Ludax: A GPU-Accelerated Description Language for Board Games

Graham Todd[◇], **Alexander G. Padula**[♠], **Dennis J.N.J. Soemers**[♣], **Sam Earle**[◇], **Julian Togelius**[◇]

Keywords: Games, hardware acceleration, description languages

Summary

Games have long been used as benchmarks and testing environments for research in artificial intelligence. An important step in supporting this research was the development of *game description languages*: frameworks that compile domain-specific code into playable and simulatable game environments, allowing researchers to generalize their algorithms and approaches across multiple games without having to manually implement each one. More recently, progress in reinforcement learning (RL) has been greatly aided by advances in *hardware acceleration*. Libraries like JAX allow practitioners to take full advantage of cutting-edge computing hardware, often speeding up training and testing by orders of magnitude. Here, we present a synthesis of these strands of research: a domain-specific language for board games which automatically compiles into hardware-accelerated code. Our framework, `Ludax`, combines the generality of game description languages with the speed of modern parallel processing hardware and is designed to fit neatly into existing deep learning pipelines. We envision `Ludax` as a tool to help accelerate games research generally, from RL to cognitive science, by enabling rapid simulation and providing a flexible representation scheme. We present a detailed breakdown of `Ludax`'s description language and technical notes on the compilation process, along with speed benchmarking and a demonstration of training RL agents.

Contribution(s)

1. We introduce `Ludax`, a new game description language for board games which automatically compiles into GPU-accelerated environments suitable for reinforcement learning
   Context: Inspired by previous separate efforts in both game description languages (e.g. `Ludii` (Piette et al., 2020)) and hardware acceleration for board games (e.g. `PGX` (Koyamada et al., 2023))
2. We demonstrate that `Ludax` achieves competitive throughput and training speed with handcrafted JAX environments while supporting a larger number of games and variants than existing libraries
   Context: We compare speeds to `PGX` for games which are supported by both systems and measure throughput on additional game exemplars not covered by any existing JAX implementations
3. We describe a general procedure for compiling description languages into JAX-compliant programs which could be adapted to other domains and problem settings
   Context: `Ludax` provides an alternative approach for compiling into JAX programs compared to extant description languages that exist for other domains (Chandra et al., 2025; Gimelfarb et al., 2024; Earle et al., 2025)

Ludax: A GPU-Accelerated Description Language for Board Games

Graham Todd[◇], **Alexander G. Padula**[♠], **Dennis J.N.J. Soemers**[♣], **Sam Earle**[◇], **Julian Togelius**[◇]
gdrtodd@nyu.edu

[◇]**New York University Tandon**
[♠]**ETH Zurich**
[♣]**Maastricht University**

Abstract

Games have long been used as benchmarks and testing environments for research in artificial intelligence. An important step in supporting this research was the development of *game description languages*: frameworks that compile domain-specific code into playable and simulatable game environments, allowing researchers to generalize their algorithms and approaches across multiple games without having to manually implement each one. More recently, progress in reinforcement learning (RL) has been greatly aided by advances in *hardware acceleration*. Libraries like JAX allow practitioners to take full advantage of cutting-edge computing hardware, often speeding up training and testing by orders of magnitude. Here, we present a synthesis of these strands of research: a domain-specific language for board games which automatically compiles into hardware-accelerated code. Our framework, `Ludax`, combines the generality of game description languages with the speed of modern parallel processing hardware and is designed to fit neatly into existing deep learning pipelines. We envision `Ludax` as a tool to help accelerate games research generally, from RL to cognitive science, by enabling rapid simulation and providing a flexible representation scheme. We present a detailed breakdown of `Ludax`'s description language and technical notes on the compilation process, along with speed benchmarking and a demonstration of training RL agents.

1 Introduction

For the past 75 years, games have served as vital tests and benchmarks for artificial intelligence research. While many specific games have been completely solved (Schaeffer et al., 2007) or optimized beyond the abilities of the strongest human players (Campbell et al., 2002; Silver et al., 2017), the general space of games remains a fertile ground for measuring improvements in reasoning, planning, and strategic thinking. A critical part of this progress, however, is the ability to test approaches and algorithms on a set of environments that are both diverse and computationally efficient.

To help drive further games and learning research, we introduce `Ludax`: a domain-specific language for board games that compiles into GPU-accelerated code written in the JAX library (Bradbury et al., 2018). `Ludax` draws on two main inspirations: (1) `Ludii` (Piette et al., 2020), a general purpose description language for board games already used to represent more than 1400 games from throughout history and around the world, and (2) `PGX` (Koyamada et al., 2023), a collection of optimized JAX-native implementations of classic board games and video games designed to facilitate rapid training and evaluation of modern reinforcement learning (RL) agents. `Ludax` presents a flexible and general-purpose game representation format that can be leveraged for efficient simulation and learning on modern computing hardware.

Ludax primarily supports two-player, perfect-information, turn-based board games played by placing, capturing, and moving pieces. This set of mechanics is broad enough to capture a wide range of existing games (e.g. *Connect Four*, *Hex*, *English Draughts*, ...) as well as many unexplored *novel* games and variants that fall within that class. Further, Ludax is designed to be easily expandable: like with Ludii, implementing new game mechanics in Ludax only requires implementing new atomic components in the underlying description language. These components can then be combined compositionally with existing elements of the language to produce an entirely new *range* of possible games, instead of each game needing to be implemented separately.

Another design goal for Ludax is ease of use, both in terms of game design and experimentation. The syntax of the description language is "ludemic" (Piette et al., 2020) – splitting game rules into clear sections governing the game's setup, play mechanics, and end conditions. Like with Ludii, game programs in Ludax resemble English descriptions of rules (see Figure 1). Further, by leveraging the structure of the existing PGX library, environments instantiated in Ludax can be easily combined with existing frameworks for GPU-accelerated search, reinforcement learning, or evolution (DeepMind et al., 2020; Tang et al., 2022). Ludax also supports a basic web interface for interactive debugging and potential user-studies.

Ludax is fundamentally a platform for accelerating board game research. In an era of increasingly complicated tasks and benchmarks, relatively simple board games may seem to be less interesting research domains (especially as many games have been more-or-less "solved" by modern methods). However, Ludax is not just a collection of new tasks. By decoupling rapid execution from the intensive process of writing new environment code, Ludax can power new research in a variety of directions. For instance, Ludax can be used to analyze RL generalization (Soemers et al., 2025) by defining a wide range of modifications for a target game task (akin to a platform like Minihack (Samvelyan et al., 2021)) or help improve studies of game generation by enabling the rapid evaluation of procedurally-generated rulesets (Todd et al., 2024; Collins et al., 2025). Finally, Ludax can help advance recent research into world modeling (Ying et al., 2025) by (1) providing a wide and easily-refreshable set of environments to test on efficiently and (2) allowing automated systems to propose and refine world models in these "novel games" by writing high-level and semantically-meaningful DSL code.

To our knowledge, Ludax is the first board game description language which compiles into GPU-accelerated code. In the following sections, we provide a detailed description of the language syntax, compilation process, and Ludax's expressive range. We also provide speed benchmarking compared to both Ludii and PGX, as well as an initial demonstration of training learned agents. Finally, we conclude with a discussion of potential use cases and future directions. Ludax is open-source under the Apache 2.0 license, and the code is available at: https://github.com/gdrtodd/ludax.

2 Related Work

Game Description Languages: Game description languages have been used for many years and in a variety of domains. The Stanford GDL (Love et al., 2008; Genesereth & Thielscher, 2014; Schiffel & Thielscher, 2014; Thielscher, 2017) is among the most influential, helping to popularize research in general game playing (Pitrat, 1968) through its use in the International General Game Playing Competition (Genesereth et al., 2005; Genesereth & Björnsson, 2013). Other notable examples include VGDL (Ebner et al., 2013; Schaul, 2013; 2014) (primarily known from its use in the General Video Game AI framework (Perez-Liebana et al., 2019)), RBG (Kowalski et al., 2019), Ludi (Browne, 2009), and its successor Ludii (Piette et al., 2020). GDLs have also been used to describe the rules of card games (Font et al., 2013) as well as to represent human goals in naturalistic simulated environments (Davidson et al., 2022; 2025). Modern game description languages have tended to move away from a basis in formal logic in favor of greater human usability, though there are benefits in efficiency gained by the use of regular languages (Kowalksi et al., 2020; Miernik et al., 2026).

