Algorithm Configuration in Sequential Decision-Making

Luca Begnardi[1,2] (✉), Bart von Meijenfeldt[1,2], Yingqian Zhang[1,2], Willem van Jaarsveld[1], and Hendrik Baier[1,2]

[1] Eindhoven University of Technology
{l.begnardi, b.m.v.meijenfeldt, YQZhang, W.L.v.Jaarsveld, h.j.s.baier}@tue.nl
[2] Eindhoven Artificial Intelligence Systems Institute

Abstract. Proper parameter configuration of algorithms is essential, but often time-consuming and complex, as many parameters need to be tuned simultaneously and evaluation can be expensive. In this paper, we focus on sequential decision-making (SDM) algorithms, which are applied to problems that require a series of decisions to be taken sequentially, aiming for an optimal cumulative outcome for the agent. To do this, every time the agent needs to make a decision, SDM algorithms take the current state of the environment as input and provide a decision as output. We propose a taxonomy of algorithm configuration approaches for SDM and introduce the concept of Per-State Algorithm Configuration (PSAC). To perform PSAC automatically, we present a framework based on Reinforcement Learning (RL). We demonstrate how PSAC by RL works in practice by applying it to two SDM algorithms on two SDM problems: Monte Carlo Tree Search, to solve a collaborative order picking problem in warehouses, and AlphaZero, to play a classic board game called Connect Four. Our experiments show that, in both use cases, PSAC achieves significant performance improvements compared to fixed parameter configurations. In general, our work expands the field of automated algorithm configuration and opens new possibilities for further research on SDM algorithms and their applications. Code is available at: https://github.com/ai-for-decision-making-tue/Per-State_Algorithm_Configuration.

Keywords: algorithm configuration · sequential decision-making · reinforcement learning.

1 Introduction

Proper parameter configuration is essential to ensure algorithms operate efficiently and effectively to solve specific problems. This is usually a time-consuming and difficult task, especially when many parameters need to be tuned or when the evaluation of parameter settings is expensive to compute. This algorithm configuration (AC) problem has been widely studied by the AI community [15]. Its approaches can be categorized as static or dynamic, depending on whether the

parameters are set at the beginning of algorithm execution or changed at every iteration of the running algorithm [7,2]. Static approaches can be further divided into Per-Distribution AC, where the goal is to find a single set of parameters that works well across an entire distribution of problem instances, and Per-Instance AC (PIAC), which tries to find a function that maps each individual problem instance to a specific parameter configuration. The algorithm configuration literature mainly focuses on approaches for solving optimization problems, such as local search or genetic algorithms, which aim to find an optimal solution by cleverly exploring the space of solutions, typically by iteratively improving candidate solutions. In this paper, we focus on a different class of problems, namely sequential decision-making (SDM) problems. These model real-world problems where the environment changes over time and requires making a series of decisions that lead to a cumulative outcome. In contrast to the class of algorithms usually considered in algorithm configuration, SDM algorithms can work online by outputting one decision at a time instead of a complete solution (sequence of decisions). They must, therefore, be applied multiple times with different initial environment conditions to obtain a complete solution. This approach introduces a new dimension which should be taken into account by algorithm configuration methods. While SDM algorithms are often applied in practice with a fixed set of parameters for decision-making in a given problem instance, it could be beneficial to change them according to the current state of the instance at hand. The initial board position in a game of chess for example, when all the pieces are still available, is usually very different in terms of complexity from the board state in a position close to the end of the game, when most of the pieces have likely been captured. This can have a substantial impact on the behavior and performance of SDM algorithms, leading to the natural idea of setting algorithm parameters accordingly.

We formalize this as Per-State Algorithm Configuration (PSAC) and propose a framework based on Reinforcement Learning (RL) to learn how to do it in an automated way. We then present results that demonstrate the effectiveness of our method of applying PSAC on two popular SDM algorithms: Monte Carlo Tree Search (MCTS) [11] and AlphaZero (AZ) [18], a state-of-the-art combination of MCTS and deep learning. We test the first on a warehouse assignment problem and the second on the game of Connect Four. These results highlight the potential of our algorithm configuration approach to improve the performance of sequential decision-making algorithms.

Our contributions are as follows. (1) We extend the state-of-the-art taxonomy of AC approaches provided by [2] to be applicable to sequential decision-making problems, by introducing Per-State Algorithm Configuration (PSAC). (2) We propose a framework based on reinforcement learning to perform PSAC in an automated way. (3) We test automated as well as manual PSAC on use cases belonging to two different domains: games and real-world logistics.

2 Related Works

Algorithm Configuration Most approaches to AC focus on determining the best set of parameters to apply to a certain algorithm, before it starts running. This can be done manually, by a human, or automatically, by a machine [2]. A common approach is to configure the algorithm based on the problem instance that it is trying to solve. This is known as Per-Instance Algorithm Configuration (PIAC). Recently, the idea of automatically adapting these parameters during the algorithm's execution has gained more interest. This approach is referred to as Dynamic Algorithm Configuration (DAC) and has two main lines of research: DAC by optimization and DAC by reinforcement learning. In particular, the second one was formalized in [7], and has then been applied to several classes of algorithms, including Genetic Algorithms (GA), Adaptive Large Neighborhood Search (ALNS), anytime planning algorithms, classical AI planning and deep learning algorithms such as Stochastic Gradient Descent (SGD) [2,14,20,6].

AC for sequential decision-making is a less frequently explored concept in the algorithm configuration literature. Since SDM algorithms are widely used in the field of game AI, the tuning of their parameters has been tackled with a variety of approaches over the decades. The most common approach for algorithms expected to play only a single game (instance) has been constant per-instance configuration [3]. When algorithms are expected to face various unseen games, such as in General Game Playing or General Video Game Playing, a constant configuration across all games is common, tuned offline on a selection of games [8].

