.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *International Conference on Learning Representations*, 2018.

Ruiyang Xu and Karl Lieberherr. Learning self-play agents for combinatorial optimization problems. *The Knowledge Engineering Review*, 35, 2020.

Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform badly for graph representation? *Advances in Neural Information Processing Systems*, 34:28877–28888, 2021.

Cong Zhang, Wen Song, Zhiguang Cao, Jie Zhang, Puay S. Tan, and Xu Chi. Learning to dispatch for job shop scheduling via deep reinforcement learning. *Advances in Neural Information Processing Systems*, 2020.

Fangwei Zhong, Peng Sun, Wenhan Luo, Tingyun Yan, and Yizhou Wang. Ad-vat+: An asymmetric dueling mechanism for learning and understanding visual active tracking. *IEEE transactions on pattern analysis and machine intelligence*, 43(5):1467–1482, 2019.

A PROOFS

A.1 PROOF OF LEMMA 1

Lemma: Let $\pi, \tilde{\pi}$ and μ be state-dependent policies. For any states $s_t, s'_l \in \mathcal{S}$ and action $a_t \in \mathcal{A}$, we have

$$A^{\pi}(s_t, a_t) = A^{\pi,\mu}(s_t, s'_l; a_t), \text{ and}$$
$$\Big[\sum_{a_t} \tilde{\pi}(a_t|s_t) Q^{\pi}(s_t, a_t)\Big] - V^{\pi}(s_t) = \Big[\sum_{a_t} \tilde{\pi}(a_t|s_t) Q^{\pi,\mu}(s_t, s'_l; a_t)\Big] - V^{\pi,\mu}(s_t, s'_l).$$

Proof: We recall the definitions for the sake of clarity:

$$V^{\pi,\mu}(s_t, s'_l) := \mathbb{E}_{\substack{\zeta_t \sim \eta^{\pi}(\cdot|s_t) \\ \zeta'_l \sim \eta^{\mu}(\cdot|s'_l)}} \left[r(\zeta_t) - r(\zeta'_l)\right] \qquad Q^{\pi,\mu}(s_t, s'_l; a_t) := r(s_t, a_t) + V^{\pi,\mu}(a_t s_t, s'_l)$$
$$A^{\pi,\mu}(s_t, s'_l; a_t) := Q^{\pi,\mu}(s_t, s'_l; a_t) - V^{\pi,\mu}(s_t, s'_l)$$

Note that

$$\begin{aligned} V^{\pi}(s_t) = \mathbb{E}_{\zeta_t \sim \eta^{\pi}(\cdot|s_t)}\left[r(\zeta_t)\right] &= \mathbb{E}_{\zeta'_l \sim \eta^{\mu}(\cdot|s'_l)} \mathbb{E}_{\zeta_t \sim \eta^{\pi}(\cdot|s_t)}\left[r(\zeta_t)\right] \\ &= \mathbb{E}_{\substack{\zeta_t \sim \eta^{\pi}(\cdot|s_t) \\ \zeta'_l \sim \eta^{\mu}(\cdot|s'_l)}}\left[r(\zeta_t)\right], \end{aligned}$$

by the law of iterated expectations and because the realization of a trajectory following π does not depend on μ and vice versa. Hence by linearity of expectations

$$V^{\pi}(s_t) - V^{\mu}(s'_l) = \mathbb{E}_{\substack{\zeta_t \sim \eta^{\pi}(\cdot|s_t) \\ \zeta'_l \sim \eta^{\mu}(\cdot|s'_l)}}\left[r(\zeta_t) - r(\zeta'_l)\right] = V^{\pi,\mu}(s_t, s'_l), \tag{6}$$

and it follows that

$$\begin{aligned} A^{\pi}(s_t, a_t) &= Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \\ &= Q^{\pi}(s_t, a_t) - V^{\mu}(s'_l) - (V^{\pi}(s_t) - V^{\mu}(s'_l)) \\ &= r(s_t, a_t) + V^{\pi}(a_t s_t) - V^{\mu}(s'_l) - (V^{\pi}(s_t) - V^{\mu}(s'_l)) \\ &\overset{(6)}{=} r(s_t, a_t) + V^{\pi,\mu}(a_t s_t, s'_l) - V^{\pi,\mu}(s_t, s'_l) \\ &= Q^{\pi,\mu}(s_t, s'_l; a_t) - V^{\pi,\mu}(s_t, s'_l) \\ &= A^{\pi,\mu}(s_t, s'_l; a_t). \end{aligned}$$

