Towards Cognitive-Plausible Explanations for Board Game Agents with Genetic Programming

Manuel Eberhardinger[1,4,*], Florian Rupp[2], Florian Richoux[3], Johannes Maucher[1] and Setareh Maghsudi[4]

[1]Stuttgart Media University, Institute for Applied Artificial Intelligence, Stuttgart, Germany
[2]Technical University of Applied Sciences Mannheim, Mannheim, Germany
[3]AIST, Tokyo, Japan
[4]Ruhr-University Bochum, Bochum, Germany

Abstract

Recent advances in board game agents have led to superhuman performances in various games, but their decision-making processes remain opaque even to experienced players and are not cognitive-plausible from a human perspective. Based on criteria for cognitive-plausible systems, we introduce an approach that emphasizes interpretable knowledge representation through pattern-based and hierarchical structures. Our approach uses Genetic Programming to learn feature programs that capture spatial relationships on the game board. In addition, it utilizes decision trees to model agent behavior based on these feature programs. The result is a hierarchical, pattern-based model for generating post-hoc explanations for board game agents. We show that our method generates comprehensible explanations in *Tic-Tac-Toe*. We then discuss the current limitations in generalizing the approach to more complex games using *Connect 4* as an example.

Keywords

Explainable Reinforcement Learning, Genetic Programming, Board Games, Human-like Game Playing

1. Introduction

Although board game agents have demonstrated impressive superhuman playing strength in various domains [1, 2], they are not fully comprehensible in terms of why an action is taken, even for professional players. Furthermore, from a cognitive science perspective, the decision-making process of these agents does not resemble the way humans play games [3, 4]. Mándziuk proposes multiple "facets of cognitive-plausible playing systems" about knowledge acquisition and representation [4]. The method we propose addresses both postulates from Mańdziuk [4] concerning knowledge representation: (P1) "Game-related concepts may be effectively represented and processed with the use of pattern-based representation." (P2) "game-related knowledge should be represented in a hierarchichal structure with various inter- and intra-level connections." To that end, we present an approach for explaining board game agents by learning feature programs, i.e., programs that represent single or multiple grid cells and their relation to other cells with Genetic Programming. This is similar to P1 and the work about visual routines – a method to extract spatial information from scenes [5]. In addition, feature programs can represent intra-level connections of P2. By learning a decision tree on the feature activation of these programs, our method can highlight activated features for given state-action pairs, i.e., a hierarchical structure representing the inter-level connections of P2.

An important point worth discussing in advance is whether these postulates are necessary or sufficient for human comprehensibility and cognitive-plausibility. While pattern-based and hierarchical representations may be necessary because they correspond to human perception of recurring structures and the organization of knowledge, they are unlikely to be sufficient [6]. Human comprehensibility also depends on other aspects, such as the abstraction and combination of strategies [7], analogical

Joint AIIDE Workshop on Experimental Artificial Intelligence in Games and Intelligent Narrative Technologies, November 10-11, 2025, Edmonton, Canada.

*Corresponding author.

eberhardinger@hdm-stuttgart.de (M. Eberhardinger)
0009-0009-2897-9250 (M. Eberhardinger)

thinking [8], and the ability to convey explanations in intuitive terms. We will show in our approach that adherence to the postulates contributes to more interpretable models, but a broader spectrum of mechanisms is needed to better approximate cognitive-plausibility [9].

We show that the proposed method generates comprehensible explanations for *Tic-Tac-Toe*. In addition, we present preliminary results for *Connect 4* and discuss the limitations that need to be overcome to apply the method to more complex games.

Our contributions are:

- We present first steps towards a cognitive-plausible approach to explain board game agents.
- We show a proof of concept for generating explanations using the game *Tic-Tac-Toe* as an example.
- We conduct a case study which outlines the improvements necessary to apply the method on more complex games such as *Connect 4*.

The remainder of the paper is structured as follows: In Section 2 we review related work about genetic programming in the context of board games, as well as program synthesis approaches applied to explainability and interpretability in games. Section 3 introduces the domain-specific language and the representation of programs. The data collection process and methodology are described in Section 4. Subsequently, Section 5 presents and discusses the results of our case studies on *Tic-Tac-Toe* and *Connect 4*. Finally, Section 6 summarizes our contributions and outlines directions for future work.

