A Good Novelist Should be a Good Coder: From Language Critics to Automatic Code Generation

Stanford CS224N Custom Project

Brian Munoz Calonge
Department of Computer Science
Stanford University
`brmunoz@stanford.edu`

Mu-sheng Lin
Department of Computer Science
Stanford University
`vrlinp@stanford.edu`

Abstract

Code projects could be modeled as Reinforcement Learning (RL) embodied environments where the agents interact on behalf of the programmers. In this study, we seek to develop an automatic code-generating process with the assistance of Large Language Models (LLMs) as the task of machine translation from natural language to codes. Our innovation includes *Modularity* (to relax the limit of context length of LLMs), *Factorization* (to improve sample efficiency and to facilitate debugging), and *Language Critic* (to automate code generation robustly). These three components help the automatic code-generating process progressively and robustly with supporting evidence from both simulation results and prompt experiments via LLMs.

1 Key Information to include

- Mentors: Tathagat Verma (Internal). Fan-Yun Sun and Logan Cross (External)
- Equal contribution from both authors (research, language Critic proof-of-concept, analysis, conclusion). Brian: Work on LLM prompts. Mu-sheng: Factor graph improvements.

2 Introduction

Currently, code generation for simple tasks can be achieved by LLMs through natural language prompts (Wang et al., 2023). Even though individual scripts could be generated in seconds, for complex projects, 1) high-level abstraction among scripts and 2) dynamic interacting behaviors of variables, are still missing, meaning that the designs still need human effort involved to organize the overall projects to be consistent during the continuous development process.

Tools to automate code generation include GitHub Copilot, the commercial product Copilot (2024), or open-source alternatives, Code Llama Rozière et al. (2024), StarCoder Li et al. (2023), and DeepSeek Guo et al. (2024), just to name a few. These code-generation tools could not solve the aforementioned challenges yet. In addition, the generated codes still require manually debugging for end-to-end projects.

To solve the challenges of high-level abstraction among scripts and dynamic-interacting behaviors of variables, we propose to use LLMs together with Reinforcement Learning (RL) techniques as the approach: LLMs could understand the design goals and partition the project into feasible modules (or individual scripts) though prompts while RL helps to emulate the roles of programmers with iterative efforts in the design process (such as implementing code, debugging error, and creating new features) as the agents; the whole continuous development process of the code with dynamically-interacting behaviors of variables are represented as embodied environments.

The structure of this study is as follows: Related works of embodied environment design, language, and causality on RL and capability of scaled LLMs are briefed in Sec. 3. The approach and proposals for automatic code generation are detailed in Sec. 4. Experiments on the emulation of automated code generation are conducted through the use of virtual games with LLMs, as detailed in Sec. 5, including some prompting experiments. Analysis and possible enhancement of the proposals are discussed in Sec. 6. In Sec. 7, we conclude this study together with interesting directions for future work.

The contributions of this work are summarized as follows:

- Model code dynamics of embodied environments as POMDPs.
- Propose *Modularity* as the approach to partition the project into feasible modules by LLMs.
- Propose using LLMs as *Language Critic* with *Causal Factor Graphs* in the RL process.
- Propose *Self-encouraged Learning* under curriculum learning via detecting the progressive difficulty with the Language Critic.

3 Related Work

Here we introduce concepts that are critical for our study that are not generally associated with pure Natural Language Processing (NLP), yet in recent years have gained popularity for interdisciplinary research, as methods across different branches of Artificial intelligence (AI) exhibit properties that can be leveraged for improvement on current approaches.

3.1 Unsupervised Environment Design (UED)

Designing a high dimensional join distribution of the RL environment is difficult. Dennis et al. (2020) suggest designing an *underspecified environment* (UED), by providing the distributions of free parameters that control the features and behavior of the environment (formal definition in Sec. 4.1). Compared to their approach, we propose modularity and factorization with LLMs (expanded in Sec. 4.2) to generate the embodied environments.

3.2 Languages on Reinforcement Learning

3.2.1 Language on World Model

The world model Ha and Schmidhuber (2018) uses the autoregressive model to predict the environment's next state unsupervisedly for temporal representations. The spatial representations of the model come from their innovation of using vision representations/tokens from the variational autoencoder (VAE) Kingma and Welling (2022). Refer also to more recent implementation Hafner et al. (2023). In Lin et al. (2023), vision-token-only world model is extended with the language-conditioned world model. In our study, we use LLMs to track the interaction dynamics among modules with the factor graph, instead of learning from the autoregressive world model.

3.2.2 Language on Agent Model

In Hanjie et al. (2021), the agent's policy is formed by conditioning on the text manuals of the games (descriptions of the entities in the world and their dynamics) and the three most recent observations. Mu et al. (2022) provide agents with goal location descriptions in language to enhance and generalize the agents' policy. In our study, the agent simply follows the code domain factors generated by the LLMs (function calls) deterministically.

