TU WIEN Informatics

Automatically Testing the Out-of-Distribution Reasoning Capabilities of LLMs/AGI Systems with Generative Formal Games

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Data Science

eingereicht von

Philip Vonderlind, BSc
Matrikelnummer 11739128

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Inf. Dr.rer.nat. Thomas Lukasiewicz
Mitwirkung: Simon Frieder, MSc, MPhil

Wien, 26. August 2024

Philip Vonderlind	Thomas Lukasiewicz

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Automatically Testing the Out-of-Distribution Reasoning Capabilities of LLMs/AGI Systems with Generative Formal Games

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Data Science

by

Philip Vonderlind, BSc
Registration Number 11739128

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Inf. Dr.rer.nat. Thomas Lukasiewicz
Assistance: Simon Frieder, MSc, MPhil

Vienna, August 26, 2024

Philip Vonderlind	Thomas Lukasiewicz

Erklärung zur Verfassung der Arbeit

Philip Vonderlind, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 26. August 2024

Philip Vonderlind

Acknowledgements

This thesis would not have been possible without the support of my supervisor, Simon Frieder, who guided me through the ups and downs of the research process. The advice I received during my journey not only helped me finish this work but also gave me insights into my future work and career. Furthermore, I want to thank all of the professors, assistants, and everyone else who provided me with the knowledge I acquired at Vienna University of Technology during my time here. Finally, I would like to dedicate the results of this work to my family and friends for providing me with the emotional support required to push through the final phase of my studies at Vienna University of Technology.

Abstract

In recent years, large language models have become prominent tools for a variety of reasoning tasks, ranging from coding to solving puzzles. However, as most of these models are offered as black-box online services, testing their reasoning performance using traditional benchmark datasets may not reflect their true capabilities due to the memorization of public data. To solve this problem, I establish as a main contribution a framework that automatically generates structured datasets of formal games, which can be used to evaluate the out-of-distribution capabilities of language models offered as a service. The games generated by our framework are based on a novel domain-specific language I call *Grid-Games*. Furthermore, I introduce a complexity metric that categorizes each generated game based on intrinsic task difficulty. To test the distribution shift of known games and generated games, I conduct experiments on three prominent language models and compare the performances. Our main finding is that there exists a large shift in performance between the new and unknown *Grid-Games*, which are not included in any training data, and the known game of *Tic-Tac-Toe*, that I used as an exemplary game that likely was in the training data of all tested large language models.

Contents

CHAPTER 1

Introduction

Since the inception of ChatGPT in November of 2022,[1] large language models (LLMs) have become a household name. As with any technical artifact, continuously testing these models to measure their performance is necessary to ensure stable results. However, for most of the current state-of-the-art models, such as GPT 4o [Ope], Claude 3 [AI], and others, there is no indication of which data was used for training, even if weights are public. As with any LLM, test data should be out-of-distribution. To approach this issue, I propose automatically generating structured datasets of formal games to evaluate LLMs' out-of-distribution (OOD) reasoning capabilities.

The previous methods indirectly assess the state-of-the-art systems of artificial intelligence by measuring whether a system was able to solve the game (and how well the system compares to humans –such as by using Elo scores [Elo78]– over multiple games, or whether it beats humans on average). Our approach is universal and can assess any LLM or AGI system that receives textual inputs and outputs.

Our main use case pertains to language-model-as-a-service (LMaaS) context [MPF+24], such as GPT-4 [Ope23], Claude 2.1 [Ant], Bard [Gooa], Gemini [Goob], and others, where users have little control over the LLM, since full papers, and, in some cases, even model cards, are missing.

Since new LLM models are silently released, it is necessary to continuously test the models in production environments to ensure that their performance on production tasks remains unchanged. For reasoning tasks, this is challenging since typically little is known about the training data LMaaS ingest [Ope23]. If a model has good performance on a dataset, this does not exclude that the model has simply memorized it - hence the need for generative datasets. By using a *data generator* instead of a *data set* to generate new unseen data points, our approach is largely immune to the current widespread problem that public evaluation benchmarks of LLMs are rendered meaningless in this way.

[1] https://openai.com/index/chatgpt/

At a high level, our approach works by probabilistically sampling an entire game out of an infinite number of possible games, many of which are new and not covered by existing training data sets or other public datasets, together with a synthetic solver that supervises the games' reasoning steps and verifies a reasoning step candidate is correct. These are embedded in a framework where such a generated game is explained, in natural language, to an LLM. The LLM then has to solve several challenges related to the game, that are designed to test the cognitive ability of game understanding:

- Understanding the description of the game's rules and win conditions.
- Reading the natural language representation of the current game state and relating it to the "original" mode of representation (e.g. for board games, I may choose to represent a grid as a list of assignments for each position).
- Deducing which actions are legal using the game state and the rules.
- Taking an action that may lead to satisfying a win condition given by the rules.

An overview of our approach is displayed in Figure 1.1. The Solver records the LLM output generated by the Prompter and monitors the LLMs' progress in playing (a part of) a sampled game. The Metric module then aggregates all of the played-out game steps to assess the level of intelligence displayed by the LLM. This entire pipeline is fully automatic.

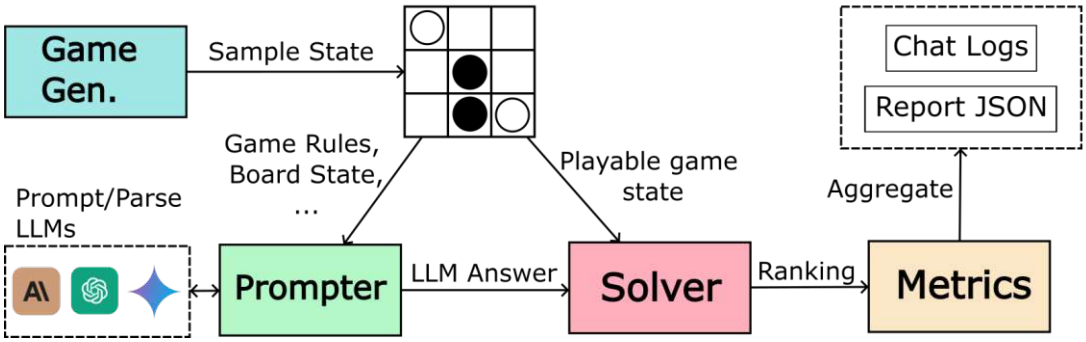

Figure 1.1: Architecture of our approach to generate OOD data

Concluding, I identify the following problems that current reasoning benchmarks for LLMs experience:

- LLMs are trained on data that, therefore, should not used as testing data;
- If new test datasets are offered publicly, it is not possible to exclude them from potentially being used to pre-train LLMs, which invalidates that dataset for reasoning capability evaluation.

Our approach solves both of these problems. To analyze the differences between in-distribution games and those generated by our framework, I arrive at the following research question for this thesis:

*Is there a **distribution shift** between out-of-distribution games generated by our approach and in-distribution games?*

To answer this research question, I will propose a framework that can automatically generate and evaluate formal games in a form that is processable by LLMs. Furthermore, I will provide a complexity measure that is used to categorize the generated games in Section 3.3.

Since the games are generated and diverse, I expect that fine-tuning on an existing subset of formal games will not dramatically affect overall reasoning capabilities, as benchmarked on another generated game. I note, however, that if large numbers of formal games are generated, and the LLMs are fine-tuned on them, I expect it to raise the reasoning abilities of LLMs – this is consistent with the fact that humans, when systematically training their formal reasoning skills on various tasks, gradually become better. Nonetheless, this doesn't invalidate our approach.

Summarizing, our contributions are:

- *I introduce, to our knowledge, the first fully automatic regression test based on games for text-based systems, such as LLMs, that measures the reasoning capabilities of such a system in an out-of-distribution way (Section 3.4);*
- *Our approach is modular, and thus future-proof, as tests can be made more complex as LLMs evolve towards AGI systems (Section 3.4);*
- *I survey the current state-of-the-art in multiple domains and categorize their approaches (Section 2);*
- *I provide experiments that aim to show the distribution shift between generated games and an in-distribution game (Section 3.4.1).*

CHAPTER 2

Related Work

Our approach combines five important elements:

- Automatic Evaluation of LLMs;
- Solving Formal Games using LLMs;
- Generating Formal Games using LLMs;
- Analysis of LLMs in the Context of Formal Games;
- Solving Formal Games using Formal Methods.

For each of these areas, I analyze the current state-of-the-art and split the approaches into multiple sub-categories. For a few select papers, I provide a more in-depth comparison between our work and theirs in Section 2.1.

2.1 Automatic Evaluation

In our review of the current literature, I have found four approaches that aim to solve the problem of automatic evaluation of text artifacts. The following sections will describe the current state of the art and related works for each approach.

2.1.1 Generating Executable Code

[OSG+23] introduce BioProt, a dataset of biology protocols that are represented in natural language and pseudocode from a set of admissible pseudocode functions. To automate the evaluation of scientific experiments in a biology laboratory, the authors utilize GPT-3 [BMR+20] and GPT-4 [Ope23] to automatically reconstruct pseudocode from a natural language description of the experiments. This allows them to easily measure the performance of the experiment's approach using pseudocode.

2.1.2 Prompting GPT Models with an Evaluation Schema

In recent years, there has been a growing body of research focused on the automatic evaluation of text generated by Large Language Models (LLMs) in various NLP tasks. Notably, several studies have investigated novel evaluation protocols and methodologies that leverage LLMs as evaluators. [FNJL23] introduces a method for defining evaluation protocols for specific evaluation aspects and tasks and obtaining scores based on how closely the generated text aligns with the evaluation requirements. Similarly, [WLM+23] explore the concept of instructing an LLM to evaluate text based on task instructions and providing a score.

Building on this foundation, [LIX+23] present a method where an LLM generates an evaluation protocol using Auto-CoT and the evaluators' output is formatted as a form for the final scoring. [CH23] propose a question-based approach, asking GPT-3 to assess specific aspects of generated text on a numerical scale. [YKG+23] propose a similar approach, Skill-Mix, which aims to test different NL-Skills of LLMs (metaphors, allegories, etc.) from a list of pre-defined skills and topics. They utilize GPT-4 [Ope23] (and some open models) as a Grader LLM with a specific prompt template, which assigns points for each skill found in the generated text.

In addition to these studies, [LC23] extended the evaluation process by using a single-prompt template and returning evaluations in JSON schema format. [WLC+23] introduce techniques to improve zero-shot evaluation using LLMs, involving few-shot prompting, among other strategies.

Furthermore, [ZWS+23] conducted experiments on improving LLM judging abilities for reasoning tasks and introduced benchmarks (MT-Bench and ChatbotArena) for comparing LLMs to human ratings. Lastly, [LMG23] focuses on enhancing scoring by employing pairwise comparisons between different generations of text. [CL23] improved upon the GPT-Eval approach by allowing LLMs to provide explanations and rationalizations for their evaluations.

These diverse sources collectively contribute to the evolving landscape of automatic evaluation methodologies, shedding light on the potential of LLMs as alternative evaluators and offering valuable insights for the evaluation of text generated by LLMs.

2.1.3 Finetuning an LLM on an Evaluation Dataset

A notable advancement is presented in [WYZ+23], where the authors establish a benchmark tailored for evaluating the instruction tuning optimization of LLMs. They showcase a fine-tuned instance of LLama [TLI+23], PandaLM, optimized for human-aligned performance, presenting it as a more cost-effective alternative to GPT-4 for automatic evaluation tasks. Furthermore, the work [WYT+23] introduces a small 7B parameter model named LL23, fine-tuned to act as a critic by providing refinement suggestions and identifying errors in outputs from other LLMs, aiming to enhance alignment and accuracy in generated responses.

Moreover, the [LSY+23] paper extends the concept of fine-tuning a LLama model on a specially crafted dataset for evaluation, paralleling the approach of PandaLM. The authors propose Auto-J, a 13B parameter model trained on user queries and LLM-generated responses on real-world tasks, which they compare to GPT-4. Lastly, [KK23] broadens the scope by presenting various decoder-based LLMs fine-tuned for tasks such as machine translation and semantic similarity assessment. These models are evaluated across multiple benchmarks.

