From Code to Play: Benchmarking Program Search for Games Using Large Language Models

Manuel Eberhardinger, James Goodman, Alexander Dockhorn, Diego Perez-Liebana, Raluca D. Gaina, Duygu Çakmak, Setareh Maghsudi, Simon Lucas

***Abstract*—Large language models (LLMs) have shown impressive capabilities in generating program code, opening exciting opportunities for applying program synthesis to games. In this work, we explore the potential of LLMs to write usable code for a wide range of gaming applications, focusing on two programming languages, Python and Java. We use a hill-climbing algorithm, where the mutations and seeds of the initial programs are controlled by LLMs. For Python, the framework covers various game-related tasks, including five miniature versions of Atari games, ten levels of *Baba is You*, an environment inspired by *Asteroids*, and a maze generation task. For Java, the framework contains 12 games from the TAG tabletop games framework. Across 29 tasks, we evaluated 11 language models for generating Python and Java code. Our findings suggest that the performance of LLMs depends more on the task than on model size. In addition, the experiments show that running the hill-climbing algorithm multiple times with fewer iterations is better than a single run with more iterations.**

***Index Terms*—Game AI, Large Language Models, Program Synthesis**

I. Introduction

Before the emergence of large language models (LLMs) for code [1], program synthesis in imperative or object-oriented languages like Python or Java was considered a highly challenging task due to the combinatorial explosion of the search space [2]. Therefore, most solvable tasks were restricted to simple problem domains such as string manipulation or list sorting, typically implemented within a predefined domain-specific language (DSL) [3]. Similarly, program synthesis for games was limited to simple problems with well-defined search spaces, achievable only by incorporating high-level game-specific concepts into the DSL [4], [5], [6], [7].

The goal of program synthesis is the automatic generation of programs that satisfy certain requirements, such as input-output examples (e.g. five pairs of unsorted lists and their sorted counterparts), natural language descriptions of the program to be generated or formal specifications such as logical constraints [2]. Automatic construction of programs for high-level programming languages in game research has hardly been explored. Most discussions merely outlined its potential applications [8], or focused on the missing aspects of automated game development systems to move from game description languages to programming languages [9].

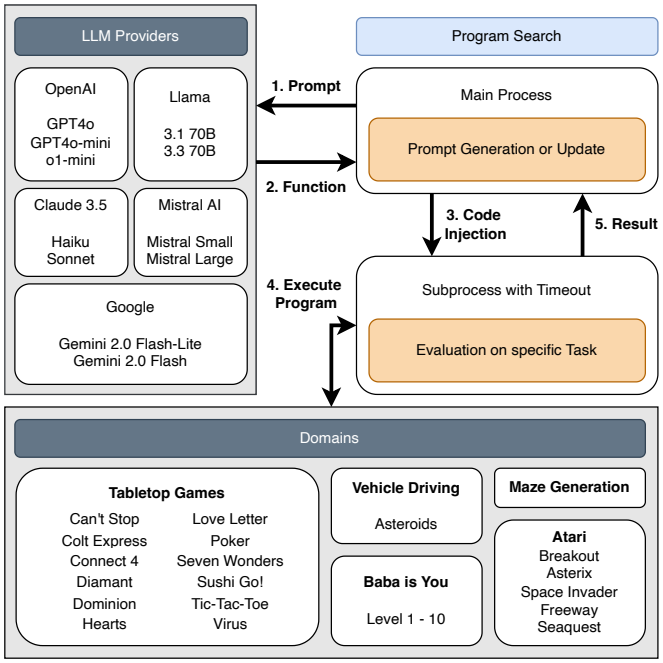

Fig. 1: The general framework for program search begins by generating an initial task prompt (1), which is processed by one of the integrated LLMs to produce a function (2). This function is then evaluated within a subprocess (3), which executes the program in the given task (4) and then the results are reported back to the main process (5). The main process updates the prompt based on the evaluation outcomes and either returns it to the LLM (repeat from 1) for further refinement or concludes if the evaluation criteria are reached.

Recently, methods for LLM-based program search have been introduced for the automatic design of playable games based on program code [10], [11], [12] and to generate game content through JSON representations [13]. LLMs have also been adapted to synthesize programmatic policies in Python, which are then converted into a DSL suitable for the target environment [14], as well as to construct world models in Python that approximate reward and state transition functions for simple games, enabling action plan generation [15].

In this work, we explore the potential of LLM-based program search for a wider range of games without depending on a

Manuel Eberhardinger is with the Institute of Applied AI, Stuttgart Media University, Nobelstr. 10, 70569 Stuttgart, Germany *(Corresponding Author; email: eberhardinger@hdm-stuttgart.de)*

James Goodman, Diego Perez-Liebana, Raluca D. Gaina and Simon Lucas are with the School of Electronic Engineering and Computer Science, Queen Mary University of London, E1 4NS London, U.K.

Alexander Dockhorn is with the SDU Metaverse Lab, University of Southern Denmark, Campusvej 55, DK-5230 Odense

Duygu Çakmak is with Creative Assembly, RH12 1JW Horsham, U.K.

Setareh Maghsudi is with the Chair of Learning Technical Systems, Ruhr-University Bochum, Universitätsstr. 150, 44801 Bochum, Germany

predefined JSON converter [13] or on predefined specifications such as a DSL (e.g., Ludii [10], the video game description language [11], or Karel [14]). Our aim is to enable LLMs to synthesize program code that can be used directly, without requiring additional transformations or prior specifications. We evaluate this approach across different domains using two programming languages: Python and Java. In Python, we focus on synthesizing programmatic agent policies and functions for procedural content generation (PCG). In Java, the method is integrated into TAG, a tabletop games framework, in which LLMs design heuristics for board games [16].

Our goal is not to propose a new method for program synthesis, but to introduce an easy-to-use and extensible framework to evaluate the current performance of LLMs for the synthesis of game-related program code. To achieve this, we have integrated five different LLM providers with a total of 11 different models. For the synthesis of Python code, the framework consists of five miniature versions of Atari games where the input is represented symbolically [17], ten levels of the game *Baba is You*, in which various game mechanics are tested [18], a vehicle driving environment based on the game *Asteroids*, and procedural content generation in the form of mazes. For Java, the framework consists of 12 tabletop games of the TAG framework [16]. In total, we evaluate the LLMs on 29 different tasks. An overview of our proposed framework, as well as games and LLMs used, is shown in Figure 1.

Our contributions are:

- We perform an empirical study to evaluate the current state-of-the-art of LLM-based program search for games.
- We introduce an easy-to-use and extensible framework with 29 tasks that evaluate various aspects of game mechanics.
- We open-source our code upon publication. Currently, only the example prompts for the experiments are available in the repository[1].

The structure of the paper is as follows: We review related work on program synthesis for games in Section II. Afterwards, the framework is introduced in Section III, followed by a short summary of the LLMs used in Section IV. Section V first describes the experimental setup (Section V-A) and then the experiments for the game applications, starting with miniature versions of the Atari games, followed by the Asteriods-inspired vehicle driving experiments. Then we discuss the puzzle game Baba is You is and the maze generation experiments. We conclude the first part of the experiments with an overall evaluation of the generated Python code. The second part of the experiments, starting in Section V-G, focuses on the TAG framework which uses Java as the programming language. In addition to the standardised experiments, longer-running experiments that evaluate the impact on the number of iterations for the five games in the Atari domain are reported in Section VI. Finally, we discuss practical insights gained from the experiments in Section VII, and conclude the paper in Section VIII.

[1]https://github.com/ManuelEberhardinger/Benchmarking-Language-Model-Based-Program-Search-for-Games

II. Related Work

There are a considerable number of studies that use program synthesis approaches for games. Butler et al. used SMT-solvers to search for programs within a Lisp-based DSL, enabling the generation of diverse boss fights in Megaman [4] and puzzles for the game Nonograms [5]. In [6], a method was introduced for learning combinations of logical programs to solve simple grid-based games like Nim. Cropper et al. [19], [20], [21] developed a comprehensive benchmark of 50 games to recover game rules from gameplay traces using inductive logic programming (ILP). Furthermore, Evans et al. [22] applied a differentiable form of ILP to learn interpretable rules for Sokoban. Recently, a method for learning programmatic policies for zero-shot coordination problems in cooperative tasks was introduced and demonstrated in the game Overcooked [23]. In contrast to learning programmatic policies, there is also work focusing on using program synthesis to explain the decision-making process of game-playing agents [24].

Mariño et al. [7] utilized program search to develop strategies for the game MicroRTS, comparing the resulting programmatic policies with those created by human developers. Their findings demonstrated that the synthesized programs performed comparably to those written by humans. Subsequent research built on this foundation by introducing improved search techniques, including bilevel search [25], by guiding the program search [26] or by searching in semantic spaces [27]. Recently, an approach for combining LLMs with local search algorithms was proposed for MicroRTS [28], where the authors showed that providing initial programs with LLMs found better solutions faster and improved the performance of the final programs.