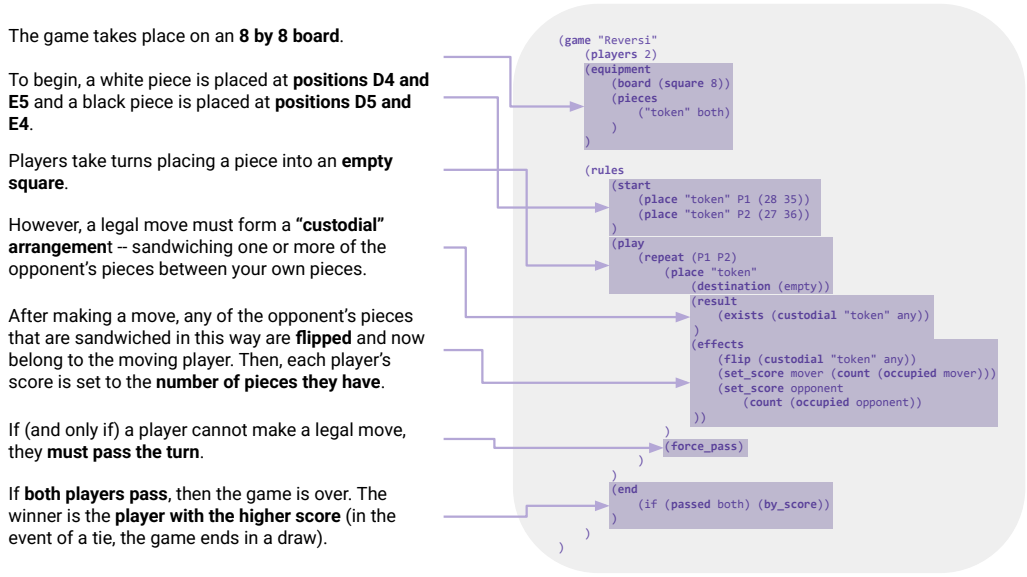

Figure 1: **Natural language description of *Reversi* along with its corresponding translation into `Ludax`**. `Ludax` uses "ludemic" syntax that represents high-level game components as separate program sections and aims to be easily interpretable to non-experts.

GPU-Accelerated Environments: Recent years have seen a proliferation of learning environments implemented in the JAX library or other frameworks that enable hardware (typically GPU) acceleration. Examples include single-agent and multi-agent physics simulators (Freeman et al., 2021; Makoviychuk et al., 2021; Bettini et al., 2022), ports of both classic and recent reinforcement learning tasks (Dalton & Frosio, 2020; Lange, 2022; Koyamada et al., 2023; Matthews et al., 2024; Suarez, 2025), combinatorial optimization problems (Bonnet et al., 2023), multi-agent coordination problems (Rutherford et al., 2023), and driving simulators (Gulino et al., 2023; Kazemkhani et al., 2025). While these efforts have spurred significant progress and span a wide range of domains and task formulations, each of them implement a fixed environment or set of environments. As such, they cannot easily be extended to novel environments without first writing new hardware-accelerated code. `Ludax` stands alongside a number of description languages for other domains (e.g. probabilistic programming, planning, single-player puzzles) that leverage JAX for efficient execution (Chandra et al., 2025; Gimelfarb et al., 2024; Earle & Togelius, 2025; Earle et al., 2025).

3 Description Language Details

`Ludax`'s game description language draws heavily on the `Ludii` description language, particularly in its use of "ludemic" syntax that represents game rules in terms of high-level and easily-understandable components (Piette et al., 2020). The complete grammar file and syntax details are available in the Supplemental Material.

3.1 Equipment and Start Rules

The `equipment` section contains information about the physical components used by the game. This includes the size and shape of the board (e.g. whether it is rectangular or hexagonal), the piece types used by players during the game, and optionally any "board regions" that are referred to by later rules (e.g. particular tiles that are blocked off from play). The `start` section is an optional section that contains the rules for the game's setup. For most games, play begins on an empty board and the `start` section is omitted. In some games, such as *Reversi* (see Figure 1), pieces are placed in a particular arrangement at the start of play.

3.2 Play Rules

Typically, the play rules of each game are the most involved, as they detail the core mechanics and dynamics of the game. The play section is itself broken into one or more subsections called "play phases." Each phase has its own rules for player actions and turn-taking, as well as specific conditions for when to transition to another phase. Most games have only a single phase in which players alternate turns until the game is over, though Ludax also supports games with more complicated dynamics.

The core of each phase is a "play mechanic" that encodes the ways that players take their turns. In the context of reinforcement learning, a play mechanic specifies both the action space ($\mathcal{A}$) and the transition function ($\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$). At a lower level, each play mechanic also defines a "legal action mask function" that returns whether each action is valid from the current game state. Ludax supports two kinds of play mechanics: place and move, the latter of which is broken down into step, hop, and slide. Each play mechanic takes arguments that define exactly how it is implemented. For instance, a place rule takes a destination constraint which can be used to restrict where pieces can be placed, as in *Connect Four*. Arguments to hop can be used to specify which pieces can be hopped over and in which directions, and the different movement types can be composed to create intricate rulesets like *English Draughts* (see Appendix A). Finally, a play mechanic might optionally define one or more effects that modify the game state after the action is performed. Effects are used to handle mechanics like capturing or flipping pieces, as well as updating each player's score if the game uses it – see *Reversi* again for an example.

High-level operators like place and slide eventually ground out to low-level components that encode individual game mechanics. These components, referred to as masks, functions, and predicates based on their return values, take in the current game state and compute a single, specific game property. For instance, the commonly-used line function returns the number of contiguous lines of a particular piece of a specified length and orientation while the custodial mask checks for the presence of a "custodial" arrangement in which a line of pieces belonging to one player are "sandwiched" by a pair of pieces belonging to the other player. Ludax implements a wide range of low-level mechanics, and crucially these mechanics can be combined compositionally using first-order logic (excluding quantification) to form more complicated expressions. As an example, the condition "*if a player makes a line of 4 stones in a row or a 2 by 2 square of stones...*" would be rendered as follows:

```
(or (line "stone" 4) (pattern "stone" (square 2)))
```

3.3 End Rules

The last section of a game description in Ludax details the criteria that terminate a game. The end section contains one or more "end conditions" which award a victory or loss to one to more players based on a binary predicate (e.g. forming a line or reaching a score threshold). A game can have multiple end conditions, which are then checked in order (so that win/loss conditions take precedence over draw checks, for instance).

3.4 Design Considerations

While Ludax draws heavily from the Ludii description language, there are some important differences which go beyond just changes in syntax. Take *Connect Four* as an example: Ludii aims to represent the game mechanically in a way that matches the natural language description of its rules as closely as possible. Accordingly, the canonical representation of *Connect Four* in Ludii features pieces that "Drop" into the "LastColumn" chosen by the player. Ludax represents the action space differently – players simply place a piece onto an empty board cell, with actions that are not directly above an existing piece or the bottom of the board marked as illegal. Mechanically, the two implementations of *Connect Four* are identical – the difference lies in how they are encoded (especially to simulated players or reinforcement learning agents). While Ludax, like Ludii, strives to

represent game descriptions intuitively, we primarily aim to provide a unified representation format across games so that general game-playing agents can more easily transfer knowledge and expertise from one game to another. As such, the size and shape of the action space for any game in `Ludax` is determined exclusively by the size of the board and the high-level play mechanics, regardless of how those mechanics might be translated into natural language. This choice is also partially motivated by the specifics of working with the JAX library (see Section 4) and has implications for benchmarking and downstream use-cases (see Section 6).

4 Compiling Game Descriptions into Game Environments

`Ludax` uses the `Lark` Python library to parse game descriptions into JAX environments, but the general approach is flexible enough to be used with different domains and parsing toolkits. Broadly speaking, `Ludax` operates by defining the leaves of the grammatical parse tree (i.e. individual `masks`, `functions`, and `predicates`) as atomic functions written in JAX, which are then dynamically composed from the bottom-up to form higher-level operators used by the environment class. Consider again the following game expression:

```
(or (line "stone" 4) (pattern "stone" (square 2)))
```

During compilation, the leaf-level nodes `(line ...)` and `(pattern ...)` are converted into JAX functions which map from the current game state to (in this case) a boolean truth value, and those functions are then passed up the parse tree. Higher-level nodes, such as `(and ...)`, receive the JAX functions corresponding to each of their children and return a *new* JAX function that also takes the game state as input and implements the appropriate operation (in this case, boolean conjunction). In pseudocode, this looks like the following:

```
def predicate_and(self, children):
   def predicate_fn(state):
      children_values = [child_fn(state) for child_fn in children]
      return all(children_values)

   return predicate_fn
```

In actuality, both the "children functions" and the combined "predicate function" must be written to be compatible with JAX's vectorization scheme and just-in-time (JIT) compilation. This imposes a number of implementational constraints, most notably that the size and shape of all arrays must be fixed at compile time. This means, for instance, that the dimensions of the "legal action mask" (and, hence, the size of the action space in general) cannot change as the game progresses and values like the number of iterations in a loop or the positions of a lookup mask must be pre-specified. Crucially, however, values that are determined during *parsing* (such as the number of children for a given node, or the value of any arguments) can be safely passed into compiled JAX functions as static constants. This fact is what allows `Ludax` to dynamically create JAX functions that nonetheless obey the constraints of vectorization and JIT compilation. At the top of the parse tree, these composed JAX functions are ultimately used to define the behaviors that appear in the environment's `step` function, such as applying the player's action to the board and handling move effects.