2.1.4 Using an Ensemble of LLMs

[LLL23] propose an evaluation method that moves beyond traditional static dataset evaluations. It involves multiple LLMs assuming various roles based on a task description. These LLMs engage in multi-round discussions to solve and evaluate a problem, with a referee LLM overseeing the process and aggregating final evaluations from participant LLMs. Roles in this framework can include positions like programmers and code reviewers, allowing for a dynamic and interactive evaluation process.

Complementing this, [SPH+23] introduces an approach where an LLM is used with few-shot prompting to generate evaluation "rubrics". These templates are then applied to assess the intermediate steps of LLM generations using another LLM. This method provides a structured and nuanced way to evaluate LLM outputs, focusing on the process rather than just the result.

Another significant contribution is [BYC+23]. This approach also employs one or more LLMs as evaluators. These LLMs question the generating LLM based on its output, which is then used to generate a rating. The use of ensembling in this method helps produce a more unbiased evaluation, offering a multifaceted view of LLM performance. Each of these studies represents a step forward in automatic LLM evaluation, shifting from static to dynamic and interactive methodologies that better capture the complexities of LLM outputs.

Finally, [QWL+23] propose GameEval, an open-source evaluation framework that uses dialogue-based social deduction games to compare the reasoning capabilities between multiple LLMs that act as the players.

2.2 Solving Formal Games using LLMs

In this section of our study, I delve into several innovative approaches that enhance the reasoning and problem-solving capabilities of LLMs.

2.2.1 Prompt Engineering & Classic Algorithms

A first view into GPT-3's ability to solve planning problems without direct recall is discussed in [JWJ23], where the authors present a way to prompt GPT-3 into executing

algorithmic logic, akin to a Turing machine, without resorting to code recall or hallucination. Their Iterations by Regimenting Self-Attention (IRSA) technique is tested on classic computer science problems and logic puzzles, demonstrating GPT-3's capability to handle structured, algorithmic tasks.

[YYZ+23] advance this concept by proposing an algorithm that converts Chains of Thought (CoT) [WWS+23] into a decision tree. This tree can be navigated and analyzed using depth-first and breadth-first search strategies, allowing for more systematic and comprehensive exploration of problem-solving pathways.

Further developments are discussed in [CMS+23], where the authors analyze the logical reasoning abilities of various LLMs. They propose enhanced prompting techniques, both generative and retrieval-based, to improve reasoning skills. Additionally, they introduce a specialized dataset, LLM-LR, for pre-training a LLaMA2-13B-PT model, focusing on boosting its logical reasoning performance.

[HGM+23] presents an approach akin to the Tree of Thoughts but employs Monte-Carlo Tree Search along with a World Model. This method, named Reasoning via Planning (RAP), allows the LLM to evaluate the utility of different paths in the decision tree, integrating planning techniques into language model reasoning.

Lastly, several authors [LCSH23] and [AGS+23] have introduced methods that achieve satisfactory performance on multiple games (Resistance-Avalon, social deduction game called "Base", respectively) using different LLMs (GPT-3.5 and LLama-2, GPT-4 and GPT-3.5, respectively) as the players.

2.2.2 Combining Logic Provers and LLM Prompting

In [OGL+23], the authors present an innovative method where an LLM is employed to convert natural language logic puzzles into first-order logic expressions. This hybrid approach leverages an automated theorem prover to scrutinize these expressions for errors and, where feasible, deduce solutions, marking a significant stride in integrating LLMs with formal logic systems for problem-solving.

2.2.3 Fine-Tuning LLMs for Logical Reasoning

In [LKs+23], the authors provide a comprehensive survey of logical reasoning in the context of Large Language Models (LLMs), culminating in the introduction of a novel benchmark, LogiGLUE, for which they fine-tune a FLAN-T5 LLM.

Furthermore, [BGP+23] introduces three new datasets based on ReClor [YJDF20] and LogiQA [LCL+20] for evaluating LLMs on out-of-distribution logical reasoning. This study also presents a large-scale training set with varied tasks to enhance model generalization and robustness, specifically applied to fine-tuning LLama models of different sizes.

2.2.4 Multi-Agent Prompt Engineering and Algorithms

In exploring the domain of formal games within Large Language Models (LLMs), recent studies have seen success in using LLMs in different roles to collaborate on solving a task. A notable example is the work of [WMW+23a] who introduced "Solo Performance Prompting" (SPP), a technique where a single LLM adopts multiple personas, each with distinct skill sets, to engage in self-collaboration. This approach transforms the LLM into a cognitive synergist, showing promising results in tasks like Creative Writing, Codenames Collaborative, and Logic Grid Puzzles.

Complementing this, [WMW+23b] proposed an architecture inspired by the prefrontal cortex of the brain, utilizing multiple GPT-4-based modules. This architecture, evaluated using CogEval and the Tower of Hanoi problem, mimics the human prefrontal cortex's functionality, focusing on planning and problem-solving in LLMs.

Further advancing the field, the "AgentVerse" framework was introduced by [CSZ+23], leveraging a multi-agent system where each LLM plays a different role. This framework demonstrated enhanced performance in various tasks, including coding and Logic Grid Puzzles, underscoring the potential of multi-agent collaboration in LLMs. These developments highlight a growing trend in leveraging diverse architectures and collaborative frameworks to enhance LLMs' capabilities in solving formal games and complex tasks.

2.3 Generating Formal Games in Context of LLMs

In this section, I will focus on existing datasets and generation approaches for formal games in the context of LLMs.

2.3.1 Human Annotated Datasets

The exploration of formal games in Large Language Models (LLMs) has been an area of extensive research, with numerous studies contributing to our understanding of their capabilities. [BCL+23] study evaluated ChatGPT using 23 datasets covering a range of tasks in NLP, also introducing a novel multimodal dataset. Furthermore, [LNT+23] introduced LogiEval, an out-of-distribution (OOD) logical reasoning dataset, to test the logical reasoning abilities of ChatGPT and GPT-4 against established benchmarks. The "True Detective" benchmark, proposed by [DF23], presented a unique challenge in abductive reasoning, particularly for GPT-3.5 and GPT-4, using puzzles based on detective/mystery games.

Further advancements include the introduction of the BIG-Bench by [S+23], a large-scale, human-annotated benchmark covering a wide range of topics and languages, designed to challenge current and future LLMs. Furthermore, [lTN+23] propose the GLoRE benchmark, encompassing 12 datasets for various logical reasoning tasks, which they use to test multiple LLMs, including LLaMA, Falcon, ChatGPT, and GPT-4. Additionally, [SPP+23] introduce an OOD dataset focusing on general deductive reasoning,

allowing for a more controlled evaluation of deduction rules and complexity in reasoning tasks across several LLMs. [PPR23] conduct a comparative study that assesses the mathematical and logical problem-solving abilities of GPT-3.5, GPT-4, and Google Bard using a set of 30 questions, both known and original. These studies collectively underscore the growing interest in understanding and enhancing the reasoning capabilities of LLMs in the context of formal games and complex problem-solving scenarios.

2.3.2 Generation using LLMs

[WTY+23] introduce ByteSized32, a code generation pipeline that can test the planning abilities of LLMs by generating Python code that implements text-based, common-sense planning tasks. The generation pipeline is based on 32 hand-crafted game templates that a generator LLM uses as a base for its implementation. Automatic evaluation for the code is provided in the form of task adherence, error rate, and win rate of a GPT-4 agent. On a similar note, [PCOT23] introduces ACES, a framework that can automatically generate code puzzles in Python. Multiple other papers using LLMs for task generation are elaborated on in Appendix A and listed in Table 2.1.

2.3.3 Generation using Formal Approaches

[SL23] propose a novel dynamic dataset, generated using formal methods, for dynamic epistemic modal logic using the "Muddy Children" and "Drinking Logicians" problems. They evaluate the LLMs Pythia, GPT-3 and GPT-4 on the novel tasks.

2.4 Analysis of LLMs in the Context of Formal Games

Various studies have delved into analyzing the capabilities and limitations of LLMs when it comes to solving formal games and logic problems. An extensive study by [BCE+23] analyzes the capabilities of GPT-4 on several challenging tasks covering a broad range of domains, including extremely challenging areas such as logic and math problems. [FPC+23] build on this and propose three novel datasets, GHOSTS (and, building on GHOSTS, the miniGHOSTS, and microGHOSTS datasets), which aim to test ChatGPT and GPT-4 on a higher level of mathematics. The authors also provide a novel framework for analyzing failure modalities concerning the proposed solutions generated by the LLM, which they use to provide deep insights into the shortcomings of GPT-4 and ChatGPT as assistants to mathematicians.

Another pivotal study conducted by [WTB+22] analyzes the impact of scaling LLMs, highlighting 'emergent abilities' that are only noticeable at larger scales. This research covers a broad spectrum of tasks, including formal games, and emphasizes the unique capabilities that manifest at this scale. [Gro23] focuses specifically on ChatGPT, where the authors manually analyze its performance on 144 puzzles from a textbook, categorizing logical faults into 67 distinct types. This meticulous study provided a detailed insight into the reasoning capabilities and limitations of ChatGPT. Complementing these, [GBWD23]

critically assesses the abstract reasoning abilities of multiple leading LLMs across various benchmarks. Finally, [XLV+24] test the Abstract Reasoning Corpus (ARC), proposed by [Cho19], on multiple LLMs. To enable the LLMs to play the visual-based challenges, they create a text-based encoding for it (and an object-based one using a graph-based approach) and introduce a 1D-ARC dataset that they use to test if using textual representations of ARC makes GPTs perform worse.

This research contributed to a growing understanding that, despite their advancements, LLMs still face significant challenges in areas like abstract reasoning. Collectively, these studies provide a nuanced view of the strengths and weaknesses of LLMs, particularly in tasks requiring complex reasoning and problem-solving skills.

2.5 Solving Formal Games Using Formal Methods

Another approach to solving formal games is utilizing and adapting classic methods such as constraint-satisfaction-problem solvers and others. The following will give a brief overview of the methods being applied in the current research literature. [BGG21] introduce a framework for solving constraint satisfaction problems step-wise, using an iterative method to construct explanation sequences, evaluated on Logic Grid Puzzles. Progressing further, [WLZ+21] developed a hybrid system combining a machine learning model with task-specific reasoning modules and symbolic knowledge integration, tested on the reasoning parts of the LSAT.

[GN21] introduced a system for solving logical puzzles in natural language, emphasizing the explainability of solutions with graphical proofs. This system was evaluated on various puzzle types, including Knight and Knaves puzzles. Additionally, a comprehensive collection of logical games modeled in First-Order Logic was provided by one of the authors in [Gro21], spanning a broad range of logical reasoning domains.

Furthermore, [KKSR22] released a dataset of New York Times crossword puzzles, evaluated on sequence-to-sequence models, retrieval-based methods, and CSP solvers, thereby introducing a new NLP benchmark in the form of crossword puzzles. This was complemented by the Berkeley Crossword Solver, introduced by [WTX+22], which combined neural question-answering models with belief propagation and local search techniques, and tested on New York Times puzzles.

Moving into a more general direction than crossword puzzles, [EGH+23] propose DEMISTIFY, a tool using Minimal Unsatisfiable Subsets for creating interpretable explanations for solving pen and paper puzzles, showcasing the tool's capability in logical deduction. Finally, [SBT23] introduce a novel system to accelerate search-based solvers for Witness-type Triangle Puzzles using an automatically learned, human-explainable predicate, tested on puzzles from the game "The Witness". These developments collectively illustrate the diverse and evolving approaches in leveraging various non-LLM-based approaches for solving and understanding a wide range of formal games and puzzles.

