From Natural Language to Extensive-Form Game Representations

Shilong Deng
University of Liverpool
Liverpool, United Kingdom
shilong.deng@liverpool.ac.uk

Yongzhao Wang
The Alan Turing Institute;
University of Liverpool
Liverpool, United Kingdom
yongzhao.wang@turing.ac.uk

Rahul Savani
The Alan Turing Institute;
University of Liverpool
Liverpool, United Kingdom
rahul.savani@liverpool.ac.uk

ABSTRACT

We introduce a framework for translating game descriptions in natural language into game-theoretic extensive-form representations, leveraging Large Language Models (LLMs) and in-context learning. We find that a naive application of in-context learning struggles on this problem, in particular with imperfect information. To address this, we introduce *GameInterpreter*, a two-stage framework with specialized modules to enhance in-context learning, enabling it to divide and conquer the problem effectively. In the first stage, we tackle the challenge of imperfect information by developing a module that identifies information sets and the corresponding partial tree structure. With this information, the second stage leverages in-context learning alongside a self-debugging module to produce a complete extensive-form game tree represented using pygambit, the Python API of a recognized game-theoretic analysis tool called Gambit. Using this python representation enables the automation of tasks such as computing Nash equilibria directly from natural language descriptions. We evaluate the performance of the full framework, as well as its individual components, using various LLMs on games with different levels of strategic complexity. Our experimental results show that the framework significantly outperforms baseline approaches in generating accurate extensive-form games, with each module playing a critical role in its success.

KEYWORDS

Code Generation; Extensive-Form Games; Gambit; Game Translation; Large Language Models

ACM Reference Format:
Shilong Deng, Yongzhao Wang, and Rahul Savani. 2025. From Natural Language to Extensive-Form Game Representations. In *Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Detroit, Michigan, USA, May 19 – 23, 2025*, IFAAMAS, 9 pages.

1 INTRODUCTION

Recently, large language models (LLMs) have shown remarkable proficiency in handling complex tasks across various domains, including code generation [4, 5, 20, 31] and question answering [15, 21, 37]. Their success has sparked interest in exploring their potential across an even broader range of applications. Within the field of multi-agent systems, a primary research direction focuses on developing LLMs' capabilities for reasoning about *games* and making decisions directly from textual information. For instance, Fu et al. [11] applied LLMs to a bargaining game where LLMs serve as bargaining agents, engaging in price negotiations across several rounds. In this scenario, a successful bargaining agent must anticipate the behavior and private information of others, which requires strong game reasoning skills.

Although there has been initial progress in this area, conducting game-theoretic analysis directly from textual descriptions (such as natural language game descriptions) remains challenging due to the varying degrees of strategic complexity in games, including imperfect information, chance events, and repeated interactions. Broadly, there are two technical approaches to tackle this task. The first approach involves training LLMs specifically to perform game-theoretic analysis. This method enables LLMs to conduct analysis directly but often requires extensive training data in games and their descriptions, as well as significant computational resources. The second approach utilizes LLMs to interpret game descriptions and generate structured representations that can then be analyzed using game-theoretic methods. Rather than equipping LLMs with full reasoning abilities, this approach integrates LLMs into the automated reasoning process, which can reduce computational demands.

In this work, we pursue the second approach, presenting *GameInterpreter*, a framework that translates natural language game descriptions into the "extensive-form". An extensive-form game (EFG) is a standard game-theoretic representation for sequential games [38]. It is a rooted *tree* with additional information structure called *information sets*, which group together decision *nodes* that are indistinguishable to a player. Our framework relies on LLMs and in-context learning, where the LLM uses context or examples in the input prompt to complete tasks without the need for fine-tuning or further training. However, due to the strategic complexities outlined above, directly describing the task in the prompts for in-context learning is insufficient. Among these complexities, we particularly emphasize the issue of imperfect information – that is, where at least one player does not have full knowledge about the current state of the game – which leads LLMs with naive in-context learning to produce incorrect game representations, as demonstrated in our experiments.

To address this, we take a divide-and-conquer approach using a two-stage process. In the first stage, we focus on any imperfect information in the game, by guiding LLMs through examples of dealing with imperfect information (e.g., identifying information sets) and the corresponding partial tree structures. With this foundation, the second stage leverages in-context learning to generate the complete EFG for the target game. The EFG is created using pygambit, the Python API for the widely used game-theoretic tool Gambit [32], which also enables automating tasks such as computing Nash equilibria from natural language descriptions. Additionally, we introduce a self-debugging module that returns pygambit error messages to the

Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Y. Vorobeychik, S. Das, A. Nowé (eds.), May 19 – 23, 2025, Detroit, Michigan, USA. © 2025 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org).

LLMs, which allows the LLM to correct the errors in its previous answers and helps to ensure that a valid EFG is created.

We assess the performance of our framework, as well as its individual components, across various LLMs, on games featuring differing levels of strategic complexity, covering different numbers of players, degrees of imperfect information, perfect/imperfect recall, and various game tree depths. We use two datasets of game descriptions, one newly designed for this paper, and another from a recent paper by others Mensfelt et al. [26]. The LLMs we employ are GPT-3.5 [3], GPT-4 [1], and GPT-4o [29], and we evaluate their ability to generate correct EFG files by incrementally adding modules until the full framework pipeline is assembled. Our findings indicate that the full pipeline significantly enhances performance across all LLMs, with the best-performing model successfully solving all test games in our custom dataset. Additionally, we confirm that each module significantly contributes to better performance. The second dataset of Mensfelt et al. [26] mainly comprises two-player simultaneous-move games, with many different game descriptions for the same underlying bimatrix game; our full pipeline achieves 100% accuracy on these games, demonstrating robustness to varying game descriptions. Our main contributions are:

(1) An in-context LLM framework for translating game descriptions in natural language into extensive-form representations;
(2) An imperfect information retrieval module that identifies information sets and the corresponding partial tree structure;
(3) A self-debugging module;
(4) A comprehensive evaluation of our framework, which demonstrate that it significantly outperforms baseline approaches.

A longer version of this paper is available, along with an associated repository containing the inputs and outputs for our experiments [7].