Furthermore, we have

$$\begin{aligned} \Big[\sum_{a_t} \tilde{\pi}(a_t|s_t) Q^{\pi}(s_t, a_t)\Big] - V^{\pi}(s_t) &= \Big[\sum_{a_t} \tilde{\pi}(a_t|s_t)\big(r(s_t, a_t) + V^{\pi}(a_t s_t)\big)\Big] - V^{\pi}(s_t) \\ &= \Big[\sum_{a_t} \tilde{\pi}(a_t|s_t)\big(r(s_t, a_t) + V^{\pi}(a_t s_t) - V^{\mu}(s'_l) + V^{\mu}(s'_l)\big)\Big] - V^{\pi}(s_t) \\ &\overset{(6)}{=} \Big[\sum_{a_t} \tilde{\pi}(a_t|s_t)\big(\underbrace{r(s_t, a_t) + V^{\pi,\mu}(a_t s_t, s'_l)}_{=Q^{\pi,\mu}(s_t, s'_l; a_t)} + V^{\mu}(s'_l)\big)\Big] - V^{\pi}(s_t) \\ &= \Big[\sum_{a_t} \tilde{\pi}(a_t|s_t) Q^{\pi,\mu}(s_t, s'_l; a_t)\Big] + \underbrace{\Big[\sum_{a_t} \tilde{\pi}(a_t|s_t) V^{\mu}(s'_l)\Big]}_{=V^{\mu}(s'_l)} - V^{\pi}(s_t) \\ &= \Big[\sum_{a_t} \tilde{\pi}(a_t|s_t) Q^{\pi,\mu}(s_t, s'_l; a_t)\Big] - (V^{\pi}(s_t) - V^{\mu}(s'_l)) \\ &= \Big[\sum_{a_t} \tilde{\pi}(a_t|s_t) Q^{\pi,\mu}(s_t, s'_l; a_t)\Big] - V^{\pi,\mu}(s_t, s'_l). \end{aligned}$$

□

A.2 Derivation of the Logit Update (3)

In Danihelka et al. (2022), the improved policy π' is obtained by the logit update

$$\text{logit}^{\pi'}(a) := \text{logit}^{\pi}(a) + \sigma(\hat{Q}^{\pi}(s,a)).$$

Subtracting the constant $\sigma(\hat{V}^{\pi}(s))$ does not alter the subsequent softmax-output of the logits, so we can equivalently set

$$\begin{aligned}\text{logit}^{\pi'}(a) := \text{logit}^{\pi}(a) + \sigma(\hat{Q}^{\pi}(s,a)) - \sigma(\hat{V}^{\pi}(s)) &\overset{\sigma \text{ linear}}{=} \text{logit}^{\pi}(a) + \sigma(\hat{Q}^{\pi}(s,a) - \hat{V}^{\pi}(s)) \\ &= \text{logit}^{\pi}(a) + \sigma(\hat{A}^{\pi}(s,a)).\end{aligned}$$

B Additional algorithm details

B.1 A note on self-play in the two-player game

The mechanics of the proposed two-player game differ from classical board games such as chess or Go, as a player's action does not influence the opponent's state. We can interpret trajectories in the original MDP as strategies in the game: if some trajectory α gives a higher final reward than trajectory β, its corresponding strategy α *weakly dominates* strategy β (Leyton-Brown & Shoham, 2008). This is desired and means that playing α will always yield an outcome z at least as good as playing β, no matter what the opponent does. Usually, in AlphaZero-type self-play for board games, both players choose their moves utilizing MCTS. By limiting the tree search to the learning actor, we can distill even incremental policy improvements to the policy network effectively.

B.2 Modified tree search

We provide an illustration of the modified *selection, expansion* and *backpropagation* in Figure 3. As outlined in Section 4.2, we do not evaluate state pairs with V_ν in nodes where it's the greedy actor's turn. This is due to computational efficiency. Suppose the greedy actor is player -1, and it is the greedy actor's turn in a node $\tilde{\mathcal{N}} = (as^1, s^{-1}, -1)$, which was added to the tree in an expansion step. By the modified tree search, an action is sampled from $\pi_{\theta^B}(s^{-1})$. As the policy only depends on the player's state, $\pi_{\theta^B}(s^{-1})$ can be reused for the action sampling in a node $(\tilde{s}^1, s^{-1}, -1)$ for *any* state $\tilde{s}^1$ without additional network evaluations, as the value function does not need to be queried. This can reduce the number of network evaluations in subsequent simulations.

This is different from the experiments in Stochastic MuZero (Antonoglou et al., 2022), where the authors report significantly improved results for AlphaZero on Backgammon and 2048 when a Q-value is learned for afterstates (chance nodes). For GAZ PTP, we did not experience notable improvements and trade the learned Q-value for increased computational efficiency.

For evaluation at test time, the greedy actor's moves are not sampled in the tree search but chosen greedily to increase the search tree's depth.

B.3 A note on the arena mode

As only the sign of the reward difference $r(\zeta_{0,\pi_\theta}) - r(\zeta^{\text{greedy}}_{0,\pi_{\theta^B}})$ is considered to compute the outcome of the game, and not its magnitude, training a non-stationary policy through self-play can exhibit behavior where the overall performance of a policy in the original MDP gradually degrades, even though the agent gets better at beating itself. In the optimal case, we would like to replace the frozen parameters θ^B with θ, if

$$\mathbb{E}_{s_0\sim\rho_0}\mathbb{E}_{\zeta_0\sim\eta^{\pi_\theta}(\cdot|s_0)}[r(\zeta_0)] > \mathbb{E}_{s_0\sim\rho_0}\mathbb{E}_{\zeta_0'\sim\eta^{\pi_{\theta^B}}(\cdot|s_0)}[r(\zeta_0')].$$

This cannot be guaranteed alone by the binary reward in our self-competitive framework. We settle for a *fixed* arena set $\mathcal{J}_{\text{arena}}$ on which the policies are pitted *greedily*, to ensure an improved greedy actor and avoid cycling performance due to stochasticity as far as possible.