Precomputation: An important optimization for hardware accelerated environments is to express functions as batched matrix operations rather than iterative procedures. For instance, rather than checking for a line of pieces in *Tic-Tac-Toe* by starting at the position of the last move and scanning out in each direction (as `Ludii`'s implementation does), `PGX` hard-codes the set of board indices that correspond to each possible line of three in the game (i.e. `[[0, 1, 2], [0, 3, 6], ...]`) and performs a single multi-dimensional index into the board array. `Ludax` adopts and generalizes this approach: during parsing of `line`, for example, the line indices are computed with respect to the size and shape of the game board (i.e. rectangular, hexagonal, ...) as well as the length and orientation of the desired line (i.e. diagonal, vertical, ...). Again, because these values

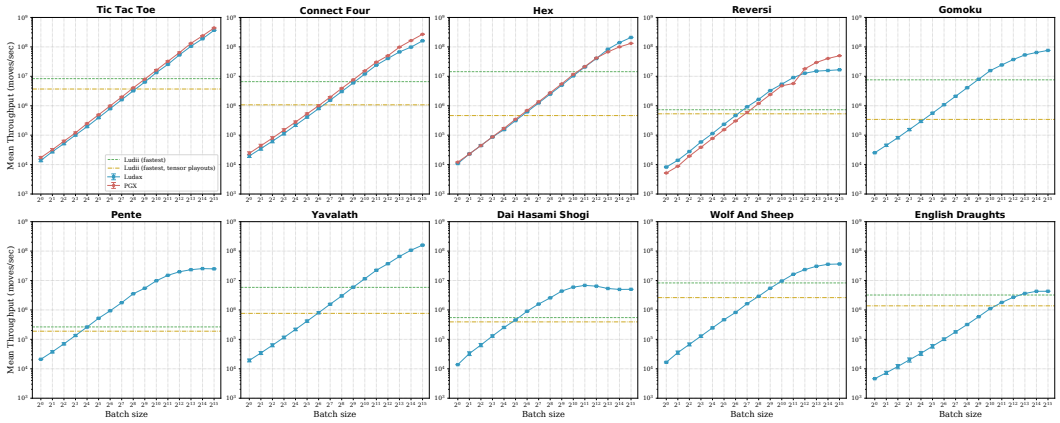

Figure 2: **Average throughput (moves per second) on various exemplar games for `Ludax`, `Ludii`, and `PGX`.** The first four games are implemented in all three frameworks, while the remaining games are implemented only in `Ludax` and `Ludii`. Speeds for `Ludax` and `PGX` are reported for 500 episodes of various batch sizes on a workstation with a single NVIDIA 4090 GPU and 36 threads on 18 CPU cores, while speeds for `Ludii` are the fasted recorded throughput for parallel execution on the same workstation across 1, 16, and 32 threads. Error bars are standard deviations over the 500 episodes.

depend only on attributes that are determined during parsing, they can be passed into JAX functions as constants and provide a substantial speedup when available. Precomputation naturally causes a trade-off between compile-time and run-time efficiency: we opt to use precomputation whenever possible, though some `masks` and `functions` cannot be expressed this way and must instead be evaluated at run-time.

Dynamic State Attributes: Different games require tracking different kinds of information about the current game state (such as scores or captured pieces). When `Ludax` compiles a game, it automatically extracts the attributes required to instantiate a game state and omits the others, thereby reducing the memory footprint of the entire state object. `Ludax` also automatically adds intermediary operations to each call of the environment's `step` function that help speed up later computations. For example, *Hex* requires finding the connected components of the board to determine whether opposite edges are linked by a player's pieces. Rather than re-compute the connected components from scratch after each move, `Ludax` again adopts an approach from `PGX` by detecting the presence of the `connected` function and maintaining a separate array of connected components that is locally updated after each move. This general procedure can be used to accommodate games with atypical or computationally expensive rules without affecting the runtime of other games.

5 Expressive Range

`Ludax` supports a class of two-player, perfect-information board games played by placing, capturing, and moving game pieces of different types (and can be adapated to support simple gridworlds – see Appendix D). Supported games range from the very simple (e.g. *Tic-Tac-Toe*) to the relatively complex (e.g. *English Draughts*, which features multiple kinds of pieces, forced captures, and sophisticated logic for granting extra turns). Existing games can also be easily modified, for instance by creating *misère* variants or modifying the size and shape of the board. More importantly, the low-level mechanics used to implement individual games can be combined compositionally to form a vast array of unique interactions and dynamics. While there are many board games `Ludax` does not support (some of which are implemented in alternate libraries like `PGX`), `Ludax` has the additional benefit of being a description language and not a static catalog of games, meaning it can be easily expanded with the introduction of new low-level mechanics.

6 Benchmarking

We benchmark the speed of Ludax on a set of 10 games, 4 of which are also implemented in both Ludii and PGX (allowing for a full comparison) and 6 of which are implemented only in Ludax and Ludii. Again, we emphasize that these 10 games are just *exemplars* of the class of games that Ludax supports, not an exhaustive list. A full description of each benchmark game is available in the Supplementary Material. We perform each of our benchmarking experiments on a workstation with a single NVIDIA 4090 GPU and an Intel i9-9980XE processor (36 threads on 18 CPU cores and 128GB of memory).

In Figure 2 we plot the throughput (in steps per second) under a uniformly random action policy for each game environment against the batch size (log scale on both axes), with the standard deviation of throughputs across episodes as error bars. Ludii supports parallelization via multi-threading: we report the fastest throughput on the same workstation when parallelized on 1, 16, and 32 threads. We note, however, that Ludii does not natively support reinforcement learning with fixed-size observation and action spaces (see Soemers et al. (2022a) for a detailed discussion). To give a more realistic comparison in the context of RL specifically, we additionally report the throughput for Ludii when it constructs a fixed-size tensor for each state and indexes into a fixed-size approximation of the action space for each legal action. Evaluations for Ludax and PGX were obtained by performing 100 warmup full-game episodes at the specified batch size, followed by measuring the speed over 500 episodes, with each evaluation taking at most a few minutes to complete. Evaluations for Ludii were obtained by running warmup episodes for 10 seconds, followed by measuring the speed over 30 seconds of episodes.[1] For games with potentially unbounded length (e.g. *Dai Hasami Shogi*), we terminate games for both Ludax and Ludii after 200 total turns.

Overall, Ludax achieves speeds that are competitive with state-of-the-art JAX environments optimized for a single game. PGX takes a slight edge in throughput at large batch sizes for *Connect Four* and *Reversi*, but Ludax remains within an order of magnitude (and even outspeeds PGX in *Hex*[2]). Ludax also outspeeds Ludii across all 10 games, achieving a maximum speedup of between ~3x (*English Draughts*) and ~450x (*Hex*) in the tensor playout regime. Some of this speedup is likely due to Ludax's comparatively smaller representation space, though Ludii also has optimized playout implementations tailored towards many of the categories of games covered by Ludax which might be difficult to apply in the context of deep learning (Soemers et al., 2022b).

7 Trained Agents

Finally, we demonstrate the feasibility of training reinforcement learning agents using the Ludax framework. We train our agent on the game *Reversi* (also known as *Othello*) using the AlphaZero-style (Silver et al., 2017) training script from the PGX library[3]. We use the same ResNetV2 (He et al., 2016) network architecture and training hyperparameters as PGX (full details available in the Supplementary Material) and train three separate runs on a single A100 GPU. Each run lasted roughly 57 million frames and took roughly three hours to complete.

We compare the performance of agents trained in the Ludax and PGX environments against the baseline *Reversi* agent provided by the PGX library in Figure 3. Evaluations were performed by playing two batches of 1024 games (one with the learned agent as the first player and one as the second player), with actions sampled from the normalized output of the policy head at each step. We see that both learned agents achieve remarkably similar performances against the baseline, with little to no differences in learning speed or stability. Although the PGX implementation of the *Reversi* environment is slightly more efficient, this translated into only marginal improvements in overall

[1] We opted to measure speed for Ludax and PGX using a fixed number of episodes because JAX's compilation procedure makes it difficult to halt execution after a specific elapsed wall time.