2 RELATED WORK

LLMs with Game Theory. Many papers have explored the use of LLMs as agents to play games, ranging from simple matrix games [2, 23, 30, 36], to much more complex environments [25, 33, 40]. Akata et al. [2] revealed the different behavioral patterns of LLMs when playing in various types of games. Shi et al. [33] examined the ability LLM agents to cooperate in the Avalon game, and developed a memory-based module to facilitate the cooperation. Xu et al. [40] tested LLM agents in the incomplete information game Werewolf. They observed emergent strategic behaviors such as deception during gameplay. Fan et al. [10] analyzed the rationality of LLMs as agents, focusing on three specific aspects: building a clear desire, refining beliefs about uncertainty, and taking optimal actions. Silva [36] explored if LLMs can be used as an equilibrium solver for games, and highlighted the difficulty of this for games with only mixed-strategy equilibria, providing enhancements to address this.

Besides using LLMs as agents, game-theoretic approaches could be utilized to improve the performance of LLMs. Gemp et al. [12] introduced a method that feeds the outputs from game-theoretic methods (e.g., an equilibrium distribution over instructions) to LLM agents in dialogues that can be formed as EFGs. They demonstrated that the integration with game-theoretic outputs could enhance the LLM generations compared to a baseline LLM that lacks access to game-theoretic supports. Ma et al. [24] studied the value alignment problem in LLMs. They gamified the attacks and counterattacks among LLMs and used equilibrium solutions to improve the level of value alignment of LLMs. Similarly, Jatova et al. [16] framed the generation of toxic content and defence against this as a strategic game between a language model and an adversarial prompt generator, with its equilibria shown to reduce harmful outputs.

Game Description Translation. We are aware of three works that directly addresses the task of game description translation [6, 26, 27]. The earliest one by Mensfelt et al. [26] is contemporaneous and independent work with ours. Rather than using EFGs, for representing games, Mensfelt et al. [26] employed logic representations as used by logic programming solvers. A further key difference between our work and theirs lies in the scope of games analyzed. Mensfelt et al. [26] focused on simultaneous-move games, with 110 bimatrix games of size 2x2[1], one bimatrix game of size 3x3, and one sequential game that corresponds to a bimatrix game of size 2x4. In their subsequent work, Mensfelt et al. [27] examined 55 simultaneous-move 2x2 bimatrix games. In contrast, our work explores more complex scenarios, with multiple sequential moves and complicated (imperfect) information structures. Both work leverage the ability of LLMs to generate code, which we discuss next. The same framework was also adopted in their follow-up work by Mensfelt et al. [27]. In this follow-up work, they presented a necessary condition for automatically verifying the correctness of the generated game. In addition to these two works, Daskalakis et al. [6] converted the sequential decision-making process described in the game derived from a story to an EFG. They achieved this by utilizing LLMs to introduce additional decision nodes, representing alternative choices players could have made in the story. Once the EFG is constructed, Gambit is used to compute the Nash equilibrium, providing a prediction of the players' behavior.

More broadly, several studies have used LLMs to translate general texts (i.e., not necessarily game descriptions) in natural language into formal specifications (i.e., required format of a software system or hardware component) [13, 18, 19, 41]. Hahn et al. [13] examined the ability of fine-tuned language models to convert natural language into formal specifications, which can be used in software verification, theorem proving, and industrial hardware. Zhai et al. [41] and Leong and Barbosa [19] focused on translating textual requirement descriptions into Java formal specifications. Leite et al. [18] employed LLMs to generate specifications for smart contracts.

LLMs for Code Generation. Code generation is the process of automatically creating source code using LLMs based on natural language task descriptions. Since the introduction of models like Codex [4], Alphacode [20], Pangu-Coder [5], and LLaMa Coder [31], general code generation has made significant advancements with the emergence of models such as LLaMa 3 [9] and GPT-4 [1]. Building on these models, many studies [17, 22, 35] further improved the performance of code generation through reinforcement learning or self-debugging prompts. In our work, we use code generation with self-debugging to generate EFG representations with pygambit.

3 PRELIMINARIES

3.1 In-Context Learning

In-context learning refers to an LLM's ability to learn new information or skills by observing examples or instructions provided in its

[1] A 2x2 bimatrix game corresponds to an EFG with 3 nodes and 4 outcomes (i.e., leaves).

input, without any additional training or fine-tuning [8]. Suppose we have an LLM, represented by a function $\mathcal{M}$. It has been trained to predict the next token or word, probabilistically mapping an input prompt to an output. An input prompt is a sequence of tokens $z = (z_1, z_2, \ldots, z_n)$. Given z, the probabilities for the next token y are then $\mathcal{M}(y|z) = P(y|z_1, z_2, \ldots, z_n)$. For in-context learning, consider an input prompt that contains a sequence (Q_i, A_i) of examples, which could be in the form of question-answer pairs, or these could be examples of how to solve examples of a specific task. We use these as context to predict an answer A_{n+1} for a new question Q_{n+1}, with our probabilistic answer being $\mathcal{M}(A|(Q_1, A_1), (Q_2, A_2), \ldots, (Q_n, A_n), Q_{n+1})$. Importantly, with in-context learning, the model does not adjust its weights, but instead uses the examples in its the context window to to refine its conditional probability for the next token.

3.2 Extensive-Form Game Representations

Extensive-form representations explicitly capture *sequential* decision making, in contrast to the strategic form, which condenses the game into a payoff matrix, interpreted as a simultaneous-move game. Therefore, EFGs are more expressive than strategic-form games, making them our preferred target format game translation. An extensive-form game consists of the following three elements:

Game tree: The central element of an EFG is a *rooted, directed tree*. Each decision node in the tree is assigned to either one of the players or to a "chance node," which represents nature[2]. Directed edges are known as moves or actions. A play of the game starts at the root and advances through the tree as the player that owns the current decision node chooses a move at the node (or a move is drawn from a discrete probability distribution at a chance node). The play ends when a terminal node (leaf) of the tree is reached.

Outcomes and payoffs: Every terminal node corresponds to an outcome, with an associated payoff vector which prescribes the payoff for each player under this outcome.

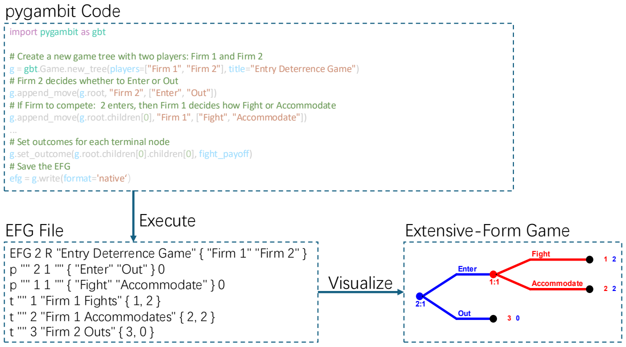
Information sets: An information set groups together nodes of a given player, with the interpretation that a player knows they are at some node in an information set, but not which one[3]. Information sets allow us to model a player's lack of knowledge of past moves of other players including nature (or the same player in the case of imperfect recall). If all information sets are singletons the game is said to have perfect information, otherwise it is said to have imperfect information.

