3.8 LLM-based Program Search for Games

Manuel Eberhardinger (Hochschule der Medien – Stuttgart, DE), Duygu Cakmak (Creative Assembly – Horsham, GB), Alexander Dockhorn (Leibniz Universität Hannover, DE), Raluca D. Gaina (Tabletop R&D – London, GB), James Goodman (Queen Mary University of London, GB), Amy K. Hoover (NJIT – Newark, US), Simon M. Lucas (Queen Mary University of London, GB), Setareh Maghsudi (Ruhr-Universität Bochum, DE), and Diego Perez Liebana (Queen Mary University of London, GB)

Before the advent of large language models (LLMs) for code [1], program synthesis was considered a difficult problem due to the combinatorial explosion of the search space [2], and so most solvable tasks were based on simple string manipulations or list sorting problems in a predefined domain-specific language (DSL) [3]. Program synthesis for games was also limited to simple problems with a well-defined search space, which was only feasible by incorporating high-level concepts of the game into the DSL [4, 5, 6].

Considerations on the use of program synthesis with higher programming languages such as Python or Java for games research were rarely made and only possibilities were outlined [7] or it was evaluated how to bring automated game design systems from game description languages to the use of programming languages [8].

Recently, methods for LLM-based program search for the automatic design of playable games based on program code [9, 10, 11] or game content based on JSON representations [12] have been presented. In addition, LLMs are also adapted for synthesizing programmatic policies in Python, which are then converted into a DSL usable in the given environment [13] or for building a world model based on python code, approximating the reward and state transition functions for simple games, which are then used for generating an action plan [14].

In this working group, we explore the possibilities of LLM-based program search for a broader range of applications for games without relying on a predefined specification such as a DSL, e.g. Ludii [9], the video game description language [10] and Karel [13], or a predefined converter for JSON [12]. The goal is that LLMs synthesize program code that is directly usable without further transformation or prior specification. We evaluate our approach on different domains in one of two programming languages, Python and Java. In Python, programmatic agent policies and functions for PCG are synthesized. In Java, the framework is included into TAG, a tabletop games framework, where heuristics for board games are designed [16].

3.8.1 Framework

The general framework is based on an evolutionary hill-climbing algorithm where the mutations and the seed of the initial program are performed by an LLM [13, 15]. Figure 10 displays the whole framework. We start by generating a task prompt to obtain an initial Python or Java function, which is then executed in a safe environment in a subprocess. This ensures that the main process can terminate the function after a certain time period, preventing the synthesized code from running indefinitely. If the function has been executed successfully, the task prompt is updated with the evaluation metric achieved and some additional information, depending on the environment, e.g. the action trace of the executed function. If an error occurs, e.g. the code cannot be parsed due to incorrect syntax, runtime errors due to incorrect indexing of arrays or similar problems, the description of the error is

