PIANIST: Learning Partially Observable World Models with LLMs for Multi-Agent Decision Making

Jonathan Light[1] **Sixue Xing**[1] **Yuanzhe Liu**[1] **Weiqin Chen**[1] **Min Cai**[2] **Xiusi Chen**[3]
Guanzhi Wang[4] **Wei Cheng**[5] **Yisong Yue**[4] **Ziniu Hu**[6]
[1]Rensselaer Polytechnic Institute, [2]Shenzhen University,
[3]University of Illinois Urbana-Champaign, [4]California Institute of Technology,
[5]NEC laboratories America, [6]xAI

Abstract

Effective extraction of the world knowledge in LLMs for complex decision-making tasks remains a challenge. We propose a framework PIANIST for decomposing the world model into seven intuitive components conducive to zero-shot LLM generation. Given only the natural language description of the game and how input observations are formatted, our method can generate a working world model for fast and efficient MCTS simulation. We show that our method works well on two different games that challenge the planning and decision making skills of the agent for both language and non-language based action taking, *without any training on domain-specific training data or explicitly defined world model.*

1 Introduction

Recent studies have shown how LLMs, trained on massive amounts of online data, can be used as a world model to conduct planning [1, 2]. However, using LLMs as world models have not been as well explored in multi-agent, partial information settings such as in language games and other board games. These settings present unique challenges due to (1) the complexity of all the possible action, (2) partial observablity, and (3) other, possibly adversarial or stochastic, agents. These complexities mean that directly using the LLM as a policy for planning is not as feasible [3]. More related works in App. D.

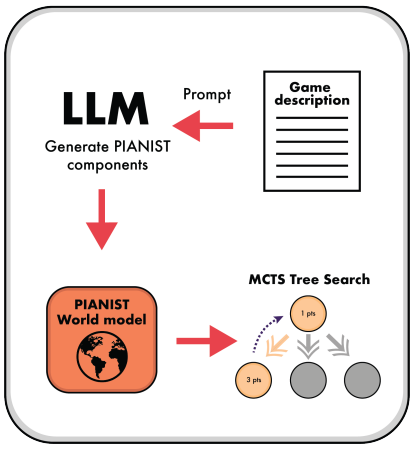

Figure 1: **Overview of PIANIST**. Starting with just the game description, the LLM generates a complete multi-agent, partial information world model, which can then be used for planning via search.

In this work, we introduce a framework PIANIST that allows us to use the LLM to more easily learn and plan with a PIANIST world model. Specifically, PIANIST separates the world model into seven different components that we use the LLM to generate. This includes the forward transition function, the action function, and the information partition function, all of which we prompt the LLM to generate in the form of code, which is easily executable and verifiable. We show that our method works well on two different games – one card based, and one discussion based – showing strong performance from LLM-agents that use our framework.

2 Background

2.1 Decision problem formulation

We formulate decision making tasks as a partially observable Markov decision process (POMDP) with an explicit environment actor which makes it more LLM-friendly to model.

Problem definition. Given a state space $\mathcal{S}$ and action space $\mathcal{A}$, a policy function ϕ in policy space Φ maps states to probability distributions over actions, $\phi : \mathcal{S} \to \Delta\mathcal{A}$. An environment $\mathcal{E} = \langle \mathcal{S}, \mathcal{A}, \mathcal{N}, T, R, A, \phi_\epsilon \rangle$ includes the state and action spaces, actors $\mathcal{N}$, a **transition function** $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, a **reward function** $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^{|\mathcal{N}|}$ for actor rewards, and an **action function** $A : \mathcal{S} \to \mathcal{N}, \mathcal{P}(\mathcal{A})$ determining legal actions. The environment actor's policy ϕ_e handles stochastic transitions, allowing for both deterministic and stochastic, single or multi-agent settings. In partial information settings, an **information partition** function $P : \mathcal{S} \times \mathcal{N} \to \mathcal{I}$ maps hidden states to information sets. Then the policy ϕ maps from information sets to action distributions, $\phi : \mathcal{I} \to \Delta\mathcal{A}$.

Goal. Given an environment $\mathcal{E}$, the goal for each actor $i \in \mathcal{N}$ is to find a policy ϕ_i^* that maximizes their cumulative reward, given that other players are also playing their optimal policy ϕ_{-i}^*: $\phi_i^* = \text{argmax}_{\phi_i} \mathop{\mathbb{E}}_{\tau \sim (\phi_i, \phi_{-i}^*)} \left[\sum_{(s,a) \in \tau} R_i(s, a) \right]$, where $\tau = (s_0, a_0, ...)$ is the simulated trajectory according to the strategic profile (ϕ_i, ϕ_{-i}) and the transition function T, with $a_t \sim \phi(a_t|s_t)$ and $s_{t+1} = T(s_t, a_t)$. $\boldsymbol{\phi}$ is commonly known as a Nash equilibrium, since no player i has any incentive to individually deviate from their optimal policy ϕ_i^*.

2.2 Decision-making games

We evaluate the performance of OMEGAZERO compared to other algorithms, both rule based and deep reinforcement learning based, on two board games representing games of two different genres.

GOPS (Goofspiel) is a multi-round, two-player simultaneous action game commonly studied in game theory [4, 5]. Each player is dealt identical hands of cards numbered 1 to k, and a shuffled prize deck, also numbered 1 to k, is revealed one card at a time. Both players simultaneously play a card from their hand; the higher card wins the prize, and both cards are discarded. After k rounds, players sum the values of their won prize cards, and the higher total determines the winner. Good players anticipate future moves and assess the value of each prize. Long-horizon games challenge LLM agents, as they struggle to connect near-term actions with long-term outcomes.

Taboo (2-player text version) is a cooperative game where one player is the clue-master and the other is the guesser. The clue-master is given a target word and a list of taboo words they cannot use in their clues. Each round, the clue-master makes a statement, and the guesser responds with one guess. The game ends when the guesser correctly identifies the word, makes five guesses, or the clue-master accidentally uses a taboo word. If a taboo word is used, the team scores 0; otherwise, the score is five minus the number of guesses. A good clue-master *anticipates the guesser's thought process* to help narrow down the options. The novelty of each word adds to the challenge.

3 Methodology

3.1 PIANIST: Extracting LLM World Knowledge

We present a new framework for extracting world knowledge from LLMs by dividing the world model into seven intuitive components that the LLM can understand. With this extracted model, we can apply model-based reinforcement learning techniques like MCTS or TD-learning. Most components are generated by prompting the LLM with the game description and a predefined Python parent template class. See App. C for examples of LLM generated models.

- $\mathcal{I}$**: Information sets.** The agent observes information sets, and we provide code for representing them along with a natural language game description as an interface between the real world and the agent. This and the game description are the only game-specific information given.
- $\mathcal{S}$**: Hidden states.** The agent records any relevant hidden information here.
- $\mathcal{N}$**: Actors.** Used to specify what the actor names are for the action function and reward function.
- A**: Action function.** For large action spaces, the function returns the top k most likely actions. For language actions, an LLM generates the top k text options. See section 3.3 for details.