[2] Though Ludax's and PGX's implementations differ slightly with the latter including the "swap rule," this should not have a large effect on throughput under random playouts

[3] https://github.com/sotetsuk/pgx/blob/main/examples/alphazero/train.py (used under Apache 2.0 license)

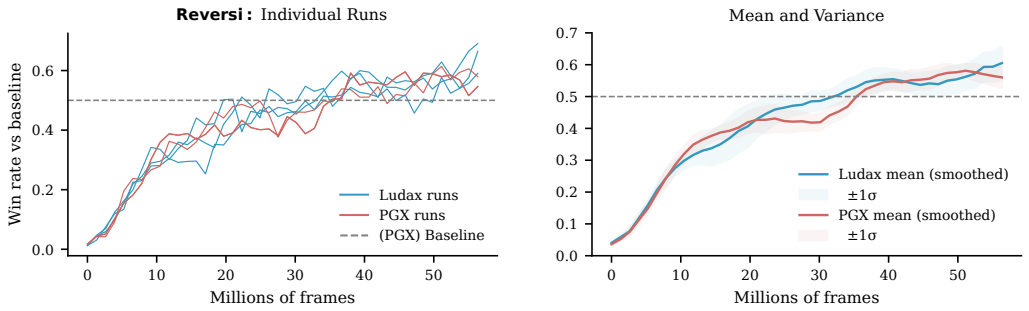

Figure 3: **Performance of reinforcement learning agents trained in the `Ludax` and `PGX` implementations of *Reversi* against the `PGX` baseline agent.** On the left, we plot the average winrate of the learned agents against the baseline over time and across three separate runs. On the right, we plot the average and variance of the winrates. Each run took roughly 3 hours to complete on a workstation with a single A100 GPU.

runtime (about 1.5%) owing to the shared overhead of network forward passes and weight updates. We also report training runs for Hex and Connect Four in Appendix F. Like `PGX`, `Ludax` offers a familiar API and an efficient set of environment implementations with which to train learning-based player agents.

8 Limitations and Future Work

While we aim to continually expand the range of games expressible in `Ludax`, it will likely never match the full generality of a language like `Ludii`. As such, other frameworks may be more appropriate for use-cases in which a broad range of games is more important than rapid simulation. `Ludax` also does not support genres other than board games (e.g. video games, card games, ...) – we leave the development of hardware accelerated description languages for such domains as an exciting area of future work. Compared to bespoke JAX implementations of board games (such as in the `PGX` library), environments in `Ludax` have slightly worse throughput – though the gap is marginal in a standard RL training setup. We deploy a number of optimizations to help close the efficiency gap when possible (see Section 4), but there are ultimately unavoidable trade-offs between speed and generality. For the purpose of training or benchmarking single-task agents on existing games, hard-coded simulators may remain the superior choice.

In light of these limitations, the most obvious avenues of extension for `Ludax` are the implementation and optimization of additional games and mechanics mechanics like irregular board shapes (e.g. *Tabu Y*) and mechanically distinct gameplay phases (e.g. *Nine-Men's Morris*). We also aim to provide a more robust visual interface for `Ludax`, both for the purpose of facilitating human-subject research (Carvalho et al., 2025) and the potential development of more "human-like" artificial agents which process the game board visually and select actions spatially.

Looking further ahead, we are excited about the potential application of `Ludax` to the study of *automatic game design*. Such systems depend on both a broad representation space and rapid evaluation of novel games – see Appendix G for a preliminary investigation of `Ludax`'s suitability for this kind of research. `Ludax` may also prove useful to research on *human behavior and play*, for instance by expanding and accelerating work that models human player choices (Zhang et al., 2024). Finally, `Ludax` offers an avenue to extend recent research in *general game playing* (e.g. with large language models (Schultz et al., 2024)) by providing a wide base of efficient game implementations that can in turn be leveraged for tree search algorithms or training world models.

9 Conclusions

We introduce a novel framework for games research that combines the generality of game description languages with the efficiency of modern hardware-accelerated learning environments. Our framework, `Ludax`, represents a broad class of two-player board games and compiles directly into code in the JAX Python library. Games in `Ludax` achieve speeds that are competitive with hand-crafted JAX implementations and faster than the widely-used `Ludii` game description language, and `Ludax` environments can easily be deployed in existing pipelines for deep reinforcement learning. Our framework helps widen and accelerate games research, with the potential to unlock new approaches in RL generalization, automatic game generation, and cognitive modeling.

References

M. Bettini, R. Kortvelesy, J. Blumenkamp, and A. Prorok. VMAS: A vectorized multi-agent simulator for collective robot learning. In *Proceedings of the 16th International Symposium on Distributed Autonomous Robotic Systems*, DARS '22. Springer, 2022.

Clément Bonnet, Daniel Luo, Donal Byrne, Shikha Surana, Vincent Coyette, Paul Duckworth, Laurence I Midgley, Tristan Kalloniatis, Sasha Abramowitz, Cemlyn N Waters, et al. Jumanji: a diverse suite of scalable reinforcement learning environments in jax. *CoRR*, 2023.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

C. B. Browne. *Automatic Generation and Evaluation of Recombination Games*. Phd thesis, Faculty of Information Technology, Queensland University of Technology, Queensland, Australia, 2009.

Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134 (1-2):57–83, 2002.

Wilka Carvalho, Vikram Srinivas Goddla, Ishaan Sinha, Hoon Shin, and Kunal Jha. Nicewebrl: a python library for human subject experiments with reinforcement learning environments. In *NeurIPS 2025 Workshop on Bridging Language, Agent, and World Models for Reasoning and Planning*, 2025.

Kartik Chandra, Tony Chen, Joshua B Tenenbaum, and Jonathan Ragan-Kelley. A domain-specific probabilistic programming language for reasoning about reasoning (or: a memo on memo). *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):784–814, 2025.

Katherine M Collins, Graham Todd, Cedegao E Zhang, Adrian Weller, Julian Togelius, Junyi Chu, Lionel Wong, Tom Griffiths, and Joshua B Tenenbaum. Generation and evaluation in the human invention process through the lens of game design. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 47, 2025.

S. Dalton and I. Frosio. Accelerating reinforcement learning through gpu atari emulation. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 19773–19782. Curran Associates, Inc., 2020.

Guy Davidson, Todd M Gureckis, and Brenden Lake. Creativity, compositionality, and common sense in human goal generation. In *Proceedings of the annual meeting of the cognitive science society*, volume 44, 2022.

Guy Davidson, Graham Todd, Julian Togelius, Todd M Gureckis, and Brenden M Lake. Goals as reward-producing programs. *Nature Machine Intelligence*, 7(2):205–220, 2025.

DeepMind, Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Antoine Dedieu, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, George Papamakarios, John Quan, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Laurent Sartran, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Miloš Stanojević, Wojciech Stokowiec, Luyu Wang, Guangyao Zhou, and Fabio Viola. The DeepMind JAX Ecosystem, 2020. URL http://github.com/deepmind.

Sam Earle and Julian Togelius. Autoverse: Evolving symbolic neural cellular automata environments to train player agents. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 175–178, 2025.

Sam Earle, Graham Todd, Yuchen Li, Ahmed Khalifa, Muhammad Umair Nasir, Zehua Jiang, Andrzej Banburski-Fahey, and Julian Togelius. Puzzlejax: A benchmark for reasoning and learning, 2025. URL https://arxiv.org/abs/2508.16821.

Marc Ebner, John Levine, Simon M Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. Towards a video game description language. 2013.

Jose M Font, Tobias Mahlmann, Daniel Manrique, and Julian Togelius. A card game description language. In *Applications of Evolutionary Computation: 16th European Conference, EvoApplications 2013, Vienna, Austria, April 3-5, 2013. Proceedings 16*, pp. 254–263. Springer, 2013.

C Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax-a differentiable physics engine for large scale rigid body simulation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

M. R. Genesereth and Y. Björnsson. The international general game playing competition. *AI Magazine*, 34(2):107–111, 2013.

Michael Genesereth and Michael Thielscher. *General Game Playing*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2014.

Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005. URL http://www.aaai.org/ojs/index.php/aimagazine/article/view/1813.

Michael Gimelfarb, Ayal Taitler, and Scott Sanner. Jaxplan and gurobiplan: Optimization baselines for replanning in discrete and mixed discrete and continuous probabilistic domains. In *34th International Conference on Automated Planning and Scheduling*, 2024. URL https://openreview.net/forum?id=7IKtmUpLEH.

C. Gulino, J. Fu, W. Luo, G. Tucker, E. Bronstein, Y. Lu, J. Harb, X. Pan, Y. Wang, X. Chen, J. D. Co-Reyes, R. Agarwal, R. Roelofs, Y. Lu, N. Montali, P. Mougin, Z. Yang, White B, A. Faust, R. McAllister, D. Anguelov, and B. Sapp. Waymax: An accelerated, data-driven simulator for large-scale autonomous driving research. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2023.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (eds.), *Computer Vision – ECCV 2016*, pp. 630–645, Cham, 2016. Springer International Publishing.

Saman Kazemkhani, Aarav Pandya, Daphne Cornelisse, Brennan Shacklett, and Eugene Vinitsky. Gpudrive: Data-driven, multi-agent driving simulation at 1 million fps. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2408.01584.

L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou (eds.), *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pp. 282–293. Springer, Berlin, Heidelberg, 2006.

J. Kowalksi, R. Miernik, M. Mika, W. Pawlik, J. Sutowicz, M. Szykuła, and A. Tkaczyk. Efficient reasoning in regular boardgames. In *Proceedings of the 2020 IEEE Conference on Games*, pp. 455–462. IEEE, 2020.