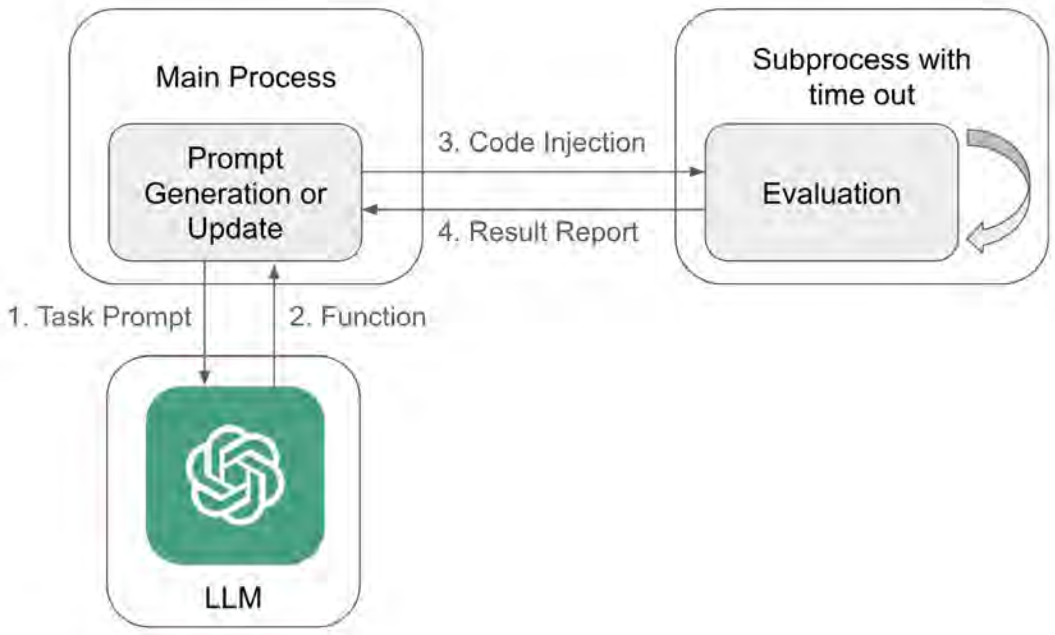


Figure 10 The general framework for the program search. At the beginning, an initial prompt is generated, which is then processed by the LLM and returns a function. Subsequently, the returned function is evaluated in a sub-process and the results are reported back to the main process, where the prompt is updated and then returned to the LLM or the evaluation criteria is met.

used to update the task prompt. These steps are repeated iteratively until the evaluation criteria, the fitness function, for the problem domain is fulfilled or the specified number of generations is reached. The individual steps are summarized in Algorithm 1.

Algorithm 1 The algorithm for the framework.

```
$prompt \leftarrow get\_task\_prompt()$
$p \leftarrow query\_llm(prompt)$
$r \leftarrow inject\_and\_run\_code(p)$
$f \leftarrow evaluate\_fitness(r)$
while not $fulfilled\_criterion(f)$ do
    $prompt \leftarrow update\_task\_prompt(r, f)$
    $p \leftarrow query\_llm(prompt)$
    $r \leftarrow inject\_and\_run\_code(p)$
    $f \leftarrow evaluate\_fitness(r)$
end while
return $p$
```

3.8.2 Game Applications

Since our framework is independent of the used LLM, we use Llama 3.1 [19] or ChatGPT based on GPT-4 [20] in the following experiments.

3.8.2.1 Programmatic Polices: 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 10x10 grid, with each

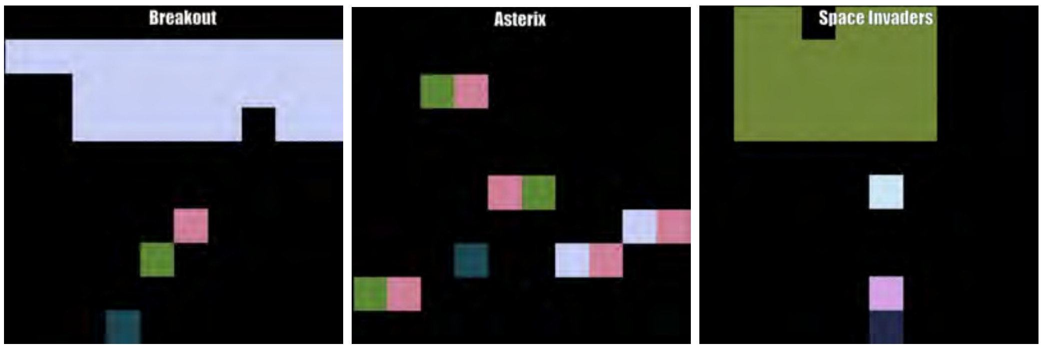


Figure 11 The three miniature versions of Atari games, Breakout, Asterix and Space Invaders, which are used for the synthesis of programmatic policies.

grid cell representing an object such as walls, enemies or the agent. Previously, Minatar was used in [18] to explain the behavior 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. For all Minatar experiments Llama 3.1 [19] with a search budget of 200 iterations, i.e. 200 programs, with a total of five refinements for each program, is used. The agent only receives the first state of the game with a description of the available objects of the state, actions of the environment and a description of the game, that is taken from Young and Tian [17]. In these experiments we use three Minatar environments, Breakout, Asterix and Space Invaders, which are shown in Figure 11.

Breakout is a game where the goal is to destroy all the bricks with the ball by controlling the paddle to bounce the ball each time before it goes out off the screen. With each destroyed brick the agent receives a reward of one. Listing 1 shows the best synthesized program.

