On-the-Fly Model Checking with Neural MCTS

Ruiyang Xu(✉) and Karl Lieberherr

Khoury College of Computer Sciences, Northeastern University, Boston, MA 02115, USA
{xu.r,k.lieberherr}@northeastern.edu

Abstract. Recent progress in AI, which combines deep learning with classical search algorithms, has shown remarkable performance improvements for several challenging board games, such as Go and Chess. In this paper, we propose a method to apply this new technique to model checking problems. In particular, we leverage the game-theoretical semantics of logic expressions (recursive first-order logic in our case) to turn a model checking problem into a two-player perfect information win-lose game. The game can then be played and learned by a deep learning and search algorithm (neural MCTS). The existence of a winning strategy of a player indicates that either the model-checked property can be verified or there is a counterexample. We modified the classical neural MCTS algorithm to ensure it can handle cycles when searching in state space. We also propose a way to incorporate fairness constraints into the learning and search process. We test our idea on two labeled transition systems (one is from a numerical game, and the other is the classical Dining Philosophers problem). Our experimental results show an outperformance of our method compared with reinforcement-learning-based model checking approaches.

Keywords: Neural MCTS · Model checking · On-the-fly

1 Introduction

The world has entered a new era where large distributed concurrent systems have been developed to serve numerous clients worldwide. Those systems cover a large part of our daily life, such as finance and transportation. A tiny mistake in the design could potentially incur a severe security issue and cause economic losses. Therefore, it is crucial to keep the design of those systems correct. That is why model-checking deserves to be paid more attention to nowadays. However, the model-checking community has long been troubled by the *state explosion problem* [7], namely as the number of state variables in the system increases, the size of the system state space grows exponentially.

The past few years have witnessed a combination of search algorithms and machine learning (ML) techniques (i.e., neural MCTS) showing a remarkable

J. V. Deshmukh et al. (Eds.): NFM 2022, LNCS 13260, pp. 557–575, 2022.
https://doi.org/10.1007/978-3-031-06773-0_30

performance improvement when dealing with games that have large state spaces, such as Go and Chess [22–25]. Because states information is learned and stored by neural networks, those algorithms' memory usage can be considered constants. Being motivated by the recent progress of AI in gameplay, we came up with an alternative approach to tackle the state explosion problem. Specifically, it has been known for decades that the game-theoretical semantics [13], which shows a duality between logic and game, can be utilized for model-checking [27]. Although the goal of model-checking is to find errors that can appear in some rare cases, the game-theoretical semantics allows one to reduce the problem of verifying a property into a two-player semantic game defined on the logic used to describe that property. Such a reduction (or gamification) allows one to apply a self-play-based ML algorithm to learn from the game and eventually solve the problem.

A high-level view of the Neural MCTS algorithm for an interpreted sentence ϕ for some logic L with game-semantics is given by the following loop:
Repeat

- Find faulty predictions (TRUE or FALSE) for ϕ and its sub-formulas, called $curriculum(\phi)$, through self-play of $Game(L, \phi)$.
- Learn from the faulty predictions $curriculum(\phi)$, which gives negative reward information based on winning/losing predictions and self-play outcomes.
- Update approximation to value function, which predicates the winning/losing chance, and policy functions, which approximate a serial of Skolem functions used as a strategy to prove or disprove the formula.

Until convergence: there are no faulty predictions for ϕ and its sub-formulas.

Learning based on self-play is basically a process of finding faulty predictions for each player. A faulty prediction is one where the prediction of the game outcome (from the value function) contradicts the actual outcome. Since the game is zero-sum with no draws, the two players will continuously compete by mutually creating curricula for each other to learn until there is no faulty prediction anymore. Consequently, a winning strategy learned by an ML algorithm for that game shows how to verify/falsify that property. It should be noted that even when the two players agree with each other, their prediction might still be wrong because the agreement on a truth value might be based on weak players. This is a common issue for all ML-based model checkers because ML algorithms are probabilistic, and 100% correctness is not guaranteed. Therefore, "*they should favor the discovery of errors rather than focusing on guaranteeing correctness*" [8]. That is also the reason why learning a strategy to falsify a property is especially useful when constructing a counterexample from the model-checking problem.

Leveraging such a duality, in this paper, we show an approach to adapt neural MCTS to on-the-fly model-checking through semantic-game-based gamification. In particular, we use recursive first-order logic as our model specification language, which provides us with a novel approach to generate state representations. In addition to that, we also propose a method to impose fairness constraints by concatenating a normalized counter vector to the state representation. We test our approach on two problems: model-checking an alternating reachability property on a numeric game and a liveness property on the well-known dining philosopher problem. Our preliminary experimental results show that turning model-checking problems into games and solving them with cutting-edge game-play AI technology might be a promising research direction.

2 Preliminary

2.1 Model Checking with L_μ

Modal μ-calculus (L_μ) has been used broadly in model checking, a logic used to describe properties of the target labeled transition system (LTS). An LTS is defined as a tuple (S, I, A, T), where S is a set of states, I is a nonempty subset of S of initial states, A is a set of actions, and relation T formulates transitions among different states, which are labeled with actions. With an LTS, one can abstract and model the possible development of a system. The problem of model-checking a L_μ formula on a transition system is to decide whether the LTS satisfies the formula.

The syntax of L_μ is:

$$\Phi := \mathbf{True} \mid \mathbf{False} \mid X \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg\Phi \mid [A]\Phi \mid \langle A\rangle\Phi \mid \nu X.\Phi \mid \mu X.\Phi,$$

where X ranges over a set of variables, regarded as names of predicates. We also use $\sigma X.\Phi$ to stand for either $\nu X.\Phi$ or $\mu X.\Phi$. The semantics of L_μ can be represented with Monadic Second-order Logic (MSOL) since it has been proved that L_μ is the bisimulation invariant fragment of MSOL [15]. To be specific, let $\Phi[x]$ be the MSOL translation of a L_μ formula Φ, with one free variable x. Then a L_μ formula can be translated into an MSOL formula recursively in the following way:

$$\begin{aligned}
&X[x] = X(x)\\
&(\Phi_1 \wedge \Phi_2)[x] = \Phi_1[x] \wedge \Phi_2[x]\\
&(\Phi_1 \vee \Phi_2)[x] = \Phi_1[x] \vee \Phi_2[x]\\
&(\neg\Phi)[x] = \neg\Phi[x]\\
&\langle A\rangle\Phi[x] = \exists y \in S.\ (xTy) \wedge \Phi[y]\\
&[A]\Phi[x] = \forall y \in S.\ (xTy) \rightarrow \Phi[y]\\
&\mu X.\Phi[x] = \exists X \subseteq S.\ (\forall y \in S.\ \Phi[y] \rightarrow X(y)) \wedge X(x)\\
&\nu X.\Phi[x] = \exists X \subseteq S.\ (\forall y \in S.\ X(y) \rightarrow \Phi[y]) \rightarrow X(x)
\end{aligned} \tag{1}$$

where $X(x)$ means for some set $X \subseteq S$, $x \in X$.