3.3 Code Generation with Gambit

Gambit is a software suite for game-theoretic analysis [32]. Gambit can compute game-theoretic solutions such as Nash equilibria and equilibrium refinements for strategic-form games and EFGs. It has a Python API known as *pygambit*. With pygambit, we create an internal representation of an EFG and export it to a standard file format, specifically an .efg file. This file can then be analyzed or visualized using Gambit or other compatible software. Figure 1 provides an illustration.

The following pygambit functions for creating and manipulating EFGs are used in our guidance examples within our framework:

(1) **new_tree**(): create a trivial game tree with one node;
(2) **append_move**(): add a move at a terminal node;
(3) **add_outcome**(): introduce a new outcome to the game;
(4) **set_outcome**(): assign payofs to an outcome;
(5) **set_chance_probs**(): set chance node move probabilities;
(6) **set_infoset**(): assign a node to an information set.



```
import pygambit as gbt

# Create a new game tree with two players: Firm 1 and Firm 2
g = gbt.Game.new_tree(players=["Firm 1", "Firm 2"], title="Entry Deterrence Game")
# Firm 2 decides whether to Enter or Out
g.append_move(g.root, "Firm 2", ["Enter", "Out"])
# If Firm to compete:  2 enters, then Firm 1 decides how Fight or Accommodate
g.append_move(g.root.children[0], "Firm 1", ["Fight", "Accommodate"])
...
# Set outcomes for each terminal node
g.set_outcome(g.root.children[0].children[0], fight_payoff)
# Save the EFG
efg = g.write(format='native‘)
```

```
EFG 2 R "Entry Deterrence Game" { "Firm 1" "Firm 2" }
p "" 2 1 "" { "Enter" "Out" } 0
p "" 1 1 "" { "Fight" "Accommodate" } 0
t "" 1 "Firm 1 Fights" { 1, 2 }
t "" 2 "Firm 1 Accommodates" { 2, 2 }
t "" 3 "Firm 2 Outs" { 3, 0 }
```

Figure 1: Example of generating and then visualizing an EFG file for a simple Market Entry Game, via pygambit.

4 THE GAME INTERPRETER FRAMEWORK

In Figure 2, we present the full GameInterpreter framework for translating natural language game descriptions into EFG files. It involves two stages: imperfect information retrieval and complete EFG generation. In the first stage, we address the challenge of handling imperfect information by employing in-context learning to identify *non-singleton* information sets and their associated partial tree structures. The inputs for this in-context learning, detailed further below, include: general information about the task and the use of the pygambit API, a description of the target game, and instructions with examples for extracting imperfect information. At this stage, the expected output is a code block. For imperfect information games, the code block (should[4]) include a set of information sets defined by the function **set_infoset()**, which groups decision nodes that a player cannot distinguish between, accompanied by reasoning provided in the code comments. For perfect information games, the output contains only code comments, which includes a concluding statement such as "there is no need to set any information sets in this game," along with reasoning for this conclusion.

Notably, generating these information sets provides insights into the EFG tree structure. For example, Figure 3 shows an EFG with two players. Player 1 moves first by choosing one of three actions: L, C, or R. If either C or R is selected, then player 2 does not know which was selected (i.e., imperfect information). In contrast, if player 1 chooses action L, the resulting subgame has perfect information. After applying our method from stage one, the LLM can separate the perfect information part of the tree from the part with imperfect information. Then it will only assign nodes to information sets (using

[2] Chance nodes allow random behavior such as card deals in poker.

[3] Thus, all nodes in the information set must have exactly the same available moves.

[4] We cannot guarantee the LLM produces what it was asked to, but for brevity in the rest of our description we simply say that the output "includes" rather that "should include".

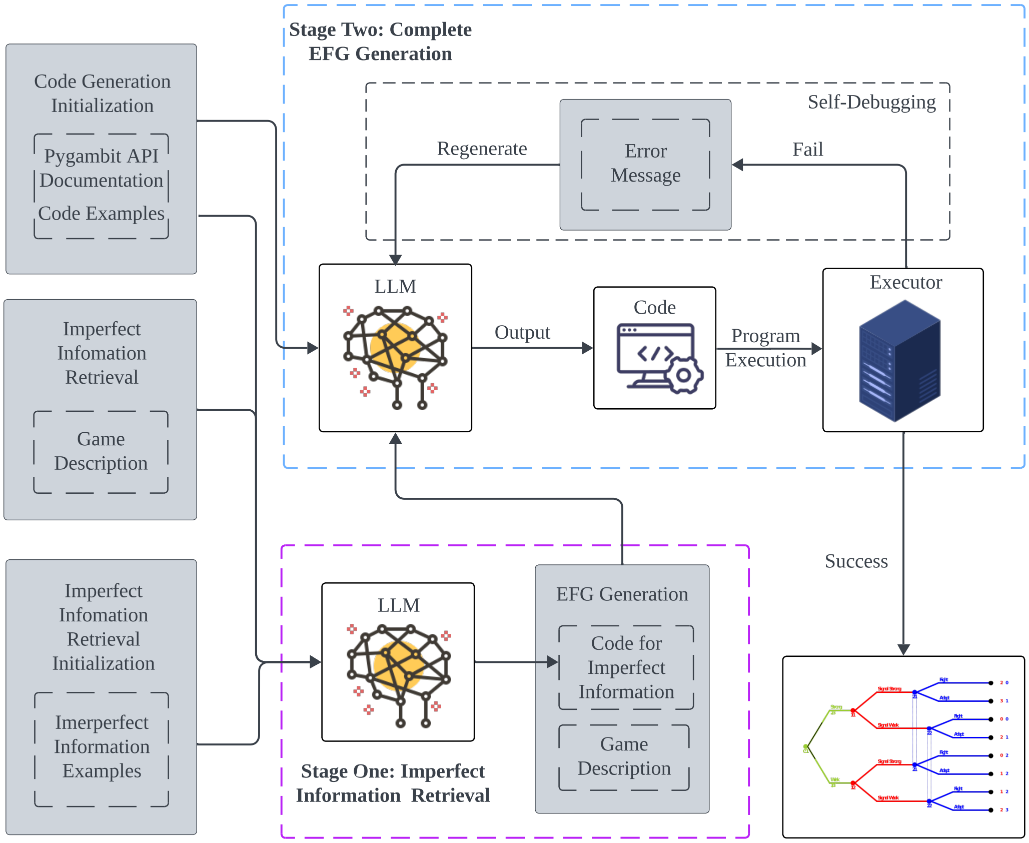

Stage Two: Complete EFG Generation
Self-Debugging
Regenerate
Error Message
Fail
Code Generation Initialization
Pygambit API Documentation
Code Examples
LLM
Output
Code
Program Execution
Executor
Imperfect Infomation Retrieval
Game Description
Success
Imperfect Infomation Retrieval Initialization
Imerperfect Information Examples
LLM
EFG Generation
Code for Imperfect Information
Game Description
Stage One: Imperfect Information Retrieval