Related to dynamic AC, [19] proposed a method to tune the parameters of MCTS from scratch whenever facing a new game. The term DAC could also be used to describe a variety of search algorithm enhancements that work by acquiring certain information during each individual search, such as RAVE for MCTS agents [9] or the "killer heuristic" for Alpha-Beta agents [10], as well as some techniques specifically developed for time management of game-playing algorithms [12,22].

Per-state AC methods are much more rare; exceptions include time management techniques [4], which have not been generalized outside their game domains. In the field of classical motion planners, [21] also studied how to use RL in autonomous navigation systems to adapt planner parameters to the current state of the world. Since motion planners interact with an environment that can be defined by a Markov Decision Process (MDP), the authors design a meta-MDP on which RL operates. We are currently not aware of other instances of per-state algorithm configuration in the literature. We aim to generalize this idea to other SDM algorithms and application domains.

3 Algorithm Configuration in Sequential Decision-Making

We extend the existing taxonomy of algorithm configuration approaches by introducing the new concept of per-state configuration, and propose a framework

Table 1. Taxonomy of approaches for SDM Algorithm Configuration

Approach	Objective
Per-Distribution (AC)	Constant parameter configuration
Per-Instance (PIAC)	Function that maps an instance to a configuration
Per-State (PSAC)	**Function that maps a problem state and instance to a configuration**
Dynamic (DAC)	Function that maps an algorithm state and a problem state and instance to a configuration

of algorithm configuration for SDM (Table 1). One aspect to note is that while these approaches are reported as incremental, they are effectively independent of each other. For example, while in a warehouse optimization problem it might be interesting to configure an SDM algorithm depending on both the current state of the warehouse and the problem instance, so that it can be applied to multiple warehouses, when playing strategic games, it is often sufficient to specialize on a single problem instance, solely adapting to the encountered game states. Since the focus of the current study is on the novel concept of per-state configuration, in the remainder of this paper, we will consider the problem instance as fixed.

3.1 Formulation

To formalize PSAC, we first provide a formal definition of sequential decision-making algorithm.

Definition 1. ***Sequential decision-making algorithm***

Given an instance $i \in I$ of a sequential decision-making (SDM) problem, which can be described as a Markov Decision Process (MDP) $\langle S, A, P, R \rangle$, a SDM algorithm $\mathcal{A}$ is applied, with parameters $\theta \in \Theta$, every time a decision needs to be made, taking the current state $s_t \in S$ as input and outputting the probability of taking the decision $a_t \in A$, $\mathcal{A} : S \times A \times \Theta \to \mathbb{R}_+$. The goal of a SDM algorithm is to maximize the expected cumulative return $R(T)$ over a time horizon T.

Popular SDM algorithms include simple myopic heuristics such as greedy algorithms, advanced search algorithms such as MCTS, and more sophisticated learning-based algorithms such as reinforcement learning.

As previously discussed, SDM algorithms are usually configured manually for given problem instances, and the selected parameter configuration θ is then kept fixed for every application of the algorithm to the same problem instance, disregarding its current state. To adapt the parameters to the current state of the environment, we need a function that maps s_t to the best set of parameters that should be applied to $\mathcal{A}$ to compute the next action a_t, with the objective of optimizing the cumulative outcome of applying these actions sequentially.

Definition 2. ***Per-State Algorithm Configuration (PSAC)***

Given $\langle \mathcal{P}, \mathcal{A}, \Theta, \mathcal{F} \rangle$:

- *A target instance i of a SDM problem $\mathcal{P}$, represented by the MDP $\langle S, A, P, R \rangle$.*

- *A **SDM algorithm** $\mathcal{A}$ that, if applied to state $s \in S$ with configuration parameters $\theta \in \Theta$, outputs the probability of taking decision $a \in A$, $\mathcal{A} : S \times A \times \Theta \to \mathbb{R}_+$.*
- *A space of **per-state configuration policies** $f \in \mathcal{F}$ with $f : S \times \Theta \to \mathbb{R}_+$ that map problem states to parameter configuration probabilities for $\mathcal{A}$.*

The SDM algorithm $\mathcal{A}$ interacts with the MDP to collect episodes $\tau = \{s_t, a_t, r_t\}_{t=0}^{T}$. defining a distribution over episodes

$$P(\tau) = P_0(s_0) \prod_{t=0}^{T} \mathcal{A}(a_t|s_t, \theta) P(s_{t+1}|s_t, a_t) \tag{1}$$

where $P_0(s_0) : S \to \mathbb{R}_+$ is a distribution over initial states.
If we let the configuration policy f select, at each timestep t, which θ should be applied to $\mathcal{A}$, we have

$$P(\tau) = P_0(s_0) \prod_{t=0}^{T} f(\theta_t|s_t) \mathcal{A}(a_t|s_t, \theta_t) P(s_{t+1}|s_t, a_t) \tag{2}$$

The goal is to find the configuration policy f that maximizes the expected cumulative return over $P(\tau)$:

$$G(f) = \mathbb{E}_{\tau \sim P(\tau)} \left[\sum_{t=0}^{T} \gamma^t r_t \right]. \tag{3}$$

3.2 Reinforcement Learning

Given the previous definition, maximizing the objective of equation 3 can easily be framed as a reinforcement learning problem, where the agent does not interact with the original MDP, but with a meta-MDP, which is an MDP defined by the tuple $\langle S, \Theta, P_\Theta, R \rangle$, where the states and rewards are the same as those of the original problem $\mathcal{P}$, the actions are the parameter configurations of algorithm $\mathcal{A}$, and the transition probability is now a function of the new action set $P_\Theta : S \times S \times \Theta \to \mathbb{R}_+$, which can be defined based on the original transition probability as $P_\Theta(s_{t+1}|s_t, \theta_t) = \sum_{a_t \in A} \mathcal{A}(a_t|s_t, \theta_t) P(s_{t+1}|s_t, a_t)$. Interacting with this new meta-MDP, we collect episodes $\tau_\Theta = \{s_t, \theta_t, r_t\}_{t=0}^{T}$, from which equation 2 can be updated as:

$$P(\tau_\Theta) = P_0(s_0) \prod_{t=0}^{T} f(\theta_t|s_t) P_\Theta(s_{t+1}|s_t, \theta_t) \tag{4}$$

The resulting framework is visualized in Figure 1.