The average reward in the final evaluation was 9.06 out of 1000 test runs, which is comparable to the DQN network with a replay buffer trained on this problem [17]. The best run achieved a reward of 103, i.e. more than three runs of the game without defeat.

Listing 1 Best program found to play Breakout, which returns an average reward of 9.06 of 1000 runs.

```python
import numpy as np

def policy(state):
    state = np.array(state)
    ball_position = np.where(state == 'ball')
    paddle_position = np.where(state == 'paddle')

    if ball_position[1].size > 0 and paddle_position[1].size > 0:
        ball_x = ball_position[1][0]
        paddle_x = paddle_position[1][0]

        if ball_x < paddle_x:
            return 'left'
        elif ball_x > paddle_x:
            return 'right'
        else:
            return 'noop'
    else:
        # If the ball or paddle is not found,
        # just return 'noop' for now
        return 'noop'
```

Asterix is a game where the goal is to control the player across the screen and collect gold while avoiding enemies. The player gets one reward for each collected gold and the game is over when the player comes in contact with an enemy. The synthesized code is 60 lines long (omitted here for brevity). The best run of 1000 test runs achieved an reward of 47 while the average reward of all test runs was 5.01. This is also comparable to the worst DQN network without a replay buffer from the baselines [17].

Space Invaders is a game where the player controls a cannon and must shoot aliens while dodging bullets launched from the alien spaceship. Additionally, the player must prevent the aliens from reaching the bottom of the screen. Listing 2 shows the best program found during the search process, which has an average reward of 20.89 when evaluating 1000 test runs, which is better to the worst neural network architecture DQN without a replay buffer and comparable to the DQN [17]. The best achieved reward was 47, where the agent almost destroy two appearing alien ships in a single episode before the aliens reached the agent.

Listing 2 Best program found to play Space Invaders, which returns an average reward of 20.89 of 1000 runs.

```python
import numpy as np

def play_space_invaders(state):
    # Get the current position of the cannon
    cannon_position = np.where(state == 'cannon')[1][0]

    # Check if there are any aliens in the current row
    alien_row = np.where((state[0:8, :] != 'empty') & (state[0:8, :] != '
alien_left') & (state[0:8, :] != 'alien_right'))

    if len(alien_row[0]) > 0:
        # Find the closest alien to the cannon
        closest_alien_position = np.min(np.abs(alien_row[1] -
cannon_position))

        # Move towards the closest alien
        if closest_alien_position < 0:
            return 'left'
        elif closest_alien_position > 0:
            return 'right'
    else:
        # If there are no aliens in the current row, move to the center
of the screen
        if cannon_position < 5:
            return 'right'
        elif cannon_position > 5:
            return 'left'

    # Check for enemy bullets above the cannon and try to avoid them
    enemy_bullets = np.where(state[:, cannon_position] == 'enemy_bullet')
    if len(enemy_bullets[0]) > 0:
        if cannon_position < 9:
            return 'right'
        else:
            return 'left'

    # If no aliens or enemy bullets are found, try to fire
    return 'fire'
```

3.8.2.2 Vehicle Driving

For this task ChatGPT [20] is used to generate a python function which should drive a vehicle from its current position to a target position as quickly as possible and remain there once it arrived. Listing 3 shows the synthesized code after a few iterations. Figure 12 shows the distance from the vehicle to the target for the corresponding time steps on the left and the 2D trajectory of the vehicle on the right. The generated code is only partially successful: while it aims for and gets close to the target, it fails to stop and instead loops around it.

Listing 3 A python function that aims to drive a vehicle to the target in 2D space - see description in the text.

```python
def get_action(vehicle: Vehicle, target: Vec2d) -> int:
    # Get the current position of the vehicle
    current_position= vehicle.s()

    # Calculate the vector from the vehicle's current position to the
    target
    target_vector = target - current_position

    # Calculate the angle between the current heading of the vehicle and
    the target vector current_heading = vehicle.heading()
    angle_to_target = math.degrees(math.atan2 (target_vector.y,
    target_vector.x)) - math.degrees(current_heading)

    # Normalize the angle to be between -180 and 180 degrees
    if angle_to_target > 180:
        angle_to_target -= 360
    elif angle_to_target < -180:
        angle_to_target += 360

    # Decide action based on the angle to the target
    if angle_to_target > 10:
        return 1 # Turn right
    elif angle_to_target < -10:
        return -1 #Turn left
    else:
        return 0 # Maintain current heading
```

3.8.2.3 Baba is You

Baba is you is a complex puzzle game in which a 2D grid environment is manipulated by the player to reach a given goal. The environment consists of word blocks and corresponding entities that can be pushed. 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 the level.