It is to be noted that the definition of fix-point operator **LFP** and **GFP** is intricate, for which we use the explanation from [28]. To put it simply, let $\Phi(x; X)$ be any MSOL predicate parameterized on some set X. And suppose $S_X = \{x|\Phi(x; X)\}$, then a MSOL predicate actually also defines a function which maps X to S_X. The fix-point of the function can be computed by recursively calling the function with its output. To be specific, the **LFP** of a predicate $\Phi(x;X)$ is derived by a sequence of function calls:

$$S_0 = \{x|\Phi(x;\varnothing)\}, S_1 = \{x|\Phi(x;S_0)\}, ..., S_n = \{x|\Phi(x;S_n)\}$$

$$\textbf{LFP}.\ \Phi(x;X) := S_n,$$

and similarly, the **GFP** of a predicate $\Phi(x;X)$ can be derived by the sequence: $\Phi(x;X)$ is derived by a sequence of function calls:

$$S_0 = \{x|\Phi(x;S)\}, S_1 = \{x|\Phi(x;S_0)\}, ..., S_n = \{x|\Phi(x;S_n)\}$$

$$\textbf{GFP}.\ \Phi(x;X) := S_n.$$

As a result, the fix-point operators and the two modal operators make it possible to express the finite or infinite temporal properties of an LTS.

2.2 Game Theoretical Semantics

Game theoretical semantics is an approach that rebuilds the logical concepts with game-theoretic concepts. A logical formula is interpreted as a game between two players, one in the Proponent role and the other in the Opponent role. The game runs recursively on the computational order of the logical operators. The game ends when a primitive predicate is achieved, and the Proponent wins if the formula evaluates to true; otherwise, the Opponent wins. A winning strategy can be represented by a finite sequence of Skolem functions (which are useful tools to improve the system design) corresponding to the moves made by the player relative to those played by the other one [21].

To better understand the concept, we first introduce the game theoretical semantics of first-order logic (FOL) [13,21]. A semantic game is represented as a tuple $(\Psi,\ \mathrm{P},\ \mathrm{OP})$, where the Ψ is a formula interpreted by a structure M. P and OP denote the game role for each of the two players, and, initially, player-1 plays the P role. The game rule can be summarized in Table 1.

Table 1. The game semantics for a FOL formula φ. In this table, OP stands for Opponent, and P stands for Proponent. The game ends at an atomic predicate φ. It is to be noted that the negation switches the role of the two players; namely, strategies for P in a game for $\neg\Psi$ are strategies for OP in the game for Ψ.

Formula	Operation	Subgame
$\forall x \in A : \Psi(x)$	OP picks x_0 from A	$(\Psi[x/x_0],\ \mathrm{P},\ \mathrm{OP})$
$\exists x \in A : \Psi(x)$	P picks x_0 from A	$(\Psi[x/x_0],\ \mathrm{P},\ \mathrm{OP})$
$\Psi \wedge X$	OP picks $\Theta \in \{\Psi, X\}$	$(\Theta,\ \mathrm{P},\ \mathrm{OP})$
$\Psi \vee X$	P picks $\Theta \in \{\Psi, X\}$	$(\Theta,\ \mathrm{P},\ \mathrm{OP})$
$\neg\Psi$	N/A	$(\Psi,\ \mathrm{OP},\ \mathrm{P})$
φ	N/A	N/A

The concept of game theoretical semantics can also be extended to L_μ [12,20,27]. However, since L_μ is involved with a transition system, the game rule is more complex (Table 2). Even though the modal operators $[A]$ and $\langle A\rangle$ resemble the quantifiers in FOL, the fix-point operator is utterly distinct, which actually grants L_μ more expressiveness than FOL [18]. As a result, the winning condition is no longer as simple as the one with FOL. Instead of deciding whether a formula can be evaluated as true or false, one may encounter situations where one of the players cannot pick any transition in the system because of a deadlock. Alternatively, one may encounter situations where the game is just running forever because of a cycle in the transition system. In summary, the updated winning condition can be summarized as the following [27]:

- Proponent wins, when either
  - within finitely many moves, the formula can be evaluated to **true** for the underlying transition system.
  - within finitely many moves, the Opponent gets into a deadlock where no transition is available.
  - the game can run forever because of a greatest fix-point operator $\nu X.\Psi$, which indicates that a safety property can always hold.
- Opponent wins, when either
  - within finitely many moves, the formula can be evaluated to **false** for the underlying transition system.
  - within finitely many moves, Proponent gets into a deadlock where no transition is available.
  - the game can run forever because of a least fix-point operator $\mu X.\Psi$, which indicates that a liveness property can never hold.

Table 2. The game semantic for a L_μ formula φ with underlying transition system state S, where $X \lhd \sigma X.\Psi$ means X is bound by $\sigma X.\Psi$. Notice that the game might not always end at a primitive predicate. Due to the nature of L_μ, it may end at a deadlock or just run forever.