In [29], Genetic Programming (GP) is used to search for evaluation functions within a predefined DSL for the board game 7 Wonders. Similarly, Sturtevant and White generate evaluation functions automatically by using reinforcement learning to learn more complex features by combining atomic features for the game Hearts [30]. This approach resembles our experiments with the TAG framework, where heuristic functions are synthesized; however, we use Java without relying on predefined concepts. GP has also been applied to generate game agents for various scenarios, including a fighting game [31], a platformer game [32], a puzzle game [33], and to create explanations for a maze runner agent [34]. Additionally, Wilson et al. [35] used Cartesian GP to develop programs for Atari games, processing pixel observations through predefined mathematical, statistical, or list functions.

A recent approach from cognitive science, known as language-informed thinking [36], combines large language models (LLMs) with Bayesian inference. This method enables LLMs to pose questions in natural language, which are then translated into a language of thought [37] represented as a probabilistic programming language. Grand et al. extended this approach to the board game Battleship, demonstrating that the questions generated by LLMs aligned closely with the performance of human players [38].

Verma et al. [39], [40] employed neurosymbolic methods to synthesize programmatic policies for a car racing game, demonstrating that these programs were more robust than

neural network policies while achieving comparable rewards. While this approach shares similarities with the vehicle driving experiments in our work, it is more constrained, as the search space is limited to the provided DSL and our vehicle driving problem requires a planning algorithm to be solvable.

In [41], a reactive programming language with a novel program synthesis algorithm is introduced to discover causal structures represented as state machines in combination with program code. They evaluate the proposed method for 2D grid games similar to Super Mario.

Voyager [42] is a lifelong learning agent for the Minecraft environment that uses an LLM to synthesize code in the Mineflayer API[2], which is then executable to obtain the actions. In addition, a second LLM is used as a high-level planner to create a task curriculum for the agent. Moreover, Ma et al. [43] proposed an evolutionary approach using LLMs to synthesize reward functions for complex control tasks, achieving superior performance compared to human-engineered reward functions.

The key distinction between our work and the discussed literature is the use of high-level programming languages (Python and Java), making our approach applicable to a broader range of tasks without relying on predefined building blocks or a programming library. The work that is most similar to ours and is also used for game environments is [15], where Python code is synthesized to approximate a world model. However, this work is limited to a single, different type of task.

III. Framework

The general framework we propose is based on a hill-climbing algorithm where the mutations and the seed of the initial program are performed by an LLM based on evolutionary approaches using LLMs [44], [45]. Thus, our framework belongs to the group of neurosymbolic programming methods [46], as we use an LLM to generate programs that are checked for correctness and functionality by symbolic verifiers, in our case the Python interpreter and Java compiler. The overview of the framework is illustrated in Figure 1, which shows the high-level interaction between the different modules and processes. Synthesized program code by LLMs is always executed within a safe subprocess environment, ensuring that the main process can terminate it after a certain time limit to prevent infinite execution of the program.

A detailed description of the complete algorithm is provided in Algorithm 2. Our framework consists of two iterative processes that control the length of the search, defined by the input parameter `iterations`, as well as the number of attempts to repair the program in each iteration defined by the input parameter `maxAttempts`. Each iteration starts by generating or updating a task prompt to obtain an initial Python or Java function from Algorithm 1. The procedure of Algorithm 1 queries the LLM and tries to fix compilation or syntax errors for the given number of `maxAttempts`. The program code is then returned to Algorithm 2 and is injected and executed in a subprocess. If the function is executed successfully and returns an improved reward, we update the prompt with the achieved evaluation metric and all relevant environment-specific details, such as the action trace of the executed function. If a runtime error occurs, the error description is included in the prompt for refinement of the program, so that it is possible for the LLM to fix these errors in the next iteration. These steps are repeated iteratively until the evaluation criteria defined by the fitness function are satisfied or the specified number of `iterations` is reached.

We explain the domain-specific adaptations of the framework in the respective chapters in section V. While the overall framework is similar for all tasks, domain-specific adaptations are necessary, such as the description of the environment or the game logic, as well as the objective of the game.

[2]https://github.com/PrismarineJS/mineflayer/tree/master

Algorithm 1 The algorithm for querying an LLM with the number of attempts to generate/update or repair the program.

```
procedure QUERYLLMWITHREPAIR(prompt, maxAttempts)
    program ← QUERYLLM(prompt)
    j ← 1
    while j < maxAttempts and not CanExec(program) do
        prompt ← GETREPAIRPROMPT(program)
        program ← QUERYLLM(prompt)
        j ← j + 1
    end while
return program
end procedure
```

Algorithm 2 The algorithm for our proposed framework, which uses the procedure from Algorithm 1 in each iteration to improve or generate a program.

```
procedure PROGRAMSEARCH(task, iterations, maxAttempts)
    bestFitness ← 0
    i ← 0
    bestProgram ← NULL
    repeat
        if bestProgram != NULL then
            prompt ← UPDATETASKPROMPT(task, bestResult, bestFitness, bestProgram)
        else
            prompt ← GETTASKPROMPT(task)
        end if
        program ← QUERYLLMWITHREPAIR(prompt, maxAttempts)
        result ← INJECTANDRUNCODE(task, program)
        fitness ← EVALUATEFITNESS(task, result)
        if fitness > bestFitness then
            bestProgram ← program
            bestResult ← result
            bestFitness ← fitness
        end if
        i ← i + 1
    until CHECKCRITERION(fitness) or i == iterations
return program
end procedure
```

TABLE I: The used LLMs with the specific version or date of the release.

LLM	Release Date
Claude 3.5 Sonnet	2024-10-22
Claude 3.5 Haiku	2024-10-22
Gemini 2.0 Flash	2025-02
Gemini 2.0 Flash-Lite	2025-02
Mistral Large	2024-11
Mistral Small	2025-03
o1-mini	2024-09-12
GPT 4o	2024-08-06
GPT 4o mini	2024-07-18
Llama 3.3 70B	2024-12-06
Llama 3.1 70B	2024-07-23

IV. Large Language Models

For our benchmark, we integrated five LLM providers for generating Python and Java code in our framework, using one smaller and one larger version for each model type. From OpenAI, we utilize models from the GPT-4o family[3], based on GPT-4 [47]. For the new ChatGPT models in the o1 generation, we use o1-mini, which offers performance comparable to o1-preview for coding tasks.[4] We also incorporate the latest models from Mistral[5], Claude 3.5[6] from Anthropic, based on Claude 3 [48], and the 2.0 Gemini Flash models[7] [49], provided by Google, for both programming languages. Two open source models from the Llama family are used, Llama 3.1 70B [50] and Llama 3.3 70B, which performs similarly to Llama 3.1 405B for code-related tasks.[8] Details on the model versions and their release dates are in Table I.

V. Game Applications

In the following section, we first describe the experimental setup, followed by the experiments that were conducted for each of our target domains.

A. Experimental Setup

In general, the experiments reported run 10 independent `trials` of Algorithm 2, and the final recommendation is the best performing agent from these 10 trials. In each trial, 10 iterations of search were performed, with 3 queries used to create/improve or repair the policy.

B. Programmatic Policies: Minatar

Minatar [17] is a collection of five games that are miniature versions of Atari games. In Minatar, the games are represented as a symbolic state space on a $10\times10\times n$ grid, where n represents the number of channels, and each channel represents an object such as walls, enemies or the agent. Minatar is an ideal test bed for experiments, as the games are more efficient to learn without changing the game mechanics of the original game. Previously, Minatar was used in [24] to explain the behaviour of agents through program synthesis, but it was only possible to explain short sub-trajectories since enumerative search-based methods were used to search through a predefined domain-specific language that resembles Lisp. In our experiments, we use all available Minatar environments, which are shown in Figure 2. The games are:

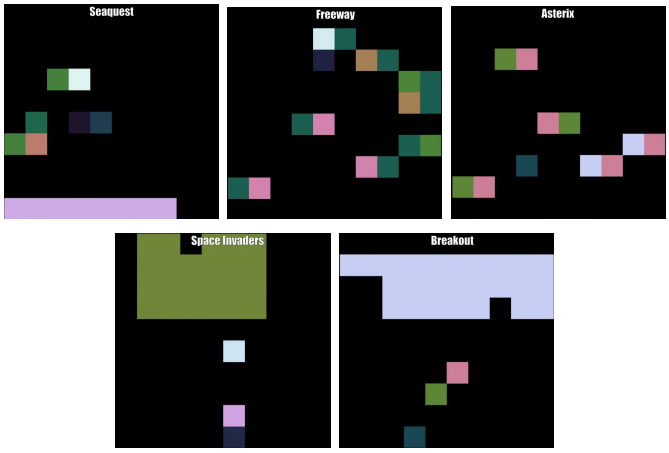

Fig. 2: The five miniature versions of the Atari games (left to right, top to bottom: Seaquest, Freeway, Asterix, Space Invaders and Breakout), which are used for the synthesis of the programmatic strategies. Each colour represents a different type of object, e.g. the paddle in dark blue, the ball in green and the track of the ball in pink for the game Breakout.