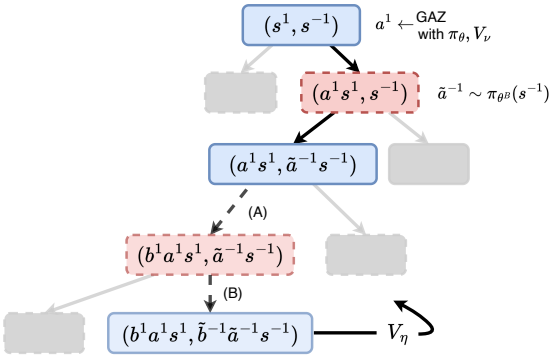

Figure 3: Example MCTS for the learning actor: In nodes with solid lines (blue), the learning actor (player 1) is to move, and in nodes with dashed lines (red), the greedy actor (player -1) is to move. In the selection phase (solid arrows), an action is selected in solid nodes according to the search principles of GAZ, based on π_θ and the completed Q-values. In nodes with dashed lines, an action is sampled from π_{θ^B}. Suppose the dashed edge (A) is expanded in the expansion phase, then it is the greedy actor's turn in the following node: an action is sampled from $\pi_{\theta^B}(\tilde{a}^{-1}s^{-1})$ and the corresponding edge (B) is immediately expanded as well. Only then is the predicted value of the following solid node backpropagated through the search path.

B.4 Replay buffer

Given trajectory $\zeta_0^p = (s_0^p, a_0^p, \ldots, s_{T-1}^p, a_{T-1}^p, s_T^p)$ for player $p \in \{1, -1\}$, training data is stored in replay buffers at the end of an episode as follows: For the value network, the final game outcome z is bootstrapped from the perspective of both players, i.e. we store tuples (s_t^1, s_t^{-1}, z) and $(s_t^{-1}, s_{t+1}^1, -z)$. For the policy network, we store tuples $(s_t^l, \mathcal{I}\pi(s_t^l))$ only for the learning actor $l \in \{1, -1\}$, where $\mathcal{I}\pi(s_t^l)$ is the improved policy obtained through the GAZ tree search at state s_t^l.

B.5 Loss functions

We train the value network by minimizing the squared error $(V_\nu(s, s') - \tilde{z})^2$ for a sampled tuple $(s, s', \tilde{z})$ in the replay buffer, and the policy network by minimizing the Kullback-Leibler divergence $\mathrm{KL}(\mathcal{I}\pi(s) \parallel \pi_\theta(s))$ for a tuple $(s, \mathcal{I}\pi(s))$.

C Experimental details

C.1 General setup

C.1.1 Training loop

All algorithmic variants fit into the asynchronous training loop commonly used for MuZero, where a *learning process* receives generated trajectories, stores them in a replay buffer, and performs training steps for the network. Multiple *playing processes* generate trajectories from initial states randomly sampled on the fly, using periodically updated network checkpoints from the learning process for MCTS. We generate experience with 100 playing processes and update the network checkpoint every 100 training steps in all variants.

C.1.2 GAZ MCTS

Given a node $\mathcal{N}$ in the search tree, denote by $N(a)$ the visit count of the edge $(\mathcal{N}, a)$ for action a. We follow Danihelka et al. (2022) for the choice of the monotonically increasing linear function σ in (3) and set

$$\sigma(q) = (c_{\text{visit}} + \max_b N(b)) \cdot c_{\text{scale}} \cdot q.$$

Algorithm 2: GAZ Greedy Scalar Training

Input: ρ_0: initial state distribution
Input: $\mathcal{J}_{\text{arena}}$: set of initial states sampled from ρ_0
Init policy replay buffer $\mathcal{M}_\pi \leftarrow \emptyset$ and value replay buffer $\mathcal{M}_V \leftarrow \emptyset$
Init parameters θ, ν for policy net $\pi_\theta \colon \mathcal{S} \to \Delta\mathcal{A}$ and value net $V_\nu \colon \mathcal{S} \times \mathbb{R} \to [-1, 1]$
Init 'best' parameters $\theta^B \leftarrow \theta$
foreach episode **do**
  Sample initial state $s_0 \sim \rho_0$
  Perform greedy rollout using π_{θ^B} and obtain $R \leftarrow r(\zeta^{\text{greedy}}_{0,\pi_{\theta^B}})$
  for $t = 0, \ldots, T-1$ **do**
    Perform policy improvement $\mathcal{I}$ with MCTS using $V_\nu(\cdot, R)$ and policy $\pi_\theta(\cdot)$
    Receive improved policy $\mathcal{I}\pi(s_t)$, action a_t and new state s_{t+1}
    Store $(s_t, \mathcal{I}\pi(s_t))$ in $\mathcal{M}_\pi$
  Have trajectory $\zeta \leftarrow (s_0, a_0, \ldots, s_{T-1}, a_{T-1}, s_T)$
  $z \leftarrow \begin{cases} 1 & \text{if } r(\zeta) \geq R, \\ -1 & \text{else} \end{cases}$ ▷ outcome as reshaped binary reward
  Store tuples (s_t, R, z) in $\mathcal{M}_V$ for all timesteps t
  Periodically update $\theta^B \leftarrow \theta$ if $\sum_{s_0 \in \mathcal{J}_{\text{arena}}} \left(r(\zeta^{\text{greedy}}_{0,\pi_\theta}) - r(\zeta^{\text{greedy}}_{0,\pi_{\theta^B}}) \right) > 0$