- T, R**: Transition-reward function.** Combining state prediction and reward assignment for each player minimizes LLM errors. Deterministic transitions further reduce generation errors.
- P**: Information partition function.**
- $I : \mathcal{I} \to \mathcal{S}$**: Information realization function.** This maps information sets to their most likely hidden states, enabling the agent to simulate transitions between hidden states.

Together, we have the **P**artition function, **I**nformation set space, **A**ction space function, **N** players, **I**nformation realization function, **S**tate space, and **T**ransition-reward function, or PIANIST for short. If the LLM generates incorrect code, we use a reflexion approach to correct it and regenerate [6, 7]. We choose to learn hidden states and transitions because it's more intuitive for the LLM to understand actions and transitions at hidden states rather than at information sets.

3.2 Integrating PIANIST with Search

Our pseudocode for PIANIST-guided MCTS is in Alg. 1, with a diagram in Fig. 2, and further details in App. A. During MCTS, we sample a realization of the observed information set using the information realization function. A trajectory is then simulated by selecting the action with the highest UCT value (eq. 1) for the acting player. Since actors cannot distinguish hidden states within the same information set, UCT values are averaged across all states in that set, weighted by visit counts, preventing the use of hidden information. The simulation continues until a state s with unexplored actions is reached, where a random unexplored action a is chosen. The transition function provides the next state s' and reward r, which are recorded. The action function, partition function, and value heuristic are used to record the actions, information set, and value estimate for s'. A random rollout or LLM-generated heuristic computes the value estimate. Backpropagation is then performed from s' up the tree using the backpropagation equation (eq. 3).

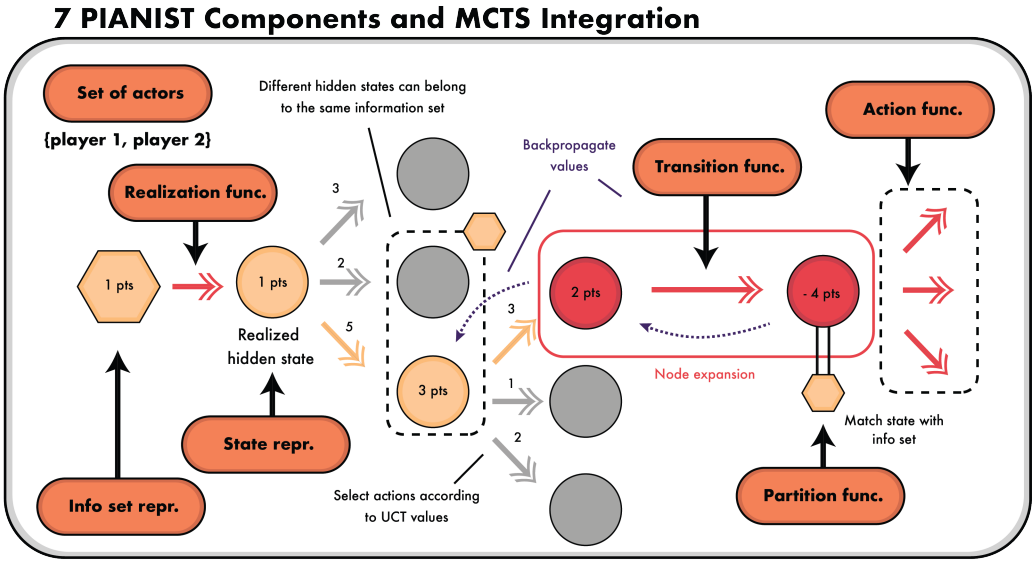

Figure 2: **Integrating PIANIST components with MCTS**. The realization function samples a hidden state for simulation, while the transition, action, and partition functions are used to expand new states. States are selected based on UCT values, aggregated across information sets for partial information. Though the diagram shows values for a single player, in practice, values for all players are inferred and updated simultaneously. See App. A for details and Fig. 3 for generation order.

3.3 Handling language actions

Language-based games are particularly challenging for traditional RL methods due to their need for *language abilities and extensive semantic knowledge*. In these games, the action space for language-based actions, such as discussion, is practically infinite, consisting of all possible word and token combinations. Current search methods are only effective in finite action spaces. Additionally, RL methods alone cannot inherently understand language or be trained to do so through self-play. We address this by utilizing LLMs to propose likely high-level dialogue actions for players. This allows us to (1) prune improbable actions and (2) focus on a few high-level strategy categories,

reducing the search space. The LLM only suggests possible actions, while the search algorithm assigns probabilities, mitigating the bias issue commonly found in LLM decision making [8].

4 Experiments

We evaluated our model against three different opponents. For the **ground-truth models**, we used ground truth models with MCTS search, combined with a random-rollout value heuristic, and played them against our LLM-generated agent, which also uses an LLM-generated value heuristic. Ground-truth include the true $\mathcal{S}, \mathcal{A}, \mathcal{N}, T, R, A$ models used during actual gameplay. For **LLM as policy**, we directly queried the LLM for actions in ReAct style [9], which includes a thought phase before action. For **human opponents**, we recruited 10 individuals to play 30 games of 6-card GOPS and 30 games of Taboo. In Taboo (a cooperative game), we paired each agent with a human-crafted model as the teammate (guesser), as the clue-giver role is more difficult. In GOPS, the two agents played directly against each other. We report both win rate and score for both games. In GOPS, win rate refers to whether a player had a higher score than their opponent, while score represents the point difference based on how many score cards were won. In Taboo, win rate measures whether the team guessed the word on the first try, and score is based on how quickly the team guessed the correct word. Note the possibility for tieing in GOPS.

As shown in Table 1, PIANIST performs similarly to ground-truth models, indicating that the LLM can generate an accurate world model using our framework. Additionally, Table 2 shows that our world model helps the agent plan more effectively than directly querying the LLM for actions. However, our agent struggles to consistently beat humans at GOPS and Taboo, highlighting the need for further research on improving LLM agents for complex decision-making environments on both action and language games (Table 3). Overall, despite using LLMs to generate its world model zero-shot, our agent demonstrates strong performance, showcasing the effectiveness of PIANIST in extracting world knowledge for tree search. This suggests that future work could explore more nuanced adaptations of the framework to balance decision-making performance across different games, potentially enabling more robust generalization in varied multi-agent environments.

Table 1: **PIANIST vs Ground truth models,** comparing performance when we replace ground truth models with LLM generated models.

Game	Setting	# games	PIANIST		Ground-truth	
			Winrate	Score	Winrate	Score
GOPS	6-card	300	52.3±6.2%	-0.21	48.7±1.9%	0.21
	12-card	300	47.3±2.9%	-0.07	47.0±5.7%	0.07
Taboo	as clue giver	15	60.0±8.7%	3.53	53.3±12.9%	3.06

Table 2: **PIANIST vs LLM as Policy**, where the LLM directly chooses which actions to take [9].