3.2.3 Language on Reward Model

To solve the sparse reward issue, *intrinsic rewards* is suggested (Oudeyer et al. (2007)) to encourage agents to explore the environment. Mu et al. (2022) implement the language-parameterized exploration method by adding an exploration bonus for visiting linguistically-novel states. Shinn et al. (2023) propose to control reinforced language agents with linguistic feedback and store the agents' self-reflective feedback in an episodic memory buffer. In our study, the proposed *Language Critic* (in Sec. 4.2.3) also uses language feedback but we suggest using a causal factor graph to trace back the possible errors in the environment in counterfactual ways.

3.3 Causality on Reinforcement Learning

Causality (Pearl, 2009a) relates to the cause and effect of actions taken by an agent in an environment. Following this idea, a *counterfactual* describes the actions that could have been taken by this agent which would have yielded different effects (Figure 7 in A.4). This thought process allows us to examine past events in hindsight and speculate on what could have happened if we had modified specific variables in given situations. This methodology, when used in RL, is referred to as Causal RL (Schölkopf et al., 2021), which –by explicitly integrating causality into the modeling process– offers a systematic method for overcoming intricate decision-making challenges increasing the clarity of the modeling process. Similarly, in our study, we leverage the causality ability of LLMs together with factor graphs to employ causality inference to help us in code generation (explained details of the Language Critic in Sec. 4.2.3).

3.4 Artificial General Intelligence

Andreas (2022) claims that –though with some global incoherence of generated context– sometimes LLMs could infer partial representations of the beliefs and intentions from the training example (authors' mindsets) and could causally link these concepts to their generative prediction. Bubeck et al. (2023) experiment on an early version of GPT4 that the model could perform "vision" under a pure language model, which means LLMs could "track" the interaction dynamics of described components in the provided scenes. In our study, we particularly use this "vision under language" ability together with causal reasoning to backtrace the causal factor graph (refer to the simple example in A.3 and Sec. 4.2.3 on code generation process).

4 Approach

Starting by framing the code project as the Partially Observable Markov Decision Process (POMDP) modified with parameters generated by language prompts, we propose, for the automatic code generation, three components corresponding to the following phased approach: 1) Modularity, 2) Factorization, and 3) Language Critic. The phases address different issues in incremental order when LLMs are leveraged to generate code progressively and robustly.

4.1 Partially Observable Markov Decision Process

The Partially Observable Markov Decision Process (POMDP) is defined as $(S, A, T, R, O, \Omega, \gamma)$, where S is the state space, A is the action space, T is the state transition model, R is the reward (depends on S and A), O is the set of observations, Ω is the observation function/probability, $p(o|s,a)$ and γ is the discount factor. Furthermore, the *embodied environments* can be described by the parameters, q_{text}:

$$S^{'}, T^{'}, R^{'}, \Omega^{'} = g(S, T, R, \Omega, q_{text})$$

where T, R and Ω are Turing computable functions. We leverage Large Language Models (LLMs), such as $GPT4$, to design the function g. Specifically, we aim to model the code design problem as a *machine translation* from natural language domain L to code domain C and further decompose the g into factors (or function calls) by LLMs.

4.2 Code Generation Process

The automatic code-generating process could be modeled as Reinforcement Learning (RL) with Large Language Models (LLMs). We divide the approach into the following three phases:

4.2.1 Phase 0: Modularity by LLMs

Current LLMs fall short of designing learning-efficient POMDPs end-to-end that are scalable. Here we propose to use *modularity* to assist the LLM in generating embodied environments to track the interaction dynamics among tasks under the overall design goal in high-level abstraction. One *module* is defined as the sequence or set of factors (or functions) that belong to one task. By implementing this we effectively partition the main project into modules that each has its own task. [1] For instance, one dialogue L_i to the LLM updates the set of state variables $\{i_0, i_1, \ldots, i_n, t_k\}$ to $\{i_0^{'}, i_1^{'}, \ldots, i_n^{'}, t_{k+1}\}$ in language domain (not observable), mapping (or translating) the L_i into a sequence of factors, $C_m^i, \ldots, C_1^i, C_0^i$, which updates the set of state variables $\{c_0, c_1, \ldots, c_n, t_k\}$ to $\{c_0^{'}, c_1^{'}, \ldots, c_n^{'}, t_{k+1}\}$ in code domain. The collection of the sequence of factors under one task becomes one module that participates in our generated POMPD environment in which the LLM tracks the interaction-dynamics among modules. One benefit of modularity is to overcome the context length constrained by LLMs.

Our baseline for this project is the generated embodied environment in code for the games Flappy Bird, Catcher, Snake, and Pong using a set of tailored prompts with *GPT3.5* and *GPT4*. In the environments that were generated successfully, we trained agents using Proximal Policy Optimization (PPO) and Deep Q-Network (DQN) and obtained scores based on the rewards described in Kubovčík (2023). Refer to Sec. 5.2 for more details. Notice that the agent is implemented as a fully connected network of multilayer perceptron (MLP).