- **Seaquest**: The agent controls a submarine and is able to shoot bullets. The objective is to save as many divers as possible, while also shooting enemy submarines or sharks. Each time an enemy is struck, the reward is increased by one. When the submarine saves the divers, the agent also receives a reward.
- **Freeway:** The agent controls a chicken that needs to cross a road during rush hour, while avoiding the traffic. For each chicken that crosses the road safely, the agent receives one point.
- **Asterix**: The objective of the game is to collect gold while avoiding enemies. The player gets one point for each collected gold and the game is over when the player is hit by an enemy.
- **Space Invaders**: The agent controls a cannon and shoots aliens while dodging bullets launched from the alien spaceship. Additionally, the player must prevent the aliens from reaching the bottom of the screen. For each destroyed alien, one point is received.
- **Breakout**: The goal is to destroy all the bricks with the ball by controlling the paddle to bounce the ball off before it goes out off the screen. With each destroyed brick the agent receives one point.

The LLMs were prompted to generate a Python function which can be used as a policy to play the game. The prompt contains information about the game rules, the objective of

[3]https://platform.openai.com/docs/models
[4]https://openai.com/index/openai-o1-mini-advancing-cost-efficient-reasoning/
[5]https://docs.mistral.ai/getting-started/models/models_overview/
[6]https://docs.anthropic.com/en/docs/about-claude/models
[7]https://ai.google.dev/gemini-api/docs/models/gemini
[8]https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/

TABLE II: Max Reward of the best program for the Minatar experiments with 50 evaluation episodes.

Model	Seaquest	Freeway	Breakout	Asterix	SpaceInvader
Claude Sonnet	2.43	7.22	18.43	11.37	29.07
Claude Haiku	3.68	7.33	7.13	6.78	21.91
Gemini Flash	2.98	8.07	12.3	6.09	26.34
Gemini Lite	0.58	6.49	15.56	5.06	16.57
Mistral Large	1.3	8.48	12.03	6.48	20.89
Mistral Small	0.26	8.2	5.59	5.88	22.41
o1 mini	5.73	9.52	21.81	10.39	22.45
GPT 4o	1.15	10.05	15.26	10.91	23.83
GPT 4o mini	0.47	8.44	8.56	6.55	27.45
Llama 3.3 70B	4.01	7.74	8.85	5.8	17.1
Llama 3.1 70B	4.08	8.02	4.39	5.54	13.04

```
[empty, empty, empty, empty, empty,  empty, empty, empty, empty, empty]
[brick, brick, brick, brick, brick,  brick, brick, brick, brick, brick]
[brick, brick, brick, brick, brick,  brick, brick, brick, brick, brick]
[brick, brick, brick, brick, brick,  brick, brick, brick, brick, brick]
[empty, empty, empty, empty, empty,  empty, empty, empty, empty, empty]
[empty, empty, empty, empty, empty,  empty, empty, empty, empty, empty]
[empty, empty, empty, empty, empty,  empty, empty, empty, empty, empty]
[empty, empty, empty, empty, empty,  empty, empty, empty, empty, empty]
[empty, empty, empty, empty, empty,  empty, empty, empty, empty, empty]
[empty, empty, empty, empty, paddle, empty, empty, empty, empty, empty]
```

Fig. 3: The text description of the state for the Breakout game which is included in the prompt.

the agent and also the possible actions of the environment and available game objects. The description of the game was taken from Young and Tian [17]. The prompts for the games are available in the code repository[9]. The LLM receives only the initial state, which is preprocessed from the state input of the environment, a one-hot encoded 3D array, into a 2D array with text descriptions for each grid cell representing the cell's object. Figure 3 shows an example of the converted state for Breakout. All other games convert the state in a similar way so that the LLM can process the state input semantically.

Each of the games tests the LLM for different game concepts. Space Invaders, Breakout and Freeway restrict the agent's movement by only allowing horizontal or vertical movement. Space Invader and Seaquest allow the player to fight the enemy, while in Asterix and Freeway the player can only avoid the enemies. In Asterix, the player must also collect items in order to receive a reward. Seaquest is the most difficult game, as the player has to collect six divers and then reach the surface so that the divers can leave the submarine, but at the same time the player has to shoot down enemies. Breakout is one of the easier games compared to the others, as there are no opponents and the player only has to anticipate where the ball will land in order to bounce it off with the paddle.

For each program, 50 evaluation episodes were performed. All trials resulted in successful programs, i.e., programs that were executable and returned a positive reward. Table II presents the max. reward of the best program found and Table III shows the average reward with the standard deviation of the 10 programs found in each trial.

In general, larger models outperform their smaller counterparts in average reward, with o1 mini and Claude Sonnet achieving the highest overall performance. This trend is also reflected in the maximum reward: Claude Sonnet generated the best programs for Space Invaders and Asterix, while o1 mini produced the best results for Seaquest and Breakout. GPT 4o found the best performing program for Freeway, although o1 mini slightly outperforms it on average.

[9]https://github.com/ManuelEberhardinger/Benchmarking-Language-Model-Based-Program-Search-for-Games

TABLE III: Average reward of the 10 programs found after each trial. We also report the standard deviation σ of the 10 programs after each trial. The min. reward for all games is 0, which indicates that there is a high variance in between the trials.

Model	Seaquest	Freeway	Breakout	Asterix	SpaceInvader
Claude Sonnet	1.56±0.38	3.32±2.78	8.25±4.1	6.52±2.46	22.38±5.04
Claude Haiku	1.07±0.96	3.8±3.03	3.77±1.85	5.56±0.85	14.5±5.53
Gemini Flash	1.04±0.75	3.31±2.83	5.82±3.34	4.36±1.22	9.98±9.27
Gemini Lite	0.14±0.19	0.65±1.95	6.8±5.25	3.13±1.69	5.02±4.12
Mistral Large	0.29±0.45	6.6±2.3	5.69±2.67	4.75±1.12	12.62±4.85
Mistral Small	0.07±0.09	3.44±3.15	2.96±1.97	3.51±1.67	9.46±6.76
o1 mini	1.49±1.56	7.9±1.81	8.58±4.99	7.53±1.46	19.48±3.56
GPT 4o	0.36±0.36	7.75±1.56	6.6±3.97	7.36±2.85	15.29±4.7
GPT 4o mini	0.16±0.18	3.94±3.97	3.53±2.47	3.88±1.98	6.95±7.85
Llama 3.3 70B	0.79±1.14	3.22±2.26	3.42±2.86	3.87±1.39	6.16±4.07
Llama 3.1 70B	0.7±1.17	5.23±2.51	2.06±1.33	3.03±1.58	5.76±3.9

In the case of Space Invaders, GPT 4o mini achieved the second best program, though this appears to be a special case, since its average reward is approximately three times lower than that of Claude Sonnet, suggesting that the high performance may be due to chance. The standard deviation across the top 10 programs is often quite high, indicating that multiple trials of program search may be more effective than a single long run. To explore this hypothesis, we conduct a long running experiment in the Minatar domain, presented in section VI, where we also show reward curves to determine when the best programs is found. Interestingly, in some cases smaller models outperform their larger counterparts, e.g., Claude Haiku in Seaquest and Freeway and Gemini Flash Lite in Breakout. It can be seen from both tables that it is more difficult to find good programs for more complicated games such as Seaquest.

Looking at the best programs for each LLM in Seaquest, o1 mini is the only model that defines multiple utility functions to check if it can shoot enemies, or implements one-step look-ahead (OSLA) to check if nearby cells are safe to move to.

In Freeway, all the best performing programs show similar behaviour for each LLM and implement some form of OSLA to check whether it is safe for the chicken to cross the road. GPT 4o's best performing program even uses three look-ahead steps, i.e. it only moves forward if the three cells above the chicken are empty.

For Breakout, the best programs are all able to locate the ball and also determine the direction in which the ball is moving. Claude Sonnet and o1 mini even check whether the ball bounces off walls. The best performing program of o1 mini simulates the movement of the ball until it is in the same row as the paddle.

In Asterix, the best performing programs with a reward over 10 prioritise reaching the gold while checking whether it is moving towards an enemy and, if so, avoiding the enemy. The

TABLE IV: The minimum distance of the best program for the Asteroids ship driving experiments for different rotation speeds ω in degrees per second. D_{avg} shows the average distance of the best programs for the different rotation speeds. For each program, the same set of 5 evaluation tasks were used, each with different target positions and initial states of the ship. The heatmap uses an inverted colour gradient, i.e., values close to zero are darker.