Configuration	Operation	Subgame
$([A]\Psi)[x]$	OP picks a transition $x \xrightarrow{a\in A} y$	$(\Psi[y],\ \text{P},\ \text{OP})$
$(\langle A\rangle\Psi)[x]$	P picks a transition $x \xrightarrow{a\in A} y$	$(\Psi[y],\ \text{P},\ \text{OP})$
$(\Psi \wedge X)[x]$	OP picks $\Theta \in \{\Psi, X\}$	$(\Theta[x],\ \text{P},\ \text{OP})$
$(\Psi \vee X)[x]$	P picks $\Theta \in \{\Psi, X\}$	$(\Theta[x],\ \text{P},\ \text{OP})$
$(\neg\Psi)[x]$	N/A	$(\Psi[x],\ \text{OP},\ \text{P})$
$(\sigma X.\Psi)[x]$	N/A	$(\Psi[x],\ \text{P},\ \text{OP})$
$X[x]$	N/A	$(\Psi[x],\ \text{P},\ \text{OP})$, $X \lhd \sigma X.\Psi$
$\varphi[x]$	N/A	N/A

The game semantics of L_μ gives a local view on the model checking problem, while the MSOL semantics of L_μ provides a global view. Typically those fix-point operators, from a global view, define a closure of LTS state in which certain temporal property always holds; yet from a local view, they describe recursive behaviors so that the evolution of LTS forms a cycle. The local view, acquired from game semantics, turns out to be more intuitive to help us understand or design a L_μ property. For instance, $\nu X.\Phi$ defines a "good" cycle which means something good should always happen; otherwise, the system is not well designed; while $\mu X.\Phi$ specify a "bad" cycle which means, eventually something good should happen; otherwise the system is not well designed.

2.3 Learning with Neural MCTS

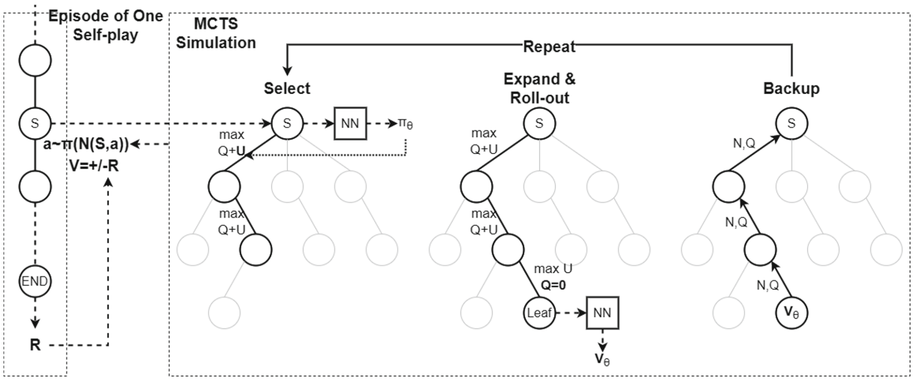


Fig. 1. The workflow of the neural MCTS algorithm.

MCTS has been applied to solving combinatorial games for a long time [6], while recently, combining deep neural networks with MCTS showed success in improving solver competence in many practical combinatorial games. The concept of neural MCTS was proposed independently in Expert Iteration [1], and AlphaZero [25]. In a nutshell, neural MCTS uses the neural network as policy and value approximators. During each learning iteration, it carries out multiple rounds of self-plays. Each self-play runs several MCTS simulations to estimate an empirical policy at each state, then sample from that policy, take a move, and continue. After each round of self-play, the game's outcome is backed up to all states in that game episode. Those game episodes generated during self-play are then stored in a replay buffer, which is used to train the neural network (Fig. 1).

During one self-play episode, for a given state, the neural MCTS runs a given number of simulations on a game tree, rooted at that state to generate an empirical policy. Each simulation, guided by the policy and value networks, passes through 4 phases:

1. **SELECT**: At the beginning of each iteration, the algorithm selects a path from the root (current game state) to a leaf (either a terminal state or an

unvisited state) according to an upper confidence boundary (UCB, [3,4,17]). Specifically, suppose the root is s_0. The UCB determines a sequence of states $\{s_0, s_1, ..., s_l\}$ by the following process:

$$\begin{aligned} a_i &= \arg\max_a \left[Q(s_i, a) + \underbrace{\alpha \pi_\theta(s_i, a) \frac{\sqrt{\sum_{a'} N(s_i, a')}}{N(s_i, a) + 1}}_{U(s_i,a)} \right] \\ s_{i+1} &= \text{move}(s_i, a_i) \\ Q(s_i, a) &= N(s_i, a) = 0, \text{if } s_{i+1} \text{ is unvisited} \end{aligned} \tag{2}$$

where α is a tunable parameter, $N(s, a)$ counts the times of visiting (s, a) during the MCTS simulations, and $Q(s, a)$ is a state-action value estimator. The UCB is also guided by a policy estimator $\pi_\theta(s, a)$. It has been proved in [9] that selecting actions using Eq. 2 is equivalent to optimize the empirical policy

$$\hat{\pi}(s, a) = \frac{1 + N(s, a)}{|A| + \sum_{a'} N(s, a')}$$

where $|A|$ is the size of the action space, so that it approximates the solution of a regularized policy optimization problem. As a result, MCTS simulation can be regarded as a regularized policy optimization [9]. As long as the value network is accurate, the MCTS simulation will optimize the output policy to maximize the action value output while minimizing the change to the policy network.

2. **EXPAND**: Once the selected phase ends at an unvisited state s_l, the state will be fully expanded and marked as visited. During the next selection iteration, all its child nodes will be considered leaf nodes.
3. **ROLL-OUT**: The roll-out is carried out for any unvisited state s_l. If s_l is a terminal state, the game outcome $R(s_l)$ will be used as the state value backup for the **BACKUP** phase, otherwise, the algorithm will use a value network to estimate the result of the game (from current state s_l) and use that value $V_\theta(s_l)$ for **BACKUP**.
4. **BACKUP**: This is the last phase of an MCTS simulation in which the algorithm backs up the state value and updates the state-action value estimator for each node in the selected states sequence. To illustrate this process, suppose the selected states and corresponding actions and players are

$$\{(s_0, a_0, p_0), (s_1, a_1, p_1), ...(s_{l-1}, a_{l-1}, p_{l-1}), (s_l, \_, p_l)\}$$