2.5.1 Solving Formal Games with Planning

Another line of research relevant to our work is the domain of "Planning", which concerns itself with algorithms and methods designed to plan interaction with environments using tools such as the PDDL programming language for domains such as Blocksworld. Most of the research in planning studies robotics or other agent-driven environments, there are however several studies that go more into the direction of reasoning. Hence, I give an overview of relevant works in the following sections.

Combining with External Solvers

The integration of Large Language Models (LLMs) with planning and logic solvers represents a burgeoning area of research, especially in the context of formal games and problem-solving via planning. In [XYZ+23], researchers propose using PDDL (Planning Domain Definition Language) to translate natural language goals into planning goals, evaluated using Blocksworld and Alfred-J, domains pertinent to robotics. This approach was further advanced by [LJZ+23], where an architecture was developed to convert natural language planning into PDDL code using an LLM, process it through a planning solver, and then retranslate the results back into natural language. This method was evaluated using the Blocksworld domain.

Additionally, [JJL+23] employed LLMs to iteratively generate Blocksworld PDDL code, using an SMT (Satisfiability Modulo Theories) reasoner as a verifier to iteratively improve plans. Furthermore, [YCDD23] introduced SatLM, which involves an LLM constructing SAT (Satisfiability) problems in First-Order Logic, input into a SAT-Solver, and tested on various reasoning benchmarks, such as LSAT and BOARDGAMEQA.

Parallel to these developments, [YIL23] saw the coupling of LLMs with logic programming, specifically using LLMs to extract declarative knowledge representations (answer set programs) from natural language task descriptions. This was followed by utilizing an ASP (Answer Set Programming) solver module to generate valid solutions. Similarly, a study by [IYL23] proposed a method for LLMs to convert logic puzzle tasks from natural language into ASP, enabling automatic solution generation by an ASP solver. These innovations highlight the potential of LLMs in enhancing robust and general reasoning capabilities, particularly in formal games and complex problem-solving scenarios, by synergizing with various logic and planning solvers.

Prompt Engineering & Algorithms

In [SDS+23], researchers introduce a 4-stage LLM system using GPT-4 for solving planning problems across various domains. This system involves generating a domain summary, strategizing in Python code, code validation, and an automated debugging process that iteratively corrects errors up to four times. Following this, [GSG23] developed a prompt compiler for LLMs that systematically generates strategic reasoning prompts, particularly for matrix and strategy games. This compiler leveraged tree-based search algorithms to construct state values and beliefs for the prompts.

Furthermore, [NCY+23] propose a novel approach for prompting LLMs cost-effectively, enhancing the diversity in the generated results. This improvement in diversity was shown to enhance performance on several reasoning and planning benchmarks. [TWL+23] introduce the Inferential Exclusion Prompting (IEP) framework. IEP enhances Chain of Thought (CoT) reasoning by guiding LLMs through a planning process that includes an elimination step, specifically designed for puzzles and other reasoning challenges. Finally, [GYY+23] propose Suspicion-Agent, a prompting framework for LLMs that allows them to utilize multiple planning steps to play two-player perfect information games with satisfactory performance.

Combination with Ideas from Reinforcement Learning

Some of the advancements have focused on enhancing the planning and problem-solving capabilities of Large Language Models (LLMs) through the integration of Reinforcement Learning (RL) and heuristic methods. [HACY20] propose a new framework for Interactive Fiction (IF) Games, Jericho, which aims to make text-based adventure games playable by Reinforcement Learning agents by extracting lower dimensional action spaces and building a Game-World Tree from NL. Their environment aims to be used for training RL agents for planning and commonsense reasoning tasks packaged inside of the IF games. In [LHZ+23], the authors introduce RAFA, a system designed to improve LLMs' reasoning and long-term planning abilities using RL actor-critic techniques. RAFA formalizes the interaction with an LLM as a Markov Decision Process (MDP), representing a significant step in integrating RL principles with LLMs.

Furthermore, [HMR23] introduce the SayCanPay, a planning system consisting of three stages. This system utilizes a fine-tuned LLM (such as Vicuna or Flan-T5) to select actions, which are then assessed by an affordance model based on their feasibility. A heuristic estimator subsequently scores each feasible action based on its expected reward, aligning with classic RL concepts.

Finally, [ZYSR+23] develop an extension to the RAP and ReAct frameworks. This extension incorporates external feedback from the environment, including state and rewards, to guide a search process akin to Monte Carlo Tree Search (MCTS). This approach unifies reasoning, planning, and acting in LLMs, demonstrating a sophisticated application of search algorithms and RL in enhancing LLM capabilities. Together, these studies showcase innovative methods for integrating RL and heuristic planning with LLMs, advancing their application in complex decision-making and problem-solving scenarios.

Finetuning LLMs

Another approach in current literature to solve planning problems is to fine-tune LLMs using various planning-based datasets. [YGW23] fine-tune a small LLaMA 7B model specifically for outcome verification in mathematical reasoning problems like GSM8K and the "Game of 24". This approach involves converting math problems into planning

problems by focusing on verifying intermediate steps and showcasing a novel method to enhance LLMs' capabilities in mathematical reasoning.

Another study by [SFLG23], evaluates various prompting techniques to improve the reasoning and planning abilities of LLMs like GPT-4 and ChatGPT. The authors fine-tune a model named "ConDec" via Contrastive Decoding, augmented by hard negatives constructed by an external reasoner. This study aims to improve LLM performance on the EntailmentBank benchmark, indicating a focus on enhancing logical reasoning through tailored fine-tuning approaches.

Furthermore, [WCO+23] introduce the concept of "Planning Tokens", a method to prefix each step of a reasoning chain to improve LLM performance. The study involved fine-tuning three base models (Phi-1.5, LLaMA 2 7B & 13B) using a dataset containing these "Planning Tokens" and compared the results with other approaches like Chain of Thought (CoT). This technique signifies an innovative approach to guiding LLM reasoning through structured prompt engineering and fine-tuning. Collectively, these studies highlight the evolving strategies in enhancing LLMs for planning and problem-solving in formal games and mathematical contexts, blending fine-tuning, prompt engineering, and outcome verification techniques.

Finally, [LSS+24] introduce a search-augmented Transformer architecture and training regime. Their proposed approach uses the optimal search paths and execution traces from an A* search algorithm to train an initial model. Then, they use synthetic data generated by the initial model to bootstrap a final model that produces shorter, still optimal search paths. The authors also conduct a number of experiments comparing their models on two search tasks in different configurations.

2.6 Comparison to Similar Approaches

In this section, I discuss what sets our work apart from several other methods proposed by other authors. I provide an overview of each key work and elaborate on the differences of what makes our approach unique. An overview of the contributions made by each work can be seen in Table 2.1. For selected papers, I provide a comparison in full text in Appendix A.

As I introduce our own domain-specific language (DSL) for combinatorial grid games, I felt it necessary to provide context as to why I would not use an existing DSL, such as the CGSuite developed by Siegel [Sie]. CGSuite is a desktop application that contains a custom domain-specific scripting language for combinatorial games. The system is based on the mathematical foundation of game theory, which provides users with the ability to analyze games in a game theoretic sense. This is very useful if one is interested in the mathematical properties and peculiarities of a specific formal game. Editing script, creating games and states, as well as all other functions, have to be done via a "Worksheet" in the desktop application.

Publication	Tests OOD reasoning capabilities	Compares to humans	Dataset Generation	Ranks game complexity	Automatic Evaluation	Domain
ACES [PCOT23]			LLM		✓	P3 Programming Puzzles
Parse-and-Solve [CWF+22]	✓	✓	Human-Annotated		✓	Human Annotated Household Tasks
PlanBench [VOSK23]		✓	LLM		✓	Blocksworld
GLAM [CRW+23]	✓		Simulation		✓	BabyAI
Spinning Tops [CGT+20]			Simulation	✓		OpenSpiel
ByteSized32 [WTY+23]			LLM + Hand-written templates		✓	Common-sense Household Tasks
SearchFormers [LSS+24]			A* traces & Bootstrapping			Maze & Sokoban
Our approach	✓		LLM + Simulation	✓	✓	Various formal games

Table 2.1: Comparison between different approaches

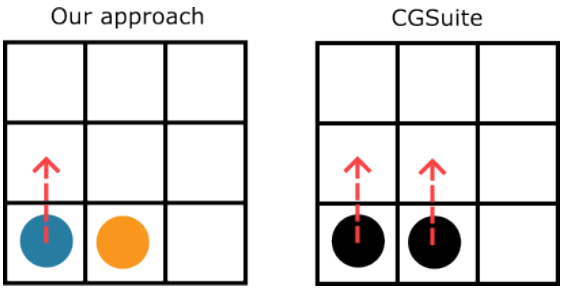

Figure 2.1: Comparison between possible moves of *Left*

A key difference between our approach and CGSuite is the way game pieces are represented. Our pieces are tuples that contain the piece's coordinates, their type, and the player they belong to $(row, col, type, player)$. In CGSuite's DSL, the game pieces are represented solely by their coordinates and their type. This detail might seem minute but has a great impact on the games that can be represented. This is because in every grid game, both in CGSuite and our work, a player's actions depend on the pieces they own and their type. With CGSuite's DSL, there currently is no way to differentiate ownership of a piece apart from their type (e.g., white versus black game pieces in Go). With our DSL it is naturally possible for both players to own pieces of every type. This also allows us to define restrictions (or, as I will call them later, conditions) on rules available for a specific type based on the player who owns the game piece. I can also represent win and loss conditions based on every piece type for both or just one of the players. Both of these features are not possible in CGSuite's DSL. A very simple game that CGSuite can not represent, but by our DSL can be, is shown below in Example 2.6.1. Concluding, I chose to provide our own, simple but robust DSL that precisely encompasses the domain of games I focus on. The language I introduce in Section 3.2 is small, concise, easy to implement, and integrates well with the rest of our automatic evaluation framework.

Example 2.6.1. I provide a simple example of a game where the representation power of our DSL differs to the CGSuite one. As I will use the notation of our DSL, I recommend readers to read through Section 3.2.1 first. Assume that the game has one type of piece available, $\mathcal{T} = \{1\}$, and one rule $r_1 = (\text{up})$, which moves a game piece up one cell in the grid. Furthermore, I now define that pieces of type 1 can move using r_1. Up until now, everything I have done can be represented by both DSLs.

Now, I introduce a grid of size 3×3 with two pieces of type 1 on them at positions $(3, 1)$ and $(3, 2)$. Assume now that I want to let each player have one of these pieces. In this example, assume that *Left* owns the piece at $(3, 1)$. If *Left* now wants to move, with our approach, their options are restricted to only moving the piece at $(3, 1)$ with r_1, even though both pieces have the same type and rules available. This would not be possible to represent in CGSuite's DSL without adding additional types, which would result in a different game. With CGSuite, *Left* could move both pieces using r_1, as there is no differentiation between players. A visual comparison of this is shown in Figure 2.1.

CHAPTER 3

Methodology

In the following, I describe how to design a generative dataset of formal games where various aspects, such as the game complexity, structure of the game, and the formatting of the prompts can be controlled by the user.

3.1 Definition of a Formal Game for LLM Evaluation

3.1.1 The Landscape of Formal Games

Games can be studied on various levels of formality and abstraction, from very general [JS85] to concrete [Sie13].

In thd following will give a general definition of the concept of a "game" that is sufficiently general in order to be used to evaluate the performance of an LLM – but not too general; e.g., some games are typically formalized using infinite action spaces, which I will exclude. I elaborate on our reasons for exclusion in Section 3.1.2.

Our reason for introducing a formal definition is that it lends itself to a specification language for games.

I argue that games are a sufficiently well-suited class to use as a test bed to assess LLMs' reasoning capabilities, since, e.g., what is typically referred to as a "board game", falls under this class, and already can solicit highly complex cognitive feat, that human effort to be mastered and that is correlated to general reasoning performance [Cho19].

3.1.2 Excluded Game Frameworks

Our definition of a game explicitly constrained everything to finite objects. This choice was taken to ensure that everything will be computable within our framework introduced in Section 3.2.