For our experiments, we used a Python version [23] of the Keke is You AI framework [24]. Similar to the other games, we prompted the LLM to provide a policy given a short description of the game and the initial state of the level. The function to be written should use the current state as input and provide movement direction.

In our tests, the agent was able to solve simple test levels as the one shown in Figure 13. The policy returned by the optimization is shown in Listing 4. Complex object manipulation to change the rules while playing a level has not occurred. This may be overcome by future versions of the used LLM models or more complex prompting techniques.

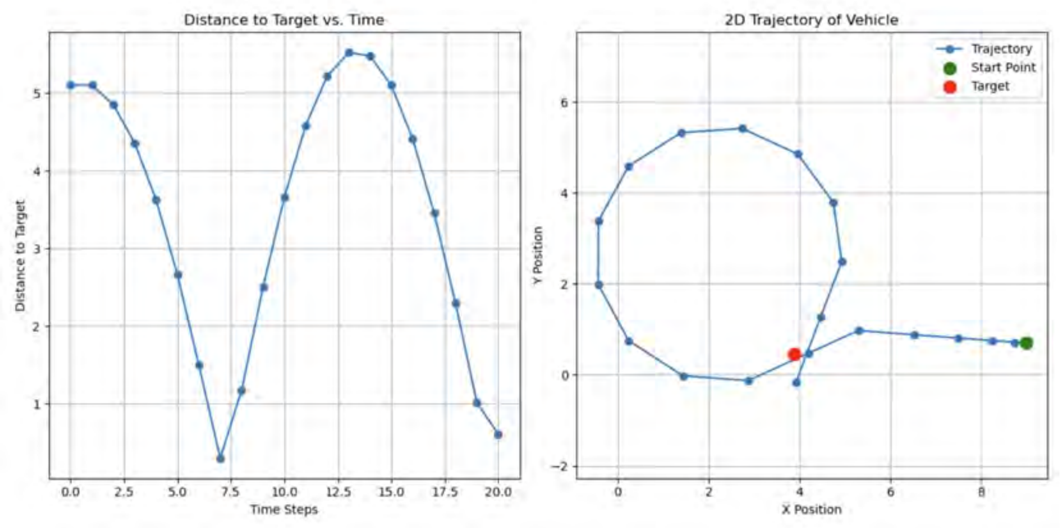


Figure 12 Left: The distance of the vehicle to the target for the corresponding time steps. Right: The 2D trajectory of the vehicle. It starts at the green point and aims for the red target. However, the generated code fails to stop the car close to the target, and instead will endlessly loop around it.

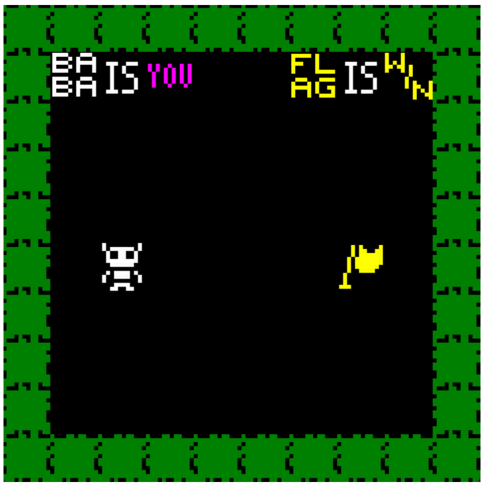


Figure 13 First demo level of the Keke AI Py framework.