Game	Setting	# games	PIANIST		LLM-policy	
			Winrate	Score	Winrate	Score
GOPS	6-card	30	66.6±9.4%	0.3	33.3±9.4%	-0.3
	12-card	30	60.0±8.2%	0.2	33.3±9.4%	-0.2
Taboo	as clue giver	15	60.0±8.7%	3.53	53.3±12.9%	3.2

Table 3: **PIANIST vs the Humans**. Humans were given the same prompt as the LLM before play.

Game	Setting	# games	PIANIST		the humans	
			Winrate	Score	Winrate	Score
GOPS	6-card	40	54.9±8.3%	2.37	40.1±7.1%	-2.41
Taboo	as clue giver	15	60.0±8.7%	3.53	86.7±8.7%	4.33

References

[1] Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*, 2023.

[2] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[3] Zirui Zhao, Wee Sun Lee, and David Hsu. Large language models as commonsense knowledge for large-scale task planning. *Advances in Neural Information Processing Systems*, 36, 2024.

[4] Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael Bowling. Monte carlo sampling for regret minimization in extensive games. *Advances in neural information processing systems*, 22, 2009.

[5] Sheldon M Ross. Goofspiel—the game of pure strategy. *Journal of Applied Probability*, 8(3):621–625, 1971.

[6] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

[7] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

[8] Zelai Xu, Chao Yu, Fei Fang, Yu Wang, and Yi Wu. Language agents with reinforcement learning for strategic play in the werewolf game. *arXiv preprint arXiv:2310.18940*, 2023.

[9] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[10] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models. *arXiv preprint arXiv: Arxiv-2108.07258*, 2021.

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec

Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[12] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

[13] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *arXiv preprint arXiv: Arxiv-2206.07682*, 2022.

[14] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv: Arxiv-2204.02311*, 2022.

[15] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models. *arXiv preprint arXiv: Arxiv-2210.11416*, 2022.

[16] Jifan Yu, Xiaozhi Wang, Shangqing Tu, Shulin Cao, Daniel Zhang-Li, Xin Lv, Hao Peng, Zijun Yao, Xiaohan Zhang, Hanming Li, et al. Kola: Carefully benchmarking world knowledge of large language models. *arXiv preprint arXiv:2306.09296*, 2023.

[17] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):1–26, 2024.

[18] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023.

[19] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

[20] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv: Arxiv-2207.05608*, 2022.

[21] Joon Sung Park, Joseph C O'Brien, Carrie J Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023.

[22] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2024.

[23] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023.

[24] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

[25] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690, 2024.

[26] Zhiting Hu and Tianmin Shu. Language models, agent models, and world models: The law for machine reasoning and planning. *arXiv preprint arXiv:2312.05230*, 2023.

[27] Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 36:79081–79094, 2023.

[28] Xidong Feng, Ziyu Wan, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. Alphazero-like tree-search can guide large language model decoding and training. *arXiv preprint arXiv:2309.17179*, 2023.

[29] Bodhisattwa Prasad Majumder, Bhavana Dalvi Mishra, Peter Jansen, Oyvind Tafjord, Niket Tandon, Li Zhang, Chris Callison-Burch, and Peter Clark. Clin: A continually learning language agent for rapid task adaptation and generalization. *arXiv preprint arXiv:2310.10134*, 2023.

[30] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19632–19642, 2024.

[31] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.

[32] Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, et al. Language to rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647*, 2023.

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

[34] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[35] Meta Fundamental AI Research Diplomacy Team (FAIR)†, Anton Bakhtin, Noam Brown, Emily Dinan, Gabriele Farina, Colin Flaherty, Daniel Fried, Andrew Goff, Jonathan Gray, Hengyuan Hu, et al. Human-level play in the game of diplomacy by combining language models with strategic reasoning. *Science*, 378(6624):1067–1074, 2022.

[36] Sahar Abdelnabi, Amr Gomaa, Sarath Sivaprasad, Lea Schönherr, and Mario Fritz. Llm-deliberation: Evaluating llms with interactive multi-agent negotiation games. *arXiv preprint arXiv:2309.17234*, 2023.

[37] Yao Fu, Hao Peng, Tushar Khot, and Mirella Lapata. Improving language model negotiation with self-play and in-context learning from ai feedback. *arXiv preprint arXiv:2305.10142*, 2023.

[38] Noam Brown and Tuomas Sandholm. Superhuman ai for multiplayer poker. *Science*, 365(6456):885–890, 2019.

A MCTS Details

The Monte Carlo Tree Search (MCTS) process, illustrated in Figure 2, simulates possible future game states by expanding nodes in a search tree. Each node corresponds to a state, and edges correspond to actions. MCTS operates by iteratively simulating trajectories (denoted as τ) from the current realized hidden state until an unexpanded state is reached. The steps involved in MCTS include four key phases: selection, expansion, simulation, and backpropagation.

A.1 Upper Confidence Bound for Trees (UCT) and Action Selection

At each node, the next action $\boldsymbol{a}^*$ is selected according to the Upper Confidence Bound for Trees (UCT) formula, given by Equation (1):

$$\boldsymbol{a}^* = \arg\max_{\boldsymbol{a}} \left(\mathop{\mathbb{E}}_{\boldsymbol{s} \in I_{\boldsymbol{s}}} \left[r_i + \gamma V_i(\boldsymbol{s}') + C \sqrt{\frac{\log n(\boldsymbol{s})}{n(\boldsymbol{s}')}} \mid \boldsymbol{s}', \boldsymbol{r} = T(\boldsymbol{s}, \boldsymbol{a}) \right] \right) \tag{1}$$

In this equation:

- $\boldsymbol{a}^*$ is the action that maximizes the expression.
- $\boldsymbol{s}$ represents the current state, and $\boldsymbol{s}'$ is the next state reached after taking action $\boldsymbol{a}$.
- r_i is the immediate reward for agent i when transitioning from state $\boldsymbol{s}$ to $\boldsymbol{s}'$.
- $V_i(\boldsymbol{s}')$ is the value function that estimates the future reward from the next state $\boldsymbol{s}'$ for agent i.
- γ is the discount factor, which balances immediate and future rewards.
- $n(\boldsymbol{s})$ is the number of visits to the current state $\boldsymbol{s}$, while $n(\boldsymbol{s}')$ is the number of visits to the next state $\boldsymbol{s}'$.
- C is a constant controlling exploration versus exploitation, and $\log n(\boldsymbol{s})/n(\boldsymbol{s}')$ encourages exploring less-visited actions.
- $T(\boldsymbol{s}, \boldsymbol{a})$ represents the transition dynamics, mapping the current state and action to the next state and reward.