Games with infinite action spaces (such as video games, where one can choose, for example for a continuum of values, a specific one (e.g., the acceleration in a certain direction), are also often discretized when implemented. A recent example of this is AlphaStar, the large-scale RL agent for the video game Starcraft 2 developed by [MOS+23].

I note that this discretization can dramatically change important game-theoretic concepts, such as Nash equilibria.

To further illustrate the effects of discretization, I will provide examples of two simple games in Section 3.1.2 and Section 3.1.2. In both examples, I will show that leaving the continuous space will alter the strategies and outcomes. For the reasons presented there, I explicitly exclude discretized versions of games with continuous elements from our framework, to avoid game-theoretic issues that are not relevant for LLM evaluation.

Penalty Kick

For the following, I will describe the game of "Penalty Kick", as described by [Spa]. The game is a simple two-player game, where the kicker tries to score a goal against the goalkeeper. To play the game, the kicker chooses a direction to aim at, while the goalkeeper chooses a direction to jump in to try and save.

As [Spa] describe, in a real-world continuous setting, the kicker might favor one side more and could choose different angles to aim at. Similarly, the goalkeeper can choose different angles and jump power to dive towards the ball. This leads to different ways that a real-world penalty kick game can be formalized, to be analyzed by game theory. To subsequently discretize the game, one could remove all of these choices and simplify them to the options left, right, and center for both players.

This reduction of the action space has several effects on the outcome:

- The predictability of both players is increased due to the lack of nuance. For the kicker, using his assumed stronger right-side kick may not be effective anymore, as the goalkeeper does not need to develop effective strategies against slight differences in angle or strength.
- This causes the strategy space to become smaller and more influenced by chance than skill: Each player only has three choices against each of his opponent's three moves. Previously, the kicker could have used his strong right kick with various angles, whereas now he may be forced to choose equally from all three options.

Simple Market Entry Game

Another classic example from Economics is the "Market Entry Game". The game simulates firms that decide to produce a quantity of goods, which they base on market demand and their competition. Each firm can choose a continuous number of goods to produce. This allows them to fine-tune their production to meet market demands and optimize profits under the supply being produced by the other companies.

In the discretized version, each company is limited to a fixed number of quantities they can produce.

This has various effects on the outcome of the game:

- The most important effect is the loss of a precise equilibrium between the producing firms. Each of them can now either have excess or too little product stock available. This will in turn remove their ability to maximize their profits against each other to achieve an ideal equilibrium.
- Due to the previous effects the equilibrium will likely also not be as satisfying for the firms as in the continuous version, since the discrete options may not allow for precise optimization.
- Each firm will have less flexibility to respond to changes in the market's supply and demand, causing the market to become less competitive.
- Predictability of each firm's actions will increase, as their strategy space is limited. This potentially allows other companies to easier respond to changes made by others.

Generalizations

I now review various further generalizations and explicitly argue why they are not appropriate for our framework.

- **Differential Games:** These are games where the "time" is continuous, and players can make a decision at any point in time. Formalizing such games adds considerable mathematical complexity but does not yield new insights for the reasoning performance of an LLM, as LLMs do not process information continuously
- **Cooperative Games:** These are games where strategies on the level of formed coalitions are considered. These types of games are also unsuitable for analyzing LLMs performance since the performance would depend on multiple interactions between multiple agents. This is an unnecessary complication, since I are only interested in testing the reasoning capabilities of one LLM at a time, not their ability to cooperate. Furthermore, the performance of an LLM would also heavily be influenced by the actions taken by other agents playing the game.

3.1.3 Derived Game-Theoretic Concepts

In game theory, players playing the game have to make decisions based on the information available to them. This choice between *alternatives* is also referred to as *move*. There are two kinds of moves available to players: *personal moves* (choice made by a player) or a *chance move* (the choice depends on some stochastic device). A plan encoding every possible situation into moves to take for a player is called his *strategy*. Players can use

multiple strategies, which are called his *strategy set*. Furthermore, players have a *pay-off function*, which tells them the real-valued outcome of their strategy. The payoff function is thus a map $\mathcal{K}\colon \Sigma \mapsto \mathcal{R}$, where a *strategy* σ_i is a subset of the set of all strategies Σ_i. The execution of a game from start to finish is called a *play*.

This method of defining a game is referred to as the *normal form* or also *strategic form*. A game can be called *zero-sum* if the sum of all payoffs K_i equals to zero:

$$\sum_{i=1}^{n} K_i(\sigma_1, \ldots, \sigma_n) = 0,$$

where i is the index of each player. In contrast to zero-sum, there are also *variable-sum* games, where each K_i can have different payoffs depending on the player's strategy [Fav18].

Another important concept is that of *mixed* and *pure strategies*. A *mixed strategy* is defined as a probability vector x_i^k for the ith player. It means that he will play strategy σ_i^k with probability x_i^k. The original strategies $\sigma_i^{(k)}$ are called *pure strategies*.

When players take their moves at the same time, I speak of a *simultaneous game*. On the other hand, games with alternating moves between players are called *sequential games*. Some games may not fit into either.

3.1.4 Remark on Markov Decision Processes

There is an obvious relationship between games and Markov decision processes, which I describe in this Section.

The main differences boil down to the fact that these two mathematical objects formalize different aspects that occur when playing a game: The definition of a game takes a broad perspective, and the concepts that game theory introduces are tailored to understanding "global" questions, such as the theoretically best possible strategy for any player given player in a given game, Markov decision processes (MDP) take a "local" perspective. An MDP aims to model environments that require stochastic decision-making by some agent. The goal is then to find policies (i.e., probability distributions over the actions space for a given state) that maximize the expected cumulative rewards. This policy may not (and usually will not) be theoretically optimal.
What distinguishes an MDP from classical Game Theory is, that it is oriented towards solving the "inverse problem". In game theory, one starts with a set game, where the rules, possible strategies, and the pay-off functions are all known. The goal is then to find the optimal strategy for rational players. MDPs reverse this and model a probabilistic environment of the game without explicit knowledge about the rules and the strategy space. This environment is then used to find a policy that maximizes the reward for each player. This can be argued to be the same as "learning the rules" of the game and then exploiting them.

3.1.5 More Specific Definitions of a Game and Game Representations

While the definition given above is intuitively appealing, as it allowed us to precisely describe how it can be generalized in different ways and how various parts of the definition needed to be modified for that purpose (as well as why these modifications are not sensible from the standpoint of LLM evaluation) in practice, other representations may be used to fit a certain use case. The extensive form I elaborate on in Section 3.1.5 allows for the direct application of graph-based search algorithms, like Monte Carlo tree search (MCTS). I employ MCTS as the foundation for our action ranking in Section 3.4 and the complexity metric in Section 3.3.

Matrix Form

Starting simple, games in normal form can be easily represented as a matrix, containing all combinations of move alternatives. An example of the prisoner's dilemma is shown in Table 3.1 (I use the example provided by [Fav18]).

	Confess	Lie
Confess	$-8, -8$	$0, -10$
Lie	$-10, 0$	$-1, -1$

Table 3.1: Prisoner's Dilemma in normal form

Extensive Form

One of the most common ways to represent games is to utilize the more general graph-based extensive form proposed by [KT53]. Kuhn bases his graph-theoretic definition on the set-theoretic approach that [JvN04] takes but generalizes and simplifies it. Thus, I ground our exposition in [KT53].

As the first step, some general definitions for the necessary components will be given. I define a *game tree* K as a finite tree with a vertex 0 (i.e., the root node). *Alternatives* at a vertex $X \in K$ are the edges e at X lying in components of K (not containing 0), if I cut K at X. These j alternatives at X are indexed by positive integers $1, \ldots, j$. Those vertices that have alternatives are called *moves* (as in the normal form). The remaining vertices are referred to as *plays* (i.e., the leaf nodes of the tree). A *play* of a game is defined as a unique path from vertex 0 to a play. A set $A_j, j = 1, 2, \ldots$ contains all of the moves with j alternatives and is called an *alternative-partition*.

Having done these definitions, I can go on to define a game using them. A general n-person game can be defined as a game tree K if it follows the given specifications:

1. Moves are partitioned into $n+1$ sets $P_0, P_1, \ldots, P_n$. These are referred to as *player partitions*. Sets $P_1, \ldots, P_n$ are the *personal moves* of players $i = 1, \ldots, n$. P_0 contains the *chances moves*.

2. Moves can be further partitioned into sets U as a refinement of the player and alternative partitions. U is defined to not contain two moves on the same play and $U \in P_i \cap A_j$. These sets U are called *information sets*.

3. Each $U \subset P_0 \cap A_j$ has a defined probability distribution on the integers $1, \ldots, j$ with a positive probability for each element (e.g., taking a chance move assigns a probability to each alternative).

4. The *pay-off function* is defined as an n-tuple of real numbers

$$h(W) \coloneqq (h_1(W), etc., h_n(W))$$

for each play W and each player.

A play is then constructed by beginning the vertex 0 and progressing by choosing moves. Suppose the game has progressed to a move X. Players whose information sets contain X then choose one of the j alternatives if the move is a personal one. In the case of a chance move, an alternative is chosen by the probabilities for the information set containing X. Thus, a finite play W with a start at vertex 0 is constructed. Finally, the players are paid the amount $h_i(W), i = 1, \ldots, n$.

Now, I can define a game with *perfect information* as one where all information sets have exactly one element.

Let us further define a *pure strategy* using this notation. A pure strategy for a player i is a function π_i mapping the set $\mathcal{U}_i = \{U | U \cap P_i\}$ of personal moves into positive integers so that $U \cap A_j$ implies $\pi_i(U) \leqslant j$. This means that at each vertex $X \in U$, π_i chooses one of the j alternatives (i.e., edges e). Simply spoken, a pure strategy can be considered a pre-defined plan for a player i at each decision point in the game. The pure strategies $\pi = (\pi_1, \ldots, \pi_n)$ define a probabilistic distribution on the alternatives. Given an edge e, that is a personal move in the information set U, I can define the probability as follows:

$$p_\pi(e) \coloneqq \begin{cases} 1 & \text{if } \pi_i(U) = v(e), \\ 0 & \text{otherwise.} \end{cases}$$

For a chance move, this is just the probability assigned to it. Thus, the probability distribution of the plays of K is given by: $p_\pi = \prod_{e \in W} p_\pi(e)$ for all W. Using this, I can now define the *expected pay-off* as $H_i\pi$ for player i using the pure strategies $pi_1, \ldots, \pi_n$ by $H_i(W) \coloneqq \sum_W p_\pi(W) h_i(W), i = 1, \ldots, n$

Similarly, I can define a *mixed strategy* as μ_i for player i. A mixed strategy can be understood as a probability distribution over pure strategies: For any n-tuples of mixed strategies $\mu = (\mu_1, \ldots, \mu_2)$ for n players, the probability distribution is defined by:

$$p_\mu(W) \coloneqq \sum_\pi q_{\pi_1} \ldots q_{\pi_n} p_\pi(W) \text{ for all } W.$$

The expected pay-off is defined in the same way as for pure strategies, using p_μ instead.

An example of the extended form of the game Tic-Tac-Toe can be seen in Figure 3.1.