We set the constants to $c_{\text{visit}} = 50$ and $c_{\text{scale}} = 1.0$, which has shown to be stable across various simulation budgets (Danihelka et al., 2022). We complete the vector of Q-values using the authors' value interpolation proposed in their appendix: For every unvisited action, the Q-value approximation in (3) is set to $\hat{V}$, where

$$\hat{V} := \frac{1}{1 + \sum_b N(b)} \left(V + \frac{\sum_b N(b)}{\sum_{b,\text{ s.t. } N(b)>0} \pi(b)} \sum_{a,\text{ s.t. } N(a)>0} \pi(a) Q(a) \right). \tag{7}$$

Here, V is the value approximation of node $\mathcal{N}$ coming from the value network. We abuse the notation by omitting states (resp. state pairs for GAZ PTP), as (7) is used in all GAZ variants. In particular, unvisited actions are given zero advantage in (3).

C.1.3 GAZ NORMALIZATION

Rewards in single-player tasks usually have different scales and are not limited to ± 1. We follow Schrittwieser et al. (2020); Danihelka et al. (2022), and perform a min-max normalization in GAZ Single Vanilla and GAZ Single N-Step (prior to rescaling by σ) on the vector of completed Q-values with the values observed in the tree up to that point. I.e., the maximum value is given by $\max_{s \in \text{tree}} \hat{Q}(s, a)$ (and analogously for the minimum value). This pushes normalized advantages into the interval $[-1, 1]$.

C.1.4 GAZ GREEDY SCALAR

We provide a listing of the training algorithm for GAZ Greedy Scalar in Algorithm 2.

C.1.5 GENERAL NETWORK ARCHITECTURE

Feed-forward In the following, a feed-forward network (FF) always refers to a multilayer perceptron (MLP) with equal input and output dimensions and one hidden layer of four times the input dimension, with GELU activation (Hendrycks & Gimpel, 2016).

State encoding We model a state-encoding network $f \colon \mathcal{S} \to \mathbb{R}^d$, followed by a policy head $g \colon \mathbb{R}^d \to \Delta\mathcal{A}$ and a state-value head h, which is of the form $h \colon \mathbb{R}^d \times \mathbb{R}^d \to [-1, 1]$ for GAZ

PTP. For both TSP and JSSP, f is based on the Transformer architecture and its underlying multi-head attention (MHA) layers (Vaswani et al., 2017). For some state $s \in \mathcal{S}$, the network f outputs a concatenation of vectors $f(s) = [\tilde{\boldsymbol{s}}; \tilde{\boldsymbol{a}}_1; \dots, \tilde{\boldsymbol{a}}_m]$ where $\tilde{\boldsymbol{s}} \in \mathbb{R}^{\tilde{d}}$ and $\tilde{\boldsymbol{a}}_i \in \mathbb{R}^{\tilde{d}}$ are latent representations of the state s and actions $a_1, \dots, a_m$. The network architecture of f is identical for GAZ PTP and all single-player variants (see C.2.3 and C.3.4).

Value head The value head consists of an MLP with three hidden layers of size $2\tilde{d}$ with GELU activation. The input and output of the MLP differ between variants:

- GAZ PTP: We input a concatenation $[\tilde{\boldsymbol{s}}^1; \tilde{\boldsymbol{s}}^{-1}] \in \mathbb{R}^{2\tilde{d}}$ of latent state vectors for both players. The output is mapped to $[-1, 1]$ via $\tanh$-activation.
- GAZ Greedy Scalar: We input a concatenation $[\tilde{\boldsymbol{s}}^1; R] \in \mathbb{R}^{\tilde{d}+1}$, where $R \in \mathbb{R}$ is the outcome of the greedy rollout. As in GAZ PTP, the output is mapped to $[-1, 1]$ via $\tanh$-activation.
- GAZ Single Vanilla/N-Step: Only the latent state $\tilde{\boldsymbol{s}} \in \mathbb{R}^{\tilde{d}}$ serves as the input, with linear output.

Policy head For the policy head g, the logit for an action $a_i \in \{a_1, \dots, a_m\}$ is computed using a pointing mechanism based on the attention of $\tilde{\boldsymbol{s}}$ and the $\tilde{\boldsymbol{a}}_i$'s similarly to (Bello et al., 2016) and (Kool et al., 2018):

We compute (single-head) attention weights $u_1, \dots, u_m \in [-C, C] \subseteq \mathbb{R}$ via

$$u_i = C \cdot \tanh\left(\frac{(W^Q \tilde{\boldsymbol{s}})^T (W^K \tilde{\boldsymbol{a}}_i)}{\sqrt{\tilde{d}}}\right), \tag{8}$$

with constant $C = 10$ and learnable linear maps $W^Q, W^K \colon \mathbb{R}^{\tilde{d}} \to \mathbb{R}^{\tilde{d}}$. The weight u_i is interpreted as the logit for the probability of action a_i and is set to $-\infty$ for infeasible (masked) actions.