A.2 Probability Distribution Over the Information Set

Since our MCTS handles partial observability, a probability distribution is defined over the information set $I_{\boldsymbol{s}}$, which contains all possible hidden states $\boldsymbol{s}$ that are consistent with the observed information. This distribution is given by:

$$\mathbb{P}(\boldsymbol{s}) = \frac{n(\boldsymbol{s})}{\sum_{\boldsymbol{s}' \in I_{\boldsymbol{s}}} n(\boldsymbol{s}')} \tag{2}$$

Where:

- $\mathbb{P}(\boldsymbol{s})$ is the probability of being in state $\boldsymbol{s}$ within the information set $I_{\boldsymbol{s}}$.
- $n(\boldsymbol{s})$ is the visit count of state $\boldsymbol{s}$, and the denominator normalizes over all possible states $\boldsymbol{s}' \in I_{\boldsymbol{s}}$.

A.3 Backpropagation

After simulating a trajectory τ, MCTS backpropagates the results of the simulation to update the value estimates and visit counts for all state-action pairs $(\boldsymbol{s}, \boldsymbol{s}')$ along the trajectory. This update is performed using Equation (3):

$$\forall i \in \mathcal{N} \quad V_i(\boldsymbol{s}) \leftarrow V_i(\boldsymbol{s}) + \frac{1}{n(\boldsymbol{s})} \left(r_i + \gamma V_i(\boldsymbol{s}') - V_i(\boldsymbol{s}) \right), \quad n(\boldsymbol{s}) \leftarrow n(\boldsymbol{s}) + 1 \tag{3}$$

Algorithm 1: Monte Carlo Tree Search for Partial Information with Information Sets

Input: Initial information set $\boldsymbol{h}_0$, number of iterations M
Output: Best action $\boldsymbol{a}^*$
Function `Select(`$node$`):`
  while *node is non-terminal* **do**
    if *all node.actions have been tried* **then**
      $node \leftarrow$ `BestChild(`*node*`)`;
    end
    else
      $\boldsymbol{a} =$ random untried action;
      return $node, \boldsymbol{a}$;
    end
  end
  return $node$ (terminal state);
Function `Expand(`$parent, \boldsymbol{a}$`):`
  $\boldsymbol{s} \leftarrow T(parent.s, \boldsymbol{a})$;
  Create new node $child$ with parent $parent$ and $child.s \leftarrow \boldsymbol{s}$;
  $child.actions, child.actor \leftarrow A(\boldsymbol{s})$;
  $child.h \leftarrow P(\boldsymbol{s})$;
  $child.values \leftarrow$ `EstimateValues` ($child$);
  return $child$;
Function `EstimateValues(`$node$`):`
  Use random rollout or some other value heuristic to estimate the value of state $node.s$ for each player;
  return estimated values;
Function `Backpropagate(`$node$`):`
  $next\_values = node.values$;
  while $node$ *is not null* **do**
    $node \leftarrow$ parent of $node$;
    Update $node.values$ with the $next\_values$;
  end
Function `BestChild(`$root$`):`
  return child of $root$ with highest average reward;
Function `InformationSet(`$node$, $\boldsymbol{h}$`):`
  Generate the updated information set $\boldsymbol{h'}$ for $node$ based on observed actions and outcomes within $\boldsymbol{h}$;
  return $\boldsymbol{h'}$;
Function `Realize(`$\boldsymbol{h}$`):`
  $nodes \leftarrow$ all nodes in graph with $node.h = \boldsymbol{h}$;
  if $nodes$ *is empty* **then**
    Create a new node $node$ with $node.h = \boldsymbol{h}$ and $node.s = I(\boldsymbol{h})$ and add to graph
  end
  else
    $node \leftarrow$ RandomChoice($nodes$);
  end
  return $node$;
begin
  for $i = 1$ **to** N **do**
    $root \leftarrow$ `Realize(`$\boldsymbol{h}_0$`)`;
    $node, \boldsymbol{a} \leftarrow$ `Select(`$root$`)`;
    $child \leftarrow$ `Expand(`$node, \boldsymbol{a}$`)`;
    `Backpropagate(`$child$`)`;
  end
  $\boldsymbol{a}^* \leftarrow$ `BestChild(`$root$`)`;
end

Here:

- $V_i(\boldsymbol{s})$ is updated for agent i by adding a fraction of the difference between the expected future reward (given by $r_i + \gamma V_i(\boldsymbol{s}')$) and the current value estimate $V_i(\boldsymbol{s})$.
- r_i is the reward obtained from transitioning between $\boldsymbol{s}$ and $\boldsymbol{s}'$.
- γ is the discount factor, which balances immediate and future rewards.
- $n(\boldsymbol{s})$ is incremented to reflect that the state $\boldsymbol{s}$ has been visited once more.

This backpropagation process ensures that the value estimates $V_i(\boldsymbol{s})$ are refined based on simulated outcomes, allowing the MCTS process to converge on more accurate policies over time.

By iteratively simulating trajectories, selecting actions, expanding nodes, and backpropagating rewards, MCTS effectively balances exploration and exploitation, making it a powerful search algorithm for solving decision-making problems in partially observable environments.

B PIANIST Generation details

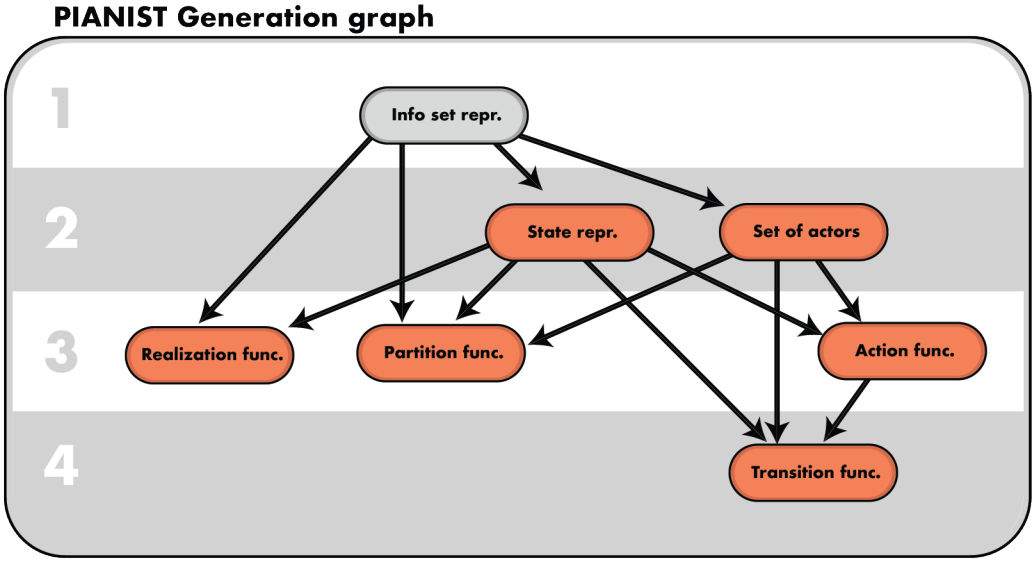

Figure 3: **Directed generation graph for PIANIST**. We display the sequential generation order for the various components of PIANIST, with dependencies shown by directed arrows. Generating and testing objects in this order minimizes the probability of execution failure. The initial information set representation is given by the environment to allow an unified interface with the environment. Modularization also means we can test each component individually.