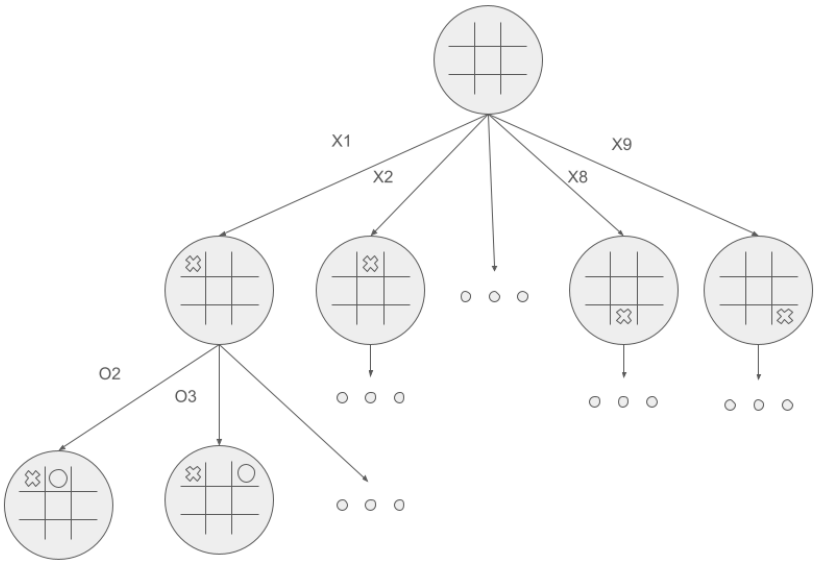

Figure 3.1: Extensive Form of a game of Tic-Tac-Toe

Using a state and action-based formalization (like in an MDP or the extensive form) will be crucial in many parts of our work. It allows us to easily track the history of a game, which I can then use to measure, among others, the complexity of a specific game.

The notion of actions also flows nicely into our generative formal framework for games, which I will introduce in Section 3.2.

3.2 Generating Games

A game can be abstracted into its core mechanics: rules for taking action, win and loss conditions. I provide a formal framework, parameterized by probability distributions, that allows for generating these components. I focus solely on 2D-grid, combinatorial games, as the rules and win conditions for this class of games can easily generated and then represented in natural language. I introduce the framework formally in Section 3.2.1.

Because all games are new, and an infinite amount of them can be generated, this solves an important problem with data on LLMs:

- LLMs are trained on data that, therefore, should not used as testing data;
- If new test datasets are offered publicly, it is not possible to exclude them from potentially being used to pretrain LLMs.

Our approach solves both of these problems.

I note that even if there is an infinite amount of games present, it can be the case that they are distributed around a distribution with a spike so fine-tuning on a sufficiently large dataset of generated games will increase test scores – a scenario I want to avoid since I want to design a testing benchmark that can be successfully used by a user who does not have access to information regarding the model internals, as is frequent in the languge-model-as-a-service (LMaaS) scenario [LMPF+23]).

To guard against this, I design our approach in a way that has a long tail: The games I design are diverse and I introduce various techniques to create diverse natural-language explanations of these games.

The entire set of possible games is partitioned by their complexity, which is derived from these data. Our generative dataset is designed in such a way that for a fixed level of complexity, an infinite amount of games can be theoretically generated. The user can set the complexity to a desired value.

3.2.1 2D-grid Game Formalization

Our generation framework is inspired by the top three approaches developed to solve the ARC Challenge introduced by [Cho19]. The three frameworks developed by [Ice], [dMAC], and [LG] respectively, all have one thing in common: they rely on a custom DSL and program synthesis to exploit the intrinsic rules of the puzzles included in the ARC dataset. This is the reverse of what I are trying to do: they receive a game as input and try to find rules for them. The approach I will introduce in this section does the opposite. I define a DSL that describes all rules and conditions and sample new games by combining them.

For generating games, I focus solely on combinatorial games that are played on a two-dimensional grid. A combinatorial game has no elements of chance and exactly two players [Sie13].

I will focus on normal play since this is the most well-explored class of combinatorial games, as elaborated on in Section 4 of [Sie13]. In normal play, the last player to move wins. Our game-ending conditions (win or loss) are created by a player moving a game piece, thus either satisfying a win condition for himself or a loss condition for the opponent. In miseré play, this is reversed, and the last player to move would lose. This class of games is not as intuitive to play and understand, as most games that humans enjoy are normal play games.

The Grid

In the following, I will refer to the players by the names *Left* and *Right*. For some functions, I will use the digits 0 to refer to *Left* and 1 to refer to *Right* to make the notation more concise. Our games are played on $n \times k$ grids, which can be represented

as an $n \times k$ matrix where $1 \leqslant n, n \in \mathbb{N}$ and $1 \leqslant k, k \in \mathbb{N}$. I will refer to the grid as the board sometimes. Each position in the grid can only contain at most one game piece, which I will define next.

Pieces

At the core of playing a game on a grid are the game pieces that can be moved by the players. I start by giving an abstract notion of what a game piece is. A game piece is defined as a tuple of its coordinates, type, and the player to whom the piece belongs:

$$p \coloneqq (row, col, type, player),\ 1 \leqslant row \leqslant n, 1 \leqslant col \leqslant k, player \in \{0, 1\}, 1 \leqslant type,$$

where $row, col, type$ are all natural numbers. I refer to this set of all possible p's as $\mathcal{G}$. To refer to each element of the tuple $p \in \mathcal{G}$, I use the following indexing notation:

$$\begin{aligned} p_{\text{row}} &\coloneqq p_1, \\ p_{\text{col}} &\coloneqq p_2, \\ p_{\text{type}} &\coloneqq p_3, \\ p_{\text{player}} &\coloneqq p_4. \end{aligned}$$

Each coordinate pair (row, col) on the grid contains at most one game piece. I denote the set of all game pieces p placed on the grid as $\mathcal{P} \subseteq \mathcal{G}$. Additionally, I denote the set of all types used for game pieces in $\mathcal{P}$ as $\mathcal{T}$. Depending on the game, there might be varying numbers of piece types used and they might have different rules that apply to them (as I will see in further below). Finally, I create two subsets of $\mathcal{P}$ containing each player's game pieces. I denote these subsets of $\mathcal{P}$ as $\mathcal{P}_{\text{Left}}$ and $\mathcal{P}_{\text{Right}}$ respectively.

Example 3.2.1. Let us assume a $n \times k$ grid with $n = k = 3$. I create the set $\mathcal{P}$ with two game-pieces in it, $p_1 = (1, 2, 1, 0)$ and $p_2 = (1, 1, 2, 1)$. The grid with two game pieces on it is displayed in Figure 3.2. For the following examples, I will use blue and orange as colors for *Left* and *Right* respectively. For the game-piece types, I will use a circle for type 1 and a square for type 2.

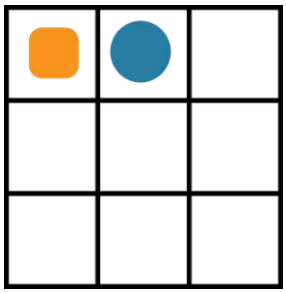

Figure 3.2: Example of a simple board position

Rules

The game's rules define which actions players can take to alter the state of their game pieces. Each rule is made up of a configuration of atomistic, basic rules, which are formalized as the following sets of functions:

$$\begin{gathered}\mathcal{M} \coloneqq \{\text{up}, \text{down}, \text{left}, \text{right}, \text{se}, \text{sw}, \text{ne}, \text{nw}\},\\ \mathcal{M}_{\text{capture}} \coloneqq \{\text{up}_{\text{C}}, \text{down}_{\text{C}}, \ldots, \text{nw}_{\text{C}}\},\\ \mathcal{S} \coloneqq \{\text{switch}, \text{exchange}\},\\ \text{place}\,.\end{gathered}$$

I define the set of all basic rules as

$$\mathcal{R}_{\text{B}} = \mathcal{M} \cup \mathcal{M}_{\text{capture}} \cup \mathcal{S}.$$

Below is a specification[1] of all these functions and their parameters :

- up : $\mathcal{P} \to \mathcal{P}$: Moves a piece up one row by subtracting 1 from the pieces' *row* value.
- down : $\mathcal{P} \to \mathcal{P}$: Moves a piece down one row by adding 1 to the pieces' *row* value.
- left : $\mathcal{P} \to \mathcal{P}$: Moves a piece left one column by subtracting 1 from the pieces' *col* value.
- right : $\mathcal{P} \to \mathcal{P}$: Moves a piece right one column by adding 1 to the pieces' *col* value.
- nw : $\mathcal{P} \to \mathcal{P}$: Moves a piece north-west by subtracting 1 from the pieces' *col* and *row* values.
- ne : $\mathcal{P} \to \mathcal{P}$: Moves a piece north-east by adding 1 to the pieces' *col* value and subtracting 1 from *row* value.
- sw : $\mathcal{P} \to \mathcal{P}$: Moves a piece south-west by adding 1 to the pieces' *row* value and subtracting 1 from the *col* value.
- se : $\mathcal{P} \to \mathcal{P}$: Moves a piece south-east by adding 1 to the pieces' *col* and *row* values.
- switch : $\mathcal{P} \times \mathcal{P} \to \mathcal{P}$: Takes two pieces as input and switches their coordinate values *row* and *col*, then returns the re-positioned first piece.
- exchange : $\mathcal{P} \times \mathcal{T} \to \mathcal{P}$: Takes a piece and a type as input and exchanges the pieces' type by the parameter, then returns the adapted piece.
- place : $\mathbb{N} \times \mathbb{N} \times \mathcal{T} \to \mathcal{P}$: Takes coordinates and type as input, places a piece with the parameters as values on the grid and returns it.

[1] For convenience, I describe them in natural language.

All functions, except for place, take one or more pieces as input and modify their coordinates or types, then return the modified piece. The rules in $\mathcal{M}_{\text{capture}}$ define the same directions of moves as those in $\mathcal{M}$, except they capture an opponent's piece (i.e., remove from the grid) that is in their target destination.

A composite rule is then created by concatenating basic rules from $\mathcal{R}_{\text{B}}$ for $c \geqslant 1$ times:

$$\text{MR}(c) \coloneqq \bigtimes_{i=1}^{c} \mathcal{R}_{\text{B}} \cup \{\text{place}\}.$$

Example 3.2.2. I continue with the previous Example 3.2.1. Let $c = 3$. I define a combination of three basic rules from MC(3), for example:

$$r_1 = \text{down}(\text{down}(\text{right})).$$

The rule I just defined moves a game piece two up and one to the right. Applying r_1 to p_2 from previously (recall $p_2 = (1, 1, 2, 1)$), I obtain: $r_1(p_2) = (3, 2, 2, 1)$. The resulting, updated grid is shown in Figure 3.3.

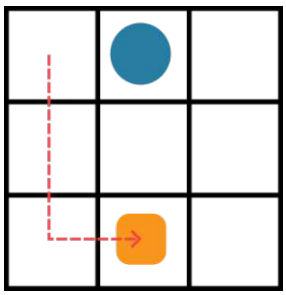

Figure 3.3: Board position after applying r_1 to p_2

Adding Conditions to Rules

In many games, certain rules only apply if a pre-condition is met for the pieces they would be enacted on. An example of this is "Castling" [Wik] in Chess, which has several pre-conditions to allow the King and Rook to move towards each other and then exchange positions. To capture these sorts of conditions, I create a helper function $\text{eq}\colon \mathcal{P} \to \{\text{True}, \text{False}\}$:

$$\text{eq}_{row,col,player}(row, col, type, player) \coloneqq \begin{cases} \text{True}, & (p_{\text{row}}, p_{\text{col}}, p_{\text{player}}) = (row, col, player), \\ \text{False}, & \text{otherwise.} \end{cases}$$

The eq function takes a game piece from $\mathcal{P}$ as input and compares its coordinates and player index against pre-defined values. The indices $row, col, player$ have the same ranges as the components of the game piece tuples introduced in Section 3.2.1. In addition,

to define conditions that cover ranges of values, I introduce a special value of -1 for the indices $row, col, player$. This special value will validate any comparison against it as True. To allow for combinations of multiple conditions, I introduce three functions that represent boolean logic operators:

$$\text{and}\colon \{\text{True}, \text{False}\} \times \{\text{True}, \text{False}\} \to \{\text{True}, \text{False}\},$$
$$\text{and}(x, y) \coloneqq \begin{cases} \text{True, if } x \text{ is True and } y \text{ is True,} \\ \text{False, otherwise.} \end{cases}$$

$$\text{or}\colon \{\text{True}, \text{False}\} \times \{\text{True}, \text{False}\} \to \{\text{True}, \text{False}\},$$
$$\text{or}(x, y) \coloneqq \begin{cases} \text{True, if } x \text{ is True or } y \text{ is True,} \\ \text{False, otherwise.} \end{cases}$$

$$\text{not}\colon \{\text{True}, \text{False}\} \to \{\text{True}, \text{False}\},$$
$$\text{not}(x) \coloneqq \begin{cases} \text{True, if } x \text{ is False,} \\ \text{False, otherwise.} \end{cases}$$

Conditions defined by eq can be combined with the following of the boolean operator functions: $\{\text{or}, \text{not}\}$. Using the and operator is not sensible, since I only want to check the condition of one piece at a time and it can't have multiple coordinates or players assigned. I define this as a function $\text{MC}(j)$, where j is a natural number:

$$\text{MC}(j) \coloneqq \bigtimes_{i=0}^{j-1} \{\text{or}, \text{not}\} \times \{\text{eq}\}.$$

If I think of the function composition as a tree, the leaves have to be functions eq. They perform the equality checks that make up the functionality of the condition. The conditions restrict the locations on the grid where a specific rule can be applied and by which player. Finally, I introduce a function that creates a tuple that contains a rule and a condition:

$$\text{MCR}(c, j) \coloneqq \text{MR}(c) \times \text{MC}(j).$$

To move a piece $p \in \mathcal{P}$ using an element (rule, condition) from MCR, the condition has to evaluate to True when applied to p. Only then can the rule be applied to p. I denote the set of the rule-and-condition tuples used in a specific game as $\mathcal{R}$. These conditions apply universally to all types on purpose. For assigning certain rules to types, I introduce the function $\mathcal{A}$ at a later point.