Figure 2: Full GameInterpreter pipeline with all modules included. The rectangles with a gray background represent the prompts given to the LLM and details of these prompts are shown in Table 1.

set_infoset()) only in the imperfect information part, and it also provides a textual description of the game's tree structure[5].

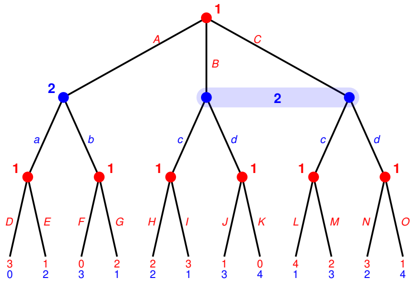


Figure 3: EFG with imperfect information.

The second stage employs in-context learning to generate a complete EFG file. The inputs for in-context learning in this stage also contain the code generation initialization, instructions for generating the entire EFG, and additionally the output from stage one. Besides, a self-debugging module is introduced to help the LLM produce runnable pygambit code. This module sends any error messages back to the LLM for code revision, providing both a description of the issue and instructions for resolving it, which are essential for fixing mistakes in previous answers. This self-debugging approach, also referred to as self-reflection, is discussed in detail by Shinn et al. [34].

Table 1 outlines the prompt templates used for the inputs of in-context learning. A detailed explanation of each template is provided below.

Code Generation Initialization (CGI). The CGI is designed to enhance the LLM's understanding of how to use the pygambit library, beyond its prior knowledge, by examining example code. We provide two examples, each containing a game description and the corresponding pygambit code for generating the EFG. We also provide additional guidance through pygambit API documentation.

Imperfect Information Retrieval Initialization (IIRI). This initialization aims to guide the LLMs to extract imperfect information from a game description and use the **set_infoset()** function to express the imperfect information. In the prompt, we present three

[5] Note that the node assignment already reveals insights into the tree structure such as the action that leads to the node in the information set.

demonstrations. Each demonstration includes a game description, the reasoning process for identifying imperfect information, and the code that groups decision nodes using the **set_infoset()** function.

Imperfect Information Retrieval. The prompt for the retrieval process includes specific guidance on the task of extracting incomplete information from a target game description. For example, we request that **set_infoset()** is used, and also ask the LLM to provide its reasoning, as in the Chain-of-Thought (CoT) methodology [39].

EFG Generation. Finally, in stage two, the EFG generation prompt asks the LLM to create an EFG file based on the target game description. For imperfect information games, it explicitly incorporates the output from stage one. For perfect information games, it utilizes the conclusion of stage one that no imperfect information is present in the games. Guidance on common pygambit bugs is provided and, again, we ask the LLM to give its reasoning as in the CoT approach.

Self-debugging. The prompts are only used if an error is encountered with the LLM's pygambit code output. The prompts guide the LLMs to correct these errors in a next attempt, and include two components: the python interpreter's error message and instructions on addressing common bugs encountered when using the pygambit library. To ensure a fair comparison, in all settings without self-debugging, if errors are encountered, we provide the LLM with a prompt requesting a new response without including any bug-related information or additional instructions. This setup isolates the effect of self-debugging when comparing to settings without it.

5 EXPERIMENTAL SETUP

In our experimental evaluation, we used two datasets: a custom dataset created specifically for this study, which focuses on sequential games with a single description provided for each underlying game, and a dataset from Mensfelt et al. [26], which emphasizes bimatrix (simultaneous-move) games and includes multiple descriptions for the same underlying game. The latter dataset is particularly useful for assessing the robustness of our method to variations in descriptions.

New custom dataset: Our 18 game descriptions in this dataset, each correspond to a different underlying game. These games have a range of different strategic complexities, as shown in Table 2, where we display certain characteristics of these games: binary features such as whether they are perfect information, or zero-sum; and numeric features such as the maximum depth of the game tree and the number of players, decision nodes, and leaves. The 18 games are chosen to cover classic games from the literature such as Kuhn poker and Tic-Tac-Toe. Some of these games were adapted from their standard forms and others were not not taken from literature or teaching materials, to mitigate the risk that the LLMs had memorized the answers to our requests[6]. For example, our game "Nim with five in one pile" is adapted from an .efg file obtained from the Gambit website. However, we modified the payoffs by switching the game from normal play to misère. Additionally, the three games listed as "Extra" Games One, Two, and Three in Table 2 were created by us and are not derived from any pre-existing materials available online. Also, games like Colonial Control and Tic-Tac-Toe did not historically have EFG files or pygambit code available on the web, so the LLM could not rely on memorization to translate these games.

Inputs	Prompts

Code Generation Initialization	Given a game description in natural language, you will be asked to generate python code for the Gambit API (pygambit) to construct a corresponding extensive-form game in Gambit. Here are two examples of how to use pygambit library:
	{CODE EXAMPLE ONE}
	{CODE EXAMPLE TWO}
	Below is the documentation for several relevant functions in the pygambit library:
	{API DOCUMENTATION}

Imperfect Information Retrieval Initialization	A challenge of this task is to represent the imperfect information in the game with pygambit. Given the game description below, please infer the imperfect information structure in the game. Make sure that if there are multiple decision nodes of a player who cannot tell among these nodes which node they are at, then these nodes are all grouped in the same information set. In short, an information set belongs to a player and should contain all nodes of that player such that the player will know that they are at one of these nodes but they will not know exactly which one they are at.
	{IMPERFECT INFO EXAMPLE ONE}
	{IMPERFECT INFO EXAMPLE TWO}
	{IMPERFECT INFO EXAMPLE THREE}

Imperfect Information Retrieval	{GAME DESCRIPTION} You MUST ONLY include the necessary **set_infoset()** functions in the Python code block. Do NOT include any other code in the code block. Think step by step and write your reasoning in comments (step-by-step thought process) within the code.