Jakub Kowalski, Mika Maksymilian, Jakub Sutowicz, and Marek Szykuła. Regular boardgames. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, volume 33, pp. 1699–1706. AAAI Press, 2019.

Sotetsu Koyamada, Shinri Okano, Soichiro Nishimori, Yu Murata, Keigo Habara, Haruka Kita, and Shin Ishii. Pgx: Hardware-accelerated parallel game simulators for reinforcement learning. *Advances in Neural Information Processing Systems*, 36:45716–45743, 2023.

Robert Tjarko Lange. gymnax: A JAX-based reinforcement learning environment library, 2022. URL http://github.com/RobertTLange/gymnax.

N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General game playing: Game description language specification. Technical Report LG-2006-01, Stanford Logic Group, 2008.

V. Makoviychuk, L. Wawrzyniak, Y. Guo, M. Lu, K. Storey, M. Macklin, D. Hoeller, N. Rudin, A. Allshire, A. Handa, and G. State. Isaac gym: High performance GPU based physics simulation for robot learning. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1. Curran, 2021.

Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Jackson, Samuel Coward, and Jakob Foerster. Craftax: A lightning-fast benchmark for open-ended reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2024.

Radosław Miernik, Marek Szykuła, Jakub Kowalski, Jakub Cieśluk, Łukasz Galas, and Wojciech Pawlik. Regular games–an automata-based general game playing language. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29600–29607, 2026.

Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D. Gaina, Julian Togelius, and Simon M. Lucas. General video game AI: A multitrack framework for evaluating agents, games, and content generation algorithms. *IEEE Transactions on Games*, 11(3):195–214, 2019.

É. Piette, D. J. N. J. Soemers, M. Stephenson, C. F. Sironi, M. H. M. Winands, and C. Browne. Ludii – the ludemic general game system. In G. De Giacomo, A. Catala, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang (eds.), *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pp. 411–418. IOS Press, 2020.

Jacques Pitrat. Realization of a general game-playing program. In *IFIP Congress (2)*, pp. 1570–1574, 1968.

Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Garðar Ingvarsson, Timon Willi, Akbir Khan, Christian Schröder de Witt, Alexandra Souly, et al. Jaxmarl: Multi-agent rl environments in jax. *CoRR*, 2023.

M. Samvelyan, R. Kirk, V. Kurin, J. Parker-Holder, M. Jiang, E. Hambro, F. Petroni, H. Küttler, E. Grefenstette, and T. Rocktäschel. Minihack the planet: A sandbox for open-ended reinforcement learning research. In *Advances in Neural Information Processing Systems*, 2021.

Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

T. Schaul. A video game description language for model-based or interactive learning. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*, pp. 193–200. IEEE, 2013.

Tom Schaul. An extensible description language for video games. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):325–331, December 2014. ISSN 1943-068X. DOI: 10.1109/TCIAIG.2014.2352795.

Stephan Schiffel and Michael Thielscher. Representing and reasoning about the rules of general games with imperfect information. *Journal of Artificial Intelligence Research*, 49:171–206, 2014.

John Schultz, Jakub Adamek, Matej Jusup, Marc Lanctot, Michael Kaisers, Sarah Perrin, Daniel Hennes, Jeremy Shar, Cannada Lewis, Anian Ruoss, et al. Mastering board games by external and internal planning with language models. *arXiv preprint arXiv:2412.12119*, 2024.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. DOI: 10.1126/science.aar6404. URL https://www.science.org/doi/abs/10.1126/science.aar6404.

D. J. N. J. Soemers, V. Mella, C. Browne, and O. Teytaud. Deep learning for general game playing with Ludii and Polygames. *ICGA Journal*, 43(3):146–161, 2022a.

D. J. N. J. Soemers, É. Piette, M. Stephenson, and C. Browne. Optimised playout implementations for the Ludii general game system. In C. Browne, A. Kishimoto, and J. Schaeffer (eds.), *Advances in Computers Games (ACG 2021)*, volume 13262 of *Lecture Notes in Computer Science*, pp. 223–234. Springer, Cham, 2022b.

Dennis JNJ Soemers, Spyridon Samothrakis, Kurt Driessens, and Mark Winands. Environment descriptions for usability and generalisation in reinforcement learning. In *Proceedings of the 17th International Conference on Agents and Artificial Intelligence*, pp. 983–992, 2025.

Joseph Suarez. Pufferlib 2.0: Reinforcement learning at 1m steps/s. In *Reinforcement Learning Conference*, 2025.

Yujin Tang, Yingtao Tian, and David Ha. Evojax: Hardware-accelerated neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 308–311, 2022.

M. Thielscher. GDL-III: A description language for epistemic general game playing. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 1276–1282, 2017.

Graham Todd, Alexander G Padula, Matthew Stephenson, Éric Piette, Dennis Soemers, and Julian Togelius. Gavel: Generating games via evolution and language models. *Advances in Neural Information Processing Systems*, 37:110723–110745, 2024.

Lance Ying, Katherine M Collins, Prafull Sharma, Cedric Colas, Kaiya Ivy Zhao, Adrian Weller, Zenna Tavares, Phillip Isola, Samuel J Gershman, Jacob D Andreas, et al. Assessing adaptive world models in machines with novel games. *arXiv preprint arXiv:2507.12821*, 2025.

Cedegao E Zhang, Katherine M Collins, Lionel Wong, Adrian Weller, and Josh Tenenbaum. People use fast, goal-directed simulation to reason about novel games. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 46, 2024.

```
(game "Reversi"
    (players 2)
    (equipment
        (board (square 8))
        (pieces
            ("token" both)
        )
    )

    (rules
        (start
            (place "token" P1 (28 35))
            (place "token" P2 (27 36))
        )
        (play
            (repeat (P1 P2)
                (place "token"
                    (destination (empty))
                    (result
                        (exists (custodial "token" any))
                    )
                    (effects
                        (flip (custodial "token" any))
                        (set_score mover (count (occupied mover)))
                        (set_score opponent
                            (count (occupied opponent))
                        ))
                )
                (force_pass)
            )
        )
        (end
            (if (passed both) (by_score))
        )
    )
)
```

```
(game "ConnectFour"
    (players 2)
    (equipment
        (board (rectangle 6 7))
        (pieces
            ("token" both)
        )
    )

    (rules
        (play
            (repeat (P1 P2)
                (place "token"
                    (destination (and (empty)
                        (or
                            (edge bottom)
                            (adjacent (occupied) direction:up)
                        )
                    ))
                )
            )
        )

        (end
            (if (line "token" 4) (mover win))
            (if (full_board) (draw))
        )
    )
)
```

```
(game "Yavalath"
    (players 2)
    (equipment
        (board (hexagon 9))
        (pieces
            ("token" both)
        )
    )

    (rules
        (play
            (repeat (P1 P2)
                (place "token" (destination (empty)))
            )
        )
        (end
            (if (line "token" 4) (mover win))
            (if (line "token" 3) (mover lose))
        )
    )
)
```

```
(game "WolfAndSheep"
    (players 2
        (set_forward (P1 down) (P2 up))
    )
    (equipment
        (board (square 8))
        (pieces ("wolf" P1) ("sheep" P2))
    )

    (rules
        (start
            (place "wolf" P1 (3))
            (place "sheep" P2 (56 58 60 62))
        )
        (play
            (repeat (P1 P2)
                (move
                    (or
                        (step "wolf" direction:diagonal)
                        (step "sheep" direction:
                            (forward_left forward_right)
                        )
                    )
                )
            )
        )

        (end
            (if (and
                (exists (and (occupied mover) (edge forward)))
                (mover_is P1))
            (mover win))
            (if (no_legal_actions) (mover win))
        )
    )
)
```

Figure 4: **Ludax syntax for *Reversi*, *Connect Four* (classic board games), *Yavalath* and *Wolf and Sheep*.** Ludax supports a wide range of games of vary complexity.