In GAZ PTP, the above process is performed with $\boldsymbol{w} = \mathrm{FF}(\boldsymbol{y}) + \boldsymbol{y}$ instead of $\tilde{\boldsymbol{s}}$, where $\boldsymbol{y} = \mathrm{SHA}(\tilde{\boldsymbol{s}}; \tilde{\boldsymbol{a}}_1, \dots, \tilde{\boldsymbol{a}}_m))$ is a transformation of the vector $\tilde{\boldsymbol{s}}$ through a layer of single-head attention (SHA) with $\tilde{\boldsymbol{s}}$ as the only query, and keys $\tilde{\boldsymbol{a}}_1, \dots, \tilde{\boldsymbol{a}}_m$. This is to simplify the task of aligning the state encoding f for the value and policy head, as in GAZ PTP the value head operates on two separate state encodings. The same design choice did not make any difference in GAZ Greedy Scalar/Single Vanilla/N-Step, so we removed it to speed up computation.

C.1.6 Hyperparameters

Arena episodes are played every 400 episodes in GAZ PTP and GAZ Greedy Scalar. In all experiments, the replay buffer holds data of the latest 2000 episodes. We set the self-play parameter to $\gamma = 0.2$. We use Adam (Kingma & Ba, 2014) as an optimizer, with a constant learning rate of 10^{-4}, sampling batches of size 256 at each training step. Gradients are clipped to unit L_2-norm.

C.2 TSP

C.2.1 Environment

An initial state s_0 is given by n nodes, where $s_0 = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\} \subseteq [0, 1]^2 \subseteq \mathbb{R}^2$. An ordered tour is constructed sequentially by picking one node to visit after the other, iteratively completing a partial tour. In particular, actions are represented by (unvisited) nodes. The agent decides from which node it starts the tour.

We take a relative view at timestep $t > 0$ and represent a state (partial tour) for $t > 0$ by a tuple $s_t = (l_t, \boldsymbol{x}_{t,\text{start}}, \boldsymbol{x}_{t,\text{end}}, X_t = \{\boldsymbol{x}_{t_1}, \dots, \boldsymbol{x}_{t_{n-t}}\})$, where l_t is the length of the current partial tour, $\boldsymbol{x}_{t,\text{start}} = a_{t-1}$ is the last node in the partial tour, $\boldsymbol{x}_{t,\text{end}} = a_0$ is the first chosen node (and must be eventually returned to) and X_t is the remaining set of unvisited nodes from which the next action is picked. The terminal state s_T is a complete solution with $X_T = \emptyset$. The negative length of the full tour is given as a reward at the end of the episode (and zero rewards in between). All tour lengths are scaled by division with $\sqrt{2}n$ (the supremum of a possible tour length with n nodes in the unit square).

C.2.2 DATA GENERATION AND TRAINING

Initial states for training are uniformly sampled on the fly. We fix 300 states for the arena set $\mathcal{J}_{\text{arena}}$, which is identical for GAZ PTP and GAZ Greedy Scalar. During training, the model is evaluated periodically on a fixed validation set of 100 states to determine the final model. The final model is evaluated on the 10,000 instances of Kool et al. (2018), which were generated with the random seed 1234. The agent is run for 100k episodes, keeping a ratio of the number of played episodes to the number of optimizer steps of approximately 1 to $0.1n$. The network architecture is independent of the number of input nodes, but for comparability, we train it from scratch for each $n \in \{20, 50, 100\}$. We sample (at most) 16 actions without replacement at the root of GAZ's search tree. The simulation budget in the tree search is 100 for our approach and 200 for all single-player variants.

We augment training data by applying a random reflection, rotation, and linear scaling within the unit square to states sampled from the replay buffer.

C.2.3 STATE ENCODING NETWORK

The state encoding network f consists of a sequence-to-sequence Transformer architecture similar to the encoder in Kool et al. (2018). We use a latent dimension of $\tilde{d} = 128$ in all TSP experiments. The network is composed of the following components:

- Learnable lookup embeddings $E^{\text{token}}, E^{\text{start}}, E^{\text{end}}, E^{\text{start-ind}}, E^{\text{end-ind}}$ in $\mathbb{R}^{\tilde{d}}$.
- Affine maps $W^{\text{len}}, W^{\text{num}} \colon \mathbb{R} \to \mathbb{R}^{\tilde{d}}$ and $W^{\text{node}} \colon \mathbb{R}^2 \to \mathbb{R}^{\tilde{d}}$.
- A simple stack of five Transformer blocks with eight heads in the self-attention and layer normalization before the MHA and the FF (Wang et al., 2019). The structure of a Transformer block is summarized in Figure 4.