4.2.2 Phase 1: Factorization by LLMs

Factorization decomposes system state variables within one module into separable factors as shown in Fig. 2 by the LLMs (LLM-driven factorization (Balaji et al., 2021) of POMDPs). Compared to the MLP implementation of the agent in Sec. 4.2.1, this approach has two benefits: 1) sample efficiency of the learning process and 2) accessibility to the debugging process. The former is to reduce the model update complexity by separating factors while the latter helps to verify the agent under the generated environment through the divide and conquer philosophy (details described in Sec. 4.2.3). The experiment results (refer to Sec. 5.4) show improvement, compared to the baselines.

4.2.3 Phase 2: Language Critic by LLMs

Pearl's Ladder of Causation

Pearl (2018) describe the "Ladder of Causation" in three layers: 1) Association, 2) Intervention, and 3) Counterfactual. Causal Hierarchy Theorem (Bareinboim et al. (2022)) proves that layers collapse with measure (probability) zero, which implies that "to answer the question in layer i, one needs knowledge at layer i or higher" (Bareinboim et al. (2022)). Here are examples from machine learning fields we think are relevant to the definition: L1: BERT style model for join distributions or purely factor graphs, L2: RL on the factor graphs, and L3: "what if" by LLMs.

Artificial General Intelligent of Scaled LLMs

Scaled LLMs, such as GPT4, show the ability of *counterfactual* (L3 category in Sec. 4.2.3). This is a crucial ability since LLMs could exploit the beliefs (Andreas (2022)) and experiences (common sense or domain-specific knowledge, Bubeck et al. (2023)) from training large corpus to avoid unnecessary simulation in traditional RL on *sample efficiency* (zero-shot in context learning).

Language Critic with Causal Factor Graph

To automate the code generation process, we model the code project as an embodied environment created by the LLMs, with the agent serving on behalf of the programmer who modifies this environment to accomplish a specific design objective, e.g. a virtual game. The *Language Critic* (by counterfactual, refer to A.4) adjusts the actions of both generated environments and the agents based on the design goal by means of LLMs (in contrast to language reward described in Sec. 3.2.3) and evaluate the actions (backward) based on the design goals (like the value functions but only short term within the context window described in Fig. 2). To be specific, as shown in Fig. 1, the language critic (implemented by using the LLMs) records the accumulated *causal factor graph* by storing active factors (with state variables from modularity metadata) from both the agent and the generated environment and produces the feedback (by counterfactual) based on a set of design goals, either from human feedback (Ouyang et al. (2022)) or predefined metrics (Dubois et al. (2024), (Si et al., 2024)), and then acts to adjust the agents and embodied environment to meet these objectives.

For instance, when the game is over (true reason that the gaps between pipes being too narrow in the generated Flappy Bird game), the agent may experiment several times, then starts tracing back,

---

[1] The concept of modularity is inspired by the design pattern of MVC (Model-View-Controller).

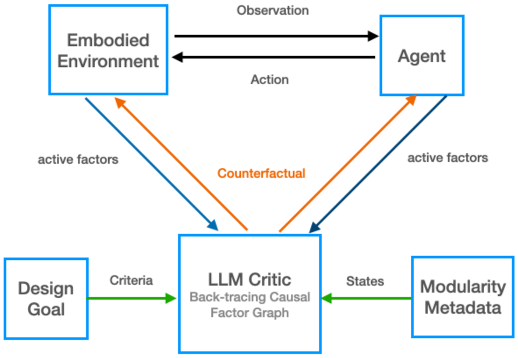


Figure 1: The System Diagram with Language Critic

as in Fig. 2, on the *causal factor graph* and eventually counterfactual reasoning on the solutions to deduce that widening the pipe gap would allow the game's character to pass through, enabling the game to proceed (Fig. 3). In *curriculum learning* setting, a Language Critic could detect progressive difficulty (we call this self-encouraged learning) by monitoring 1) design goal metric events or 2) reward change events in Fig. 2.

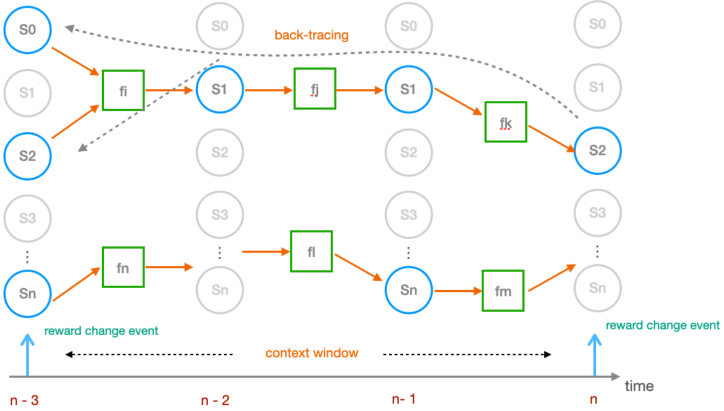


Figure 2: Back-tracing on Causal Factor Graph by LLMs

5 Experiments

5.1 Data

We generate environments by prompting the LLMs. In the first phase of our project, we train RL agents on these generated environments. Hence, the data is obtained via the direct interactions of agents with the environments of the games described in our baseline as shown in Section 4.2.1. The rest two phases use the same approach.