4 Benchmark Problems

To illustrate the generality of Per-State Algorithm Configuration (PSAC), we decided to focus on two different sequential decision-making problems, which are well-known and relevant for different audiences and pose different challenges: game playing and stochastic combinatorial optimization.

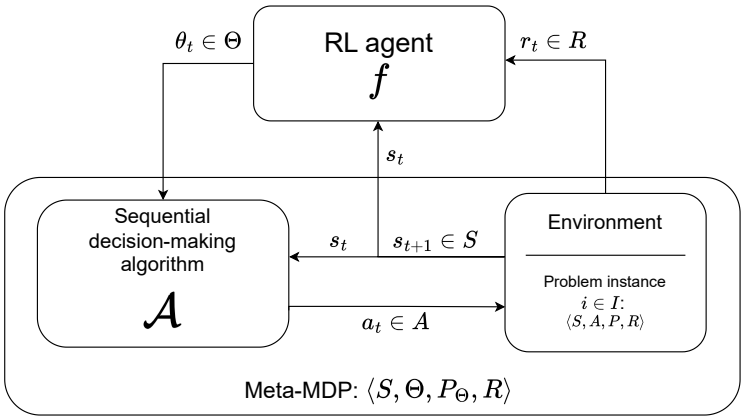


Fig. 1. Illustration of the PSAC framework. At each timestep t, the RL agent f interacts with the meta-MDP, observing the state of the current state and deciding which parameter configuration θ_t should be applied to the algorithm $\mathcal{A}$ (meta-level actions). Within the meta-MDP, the algorithm takes s_t and θ_t to compute the decision a_t that should be applied to the MDP (low-level actions).

4.1 Connect Four

We selected Connect Four (C4) as the game use case to strike the right balance between the computational resources required (e.g., each of the AZ policies [18] were trained on a minimum of 5 TPU years) and not being trivial to play well. C4 is a two-player game played on a vertically suspended grid with 6 rows and 7 columns. The players alternate between picking one of the 7 columns to place their disc in. The disc will then drop to the bottom unfilled row of the column. A player cannot play a disc in a column where all 6 rows are filled. Both players have different colored discs to differentiate theirs from their opponent's discs. The first player to connect four of their discs by placing them in a single line uninterrupted by any opponent discs wins the game. If all 7 columns are filled and neither player has managed to connect four discs, the game ends in a draw.

To create a single-player environment, we focus only on the learning of our agent and fix the opponent. In each episode, our agent will be randomly assigned to be the first or second player to move. We define the MDP as follows.

State. At each timestep t, the state s_t is the discs which are filling the slots in the 6×7 grid. Each slot can either be filled by a disc of the first player or second player, or be empty.

Actions. At each decision step, the player selects which column to drop one of its discs in. A column can only be selected if not all its rows are filled yet. After our action, if the episode is not finished, the opponent selects an action. If again the episode is not finished, the agent receives a new state.

Reward. The reward function is simply based on whether the game is won, lost or else, ongoing or a draw:

$$r(t) = \begin{cases} 1 & \text{if the agent played a winning move} \\ -1 & \text{if after the agent's move the opponent played a winning move} \\ 0 & \text{otherwise} \end{cases}$$

Objective. Since there are no intermediate rewards in this case, the objective can be both written as a summation or simply the reward of the last state: $R(T) = \sum_{t=0}^{T} r(t) = r(T)$.

4.2 Collaborative Order Picking

The collaborative human-robot order picking problem is a stochastic combinatorial optimization problem set in a modern warehouse, where human pickers and autonomous mobile robots (AMR) work side by side. AMRs handle the movement of goods, so that the pickers can focus on picking. The AMRs are usually associated with fixed sequences of pick-locations to visit, called pickruns, therefore, the task is to sequentially assign to each picker the next item to retrieve in order to further optimize the pick-rate. Since warehouses are highly stochastic, classical one-shot algorithms, which derive the solutions for the complete list of items, are not effective as they are not robust to random events occurring during plan execution. Therefore, the problem is a suitable benchmark for SDM algorithms such as MCTS.

We developed a discrete event simulation of the collaborative picking problem. The warehouse is represented as a rectangular graph, where nodes are discretized pick locations and edges are paths available between them. The simulated warehouse has a simple layout, with parallel shelves and picking aisles, without any cross-aisles. While humans can move freely, AMRs can only follow S-shaped routes and are not allowed to overtake other AMRs in the same lane. At the beginning of the simulation, every picker is in idling state. The system will then start assigning pickruns to AMRs, which are then dispatched and will immediately start traveling towards the first pick location in their list. The pickers are then also assigned to one of these pick locations depending on some configurable strategy and will start traveling to reach it. Both pickers and AMRs follow precomputed shortest paths to reach their destinations, taking into account the above-mentioned constraints, such as the S-routes. As soon as a picker and an AMR meet, the actual picking starts with a duration sampled from a parametrized distribution. Once the pick operation is over, the picker will receive a new picking task while the AMR will move to the next location. If the current pick location is the last in the pickrun, the AMR starts the drop-off operation, for which it has to reach one of the unloading stations located next to the bottom lane with need time sampled from a parametrized distribution. The AMR will then be assigned with a new pickrun and dispatched again. The speed of both AMRs and pickers is controlled by a random distribution every time a movement is processed. Finally, the system allows one to specify a set of distributions for

pick durations. This could represent, for example, that items of different sizes are being picked. These different distributions can be applied at different times during the simulation, making the overall environment non-stationary.