We illustrate the encoding procedure in Figure 4. For an intermediate state $s_t = (l_t, \boldsymbol{x}_{t,\text{start}}, \boldsymbol{x}_{t,\text{end}}, X_t = \{\boldsymbol{x}_{t_1}, \dots, \boldsymbol{x}_{t_{n-t}}\})$, we construct an input sequence

$$
\begin{aligned}
&(E^{\text{token}}, W^{\text{len}}(l_t), W^{\text{num}}(n-t),\\
&W^{\text{node}}(\boldsymbol{x}_{t,\text{start}}) + E^{\text{start-ind}}, W^{\text{node}}(\boldsymbol{x}_{t,\text{end}}) + E^{\text{end-ind}},\\
&W^{\text{node}}(\boldsymbol{x}_{t_1}), \dots, W^{\text{node}}(\boldsymbol{x}_{t_{n-t}})).
\end{aligned}
$$

The sequence element corresponding to E^{token} is a token representing the state, similar to the class token in natural language processing (NLP) (Devlin et al., 2019). The two-dimensional nodes are affinely embedded into $\mathbb{R}^{\tilde{d}}$. As Transformer architectures are invariant to sequence permutations by design, we add the learnable lookup embeddings $E^{\text{start-ind}}, E^{\text{end-ind}}$ to the start and end nodes to indicate them. This is comparable to position embeddings in NLP. For an initial state s_0, we use $W^{\text{len}}(0)$ for the second sequence element. Further, there are no start and end nodes in the initial state yet, so we use the learnable embeddings $E^{\text{start}}, E^{\text{end}}$ instead of the affine embeddings $W^{\text{node}}(\boldsymbol{x}_{t,\text{start}}), W^{\text{node}}(\boldsymbol{x}_{t,\text{end}})$.

The sequence is passed to the stack of Transformer blocks. For each attention head and pair of nodes $\boldsymbol{x}, \boldsymbol{y}$ in the sequence, we add a spatial bias $w_h \cdot \|x - y\|_2 + b_h \in \mathbb{R}$ to the attention weight corresponding to $\boldsymbol{x}, \boldsymbol{y}$, similarly as in Graphormer architectures (Ying et al., 2021).

As outlined in C.1.5, f eventually outputs $f(s_t) = [\tilde{\boldsymbol{s}}; \tilde{\boldsymbol{a}}_1; \dots, \tilde{\boldsymbol{a}}_{n-t}]$, where $\tilde{\boldsymbol{s}}$ is the first output sequence element corresponding to the state token E^{token}, and $\tilde{\boldsymbol{a}}_i$ corresponds to the output sequence element of the i-th remaining node $W^{\text{node}}(\boldsymbol{x}_{t_i})$.

The main structural difference to Kool et al. (2018) is that in our case the latent representation of a state is not computed autoregressively, but is re-done at each state from the partial tour length and two-dimensional coordinates of remaining nodes.

C.3 JSSP

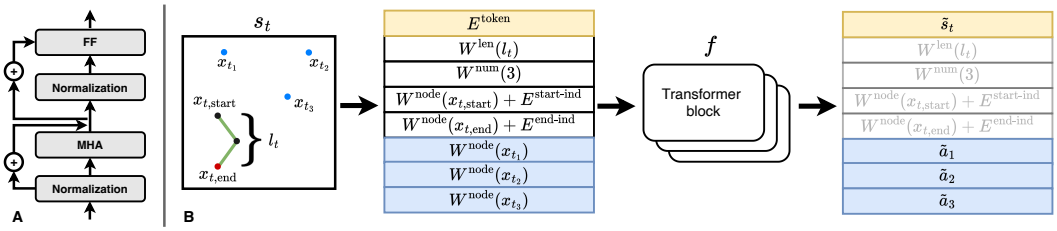

Figure 4: (A) Structure of a Transformer block with pre-normalization. We use layer normalization for TSP. The Transformer block takes in a sequence of elements in $\mathbb{R}^{\tilde{d}}$ and outputs a sequence in $\mathbb{R}^{\tilde{d}}$ of the same length. (B) Schematic view of the relative state encoding for TSP experiments: The agent initially chose to start the tour with the node at the bottom (red). In an intermediate state s_t, the last node in the partial tour is the current start node $x_{t,\text{start}}$. The node $x_{t,\text{end}}$ corresponds to the initial node in the tour, as the agent eventually must return to it to complete the tour. A sequence in $\mathbb{R}^{\tilde{d}}$ is constructed from the state and passed through a stack of Transformer blocks. The output sequence elements corresponding to the state token E^{token} and the unvisited nodes constitute the output of the network f.

C.3.1 Problem formulation

We recall the formulation of the standard JSSP. We are given a set of k jobs $J = \{j_1, \dots, j_k\}$, each consisting of m operations which need to be scheduled on m machines. Each job j_i is a *permutation* $(o_{i,l})_{l=1}^m$ of the machines, where $o_{i,l} \in \{1, \dots, m\}$ indicates on which machine the l-th operation of job j_i needs to run. Finishing an operation takes some processing time $p_{i,l} \in (0, 1]$. The operations of a job must run in order (precedence constraints), a machine can only process one operation at a time, and there is no preemption. The objective is to find a schedule with minimum makespan.