2. Related Work

The majority of research that uses GP in the context of board games, search for heuristics for evaluating game states. These heuristics are evolved over mathematical functions for games such as Lose Checkers, Reversi, or Seven Wonders [10, 11, 12]. However, searching for heuristics is undesired, as a search algorithm is still necessary. Besides, using heuristics in a search algorithm is not fully explainable, and, in most cases, is cognitively implausible. Nevertheless, heuristics for evaluating game states, are more interpretable than using a combination of deep learning and search algorithms, similar to AlphaZero [1].

Silver et al. learn logical programs which are extracted from decision trees for simple, game-related tasks [13]. In [14], an inductive logic programming algorithm is introduced that finds comprehensible game strategies by providing the game rules as building blocks for logic programs for simple board games. Currently, our method does not use game rules, as they can be derived from the state-action pairs. However, our method is more difficult to learn without game rules.

The use of programs for explainability in the context of game AI was introduced for a maze-runner agent and two simplified Atari games: Space Invaders and Asterix [15]. In contrast to generating explanations, large language models are used to directly evolve programs as policies for a wide range of digital games, as well as programs as heuristic functions for board games [16].

The closest work to ours is by Soemers et al. [17], in which game tactics are extracted from linear models trained with self-play and converted into decision and regression trees. To be able to compare game traces collected from human participants with those collected by black-box agents in future work, we decided to generate post-hoc explanations from game traces without using the logits of trained policies, i.e., the raw output predictions of trained machine learning models before turning them into probabilities with the softmax function. Therefore, we only use state-action pairs collected from game agents without logits of a trained oracle.

3. Program, Domain-specific Language & Problem Domain

Programs are represented by a typed domain-specific language (DSL) which closely resembles the Lisp programming language [18]. The search space is constructed by using a uniformly distributed probabilistic grammar [19] shown in Table 1. Our DSL is inspired by the work about visual routines [5, 13], which uses a fixed set of basic operations to assemble different *perception* functions to reason

Table 1
The domain-specific language used. The type column shows one type for values and several types separated by an arrow for functions. The type after the final arrow indicates the function's return type. The types before the last arrow are the types of the input parameters.

Primitives / Values	Description	Type
0, 1, 2	integer values	int
$1	2D grid observation of the game board	board
X, O, Empty	possible objects on the board	object
stateObject	an object on the board with coordinates	so (stateObject)
if	standard if-clause	bool → func → func → func
eq-obj?	checks if two objects are equal	stateObject → object → bool
cell	cell object with x and y coordinate	int → int → cell
get	get a object on the board	board → cell → so
get-game-piece	get the object type of a stateObject	so → object
get-board-feature	get a feature and its activation for a given object	board → so → object → feature
not	negates a Boolean value	bool → bool
and	conjunction of two Boolean values	bool → bool → bool
or	disjunction of two Boolean values	bool → bool → bool

Listing 1 An example feature program generated from the DSL. $1 is the input state of the board game. This program is activated if the cells on position (2,0) and (1,0) are the same, so this feature could be used in combination with another feature to infer if position (0,0) lead to a win or lose of a game in *Tic-Tac-Toe.*

```
(get-board-feature
        $1
        (get  $1 (cell  2  0 ))
        (get-game-piece (get  $1 (cell  1  0 )))
)
```

about properties of shapes and spatial relations. Since our focus is on board games with a grid-like board and game pieces, the building blocks of our DSL consist of control flow production rules (if clauses and Boolean operators), integers, and methods to get pieces on the board and compare them. The primitive `get-board-feature(board, stateObject, object) -> feature`, gets one or multiple cells and checks if they are of the same object as the third parameter. If they are the same the feature program is activated. `stateObject` are always cells of the current state of the board game, while `object` is always a type provided by the game or environment. The board feature is later able to check if for a specific state this feature program is activated, so board features are actually independent from games and board states and, thus, transferable to other games in the future.

Listing 1 shows an example feature program generated from the DSL. $1 is the input state of the board game. This program is activated if the cells on position (2,0) and (1,0) are the same, so this feature could be used in combination with another feature to infer if position (0,0) lead to a win or lose of a game in *Tic-Tac-Toe*

Problem Domain To simplify the problem domain, we currently restrict ourselves to simple, traditional board games which are fully observable and turn-based two player games such as *Tic-Tac-Toe* and *Connect 4*. These games allow us to verify that the created explanations are also correct, since the decision making processes of these games are relatively simple compared to modern games, which depend on complex game mechanics.