We now define the MDP formulation.

State. The state s_t is the set of all the nodes in the graph. Each node contains features related to the controlled picker, its distance from the controlled picker, based on both AMRs and pickers moving capabilities, the presence of AMRs, free and busy pickers, and whether a picking operation is currently happening in the location. In addition, we have two global features, related to the current pick-time distribution, including a one-hot encoding of the current pick-time distribution $\mathcal{T} \in \mathbf{T}$ and the simulation time left before the next pick-time distribution change, since the shift points are fixed and known in advance.

Actions. At each decision step only one picker can be idle, therefore the set of actions contains all the locations which are the next pick-locations in at least one AMR's pickrun, plus the current location of the picker, which will trigger a waiting event to essentially postpone the decision if it is not a pick-locations included in a pickrun.

Reward. The reward function is simply based on whether an item was collected or not since the previous timestep:

$$r(t) = \begin{cases} 1 & \text{if an item was collected between timestep } t\text{-1 and timestep } t \\ 0 & \text{otherwise} \end{cases}$$

Objective. Since the reward function effectively counts the number of items collected during an episode, the objective is to maximize that number given a fixed duration T of the episodes: $R(T) = \sum_{t=0}^{T} r(t)$.

5 Experimental setup

5.1 Algorithms

We applied PSAC to two popular SDM algorithms: MCTS and AZ. Below, we provide a brief description of how they work.

Monte Carlo Tree Search is one of the most commonly used SDM algorithms. For each encountered state, it builds a search tree incrementally and uses random sampling and simulations to estimate the values of actions and find the best one to apply. The UCT (i.e., Upper Confidence bounds applied to Trees) algorithm is a key component of MCTS, balancing exploration and exploitation by selecting actions that maximize the UCT value. The UCT is computed by $UCT = \frac{w_i}{n_i} + C\sqrt{\frac{\ln N}{n_i}}$, where w_i is the total reward of node i, n_i is the number of times node i has been visited, N is the search budget and C is a constant that balances exploration and exploitation. Every application of MCTS involves four main phases, which are repeated N times:

Selection: Starting from the root node, the algorithm traverses the tree by selecting promising nodes based on the UCT value.

Expansion: If the selected node is not a terminal state, new child nodes are added to the tree.
Simulation: A random simulation is run from the new node until either a terminal state is reached or a set amount of time, called rollout length $\boldsymbol{l}$, has passed in the simulation, to estimate the outcome.
Backpropagation: The results of the simulation are used to update the values of the nodes along the path from the new node to the root.
At the end of the process, the algorithm returns the most visited action (from the root node) to be applied in the real environment.

AlphaZero is an algorithm that adapts the MCTS algorithm to be guided by neural networks (NN). In the selection phase, an adapted version of the UCT algorithm is used, which, in addition to the average reward and the number of node visits, also considers the prior probability of selecting actions. Furthermore, a prediction is used to estimate the outcome, rather than a simulation. Both the prior probability and the outcome estimate are predicted by NNs. The NNs are trained by playing against a copy of itself in a reinforcement learning setting, where actions are selected by MCTS (which is, in turn, dependent on the NNs) to align the networks with use in MCTS. AZ has shown impressive results across a range of board games [17].

In order to test our framework in a controlled way, we decided to apply AZ to C4 and MCTS to the collaborative picking use case.

5.2 Configuration parameters

In each of our experiments, we chose to configure only a single parameter with limited range. Note that our theoretical framework does not limit the number of parameters that can be configured, but we made this choice in order to keep the analysis straightforward, while clearly showing the effectiveness of the approach.

In the case of C4, we believed that optimizing the exploration constant $\boldsymbol{C}$ would have the greatest impact. To ensure we can pick a wide variety of values for $\boldsymbol{C}$ without exploding the search space, we decided to pick our values for $\boldsymbol{C}$ on an exponential scale with base 2. The smallest value allowed is 2^{-7} and the largest 2^2 for a total of 10 values.

Since the collaborative order picking problem is highly stochastic and non-stationary, it is reasonable to assume that there should be a relation between the length of the rollouts, the amount of reward observed on average per timestep and the accuracy of the state-value estimation. Since the picking time can change, at different points in a single trajectory, the same value of rollout length $\boldsymbol{l}$ could potentially lead to collect very different amounts of reward. For example, if the average picking time is longer than the rollout length, no rewards will be observed, and the state cannot be evaluated properly. On the other hand, if the picking time is short and the rollout length is long, the stochasticity of the environment would make the observed rewards more and more uncertain, leading to an unreliable evaluation of the leaf state. For this reason, we chose

the rollout length parameter $\boldsymbol{l}$ as the target of our experiment. In particular, in every rollout, the simulation will run for a number of simulation seconds between 5 and 100, with discrete steps of 5, for a total of 20 possible values.
All other parameters we kept fixed to values that seemed reasonable for the use case, as summarized in Table 2.

Table 2. Parameter Settings for the proposed use cases

Parameter	**Collaborative Order Picking**	**Connect four**
Search budget N	250	100
Discount factor γ	0.99	0.99
Exploration constant $\boldsymbol{C}$	0.25	variable
Rollout strategy	Random	Value Function
Maximum number of chance nodes n_c	10	-

5.3 Simulation setup

Connect Four In C4 we used the standard dimensions of the grid, 6 rows and 7 columns. To make comparisons easier, we used the same AZ algorithm for our opponent and the PSAC implementation. We used the OpenSpiel library [13] to train the AZ algorithm. The AZ algorithm utilizes a residual NN with 64 nodes and a depth of 3 layers. We used the default values for all the training parameters except for the number of steps, which is set by default to run indefinitely. The parameters are as follows.
learning rate: 0.001, steps: 1000, N:100, replay buffer size:2^{16}, replay buffer reuse: 3, batch size: 2^{10}, $\boldsymbol{C}$: 2, alpha: 1, and epsilon: 0.25.