Let v_l be either the actual game outcome $R(s_l)$ or the estimated outcome $V_\theta(s_i)$. The value is then backed up in the following way

$$\{(s_0, a_0, p_0, v_0), (s_1, a_1, p_1, v_1), ...(s_{l-1}, a_{l-1}, p_{l-1}, v_{l-1}), (s_l, \_, p_l, v_l)\},$$

where $v_i = (-1)^{|p_{i+1} - p_i|} v_{i+1}$. In other words, for any two-player game, the leaf state value v_l is backed up in a fashion such that states play by the same player as the leaf state will be assigned the same value v_l, while states play by another player will be assigned the opposite value $-v_l$. The backed up state values are then be used to update the counter N and state-action value estimator Q:

$$\begin{aligned} N(s_t, a_t) &\leftarrow N(s_t, a_t) + 1 \\ Q(s_t, a_t) &\leftarrow Q(s_t, a_t) + \frac{V_\theta(s_r) - Q(s_t, a_t)}{N(s_t, a_t)} \end{aligned} \tag{3}$$

Once the given number of simulations has been reached, the algorithm returns the empirical policy $\hat{\pi}(s)$ for the current state s. An action is then sampled from $\hat{\pi}(s)$, and the game moves to the next state by playing that action. In this way, MCTS generates the players' states and actions alternately until the game ends with some outcome R after T steps, which gives an episode for the game. Each episode is defined as a sequence of tuples $(s_i, p_i, \hat{\pi}_i, v_i)$, where s_i is the game state at step i, p_i is the player at step i, $\hat{\pi}_i$ is the empirical policy generated at step i, and $v_i = (-1)^{|p_i - p_T|}R$ is the value signal from the outcome, which will become a contradictory signal once the prediction from the value network is faulty. After a given number of self-plays, all episodes will be stored into a replay buffer so that it can be used to train and update the value network V_θ (with all v_i's) and policy network π_θ (with all $\hat{\pi}_i$'s).

3 Methodology

3.1 Recursive-FOL

In this work, we use recursive first-order logic (recursive-FOL) for model checking a finite LTS. The recursive FOL is essentially an extension of FOL with fixpoint operators, which allows a predicate to be defined by referring back to itself. However, unlike L_μ, which defines properties functionally, recursive-FOL provides us more flexibility and allows us to describe model checking properties in a modular way so that a property can be defined with multiple sub-components, which is used later for deriving vector representations (see Sect. 3.2). To be specific, a recursive-FOL property can be defined by the following grammar:

$$\begin{aligned} \langle\text{property}\rangle &\models \langle\text{predicates}\rangle \\ \langle\text{predicates}\rangle &\models \langle\text{predicate}\rangle \mid \langle\text{predicate}\rangle \; ; \langle\text{predicates}\rangle \\ \langle\text{predicate}\rangle &\models \mathbf{LFP}.X(s) := \langle\text{fol-expr}\rangle \\ &\quad \mid \mathbf{GFP}.X(s) := \langle\text{fol-expr}\rangle \\ &\quad \mid X(s) := \langle\text{fol-expr}\rangle \\ \langle\text{fol-expr}\rangle &\models \mathbf{True} \mid \mathbf{False} \mid \varphi(s) \mid X(s) \mid \neg\langle\text{fol-expr}\rangle \\ &\quad \mid \langle\text{fol-expr}\rangle \vee \langle\text{fol-expr}\rangle \\ &\quad \mid \langle\text{fol-expr}\rangle \wedge \langle\text{fol-expr}\rangle \\ &\quad \mid \exists a \in A_s.\ X(s^a) \\ &\quad \mid \forall a \in A_s.\ X(s^a) \end{aligned}$$

where φ ranges over all primitive predicates, X ranges over the identifiers of the predicates, s is the LTS state variable for each predicate, A_s is the action space for the current state, and s^a is the successor state that $s \xrightarrow{a} s^a$.

The motivation of applying recursive-FOL to model checking comes directly from the MSOL interpretation of L_μ (see Sect. 2.1). However, different from MSOL, which implies fix-point operator intrinsically, adding a fix-point operator to FOL is tricky and error-prone [11]. Specifically, suppose we have a FOL predicate $\Phi(x; P)$, parameterized by a variable x and another predicate $P(x)$. To make sure an extended FOL formula (say $\mathbf{LFP}.\Phi(x; P)$) is well-formed, the function $F(P) = \{x|\Phi(x; P)\}$ must be monotone, which means either $F(P) \subseteq \{x|P(x)\}$ or $F(P) \supseteq \{x|P(x)\}$. It is to be noted that, in general, whether a FOL predicate $\Phi(x; P)$ is monotone is undecidable. Nevertheless, one can still construct monotone predicates by forcing each occurrence of P in $\Phi(x; P)$ to be positive [10]. In this work, we assume the user always defines a well-formed formula. This assumption comes from a practical consideration that any two-player extensive form game can be abstractly described as:

$$\mathbf{LFP}.\Phi(s, p; P) := Q(s, p) \vee \exists a \in A_s^p.\ \neg P(s^a, \bar{p}),$$

where s^a is the state following $s \xrightarrow{a} s^a$, $\bar{p}$ is the opposite player of p, $Q(s, p)$ means that the game ends at s and player p wins the game, and $\Phi(s, p; P)$ means that given the current game state s and player p, the current player p will eventually win the game. As a result, the predicate is defined by the formula and the underlying structure of the game states. In other words, if the game can generate infinitely many states, then the formula above is not well-formed.

Next, we show how to transform a L_μ formula to recursive-FOL. Since a modular approach is used to define a predicate, we need to first decompose a L_μ formula into different predicates. For example, suppose the given formula is

$$\nu Z.(p \vee \mu X.(q \vee [A]X)) \wedge \langle A\rangle Z, \tag{4}$$

we can rewrite it into two individual fix-point predicates, with a distinct state variable s as:

$$\begin{aligned}
&\mathbf{GFP}.Z(s) := (p[s] \vee X[s]) \wedge (\langle A\rangle Z)[s] \\
&\mathbf{LFP}.X(s) := q[s] \vee ([A]X)[s].
\end{aligned}$$

After decomposing every fix-point operator into individual predicates, we finish by transforming recursively with the following rules:

$$\begin{aligned}
&p[x] = p(x) \\
&(\Phi_1 \wedge \Phi_2)[x] = \Phi_1[x] \wedge \Phi_2[x] \\
&(\Phi_1 \vee \Phi_2)[x] = \Phi_1[x] \vee \Phi_2[x] \\
&(\neg\Phi)[x] = \neg\Phi[x] \\
&(\langle A\rangle\Phi)[x] = \exists a \in A_x.\ Z[x^a] \\
&([A]\Phi)[x] = \forall a \in A_x.\ Z[x^a],
\end{aligned} \tag{5}$$

where p means a primitive predicate that can always be evaluated, given the current state variable x. A_x is the set of all legal actions of an LTS at state x, and x^a represents the state such that $x \xrightarrow{a} x^a$.