EFG Generation	{GAME DESCRIPTION}
	The CODE for representing the imperfect information of the game is as follows:
	{CODE FOR IMPERFECT INFORMATION}
	{GUIDANCE ON CODE}
	Then, could you write python code to generate the EFG for this game using the pygambit library in the example?
	Let's think step by step and write the reasoning in the code comments.

Error Message	Your code contains an error. Please review and fix it before trying again. {ERROR MESSAGE} {GENERAL GUIDANCE ON ERRORS}

Table 1: Input prompts of the framework.

Dataset from Mensfelt et al. [26]. This dataset consists of 112 descriptions in total, which correspond to: 110 bimatrix games of size 2x2, 1 bimatrix game of size 3x3, and 1 is a sequential game that corresponds to a 2x4 bimatrix game. The 110 bimatrix games are based on five classic games: Battle of the Sexes, Hawk-Dove, Matching Pennies, Prisoner's Dilemma, and Stag Hunt. Each classic

[6] In our experiments, when we ask directly for an .efg file (Setting A below) or pygambit code (Settings B,C,D below) as output, this could happen if the LLM had seen this file or code during training. To address this concern, we carefully curated our test set to include many games for which it is highly unlikely that corresponding pygambit code or .efg files were present in the training data.

game is presented in two formats: one with numerical payoffs and one without. For each format, there are a total of 11 descriptions, including one standard explanation from the game theory literature and 10 variations generated using GPT-4o, which reinterpret these games as diverse real-life scenarios.

Baselines. We evaluate the effectiveness of each module in our framework by experimenting with various settings, both including and excluding the modules. In our most basic setting, we task the LLM with generating an EFG file directly based on a game description, without requiring it to utilize the pygambit API. We then investigate four settings that require the pygambit API:

- **Setting A**: The setting that utilizes minimal EFG generation prompts in Table 1, without incorporating additional information for imperfect information retrieval;
- **Setting B**: Setting A with self-debugging;
- **Setting C**: Setting A with imperfect information retrieval;
- **Setting D (Full Pipeline)**: Setting A with both self-debugging and imperfect information retrieval.

Figure 4 provides an illustration. For each of these settings, we combine them with three different LLMs. Additionally, we compare our setting D, the full pipeline, with the approach by Mensfelt et al. [26], which uses logic programming to represent target games.

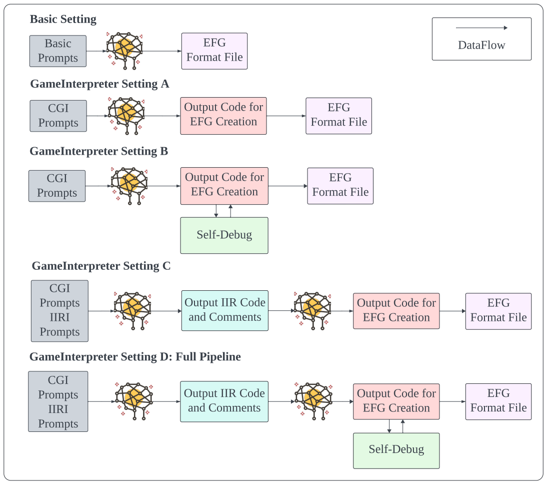

Basic Setting
Basic Prompts
EFG Format File
DataFlow
GameInterpreter Setting A
CGI Prompts
Output Code for EFG Creation
EFG Format File
GameInterpreter Setting B
CGI Prompts
Output Code for EFG Creation
EFG Format File
Self-Debug
GameInterpreter Setting C
CGI Prompts IIRI Prompts
Output IIR Code and Comments
Output Code for EFG Creation
EFG Format File
GameInterpreter Setting D: Full Pipeline
CGI Prompts IIRI Prompts
Output IIR Code and Comments
Output Code for EFG Creation
EFG Format File
Self-Debug

Figure 4: The five approaches we evaluated.

Evaluation. When translating game descriptions into extensive-form representations, a key task is to ensure that the generated EFG is *consistent* with the description. Formally, a game description corresponds to a family of EFGs that are consistent with that description; this family can vary in size, and could possibly even by infinite, depending on the specificity of the game details provided. For instance, if a game description includes inequalities or relationships between payoffs but lacks precise values, many payoff assignments may be consistent with it, with different corresponding EFGs. Similarly, for descriptions of simultaneous-move games (e.g., bimatrix games), a consistent EFG could depict either of the two players acting first, with imperfect information modeling the simultaneity of their choices.

For checking the consistency, we manually assess whether the game tree, information sets, and payoffs in the generated EFG are consistent with the target game description. This requires firstly checking the generated tree structure, including which players move at which nodes, starting from the root, along with how the actions at the nodes lead to moves of other players; this is informed by the labels for moves that are provided in the generated EFG files. For information sets, we ensure that nodes indistinguishable to a given player are grouped within the same information set. Finally, for payoffs, if specific values are provided in the game description, we check that the generated payoffs at the appropriate terminal nodes (outcomes) match these values. If the game description only implies relative payoff strengths, we verify that the payoffs are consistent with the implied constraints.

We use *pass@k* (the solve rate given k "samples", that is, independently generated EFG files), as proposed by Chen et al. [4], to measure the success of our translations for a specific target game description. Specifically, we execute GameInterpreter $k = 5$ times in our experiments, and we record the total number of generated EFGs, $s \in [1, \ldots, 5]$, that are consistent with the target game description. As described in more detail in Section 6, in our experimental analysis, we distinguish between the case where at least one sample was correct (i.e., $s \geq 1$), so we "*pass*@5", and the case where all samples were correct (i.e., $s = 5$), referred to as "*pass all* 5", which is stricter than *pass*@5. While improvements under *pass all* 5 are more desirable, we also evaluate *pass*@5 to highlight any relative merits of different settings we investigate, particularly in cases where the strictness of *pass all* 5 might hide differences in performance.