Model	$\omega = 10$	$\omega = 20$	$\omega = 30$	$\omega = 40$	$\omega = 50$	$\omega = 60$	$\omega = 70$	$\omega = 80$	$\omega = 90$	$\omega = 100$	D_{avg}
Claude Sonnet	133.36	97.97	87.09	69.18	70.19	72.03	75.75	66.71	69.61	65.67	80.76
Claude Haiku	101.0	119.55	92.98	71.7	79.92	67.01	62.01	74.83	69.19	75.37	81.36
Gemini Flash	171.79	109.88	81.12	85.61	69.4	74.91	76.53	74.07	67.49	74.02	88.48
Gemini Lite	169.27	106.47	91.18	72.45	62.3	73.63	57.69	101.62	100.37	68.42	90.34
Mistral Large	113.69	76.54	82.5	68.81	68.42	83.33	55.62	63.58	61.58	57.04	73.11
Mistral Small	144.33	90.16	71.24	100.13	57.9	60.84	68.51	58.6	56.92	56.53	76.52
o1 mini	123.64	101.49	86.57	77.88	83.49	64.92	79.07	71.62	57.42	71.16	81.73
GPT 4o	127.58	99.21	78.93	111.06	64.91	99.81	95.28	83.24	74.2	61.57	89.58
GPT 4o mini	144.86	134.72	104.54	111.67	89.83	94.26	84.0	96.41	95.25	96.99	105.25
Llama 3.3 70B	152.17	106.87	111.51	119.46	105.09	106.36	77.21	97.09	75.86	101.72	105.33
Llama 3.1 70B	135.31	96.67	92.75	72.29	74.85	62.51	62.15	64.76	58.73	62.49	78.25

mediocre performing models often prioritize the gold without checking if there are enemies nearby. Claude Sonnet even takes into account the trail of the enemy to check if the chosen action is safe.

In Space Invader, the good programs with a reward of over 20 correctly locate the enemy bullets, the aliens and the cannon and then use threat detection to check whether the enemy bullets need to be dodged before the enemies themselves are shot. The programs with a score below 20 do not anticipate the movement of aliens or prioritise shooting enemies over avoiding enemy bullets.

Overall, it can be said that in the Minatar games larger models on average show more sophisticated behaviour in the programs, but as can be seen with Claude Haiku or Gemini Flash Lite, this is not always the case, since for some games they achieve a better reward. Currently, only a very simple prompting strategy is used, which already gives comparable results to some of the baselines reported in [17] or even outperforms all baselines in the case of Breakout. Using more complicated prompting strategies, such as Chain of Thought [51] or adding a crossover operator could lead to improvements in the programs found.

C. Vehicle Driving

The task is to pilot an Asteroids-style spaceship from its start state to the target, where it should rest until the end of the episode. Each episode is 101 steps. At each step, there are 4 discrete actions: NO_OP, THRUST, ROTATE_LEFT, ROTATE_RIGHT. We experimented with vehicle physics in order to make an interesting challenge. Drag is set to be low, which leads to a high risk of overshooting the target unless countermeasures are taken. At each step, the agent is given an observation of the ship state and the position of the target.

The prompt includes some helper classes and functions, including a Vector2d class and the Asteroids ship, as well as a Vehicle superclass. In addition, we add strong hints to make the problem solvable for LLMs, which are summarized as follows[10]:

- Best solved using search algorithms: try One Step Lookahead, Monte Carlo Tree Search or Rolling Horizon Evolution.
- Try using a heuristic function that values facing towards the target as well as being close to the target.
- Try using Macro-actions - e.g. simply repeating each action a number of times.

Table IV shows the results of the driving experiments for different rotation speeds ω to adjust the difficulty level of steering the asteroid ship. The numbers are the minimum distance achieved by the best program for five evaluation episodes. D_{avg} is the average of all distances for each LLM. We omit the number of successful iterations because no LLM managed to stop the asteroid ship at the target position in all five evaluation episodes. A program was considered successful if it could consistently stop the vehicle within a specified tolerance t across all evaluation episodes. In our experiments, the synthesized programs succeeded in stopping the vehicle in only one or two episodes within a tolerance of $t = 10$, and thus no program qualified as successful. The best models overall are the two Mistral LLMs, which together achieve the best programs for 9 out of 10 rotation speeds. Only for $\omega = 10$ did another LLM (Claude Haiku) find the best program. In terms of D_{avg}, larger models generally outperformed their smaller counterparts but usually only by a small margin. As no LLM successfully solved the problem with a simple prompting strategy in this experiment, we consider it a compelling challenge for future research.

D. Baba is You

Baba is you is a complex puzzle game in which the player manipulates a 2D grid environment to reach a given goal. The environment consists of word blocks and corresponding entities that can be pushed around. By placing word blocks next to each other, rules can be formed. These rules are active as long as the given word block sequence remains intact. This way, players can change how objects behave, which objects they control, or which conditions must be satisfied to win.

[10]The complete prompt is given in the code repository.

(a) Level 1 (b) Level 2 (c) Level 3 (d) Level 4 (e) Level 5

(f) Level 6 (g) Level 7 (h) Level 8 (i) Level 9 (j) Level 10

Fig. 4: Demo levels used for the evaluation of LLM capabilities in the Baba is You domain.

TABLE V: Highest reward per language model and level (with number of successful trials per level). The color of the heatmap is based on the successful trials, as it is more important to know how many times the LLMs were able to solve a level out of all 10 trials or if the trials were solved randomly.

Model	**Level 1**	**Level 2**	**Level 3**	**Level 4**	**Level 5**	**Level 6**	**Level 7**	**Level 8**	**Level 9**	**Level 10**	**#Levels solved**
Claude Sonnet	95 (10)	89 (5)	0.0 (0)	95 (10)	0.0 (0)	91 (1)	94 (10)	95 (10)	90 (1)	92 (3)	8
Claude Haiku	95 (10)	0.0 (0)	0.0 (0)	95 (10)	0.0 (0)	0.0 (0)	94 (10)	95 (10)	0.0 (0)	0.0 (0)	4
Gemini Flash	95 (10)	0.0 (0)	0.0 (0)	95 (2)	0.0 (0)	0.0 (0)	94 (7)	95 (9)	0.0 (0)	0.0 (0)	4
Gemini Lite	95 (10)	0.0 (0)	0.0 (0)	95 (2)	0.0 (0)	0.0 (0)	94 (7)	95 (9)	0.0 (0)	0.0 (0)	4
Mistral Large	95 (6)	89 (1)	0.0 (0)	72 (1)	0.0 (0)	0.0 (0)	94 (9)	95 (7)	0.0 (0)	0.0 (0)	5
Mistral Small	95 (6)	0.0 (0)	0.0 (0)	63 (1)	0.0 (0)	0.0 (0)	94 (2)	95 (8)	64 (2)	0.0 (0)	5
o1 mini	95 (10)	89 (2)	0.0 (0)	95 (10)	0.0 (0)	0.0 (0)	94 (10)	95 (10)	82 (1)	92 (5)	7
GPT 4o	95 (10)	89 (1)	0.0 (0)	95 (9)	0.0 (0)	0.0 (0)	94 (8)	95 (9)	0.0 (0)	92 (1)	5
GPT 4o mini	95 (8)	0.0 (0)	0.0 (0)	0.0 (0)	0.0 (0)	0.0 (0)	90 (3)	95 (4)	0.0 (0)	0.0 (0)	3
Llama 3.3 70B	95 (10)	0.0 (0)	0.0 (0)	95 (3)	0.0 (0)	0.0 (0)	94 (6)	95 (6)	0.0 (0)	0.0 (0)	4
Llama 3.1 70B	95 (10)	0.0 (0)	0.0 (0)	95 (2)	0.0 (0)	51 (1)	94 (4)	95 (6)	0.0 (0)	0.0 (0)	5

For our experiments, we used a Python version[11] of the Keke is You AI framework [18]. For this domain, we prompted the LLMs to provide a policy, giving a short description of the game and the initial state of the level (the complete prompt is in the repository). Similar to the Minatar experiments, the state is converted into a text description. The function to be written should use the current state as input and return a movement direction or the command for waiting a turn. Each episode ends after 100 actions or once the win condition is fulfilled. A reward is awarded based on the maximum number of actions (100) minus the number of steps taken. Thus, the return can be maximized by finishing the level as fast as possible. Each level can be solved in less than 20 actions.

In our tests, we queried the agent to solve 10 simple demo levels (see Figure 4). Each of the levels focuses on one or more key mechanics of the framework such as rule interpretation (levels 1-10), rule creation (levels 2, 3, 5) or destruction (levels 6, 8, 9, 10), and object manipulation (level 7). Table V shows the results of our comparison of the LLM models' capabilities. The number of successful iterations is shown in brackets.

All agents were able to solve at least 3 out of 10 levels, with Claude 3.5 Sonnet being the only model able to solve 8. For Claude and GPT, models of the same vendor with a higher number of parameters were able to solve more levels. For the Llama models, the 3.1 version solved one more level than the 3.3 model. In case of Gemini and Mistral, both the small and large models performed about the same. Tested models were mostly successful in interpreting existing rules. As can be seen, some levels are rarely solved by any model. Creating or destroying rules and thus modifying the logic of our game world has proven difficult for all models. Many models failed in solving levels 2 and 3, which require rule creation, and levels 9 and 10, which require a rule's destruction to finish the puzzle. Slight differences in the observed success rate could be due to the low number of repetitions per level, resulting in sampling errors: levels 2, 6, 9, and 10, which are rarely solved at all, could be affected by this. Chain of thought prompting [51] may help in overcoming these more complex planning tasks.