5.2 Evaluation method

We use different evaluation methods to determine the effectiveness and performance of our results.

5.2.1 Phase 0: Modularity Evaluation

Once the code of the game environment is produced by our baseline from prompts (GPT-3.5 and GPT-4), we simulate the agents in the Ray framework with RLlib (Liang et al., 2017) under PPO and DQN algorithms. The evaluation is done first by calculating the rewards of the agent "playing" Flappy Bird (Kubovčík (2023)).

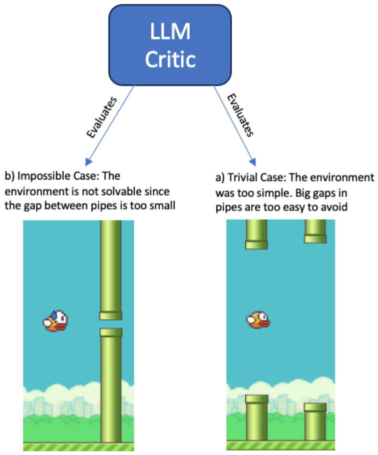


Figure 3: Self-encouraged Learning with Language Critic (Illustration)

5.2.2 Phase 1: Factorization Evaluation

Since Phase 0 only demonstrates that the RL algorithms can run under the generated environment in one particular game without showing full correctness of the generated environment, we created unit test cases that are specific to each game, in which we compare the details of the factor generated codes to the baseline codes (only with modularity). The metric is defined as the passing rate, which is the ratio of the unit tests that were successful on the generated game.

5.2.3 Phase 2: Language Critic Evaluation

We qualitatively analyze the environment using the Language Critic that we design to advance the automatic code generation process that LLMs can use causal inference to debug our codes.

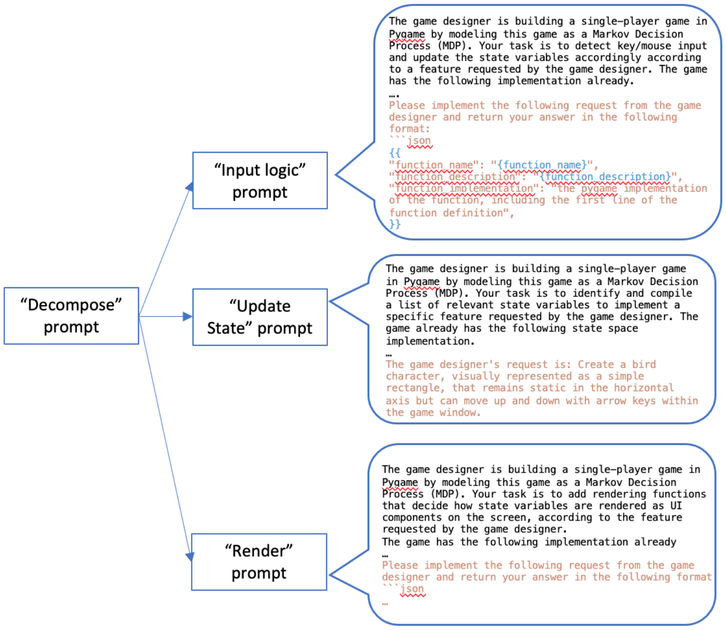
5.3 Experimental details

5.3.1 Phase 0: Modularity Experiment

1. We prepared the set of prompts with a sample structure of the embodied environment we want to generate. Specifically, we define the structure that contains 1) the *process input* prompt to the game, 2) the *logic update* prompt that handles changes in states, and 3) the *render* prompt that transforms the environment into graphics. We use a "decompose" prompt to connect the structure as shown in Figure 4
2. The prompts are processed using the GPT-3.5 and GPT-4 API. This step mainly leverages the LLMs ability for code generation in Python code.
3. The Python code containing the embodied environment structure is post-processed through a wrapper, adapted for the Gymnasium interface.
4. We trained our agent in the generated environment using PPO and DQN algorithms from the standard Ray library (RLlib), both with the same parameters: 1 GPU, 40 workers, using Adam optimizer with an initial learning rate of 0.001.

5.3.2 Phase 1: Factorization Experiment

We generated factorization code using chain-of-though (Wei et al. (2023)) with more "fine-granularity" in language. The language model could be viewed as an extrapolation process. Intuitively, by prompting step-by-step with guidance, the quality of the code for the generated environment should be better. The experiment results support this intuition in Sec. 5.4.2. To further investigate the quality of the factorization approach, we experimented with other games including Catcher, Snake, and Pong to verify that the approach could be generalized. We ran unit test cases as explained in 5.2.2.



~~~
The game designer is building a single-player game in
Pygame by modeling this game as a Markov Decision
Process (MDP). Your task is to detect key/mouse input
and update the state variables accordingly according
to a feature requested by the game designer. The game
has the following implementation already.
….
Please implement the following request from the game
designer and return your answer in the following
format:
```json
{{
"function_name": "{function_name}",
"function_description": "{function_description}",
"function_implementation": "the pygame implementation
of the function, including the first line of the
function definition",
}}
~~~