A Example Games and Syntax

In Figure 4 we present the Ludax syntax for a small set of exemplar games (*Reversi*, *Connect Four*, *Yavalath*, and *Wolf and Sheep*). Below, we present an annotated version of *English Draughts* to help illustrate aspects of Ludax's syntax and structure.

```
(game "English Draughts"

   // We define the "forward" direction for each player to
      simplify the movement logic
   (players 2
      (set_forward (P1 up) (P2 down))
```

```
)

// The game takes place on an 8-by-8 boardand uses two kinds of
    pieces
(equipment
   (board (square 8))
   (pieces
      ("pawn" both)
      ("king" both)
   )
)

(rules

   // To begin, we place pawns for each player in a checker
      board pattern along the first and last two rows
   (start
      (place  "pawn" P1 (40 42 44 46 49 51 53 55 56 58 60 62))
      (place  "pawn" P2 (1 3 5 7 8 10 12 14 17 19 21 23))
   )

   (play
      // Players alternate making moves
      (repeat (P1  P2 )

         // A move consists of one of the following options:
         (move
            (or

               // A pawn can hop over opposing pieces in either
                    of the "forward" diagonals to capture them.
                    This option has priority over non-capture
                   moves
               (hop "pawn" direction:(forward_left
                  forward_right) hop_over:opponent
                  capture:true priority:0)

               // A pawn can step into unoccupied squares in
                  the "forward" diagonals
               (step "pawn" direction:(forward_left
                  forward_right) priority:1)

               // A king hops and captures in any diagonal
                  direction...
               (hop "king" direction:diagonal hop_over:opponent
                    capture:true priority:0)

               // ...and steps the same way
               (step "king" direction:diagonal priority:1)
            )

            (effects
               // After moving, pawns on the forward edge of
                  the boardare promoted into kings
```

```
                    (promote "pawn" "king" (edge forward))

                    // If the player made a hop move (i.e. captured
                       a piece) and the same piece could hop and
                       capture again, the player gets an extra turn
                        using that piece
                    (if (and (action_was mover hop) (can_move_again
                       hop))
                       (extra_turn mover same_piece:true)
                    )
                )
            )
         )
      )

      // If making a move results in the next player having no
         legal moves, then the most recent mover wins
      (end
         (if (no_legal_actions) (mover  win))
      )
   )

   // For rendering, player one has the black pieces and player
      two has the white pieces
   (rendering
      (color P1 black)
      (color P2 white)
   )
)
```

B Ludax Grammar

Below we present the complete grammar specification for Ludax, using the syntax of the Lark Python library (raw string constants omitted for brevity).

```
// ---Root---
game: "(" "game" name players equipment rules rendering? ")"

// ---Players---
players: "(" "players" positive_int forward_assignments? ")"
forward_assignments: "(" "set_forward" p1_assignment p2_assignment
    ")"
p1_assignment: "(" P1(UP | DOWN | LEFT | RIGHT) ")"
p2_assignment: "(" P2(UP | DOWN | LEFT | RIGHT) ")"

// ---Equipment---
equipment: "(" "equipment" boardpieces regions? ")"
board: "(" "board" (board_square | board_rectangle | board_hexagon
    | board_hex_rectangle) ")"
?board_square: square_shape
?board_rectangle: rectangle_shape
?board_hexagon: hexagon_shape
```

```
?board_hex_rectangle: hex_rectangle_shape

// ---Pieces---
pieces: "(" "pieces" piece_definition+ ")"
piece_definition: "(" name (player_reference | BOTH) ")"
piece_reference: name

// ---Regions---
regions: "(" "regions" region_definition+ ")"
region_definition: "(" name (indices_arg | multi_mask_arg) ")"
region_reference: name

// ---Rules---
rules: "(" "rules" start_rules? play_rules end_rules ")"

// ---Start rules---
start_rules: "(" "start" start_rule+ ")"
start_rule: start_place
start_place: "(" "place" piece_reference player_reference (
   indices_arg | multi_mask_arg) ")"

// ---Play rules---
play_rules: "(" "play" play_phase+ ")"
play_phase: phase_once_through | phase_repeat
phase_once_through: "(" "once_through" play_mover_order
   play_super_mechanic ")"
phase_repeat: "(" "repeat" play_mover_order play_super_mechanic ")
   "
play_mover_order: "(" player_reference+ ")"

play_super_mechanic: play_mechanic force_pass?
play_mechanic: play_place | play_move
force_pass: "(" "force_pass" ")"

// ---Place rules---
play_place: "(" "place" piece_reference mover_reference? place
   _destination_constraint place_result_constraint? play_effects?
    ")"
place_destination_constraint: "(" "destination" super_mask ")"
place_result_constraint: "(" "result" super_predicate ")"

// ---Move rules---
play_move: "(" "move" move_type_definition play_effects? ")"
move_type_definition: move_type | "(" "or" move_type+ ")"
move_type: move_hop
       | move_slide
       | move_step

move_hop: "(" "hop" piece_reference direction_arg? piece_arg?
   hop_over_arg? capture_arg? priority_arg? ")"
move_slide: "(" "slide" piece_reference direction_arg?
   distance_arg? priority_arg? ")"
move_step: "(" "step" piece_reference direction_arg? priority_arg?
    ")"
```

```
// ---Effects---
play_effects: "(" "effects" play_super_effect+ ")"
play_super_effect: play_effect | play_conditional_effect
play_conditional_effect: play_if_effect | play_if_else_effect
play_if_effect: "(" "if" super_predicate play_effect ")"
play_if_else_effect: "(" "if" super_predicate play_effect "else"
   play_effect ")"

play_effect: effect_capture
        | effect_extra_turn
        | effect_flip
        | effect_increment_score
        | effect_promote
        | effect_set_score

effect_capture: "(" "capture" super_mask mover_arg? increment_
   score_arg? ")"
effect_extra_turn: "(" "extra_turn" mover_reference
   same_piece_arg? ")"
effect_flip: "(" "flip" super_mask mover_arg? ")"
effect_increment_score: "(" "increment_score" mover_reference
   function ")"
effect_promote: "(" "promote" piece_reference piece_reference
   super_mask mover_arg? ")"
effect_set_score: "(" "set_score" mover_reference function ")"

// ---Functions---
function: function_add
      | function_connected
      | function_constant
      | function_count
      | function_line
      | function_multiply
      | function_pattern
      | function_score
      | function_subtract

function_add: "(" "add" function+ ")"
function_connected: "(" "connected" piece_reference multi_mask_arg
    mover_arg? direction_arg? ")"
function_constant: positive_int
function_count: "(" "count" super_mask ")"
function_line: "(" "line" piece_reference positive_int
   orientation_arg? exact_arg? player_arg? exclude_arg? ")"
function_multiply: "(" "multiply" function+ ")"
function_pattern: "(" "pattern" piece_reference (pattern_arg |
   shape_arg) rotate_arg? player_arg? exclude_arg? ")"
function_score: "(" "score" mover_reference ")"
function_subtract: "(" "subtract" function function ")"

// ---End rules---
end_rules: "(" "end" end_rule+ ")"
end_rule: "(" "if" super_predicate end_rule_result ")"
```

```
?end_rule_result: result_win | result_lose | result_draw |
    result_by_score

// -- Result definitions --
result_win: "(" (mover_reference | BOTH) "win" ")"
result_lose: "(" (mover_reference | BOTH) "lose" ")"
result_draw: "(" "draw" ")"
result_by_score: "(" "by_score" ")"

// -- Mask definitions --
super_mask: mask | mask_like_function | super_mask_and |
    super_mask_or | super_mask_not
super_mask_and: "(" "and" super_mask+ ")"
super_mask_or: "(" "or" super_mask+ ")"
super_mask_not: "(" "not" super_mask ")"

mask: mask_adjacent
    | mask_captured
    | mask_center
    | mask_column
    | mask_corners
    | mask_corner_custodial
    | mask_custodial
    | mask_edge
    | mask_empty
    | mask_hopped
    | mask_occupied
    | mask_prev_move
    | mask_promoted
    | mask_row
    | mask_region

mask_adjacent: "(" "adjacent" super_mask direction_arg? ")"
mask_captured: "(" "captured" ")"
mask_center: "(" "center" ")"
mask_column: "(" "column" nonnegative_int ")"
mask_corners: "(" "corners" ")"
mask_corner_custodial: "(" "corner_custodial" piece_reference
    mover_arg? ")"
mask_custodial: "(" "custodial" piece_reference
    custodial_length_arg mover_arg? orientation_arg? ")"
mask_edge: "(" "edge" (edge | FORWARD | BACKWARD) ")"
mask_empty: "(" "empty" ")"
mask_hopped: "(" "hopped" ")"
mask_occupied: "(" "occupied" mover_reference? ")"
mask_prev_move: "(" "prev_move" mover_reference ")"
mask_promoted: "(" "promoted" ")"
mask_row: "(" "row" nonnegative_int ")"
mask_region: "(" "region" region_reference ")"

// "Mask-like" functions are special functions that also support
    returning a mask. Currently this is only
// used for "line", but it might include other functions in the
    future
```

```
mask_like_function: function_line

// "Multi-masks" are special keywords that are manually split into
     multiple
// sub-masks at compile time. This is mostly useful for the "
    connected" function,
// which expects a list of masks to check for connections between
multi_mask: multi_mask_corners
        | multi_mask_edges
        | multi_mask_edges_no_corners

multi_mask_corners: "(" "corners" ")"
multi_mask_edges: "(" "edges" ")"
multi_mask_edges_no_corners: "(" "edgesNoCorners" ")"

// ---Predicate definitions---
super_predicate: predicate | super_predicate_and |
    super_predicate_or | super_predicate_not
super_predicate_and: "(" "and" super_predicate+ ")"
super_predicate_or: "(" "or" super_predicate+ ")"
super_predicate_not: "(" "not" super_predicate ")"

predicate: predicate_action_was
       | predicate_can_move_again
       | predicate_equals
       | predicate_exists
       | predicate_full_board
       | predicate_function
       | predicate_greater_equals
       | predicate_last_move_in
       | predicate_less_equals
       | predicate_mover_is
       | predicate_no_legal_actions
       | predicate_passed

predicate_action_was: "(" "action_was" mover_reference
    move_type_name ")"
predicate_can_move_again: "(" "can_move_again" move_type_name ")"
predicate_equals: "(" "=" function+ ")"
predicate_exists: "(" "exists" super_mask ")" // technically
    equivalent to (>= (count mask) 1)
predicate_full_board: "(" "full_board" ")"
predicate_function: function // special syntax which is equivalent
     to "(>= function 1)"
predicate_greater_equals: "(" ">=" function function ")"
predicate_last_move_in: "(" "last_move_in" super_mask ")" //
    equivalent to (exists (and (prev_move mover) mask)), but more
    readable
predicate_less_equals: "(" "<=" function function ")"
predicate_mover_is: "(" "mover_is" player_reference ")"
predicate_no_legal_actions: "(" "no_legal_actions" ")"
predicate_passed: "(" "passed" (mover_reference | BOTH) ")"

// Additional (potentially optional) arguments for predicates
```

```
capture_arg: "capture:" boolean
custodial_length_arg: ANY | positive_int
direction_arg: "direction:" (direction | multi_direction)
distance_arg: "distance:" positive_int
exact_arg: "exact:" boolean
exclude_arg: "exclude:" multi_mask_arg
hop_over_arg: "hop_over:" (player_reference | mover_reference)
increment_score_arg: "increment_score:" boolean
mover_arg: "mover:" (mover_reference | BOTH)
multi_mask_arg: multi_mask | super_mask | "(" super_mask+ ")"
orientation_arg: "orientation:" orientation
pattern_arg: "(" positive_int indices_arg ")" // The positive int
   is the "width" of the pattern and the indices are the
   positions within the pattern
indices_arg: "(" nonnegative_int+ ")"
player_arg: "player:" (player_reference | mover_reference)
piece_arg: "piece:" piece_reference
priority_arg: "priority:" nonnegative_int
rotate_arg: "rotate:" boolean
same_piece_arg: "same_piece:" boolean
shape_arg: shape_definition

shape_definition: hexagon_shape
            | hex_rectangle_shape
            | rectangle_shape
            | square_shape

hexagon_shape: "(" "hexagon" odd_int ")"
hex_rectangle_shape: "(" "hex_rectangle" positive_int positive_int
    ")"
rectangle_shape: "(" "rectangle" positive_int positive_int ")"
square_shape: "(" "square" positive_int ")"

// Optional rendering details
rendering: "(" "rendering" rendering_detail+ ")"
rendering_detail: color_assignment
            | piece_shape_assignment

color_assignment: "(" "color" player_reference color ")"
piece_shape_assignment: "(" "shape" piece_reference piece_shape ")
   "

// General-purpose definitions
?number: SIGNED_NUMBER
?nonnegative_int: /(0|[1-9][0-9]*)/
?positive_int: /[1-9][0-9]*/
?odd_int: /[0-9]*[13579]/
?boolean: TRUE | FALSE
?edge: TOP | BOTTOM | LEFT | RIGHT | TOP_LEFT | TOP_RIGHT |
   BOTTOM_LEFT | BOTTOM_RIGHT
?direction: true_direction | relative_direction
?move_type_name: HOP | SLIDE | STEP
?multi_direction: "(" direction+ ")"
```

```
?true_direction: UP | DOWN | LEFT | RIGHT | UP_LEFT | UP_RIGHT |
   DOWN_LEFT | DOWN_RIGHT | VERTICAL | HORIZONTAL | ORTHOGONAL |
   DIAGONAL | BACK_DIAGONAL | FORWARD_DIAGONAL | ANY
?relative_direction: FORWARD | BACKWARD | FORWARD_LEFT |
   FORWARD_RIGHT | BACKWARD_LEFT | BACKWARD_RIGHT
?orientation: VERTICAL | HORIZONTAL | ORTHOGONAL | DIAGONAL |
   BACK_DIAGONAL | FORWARD_DIAGONAL | ANY
?color: WHITE | BLACK
?piece_shape: CIRCLE | SQUARE | TRIANGLE | STAR | DIAMOND
// ----------------------------

?player_reference: P1| P2
?mover_reference: MOVER | OPPONENT
?name: STRING
variable_name: /\?[a-z][a-z0-9]*/
id: /[a-zA-Z0-9_]+/
```