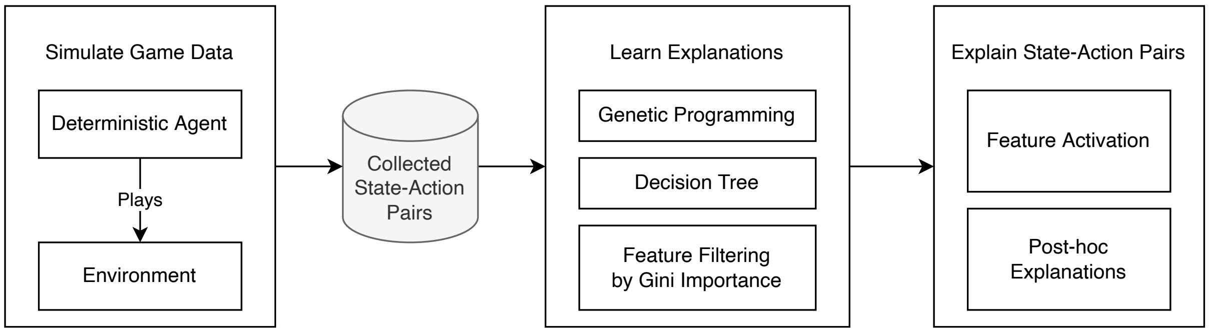


Figure 1: Overview of the proposed method: First, we collect gameplay data by using an deterministic agent that plays in an environment. Then, we use this data of collected state-action pairs to learn explanations with genetic programming in combination with a decision tree to select important features. Once the important features have been learned, the state-action pairs can be used to generate post-hoc explanations of the data by highlighting the activated features. Since the decision tree grows too large, we will only show selected paths in order to describe the results in Table 2.

4. Methodology

4.1. Overview

The goal of this work is to generate post-hoc explanations for black-box board game agents. Figure 1 shows an overview of the proposed approach. First, we collect gameplay data by using an agent that plays in an environment, here it is *Tic-Tac-Toe* or *Connect 4*. The collected state-action pairs are then used to learn explanations with genetic programming in combination with a decision tree to select important features. Once the important features have been learned, the state-action pairs can be used to generate post-hoc explanations of the data by highlighting the activated features. Since the decision tree grows too large, we will only show selected paths in order to describe the results in Table 2. In the following, we will explain the different parts of the method.

4.2. Deterministic Agent & Data Collection

To ensure a better validation, we restricted our method to deterministic minimax agents, since decision-making is not based on stochasticity, and is thus easier to interpret compared to MCTS agents or neural networks. It is also possible to use the agent to validate the methodology with deterministic data since the same action is always taken for a given state.

4.3. Genetic Programming for Learning Features

The implementation of our tree-based genetic programming algorithm is based on [20, 21]. To initialize the population, we randomly select trees and specify their return type. Then, we recursively sample their subtrees for each parameter. To perform a mutation, we randomly select a node and a tree and sample a new subtree for the node's return type. A one-point crossover is implemented, merging one tree with a random sub-tree of another tree that has the same return type. Tournament selection is used to select two candidates for crossover or one candidate for mutation from the population. For more details on the genetic programming algorithm, please refer to [20].

The fitness function maximizes the number of times features appear in the set of provided states S:

$$fitness = \begin{cases} \frac{\sum_{s \in S} \rho(s)}{|S|}, & \text{if } \rho \notin D_F \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

where ρ is the currently evolved feature program, which returns 1 if the feature is activated and 0 if the feature is not activated. We also use a regularizer to prevent the trees from growing too large. We