Parameters. We use the OpenAI API to access various LLMs, in particular: gpt-4-0125-preview, gpt-4o, and gpt-3.5-turbo. All of these models have two key hyperparameters that relate to how the next tokens are chosen, namely the temperature and p threshold for top-p sampling; both take values in $[0, 1]$. In top-p sampling [14], also known as nucleus sampling, the threshold p is used to restrict sampling of the next token to only the smallest set of most-likely candidates whose cumulative probability exceeds p. We set p as 1 (i.e., we do not restrict the next tokens at all). We set the temperature of LLMs to 0, which minimizes the amount of randomness in the chosen tokens (a choice of 1 would maximize it); note that setting the temperature to 0 makes the output as deterministic as possible, but, even with a fixed prompt, the output of these LLMs still often varies in repeat trials with the temperature set to 0, which has been attributed to issues like multi-GPU inference with varying GPU clock times. When comparing our method to the work by Mensfelt et al. [26], we match their experimental settings and adjust the temperature to 1. The maximal number of attempts for self-debugging is set to 3.

6 EXPERIMENTAL RESULTS

Overview. Table 3 displays the performance of all settings across the 18 games of our custom dataset. A grey tick indicates that between 1 and 4 of the 5 generated samples were successfully solved, a red cross means none were solved, and a green tick indicates that all 5 samples were solved. Thus a grey tick indicates a *pass*@5 and a green tick indicates a *pass all* 5 (which is by definition also a *pass*@5).

For all LLMs, setting D, the full pipeline, outperformed its counterparts. Among the LLMs, GPT-4o achieved the highest performance,

Game Names	Game Features					
	Perfect Info	Zero-Sum	Max Depth	#Players	#Nodes	#Leaves
A Three-Player Game	✗	✓	4	3	7	8
An Imperfect Recall Game	✗	✓	3	2	7	8
Absent-Minded Driver	✗	✗	2	1	2	3
Bach or Stravinsky	✗	✗	2	2	3	4
Bagwell	✗	✗	3	2	7	8
Kuhn Poker	✗	✓	4	2	25	30
Extra Game One	✗	✗	5	2	16	21
Extra Game Two	✗	✗	5	3	22	24
Market Signalling Game	✗	✗	3	2	7	8
Nuclear Crisis	✗	✗	4	2	5	6
Rock, Paper, Scissors	✗	✓	2	2	4	9
Centipede	✓	✗	4	2	4	5
Colonial Control	✓	✗	3	2	4	5
Extra Game Three	✓	✓	4	2	17	24
Market Entry Model	✓	✗	2	2	2	3
Nim (with five in one pile)	✓	✓	5	2	12	8
Simple Bargaining Game	✓	✗	5	2	5	3
Tic-Tac-Toe	✓	✓	3	2	5	5

Table 2: The games in our custom dataset used in our evaluation, along with their features.

succeeding for *pass*@5 for all 18 games, while GPT-4 also performed well, succeeding on 15 of the games, and failing only Kuhn poker, Nim, and Extra Game Two, which we note are three of the largest games that we considered. In the basic setting, as well as in settings A and B, we found that imperfect information games, like "A Three-Player Game", are challenging to solve. This motivates our approach using a first stage for imperfect information retrieval. Their poor performance shows that the LLMs had not effectively memorized the solutions for our custom dataset.

Across all settings, GPT-3.5 underperforms GPT-4 and GPT-4o, and it does not benefit from the addition of imperfect information retrieval and self-debugging. Ultimately, we see that setting D, the full pipeline, yields the best performance across all configurations.

Performance of Self-Debugging. Table 4 presents the *pass*@5 and *pass all* 5 metrics (extracted from Table 3) with and without the self-debugging module for each LLM. Under *pass*@5, both GPT-3.5 and GPT-4o show improvements, while GPT-4 remains unchanged. Under *pass all* 5, both GPT-4 and GPT-4o improve, whereas GPT-3.5 fails in all games with or without self-debugging, with the performance constrained by the limitations of the LLM itself. These findings show that the self-debugging module contributes to an overall enhancement in the framework's performance.

Performance of Imperfect Info Retrieval. In Table 5, we examine the impact of the imperfect information retrieval module by comparing setting B with the full pipeline, under the *pass*@5 and stricter *pass all* 5 metrics (extracted from Table 3). We distinguish between imperfect information games and perfect information games to analyze the module's effect on each category. Table 5 shows that the imperfect information retrieval module significantly enhances the performance of GPT-4 and GPT-4o, increasing the number of imperfect information games passed under *pass*@5. Notably, GPT-4o benefits the most from the module. In contrast, for GPT-3.5, no performance improvement was observed, suggesting that LLM itself is at fault. Finally, we note that for GPT-4 and GPT-4o, translation of perfect information games was also better with the module. We attribute this to use of details of the reasoning process generated in stage one, which aids the LLM in stage two by improving its ability to identify game types and the associated tree structures.

Experiments with Mensfelt et al. [26] Games. We further evaluate the performance of our framework using the 112 game descriptions from Mensfelt et al. [26]. For a comparison with their experimental results, we adopt their approach to regeneration attempts: rather than always performing a fixed number of attempts, they stop on the first successful attempt, or give up after 5 failed attempts.

Mensfelt et al. [26] used a logic programming approach. Across the five underlying bimatrix games in their dataset, they achieved 100% accuracy for the Hawk-Dove and Stag Hunt games, but encountered errors for descriptions of the Prisoner's Dilemma, Battle of the Sexes, and Matching Pennies games. With the same setup, our approach correctly translated all 112 of their game descriptions. For 102 of these 112 cases, the first correct translation was on the first attempt, the remaining 10 were correctly translated on the second attempt. This demonstrates the robustness of our method to variations in game descriptions in the context of 2x2 games (as future work, it would be good to explore this for more complex games too). We note that we were able to immediately apply our approach to their setting because EFGs are more general than bimatrix games. In contrast, their approach cannot be directly applied to generate EFG files, so we could not test it on EFG file generation with our custom dataset.