[11]https://github.com/ADockhorn/Keke-AI-PY

E. Procedural Content Generation

PCG is a widely studied area in game research [52], [53]. In this experiment, we explored whether LLMs can synthesize

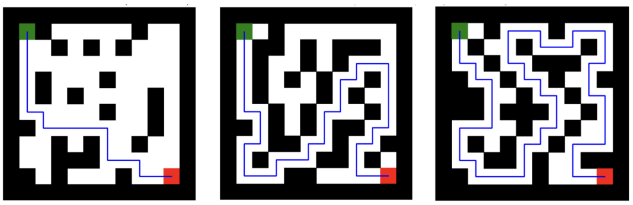

Fig. 5: Three generated mazes aiming to optimise the longest shortest path objective. Left: (score 18) example from a simple LLM generated algorithm setting wall cells with a fixed probability. Middle: (score 38) example a more sophisticated LLM algorithm involving recursion and a shortest path algorithm. Right: (score 54) example from an evolutionary algorithm directly optimising for the objective function.

Python functions capable of generating diverse game content. To assess this in a simple scenario, we tasked the LLMs with creating functions that produce random mazes adhering to specific design objectives.

The prompt advised the LLMs to use the longest shortest path objective to guide the maze generation process. This objective encourages intricate and interesting mazes. Most of the generated code ignored the hint and instead coded overly simple algorithms, placing corridors and walls in each cell with a given probability while usually ensuring that the start and end points were not on wall cells. An example generated maze is shown in the left of Figure 5. Occasionally, a better algorithm was produced that mixed randomness, recursion and graph search in ways we have not fully analysed. These algorithms sometimes produced mazes with no path between start and end, resulting in a score of -1. When they worked, they often produced reasonable mazes such as the one shown in the middle of Figure 5. The LLMs failed to find an algorithm as effective as an evolutionary algorithm applied to directly solve the objective. A sample maze from such an algorithm is shown on the right of the figure.

Note that here we are evaluating the effectiveness of the algorithms in meeting the specified objective, which is to produce mazes with the longest shortest path between start and end. Depending on the application, this could be a poor objective to maximise, with the best mazes having a mid-ranking score, such as the central maze in Figure 5.

Table VI presents the results of the maze generation experiment. Unlike the previous tasks, smaller models perform in line with, or even outperform, their larger counterparts. Among the larger models, only GPT-4o achieves performance on par with the smaller models, although it performs slightly worse in terms of D_{avg}. Mistral Large also performs similarly to Mistral Small, but both models cannot keep up with models from other LLM providers. As with Table III, the high standard deviation of D_{avg} in this experiment supports the additional long-running experiments in Section VI.

The reasoning model o1-mini outperforms both variants of GPT-4o with respect to D_{max}, but falls short compared to GPT-4o mini in terms of average distance. Overall, most models struggle with this task, with the exception of the OpenAI models and Claude Haiku. Understanding why larger language models underperform in this domain remains an open question, but is beyond the scope of this paper.

TABLE VI: Maze generation LLM results. D_{max} is the maximum distance of the shortest path of the generated mazes returned by the best program of all trials and $D_{avg} \pm \sigma$ is the average distance with the standard deviation of the 10 programs found after all trials. Each program generates five mazes for evaluating D_{max} and $D_{avg} \pm \sigma$.

Model	D_{max}	$D_{avg} \pm \sigma$
Claude Sonnet	18.0	8.98±3.79
Claude Haiku	29.4	15.4±8.73
Gemini Flash	17.0	15.7±1.28
Gemini Lite	23.2	14.36±5.46
Mistral Large	10.8	4.2±3.3
Mistral Small	10.8	4.6±3.65
o1 mini	44.4	18.56±15.92
GPT 4o	33.4	18.48±11.66
GPT 4o mini	33.4	22.02±11.76
Llama 3.3 70B	14.8	8.3±3.75
Llama 3.1 70B	11.4	6.76±2.67

F. Python Code Evaluation

Table VII shows the summary statistics of the synthesized Python code for the Minatar, Baba is You, maze generation and vehicle driving experiments, including the costs incurred. On average, the larger models do better than their small counterpart. In the case of Llama the newer 3.3 model beats the 3.1 model in terms of the successful iterations and trials. The successful iterations are, however, quite similar for all LLMs. The Mistral models have by far the worst percentage of executable programs but are still on the 4th and 6th rank in terms of overall performance, which indicates that the percentage of executable programs are not a good proxy for selecting an LLM to generate game-related Python code. o1 mini is on average the best model on all Python domains, and is also the only reasoning model in this benchmark. This indicates that reasoning models have an advantage in generating game-specific code. A more fine-grained ranking of LLMs for each domain is presented in Table X in Section VII, where we discuss the experiments in general and also give some practical recommendations.

G. Tabletop Games Framework (TAG)

The TAG framework is a bespoke Java research framework that supports the implementation of multiplayer tabletop board games. This introduces a number of new challenges:

- The games are in general more complex than the simple one-player games in previous sections.
- They are also inherently multiplayer. As such there is implicit opponent modeling required for good play strategies. The environment is no longer a 'simple' stationary MDP, but is adversarial.
- The TAG framework has a number of local libraries and coding conventions; for example decks of cards are implemented via `Deck<>` or `PartialObservableDeck<>` parameterised classes.

TABLE VII: The overall evaluation of the synthesized Python code. Cost is the total cost of all 2600 iterations for the LLM (100 per game) (the Llama 3.1 70B model is provided for free by Google Cloud as it is currently in public preview). S.Iter. is the percentage of iterations that resulted in working code and S.Trl the percentage of trials (each consisting of 10 iterations). Only programs that returned a positive reward were considered for S.Iter and S.Trl. Exec. Programs is the percentage of all generated programs that the Python interpreter could run. For each of the four domains each LLM is ranked from 1 to 11 based on the performance of the best result for the domain. This rank is averaged and used to assign an overall Rank. Gemini Flash and Llama 3.1 70B both achieved the 7th place.

Model	Cost ($)	S.Iter	S.Trl	Exec. Programs	Rank
Claude Sonnet	80.75	63.27	80.77	83.23	2nd
Claude Haiku	17.34	61.35	76.92	86.35	5
Gemini Flash	2.56	62.62	72.31	95.85	7
Gemini Lite	1.30	61.15	71.54	94.28	10
Mistral Large	43.57	60.08	70.77	75.37	4
Mistral Small	2.06	59.65	68.85	74.7	6
o1 mini	35.87	62.88	80.0	90.93	1st
GPT 4o	36.11	62.19	76.15	92.09	3rd
GPT 4o mini	2.18	62.0	67.31	91.09	9
Llama 3.3 70B	6.86	62.08	71.15	92.34	11
Llama 3.1 70B	0.00	61.08	70.38	82.59	7

These are not likely to be present in the LLM training data, and require the LLM to generalise to unseen software architecture details. This contrasts to standard Python with common libraries of the games in earlier sections.

- The language used is now Java. Integration of all language models apart from Llama used *langchain4j*[12]. Llama experiments used the Gemini Vertex AI interface to access the Google Llama model-as-a-service[13].

Algorithm 2 was applied to 12 tabletop board games (see Table VIII) implemented in TAG. These are adversarial multi-player environments with partial observability and stochasticity and varying levels of complexity. One player count in the supported range for each game was selected for use to give an even distribution of player counts between 2 and 4.

Given the additional level of complexity of these games, and different (and often dynamic) action spaces for each game, the language models were not asked to write a full policy to play the game. Instead they were asked to write a heuristic function to estimate the value of a game state for a player. This should be close to 1.0 for a position that is a definite win, to 0.0 for a position that is a definite loss (other heuristics are possible, taking account of relative scores or ordinal positions, but this simple win/loss estimate keeps things simple). This heuristic function was then used within a search algorithm; either one step lookahead (OSLA) or Monte Carlo Tree Search (MCTS) [54].

Each of these games has very different rules and implementations in TAG. To achieve the target of a scalable system that required no hand-writing and tuning of LLM prompts for each new game, two new TAG-specific elements were implemented to augment the process:

[12]https://docs.langchain4j.dev

[13]https://console.cloud.google.com/vertex-ai/publishers/meta/model-garden

TABLE VIII: TAG results by game. S.Iter is the percentage of iterations that resulted in working code and S.Trl the percentage of Trials (each consisting of 10 iterations). P is the number of players, Best Agent records the model that won the round robin tournament. SM is the number of models that produced any working code and entered an agent in the round robin tournament. BB indicates if the best agent significantly **B**eats the **B**aseline agent (OSLA or MCTS); $\approx$ means performance matches the baseline.