The game semantics of recursive-FOL is almost the same as FOL's, except that the winning condition of the L_μ semantic game has been applied. However, it should be pointed out that since the variables of fix-point operators in L_μ have been transformed to unique fix-point predicates, there is no need to track bounded variables anymore. Consequently, the semantic game plays on a group of predicates and jumps from one to another if necessary.

3.2 State Representation

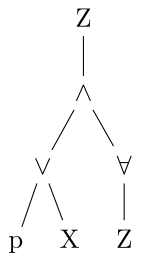
The state representation of any game state for a recursive-FOL semantic game is a vector $[i, p, \xi, \zeta]$, where i is an integer ID number for each predicate (in this case, $i(\mathrm{Z}) = 0, i(\mathrm{X}) = 1$), and $p \in \{-1, 1\}$ is the player ID. ξ and ζ are two vector components, where ξ is the vectorized representation of the current LTS state s, and ζ is an encoding of the action sequence on the syntax tree, which is initialized to all -1 for each predicate.

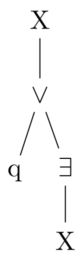
The entrance of a semantic game is always a predicate, which can be represented as a syntax tree. Once evaluated step by step, each node is either a logic operator or a predicate. A predicate indicates a leaf node for the current tree, but it also points to an entrance of another tree. We use a preorder traversal to identify each node and vectorize the action sequence on the tree structure, namely ζ.

For illustration, let's use the transformed recursive-FOL formula from Eq. 4, which contains two fix-point predicates:

$$\begin{aligned}&\mathbf{GFP}.Z(s) := (p(s) \vee X(s)) \wedge (\forall a \in A_s.\ Z(s^a))\\&\mathbf{LFP}.X(s) := q(s)s \vee (\exists a \in A_s.\ X(s^a)),\end{aligned}$$

where p and q are primitive predicates. The syntax trees of the two predicates can be drawn out as:





To better understand how to generate ζ properly, let's see the example above. The preorder indexes for each node in the two trees are:

$$\begin{aligned}&[Z : 0, \wedge : 1, \vee : 2, p : 3, X : 4, \forall : 5, Z : 6]\\&[X : 0, \vee : 1, q : 2, \exists : 3, X : 4],\end{aligned}$$

which means we can use a length-seven vector to completely encode any action sequences on these two trees during gameplay. All we need is to store the action taken on each node to the corresponding position in the vector. For instance, starting from some $Z(s)$, if a player took the left branch of the $\wedge$ operator at position 1, and then the other player took the right branch of the $\vee$ operator at position 2, then the vector ζ at leaf node X becomes:

$$[0, 0, 1, -1, -1, -1, -1].$$

Furthermore, if the game continued from predicate X, and one of the players picked the right branch of the $\vee$ operator at position 1, and then chose some move $m \in A_s$ on the $\exists$ operator at position 3, then the vector ζ at leaf node X becomes:

$$[0, 1, m, -1, -1, -1, -1].$$

3.3 MCTS with Fix-point Predicates

Applying neural MCTS to a recursive-FOL semantic game looks straightforward. However, it turns out to be non-trivial. Specifically, a semantic game might have an infinite game sequence composed of a set of states in a cycle. However, one of the players can still win the game as long as we can track the type of the leading fix-point operator (namely, the starting point of a cycle). On the other hand, the neural MCTS algorithm was not designed to handle an infinite game that uses looping states as a winning condition. As a result, we propose some modifications to the previous design to deal with this new situation.

Our method is motivated by the bounded game semantics on L_μ [12]. During the self-play and MCTS simulation, we maintain a stack $\mathfrak{L}$ and a counter $\mathfrak{C}$ to track the number of visits of each fix-point predicate along a game sequence. To be specific, for a given game state s, if s is the root node of some predicate's syntax tree, and also that predicate is a fix-point predicate, then we check if s is in the stack $\mathfrak{L}$. If it is already there, then we continuously pop from the stack the top state t and set $\mathfrak{C}[t] = 0$ until we hit s, then set $\mathfrak{C}[s] = \mathfrak{C}[s] + 1$; otherwise, we just push s to L and set $\mathfrak{C}[s] = 1$. After updating the stack, we check if $\mathfrak{C}[s] > \mathit{\Gamma}$ for some given integer bound $\mathit{\Gamma}$. s is considered to be a winning state for the Proponent/Opponent only if $\mathfrak{C}[s] > \mathit{\Gamma}$ and s is the root node of a **GFP**/**LFP** predicate.

After updating $\mathfrak{L}$ and $\mathfrak{C}$, we concatenate the visiting time of the predicate to the state representation of the corresponding state in the search tree. In other words, each tree node represents a tuple $(s, \mathfrak{C}[s.\mathbf{root}])$, where $s.\mathbf{root}$ is the root state of a predicate's syntax tree that state s is affiliated to. In this manner, neural MCTS no longer needs to deal with a potentially infinite game sequence, but it can still detect a cycle in the state space.

3.4 Fairness as a Challenge

Fairness is an essential concept in model-checking. Informally speaking, the fairness constraint requires that, in a multi-process system, each process should get an equal chance to run when it is able to run. This requirement is crucial, especially when searching for a counterexample of a liveness property. Since a model checking algorithm also decides how to schedule the running of each process, it is trivial for it to fabricate an unrealistic "counterexample" when ignoring the fairness constraint.

Classical model-checking algorithms solve this problem with a global method, which searches for all possible strongly connected components (SCCs) in the state space, then verify each component to see if it satisfies the fairness constraint. However, for a large LTS, the global method becomes intractable because of the state explosion issue.

With that being said, the main motivation to use neural MCTS in model-checking is its ability to handle large state space through a local search. We propose here a local approach to the fairness problem by maintaining a list of process access counter $\mathfrak{F}$. Therefore $\mathfrak{F}[p]$ means process with id number p has been accessed $\mathfrak{F}[p]$ times. The counter list $\mathfrak{F}$ is then be concatenated with state representation. During self-play and MCTS simulation, we check if, at the current state, $|\mathbf{max}(\mathfrak{F}) - \mathbf{min}(\mathfrak{F})| > K$ for some integer constant K. If so, then the current player loses the game immediately. It should be pointed out that $\mathfrak{F}$ needs to be normalized before it is used as an input to the neural network. Consequently, neural MCTS is forced to learn to access every process in a balanced way.