C Example PIANIST models generated by `gpt-4o`

Example LLM Generated Forward Dynamics Model (GOPS)

```python
class CustomForwardTransitor(ForwardTransitor):
    '''
    Custom forward transitor for the Game of Pure Strategy (GOPS).
    Implements the game logic for transitioning between states.
    '''

    def _transition(self, state: HiddenState, action: int, actor: int) -> Tuple[HiddenState, Dict[
        int, float]]:
        '''
        Transits to the next state given the current state, actor, and action taken by the actor.

        Args:
            state: current HiddenState
            action: card played by the actor
            actor: actor that is taking the action (0 for player 0, 1 for player 1)

        Returns:
            next_state: updated HiddenState
            rewards: reward of the transition for each player
        '''
        # Copy current state variables to modify them
        player_0_hand = state.player_0_hand
        player_1_hand = state.player_1_hand
        prize_deck = state.prize_deck
        player_0_played_cards = state.player_0_played_cards
        player_1_played_cards = state.player_1_played_cards
        played_prize_cards = state.played_prize_cards
        player_0_cumulative_score = state.player_0_cumulative_score
        player_1_cumulative_score = state.player_1_cumulative_score
        contested_points = state.contested_points

        # Record the action taken by the actor
        if actor == 0:
            player_0_played_cards += (action,)
            player_0_hand = player_0_hand - frozenset([action])
        elif actor == 1:
            player_1_played_cards += (action,)
            player_1_hand = player_1_hand - frozenset([action])

        # If both players have played cards, determine the winner
        if player_0_played_cards and player_1_played_cards:
            player_0_card = player_0_played_cards[-1]
            player_1_card = player_1_played_cards[-1]

            if player_0_card > player_1_card:
                # Player 0 wins the round
                if contested_points > 0:
                    player_0_cumulative_score += contested_points + played_prize_cards[-1]
                    contested_points = 0
                else:
                    # Award the prize card if it exists
                    if prize_deck:
                        prize_card = prize_deck.pop()
                        player_0_cumulative_score += prize_card
                        played_prize_cards += (prize_card,)
                    else:
                        contested_points = 0  # No prize card to award
            elif player_1_card > player_0_card:
                # Player 1 wins the round
                if contested_points > 0:
                    player_1_cumulative_score += contested_points + played_prize_cards[-1]
                    contested_points = 0
                else:
                    # Award the prize card if it exists
                    if prize_deck:
                        prize_card = prize_deck.pop()
                        player_1_cumulative_score += prize_card
                        played_prize_cards += (prize_card,)
                    else:
                        contested_points = 0  # No prize card to award
            else:
                # Cards are equal, points are contested
                contested_points += 1

        # Prepare rewards dictionary
        rewards = {0: player_0_cumulative_score, 1: player_1_cumulative_score}

        # Initialize new state at the point of return
        new_state = HiddenState(
            player_0_hand=player_0_hand,
            player_1_hand=player_1_hand,
            prize_deck=prize_deck,
            player_0_played_cards=player_0_played_cards,
```

```
            player_1_played_cards=player_1_played_cards,
            played_prize_cards=played_prize_cards,
            player_0_cumulative_score=player_0_cumulative_score,
            player_1_cumulative_score=player_1_cumulative_score,
            contested_points=contested_points
        )

        return new_state, rewards
```

Example LLM Generated Value Heuristic Function (GOPS)

```
class CustomValueHeuristic(ValueHeuristic):
    '''
    Custom value heuristic for evaluating the state of the GOPS game.
    This heuristic estimates the expected rewards based on the cards in hand
    and prize cards available.

    Attributes:
        player_0_weight: Weight given to Player 0's score.
        player_1_weight: Weight given to Player 1's score.
    '''

    def _evaluate(self, state: HiddenState) -> tuple[dict[int, float], dict]:
        '''
        Evaluates the state

        Args:
            state: current hidden state of the game

        Returns:
            values: estimated values of the state for each player
            notes: additional notes about the evaluation
        '''
        player_0_value = self._estimate_value(state.player_0_hand, state.prize_deck)
        player_1_value = self._estimate_value(state.player_1_hand, state.prize_deck)

        values = {
            0: player_0_value + state.player_0_cumulative_score,
            1: player_1_value + state.player_1_cumulative_score
        }

        notes = {
            'player_0_hand': state.player_0_hand,
            'player_1_hand': state.player_1_hand,
            'prize_deck': state.prize_deck,
            'player_0_value': player_0_value,
            'player_1_value': player_1_value
        }

        return values, notes

    def _estimate_value(self, hand: FrozenSet[int], prize_deck: FrozenSet[int]) -> float:
        '''
        Estimates the potential value of a player's hand based on available prize cards.

        Args:
            hand: The player's hand of cards.
            prize_deck: The current prize deck cards.

        Returns:
            estimated value for the player's hand.
        '''
        if not prize_deck:
            return 0.0  # No prize cards left to claim

        # Count the number of prize cards that can be won
        prize_values = [prize for prize in prize_deck if prize in hand]

        # If the player has cards that can win the prize cards, sum their values
        value = sum(prize_values)

        # Optionally, apply a strategic factor or heuristic for more complex evaluations
        return value
```

Example LLM Generated Actor-Action Selection Model (GOPS)

```python
class CustomActorActionEnumerator(ActorActionEnumerator):
    '''
    Custom enumerator for the Game of Pure Strategy (GOPS) that determines which player can act
        and what actions they can take.
    '''

    def _enumerate(self, state: Hashable) -> tuple[Optional[int], frozenset]:
        '''
        Enumerates the acting player and their available actions based on the current game state.
        The first player in the player order is always returned.

        Args:
            state: current state (expected to be a HiddenState)

        Returns:
            actor: the acting actor (either 0 or 1 for players).
            actions: set of actions that the actor may take (cards they have left in their hand).
        '''
        if not isinstance(state, HiddenState):
            raise ValueError("State must be an instance of HiddenState.")

        # Determine the acting player (always the first player in the order for this game)
        actor = self.player_order[0]

        # Get the actions available to the acting player
        if actor == 0:
            actions = state.player_0_hand
        else:
            actions = state.player_1_hand

        return actor, frozenset(actions)
```