Algorithm 1 Overview of the whole approach.

```
procedure GetDecisionTree($S, Y_a, D_F$)
    $X \leftarrow \emptyset$
    for all $s \in S$ do
        $X_s \leftarrow \emptyset$                          ▷ set of all program activations for a single state
        for all $\rho \in D_F$ do
            $X_\rho \leftarrow \rho(s)$                     ▷ returns a program activation after execution
            $X_s \leftarrow X_s \cup X_\rho$
        end for
        $X \leftarrow X \cup X_s$
    end for
    return FitDecisionTree($X, Y_a$)
end procedure

procedure GetExplanations($S, Y_a, g, t$)
    $D_F \leftarrow$ GeneticProgramming($S, g, t$)
    $T_{D_F} \leftarrow$ GetDecisionTree($S, Y_a, D_F$)
    $\phi \leftarrow$ GetFeatureImportances($T_{D_F}$),    $\sum_{i=1}^{n} \phi_i = 1$
    $\tilde{D_F} \leftarrow$ FilterProgramsByImportance($\tilde{D_F}, \phi$) ,    $\tilde{D_F} \subseteq D_F$      ▷ $\phi_i > 0$
    $T_{\tilde{D_F}} \leftarrow$ GetDecisionTree($S, Y_a, \tilde{D_F}$)
    $\tilde{\rho} \leftarrow$ GetDecisionPath($T_{\tilde{D_F}}, S$),    $\tilde{\rho} \subseteq \tilde{D_F}$
    return GetNaturalLanguageExplanations($\tilde{\rho}$)         ▷ explanations
end procedure
```

implement a feature program database D_F that stores the features with the highest fitness every $t = 50$ generations for a total of g generations. This results in $|D_F| = \frac{g}{t}$ feature programs in D_F.

If an evolved feature program is already present in the feature database, then we always return 0 as the fitness. This ensures that these features will most likely not be selected for further evolution. We evolve the same population after saving the best features in the feature database so that we do not have to re-initialize the population. This allows the GP algorithm to run like an open-ended evolution, adding new features throughout all iterations. This is also possible thanks to the probabilistic grammar, which defines programs through recursion.

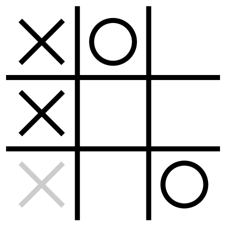
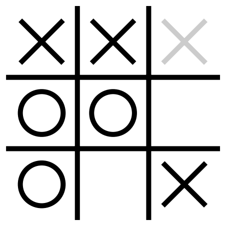
A drawback of counting how often features occur in the states is that programs such as `(get-board-feature $1 (get $1 (cell 1 0)) (get-game-piece (get $1 (cell 1 0))))` have maximum fitness because they compare the same cell. However, these programs do not contain any information for the decision-making process and are therefore filtered out in the next step.

4.4. From Features To Explanations

After collecting features for a predetermined number of generations t, we run all feature programs D_F on the given states. Next, we train a decision tree T_{D_F} using a standard stochastic greedy decision tree learner [22] that maps the output of the feature programs to the corresponding actions Y_a for each state in S. This decision tree represents the hierarchical structure of how features are connected. When we calculate the feature importance ϕ, i.e., the Gini importance, it becomes clear that only a small subset of the found features $\tilde{D_F} \subseteq D_F$ is actually used to map the activated features to the actions. Therefore, when training the decision tree, we filter the features and keep only those with an importance of $\phi > 0$. We train a second decision tree $T_{\tilde{D_F}}$ on the subset of $\tilde{D_F}$, to have a concise mapping of the nodes in the decision tree and the programs in $\tilde{D_F}$, since only the best subset $\tilde{D_F}$ of previously found programs is used.

We use the collected states as input to create post-hoc explanations and traverse the decision paths $\tilde{\rho}$, with $\tilde{\rho} \subseteq \tilde{D_F}$ of the decision tree to make the activated or non-activated feature programs visible. The

Table 2
We show the feature programs which were responsible for the decision making process of the agent for two game states. We also provide text explanations in natural language generated by GPT-4o and show whether the feature program was activated. In the upper example, the decision-making process checks if two game pieces of player X are in the first column and the first and second row. If this is the case, it will check if the third row of the first column is empty and if this is the case, the player plays the piece on this position (indicated by the gray X). In the next example, the branching of the decision path after the second program is visible. In both examples, the first two programs are checked, but as the second program is not activated for the second game state, other programs are checked afterwards. Instead of checking if the game can be won with completing the first column, the programs will check if the first row can be completed.