Game	P	S.Iter	S.Trl	SM	Best Agent	BB
Can't Stop	3	65%	91%	11	o1-mini	Yes
Colt Express	3	19%	44%	10	Llama 3.3 70B	Yes
Connect 4	2	61%	88%	11	o1-mini	$\approx$
Diamant	4	39%	89%	11	GPT 4o mini	Yes
Dominion	3	13%	44%	7	Gemini Flash	Yes
Hearts	4	31%	65%	10	GPT 4o	Yes
Love Letter	3	19%	37%	9	o1-mini	$\approx$
Poker	4	18%	50%	11	GPT 4o mini	Yes
Seven Wonders	4	25%	40%	7	GPT 4o	Yes
Sushi Go!	4	30%	58%	10	Gemini Flash	Yes
Tic-Tac-Toe	2	67%	87%	11	No differences	Yes
Virus	2	29%	55%	8	Gemini Flash	Yes

1) Automatic extraction of the game-specific APIs. This uses Java Reflections to extract information on the methods and associated Javadoc on the game state object. The entry point for this is the Class name of the main game state. All public information gathering methods on this are extracted (defined as names matching on either `get*(...)` or `is*(...)`. APIs for any class dependencies on these methods, as parameters or return values, are also extracted and this recurses until the core java libraries are reached (these are excluded).
2) Automatic rulebook digestion. This takes as input the text in the game rulebook. An approach inspired by [55] is used. The PDF rulebook is first broken down into chunks of 1000 or 2000 words (so any icons or pictures are excluded). The LLM is then given each chunk in turn and asked to summarise in 200 words or less the information about the game rules. This final set of synopses is then fed to the LLM with a prompt to, 'Summarise this information in 500 words or less.'. This provides blocks of text to include in the prompt used in the main loop of Algorithm 2 that explains the rules of the game. This rulebook digestion was done *once* for each game using the GPT 4o mini model. This ensures the comparison between models is based only on their heuristic writing ability, and they all use the same prompt.

These new tools enable a scalable and game-agnostic process to be run on all games. The input for each game is the game rulebook as a PDF file, and a Java Class name for the main game state. Additionally, the methods on the main game state were briefly reviewed for meaningful Javadoc comments, public visibility and name convention to ensure that they were picked

up by the automated API process. An example full prompt (for Sushi Go!) is included in the code repository.

The multi-player nature of these environments also necessitates a change in the evaluation criterion in Algorithm 2. Evaluation used a tournament of 500 games between the new agent and a base agent. The base agent was either a one step lookahead (OSLA) agent (for Tic-Tac-Toe and Virus) or a vanilla MCTS agent (all other games) with a budget of 10ms and a rollout of 10 actions before the generated heuristic function is applied to estimate the value of the state. TAG uses Multiplayer-UCT with the heuristic applied for each player independently after a rollout of 10 actions [56]. This small budget enables the large number of experiments to be run in a reasonable time, but will not give the best possible players, although we do get perfect play in Tac-Tac-Toe at about 30ms with no heuristic. It remains an open question how this might change with a larger budget, but what is important here is the comparative performance of the generated heuristics.

A base opponent used the same OSLA or MCTS settings and a heuristic function of the game score normalised to [0, 1]. Tic-Tac-Toe and Connect 4 do not have scores, and the base opponents for these rewarded a win (+1) or draw (+0.5), with 0 for a loss. To avoid overfitting to a specific opponent all previous (working) agents to the evaluation tournament are added to later iterations within a trial. The evaluation score of each generated heuristic is the win rate from the most recent tournament, so this includes a broader range of opponents later in the trial. This performance metric means that, unlike the single-player environments, the scores from each run are not directly comparable. This modifies lines 12-17 of Algorithm 2; EVALUATEFITNESS returns the current best-performing of the agents, and this is retained as the new *bestResult*. Each *trial* is restarted with just the base opponent.

For each game a final tournament of 25,000 games is run between the best agents from each model, for a maximum of 11 participants if all models generated at least one heuristic that compiled and executed successfully. Points in Table IX are awarded for 1st through 10th places in this final tournament for each game. This methodology compares the results of the models directly and rewards an LLM that produces a variety of heuristics, including some that are very good at the game, over one that reliably generates valid, but near-identical code each time.

Table IX summarises the results by language model. The Gemini 2.0 Flash model does best overall in the final tournament, generates valid code consistently and is much cheaper than the OpenAI models that also do very well. The older Mistral models do least well on these tasks on all measures; the larger Llama model is best at generating working code, but the performance of its best agents is relatively poor; the Anthropic models are much more expensive than the others, but this is not reflected in performance levels. The reason the smaller Gemini model is more expensive (it was 25% cheaper per token) is less working code was generated (S.Iter) so there were many more calls back to the model to fix compilation errors.

Large models do better on average than their smaller counterpart in terms of both the number of successful iterations, trials and in the quality of the best heuristics produced. However, in most cases the performance differences are small and this effect is smaller than the differences between the model families. GPT 4o mini is a fifth of the cost of GPT 4o, but still writes the best agent in 2 games, and is only a few points behind in tournament points.

TABLE IX: TAG results by model. S.Iter and S.Trl are the same as for Table IX, plus FG is the number of games for which the LLM failed to produce any working code. Cost is the total cost of all 1200 iterations (100 per game) on the LLM. For each game in a round-robin tournament between the agents, 10 Points are given for the first place, 9 for 2nd, and so on down to 1 for 10th place. Zero points are awarded otherwise, including for LLMs that failed to produce any working code for a game. The Points column is the total of these, and Rank is the ranked order of models by points.

Model	S.Iter	S.Trl	FG	Cost ($)	Points	Rank
Claude Sonnet	17%	82%	0	134.47	61	6
Claude Haiku	8%	50%	0	86.71	36	9
Gemini Flash	54%	78%	0	1.60	84	1st
Gemini Lite	28%	51%	0	2.93	63	5
Mistral Large	16%	34%	2	19.73	26	10
Mistral Small	17%	30%	7	3.24	20	11
o1-mini	58%	75%	1	23.92	69	3rd
GPT 4o	41%	62%	1	26.01	72	2nd
GPT 4o mini	32%	65%	1	4.99	67	4
Llama 3.3 70B	63%	82%	2	3.51	48	7
Llama 3.1 70B	49%	65%	1	0.00	41	8

The fact that many trials fail to produce any working code over 10 iterations show the importance of re-starts, as using a single trial for each game led to more random results.

Overall results by game are shown in Table VIII. One common reason for failure of an iteration was code that compiled but then failed to execute in all edge cases due to poor error checking for division by zero (throwing a runtime error during the evaluation tournament was counted as a failure of the iteration). There was no clear pattern of performance improvement across the 10 iterations of each trial. In some cases the later heuristics were better than the first attempts, but equally often the overall winner was the first heuristic found and later changes did not improve performance. The benefit of running more iterations for improved performance is investigated further in Section VI. The complexity of API to an LLM is not always the same as complexity of a game.

Otherwise the models were often creative in their invention of undocumented API methods causing compilation to fail. The game with the highest trial success rate, Can't Stop, is also the only game for which the base game state has no dependencies outside the core `java.lang` and `java.util` libraries, reducing the opportunities for LLMs to hallucinate about non-existent methods.

In contrast the card games in the set cause much more LLM code to fail to compile. All the 8 games in Table VIII with *S.Iter* below 35% use decks of cards, and none of the 4 games with *S.Iter* above this do. The TAG-specific infrastructure of `Deck<>` and `PartialObservableDeck<>` parameterised

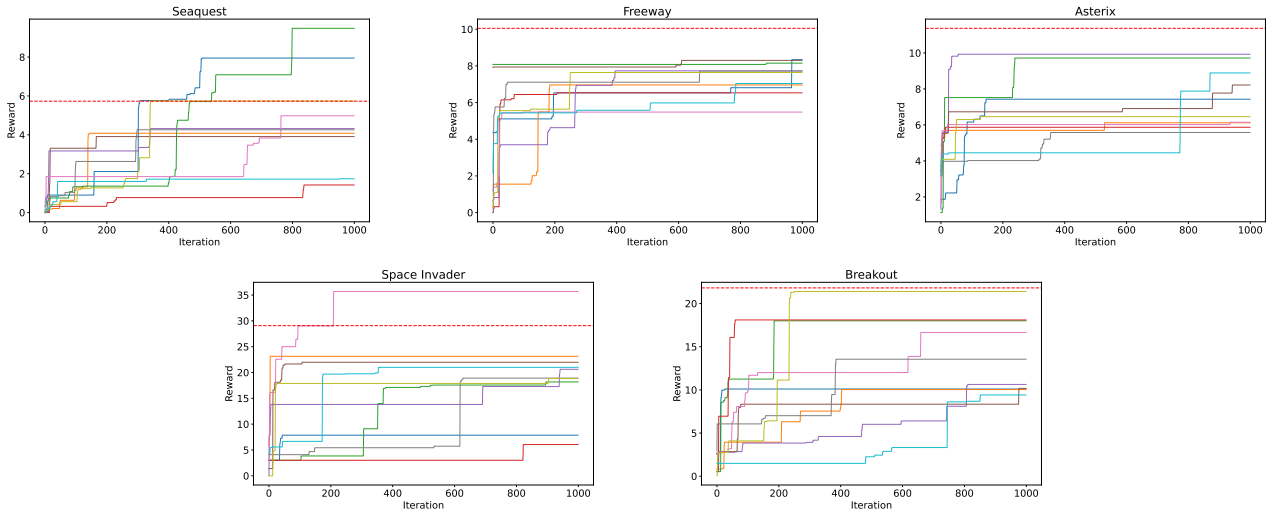

Fig. 6: The reward curves for each Minatar environment for the long-running experiments. The x-axis shows the number of iterations and the y-axis the reward of the best program. The dashed red-line shows the max. reward achieved by the best LLM from Table II. We use the same parameters as in Section V-A, only with 1000 iterations instead of 10 per trial.