C Benchmark Game Descriptions

Below, we present natural language descriptions of the rules for each of the exemplar games analyzed in Section 6.

***Tic-Tac-Toe*:** Players take turns placing a piece into an empty space on a square 3-by-3 board. If a player forms a line of three of their pieces in a row (either vertically, horizontally, or diagonally), they win. If the board is completely full but no lines have been formed, then the game ends in a draw.

***Connect Four*:** Players take turns placing a piece into the top of one of the seven columns on a 6-by-7 board. The piece then "falls" until it rests on either the bottom of the board or another piece. A player can't place a piece into a column that is already "full." If a player forms a line of four of their pieces in a row (either vertically, horizontally, or diagonally), they win. If the board is completely full but no lines have been formed, then the game ends in a draw.

***Hex*:** Players take turns placing a piece into an empty space on an 11-by-11 board composed of hexagonal tiles (forming a parallelogram, see visual depiction here). The objective for the first player is to form a continuous path of their pieces that connects the top edge of the board with the bottom edge, while the objective for the second player is to do the same but connect the left and right edges of the board. The first player to achieve their objective wins the game. Because of the geometric properties of the board, it's not possible for the game to end in a draw.

***Reversi*:** The game takes place on a square 8-by-8 board. To begin, a white piece is placed at positions D4 and E5 and a black piece is placed at positions D5 and E4 (see visual depiction here). Players take turns placing a piece into an empty space such that a line of one or more of the opponent's pieces are "sandwiched" on either end by the player's pieces. This configuration is called a "custodial" arrangement of pieces. After placing a piece, any of the opponent's pieces which are in such a custodial arrangement are flipped and now belong to the player who just moved. It's possible for a single move to form multiple custodial arrangements in different directions, in which case all of the relevant pieces are flipped. If a player cannot make a legal move, they must pass (and they cannot pass without making a move otherwise). If both players pass, then the game is over. The winner is determined by the player who has the largest number of pieces on the board at the end of the game (in the event of a tie, the game ends in a draw).

***Gomoku*:** Players take turns placing a piece into an empty space on a square 15-by-15 board. If a player forms a line of exactly five of their pieces in a row (either vertically, horizontally, or diagonally), they win. However, forming a line of six or more does not count – the player must have at

least one line of exactly five. If the board is completely full but no lines of exactly five have been formed, then the game ends in a draw.

***Pente*:** Players take turns placing a piece into an empty space on a square 19-by-19 board. If a player forms a line of five of their pieces in a row (either vertically, horizontally, or diagonally), they win. In addition, if placing a piece causes a line of exactly two of the opponent's pieces to be put into a custodial arrangement, the two pieces are captured and removed from a board. Note that placing a piece *into* a custodial arrangement formed by the opponent does not result in any pieces being captured. A player who captures at least 10 of the opponent's pieces over the course of the game wins. In the variant of *Pente* implemented in `Ludii` and `Ludax`, the first player must make their first move into the exact center of the board.

***Yavalath*:** Players take turns placing a piece into an empty space on a regular hexagonal board with a diameter of 9 spaces. If a player forms a line of four of their pieces in any direction (either diagonally or horizontally[4]), they win. However, if a player forms a line of three of their pieces in a row without also forming a line of four, they lose. If the board is completely full but no lines of four or three have been formed, then the game ends in a draw.

***Dai Hasami Shogi*:** The game takes place on a square 9-by-9 board. To begin, white pieces are placed on the bottom two rows of the board and black pieces are placed on the top two rows. Players take turns moving one of their pieces, either by sliding it any number of squares vertically or horizontally (i.e. as a rook) or by hopping over one piece (belonging to either player) vertically or horizontally into an empty square. Hopping over a piece does not capture it, but opposing pieces can be captured "custodially" (i.e. by moving to surround an enemy piece on both sides vertically or horizontally). An opponent's piece in a corner can also be captured by moving a piece to occupy both orthogonally-adjacent squares. A player wins if they manage to form a horizontal or vertical line of 5 pieces in a row if none of those pieces are in their starting rows.

***Wolf and Sheep*:** The game takes place on a square 8-by-8 board. Player 1 begins with four "sheep" pieces placed on the dark squares of the bottom row while Player 2 begins with a single "wolf" piece placed on one of the center dark squares in the top row. Players take turns moving one of their pieces to an unoccupied square – "sheep" can move in either of the forward diagonal directions (i.e. towards the top of the board) while the "wolf" can move in any diagonal direction. Player 1 wins if they manage to block all of Player 2's legal actions while Player 2 wins if they manage to get their piece to the bottom edge of the board.