````
The game designer is building a single-player game
in Pygame by modeling this game as a Markov Decision
Process (MDP). Your task is to identify and compile
a list of relevant state variables to implement a
specific feature requested by the game designer. The
game already has the following state space
implementation.
…
The game designer's request is: Create a bird
character, visually represented as a simple
rectangle, that remains static in the horizontal
axis but can move up and down with arrow keys within
the game window.
````

~~~
The game designer is building a single-player game in
Pygame by modeling this game as a Markov Decision
Process (MDP). Your task is to add rendering functions
that decide how state variables are rendered as UI
components on the screen, according to the feature
requested by the game designer.
The game has the following implementation already
…
Please implement the following request from the game
designer and return your answer in the following format
```json
…
~~~

Figure 4: One example of the LLM input for code generation process used in this project

5.3.3 Phase 2: Language Critic Experiment

We use Language Critic to "judge" (or debug) the generated codes by back-tracing the causal factor graph. We use prompts to support the concepts that 1) LLMs could trace back on causal graphs and 2) LLMs could suggest counterfactual proposals for our experiments.

5.4 Results

5.4.1 Phase 0: Modularity Results

The agents can run in the generated embodied environment, which implies partial correctness of the code generation from the LLM. The results are shown by the max of the episodic rewards in our baseline models in Figure 5. However, when the simulations run longer, the rewards do not improve for either PPO or DQN, which made us re-evaluate our approach in this phase. After further investigation, we proposed to use factorization to enhance our code generation process instead.

5.4.2 Phase 1: Factorization Results

With the conjecture that chain-of-thought could improve the generated code quality (in Sec. 5.3.2), the code is generated by new prompts with more games simulated for generalization. The performance between the baseline and proposed factor graph approach is summarized in Table 1. The results indicate that the application of factorization boosts the passing rate of the generated code, which aligns with our prediction. This supports the idea that chain-of-thought can improve the code quality in factorization as explained previously.

Method	Flappy Bird	Catcher	Snake	Pong
GPT-3.5	0.19	0.39	0.37	0.22
GPT-4	0.35	0.35	0.57	0.25
GPT-4 plus factor graph	0.78	0.66	0.44	0.57

Table 1: The Factor Approach and Baseline

5.4.3 Phase 2: Language Critic Results

This phase is in the proof-of-concept stage. The prompt experiment that supports that the LLM (GPT4) could trace back on the causal graph is shown in A.3. The counterfactual ability of LLM (GPT4) is shown in A.5. The use of a Language Critic in this phase demonstrates a qualitative improvement is possible, suggesting that incorporating causal inference and feedback mechanisms can refine the code generation process, making it more aligned with the design objectives.

6 Analysis

The experiments demonstrated improvement in code generation through the integration of Modularity, Factorization, and Language Critic mechanisms. Specifically, the transition from GPT-3.5 to GPT-4 and then to GPT-4 enhanced with factor graphs shows a progressive performance improvement.

Some technical challenges of Language Critic design are still unsolved at the current stage: 1) Grounding the useful counterfactual proposals, e.g. A.5 would be needed. LLM's hallucination of infeasible solutions might be another concern. 2) Deciding between choosing the different actions in the action space or choosing the counterfactual proposals from LLMs could impact the sample efficiency (similar to exploration vs. exploitation). We would need to estimate which one has a higher probability of increasing the rewards instead of trying all of them to help with the decision. 3) When designing the back-trace process, the back-tracing depth and the context window length are design parameters, dependent on the design goal evaluation rate or reward change events. Another aspect is the memory required to store the causal factor graph.

7 Conclusion

Our methodical approach, progressing from Modularity through Factorization to Language Critic, demonstrates that we may significantly decrease the need for manual debugging and elevate the level of automation in code generation.

However, there are limitations to acknowledge. The variety of games tested is limited, and extending these findings to more complex or diverse settings requires additional study. Moreover, the computational requirements and the adaptability of our proposed techniques in more extensive, intricate codebases need further examination for real-world code project examples.

Looking ahead, future research should delve into applying these strategies across a wider array of programming languages and development contexts. Investigating the incorporation of more sophisticated feedback mechanisms and the examination of more advanced model architectures could further refine the process of code generation. The proposed Language Critic focuses on the dynamics design of the automatic code generation process, which could be applied to extend the static website design in Si et al. (2024) to the dynamic websites. Multi-model transformer architectures Robine et al. (2023) with visual tokens and language tokens together might further improve the back-tracing process of the causal factor graphs, for example, our pipes too close could be detected right after the scene has been rendered without the need to play the game until reaching game-over.

References

Jacob Andreas. 2022. Language models as agent models.

Bharathan Balaji, Petros Christodoulou, Xiaoyu Lu, Byungsoo Jeon, and Jordan Bell-Masterson. 2021. Factored{rl}: Leveraging factored graphs for deep reinforcement learning.