Listing 4 A python function that solves the first level of the Keke AI PY framework.

```python
def program(state):
    state = state.tolist()
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 'b':
                x, y = i, j
            elif state[i][j] == 'f':
                fx, fy = i, j

    # Find the nearest word tile with the "Win" property
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] in ['B', 'b'] and \
                (i-1 >= 0 and state[i-1][j] == '3') or \
                (j+1 < len(state[0]) and state[i][j+1] == '3') or \
                (i+1 < len(state) and state[i+1][j] == '3') or \
                (j-1 >= 0 and state[i][j-1] == '3'):
                return "Right" if j > fy else "Left"

    # If no "Win" tile is found, move towards the flag
    dx = x - fx
    dy = y - fy
    if dx != 0:
        return "Up" if dx < 0 else "Down"
    elif dy != 0:
        return "Right" if dy > 0 else "Left"
    else:
        return "Wait"
```

3.8.2.4 Tabletop Games Framework (TAG)

The TAG framework is a bespoke Java research framework that supports the implementation of multiplayer tabletop board games. The ultimate goal is to use the heuristic-generation algorithm outlined in Algorithm 1 on all games in the framework. This introduces a number of new challenges:

- The games are in general more complex than the simple one-player games in previous sections.
- Related to this, 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 actively 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. These are not likely to be present in the LLM training data to any degree, and require the LLM to generalise to unseen software architecture details. This contrasts to the straightforward Python with mostly standard libraries of the game in earlier sections.

Two games were selected for initial experimentation. Tic-Tac-Toe is a simple 2-player game, and Love Letter is a slightly more complex 2-6 player game that requires reasoning over hidden information held by the other players.

The best Tic-Tac-Toe heuristic achieved a 65% win rate against a simple One Step Look Ahead (OSLA) agent, and consisted of 90 lines of code (omitted here for brevity). For Love Letter it was often difficult to get the LLM to generate valid code, let alone a heuristic that could win a game, although the best agents were able to beat random opponents.

Listing 5 illustrates the additional information needed in the prompt to obtain a working heuristic for Tic-Tac-Toe, including clear instructions not to leave TODO comments, exactly what dependencies need to be imported and details of the game-specific API that can be used to extract useful information.

Tic-Tac-Toe is an old, simple and popular game that is well-embedded in the training data. As such there was no need to explain how to play in the prompt. This was not true for Love Letter, and additional lines had to be added to explain how the game was played (and won or lost).

Listing 5 Prompt required to obtain valid code for a Tic-Tac-Toe heuristuic in TAG.

```
You are playing Tic Tac Toe.
Your job is to write the evaluation logic to help an AI play this game.
Don't leave parts unfinished or TODOs.

First, write a java class called TicTacToeEvaluator class, with only a single function with this signature:
 - public double evaluateState(TicTacToeGameState gameState, int playerId)
This is a heuristic function to play Tic Tac Toe. The variable gameState is the current state of the game, and playerId
is the ID of the player we evaluate the state for. Write the contents of this function, so that we give a higher numeric
evaluation to those game states that are beneficial to the player with the received playerId as id. Return:
  - 0.0 if player playerId lost the game.
  - 1.0 if player playerId won the game.
 If the game is not over, return a value between 0.0 and 1.0 so that the value is close to 0.0 if player playerId is close to losing,
 and closer to 1.0 if playerId is about to win the game.
Take into account the whole board position, checking for lines that are about to be completed, and possible opponent moves.
You can use the following API:
 - In TicTacToeGameState, you can use the following functions:
    - GridBoard<Token> getGridBoard(), to access the board of the game.
    - Token getPlayerToken(int playerId), to get the Token of the player passed as a parameter.
    - boolean isGameOver(), returns true if the game is finished.
    - int getWinner(), returns the Id of the player that won the game, or -1 if the game is not over.
    - int getCurrentPlayer(), returns the Id of the player that moves next.
 - GridBoard<Token> has the following functions you can also use:
   - int getWidth(), to return the width of the board.
   - int getHeight(), to return the height of the board.
   - Token getElement(int x, int y), that returns the Token on the position of the board with row x and column y.
 - Token represents a piece placed by a player. Which player the token belongs to is represented with a string. This string
   is "x" for player ID 0, and "o" for player ID 1.
   - Token(String) allows your to create token objects for any comparisons.
   - String getTokenType() returns the string representation of the token type.
Assume all the other classes are implemented, and do not include a main function. Add all the import statements required,
in addition to importing games.tictactoe.TicTacToeGameState, core.components.GridBoard and core.components.Token
```