4 Experiments

4.1 Highest Safe Rung Problem

The Highest Safe Rung (HSR) problem is a well-known puzzle [26]. The problem can be described as follows:

Consider throwing jars from a specific rung of a ladder. The jars could either break or not. If a jar is unbroken after a trial, it can be used next time. The highest safe rung is the rung that the jar will break for any trial performed above it. Given three positive numbers $\boldsymbol{k}$, $\boldsymbol{q}$, *and* $\boldsymbol{n}$, *can we always be able to locate the highest safe rung on a* $\boldsymbol{n}$*-rung ladder with at most* $\boldsymbol{k}$ *jars and* $\boldsymbol{q}$ *trials? (assuming the jars are identical with each other).*

The above problem can actually be solved by playing an alternating reachability game [12] between two players, Alice and Bob. In the beginning, Alice claims that within q trials, she would be able to locate the highest safe rung on a n-rung ladder by using at most k jars, or noted as HSR(k, q, n). Alice first makes a move during the gameplay by selecting a rung m $(1 \leq m < n)$ and performing one trial on that rung. And Bob then decides whether the jar will break or not. If the jar is broken, Alice would only have to check rungs below rung m; otherwise, she needs to check rungs above m. As a result, Alice either claims HSR$(k-1, q-1, m)$ if Bob says "break", or HSR$(k, q-1, n-m)$ if Bob says "safe". The game will end if either Alice wins by claim something like HSR$(k, q, 1)$ where $(k \geq 0 \wedge q \geq 0)$, or Bob wins by forcing Alice to claim something like HSR(k, q, n) where $((k \leq 0 \vee q \leq 0) \wedge n > 1)$. The original HSR problem can be solved if and only if Alice has a winning strategy.

Finding a winning strategy can be regarded as a model checking problem to verify a reachability property on an LTS. The LTS can be generated by applying the above game rule to a given initial state (k, q, n) (see Fig. 2 for an example). The property can be described as ***starting from the initial state, Alice will eventually win the game***. We formulate this property with recursive-FOL predicates:

$$\mathbf{LFP}.X(s) := p(s) \vee \exists a \in A_s.\ Y(s^a)$$
$$Y(s) := q(s) \wedge \forall a \in A_s.\ X(s^a)$$
$$p(s) := s.n = 1$$
$$q(s) := s.n = 1 \vee (s.k > 0 \wedge s.q > 0)$$

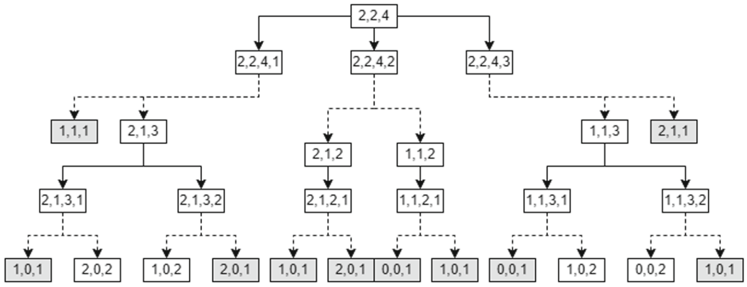


Fig. 2. The LTS for HSR$(2, 2, 4)$ where solid edges are actions taken by Alice, dashed edges are actions taken by Bob. The gray nodes are terminal states where Alice wins.

We carry out our experiment with HSR$(8, 8, 256)$ and HSR$(3, 8, 93)$. For each instance, we run five experiments. In each experiment, we record the number of Proponent's (Alice's) winning games during 100 self-plays. As the hyper-parameters, we set the number of MCTS simulations to 25, exploration coefficient α is 4. We use a four-layer multi-layer perceptron (MLP) network with shape $[256, 256, 256, 256]$ for the policy and value neural network. The neural network is trained with the Adam optimizer, with learning being 0.001. The mini-batch size is set to 64, and the training epoch is set to 10. We run the experiment until one of the players consistently wins during the self-play, indicating that a winning strategy has been learned against the other player. We executed these experiments with a Core i7-9750H 4.5 GHz CPU, 16 GB Memory, and a GTX 1650 GPU. It can be seen from the experimental result (Fig. 3) that the neural MCTS can learn a winning strategy for the Proponent in **25** iterations (each iteration takes 10 min on average). Besides, we have also verified the correctness of the learned strategy with the ground truth solution using the Bernoulli Triangle in [29].

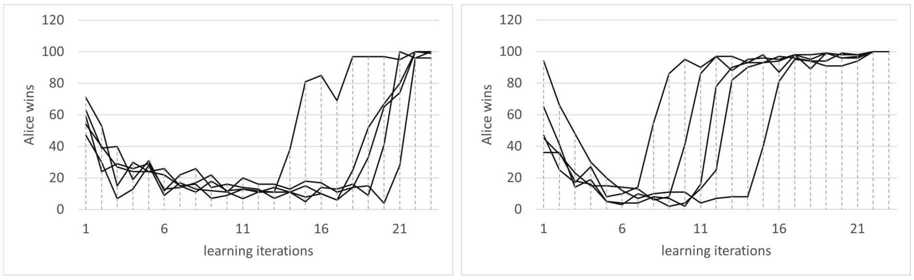


Fig. 3. Experimental results for HSR(8,8,256) (left) and HSR(3,8,93) (right). There are five trials. Each trial has 25 iterations. And each iteration contains 100 self-play. We show the wins of Alice (the Proponent) among 100 games in each iteration. It can be seen from the figure above that Alice has a U-shape learning curve in both cases, which indicates that the two players competed and learned from each other.