Games	Basic Setting			Setting A			Setting B			Setting C			Setting D		
	3.5	4	4o	3.5	4	4o	3.5	4	4o	3.5	4	4o	3.5	4	4o
A Three-Player Game	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓[4]	✓[2]	✗	✓[4]	✓[2]
An Imperfect Recall Game	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓[4]	✓[2]	✗	✓[4]	✓[4]
Absent-Minded Driver	✗	✗	✗	✗	✓[1]	✗	✗	✓[2]	✗	✗	✓[4]	✗	✗	✓[4]	✓[2]
Bach or Stravinsky	✗	✗	✗	✗	✗	✗	✓[1]	✗	✗	✗	✓[4]	✓	✓[1]	✓[4]	✓
Bagwell	✗	✗	✗	✗	✗	✓[4]	✗	✗	✓[4]	✗	✓[1]	✓[4]	✗	✓[1]	✓[4]
Kuhn Poker	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓[2]	✗	✗	✓[2]
Extra Game One	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓[3]	✓[4]	✗	✓[4]	✓[4]
Extra Game Two	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓
Market Signalling Game	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗	✓[2]	✓[4]	✗	✓[2]	✓
Nuclear Crisis	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓[2]	✓[2]	✗	✓[2]	✓[2]
Rock, Paper, Scissors	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓[2]	✓[3]	✗	✓[2]	✓
Centipede	✗	✗	✗	✗	✓[2]	✓	✗	✓[4]	✓	✗	✓[4]	✓	✗	✓	✓
Colonial Control	✗	✓	✓	✗	✓	✓	✗	✓	✓	✗	✓	✓	✗	✓	✓
Extra Game Three	✗	✓[1]	✗	✗	✓	✓[2]	✗	✓	✓[2]	✗	✓[3]	✓	✗	✓	✓
Market Entry Model	✗	✓[2]	✓[2]	✗	✓	✓	✗	✓	✓	✗	✓	✓	✗	✓	✓
Nim (with five in one pile)	✗	✗	✗	✗	✗	✓[4]	✗	✗	✓[4]	✗	✗	✓	✗	✗	✓
Simple Bargaining Game	✗	✗	✓[3]	✗	✓	✓	✗	✓	✓	✗	✓[3]	✓	✗	✓	✓
Tic-tac-toe	✗	✓	✓	✗	✓	✓	✗	✓	✓	✗	✓[4]	✓	✗	✓	✓

Table 3: Results of EFG file generation under various settings. A red cross indicates that none of the generated samples passed; a green tick shows that all five generated samples passed; a grey tick signifies that between 1 and 4 generated samples passed, with the exact number of successes shown in brackets. Perfect information games (top) and imperfect information games (bottom) are separated.

Metrics	GPT-3.5		GPT-4		GPT-4o	
	Setting C	Setting D	Setting C	Setting D	Setting C	Setting D
pass@5	0	1	15	15	17	18
pass all 5	0	0	2	6	9	11

Table 4: Performance with and without self-debugging.

Game Types	Metrics	GPT-3.5		GPT-4		GPT-4o	
		Setting B	Setting D	Setting B	Setting D	Setting B	Setting D
Imperfect info	*pass*@5	1	1	1	9	2	11
(11 games)	*pass all* 5	0	0	0	0	1	4
Perfect info	*pass*@5	0	0	6	6	7	7
(7 games)	*pass all* 5	0	0	5	6	5	7
All games	*pass*@5	1	1	7	15	9	18
(18 games)	*pass all* 5	0	0	5	6	6	12

Table 5: Performance with and without IIR.

7 CONCLUSION AND DISCUSSION

We introduced a two-stage framework for translating game descriptions in natural language into EFG representations, leveraging LLMs, in-context learning, and code generation with pygambit. The first stage addresses imperfect information via a module that identifies information sets and the corresponding partial game tree structure. In the second stage, the output from stage one is used, along with a self-debugging module, to generate a complete EFG using pygambit. We evaluate the framework's overall performance, as well as its individual components, across three LLMs on 18+112=130 game descriptions spanning 21 different games[7]. Our experimental results show that the framework significantly outperforms baseline models in generating accurate EFGs, with each module playing a critical role in its success.

One potential direction for future work is to move beyond manual consistency checks and develop a robust, automated check. Specifically, automated validation may require a "suite" of checks covering various aspects of consistency. For instance, α-rank [28] could be applied to compare rankings of strategies between target and generated EFGs; more generally different types of strategic equivalence could used to design checks. Other checks could involve game feature extraction, such as identifying the number of players, possible outcomes, and information sets. Note that it is essential to design these checks carefully to avoid introducing sources of error.

Another direction of further work relates to the fact that many games are actually parameterized[8]. A natural extension of our framework would take a description of a parameterized game family, and would then generate, instead of a single EFG file, a python function, for example, using pygambit, that takes game parameters as inputs and generates a corresponding EFG file. The development of such an extension would benefit from the previously mentioned automated checking of the consistency of an EFG file (given a parameterized game description and a specific choice of parameters).

Finally, we believe it is important to explore game description translation for larger and more complex games. In our custom dataset, Kuhn Poker, with 25 decision nodes and 30 terminal nodes, was the largest game. To tackle larger games, two significant challenges must be addressed: the generation method's capacity to handle the increased complexity and our ability to accurately validate the method's outputs. Progress on either direction mentioned above could help with this: solving parametrized games could be part of a divide and conquer approach to translating complex games, and robust automated checking of translation outputs would help with the second challenge. Additionally, alternatives to in-context learning such as supervised fine-tuning could also be effective for solving such problem if a suitable game dataset is available.

[7] Bach or Stravinksy is the Battle of the Sexes from Mensfelt et al. [26], and Rock Paper Scissors appears in both datasets, so we have 18-2+5=21 distinct games overall.

[8] For instance, parameters include the number of stages in the Centipede game, the pile sizes in Nim, or both the number of stages and the discount factor in bargaining games.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Elif Akata, Lion Schulz, Julian Coda-Forno, Seong Joon Oh, Matthias Bethge, and Eric Schulz. 2023. Playing repeated games with Large Language Models. *arXiv preprint arXiv:2305.16867* (2023).

[3] Tom B Brown. 2020. Language Models are Few-Shot Learners. *arXiv preprint arXiv:2005.14165* (2020).

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[5] Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, et al. 2022. PanGu-Coder: Program Synthesis with Function-Level Language Modeling. *arXiv preprint arXiv:2207.11280* (2022).

[6] Constantinos Daskalakis, Ian Gemp, Yanchen Jiang, Renato Paes Leme, Christos Papadimitriou, and Georgios Piliouras. 2024. Charting the Shapes of Stories with Game Theory. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[7] Shilong Deng, Yongzhao Wang, and Rahul Savani. 2025. From Natural Language to Extensive-Form Game Representations. *arXiv preprint arXiv:2501.17282* (2025). Code and results repo: https://github.com/zczlsde/GameInterpreter.

[8] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A Survey on In-context Learning. *arXiv preprint arXiv:2301.00234* (2022).

[9] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024).

[10] Caoyun Fan, Jindou Chen, Yaohui Jin, and Hao He. 2024. Can Large Language Models Serve as Rational Players in Game Theory? A Systematic Analysis. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 17960–17967.