Example LLM Generated Information Realization Function (GOPS)

```python
class CustomInformationPrior(InformationPrior):
    '''
    Custom implementation of InformationPrior for the Game of Pure Strategy (GOPS).

    This class maps an information set, which may represent a state of the game where players have
        played certain cards, to a corresponding hidden state.
    '''

    def _get_prior_state(self, information_set: Hashable) -> HiddenState:
        '''
        Returns a hidden state corresponding to the given information set. This implementation
            generates a hidden state based on the current observed state.

        It assumes that the information_set contains relevant details such as played cards and
            hands.
        '''
        # Placeholder for actual extraction of information from the information set.
        # The actual implementation would depend on the structure of information_set.

        player_0_hand = information_set.player_0_hand
        player_1_hand = information_set.player_1_hand
        prize_deck = information_set.prize_deck
        player_0_played_cards = information_set.player_0_played_cards
        player_1_played_cards = information_set.player_1_played_cards
        played_prize_cards = information_set.played_prize_cards
        player_0_cumulative_score = information_set.player_0_cumulative_score
        player_1_cumulative_score = information_set.player_1_cumulative_score
        contested_points = information_set.contested_points

        # Create and return the hidden state
        return HiddenState(
            player_0_hand=player_0_hand,
            player_1_hand=player_1_hand,
            prize_deck=prize_deck,
            player_0_played_cards=player_0_played_cards,
            player_1_played_cards=player_1_played_cards,
            played_prize_cards=played_prize_cards,
            player_0_cumulative_score=player_0_cumulative_score,
            player_1_cumulative_score=player_1_cumulative_score,
            contested_points=contested_points,
        )
```

D Related Work

LLMs for text agents. Large language models (LLMs) have demonstrated significant emergent capabilities, such as zero-shot prompting and complex reasoning [10, 11, 12, 13, 14, 15]. They also possess extensive world knowledge [16], which has spurred increasing efforts to use LLMs for decision-making in text agents [17]. One notable paradigm is ReAct [18], which employs an observation-reasoning-acting loop for agent planning with LLMs. Building on ReAct, Reflexion [7] incorporates self-reflection to enhance reasoning capabilities. Other works in this domain have utilized feedback [19, 20], memory [21], and tool use [22, 23] to further enhance agent performance. Our proposed method, OMEGAZERO, integrates these components to design an agent capable of systematic analysis and strategic decision-making. Typical prompting techniques for text agents include Chain-of-Thought [24], Tree-of-Thought [2], and Graph-of-Thought [25].

LLMs and planning. Recent works have proposed planing using the LLM as a world model [1, 26]. These works have mostly centered around using the LLM as a forward transition function (dynamics model) by querying the LLM for the next state [2], or using a planning language to describe plans [27]. Other works have explored using LLMs to guide the MCTS search process by using the LLM as a policy [3, 28]. We build upon these works by investigating what kind of world model is more conducive to extracting world knowledge from the LLM and combining it with MCTS.

Skill learning with LLMs. Recent works have explored the possibly of LLMs learning skills through learning a textual short and long term memory [7, 29], or textual insights extracted from the memories [30]. Due to the length of trajectories in our game setting and the numerical nature of the data, it is difficult to learn textual memories, so we learn high level strategies instead. We also explore how to acquire simulational self-play feedback in multiagent settings. Using LLMs to learn a functional reward model has also been applied to great success on single-agent robotic tasks [31, 32]. We build upon their work by introducing a new improvement method that can help learn a better reward model, and exploring how function learing can be applied to multiagent settings with simulated feedback.

AI in strategy games. AI has been applied to great success in board games. AlphaGo and MuZero demonstrated the power of combining MCTS, deep learning, and feedback generation using self-play in games such as Go, Chess, and Shogi [33, 34]. Language models can also be trained on human in-game discussion data and integrated with another separately trained action planner to play board games with dialogue [35]. We build upon the AI for games literature by showing that LLMs can accomplish both *(1) the training of a value heuristic like that in AlphaGo through self-play more efficiently than RL and (2) dialogue generation in discussion games with no human examples*. These adversarial environments are not just limited to board games. For example, there has been recent interest on creating LLM-agents that can negotiate [36, 37], which our method can also be applied to. Traditionally the solution to searching over a large action space has been to bucket similar actions together, such as possible raises in poker [38]. We leverage the inherent distribution in the LLM to suggest the top most probable, yet distinct, actions instead.

E Prompts used to generate components

P: Information Function Generation Prompt (Taboo)

```
SYSTEM PROMPT:

"""
You are a programmer developing an accurate game engine. Your task is to implement parts of the
     game simulator in Python. The simulator models simultaneous actions as sequential ones with
     partial observation. When players $1, ..., k$ take simultaneous actions, they do so
     sequentially without seeing the actions of previous players. These actions are first recorded
      in a ‘HiddenState‘ object before being revealed. Do not repetitively generate Hidden state.
     If new state is generated, construct at return statement. ‘ObservedState‘ and ‘HiddenState‘
     should be under ‘@dataclass(frozen=True)‘.  Use tuple instead of list to make sure that
     vectors are frozen.
"""

HUMAN PROMPT:

from typing import Optional, Tuple
    clue_word: str                              # The actual clue word (hidden from guesser)
    taboo_words: Tuple[str, ...]                # Taboo words (hidden from guesser)
    taboo_word_used: bool                        # Whether a taboo word was used
    guesses: Tuple[str, ...]                     # Guesses made (hidden from clue-master)
    clue_master_statements: Tuple[str, ...]     # Statements made (hidden from guesser)

"""
An observed state (information set) in the game is defined as follows:
"""
@dataclass(frozen=True)
class ObservedState:
    clue_word: Optional[str]                    # The word the guesser needs to guess
    taboo_words: Optional[tuple[str, ...]] # List of taboo words the clue-master cannot use
    guesses: tuple[str, ...] = tuple()  # List of words guessed by the guesser
    clue_master_statements: tuple[str, ...] = tuple()  # Statements made by the clue-master
    taboo_word_used: bool = False      # Flag to indicate if a taboo word was used
    game_over: bool = False            # Flag to indicate if the game is over
    score: int = 5                     # Initial score, will decrease based on guesses
    actor: str = "clue_master"          # Indicates whose turn it is: "clue_master" or "guesser"

"""
Write an information function ‘CustomInformationFunction‘ for this game that inherits from the ‘
     InformationFunction‘ class. Include all docstings from the parent class:
"""

class InformationFunction(AbstractLogged):
    '''
    Abstract class for mapping hidden states to information sets
    '''
    def get_information_set(self, state: Hashable, actor: Hashable) -> Hashable:
        '''
        Returns the observed state (information set) for the state

        Args:
            state: current state
            actor: actor that is observing the state

        Returns:
            information_set: information set for the state
        '''
        return self._get_information_set(state=state, actor=actor)

    @abstractmethod
    def _get_information_set(self, state: Hashable, actor: Hashable) -> Hashable:
        '''
        Returns the observed state (information set) for the state

        Args:
            state: current state
            actor: actor that is observing the state

        Returns:
            information_set: information set for the state
        '''
        pass

"""
The player names are defined as follows:
{'guesser', 'clue_master'}
"""
```