Example 3.2.3. For this example, I define a rule from $\text{MCR}(c = 1, j = 2)$ as follows:

$$r = (\text{down}, \text{or}(\text{eq}_{1,1,-1}, \text{eq}_{3,3,-1}))$$

The rule is restricted to game pieces in positions (1,1) and (3,3) by the condition. The logical mask created by the condition is displayed in Figure 3.4. Now, assume that the board state is the one created in Exercise 3.2.1. If *Right* wants to move their piece $(1, 1, 2, 1)$ down using r, they would first have to check if their piece satisfies the condition of r. Let us check it now: *Right*'s piece is of Type 2 and is located at position $(1, 1)$. The condition for r restricts the usage of down to pieces of any type at either position $(1, 1)$ or $(3, 3)$. As *Right*'s piece satisfies this condition, they are allowed to use down to move their piece to position $(2, 1)$. The resulting board state is also shown in Figure 3.4

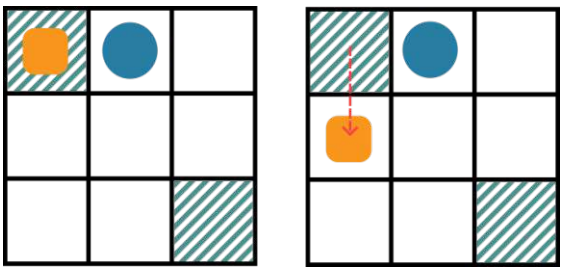

Figure 3.4: Logical Mask created by a condition and resulting board state

Actions

I denote the specific movement rules that are available to each game piece type as actions. For example, in Chess, the Knights have a different set of rules than the Queen or Rook. To implement this, I create a function $\mathcal{A}$ that maps from each type in $\mathcal{T}$ to subsets of rules in $\mathcal{R}$:

$$\mathcal{A}\colon \mathcal{T} \to 2^{\mathcal{R}},$$
$$\mathcal{A}(t) \coloneqq \{\mathcal{R}_t \mid \mathcal{R}_t \subseteq \mathcal{R}\}.$$

Example 3.2.4. Assume that I have defined 3 rules r_1, r_2 and r_3 in MCR:

$$r_1 = (\text{up}, \varnothing),$$
$$r_2 = (\text{down}, \varnothing),$$
$$r_3 = (\text{left}, \text{eq}_{-1,2,-1}).$$

r_1 and r_2 have no conditions, they can be freely used. r_3 is restricted to pieces located in any row on the second column. The set of rules now consists of $\mathcal{R} = \{r_1, r_2, r_3\}$. Assume now that the grid is the one introduced in Example 3.2.2. It contains two pieces and $\mathcal{T} = \{1, 2\}$. Using the sets $\mathcal{T}$ and $\mathcal{R}$ I can now define the function $\mathcal{A}$:

$$\mathcal{A} \coloneqq (1 \mapsto \{r_1, r_3\}, 2 \mapsto \{r_1, r_2\})\,.$$

The assignments made by $\mathcal{A}$ restrict players in moving game pieces of type 1 according to rules r_1, r_3 and game pieces of type 2 according to r_1, r_2. Assume now that *Left* wants

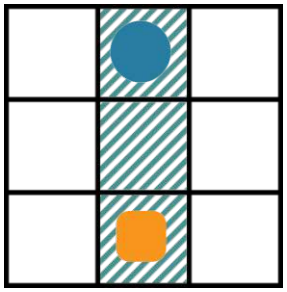

Figure 3.5: Logical Mask of condition $\text{eq}_{1,2,0}(-1, 2, -1)$

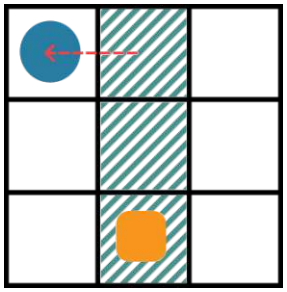

Figure 3.6: Board state after applying r_3 to p_1

to move their piece of type 1 at position $(1, 2)$. The subset of moves assigned to type 1 by $\mathcal{A}$ is $\{r_1, r_3\}$. If they choose to apply r_3, the condition, shown in Figure 3.5 is first checked:

$$p_1 = (1, 2, 1, 0),$$
$$\text{eq}_{1,2,0}(-1, 2, -1) = \text{True}.$$

For the selected piece, it evaluates to True. Thus, r_3 can be applied to the *Left*'s chosen piece and is moved one space to the left:

$$\hat{p}_1 = r_3(p_1) = \text{left}(p_1) = (1, 1, 1, 0).$$

The resulting board state is displayed in Figure 3.6.

Valid Rules

As discussed in previous paragraphs, rules are restricted by their type and conditions. In addition to these restrictions, there will be board states where rules can't be applied to some pieces, due to creating an illegal board state. This occurs if the destination of a move is out-of-bounds or overlaps another game piece. To represent the valid rules for each piece, I introduce the function $\mathcal{V}$:

$$\mathcal{V}\colon \mathcal{P} \to 2^{\mathcal{R}},$$
$$\mathcal{V}(row, col, type, player) \coloneqq \{R_p \mid R_p \subseteq \mathcal{A}(type)\},$$

where the rules in $\mathcal{R}_\mathrm{p}$ are only those that satisfy the following:

1. The piece p to be moved by the rule satisfies the rule's condition.
2. The target location is in the bounds of the grid.
3. The target location is unoccupied.

Example 3.2.5. For this example, assume the board state defined in Example 3.2.1. Furthermore, assume that there are the following rules:

$$r_1 = (\mathrm{up}, \varnothing),$$
$$r_2 = (\mathrm{down}, \mathrm{eq}_{2,2,-1}),$$
$$r_3 = (\mathrm{right}, \varnothing),$$
$$r_4 = (\mathrm{left}, \varnothing).$$

I also define the function $\mathcal{A}$ as follows:

$$\mathcal{A} \coloneqq (1 \mapsto \{r_4\}, 2 \mapsto \{r_1, r_2, r_3\})\,.$$

Now, *Right* wants to move their piece $p_2 = (1, 1, 2, 1)$. To do this, they will first have to check the valid rules for this piece using $\mathcal{V}(1, 1, 2, 1)$. Let us go through the process now, step by step. The subset of rules available to pieces of type 2 is $\mathcal{A}(2) = \{r_1, r_2, r_3\}$. First, for each of these rules, I check if p_2 satisfies their conditions. The only rule where the condition is not satisfied is r_2, as it only allows pieces at row 2 and column 2. Therefore, the set of valid rules reduces to $\{r_1, r_3\}$.

Second, I check if the location of p_2 after applying any of the rules in $\{r_1, r_3\}$ would be out of the bounds of the grid. As r_1 would move p_2 to $(-1, 1)$, it is also removed from the set of valid rules, leaving only $\{r_3\}$.

Third, I will check if the location of p_2 after applying any of the rules in $\{r_3\}$ overlaps another piece on the grid. r_3 is the only move left but would overlap p_1. Hence, it is also removed from the set of valid rules.

Therefore, the final set of valid rules for p_1 is the empty set, $\mathcal{V}(1, 1, 2, 1) = \varnothing$. Thus, *Right* can't move p_1!

Positional Win Conditions

I focus solely on normal play. This means the last player to take a move wins. This can be either winning by reaching a certain board position (e.g. three in a line in Tic-Tac-Toe) or reaching a specific global state of the whole board (e.g. opponent has no pieces left, thus can not move in Chess), as described by [Sie13]. In this section, I will introduce the first type.

I begin the definition of a positional win condition by introducing a helper function $hp : \mathcal{G} \to \mathcal{P}$, which is the indicator function of $\mathcal{P} \subseteq \mathcal{G}$:

$$\mathrm{hp}(row, col, type, player) = \mathbb{1}_{\mathcal{P}}(row, col, type, player) \coloneqq \begin{cases} 1 \text{ if } (row, col, type, player) \in \mathcal{P}, \\ 0 \text{ if } (row, col, type, player) \notin \mathcal{P}. \end{cases}$$

The hp function returns True if any game piece that equals the values defined by the parameters exists in the set $\mathcal{P}$. To allow any given value for any of the parameters, -1 is accepted and will validate that part of the expression as True.

Similar to defining rules, I can define a positional win condition w by combining $j-1$ of the boolean operators from $\{\mathrm{and}, \mathrm{or}, \mathrm{not}\}$ with the helper function hp. These combined functions always need to have a boolean function as the outermost function, with a helper function being the innermost functions. To represent the space of win conditions, I define the function $\mathrm{WP}(j)$ as follows:

$$\mathrm{WP}(j) \coloneqq \bigtimes_{i=0}^{j-1} \{\mathrm{and}, \mathrm{or}, \mathrm{not}\} \times \{\mathrm{hp}\}.$$

I refer to the set of all positional win conditions w that are used in a game as $\mathcal{W}$

Example 3.2.6. For this example, I assume a complexity of $j = 2$. A win condition with $j = 2$ will start with a boolean function and end with a helper function. I now define a win condition w_1 in $\mathrm{WP}(3)$ that checks for a line of *Left*'s game pieces on the grid:

$$w_1 = \mathrm{and}(\mathrm{hp}(0, 0, -1, 0), \mathrm{and}(\mathrm{hp}(1, 1, -1, 0), \mathrm{hp}(2, 2, -1, 0))).$$

The logical mask created by the win condition, along with a board position satisfying it, is shown in Figure 3.7.

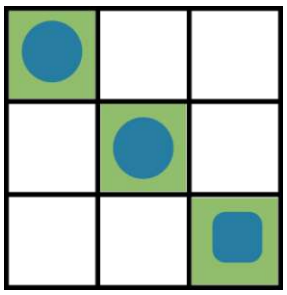

Figure 3.7: Example of a position satisfying win-condition w_1

Loss Conditions

The second type of game-ending condition concerns itself with the losing state of the board. It captures losing the game by, for example, possessing no more game pieces in Go or the King being unable to move in Chess. I begin the definition by introducing the function stuck:

$$\text{stuck}\colon \mathcal{P} \to \{\text{True}, \text{False}\},$$
$$\text{stuck}(row, col, type, player) \coloneqq \begin{cases} \text{True}, & \text{if } \mathcal{V}(row, col, type, player) = \varnothing, \\ \text{False}, & \text{otherwise.} \end{cases}$$

Additionally, I define a global loss condition that causes a player to lose if all of their pieces are stuck:

$$\text{all}\colon \{\mathit{Left}, \mathit{Right}\} \to \{\text{True}, \text{False}\},$$
$$\text{all}(player) \coloneqq \bigwedge_{p \in \mathcal{P}_{player}} \text{stuck}(p).$$

Since stuck is supposed to be able to "track" the available moves of specific game pieces and the set $\mathcal{P}$ does not define order, I have to introduce a sequence p as follows:

$$p\colon \mathbb{N} \to \mathcal{P},$$
$$\text{where } p_n \coloneqq p(n),$$

where the order of the sequence (p_n) is determined by the order in which the elements of $\mathcal{P}$ were added to the set.