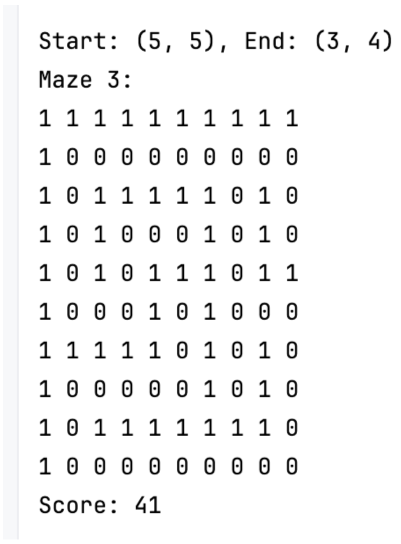
The need to hand-craft these prompts for each game does not achieve the desired scalability across the suite of games within TAG. To resolve this at the tail end of the seminar, two new TAG-specific elements were implemented to augment the process:

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. This automatically generates the section of the prompt in Listing 5 from line 15 to the end;
2. Automatic rulebook digestion. This takes as input the PDF of the game rulebook. An approach inspired by [25] in a Minecraft-like environment is used. The rulebook is first broken down into chunks of 1000 or 2000 words to fit within the input of any LLM. Each chunk is then provided in turn to the LLM and two questions asked in separate prompts:
   a. Summarise in 200 words or less the information in this text about the game rules. Do not include information on strategies to play the game.
   b. Summarise in 200 words or less the information in this text about strategies and tips to play the game well. Do not include information on the game rules.

   This generates two sets of data, one on the rules, and one on tips to play the game well (as these are often included in the rule book). Each of these sets is then fed to the LLM with a prompt to, 'Summarise this information in 500 words or less.'. This provides an additional two blocks of text to include in the prompt used in the main loop of Algorithm 1 that explain the rules of the game, and advice on how to play.

These new tools enable a more scalable and game-agnostic process to be run on all games that we plan to report results for in future work after the seminar.

3.8.2.5 Procedural Content Generation (PCG)



```
Start: (5, 5), End: (3, 4)
Maze 3:
1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 0 0
1 0 1 1 1 1 1 0 1 0
1 0 1 0 0 0 1 0 1 0
1 0 1 0 1 1 1 0 1 1
1 0 0 0 1 0 1 0 0 0
1 1 1 1 1 0 1 0 1 0
1 0 0 0 0 0 1 0 1 0
1 0 1 1 1 1 1 1 1 0
1 0 0 0 0 0 0 0 0 0
Score: 41
```

Figure 14 A maze generated with a Python function from ChatGPT, where 0s represent the path and 1s represent walls.

PCG is a widely studied area in game research [21, 22]. In this experiment, we investigated whether it is possible for an LLM to synthesize Python functions that generate diverse content that can then be used in games. To evaluate this using a simple example, we prompt ChatGPT to return functions that can generate random mazes that meet specified design objectives. Figure 14 shows a maze generated using the Python function that ChatGPT returned.

The prompt advised ChatGPT to use the longest shortest path objective to guide the maze generation process. This objective encourages intricate and interesting mazes, but ChatGPT ignored the hint. Instead, the generated code (not shown in this report) was overly simple, placing zeros and ones in each cell with a given probability while ensuring that the start and end points were not on wall cells. There are much better solutions to maze generation with the specified objective, but our program search implementation failed to find them.

3.8.3 Conclusion

In this working group, 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 also 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. 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.

For future work, we plan to extend the study and evaluate more LLMs on all domains so that deeper conclusions can be drawn about LLM-based program search for games.

References

1 Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W., *Evaluating large language models trained on code*. arXiv preprint arXiv:2107.03374, 2021.

2 Gulwani, S., Polozov, O., & Singh, R., *Program synthesis. Foundations and Trends® in Programming Languages*, 4(1-2), 1-119, 2017.

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

4 Butler, E., Siu, K., & Zook, A., *Program synthesis as a generative method*. In Proceedings of the 12th International Conference on the Foundations of Digital Games, pp. 1-10, 2017.