[11] Yao Fu, Hao Peng, Tushar Khot, and Mirella Lapata. 2023. Improving Language Model Negotiation with Self-Play and In-Context Learning from AI Feedback. *arXiv preprint arXiv:2305.10142* (2023).

[12] Ian Gemp, Roma Patel, Yoram Bachrach, Marc Lanctot, Vibhavari Dasagi, Luke Marris, Georgios Piliouras, Siqi Liu, and Karl Tuyls. 2024. Steering Language Models with Game-Theoretic Solvers. In *Agentic Markets Workshop at International Conference on Machine Learning (AMW@ICML)*.

[13] Christopher Hahn, Frederik Schmitt, Julia J Tillman, Niklas Metzger, Julian Siber, and Bernd Finkbeiner. 2022. Formal Specifications from Natural Language. *arXiv preprint arXiv:2206.01962* (2022).

[14] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *Proceedings of the International Conference on Learning Representations ICLR*.

[15] Dengrong Huang, Zizhong Wei, Aizhen Yue, Xuan Zhao, Zhaoliang Chen, Rui Li, Kai Jiang, Bingxin Chang, Qilai Zhang, Sijia Zhang, et al. 2023. DSQA-LLM: Domain-Specific Intelligent Question Answering Based on Large Language Model. In *Proceedings of International Conference on AI-generated Content (AIGC)*. 170–180.

[16] Lucas Jatova, Jacob Smith, and Alexander Wilson. 2024. Employing Game Theory for Mitigating Adversarial-Induced Content Toxicity in Generative Large Language Models. *TechRxiv* (2024).

[17] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 35. 21314–21328.

[18] Gabriel Leite, Filipe Arruda, Pedro Antonino, Augusto Sampaio, and AW Roscoe. 2024. Extracting Formal Smart-Contract Specifications from Natural Language with LLMs. In *Proceedings of the International Conference on Formal Aspects of Component Software (FACS)*, Vol. 15189. 109–126.

[19] Iat Tou Leong and Raul Barbosa. 2023. Translating Natural Language Requirements to Formal Specifications: A Study on GPT and Symbolic NLP. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 259–262.

[20] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-Level Code Generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097.

[21] Zhenyu Li, Sunqi Fan, Yu Gu, Xiuxing Li, Zhichao Duan, Bowen Dong, Ning Liu, and Jianyong Wang. 2024. FlexKBQA: A Flexible LLM-Powered Framework for Few-Shot Knowledge Base Question Answering. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 18608–18616.

[22] Jiate Liu, Yiqin Zhu, Kaiwen Xiao, QIANG FU, Xiao Han, Yang Wei, and Deheng Ye. 2023. RLTF: Reinforcement Learning from Unit Test Feedback. *Transactions on Machine Learning Research* (2023).

[23] Nunzio Lorè and Babak Heydari. 2023. Strategic Behavior of Large Language Models: Game Structure vs. Contextual Framing. *arXiv preprint arXiv:2309.05898* (2023).

[24] Chengdong Ma, Ziran Yang, Minquan Gao, Hai Ci, Jun Gao, Xuehai Pan, and Yaodong Yang. 2023. Red Teaming Game: A Game-Theoretic Framework for Red Teaming Language Models. *arXiv preprint arXiv:2310.00322* (2023).

[25] Shaoguang Mao, Yuzhe Cai, Yan Xia, Wenshan Wu, Xun Wang, Fengyi Wang, Tao Ge, and Furu Wei. 2023. ALYMPICS: Language Agents Meet Game Theory. *arXiv preprint arXiv:2311.03220* (2023).

[26] Agnieszka Mensfelt, Kostas Stathis, and Vince Trencsenyi. 2024. Autoformalization of Game Descriptions using Large Language Models. *arXiv preprint arXiv:2409.12300* (2024).

[27] Agnieszka Mensfelt, Kostas Stathis, and Vince Trencsenyi. 2024. Autoformalizing and Simulating Game-Theoretic Scenarios using LLM-augmented Agents. *arXiv preprint arXiv:2412.08805* (2024).

[28] Shayegan Omidshafiei, Christos Papadimitriou, Georgios Piliouras, Karl Tuyls, Mark Rowland, Jean-Baptiste Lespiau, Wojciech M Czarnecki, Marc Lanctot, Julien Perolat, and Remi Munos. 2019. α-rank: Multi-agent evaluation by evolution. *Scientific Reports* 9, 9937 (2019), 29.

[29] OpenAI. 2024. GPT-4o System Card. https://cdn.openai.com/gpt-4o-system-card.pdf Accessed: 2024-10-13.

[30] Kristijan Poje, Mario Brcic, Mihael Kovac, and Marina Bagic Babac. 2024. Effect of Private Deliberation: Deception of Large Language Models in Game Play. *Entropy* 26, 6 (2024), 524.

[31] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023).

[32] Rahul Savani and Theodore L. Turocy. 2024. *Gambit: The package for computation in game theory, Version 16.2.0.* http://www.gambit-project.org Version 16.2.0.

[33] Zijing Shi, Meng Fang, Shunfeng Zheng, Shilong Deng, Ling Chen, and Yali Du. 2023. Cooperation on the Fly: Exploring Language Agents for Ad Hoc Teamwork in the Avalon Game. *arXiv preprint arXiv:2312.17515* (2023).

[34] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*. 8634–8652.

[35] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based Code Generation using Deep Reinforcement Learning. *Transactions on Machine Learning Research* (2023).

[36] Alonso Silva. 2024. Large Language Models Playing Mixed Strategy Nash Equilibrium Games. *arXiv preprint arXiv:2406.10574* (2024).

[37] Yiming Tan, Dehai Min, Yu Li, Wenbo Li, Nan Hu, Yongrui Chen, and Guilin Qi. 2023. Can ChatGPT Replace Traditional KBQA Models? An In-Depth Analysis of the Question Answering Performance of the GPT LLM Family. In *Proceedings of the International Semantic Web Conference (ISWC)*, Vol. 14265. 348–367.

[38] Bernhard von Stengel. 2021. *Game Theory Basics*. Cambridge University Press.

[39] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 35. 24824–24837.

[40] Yuzhuang Xu, Shuo Wang, Peng Li, Fuwen Luo, Xiaolong Wang, Weidong Liu, and Yang Liu. 2023. Exploring Large Language Models for Communication Games: An Empirical Study on Werewolf. *arXiv preprint arXiv:2309.04658* (2023).

[41] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: Translating Natural Language Comments to Formal Program Specifications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*. 25–37.