I use a set of two of the Boolean operators, $\{\text{and}, \text{or}\}$, to describe more complex combinations of conditions defined by stuck. Similar to positional win conditions, I denote a function $\text{LC}(c)$ with a complexity $1 \leqslant c, c \in \mathbb{N}$. As with win-conditions, the last element in a stack of functions must always be the function movable, not a Boolean operator. Based on this, I define:

$$\text{LC}(c) \coloneqq \bigtimes_{i=0}^{c-1} \{\text{and}, \text{or}\} \times \{\text{stuck}\},$$

where the parameters of stuck are always an elements of the sequence (p_n). This is essential, as the values of the tuples in $\mathcal{P}$ will change during the game. Therefore, the pre-defined parameters of stuck would not "track" a specific piece in the game, which I want. Using a sequence that refers to specific elements in $\mathcal{P}$ solves this problem. To make it clear which player will lose when a condition triggers, I allow the function stuck to target only pieces belonging to a single player in each of the combined conditions in LC. Finally, I define the set of all loss conditions l in a game as $\mathcal{L}$. Naturally, all of these loss conditions depend on the initial board state, which I will introduce in the next section.

Example 3.2.7. Before I begin introducing a loss condition, I create the board shown in Figure 3.8. It contains three pieces, $\mathcal{P} = \{(3,1,1,0),(3,3,1,0),(1,1,2,1)\}$. Thus, the sequence (p_n) is defined as $p_1 = (3,1,1,0), p_2 = (3,3,1,0), p_3 = (1,1,2,1)$. Furthermore, I define the set of rules as $\mathcal{R} = \{(\text{down}, \varnothing)\}$. To create loss conditions, I assume a complexity of $c = 2$. Loss conditions with a complexity of 2 consist of a combination of an element from $b \in \{\text{and}, \text{or}\}$ and functions stuck. I define a loss condition l_1 in LC(2) as follows:

$$l_1 = \text{or}(\text{stuck}(p_1), \text{stuck}(p_2)).$$

Therefore, should *Left* not be able to move with either p_1 or p_2, they would lose the game. An example board position where this is the case, with only $(\text{down}, \varnothing)$ available as a rule, is displayed in Figure 3.8.

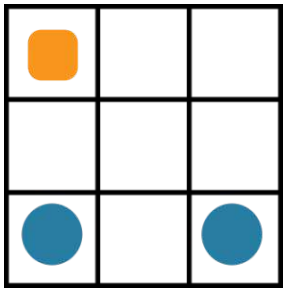

Figure 3.8: Board position that triggers loss-condition l_1 for *Left*.

Initial Board State

Some games may require the definition of a non-empty initial grid. Examples of this are Chess, Go, and Checkers. To this end, I need to define a function IP that returns the subset of all possible pieces for given grid dimensions n, k and set $\mathcal{T}$:

$$\text{IP} : \mathbb{N} \times \mathbb{N} \times \mathcal{T} \mapsto 2^{\mathcal{G}},$$
$$\text{IP}(n,k,\mathcal{T}) \coloneqq \{(row, col, type, player) \mid 1 \leqslant row \leqslant n, 1 \leqslant col \leqslant k,$$
$$type \in \mathcal{T}, player \in \{0,1\},$$

where row, col are natural numbers. This set defines all possible initial pieces for given grid dimensions and a set of types. The initial content of the set $\mathcal{P}$ is then a subset of the set defined by $\text{IP}(n,k,\mathcal{T})$.

Example 3.2.8. To start with a non-empty initial board for the players to move in, I create some game pieces. I assume an empty 3×3 grid to begin with. Furthermore, assume $\mathcal{T} = \{1,2\}$. I now create four initial pieces for $\mathcal{P}$ that are in $\text{IP}(3,3,\mathcal{T})$:

$$\mathcal{P} = \{(1,1,1,0),(1,3,2,0),(3,3,1,1),(3,1,2,1)\} \subseteq \text{IP}(3,3,\mathcal{T}).$$

This initial board configuration is displayed in Figure 3.9.

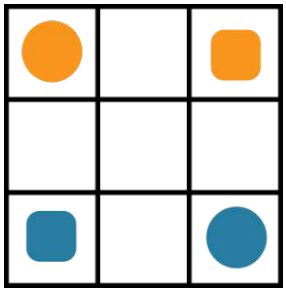

Figure 3.9: An initial board position with pieces in $\mathrm{IP}(3, 3, \mathcal{T})$.

For a complete end-to-end example of creating a game and then playing it out, I refer to Appendix C.1.

3.2.2 Sampling 2D-Grid Games

In the following, I introduce multiple methods that allow us to sample all of the components of a game I introduced in Section 3.2.1. I also reproduce the deterministic examples for each component using probabilistic methods.

Grid Dimensions

I start by outlining the process of sampling the dimensions of the grid. To make definitions more concise, introduce the set notation $[n] \coloneqq \{1, 2, \ldots, n\}$, which defines sets from 1 to n. Then, I construct the space of the variables n and k that define the dimensions of the grid as follows:

$$S_{\text{grid}} = [10] \times [10].$$

Next, I define a probability distribution over this space:

$$P(\mathbf{N} \cap \mathbf{K})\colon S_{\text{grid}} \to [0, 1].$$

To generate dimensions for the grid, I can sample a pair of values for (n, k) from this distribution.

Example 3.2.9. To reproduce the grid defined in Example 3.2.1, I assume that $P(\mathbf{N} \cap \mathbf{K})$ is deterministic: $P(\mathbf{N} = 3 \cap \mathbf{K} = 3) = 1$. I then sample this distribution to produce a pair of values for the grid dimensions n and k:

$$(3, 3) \sim P(\mathbf{N} \cap \mathbf{K}).$$

Choosing this specific distribution would always replicate the empty 3×3 grid from Example 3.2.1.

Types

As the next step, I sample the types of game pieces $\mathcal{T}$. I first define the space of all possible types as:

$$S_{\text{types}} = \Big\{ [i] \mid i \in [10] \Big\}.$$

Now, I create a probability distribution that covers this space:

$$P(\mathbf{T})\colon S_{\text{types}} \to [0, 1].$$

To fill the set of types $\mathcal{T}$, it is sufficient to sample from this distribution.

Example 3.2.10. Again, I assume that the distribution $P(\mathbf{T})$ is deterministic: $P(\mathbf{T} = [2]) = 1$. To create $\mathcal{T}$, I sample this distribution:

$$\mathcal{T} = [2] \sim P(\mathbf{T}).$$

This result replicates the set $\mathcal{T}$ from Example 3.2.4.

Initial Game Pieces

To fill a grid with an initial set of game pieces, I first have to create the space of all game pieces for the given grid dimensions n, k and set of types $\mathcal{T}$. For this, I utilize the function IP introduced earlier. Then, I can define a probability distribution that covers all possible initial game pieces for the given n, k and $\mathcal{T}$:

$$P(\mathbf{P})\colon \text{IP}(n, k, \mathcal{T}) \to [0, 1].$$

By sampling this distribution, I can create game pieces p that initially make up the contents of the set $\mathcal{P}$.

Example 3.2.11. To replicate Example 3.2.8, I first construct a distribution with spikes in all corners of the grid as follows:

$$\begin{aligned} P(\mathbf{P} = (1, 1, 1, 0)) &= 0.25, \\ P(\mathbf{P} = (1, 3, 2, 0)) &= 0.25, \\ P(\mathbf{P} = (3, 3, 1, 1)) &= 0.25, \\ P(\mathbf{P} = (3, 1, 2, 1)) &= 0.25. \end{aligned}$$

Before sampling any pieces, remember that per definition the set $\mathcal{P}$ contains unique game pieces. Thus, to create a set of four pieces, I sample until $|\mathcal{P}| = 4$:

$$\mathcal{P} = \big\{(1, 1, 1, 0), (1, 3, 2, 0), (3, 3, 1, 1), (3, 1, 2, 1)\big\} \sim P(\mathbf{P}).$$

This set of initial pieces $\mathcal{P}$ replicates the one created in Example 3.2.8.

Rules

To sample rules and their conditions, I begin by defining a probability distribution over the space created by the function $\mathrm{MCR}(c, j)$. To this end, I first introduce a joint probability distribution over the space of the parameters c, j:

$$P(\mathbf{C} \cap \mathbf{J})\colon [5] \times [5] \to [0, 1].$$

Furthermore, I define a probability distribution with fixed parameters c, j over the space $S_{\mathrm{MCR(c,j)}}$ created by $\mathrm{MCR}(c, j)$:

$$P(\mathbf{R}_c \cap \mathbf{B}_j)\colon S_{\mathrm{MCR}(c,j)} \to [0, 1].$$

Note that I use $\mathbf{B}$ as the random variable for the condition (i.e., **b**ounds) of the rule to avoid confusion with another that I previously introduced. This distribution can then be sampled to create a rule and condition for a sampled set of parameters. I denote the set of all rules and condition pairs sampled in this manner as $\mathcal{R}$.

Example 3.2.12. To replicate the rules and conditions pairs defined in Example 3.2.3, I first define the previously introduced distributions:

$$\begin{gathered} P(\mathbf{C} = 1 \cap \mathbf{J} = 2) = 1, \\ f \coloneqq \mathrm{up}, \\ g \coloneqq \mathrm{or}(\mathrm{eq}_{1,1,-1}, \mathrm{eq}_{3,3,-1}), \\ P(\mathbf{R}_1 = f \cap \mathbf{B}_2 = g) = 1. \end{gathered}$$

To now reconstruct the rules and conditions in Example 3.2.3, I sample these distributions:

$$\begin{gathered} (1, 2) \sim P(\mathbf{C} \cap \mathbf{J}), \\ (\mathrm{up}, \mathrm{or}(\mathrm{eq}_{1,1,-1}, \mathrm{eq}_{3,3,-1})) \sim P(\mathbf{R}_1 \cap \mathbf{B}_2). \end{gathered}$$

Actions

To sample the assignments made by the function $\mathcal{A}$, I will define a probability distribution over the space of all subsets of the rules in $\mathcal{R}$. I begin by defining the space of all subsets as:

$$\mathcal{S}_{\mathrm{actions}} = 2^{\mathcal{R}}.$$

I can now define a conditional probability distribution over the combination of the types $\mathcal{T}$ and the subsets of $\mathcal{R}$ as follows:

$$P(\mathbf{R} \mid \mathbf{T} = t)\colon \mathcal{S}_{\mathrm{actions}} \times \mathcal{T} \to [0, 1].$$

I condition $\mathbf{T}$ on t, as the values are known (i.e., the parameters of $\mathcal{A}$). To allow the function $\mathcal{A}$ to sample the subsets of the target domain, I have to redefine it using the

probability distribution. First, since I still want $\mathcal{A}$ to return the same subset of $\mathcal{R}$ each time it is called, I define a helper tuple of sampled subsets:

$$\mathcal{H} = (\mathcal{R}_t \mid t \in \mathcal{T}, \mathcal{R}_t \sim P(\mathbf{R} \mid \mathbf{T} = t)).$$

Using the helper tuple, $\mathcal{A}$ can now be redefined to return a sampled subset for each type in $\mathcal{T}$: $\mathcal{A}(t) \coloneqq \mathcal{H}_t$.