C.3.2 Environment

A schedule must satisfy all precedence constraints, and there is a bijection between operations of a job and the m machines. Thus, we can represent a schedule by a (not necessarily unique) sequence of jobs $(\tilde{j}_1, \dots, \tilde{j}_{k \cdot m})$, where $\tilde{j}_i \in J$ is an unfinished job of which the next unscheduled operation should be scheduled at the earliest time possible. In particular, we can represent feasible actions in the environment by the set of unfinished jobs, limiting the number of possible actions to k. Note that a timestep t in the environment corresponds to the t-th chosen action (unfinished job), and is not equal to the passed processing time in the schedule.

A state s_t is given by a tuple

$$((c_{t,l})_{l=1}^m, (e_{t,i})_{i=1}^k, J_t),$$

where

- $c_{t,l} \in \mathbb{R}_{\geq 0}$ is the finishing time of the latest scheduled operation on the l-th machine ('machine availability'),
- $e_{t,i} \in \mathbb{R}_{\geq 0}$ is the finishing time of the last scheduled operation of a job j_i ('job availability'), and
- $J_t = \{j_{t,1}, \dots, j_{t,n_t}\} \subseteq J$ is the subset of jobs with unscheduled operations ('unfinished jobs').

In particular, we have $c_{0,l} = 0$, $e_{0,i} = 0$ and $J_0 = J$ at the initial state s_0. Let $\tilde{c}_t := \min_l c_{t,l}$ in the following. The negative makespan $\max_l c_{t,l}$ is given as a reward at the end of the episode (and zero rewards in between). We illustrate the state representation in Figure 5.

C.3.3 Data generation and training

For a given number of jobs k and machines m, we generate training data on the fly by randomly sampling a machine permutation $(o_{i,l})_{l=1}^m \in S_m$ for each job j_i, and a random processing time

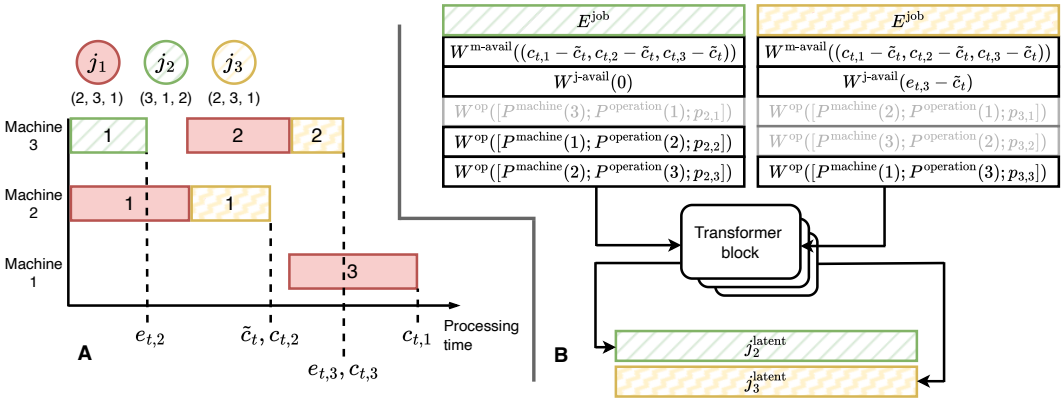

Figure 5: (A) Exemplary Gantt chart of an unfinished schedule, illustrating the relative state representation for JSSP: We assume an instance with three jobs j_1, j_2, j_3 and three machines. The tuple below a job indicates on which machines its operations must run. In the depicted unfinished schedule, all three operations of j_1, the first operation of j_2, and the first two operations of j_3 have been scheduled. The schedule can be represented by the sequence $(j_2, j_1, j_1, j_1, j_3, j_3)$. Dashed lines indicate examples of machine and job availability times. (B) Latent representations for the unfinished jobs j_2, j_3 are obtained by passing a sequence of operations to the first stack of Transformer blocks. Already scheduled operations are masked. The ouput tokens corresponding to E^{token} are used as latent job representations and form part of the sequence on which a second Transformer network operates.

Table 2: Best solutions for Taillard instances from the literature, "*" means the solution is optimal.

Ta01	Ta02	Ta03	Ta04	Ta05	Ta06	Ta07	Ta08	Ta09	Ta10
1231*	1244*	1218*	1175*	1224*	1238*	1227*	1217*	1274*	1241*
Ta21	Ta22	Ta23	Ta24	Ta25	Ta26	Ta27	Ta28	Ta29	Ta30
1642*	1600	1557	1644*	1595	1643	1680	1603*	1625	1584
Ta41	Ta42	Ta43	Ta44	Ta45	Ta46	Ta47	Ta48	Ta49	Ta50
2005	1937	1846	1979	2000	2006	1889	1937	1961	1923

$p_{il} \in (0, 1]$ for each operation. We fix 200 instances for $\mathcal{J}_{\text{arena}}$ and a small validation set of size 20. We evaluate the model on the 10 benchmark instances of Taillard (Taillard, 1993) for each size: instances ta01-ta10 (size 15×15), instances ta21-ta30 (size 20×20) and instances ta41-ta50 (size 30×20). The benchmark instances have integer processing times in $[1, 100]$, which we rescale to the unit interval by division with 100. We summarize the best upper bounds from the literature in Table 2, as reported in Zhang et al. (2020). An episode takes $k \cdot m$ actions, so to reduce computation time, we limit the simulation budget to 50 for our approach and 100 for all single-player variants, running the agent for 20k episodes. We keep a ratio of the number of played episodes to the number of optimizer steps of approximately 1 to $0.5k \cdot m$. We train from scratch for each problem size. As the action space is rather small (at most k actions at each timestep), we consider *all* feasible actions for the simulations at the root of the search tree.