Elias Bareinboim, Juan Correa, Duligur Ibeling, and Thomas Icard. 2022. On pearl's hierarchy and the foundations of causal inference (1st edition). In Hector Geffner, Rina Dechter, and Joseph Y. Halpern, editors, *Probabilistic and Causal Inference: the Works of Judea Pearl*, pages 507–556. ACM Books.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4.

GitHub Copilot. 2024. . `https://github.com/features/copilot`. Accessed: 2024-03-11.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. 2020. Emergent complexity and zero-shot transfer via unsupervised environment design. In *Advances in Neural Information Processing Systems*, volume 33, pages 13049–13061. Curran Associates, Inc.

Yann Dubois, Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2024. Alpacafarm: A simulation framework for methods that learn from human feedback.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

David Ha and Jürgen Schmidhuber. 2018. World models.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. 2023. Mastering diverse domains through world models.

Austin W. Hanjie, Victor Zhong, and Karthik Narasimhan. 2021. Grounding language to entities and dynamics for generalization in reinforcement learning.

Diederik P Kingma and Max Welling. 2022. Auto-encoding variational bayes.

Martin Kubovčík. 2023. flappy-bird-gymnasium. `https://github.com/markub3327/flappy-bird-gymnasium/`. Accessed: 2024-02-27.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you!

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. 2017. Ray rllib: A composable and scalable reinforcement learning library. *CoRR*, abs/1712.09381.

Jessy Lin, Yuqing Du, Olivia Watkins, Danijar Hafner, Pieter Abbeel, Dan Klein, and Anca Dragan. 2023. Learning to model the world with language.

Jesse Mu, Victor Zhong, Roberta Raileanu, Minqi Jiang, Noah Goodman, Tim Rocktäschel, and Edward Grefenstette. 2022. Improving intrinsic exploration with language abstractions. In *Advances in Neural Information Processing Systems*, volume 35, pages 33947–33960. Curran Associates, Inc.

Pierre-Yves Oudeyer, Frdric Kaplan, and Verena V. Hafner. 2007. Intrinsic motivation systems for autonomous mental development. *IEEE Transactions on Evolutionary Computation*, 11(2):265–286.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback.

Judea Pearl. 2009a. Causal inference in statistics: An overview. *Statistics Surveys*, 3:96–146.

Judea Pearl Pearl. 2009b. *Causality: Models, Reasoning and Inference*. Cambridge, 2nd edition.

Judea Pearl Pearl. 2018. *The Book of Why*. Basic Books, New York.

Jan Robine, Marc Höftmann, Tobias Uelwer, and Stefan Harmeling. 2023. Transformer-based world models are happy with 100k interactions.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code.

Bernhard Schölkopf, Francesco Locatello, Stefan Bauer, Nan Rosemary Ke, Nal Kalchbrenner, Anirudh Goyal, and Yoshua Bengio. 2021. Towards causal representation learning.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning.

Chenglei Si, Yanzhe Zhang, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2024. Design2code: How far are we from automating front-end engineering?

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

A Appendix

A.1 One Toy Example by chatGPT

We use chatGPT (GPT3.5) to generate a toy example: the generated python codes demonstrates that the LLM could track the interaction dynamics among the submodules. Notice that the this model gets "who moves to the door" wrong in the dialog (should be B, not C).

Prompt 1

Dialog to the ChatGPT to generate submodule dynamics in codes.

User: Let's play a game. There are two roles in the game: one is A and the other is B.
GPT: Sure, I'm up for a game! I can play the roles of A and B. What's the objective or rules of the game?
User: The objective is to track the behavior of the roles. A and B sit at the table now.
GPT: Got it! A and B are at the table. What happens next? Are there specific actions or decisions they should make, or should we just observe their behavior?
User: A decided to stay but B move to the blackboard.
GPT: Interesting choices! A has decided to stay at the table, while B has moved to the blackboard. What happens next? Any specific actions or reactions, or should we continue to observe?
User: Suddenly the lamp from the ceil near the table falls. Who might be hit by the lamp?
GPT: Given that A decided to stay at the table and B moved to the blackboard, it's likely that the lamp might hit B, as they are closer to the blackboard. However, the outcome could depend on factors like the lamp's trajectory, the speed of the fall, and the positions of A and B. What happens next?
User: Who might be hit if the lamp is exactly above the table?
GPT: If the lamp is directly above the table and both A and B are at the table, it's possible that either A or B could be hit by the falling lamp, depending on their exact positions at the