One risk of having a fixed opponent is that the agent might learn to play well against this specific opponent but is not able to play well against different opponents, i.e. does not generalize well. To mediate this, at the start of each episode, we select an opponent from a set of potential opponents for the episode. Again, to ease the interpretation of results, we decided to align this with the action space of our Meta-MDP, meaning that our set of potential opponents will be AZ agents with $\boldsymbol{C}$-values ranging from 2^{-7} to 2^2.

Another way to improve the trained agent's generalisation capabilities is to increase the diversity of the starting position. Learning how to play from a single starting position might result in the agent only learning how to play a fixed sequence of actions to win the game rather than more general patterns. We set the number of random moves at the start of each episode to 2, resulting in $7^2 = 49$ unique starting positions. We chose this number of random moves as a compromise between the variety of the starting positions and the balance of the starting position. For example, if we start with 4 random moves, positions exist in which the first player can force a win in 3 moves.

Collaborative Order Picking For the collaborative order picking case we chose to keep the parameters of the simulation environment low in terms of the number of agents involved and the size of the warehouse to allow for more controllable experiments. In all experiments, the following parameters are the same: 8 aisles, 7 pick-locations per aisle, 4 pickers, 40 AMRs, and, on average, 10 picks per AMR pickrun. The travel times of pickers and AMRs are generated from a non-negative distribution [1] with, respectively, means 1.8s and 0.8s and standard deviations 0.4s and 0.2s.[1] The duration of the drop-off operation, from the moment the AMR reaches the drop-off location, is generated from a geometric distribution with mean of 50.0s. Finally, the duration of the pick operation, from the moment both the picker and the AMR are in the right pick-location, is generated from non-negative distributions [1] $\{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_4\}$, with means {8.75, 17.5, 35.0, 70.0}s and standard deviations {1.25, 2.5, 5.0, 10.0}s. This is the only simulation parameter that will change in the experiments. Each simulation episode lasts 30 minutes of simulation time.

5.4 Deep Reinforcement Learning Algorithm

To implement PSAC we used the Proximal Policy Optimization (PPO) [16] algorithm. PPO has already been applied to many real-world SDM optimization use cases, including the collaborative human-robot order picking problem [5]. For the two domains, we used different NN backbones, depending on the input features, which are used both for the policy and the critic network. The policy network appends a softmax layer, representing the probability distribution over actions, i.e. the values of θ which can be applied to MCTS in the current timestep. The critic network appends a final layer to obtain a single value which represents the expected future reward in a state. In C4, we used a Tanh activation function since the values are bounded between -1 and 1, while in the Collaborative Order Picking use case the values can be real values, so no activation function is applied.

For C4, we used a residual NN backbone, due to the rectangular shape of the board. The input features are 4 binary 6x7 planes. The first three indicate whether a disc is on a slot on the board. The first plane considers the agent's own discs, the second the opponent's and the third whether the slots are unfilled. The fourth plane indicates whether the agent was the player to start. The residual backbone used two 3x3 residual blocks with 64 filters, batch normalization and a ReLu activation. After the two residual blocks, the output is flattened, and a single ReLu layer with 64 units is used.

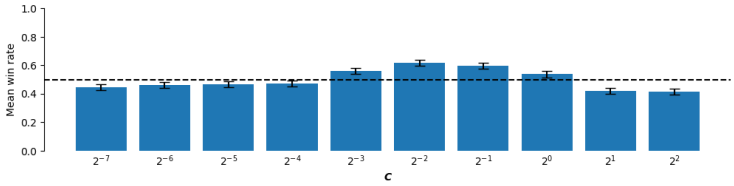
For Order Picking, the architecture consists of a multilayer perceptron (MLP) with one hidden layer of size 64, that independently processes the features of each node with the same set of weights to create node embeddings, which are then aggregated with a mean pooling layer. The two global features are also processed by two dedicated MLPs with hidden layers of size 64. The resulting embeddings are then concatenated and passed through the output layer.

[1] These times are applied every time the entities move from one location to another.

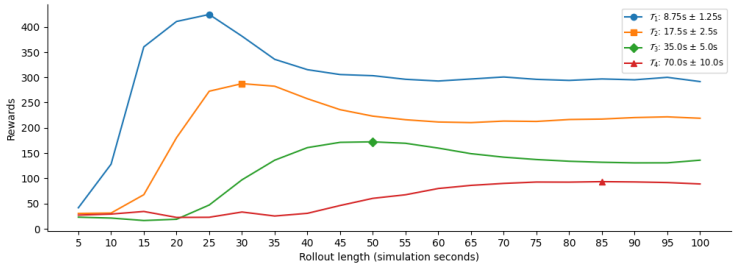
The Tianshou library was used in this work for the implementation of PPO, and all the parameters of the algorithm are default, while PyTorch was used for implementing the NN architectures. All experiments were run with 16GB of RAM and without the use of GPU. The training and testing experiments for C4 were performed using 16 cores of an AMD EPYC(TM) Rome 7H12 CPU @ 2.6 GHZ and for the collaborative order picking problem on an Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz. The final configuration policy for C4 was trained for 800,000 timesteps (∼18 hours) and 500,000 timesteps(∼8 hours) for the collaborative order-picking problem.

5.5 Baselines

Figure 2 shows the results of the per-instance baseline experiments which were run to obtain the baselines for the two use cases.



(a)



(b)

Fig. 2. Results of per-instance agents on Connect Four (a) and on the stationary environments of collaborative picking (b). (a) Mean win rate (draws counting as half wins) of a per-instance AlphaZero configuration vs. a range of AlphaZero opponents configured with $\boldsymbol{C} \in \{2^{-7}, \ldots, 2^{2}\}$. Each opponent was played 200 times, half as first to act, half as second to act, resulting in each bar representing 2000 episodes. The whiskers represent 95% confidence intervals. (b) Average rewards over 10 episodes of MCTS applied with fixed values of $\boldsymbol{l}$ to four stationary environments.