During training, we augment states sampled from the replay buffer by linearly scaling processing times $p_{i,l}$, job availability times $e_{t,i}$ and machine availability times $c_{t,l}$ with a random scalar in $(0, 1)$. Furthermore, we shuffle the machines on which operations must be scheduled (e.g. operations on some machine A are reassigned to some machine B and the other way round).

C.3.4 STATE ENCODING NETWORK

There are two types of sequences in a problem instance. (i) For each job j_i, we have a sequence of operations, where the order of operations matters. (ii) The entirety of jobs forms a sequence, where

the order does *not* matter (similar to the sequence of nodes in the TSP). The encoding network f consists of two stacked Transformer models, where the first one computes a latent representation for each job separately, and the second one operates on the sequence of these job representations to compute a state encoding. Both networks operate in a latent space of dimension $\tilde{d} = 64$.

Job encoding The Transformer network for encoding each job from the sequence of its unfinished operations consists of:

- Learnable one-dimensional embeddings $P^{\text{machine}}, P^{\text{operation}} \colon \{1, \dots, m\} \to \mathbb{R}^{\tilde{d}}$.
- Learnable lookup embedding E^{job} in $\mathbb{R}^{\tilde{d}}$.
- Affine maps $W^{\text{op}} \colon \mathbb{R}^{2\tilde{d}+1} \to \mathbb{R}^{\tilde{d}}$, $W^{\text{m-avail}} \colon \mathbb{R}^m \to \mathbb{R}^{\tilde{d}}$, and $W^{\text{j-avail}} \colon \mathbb{R} \to \mathbb{R}^{\tilde{d}}$.
- A simple stack of three Transformer blocks with four heads in the self-attention and instance normalization before the MHA and the FF.

Let $j_i \in J_t$ be an unfinished job. We construct a sequence in $\mathbb{R}^{\tilde{d}}$

$$
\begin{aligned}
&(E^{\text{job}}, \\
&W^{\text{m-avail}}((c_{t,1} - \tilde{c}_t, \dots, c_{t,m} - \tilde{c}_t)), W^{\text{j-avail}}(\max\{0, e_{t,i} - \tilde{c}_t\}), \\
&W^{\text{op}}([P^{\text{machine}}(o_{i,1}); P^{\text{operation}}(1); p_{i,1}]), \\
&\vdots \\
&W^{\text{op}}([P^{\text{machine}}(o_{i,m}); P^{\text{operation}}(m); p_{i,m}]),
\end{aligned}
$$

which is passed to the stack of Transformer blocks. We mask all operations which have already been scheduled. The first element in the output sequence corresponding to E^{job} is taken as a latent representation of the job j_i, which we denote by $j_i^{\text{latent}} \in \mathbb{R}^{\tilde{d}}$.

The job encoding procedure is depicted in Figure 5.

State encoding The network for encoding the state representation from the sequence of latent job representations consists of:

- Learnable lookup embedding E^{token} in $\mathbb{R}^{\tilde{d}}$.
- Affine maps $W^{\text{num}}, W^{\text{len}} \colon \mathbb{R} \to \mathbb{R}^{\tilde{d}}$, $W^{\text{job}} \colon \mathbb{R}^{\tilde{d}} \to \mathbb{R}^{\tilde{d}}$, and $W^{\text{m-avail-2}} \colon \mathbb{R}^m \to \mathbb{R}^{\tilde{d}}$.
- A simple stack of four Transformer blocks with four heads in the self-attention and instance normalization before the MHA and the FF.

From the set of unfinished jobs $J_t = \{j_{t,1}, \dots, j_{t,n_t}\}$, we construct a sequence

$$
\begin{aligned}
&(E^{\text{token}}, \\
&W^{\text{num}}(n_t), W^{\text{len}}(\tilde{c}_t), W^{\text{m-avail-2}}((c_{t,1} - \tilde{c}_t, \dots, c_{t,m} - \tilde{c}_t)), \\
&W^{\text{job}}(j_{t,1}^{\text{latent}}), \cdots, W^{\text{job}}(j_{t,n_t}^{\text{latent}})),
\end{aligned}
$$

which is passed to the stack of Transformer blocks. Finally, f outputs $f(s_t) = [\tilde{\boldsymbol{s}}; \tilde{\boldsymbol{a}}_1; \dots, \tilde{\boldsymbol{a}}_{n-t}]$, where $\tilde{\boldsymbol{s}}$ is the first output sequence element corresponding to E^{token}, and $\tilde{\boldsymbol{a}}_i$ corresponds to the output sequence element of the i-th unfinished job $W^{\text{job}}(j_{t,i}^{\text{latent}})$.