classes, with their own interfaces often caused problems. For example `Deck.getSize()` is used to return the number of cards, and despite this being mentioned in the prompt through recursion of the class hierarchy, many LLMs tried to call the non-existent `Deck.size()` or `Deck.length()`. A similar problem in Sushi Go! was that the card types in the game were represented by an `enum` that had values expressed in lower case; `Maki, Sashimi, Dumpling`. Despite these values being clearly stated in the Java API section of the prompt generated code more commonly used `MAKI, SASHIMI, DUMPLING`. This is presumably because an upper case convention is more standard across Java more generally and hence in the training data of the models.

Table VIII, column **BB**, shows that at least one of the LLMs wrote a heuristic that could reliably beat the baseline score heuristic in 10 of the 12 games, and was at least competitive in the other two. Running just one trial per game per LLM (10 iterations) does not reliably get baseline performance, and restarting is needed to get a good end result from multiple independent attempts.

VI. A Thousand Iterations for Minatar

To evaluate the impact of the number of iterations on program search, we perform 1000 iterations for each game available in Minatar. We keep the same values for the other parameters, i.e. 10 trials for the program search and three attempts for generation/improvement or repair of non-executable programs. Figure 6 shows the reward curves for each environment for the long-running experiments. In most games (with the exception of Freeway and Asterix) there is a high variance in the rewards achieved in each trial, suggesting that more restarts are better than running for more iterations. The reward curves for Freeway support this statement, since in some trials, the programs found after early iterations are almost as good as after the last iterations. It is also clear that the max. reward of the best trial achieves a better result for Seaquest and Space Invader and approaches the same reward for Breakout as the best LLM from Table II. For Freeway, the LLM is not able to improve the reward compared to the 10 iterations experiment, while for Asterix the reward is almost doubled but still not as good as the best LLM from the previous experiments.

TABLE X: The ranking of the LLMs on the different domains with the overall ranking of all experiments. It is clear there are major differences between the domains.

LLM	Minatar	Maze	Baba	Vehicle	TAG	Overall
Claude Sonnet	3rd	6	1st	4	6	3rd
Claude Haiku	7	4	7	5	9	6
Gemini Flash	4	7	7	7	1st	4
Gemini Lite	11	5	7	9	5	10
Mistral Large	6	10	3rd	1st	10	5
Mistral Small	9	10	3rd	2nd	11	9
o1 mini	1st	1st	2nd	6	3rd	1st
GPT 4o	2nd	2nd	3rd	8	2nd	2nd
GPT 4o mini	5	2nd	11	10	4	6
Llama 3.3 70B	8	8	7	11	7	11
Llama 3.1 70B	10	9	3rd	3rd	8	8

VII. Practical Recommendations & Discussion

Table X shows a global ranking of the LLMs for the different domains. It is clear that there are major differences between the model families; for example, Mistral models are best for vehicle driving, while the OpenAI models are best for generating mazes and the Minatar domain. o1 mini is overall the best model, which is expected since it is the only reasoning model, and thus uses more compute than the others. For generating code in Java for TAG, Gemini 2.0 Flash is the best model followed by the GPT model family. A rather unexpected result is that Llama 3.3 is worse than Llama 3.1, although Meta advertises that it achieves a performance comparable to the Llama 3.1 405B model. In general, the open-source Llama models cannot

keep up with the closed-source models of the LLM providers. However, the reasons for this are difficult to analyse as we do not know the parameter size of the paid models and there are also differences in the training pipeline and dataset. The 70B-Llama models are probably larger than the small closed-source models, but much smaller than the large models. With enough iterations and restarts, the 70B model of Llama 3.1 achieves the same performance as the large models of OpenAI and Claude, but only with 100 times more iterations. This indicates that it is in principle possible to achieve the same performance with open-source models, they just need more time to find good programs.

After discussing the game applications and the long-running experiments, there are several practical recommendations we can make for using LLMs to synthesize programs for game research. These depend primarily on the available financial resources, hardware and time.

If money is no issue, it is best to try different models and to start with the larger models as good programs are found more quickly. With financial constraints, but local hardware to run the Llama 3.1 or 3.3 70B model, e.g. on a 48GB GPU with 4-bit quantization [57], then it is better to run local models for more iterations, as a similar reward to the Claude or OpenAI LLMs can be achieved. However, depending on the simulation environment, this may take much longer.

If there are financial and time constraints, we recommend using Table X to select the best models for a given domain.

One unexpected finding is that larger models are not always better, e.g. in the maze generation experiments, where most smaller models were better than their larger counterpart. Therefore, smaller models are worth trying and should not be dismissed from the start. They can also be much more cost-efficient overall even if they are run for many more trials/iterations. The results in Table IX make clear that total model costs can vary by 2 orders of magnitude with the cheaper model giving better results on a particular problem.

From the long-running experiments it is clear that more restarts are better than always running more iterations. The number of iterations depends heavily on the problem domain, and more difficult problems also need more iterations. This is visible in Figure 6 for Seaquest, the most complicated Minatar domain, where the reward increases only in later iterations compared to Freeway or Asterix.

VIII. Conclusion

In this work we studied and evaluated the current possibilities of using LLMs for program search in the area of games for various applications. Previous work was mostly limited to a single problem or game without being easily transferable to other domains, as the DSL had to be adapted. We demonstrated that LLMs can overcome the problem of combinatorial explosion of search spaces constructed with predefined DSLs, and that LLMs are able to synthesize programmatic policies in Python for the Minatar domain, which was not possible with a custom DSL and previous methods. Furthermore, we have shown that this framework can be easily adapted to different applications by modifying the prompts, and that it often provides reasonable results even without much customization. We have shown that even with the default temperature settings on these standard language models there is a very wide range of output for the same input prompt; in this respect at least the models can be quite 'creative'. Running many independent iterations of the same task can create a varied population of outputs. This is very promising as it provides the variation required for the hill-climbing approach used here.

We observed limitations in the quality of the generated code. For example, in the simple 2D vehicle driving task, the generated code drove the car to the target but then failed to stop most of the time. Much of the generated code fails to achieve any reward at all, or in the case of Java, to compile. These limitations become more evident as the complexity of the task increases. The need to use framework-specific Java libraries in TAG leads to less than 1 in 5 attempts generating valid code. We believe limitations such as this could be overcome with more sophisticated search and better prompt engineering, but the results so far give an idea of the limitations of what can be achieved with relatively little effort. The addition of tools to the LLM interfaces and a more agentic workflow is a promising area for this future work. For example instead of asking the LLM to generate the code in one pass, it could be asked to construct useful component functions or sub-modules with documented interfaces. In a later pass the model could then be asked to combine these sub-modules (based on feedback of performance of previous combinations).

IX. Acknowledgements

This work stems from a working group in the Dagstuhl Seminar 24261 (Computational Creativity for Game Development, 2024) and it was supported by the EPSRC Centre for Doctoral Training in Intelligent Games & Games Intelligence (IGGI) (EP/S022325/1). Additionally, the research collaboration was facilitated by COST Action CA22145 - GameTable, supported by COST (European Cooperation in Science and Technology).

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, others, and W. Zaremba, "Evaluating large language models trained on code," 2021, arXiv preprint.

[2] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[3] O. Polozov and S. Gulwani, "Flashmeta: A framework for inductive program synthesis," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2015, pp. 107–126.

[4] E. Butler, K. Siu, and A. Zook, "Program synthesis as a generative method," in *Proceedings of the 12th International Conference on the Foundations of Digital Games*, August 2017, pp. 1–10.

[5] E. Butler, E. Torlak, and Z. Popović, "Synthesizing interpretable strategies for solving puzzle games," in *Proceedings of the 12th International Conference on the Foundations of Digital Games*, August 2017, pp. 1–10.