table. The outcome would then depend on the specific arrangement of A and B in relation to the falling lamp. What happens next?
User: Assume A is hit. What could be the possible decision for A to avoid being hit by the lamp?
GPT: If A is in danger of being hit by the falling lamp, they might try to take evasive action to avoid getting hurt. Possible decisions for A to avoid being hit by the lamp could include:
1. Duck or move away: A could quickly duck under the table or move to the side, away from the falling lamp's path.
2. Push the chair back: A might push their chair back to create more space and reduce the chance of being hit.
3. Use an object as a shield: If there's a nearby object, A could use it as a shield to protect themselves from the falling lamp.
4. Call for help: A might shout for assistance from someone else in the vicinity to help them avoid the falling lamp.
What does A decide to do in this situation?
User: If A decided to use an object as shield while B move to the door. We hear someone open the door. Who might be the person to open the door?
GPT: Given the current scenario where A is using an object as a shield and B has moved to the door, the person opening the door could be someone entirely new, perhaps a third character in the scene. Without additional information, it's not clear who that person might be. It could be another individual entering the room, someone responding to the commotion, or any other character relevant to the context.
Who do you want the person opening the door to be, or would you like me to suggest a character?
User: That's right. C is the new character opened the door and walked to A while B is drinking water. Suddenly, there is earthquake. Water spilled out. Who might want to change clothes since his clothes get wet by the water.
GPT: In this situation, B, who was drinking water when the earthquake occurred and caused water to spill, might want to change clothes since their clothes got wet. The decision to change clothes would be a practical response to being wet, especially if B wants to be more comfortable or avoid any potential discomfort caused by damp clothing. What does B decide to do next?
User: Generate python code to describe the whole story till now
GPT: Creating a Python code to describe the entire story so far is an interesting task. Here's a simplified example in Python code:

```python
class Character:
    def __init__(self, name):
        self.name = name

    def stay_at_table(self):
        return f"{self.name} decided to stay at the table."

    def move_to_blackboard(self):
        return f"{self.name} moved to the blackboard."

    def use_object_as_shield(self):
        return f"{self.name} used an object as a shield."

    def move_to_door(self):
        return f"{self.name} moved to the door."

    def change_clothes(self):
        return f"{self.name} decided to change clothes."

# Initial characters
A = Character("A")
B = Character("B")
C = Character("C")
```

```
# Initial actions
print(A.stay_at_table())
print(B.move_to_blackboard())

# Incident with the falling lamp
print("Suddenly, the lamp from the ceiling near the table falls.
                                ")

# Reaction to the earthquake
print("An earthquake occurs. Water spills as B is drinking water
                                .")
print(C.move_to_door())
print(C.walk_to(A))
print(B.change_clothes())
```

A.2 Single Episode of Flappy Bird Game

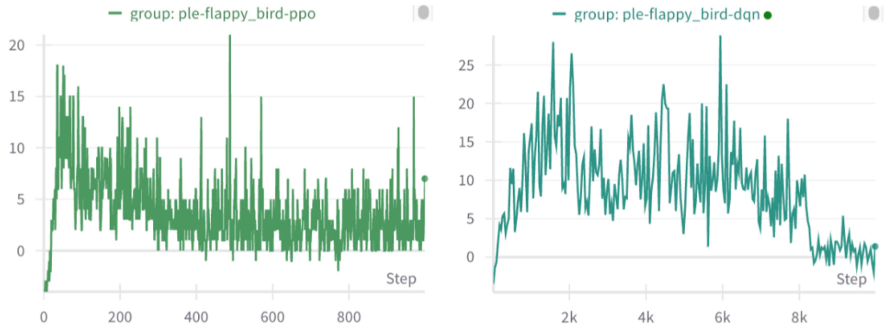


Figure 5: Max agent rewards (PPO vs DQN).

A.3 Backtrace the Causal Graph

The following example is adapted from Pearl (2009b). Could observe that GPT4 could backtrace the *causal graph* reasoning. Notice that some further instruction is needed to control the reasoning when back-tracing to state A or state B. Also GPT4 considers partial effect (magnitude) of $C \rightarrow D$.

Prompt 2

Backtrace the actions on causal factor graph.

User: Generate a causal graph: A -> C and B ->C and C -> D
GPT: Here's the causal graph based on your instructions (Fig. 6), showing the relationships: A -> C, B -> C, and C -> D. Each node represents a variable, and the directed edges indicate the direction of causation between these variables.
User: Now A means "it rains yesterday", B means "the house's sprinkler is on", C means "the grass is wet" and D means "it's slippery to walk on". If D is not true, what could we infer for A or B?
GPT: If D (it's slippery to walk on) is not true, this suggests that the grass is not sufficiently wet to create slippery conditions. In the causal chain you provided, C (the grass is wet) directly influences D. So, if D is not true, we can infer that C might not be true or not to a significant degree that it affects walking conditions. However, the absence of D doesn't directly inform us about the truth values of A (it rained yesterday) or B (the house's sprinkler is on), because:

1. It rained yesterday (A): Rain can wet the grass, but the grass may have dried up by the time someone checks if it's slippery, or the rain wasn't enough to create conditions for D.

2. The house's sprinkler is on (B): The sprinkler can also wet the grass, but similarly to rain, the water might not have been enough to make it slippery, or the sprinkler water has dried or was not directed at areas that affect walking conditions.