4.2 Dining Philosopher Problem

Our model for the dining philosopher problem is straightforward. N philosophers sit around a table with N forks among them, $N \geq 3$. At a philosopher's initial state, he can randomly choose the fork either on his right or left if another philosopher has not taken it. After taking the fork, he checks the availability of the fork on the other side. If unavailable, he concedes, releases the fork possessed, and returns to the initial state. If available, he picks it up and enters the eating state. After finishing his meal, he randomly releases one fork first, then releases the other fork before returning to the initial state. The model is parametric in the number of philosophers, where each philosopher model has ten states (Fig. 4). It is to be noted that we intentionally make our model imperfect so that when all philosophers follow this scheme, some of them may starve. We expect the model checking process to capture this design fallacy by showing us a counterexample.

In this experiment, we are interested in model-checking the property that ***if philosopher 0 is hungry, then eventually the philosopher will eat.*** This property can also be rewritten as the following recursive-FOL:

$$\begin{aligned}
&\mathbf{GFP}.Z(s) := (\neg p(s) \vee X(s)) \wedge \forall a \in A_s.\ Z(s^a)\\
&\mathbf{LFP}.X(s) := q(s) \vee \forall a \in A_s.\ X(s^a)\\
&p(s) := in\ s,\ \text{philosopher 0 is hungry.}\\
&q(s) := in\ s,\ \text{philosopher 0 is eating.}
\end{aligned}$$

The recursive-FOL expression is a bit complex. There are two fix-point predicates nested within each other. The first fix-point predicate $Z(s)$ means that ***starting from state*** s, ***it is always true that if philosopher 0 is hungry, he will eventually eat.*** While the second fix-point predicate $X(s)$ means that ***starting from state*** s, ***no matter how the system evolves, philosopher 0 will eventually eat.***

The experiment is conducted with eight instances, parameterized with N equal to 3 to 10. For each instance, we run five trials. We take down the number of

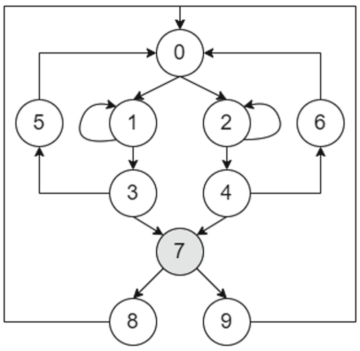


Fig. 4. The LTS for a single philosopher. State 0 is the initial state. State 1 and 2 are first picking attempts, either left or right. A philosopher will stay at one of these two states if he cannot obtain the first fork. After successfully picking his first fork, at state 3 or 4, he will try to pick the second fork. If he cannot obtain the second fork, he will go to either state 5 or 6 to release the fork picked. Otherwise, the philosopher can go to state 7, the eating state. After eating, he releases one of the forks and goes to either state 8 or 9, where he will release the other fork and go back to the initial state.

transitions required in each trial before finding a counterexample. As the hyper-parameters, we change the number of MCTS simulations to 5 and the exploration coefficient α to 1. The MLP networks with shape $[128, 128, 128, 64]$ are used for both the policy and value neural network. We run the experiment until the Opponent discovers a counterexample. It should also be noted that fairness is not negligible since we are dealing with a liveness property. We use the approach mentioned in Sect. 3.4 to add fairness constraints to our system, where we set K to be 50. The experimental results are listed in the table below (Table 3). Even though it takes time for neural MCTS to self-play and learn, the running time to find a counterexample is proportional to the number of transitions in the path, while each transition takes **50** ms on average. It can be seen that our results outperform the ones from a reinforcement learning (Q-learning to be precise) based method [5], which takes more running time but only finds a longer path. Moreover, we have also tested this problem with two off-the-shelf model checkers, SPIN [14], and PRISM [19]. Due to the state explosion issue, SPIN can only run up to N=7 on our computer (Table 4), while PRISM only runs up to N = 5 (since PRISM does not support generating of a counterexample for the CTL property in the form of A [G (“hungry” => F “eating”)]), we cannot show the path length in this case).

Table 3. Number of transitions required to find a counterexample for model-checking the given dining philosopher model. For each instance, we run 5 experiments. The number in the parentheses is the cycle lengths of the found counterexample.

	N = 3	N = 4	N = 5	N = 6	N = 7	N = 8	N = 9	N = 10
1	27 (14)	27 (17)	102 (54)	342 (271)	1046 (59)	838 (124)	3676 (553)	4742 (4405)
2	55 (48)	77 (63)	344 (309)	597 (547)	300 (154)	1217 (308)	2519(1980)	2727 (1226)
3	113 (39)	186 (175)	204 (37)	679 (509)	2248 (922)	1744 (164)	4664 (728)	2636 (1592)
4	76 (35)	98 (54)	579 (391)	344 (179)	976 (211)	1024 (831)	1956 (1017)	3480 (2350)
5	49 (37)	70 (50)	289 (268)	287 (168)	294 (60)	1218 (582)	1488 (1086)	3651 (1370)

Table 4. Model-checking the dining philosopher model with SPIN [14]. It can be seen that, even though SPIN tends to find shorter cycles, the running time increases exponentially because of the state explosion. As a result, SPIN can only model-check the problem up to N = 7.

	N = 3	N = 4	N = 5	N = 6	N = 7	N = 8
Length	2504 (18)	2165 (8)	2760 (28)	2959 (14)	9660 (12)	N/A
Time(s)	0.03	0.09	0.3	3.84	643	1.93E+03
States	5236	37302	113680	2.09E+06	2.03E+08	5.37E+08

5 Related Work

Model-checking through games was first proposed in [27], where the author applied a game-theoretical semantics to L_μ so that a model-checking problem is transformed into a two-player game. However, unlike our method, the author proposed to solve the game by a pure search algorithm with backtracking techniques. Another limitation to their method is that their system cannot handle fairness. It is to be noted that the work in [12], which is quite similar to the previous one, is more theoretical oriented rather than providing a concrete model checking algorithm. To our knowledge, we are the first work to apply modern gameplay AI to model-checking-problem-derived semantic games.

Applying machine learning to model checking for searching counterexamples has only been found in [2] and [5], both of which are reinforcement learning (Q-learning) based. They both use Büchi Automata to transform the model-checking problem into a graph search problem, which can be solved by reinforcement learning after formulating the graph search problem as a Markov Decision Process (MDP). Our method can treat as a complement to the study in this direction. However, unlike the Q-learning approach, which treats an on-the-fly model-checking task as a single MDP, we use a game-centric system that makes it possible to leverage the power of neural MCTS. We show that our method is superior to the approach from [5], but not the other one. However, [2] is only designed for liveness property, which allows it to encode state space efficiently, therefore mitigating the state explosion problem. Besides, we propose to use recursive-FOL as our model specification language, which is very close to L_μ. As a result, our method supports a more expressive specification than the ones in [5], which only supports LTL.