A: Action Function (Taboo)

SYSTEM PROMPT:

```
"""
You are a programmer developing an accurate game engine. Your task is to implement parts of the
      game simulator in Python. The simulator models simultaneous actions as sequential ones with
      partial observation. When players $1, ..., k$ take simultaneous actions, they do so
      sequentially without seeing the actions of previous players. These actions are first recorded
       in a `HiddenState` object before being revealed. Do not repetitively generate Hidden state.
      If new state is generated, construct at return statement. `ObservedState` and `HiddenState`
      should be under `@dataclass(frozen=True)`.  Use tuple instead of list to make sure that
      vectors are frozen.
"""
```

HUMAN PROMPT:

```
"""
Taboo (2-player text version) is a two player cooperative dialogue game where 1 player is the clue
      -master and 1 player is the guesser. The clue master is given the clue-word and a list of
      taboo words. Each discussion round the clue master makes one statement to the guesser, but
      cannot use any of the taboo words in their statements. The guesser can then guess one word.
      This continues until either the guesser guesses the word, the guesser has guess five times
      already, or the clue-master has spoken one of the taboo words. If the clue-master uses any of
       the taboo words, the team score is 0. Otherwise, the score is five minus the number of words
       guesser has guessed.
"""

"""
A hidden state in the game is defined as follows:
"""
class HiddenState:
    from dataclasses import dataclass
    from typing import Optional, Tuple
    clue_word: str                              # The actual clue word (hidden from guesser)
    taboo_words: Tuple[str, ...]                # Taboo words (hidden from guesser)
    taboo_word_used: bool                        # Whether a taboo word was used
    guesses: Tuple[str, ...]                    # Guesses made (hidden from clue-master)
    clue_master_statements: Tuple[str, ...]     # Statements made (hidden from guesser)
"""
Write an actor-action enumerator `CustomActorActionEnumerator` for this game that inherits from
      the `TextActorActionEnumerator` class. Include all docstings from the parent class:
"""
class TextActorActionEnumerator(ActorActionEnumerator):
    '''
    Abstract class for an actor action enumerator that can enumerate textual actions (such as
         dialogue and code)susing an LLM
    '''
    model: LLMModel

    def __init__(self, model: LLMModel, max_actions: int, player_order: Tuple[Hashable] = tuple())
         :
        super().__init__(player_order)
        self.model = model
        self.max_actions = max_actions

    def enumerate(self, state: Hashable)->tuple[Optional[Hashable], set]:
        '''
        Enumerates a (single) actor that may take actions at the state and the actions that the
             actor may take.
        If multiple actors may take actions at this state (simultaneous state), the first actor in
              the player order is returned.

        Args:
            state: current state

        Returns:
            actor: the acting actor. -1 for environment, None for terminal state
            actions: set of actions that the actor may take
        '''
        actor, actions = self._enumerate(state)
        assert len(actions) <= self.max_actions
        return actor, set(actions)

    @abstractmethod
    def _enumerate(self, state: Hashable)->tuple[Optional[Hashable], frozenset]:
        '''
        Enumerates a (single) actor that may take actions at the state and the actions that the
             actor may take.
        If multiple actors may take actions at this state (simultaneous state), the first actor in
              the player order is returned.

        Args:
            state: current state

        Returns:
            actor: the acting actor. -1 for environment, None for terminal state
            actions: set of actions that the actor may take
```

```
        For textual actions, the actions will be generated by prompting the LLM model with a
             system message and a user message, using the generate_k_responses method. An example
             of a system prompt is "You are the clue giver in the game of Codenames. The rules of
             Codenames are ...". An example of a user prompt is "State of game: ... Please give a
             clue as a single tuple (word, number), nothing else."
        '''
        pass

    def generate_k_responses(self, sys_prompt: SystemMessage, user_prompt: HumanMessage, k: int =
         -1)->list[str]:
        '''
        Generates k responses given the system prompt and user prompt

        Args:
            sys_prompt: system prompt
            user_prompt: user prompt
            k: number of responses to generate, -1 if set to self.max_actions

        Returns:
            responses: list of responses
        '''
        if k == -1:
            k = self.max_actions
        return [self.model.generate([sys_prompt, user_prompt]) for _ in range(k)]

"""
The player names are defined as follows:
{'guesser', 'clue_master'}
"""
```

I: Information Realization Function (Taboo)

SYSTEM PROMPT:

```
"""
You are a programmer developing an accurate game engine. Your task is to implement parts of the
      game simulator in Python. The simulator models simultaneous actions as sequential ones with
      partial observation. When players $1, ..., k$ take simultaneous actions, they do so
      sequentially without seeing the actions of previous players. These actions are first recorded
       in a ‘HiddenState‘ object before being revealed. Do not repetitively generate Hidden state.
      If new state is generated, construct at return statement. ‘ObservedState‘ and ‘HiddenState‘
      should be under ‘@dataclass(frozen=True)‘.  Use tuple instead of list to make sure that
      vectors are frozen.
"""
```

HUMAN PROMPT:

```
"""
Taboo (2-player text version) is a two player cooperative dialogue game where 1 player is the clue
      -master and 1 player is the guesser. The clue master is given the clue-word and a list of
      taboo words. Each discussion round the clue master makes one statement to the guesser, but
      cannot use any of the taboo words in their statements. The guesser can then guess one word.
      This continues until either the guesser guesses the word, the guesser has guess five times
      already, or the clue-master has spoken one of the taboo words. If the clue-master uses any of
       the taboo words, the team score is 0. Otherwise, the score is five minus the number of words
       guesser has guessed.
"""
"""
A hidden state in the game is defined as follows:
"""
class HiddenState:
    from dataclasses import dataclass
    from typing import Optional, Tuple
    clue_word: str                                  # The actual clue word (hidden from guesser)
    taboo_words: Tuple[str, ...]                    # Taboo words (hidden from guesser)
    taboo_word_used: bool                            # Whether a taboo word was used
    guesses: Tuple[str, ...]                         # Guesses made (hidden from clue-master)
    clue_master_statements: Tuple[str, ...]      # Statements made (hidden from guesser)
"""
An observation in the game is defined as follows:
"""
@dataclass(frozen=True)
class ObservedState:
    clue_word: Optional[str]                        # The word the guesser needs to guess
    taboo_words: Optional[tuple[str, ...]] # List of taboo words the clue-master cannot use
    guesses: tuple[str, ...] = tuple()  # List of words guessed by the guesser
    clue_master_statements: tuple[str, ...] = tuple()  # Statements made by the clue-master
    taboo_word_used: bool = False      # Flag to indicate if a taboo word was used
    game_over: bool = False            # Flag to indicate if the game is over
    score: int = 5                     # Initial score, will decrease based on guesses
    actor: str = "clue_master"          # Indicates whose turn it is: "clue_master" or "guesser"

"""
Write an information prior ‘CustomInformationPrior‘ for this game that inherits from the ‘
      InformationPrior‘ class. Include all docstings from the parent class:
"""
class InformationPrior(AbstractLogged):
    '''
    Abstract class for mapping an information set to a hidden state.

    This is particularly useful when you do not have an empirical distribution over the hidden
          states for that information set
    '''
    def __init__(self, rng: np.random.Generator = np.random.default_rng()):
        self.rng = rng
        super().__init__()

    def get_prior_state(self, information_set: Hashable) -> Hashable:
        '''
        Returns the prior state for the information set. Can be stochastic
        '''
        return self._get_prior_state(information_set=information_set)

    @abstractmethod
    def _get_prior_state(self, information_set: Hashable) -> Hashable:
        '''
        Returns a hidden state corresponding to the given information set (observed state). Since
              an information set can often map to multiple hidden states, this function may return
              results stochastically. For states involving simultaneous actions, it defaults to
              returning the hidden state where no simultaneous actions have been taken yet.
        '''
        pass

"""
The player names are defined as follows:
{'guesser', 'clue_master'}
"""
```