Feature Program	Natural Language Explanation	Activated	State
`(get-board-feature $1 (get $1 (cell 0 0)) Player)`	Is (0,0) the player's piece?	Yes	
`(get-board-feature $1 (get $1 (cell 1 0))(if (eq-obj? (get $1 (cell 1 0)) Empty) Empty Enemy ))`	Is (1,0) occupied by enemy or empty?	No	
`(get-board-feature $1 (get $1 (if (eq-obj? (get $1 (cell 2 0)) Enemy)(cell 1 0)(cell 2 0))) Empty)`	If (2,0) is occupied by enemy, check if (1,0) is empty. Otherwise check if (2,0) is empty?	Yes	
`(get-board-feature $1 (get $1 (cell 0 0 )) Player )`	Is (0,0) the player's piece?	Yes	
`(get-board-feature $1 (get $1 (cell 1 0))(if (eq-obj? (get $1 (cell 1 0)) Empty) Empty Enemy ))`	Is (1,0) occupied by enemy or empty?	Yes	
`(get-board-feature $1 (get $1 (cell 1 2))(if (eq-obj? (get $1 (cell 0 2)) Empty ) Empty (get-game-piece (get $1 (cell 1 2)))))`	If (0,2) is empty, treat (1,2) as Empty. Is (0,2) not empty?	Yes	
`(get-board-feature $1 (get $1 (if (eq-obj? (get $1 (cell 0 1)) Empty)(cell 0 1)(cell 2 1))) Empty)`	Is (0,1) empty? If not, is (2,1) empty? (checks if either is)	No	
`(get-board-feature $1 (get $1 (cell 0 2)) Empty)`	Is (0,2) empty?	Yes	

pseudo code of this procedure is given in Algorithm 1.

The following case studies provide examples of these types of explanations. Natural language explanations for these programs are afterwards created by GPT-4o [23].

5. Results & Discussion

5.1. Case Study: Tic-Tac-Toe Explanations

To demonstrate the effectiveness of the proposed approach, we selected *Tic-Tac-Toe* as the test environment because the simple game logic enables us to easily interpret the actions chosen by the minimax agent. When two minimax agents play against each other, the game always ends in a tie. Therefore, we collected data by having a random agent play against the minimax agent. In total, both agents played 100 games, which resulted in 319 state-action pairs. We only collected data after the minimax agent made a move with the X symbol because we only wanted to explain the minimax agent's decisions.

Next, we ran the genetic programming (GP) algorithm on the states for $g = 50,000$ generations, maintaining a population size of 200 and a tournament selection size of 20. Every $t = 50$ generations, the best feature program in the population was saved in the feature database. After the GP algorithm is completed, there are a total of $|D_F| = 1000$ feature programs in the database. After filtering the features by the calculated feature importance score of the decision tree, only $|\tilde{D_F}| = 21$ feature programs remain for explaining the decision-making process. Table 2 shows examples of the reasoning process, and provides an explanation of the programs in natural language, as well as a short concise description of the reasoning process in the caption.

5.2. Case Study: Connect 4

We also ran experiments on the game *Connect 4*. The data collection process is similar to that of *Tic-Tac-Toe*. However, we do not let a minimax agent play against a random agent, as most games end after the minimax agent has played four pieces. We let two minimax agents with different search depths play against each other in order to collect data which resulted in 439 state-action pairs. For the GP algorithm, we used $g = 100,000$ generations, maintaining the same population and tournament size throughout. This resulted in $|D_F| = 2000$ feature programs. After training the decision tree and filtering the programs by Gini importance, only $|\tilde{D_F}| = 379$ remained. We created post hoc explanations for a few example states, but the decision tree and path were too large and incomprehensible. Upon examining the important features, we noticed that many of them only considered one or two cells, which was sufficient for *Tic-Tac-Toe* since the game's complexity is much smaller. We identified multiple key issues that need improvement for learning features with genetic programming:

- Currently, the fitness function only counts how often a feature appears in states. To reduce the number of necessary features, the number of cells the feature looks at should also be maximized.
- The DSL should be adapted to include more high-level functions that are also plausible from a cognitive science viewpoint, such as counting pieces.
- Including the game rules in the DSL, could also lead to better features and runtime.

6. Conclusion

In this work, we introduced an approach and showed preliminary results for creating cognitive-plausible explanations of board game agents. Our approach is based on visual routines [5] and has a hierarchical, pattern-based structure that resembles a more human-intuitive way of playing games [3, 4], which we also discussed for sufficiency and necessity. We showed the branching of a decision path for explanations of two collected state-action pairs in the game *Tic-Tac-Toe* and outlined how to scale the approach to more complex games.

7. Acknowledgements

The research collaboration was facilitated by COST Action CA22145 - GameTable, supported by COST (European Cooperation in Science and Technology).

8. Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT in order to improve formulation, grammar and readability. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez et al., A general reinforcement learning algorithm that masters chess, shogi, and go through self-play, Science 362 (2018) 1140–1144.

[2] Meta Fundamental AI Research Diplomacy Team (FAIR), A. Bakhtin, N. Brown, E. Dinan, G. Farina, C. Flaherty et al., Human-level play in the game of Diplomacy by combining language models with strategic reasoning, Science 378 (2022). doi:10.1126/science.ade9097.

[3] J. Mańdziuk, Towards cognitively plausible game playing systems, IEEE Computational Intelligence Magazine 6 (2011). doi:10.1109/MCI.2011.940626.

[4] J. Mańdziuk, Human-like intuitive playing in board games, in: Neural Information Processing: 19th International Conference, ICONIP 2012, 2012, Proceedings, Part II 19, Springer, 2012, pp. 282–289.

[5] S. Ullman, Visual routines, Cognition 18 (1984) 97–159.

[6] M. C. Corballis, The recursive mind: The origins of human language, thought, and civilization-updated edition, Princeton University Press, 2014.

[7] D. Zhang, M. Thielscher, Representing and reasoning about game strategies, Journal of Philosophical Logic 44 (2015) 203–236.

[8] E. Hüllermeier, Towards analogy-based explanations in machine learning, in: International Conference on Modeling Decisions for Artificial Intelligence, Springer, 2020, pp. 205–217.

[9] J. Fürnkranz, T. Kliegr, H. Paulheim, On cognitive preferences and the plausibility of rule-based models, Machine Learning 109 (2020) 853–898.

[10] A. Benbassat, M. Sipper, Evolving lose-checkers players using genetic programming, in: Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, IEEE, 2010, pp. 30–37.

[11] A. Benbassat, M. Sipper, Evolving board-game players with genetic programming, in: Proc. of the 13th annual conference companion on Genetic and evolutionary computation, 2011, pp. 739–742.

[12] D. Robilliard, C. Fonlupt, Towards human-competitive game playing for complex board games with genetic programming, in: Artificial Evolution: 12th International Conference, Evolution Artificielle, EA 2015, Lyon, France, October 26-28, 2015. Revised Selected Papers 12, Springer, 2016, pp. 123–135.

[13] T. Silver, K. R. Allen, A. K. Lew, L. P. Kaelbling, J. Tenenbaum, Few-shot bayesian imitation learning with logical program policies, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 34, 2020, pp. 10251–10258.

[14] S. H. Muggleton, C. Hocquette, Machine discovery of comprehensible strategies for simple games using meta-interpretive learning, New Generation Computing 37 (2019) 203–217.

[15] M. Eberhardinger, J. Maucher, S. Maghsudi, Learning of generalizable and interpretable knowledge in grid-based reinforcement learning environments, in: Proc. of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, volume 19, 2023, pp. 203–214.

[16] M. Eberhardinger, J. Goodman, A. Dockhorn, D. Perez-Liebana, R. D. Gaina, D. Çakmak, S. Maghsudi, S. Lucas, From code to play: Benchmarking program search for games using large language models, IEEE Transactions on Games (2025) 1–13. doi:10.1109/TG.2025.3614499.

[17] D. J. N. J. Soemers, S. Samothrakis, É. Piette, M. Stephenson, Extracting tactics learned from self-play in general games, Information Sciences 624 (2023) 277–298. doi:10.1016/j.ins.2022.12.080.

[18] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part I, Communications of the ACM 3 (1960) 184–195. doi:10.1145/367177.367199.

[19] C. Manning, H. Schutze, Foundations of Statistical Natural Language Processing, MIT Press, 1999.

[20] M. Eberhardinger, F. Rupp, J. Maucher, S. Maghsudi, Unveiling the decision-making process in reinforcement learning with genetic programming, in: Advances in Swarm Intelligence, Springer Nature Singapore, Singapore, 2024, pp. 349–365.

[21] M. P. Johnson, P. Maes, T. Darrell, Evolving visual routines, Artificial Life 1 (1994) 373–389.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer,

R. Weiss, V. Dubourg, et al., Scikit-learn: Machine learning in python, the Journal of machine Learning research 12 (2011) 2825–2830.

[23] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford, et al., Gpt-4o system card, arXiv preprint arXiv:2410.21276 (2024).