5 Silver, T., Allen, K. R., Lew, A. K., Kaelbling, L. P., & Tenenbaum, J., *Few-shot bayesian imitation learning with logical program policies*. In Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 34, No. 06, pp. 10251-10258, 2020.

6 Marino, J. R., Moraes, R. O., Oliveira, T. C., Toledo, C., & Lelis, L. H., *Programmatic Strategies for Real-Time Strategy Games*, 2021.

7 Kreminski, M., & Mateas, M., *Opportunities for Approachable Game Development via Program Synthesis*. AIIDE Workshops, 2021.

8 Cook, M., *Software Engineering For Automated Game Design*. 2020 IEEE Conference on Games (CoG), 487-494, 2020.

9 Todd, G., Padula, A., Stephenson, M., Piette, E., Soemers, D.J., & Togelius, J., *GAVEL: Generating Games Via Evolution and Language Models*. arXiv preprint arXiv:2407.09388, 2024.

10 Hu, C., Zhao, Y., & Liu, J., *Generating Games via LLMs: An Investigation with Video Game Description Language*. arXiv preprint arXiv:2404.08706, 2024.

11 Anjum, A., Li, Y., Law, N., Charity, M., & Togelius, J., *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, pp. 1-15, 2024.

12 Hu, S., Huang, Z., Hu, C., & Liu, J., *3D Building Generation in Minecraft via Large Language Models*. arXiv preprint arXiv:2406.08751, 2024.

13 Liu, M., Yu, C. H., Lee, W. H., Hung, C. W., Chen, Y. C., & Sun, S. H., *Synthesizing Programmatic Reinforcement Learning Policies with Large Language Model Guided Search*. arXiv preprint arXiv:2405.16450, 2024.

14 Tang, H., Key, D., & Ellis, K., *Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment*. arXiv preprint arXiv:2402.12275, 2024.

15 Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., ... & Fawzi, A., *Mathematical discoveries from program search with large language models*. Nature, 625(7995), 468-475, 2024.

16 Gaina, R. D., Balla, M., Dockhorn, A., Montoliu, R., & Perez-Liebana, D., *Design and implementation of TAG: a tabletop games framework*. arXiv preprint arXiv:2009.12065, 2020.

17 Young, K., & Tian, T., *Minatar: An atari-inspired testbed for thorough and reproducible reinforcement learning experiments*. arXiv preprint arXiv:1903.03176, 2019.

18 Eberhardinger, M., Maucher, J., & Maghsudi, S., *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. 19, No. 1, pp. 203-214, 2023.

19 Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., ... & Ganapathy, R., *The Llama 3 Herd of Models*. arXiv preprint arXiv:2407.21783, 2024.

20 Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., ... & McGrew, B., *Gpt-4 technical report*. arXiv preprint arXiv:2303.08774, 2023.

21 Shaker, N., Togelius, J., & Nelson, M. J., *Procedural content generation in games*, 2016.

22 Summerville, A., Snodgrass, S., Guzdial, M., Holmgård, C., Hoover, A. K., Isaksen, A., ... & Togelius, J., *Procedural content generation via machine learning (PCGML)*. IEEE Transactions on Games, 10(3), 257-270, 2018.

23 Dockhorn, A., *Keke AI Py*, https://github.com/ADockhorn/Keke-AI-PY, 2024

24 Charity, M. & Togelius, J., *Keke AI Competition: Solving puzzle levels in a dynamically changing mechanic space*. 2022 IEEE Conference On Games (CoG). pp. 570-575, 2022.

25 Y. Wu et al., *SPRING: GPT-4 Out-performs RL Algorithms by Studying Papers and Reasoning'*, Arxiv Preprint Arxiv:2305.15486, 2023.

3.9 Computational Creativity for Game Production: What Should Be Left Untouched?

Christian Guckelsberger (Aalto University, FI), João Miguel Cunha (University of Coimbra, PT), Alena Denisova (University of York, GB), Setareh Maghsudi (Ruhr-Universität Bochum, DE), Pieter Spronck (Tilburg University, NL), and Vanessa Volz (CWI – Amsterdam, NL)

Artificial Intelligence (AI) has been an integral part of video games for a long time, supporting both offline production (e.g. distribution of game fauna) and online features (e.g. automated difficulty adjustment). The relationship between academic AI research and industry use,