In C4, per-instance agents were evaluated by running match-ups for each value of $\boldsymbol{C}$ of the action space. Figure 2(a) shows that the performance has an

unimodal shape with the performance peaking at a value of 2^{-2} for $\boldsymbol{C}$. Therefore, we used this value for the per-instance baseline agent.

For the collaborative picking use case, we manually created a configuration function, as follows: we first ran a grid search on stationary versions of the environment, where the picking time distribution $\mathcal{T} \in \mathbf{T}$ is fixed, to find the best value of $\boldsymbol{l}$ for each of those distributions. These values are highlighted in Figure2(b). The configuration function, which we will call ***manual* PSAC** from now on, as opposed to PSAC, which refers to the learned policy, picks, at each step, the value corresponding to the current distribution type $\mathcal{T}$. In addition, we also compared the results of the learned PSAC agent with a myopic greedy heuristic that assigns the pickers to the closest pick-location which is the next step in the pickrun of an AMR and is not currently assigned to another picker.

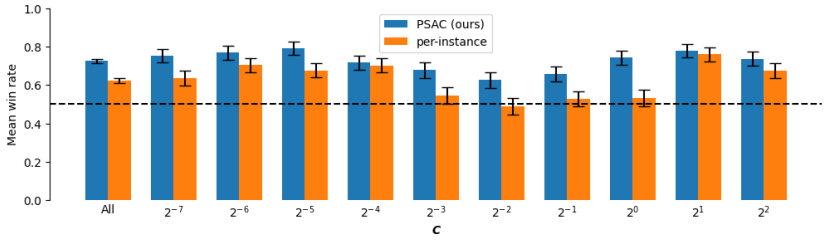


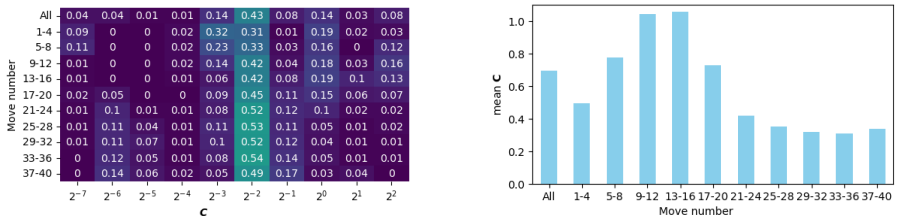
Fig. 3. Performance PSAC agent vs per-instance baseline. Mean win rate (draws counting as half wins) vs. AlphaZero opponents configured with different $\boldsymbol{C}$. Each opponent was played 500 times, half as first to act, half as second to act. The whiskers represent 95% confidence intervals.

6 Results

Connect Four To evaluate our approach, we have the PSAC agent and the optimized per-instance agent play against the same pool of opponents. The results are reported in Figure 3. Our PSAC agent significantly outperforms the per-instance agent over all the matchups. Furthermore, the performance against the majority of specific AZ agents is significantly better as well, and, in all cases, better than that against the per-instance agent.

We also show in Figure 4 the values for $\boldsymbol{C}$ that our PSAC agent selects. Surprisingly, Figure 4(a) shows that the shape of the probability mass function is not uni-modal (as for the per-instance performances in Figure 2(a)) but multi-modal. Although it generally selects the same value for $\boldsymbol{C}$ as the agent per instance, at times it also prefers to select very small values (2^{-7} or 2^{-6}) or very large values (2^2). In Figure 4(b) we can see the relation of the mean $\boldsymbol{C}$ versus the move number. At the very start of the episode, the mean values are low; around move 13, they peak and towards the end, they decrease again. We hypothesize

that this is due to the first positions have been seen very frequently so AZ is quite confident that the best move is one of its most preferred moves, later on, the positions are seen less frequently and many alternatives seem promising so it will search more broadly, then near the end, there are fewer relevant alternatives again (some moves will be clearly losing so need not be considered) and the precise nature of the end game requires looking farther into fewer variations.



(a) (b)

Fig. 4. Analysis of selected C values. Both plots are based on all the episodes of Figure 3. (a) Probability mass function of the configured $\boldsymbol{C}$ vs. move number. (b) Mean $\boldsymbol{C}$ vs. move number.

Collaborative Order Picking The rewards obtained running a grid search for the configuration of $\boldsymbol{l}$ in MCTS on the non-stationary environment, as well as running PSAC are reported in Table 3, together with the baselines discussed in the previous section. In addition, the results shown in Figure 2(b) are also added, both to compare them with the greedy baseline and to highlight the best $\boldsymbol{l}$ parameters found for each pick-time distribution.

First, we can see that in every stationary environment, the best configuration of MCTS consistently outperforms the myopic heuristic. This is not true for the non-stationary environment, where every fixed configuration of MCTS performs worse than myopic. Secondly, manual PSAC clearly outperforms the myopic one, proving that per-state configuration of MCTS is necessary to achieve good results in this environment. Lastly, we consider the performance of PSAC. Learning how to configure the $\boldsymbol{l}$ parameter of MCTS during episodes, our method is able to considerably outperform every instance of MCTS with fixed parameters, as well as the myopic heuristic. While manual PSAC still provides the best result, Figure 5 shows that PSAC learns a very similar behavior. Since manual PSAC requires manual labor and domain knowledge, it becomes impractical and less feasible to configure multiple parameters at the same time.

7 Conclusion

In this paper, we propose to adapt the state-of-the-art taxonomy of AC to SDM algorithms, introducing the concept of Per-State Algorithm Configuration. We

Table 3. Performances of per-instance MCTS, baseline heuristics and PSAC, in terms of means and standard deviations of rewards over 10 episodes.