[6] T. Silver, K. R. Allen, A. K. Lew, L. P. Kaelbling, and J. Tenenbaum, "Few-shot bayesian imitation learning with logical program policies," in *Proceedings of the AAAI Conference on Artificial Intelligence (Vol.* No. 06: 34, April 2020, pp. 10 251–10 258.

[7] J. R. H. Mariño, R. O. Moraes, T. C. Oliveira, C. Toledo, and L. H. S. Lelis, "Programmatic strategies for real-time strategy games," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 1, pp. 381–389, May 2021.

[8] M. Kreminski and M. Mateas, "Opportunities for approachable game development via program synthesis," in *AIIDE Workshops*, 2021.

[9] M. Cook, "Software engineering for automated game design," *2020 IEEE Conference on Games (CoG)*, pp. 487–494, 2020.

[10] G. Todd, A. G. Padula, M. Stephenson, E. Piette, D. J. N. J. Soemers, and J. Togelius, "GAVEL: Generating games via evolution and language models," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[11] C. Hu, Y. Zhao, and J. Liu, "Game generation via large language models," in *2024 IEEE Conference on Games (CoG)*. IEEE, 2024, pp. 1–4.

[12] A. Anjum, Y. Li, N. Law, M. Charity, and J. Togelius, "The ink splotch effect: A case study on chatgpt as a co-creative game designer," in *Proceedings of the 19th International Conference on the Foundations of Digital Games*, May 2024, pp. 1–15.

[13] S. Hu, Z. Huang, C. Hu, and J. Liu, "3d building generation in minecraft via large language models," in *2024 IEEE Conference on Games (CoG)*. IEEE, 2024, pp. 1–4.

[14] M. Liu, C.-H. Yu, W.-H. Lee, C.-W. Hung, Y.-C. Chen, and S.-H. Sun, "Synthesizing programmatic reinforcement learning policies with large language model guided search," in *The Thirteenth International Conference on Learning Representations*, 2025.

[15] H. Tang, D. Key, and K. Ellis, "Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment," *Advances in Neural Information Processing Systems*, vol. 37, pp. 70 148–70 212, 2024.

[16] R. D. Gaina, M. Balla, A. Dockhorn, R. Montoliu, and D. Perez-Liebana, "Design and implementation of tag: a tabletop games framework," 2020, arXiv preprint.

[17] K. Young and T. Tian, "Minatar: An atari-inspired testbed for thorough and reproducible reinforcement learning experiments," 2019, arXiv preprint.

[18] M. Charity and J. K. A. I. Togelius, "Competition: Solving puzzle levels in a dynamically changing mechanic space," in *2022 IEEE Conference On Games (CoG)*, 2022, pp. 570–575.

[19] A. Cropper, R. Evans, and M. Law, "Inductive general game playing," *Machine Learning*, vol. 109, pp. 1393–1434, 2020.

[20] C. Hocquette, A. Niskanen, R. Morel, M. Järvisalo, and A. Cropper, "Learning big logical rules by joining small rules," 2024, arXiv preprint.

[21] C. Hocquette, A. Niskanen, M. Järvisalo, and A. Cropper, "Learning mdl logic programs from noisy data," in *Proceedings of the AAAI Conference on Artificial Intelligence (Vol.* No. 9: 38, March 2024, pp. 10 553–10 561.

[22] R. Evans, M. Bošnjak, L. Buesing, K. Ellis, D. Pfau, P. Kohli, and M. Sergot, "Making sense of raw input," *Artificial Intelligence*, 2021.

[23] Y. Gu, Q. Liu, Z. Li, and K. Zhang, "Knowpc: Knowledge-driven programmatic reinforcement learning for zero-shot coordination," 2024, arXiv preprint.

[24] M. Eberhardinger, J. Maucher, and S. Maghsudi, "Learning of generalizable and interpretable knowledge in grid-based reinforcement learning environments," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (Vol.* No. 1: 19, October 2023, pp. 203–214.

[25] D. S. Aleixo and L. H. Lelis, "Show me the way! bilevel search for synthesizing programmatic strategies," in *Proceedings of the AAAI Conference on Artificial Intelligence (Vol.* No. 4: 37, June 2023, pp. 4991–4998.

[26] R. O. Moraes, D. S. Aleixo, L. N. Ferreira, and L. H. Lelis, "Choosing well your opponents: how to guide the synthesis of programmatic strategies," in *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, August 2023, pp. 4847–4854.

[27] R. O. Moraes and L. H. Lelis, "Searching for programmatic policies in semantic spaces," 2024, arXiv preprint.

[28] Q. A. Sadmine, H. Baier, and L. Lelis, "Language models speed up local search for finding programmatic policies," *Transactions on Machine Learning Research*, 2024.

[29] D. Robilliard and 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*, 2016.

[30] N. R. Sturtevant and A. M. White, "Feature construction for reinforcement learning in hearts," in *Computers and Games*, H. J. van den Herik, P. Ciancarini, and H. H. L. M. J. Donkers, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 122–134.

[31] G. Martinez-Arellano, R. Cant, and D. Woods, "Creating ai characters for fighting games using genetic programming," *IEEE transactions on computational intelligence and Ai in games*, vol. 9, no. 4, 2016.

[32] S. E. Gaudl, "A genetic programming framework for 2d platform ai," 2018, arXiv preprint.

[33] C. Olson, L. Wagner, and A. Dockhorn, "Evolutionary optimization of baba is you agents," in *2023 IEEE Congress on Evolutionary Computation (CEC)*, 2023, pp. 1–8, (to be published).

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

[35] D. G. Wilson, S. Cussat-Blanc, H. Luga, and J. F. Miller, "Evolving simple programs for playing atari games," in *Proceedings of the genetic and evolutionary computation conference*, July 2018, pp. 229–236.

[36] L. Wong, G. Grand, A. K. Lew, N. D. Goodman, V. K. Mansinghka, J. Andreas, and J. B. Tenenbaum, "From word models to world models: Translating from natural language to the probabilistic language of thought," 2023, arXiv preprint.

[37] J. A. Fodor, *The language of thought*. Cambridge, MA: Harvard university press, 1975, vol. 5.

[38] G. Grand, V. Pepe, J. Andreas, and J. Tenenbaum, "Loose lips sink ships: Asking questions in battleship with language-informed program sampling," in *Proceedings of the Annual Meeting of the Cognitive Science Society (Vol. 46)*, December 2023.

[39] A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri, "Programmatically interpretable reinforcement learning," in *International Conference on Machine Learning*. PMLR, July 2018, pp. 5045–5054.

[40] A. Verma, H. Le, Y. Yue, and S. Chaudhuri, "Imitation-projected programmatic reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[41] R. Das, J. B. Tenenbaum, A. Solar-Lezama, and Z. Tavares, "Combining functional and automata synthesis to discover causal reactive programs," in *Proceedings of the ACM on Programming Languages, 7(POPL)*, 2023, pp. 1628–1658.

[42] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, "Voyager: An open-ended embodied agent with large language models," *Transactions on Machine Learning Research*, 2024.

[43] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar, "Eureka: Human-level reward design via coding large language models," in *The Twelfth International Conference on Learning Representations*, 2024.

[44] N. van Stein and T. Bäck, "Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics," *IEEE Transactions on Evolutionary Computation*, 2024.

[45] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, others, and A. Fawzi, "Mathematical discoveries from program search with large language models," *Nature*, vol. 625, pp. 468–475, 2024.

[46] S. Chaudhuri, K. Ellis, O. Polozov, R. Singh, A. Solar-Lezama, Y. Yue *et al.*, "Neurosymbolic programming," *Foundations and Trends® in Programming Languages*, vol. 7, no. 3, pp. 158–243, 2021.

[47] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, others, and B. McGrew, "Gpt-4 technical report," 2023, arXiv preprint.

[48] Anthropic, "The claude 3 model family: Opus, sonnet, haiku."

[49] G. Team, P. Georgiev, V. I. Lei, R. Burnell, L. Bai, A. Gulati, G. Tanzer, D. Vincent, Z. Pan, S. Wang *et al.*, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," 2024, arXiv preprint.

[50] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, others, and R. Ganapathy, "The llama 3 herd of models," 2024, arXiv preprint.

[51] J. Wei, X. Wang, D. Schuurmans, M. Bosma, brian ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, Eds., 2022.

[52] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural content generation in games*, 2016.

[53] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, others, and J. Togelius, "Procedural content generation via machine learning (pcgml)," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, 2018.

[54] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *International conference on computers and games*. Springer, 2006, pp. 72–83.

[55] Y. Wu, S. Y. Min, S. Prabhumoye, Y. Bisk, R. Salakhutdinov, A. Azaria, T. Mitchell, and Y. Li, "SPRING: Studying papers and reasoning to play games," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[56] N. Sturtevant, "An Analysis of UCT in Multiplayer Games," *ICGA Journal*, p. 14, 2008.

[57] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in neural information processing systems*, vol. 36, pp. 10 088–10 115, 2023.