S: Hidden States (Taboo)

SYSTEM PROMPT:

```
"""
You are a programmer developing an accurate game engine. Your task is to implement parts of the
      game simulator in Python. The simulator models simultaneous actions as sequential ones with
      partial observation. When players $1, ..., k$ take simultaneous actions, they do so
      sequentially without seeing the actions of previous players. These actions are first recorded
       in a ‘HiddenState‘ object before being revealed. Do not repetitively generate Hidden state.
      If new state is generated, construct at return statement. ‘ObservedState‘ and ‘HiddenState‘
      should be under ‘@dataclass(frozen=True)‘.  Use tuple instead of list to make sure that
      vectors are frozen.
"""
```

HUMAN PROMPT:

```
"""
Taboo (2-player text version) is a two player cooperative dialogue game where 1 player is the clue
      -master and 1 player is the guesser. The clue master is given the clue-word and a list of
      taboo words. Each discussion round the clue master makes one statement to the guesser, but
      cannot use any of the taboo words in their statements. The guesser can then guess one word.
      This continues until either the guesser guesses the word, the guesser has guess five times
      already, or the clue-master has spoken one of the taboo words. If the clue-master uses any of
       the taboo words, the team score is 0. Otherwise, the score is five minus the number of words
       guesser has guessed.
"""
"""
An observed state (information set) in the game is defined as follows:
"""
@dataclass(frozen=True)
class ObservedState:
    clue_word: Optional[str]                    # The word the guesser needs to guess
    taboo_words: Optional[tuple[str, ...]] # List of taboo words the clue-master cannot use
    guesses: tuple[str, ...] = tuple()  # List of words guessed by the guesser
    clue_master_statements: tuple[str, ...] = tuple()  # Statements made by the clue-master
    taboo_word_used: bool = False      # Flag to indicate if a taboo word was used
    game_over: bool = False            # Flag to indicate if the game is over
    score: int = 5                     # Initial score, will decrease based on guesses
    actor: str = "clue_master"          # Indicates whose turn it is: "clue_master" or "guesser"

"""
The player names are defined as follows:
{'guesser', 'clue_master'}
"""
```

T: Transition-Reward Function (Taboo)

SYSTEM PROMPT:

```
"""
You are a programmer developing an accurate game engine. Your task is to implement parts of the
      game simulator in Python. The simulator models simultaneous actions as sequential ones with
      partial observation. When players $1, ..., k$ take simultaneous actions, they do so
      sequentially without seeing the actions of previous players. These actions are first recorded
       in a ‘HiddenState‘ object before being revealed. Do not repetitively generate Hidden state.
      If new state is generated, construct at return statement. ‘ObservedState‘ and ‘HiddenState‘
      should be under ‘@dataclass(frozen=True)‘.  Use tuple instead of list to make sure that
      vectors are frozen.
"""
```

HUMAN PROMPT:

```
"""
Taboo (2-player text version) is a two player cooperative dialogue game where 1 player is the clue
      -master and 1 player is the guesser. The clue master is given the clue-word and a list of
      taboo words. Each discussion round the clue master makes one statement to the guesser, but
      cannot use any of the taboo words in their statements. The guesser can then guess one word.
      This continues until either the guesser guesses the word, the guesser has guess five times
      already, or the clue-master has spoken one of the taboo words. If the clue-master uses any of
       the taboo words, the team score is 0. Otherwise, the score is five minus the number of words
       guesser has guessed.
"""

"""
A hidden state in the game is defined as follows:
"""
class HiddenState:
    from dataclasses import dataclass
    from typing import Optional, Tuple
    clue_word: str                              # The actual clue word (hidden from guesser)
    taboo_words: Tuple[str, ...]                # Taboo words (hidden from guesser)
    taboo_word_used: bool                        # Whether a taboo word was used
    guesses: Tuple[str, ...]                    # Guesses made (hidden from clue-master)
    clue_master_statements: Tuple[str, ...]     # Statements made (hidden from guesser)

"""
Write a forward transitor ‘CustomForwardTransitor‘ for this game that inherits from the ‘
     ForwardTransitor‘ class. Include all docstings from the parent class:
"""

class ForwardTransitor(ABC):
    '''
    Abstract class for a forward dynamics transition model
    '''
    @abstractmethod
    def _transition(self, state: Hashable, action: Hashable, actor: Hashable) -> Tuple[Hashable,
         dict[Hashable, float]]:
        '''

        Args:
            state: current state
            action: action taken by the actor
            actor: actor that is taking the action. -1 for environment

        Returns:
            next_state: next state
            rewards: reward of the transition for each player
        '''
        pass

    def transition(self, state: Hashable, action: Hashable, actor: Hashable)->Tuple[Hashable, dict
         [Hashable, float]]:
        '''
        Transits to the next state given the current state, actor, and action taken by the actor.
             Transitions are deterministic, with all randomness handled by the environment actor
             and the actions it takes. If multiple actors take actions at this state, record their
              actions down one transition step at a time.

        Args:
            state: current state
            action: action taken by the actor
            actor: actor that is taking the action. -1 for environment

        Returns:
            next_state: next state
            rewards: reward of the transition for each player

        Hint:
            Initialize the new_state at the point of returning, avoid creating the new_state copy
                 prematurely.
        '''
        state, rewards = self._transition(state, action, actor)
        return state, rewards

"""
The player names are defined as follows:
{'guesser', 'clue_master'}
"""
```

F Language action examples

Example proposed possible dialogue actions

Clue word: barefoot

Taboo word: shoes, socks, summer, beach

Action 1: You might feel the ground directly under your feet when you don't wear any footwear.

Action 2: It's a way to enjoy nature by feeling the earth, grass, or sand without anything covering your feet.