6 Conclusion

This paper highlights a likely promising approach for model checking systems with large state spaces. Our method is mainly based on two lines of development in computer science: the first one is from the formal methods and logic community, where we use the game-theoretical semantics of a logic to turn a logic expression into a two-player semantic game; the second one is from the AI and machine learning community, where we adapt the neural MCTS, a robust gameplay algorithm based on searching and learning, to play the semantic game derived from the logic specification. In this way, we can solve the classical model-checking problems by leveraging cutting-edge AI techniques. Besides, we propose recursive-FOL as our specification language, which is powerful in expressiveness. We also introduce a way to build fairness constraints in the game process. We compared our result with other model-checker tools and machine learning-based approaches and showed that it outperforms them.

In future work, we also plan to test our method on a more practical set of benchmarks, such as ones from the hardware model checking competition. We also work on improving the efficiency of Neural MCTS by using a meta-learning approach to build the neural network in incremental steps [16].

Finally, it is worth pointing out that, like other ML-based-model-checking methods, since the learned strategy might only win against a potentially sub-optimal strategy of the opponent, our method should only be applied to error-detection (i.e., finding counterexamples) instead of certifying the correctness. Although there are some limitations to our approach, the potential of the combination of search and learning is still considerable.

References

1. Anthony, T., Tian, Z., Barber, D.: Thinking fast and slow with deep learning and tree search. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. NIPS 2017, pp. 5366–5376 (2017)
2. Araragi, T., Cho, S.M.: Checking liveness properties of concurrent systems by reinforcement learning. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt 2006. LNCS (LNAI), vol. 4428, pp. 84–94. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74128-2_6
3. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Mach. Learn. **47**(2), 235–256 (2002)
4. Auger, D., Couëtoux, A., Teytaud, O.: Continuous upper confidence trees with polynomial exploration – consistency. In: Blockeel, H., Kersting, K., Nijssen, S., Železný, F. (eds.) ECML PKDD 2013. LNCS (LNAI), vol. 8188, pp. 194–209. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40988-2_13

5. Behjati, R., Sirjani, M., Nili Ahmadabadi, M.: Bounded rational search for on-the-fly model checking of LTL properties. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 292–307. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11623-0_17
6. Browne, C., et al.: A survey of monte Carlo tree search methods. IEEE Trans. Comput. Intellig. AI Games **4**(1), 1–43 (2012)
7. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Meyer, B., Nordio, M. (eds.) LASER 2011. LNCS, vol. 7682, pp. 1–30. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35746-6_1
8. Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. ACM Comput. Surv. **28**(4), 626–643 (1996)
9. Grill, J.B., et al.: Monte-Carlo Tree Search as Regularized Policy Optimization. arXiv:abs/2007.12509 (2020)
10. Gurevich, Y.: Toward logic tailored for computational complexity. In: Börger, E., Oberschelp, W., Richter, M.M., Schinzel, B., Thomas, W. (eds.) Computation and Proof Theory. LNM, vol. 1104, pp. 175–216. Springer, Heidelberg (1984). https://doi.org/10.1007/BFb0099486
11. Gurevich, Y., Shelah, S.: Fixed-point extensions of first-order logic. In: 26th Annual Symposium on Foundations of Computer Science (SFCS 1985), pp. 346–353 (1985)
12. Hella, L., Kuusisto, A., Rönnholm, R.: Bounded game-theoretic semantics for modal mu-calculus and some variants. In: Proceedings 11th International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2020, Brussels, Belgium, September 21–22, 2020. EPTCS, vol. 326, pp. 82–96 (2020)
13. Hintikka, J.: Game-theoretical semantics: insights and prospects. Notre Dame J. Formal Logic **23**(2), 219–241 (1982)
14. Holzmann, G.: The model checker SPIN. IEEE Trans. Software Eng. **23**(5), 279–295 (1997)
15. Janin, D., Walukiewicz, I.: On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In: Montanari, U., Sassone, V. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 263–277. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61604-7_60
16. Kadam, P., Xu, R., Lieberherr, K.J.: Dual Monte Carlo Tree Search. CoRR abs/2103.11517 (2021)
17. Kocsis, L., Szepesvári, C.: Bandit based monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006). https://doi.org/10.1007/11871842_29
18. Kolaitis, P.G.: On the expressive power of logics on finite models. In: Finite Model Theory and Its Applications. Texts in Theoretical Computer Science an EATCS Series, pp. 27–123. Springer, Heidelberg (2007). https://doi.org/10.1007/3-540-68804-8_2
19. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: a hybrid approach. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 52–66. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_5
20. Niwinski, D., Walukiewicz, I.: Games for the mu-Calculus. Theor. Comput. Sci. **163**(1&2), 99–116 (1996)
21. Rebuschi, M.: Extended game-theoretical semantics. In: Trobok, M., Miščević, N., Žarnić, B. (eds.) Between Logic and Reality. Logic, Epistemology, and the Unity of Science, vol. 25, pp. 161–182. Springer, Dordrecht (2012). https://doi.org/10.1007/978-94-007-2390-0_9

22. Schmid, M., et al.: Player of Games. CoRR abs/2112.03178 (2021)
23. Schrittwieser, J., et al.: Mastering Atari, go, chess and shogi by planning with a learned model. Nature **588**, 604–609 (2020)
24. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**, 484 (2016)
25. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., et al.: Mastering the game of go without human knowledge. Nature **550**, 354 (2017)
26. Sniedovich, M.: OR/MS games: 4. the joy of egg-dropping in Braunschweig and Hong Kong. Inf. Trans. Edu. **4**(1), 48–64 (2003)
27. Stevens, P., Stirling, C.: Practical model-checking using games. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 85–101. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054166
28. Walukiewicz, I.: Monadic second-order logic on tree-like structures. Theoret. Comput. Sci. **275**(1), 311–346 (2002)
29. Xu, R., Lieberherr, K.J.: Learning self-play agents for combinatorial optimization problems. Knowl. Eng. Rev. **35**, e11 (2020)