***English Draughts*:** Also known as *Checkers*, this game takes place on a square 8-by-8 board. Player 1 and 2 begin with 12 "pawns" placed on the dark squares of the bottom three and top three rows, respectively. A pawn can move by either stepping into an unoccupied square in one of the forward diagonals (i.e. towards the top of the board for Player 1 and towards the bottom of the board for Player 2) or by hopping over an opponent's piece in one of those directions and capturing it. If a player has the option to make a capture move, they must do so. In addition, if after making a capture, the capturing piece could make another capture then the active player gets another turn in which they must move their previously-moved piece. If a "pawn" reaches the opposite edge of the board, it is promoted to a "king" which can move and capture in any diagonal direction. The game ends when one player has no legal actions, at which point they lose.

D Gridworld Representation

While `Ludax` is designed to primarily support two-player board games, its flexible syntax allows it to encode certain kinds of simple single-player games often used as RL training environments. By giving one player a single piece capable of stepping one square orthogonally into empty board positions, defining certain board regions as either "targets" or "dangers," and omitting the second player from the move order definition, the resulting game effectively encodes a simple gridworld like

[4] `Ludax` assumes a canonical orientation for hexagonal boards in which the diameter stretches from left to right, though it is functionally equivalent to the orientation in which the diameter runs vertically)

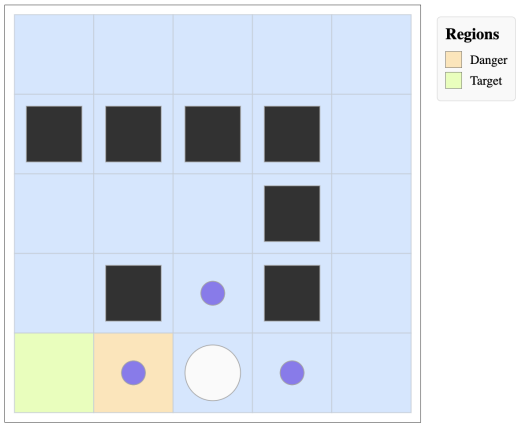

Figure 5: **`Ludax` rendering of a simple gridworld environment akin to `FrozenLake`.** `Ludax`'s syntax can be adapted to represent single-player games with movement dynamics more typical of simple video games. Player 1 (white) moves a single piece (circle) one step at a time, attempting to avoid the "danger" region in orange and reach the "target" region in green.

the classic `FrozenLake` environment. While the resulting action space is not exactly analogous to a traditional gridworld (i.e. for the `Ludax` game, the action space would be defined based on the number of legal squares), it's very straightforward to detect the specification of this kind of game at parse time and adjust the underlying observation and action space accordingly. An example of such an environment rendered using `Ludax`'s interactive debugger is provided in Figure 5.

E Training Hyperparameters

Below we provide the exact training hyperparameters used in the reinforcement learning experiments in Section 7. These are largely copied from the `PGX` implementation.

- **Model architecture:** `ResnetV2`
- **Number of channels:** 128
- **Number of layers:** 6
- **Self-play batch size:** 1024
- **Self-play simulations:** 32
- **Self-play max steps:** 256
- **Training batch size:** 4096
- **Learning rate:** 0.001
- **Evaluation frequency:** 5
- **Training iterations:** 219

Note that each "iteration" consists of generating play data for 256 steps using the self-play batch size of 1024 (see Koyamada et al. (2023)). We train the model for 219 iterations, which corresponds to $256 \times 1024 \times 219 = 57409536$ (or roughly 57 million) steps in the environment.

F Additional Training Runs

In addition to the head-to-head comparison between `Ludax` and `PGX`, we present the results of training additional agents in `Ludax`'s Hex and Connect Four environments in Figure 6. Owing to differing state and action space representations, we cannot directly compare `Ludax` agents to

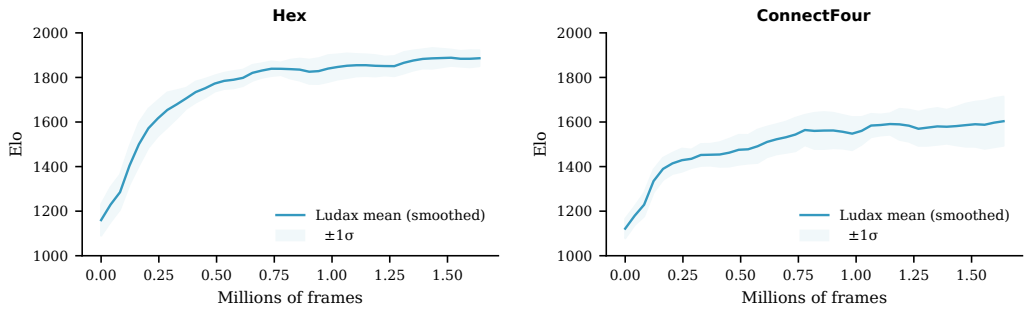

Figure 6: **Performance of reinforcement learning agents trained in the `Ludax`.** Elo scores are computed between 40 model checkpoints and averaged over three runs. Each run took roughly 1 hour to complete on four GH200s.

```
Invent simple rules for a novel two player abstract strategy game
called {name}. Implement it in the ludax language. You will find
attached the ludax's grammar as well as a few examples of games
implemented in ludax. Start by implementing a simplified version
of your rules, and then incrementally add rules that are harder to
express in ludax. At each step, make sure you write a compilable
game according to ludax's grammar.
```

Listing 1: System instruction for LLM-based generation.

`PGX`'s baseline by having the agents play against each other. Instead, we report the base-10 Elo score (similarly to Silver et al. (2018)) of 40 model checkpoints across training, averaged over three runs. We perform 64 playthroughs for each pairing of checkpoints until all Elo ratings converge. While the specific Elo scores are not meaningful in isolation, the curves demonstrate that standard RL techniques can learn effectively in a variety of `Ludax` environments.

G Game Generation

We attempt to synthesize new games in the `Ludax` DSL using two approaches: random sampling and LLM-based generation. In Table 1, we present the `GAVEL` game evaluation metrics for each method.

Random Sampling: Games are generated by naive uniform random sampling. Starting from the root game "ludeme" (i.e. production rule), we sample the next ludeme among those which are valid continuations according to the grammar. Additionally, we impose a maximum syntax tree depth of 5, beyond which a closing bracket is always given priority.

LLM-based Generation: Games are generated as a few-shot task. The model is prompted with a system instruction (Listing 1), the full grammar (Appendix B), and the game implementations from Appendix C as examples. The model is instructed to describe the rules of a new game and produce multiple `Ludax` implementations of increasing complexity; we evaluate only the final game produced. To encourage diversity, each attempt is seeded with a randomly generated and nonsensical game name such "Outstanding Rainbow Spaniel."

GAVEL-like Evaluation: Inspired by Todd et al. (2024), we assess each generated game as follows:

1. A game is *playable* if its description compiles and runs without error.
2. For each *playable* game, we run agent-vs-agent playthroughs using a custom JAX implementation of MCTS with UCB1 (Kocsis & Szepesvári, 2006).

Method	Playable	Interesting	GAVEL Score	Strategic Depth
Default Games	100%	100%	0.69 ±0.15	0.66 ±0.15
Random Sampling	4%	0%	0.00 ±0.00	0.00 ±0.00
GPT-OSS-120B	95%	83%	0.59 ±0.22	0.58 ±0.17
LLaMa-4-17B	82%	42%	0.49 ±0.21	0.68 ±0.23

Table 1: GAVEL-based evaluation metrics for 100 generated games, obtained either by uniform random sampling or an LLM. As a baseline, we report results for all default games in Appendix C. *Playable* and *Interesting* denote percentages over all generated games (*Playable* ≥ *Interesting*). *GAVEL score* and *Strategic Depth* report the median and standard deviation, computed only on playable games.

3. We compute the following heuristics from these playthroughs:

   - **Balance:** max winrate gap between players
   - **Decisiveness:** fraction of non-draw outcomes
   - **Completion:** fraction of games reaching a terminal state
   - **Agency:** fraction of turns with > 1 legal move
   - **Coverage:** fraction of board sites occupied at least once
   - **Strategic Depth:** difference in winrate between a stronger MCTS agent and a weaker one (fewer simulations).

The overall "`GAVEL` score" is the harmonic mean of the individual heuristic scores. Games with a `GAVEL` score > 0.4 are deemed potentially *interesting*. We note that this experiment is preliminary: it omits diversity measures, and the limited search budget for MCTS means they will frequently miss good moves a stronger agent might find. Nevertheless, the fact that an LLM can implement novel games in `Ludax` without finetuning suggests that `Ludax` 's grammar is intuitive and highlights its potential for game generation.

Hyperparameters: For each method, we sample 100 games. For the LLM-based methods, we use a sampling temperature of 0.2. To compute the evaluation score, we run 100 agent-vs-agent simulations for each game. The MCTS agents perform 100 iterations (i.e. traversal, expansion, and random rollout) for each action. For the "strategic depth" evaluation we compare against an MCTS agent that performs 50 iterations per action.