Method	Environment				
	$\mathcal{T}_1$: **8.75 ± 1,25**	$\mathcal{T}_2$: **17.5 ± 2.5**	$\mathcal{T}_3$: **35.0 ± 5.0**	$\mathcal{T}_4$: **70.0 ± 10.0**	**Non-stationary**
MCTS, l: 5s	41.4 ± 34.65	30.2 ± 20.8	22.8 ± 17.17	26.9 ± 15.51	33.0 ± 24.52
MCTS, l: 10s	127.8 ± 65.5	31.0 ± 18.36	21.0 ± 23.94	28.9 ± 18.11	45.9 ± 22.51
MCTS, l: 15s	360.2 ± 13.14	67.3 ± 36.8	16.2 ± 11.91	34.1 ± 12.36	66.0 ± 18.41
MCTS, l: 20s	410.6 ± 15.87	180.3 ± 47.52	18.6 ± 14.93	22.3 ± 18.04	81.6 ± 18.56
MCTS, l:25s	**424.5 ± 13.57**	272.4 ± 8.49	46.9 ± 13.64	22.6 ± 15.25	89.9 ± 10.21
MCTS, l: 30s	381.5 ± 7.41	**287.4 ± 5.87**	96.8 ± 8.74	33.0 ± 18.57	109.1 ± 12.75
MCTS, l: 35s	335.6 ± 9.22	282.3 ± 5.68	135.7 ± 7.09	25.1 ± 13.41	118.4 ± 11.17
MCTS, l: 40s	315.1 ± 8.14	257.5 ± 4.25	160.6 ± 3.83	30.4 ± 17.54	122.0 ± 10.7
MCTS, l: 45s	305.4 ± 6.93	235.8 ± 4.42	171.1 ± 3.24	45.9 ± 11.27	127.5 ± 5.24
MCTS, l: 50s	303.2 ± 8.27	223.0 ± 6.16	**172.1 ± 3.7**	60.1 ± 2.43	129.6 ± 3.5
MCTS, l: 55s	296.0 ± 8.15	215.9 ± 4.5	169.2 ± 1.99	67.2 ± 2.71	130.5 ± 4.52
MCTS, l: 60s	292.6 ± 10.52	211.4 ± 5.28	159.7 ± 2.24	79.6 ± 3.69	136.1 ± 4.21
MCTS, l: 65s	296.6 ± 6.39	210.2 ± 3.68	148.6 ± 3.1	85.8 ± 2.64	133.7 ± 3.63
MCTS, l: 70s	300.6 ± 15.27	213.2 ± 5.42	141.7 ± 1.95	89.7 ± 2.83	134.8 ± 3.57
MCTS, l: 75s	295.9 ± 7.62	212.5 ± 8.87	137.0 ± 2.45	92.4 ± 1.62	134.7 ± 3.41
MCTS, l: 80s	293.8 ± 8.28	216.3 ± 7.96	133.6 ± 2.94	92.2 ± 1.72	135.0 ± 2.83
MCTS, l: 85s	296.7 ± 12.53	217.2 ± 7.39	131.6 ± 1.56	**93.2 ± 1.54**	134.4 ± 4.45
MCTS, l: 90s	295.0 ± 5.31	220.2 ± 8.39	130.4 ± 3.38	92.6 ± 2.06	136.8 ± 4.69
MCTS, l: 95s	299.9 ± 12.49	221.6 ± 6.9	130.5 ± 3.11	91.4 ± 1.43	133.8 ± 4.49
MCTS, l: 100s	291.5 ± 8.24	218.8 ± 6.6	135.7 ± 2.93	88.6 ± 1.36	134.0 ± 3.1
Myopic heuristic	331.3 ± 10.87	245.0 ± 7.29	154.3 ± 3.74	86.3 ± 1.49	145.2 ± 5.1
Manual PSAC	-	-	-	-	**161.0 ± 6.72**
PSAC	-	-	-	-	154.45 ± 3.92

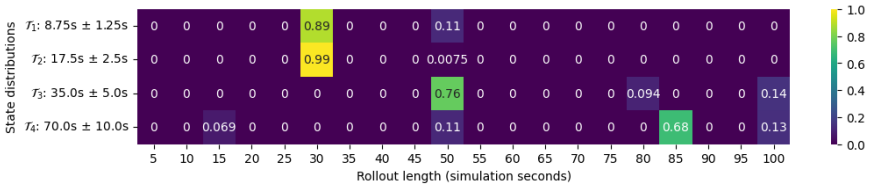


Fig. 5. Analysis of selected l values. Analysis of the l value selected by the PSAC agent for different $\mathcal{T}$ values in the non-stationary environment.

then present a framework for doing that in an automated way with RL and show significant performance improvements for MCTS and AlphaZero on two different domains: a deterministic two-player game and a stochastic combinatorial optimization problem, inspired by a real-world use case. Future research directions include testing our method on a broader set of problems, including more complex games and large-scale real-world use cases as well as simpler white-box benchmarks, which would allow us to perform a deeper ablation analysis of the framework. Additionally, we plan to apply PSAC to configure multiple parameters simultaneously, to further improve the algorithm performances. Lastly, we intend to analyze more deeply in what type of states certain parameters are selected by PSAC. In general, our work extends the current field of automated algorithm configuration, opening up new possibilities for further research on sequential decision-making algorithms and their applications.

Acknowledgments. This research was made possible by TKI Dinalog and the Top-sector Logistics and has received funding from the Ministry of Economic Affairs and Climate Policy (EZK) of the Netherlands and from the European Union's Horizon Europe Research and Innovation Programme, under Grant Agreement number 101120406. The paper reflects only the authors' view and the EC is not responsible for any use that may be made of the information it contains. This work used the Dutch national e-infrastructure with the support of the SURF Cooperative using grant no. EINF-9770.