The causal model indicates that both A and B can cause C, but the absence of D suggests C might not have occurred or not to the extent necessary to lead to D. However, we cannot directly infer the specific states of A and B without more information. It's possible that neither A nor B occurred, or they occurred but their effects were mitigated by other factors (like wind, sun exposure, or the absorbent nature of the soil) before leading to D.

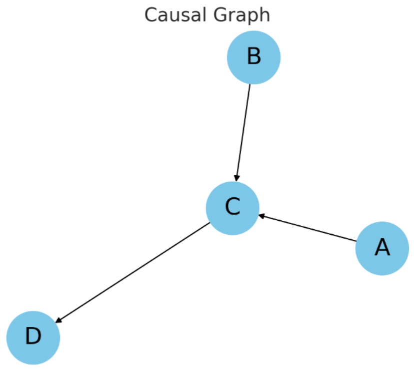


Figure 6: A Causal Graph Example

A.4 Single Stage Counterfactual

The following plot (in Fig. 7) depicts the flow chart of the single stage of the counterfactual proposal.

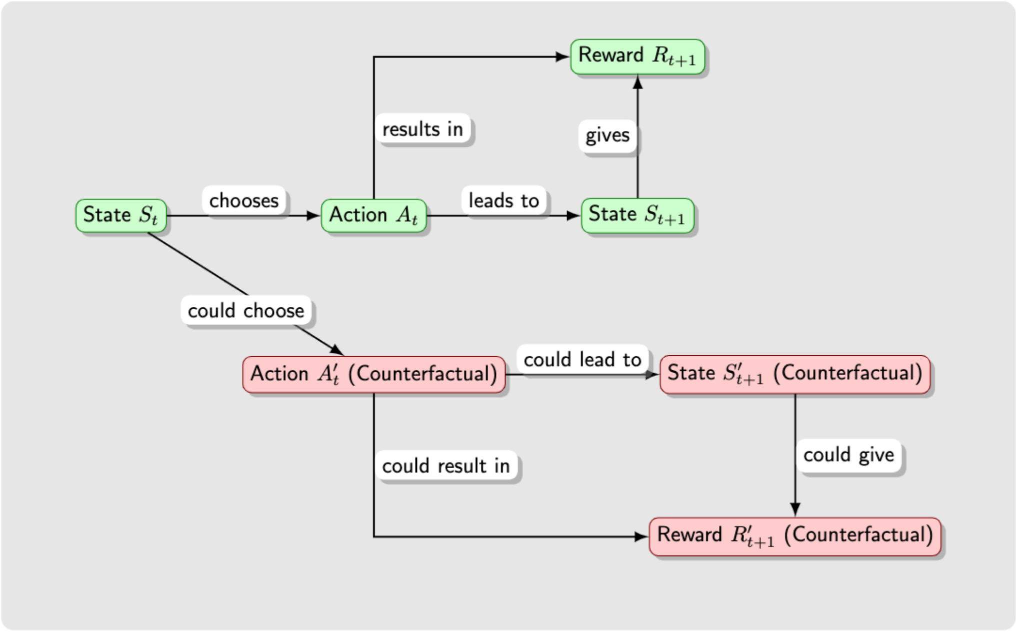


Figure 7: Simple Counterfactual Diagram

A.5 An Example of Counterfactual Proposals by GPT4

The following example is adapted from Pearl (2009b). Could observe that GPT4 could backtrace the *causal graph* reasoning. Notice that some further instruction is needed to control the reasoning when back-tracing to state A or state B. Also GPT4 considers partial effect (magnitude) of $C \rightarrow D$.

Prompt 3

Counterfactual proposals on flappy birds game from GPT4.

User: Assume we are playing the flappy bird game. After some time, the bird hits the pipe. What could be the reason of this?
GPT: In the Flappy Bird game, hitting a pipe can happen due to several reasons, often related to the game's mechanics and the player's performance. Here are some possible reasons why the bird might hit a pipe:

1. Timing Issues: The player didn't tap at the right moment to ensure the bird flies through the gap between pipes. Timing the taps is crucial to navigate through the obstacles successfully.

2. Speed Misjudgment: The player may either tap too quickly, causing the bird to ascend too high, or not tap quickly enough, resulting in the bird descending too much and hitting the lower part of the pipe.

3. Distraction: External factors could distract the player, leading to a lapse in concentration and a misjudgment in gameplay, causing the bird to collide with a pipe.

4. Lack of Experience: New or inexperienced players might find it challenging to control the bird's flight accurately due to unfamiliarity with the game's mechanics.

5. Game Difficulty: As the game progresses, the gaps between the pipes may seem to require more precise navigation, and the player might not adjust their strategy accordingly.

6. Technical Issues: Although less common, technical glitches or lag in the game can disrupt the gameplay, leading to unintended collisions with pipes.

7. Pressure: The pressure of trying to beat a high score or the stress of the increasing pace of the game might lead to mistakes.

Each of these factors highlights the challenge of mastering Flappy Bird's simple yet difficult gameplay, where precision, timing, and focus are key to achieving a high score.