References

1. Adan, I., van Eenige, M., Resing, J.: Fitting discrete distributions on the first two moments. Probability in the Engineering and Informational Sciences **9**(4), 623–632 (1995). https://doi.org/10.1017/S0269964800004101
2. Adriaensen, S., Biedenkapp, A., Shala, G., Awad, N., Eimer, T., Lindauer, M., Hutter, F.: Automated dynamic algorithm configuration. Journal of Artificial Intelligence Research **75**, 1633–1699 (12 2022). https://doi.org/10.1613/jair.1.13922
3. Baier, H., Kaisers, M.: Guiding multiplayer MCTS by focusing on yourself. In: IEEE Conference on Games, CoG 2020, Osaka, Japan, August 24-27, 2020. pp. 550–557. IEEE (2020). https://doi.org/10.1109/COG47356.2020.9231603, https://doi.org/10.1109/CoG47356.2020.9231603
4. Baier, H., Winands, M.H.M.: Time management for monte carlo tree search. IEEE Trans. Comput. Intell. AI Games **8**(3), 301–314 (2016). https://doi.org/10.1109/TCIAIG.2015.2443123, https://doi.org/10.1109/TCIAIG.2015.2443123
5. Begnardi, L., Baier, H., van Jaarsveld, W., Zhang, Y.: Deep reinforcement learning for two-sided online bipartite matching in collaborative order picking. In: Asian Conference on Machine Learning. pp. 121–136. PMLR (2024)
6. Bhatia, A., Svegliato, J., Nashed, S.B., Zilberstein, S.: Tuning the hyperparameters of anytime planning: A metareasoning approach with deep reinforcement learning. Proceedings of the International Conference on Automated Planning and Scheduling **32**(1), 556–564 (Jun 2022). https://doi.org/10.1609/icaps.v32i1.19842
7. Biedenkapp, A., Bozkurt, H.F., Eimer, T., Hutter, F., Lindauer, M.: Dynamic Algorithm Configuration: Foundation of a New Meta-Algorithmic Framework. Santiago de Compostela (2020)
8. Finnsson, H., Björnsson, Y.: Cadiaplayer: Search-control techniques. Künstliche Intell. **25**(1), 9–16 (2011). https://doi.org/10.1007/S13218-010-0080-9, https://doi.org/10.1007/s13218-010-0080-9
9. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Ghahramani, Z. (ed.) Machine Learning, Proceedings of the Twenty-Fourth International Conference (ICML 2007), Corvallis, Oregon, USA, June 20-24, 2007. ACM International Conference Proceeding Series, vol. 227, pp. 273–280. ACM (2007). https://doi.org/10.1145/1273496.1273531, https://doi.org/10.1145/1273496.1273531
10. Huberman, B.J.: A program to play chess end games. Ph.D. thesis, Stanford University, USA (1968), https://searchworks.stanford.edu/view/2190328
11. Kocsis, L., Szepesvári, C.: Bandit Based Monte-Carlo Planning. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) Machine Learning: ECML 2006, vol. 4212, pp. 282–293. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1007/11871842_29, series Title: Lecture Notes in Computer Science

12. Lan, L.C., Wu, T.R., Wu, I.C., Hsieh, C.J.: Learning to Stop: Dynamic Simulation Monte-Carlo Tree Search. Proceedings of the AAAI Conference on Artificial Intelligence **35**(1), 259–267 (May 2021). https://doi.org/10.1609/aaai.v35i1.16100
13. Lanctot, M., Lockhart, E., Lespiau, J.B., Zambaldi, V., Upadhyay, S., Pérolat, J., Srinivasan, S., Timbers, F., Tuyls, K., Omidshafiei, S., et al.: Openspiel: A framework for reinforcement learning in games. arXiv preprint arXiv:1908.09453 (2019)
14. Reijnen, R., Zhang, Y., Lau, H.C., Bukhsh, Z.: Online Control of Adaptive Large Neighborhood Search using Deep Reinforcement Learning. Proceedings of the International Conference on Automated Planning and Scheduling **34**, 475–483 (May 2024). https://doi.org/10.1609/icaps.v34i1.31507
15. Schede, E., Brandt, J., Tornede, A., Wever, M., Bengs, V., Hüllermeier, E., Tierney, K.: A survey of methods for automated algorithm configuration. Journal of Artificial Intelligence Research **75**, 425–487 (2022)
16. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal Policy Optimization Algorithms (Aug 2017), arXiv:1707.06347 [cs]
17. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (2018)
18. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of go without human knowledge. nature **550**(7676), 354–359 (2017)
19. Sironi, C.F., Liu, J., Winands, M.H.M.: Self-Adaptive Monte Carlo Tree Search in General Game Playing. IEEE Transactions on Games **12**(2), 132–144 (Jun 2020). https://doi.org/10.1109/TG.2018.2884768
20. Speck, D., Biedenkapp, A., Hutter, F., Mattmüller, R., Lindauer, M.: Learning heuristic selection with dynamic algorithm configuration. Proceedings of the International Conference on Automated Planning and Scheduling **31**, 597–605 (05 2021). https://doi.org/10.1609/icaps.v31i1.16008
21. Xu, Z., Dhamankar, G., Nair, A., Xiao, X., Warnell, G., Liu, B., Wang, Z., Stone, P.: Applr: Adaptive planner parameter learning from reinforcement. In: 2021 IEEE International Conference on Robotics and Automation (ICRA). pp. 6086–6092 (2021). https://doi.org/10.1109/ICRA48506.2021.9561647
22. Ye, W., Abbeel, P., Gao, Y.: Spending thinking time wisely: Accelerating mcts with virtual expansions. In: Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A. (eds.) Advances in Neural Information Processing Systems. vol. 35, pp. 12211–12224. Curran Associates, Inc. (2022)