Example 3.2.13. To recreate the function $\mathcal{A}$ from Example 3.2.4 via sampling, I first need to define the set of rules as follows:

$$\mathcal{R} = \{r_1, r_2, r_3, r_4\}.$$

The rules r_1 to r_4 are the same as in Example 3.2.4. Also assume that $\mathcal{T} = \{1, 2\}$. I now define the conditional probabilities of the distribution I introduced above as deterministic:

$$P(\mathbf{R} = \{r_1, r_3\} \mid \mathbf{T} = 1) = 1,$$
$$P(\mathbf{R} = \{r_1, r_2\} \mid \mathbf{T} = 2) = 1.$$

With these assignments made, the function $\mathcal{A}$ I re-defined above, will return the following subsets of $\mathcal{R}$ for each element of $\mathcal{T}$:

$$\mathcal{A}(1) = \{r_1, r_3\},$$
$$\mathcal{A}(2) = \{r_1, r_2\}.$$

Thus, I have successfully replicated Example 3.2.4 by sampling a probability distribution!

Win Conditions

To define a probability distribution over all win conditions, I first need to create a distribution over the space of the parameter j:

$$P(\mathbf{J})\colon [5] \to [0, 1].$$

Using the space $S_{\mathrm{WP}(j)}$ created by the function $\mathrm{WP}(j)$, I define the following probability distribution over it:

$$P(\mathbf{W}_j)\colon S_{\mathrm{WP}(j)} \to [0, 1].$$

To create a win condition w, I first sample a parameter j from $P(\mathbf{J})$ and then a win condition from the distribution $P(\mathbf{W}_j)$. I denote the set of all win conditions w in a game sampled in this way as $\mathcal{W}$.

Example 3.2.14. I begin the reproduction of Example 3.2.6 by defining $P(\mathbf{J})$ and $P(\mathbf{W}_j)$ as deterministic:

$$P(\mathbf{J} = 2) = 1$$
$$f \coloneqq \mathrm{and}(\mathrm{hp}(0, 0, -1, 0), \mathrm{hp}(1, 1, -1, 0), \mathrm{hp}(2, 2, -1, 0)),$$
$$P(\mathbf{W}_2 = f) = 1.$$

Finally, I replicate the win condition from Example 3.2.6 by first sampling the parameter j from $P(\mathbf{J})$, then using it to sample our win condition distribution $P(\mathbf{W}_j)$:

$$2 \sim P(\mathbf{J}),$$
$$\text{and}(\text{hp}(0,0,-1,0), \text{hp}(1,1,-1,0), \text{hp}(2,2,-1,0)) \sim P(\mathbf{W}_2).$$

Loss Conditions Similar to the previous section, I begin by defining a probability distribution over the space of the function LC(c)'s parameter c:

$$P(\mathbf{C})\colon [5] \to [0,1].$$

Using the space $S_{\text{LC}(c)}$ created by the function LC(c), I can define a probability distribution over it:

$$P(\mathbf{L}_c)\colon S_{\text{LC}(c)} \to [0,1].$$

To sample a loss condition, I begin by sampling the parameter c from $P(\mathbf{C})$ and then use it to sample from $P(\mathbf{L}_c)$. I refer to the set of all loss conditions l in a game sampled in this manner as $\mathcal{L}$. This set also includes the two default loss conditions all(*Left*) and all(*Right*) defined in Section 3.2.1.

Example 3.2.15. To reproduce Example 3.2.7, I first define the distribution $P(\mathbf{C})$ as deterministic:

$$P(\mathbf{C} = 2) = 1.$$

Next, I also define the distribution $P(\mathbf{L}_c)$ for the parameter value of $c = 2$ as deterministic:

$$f \coloneqq \text{or}(\text{movable}(p_1), \text{movable}(\text{p}_2)),$$
$$P(\mathbf{L}_2 = f) = 1.$$

First, I sample c from $P(\mathbf{C})$. Then, using this value, I sample a loss condition from $P(\mathbf{L}_c)$:

$$2 \sim P(\mathbf{C}),$$
$$\text{or}(\text{movable}(p_1), \text{movable}(\text{p}_2)) \sim P(\mathbf{L}_2).$$

The sampled loss condition is the same as in Example 3.2.7.

Sampling Component Quantities

In the previous paragraphs, I have introduced all the necessary tools to sample the components of a game. However, a crucial aspect remains undefined: sampling the quantities for rules, initial game pieces, win conditions, and loss conditions. Defining these distributions is straightforward, as they need only generate an integer. I define the

following distributions, denoting the random variables for each quantity with a subscript of the set of the respective component:

$$P(\mathbf{N}_{\mathcal{R}})\colon [10] \to [0,1],$$
$$P(\mathbf{N}_{\mathcal{P}})\colon [n \times k] \to [0,1] \text{ where } n,k \text{ are the grid dimensions},$$
$$P(\mathbf{N}_{\mathcal{W}})\colon [5] \to [0,1],$$
$$P(\mathbf{N}_{\mathcal{L}})\colon [5] \to [0,1].$$

Sampling these distributions defines the exact quantities of rules, initial game pieces, win and loss conditions for a game. For a complete end-to-end example of sampling a game and then playing it out, I refer to Appendix C.2.

3.2.3 The Gameplay Loop

I can easily construct a formal way to represent the gameplay between *Left* and *Right*. In the following, I will use game-theoretic concepts, such as states, and action sets for both players and the transition function. Each game begins with a board that has the initial pieces in $\mathcal{P}$ placed on it. I refer to this board as s_0. In all of our games, *Left* and *Right* alternate taking actions, with *Left* beginning in s_0. During their turn, a player chooses an action from A_{player}, which is a tuple of a game piece and a valid rule. Once an action a_i is chosen by the active player, the transition function is applied to it and the state. The transition function applies the rule to the game piece contained in the chosen tuple, constructing the next state:

$$T(s_{i-1}, a_i) = s_i.$$

This also passes the turn to the other player, who will choose an action, apply the transition function, and so forth. For each state created in this way, the win and loss conditions are checked. Should any of them be satisfied, the game has reached a terminal state and ends. An example of a complete game from start to finish can be found in the Appendix C.3.

3.2.4 Sampling a Whole Game

In this short section, I will provide an overview of how to sample a complete game, step by step:

1. Sample grid dimensions n and k: $n, k \sim P(\mathbf{N} \cap \mathbf{K})$.
2. Sample types in $\mathcal{T}$: $\mathcal{T} \sim P(\mathbf{T})$.
3. Sample initial game pieces in $\mathcal{P}$:
   a) Sample the amount of game pieces n_{p}: $n_{\mathrm{p}} \sim P(\mathbf{N}_{\mathcal{P}})$.
   b) Sample n_{p} game pieces and create $\mathcal{P}$: $\mathcal{P} = \{p_i \sim P(\mathbf{P}) \mid 1 \leqslant i \leqslant n_{\mathrm{p}}\}$.
4. Sample rules and their conditions in $\mathcal{R}$:

a) Sample the amount of rule and condition pairs n_{r}: $n_{\mathrm{r}} \sim P(\mathbf{N}_{\mathcal{R}})$.

b) Sample n_{r} pairs of the complexity c, j parameters for the rules:

$$\{(c_i, j_j) \sim P(\mathbf{C} \cap \mathbf{J}) \mid 1 \leqslant i \leqslant n_{\mathrm{r}}\}.$$

c) For each complexity pair, sample a rule and condition pair to create $\mathcal{R}$:

$$\mathcal{R} = \left\{(r, c) \sim P(\mathbf{R}_c \cap \mathbf{B}_j) \mid c, j \in \{(c_1, j_1), (c_2, j_2), \dots\}\right\}.$$

5. To create the subset assignments made by $\mathcal{A}$, it is sufficient to create the helper tuple $\mathcal{H}$. This tuple contains a sampled subset of rules for every type. The $\mathcal{A}$ function then maps from each type in $\mathcal{T}$ to the appropriate sampled subset in $\mathcal{H}$.

6. Sample win conditions in $\mathcal{W}$:

a) Sample the amount of win conditions n_{w}: $n_{\mathrm{w}} \sim P(\mathbf{N}_{\mathcal{W}})$.

b) Sample n_{w} complexities j for the win-conditions:

$$\{j_i \sim P(\mathbf{J}) \mid 1 \leqslant i \leqslant n_{\mathrm{w}}\}.$$

c) For each complexity j, sample a win condition to create $\mathcal{W}$:

$$\mathcal{W} = \left\{w_i \sim P(\mathbf{W}_j) \mid j \in \{j_1, j_2, \dots\}\right\}.$$

7. Sample loss conditions in $\mathcal{L}$:

a) Sample the amount of loss conditions n_{l}: $n_{\mathrm{l}} \sim P(\mathbf{N}_{\mathcal{L}})$.

b) Sample n_{l} complexities c for the loss conditions:

$$\{c_i \sim P(\mathbf{C}) \mid 1 \leqslant c \leqslant n_{\mathrm{l}}\}.$$

c) For each complexity c, sample a loss condition to create $\mathcal{L}$:

$$\mathcal{L} = \left\{l_i \sim P(\mathbf{L}_c) \mid c \in \{c_1, c_2, \dots\}\right\}.$$

3.2.5 Design Choices

This short section will introduce several design choices for the game sampling process. I will begin by introducing which distributions I choose for the sampling process, give an overview of the approximate size of the space, and then go over how unplayable games are handled.

Choosing Distributions

Let us begin with the distribution of the grid dimensions, $P(\mathbf{N} \cap \mathbf{K})$. For this, I choose a discrete distribution centered around the middle of the range of the random variables. This results in approximately square grids, with only outliers having "weird" dimensions of, e.g. 1×10. This allows for more interesting games, as most rules of higher complexity will not work many outlier grids.

For all of the other distributions, I rely on the uniform random distribution. This choice comes with the benefit of the least amount of prior information about our generated games. This avoids the issue of the LLMs being tested exploiting the priors of our generated games to enable improved performance, for example, by constructing training sets consisting of the peaks of other distributions.

Unplayable Games

Sampling from the space of all games will inevitably result in unplayable games. I will now show a few examples of generated games that would not be deemed playable.

Example 3.2.16. Assume a game generated without initial pieces, $|\mathcal{P}| = 0$. Furthermore, assume that $\mathcal{R}$ contains a few rules, but place $\notin \mathcal{R}$. Therefore, the first player will instantly lose the game, as they have no moves available, thus triggering loss condition all(0).

Example 3.2.16 shows just one of many unplayable games. Other than the special case of having no pieces on the board, the case that the rules are too complex or the conditions too restrictive is also common in unplayable games. I show an instance of this in Example 3.2.17.

Example 3.2.17. Assume that I have a 3×3 board and that each player has one game piece in opposing corners:

$$\mathcal{P} = \big\{(1,1,1,0),(3,3,1,0)\big\}.$$

For the sampled rules, I assume that $\mathcal{R}$ contains the following pairs:

$$r_1 = (\text{up}, \text{eq}_{-1,-1,0}),$$
$$r_2 = (\text{down}, \text{eq}_{-1,-1,1}).$$

Both players can not use any of the rules due to the conditions restricting their use.

Another common cause of unplayable games is those where a piece in the initial set $\mathcal{P}$ satisfies a win or loss condition. Therefore, an end state is reached before any player can take a move. Example 3.2.18 shows an initial state that instantly terminates a game without either playing using a rule.

Example 3.2.18. Assume a 3×3 board with one initial piece $p_1 = (0, 0, 1, 1)$. Furthermore, I define the set of win conditions as follows:

$$\mathcal{W} = \{\text{hp}(0, 0, 1, \text{-}1)\}.$$

The win condition is satisfied if either player reaches the top left corner of the board. Since player 1 already has a piece in this location (since it is his initial game piece), they will instantly win the game before the starting player is even allowed to move.

As there are many factors at play to generate a playable game, a vast amount of the space of all games I could generate will be unplayable. However, since games are very cheap to generate, generating unplayable games is not an issue. I conduct a simple test for any generation that allows us to filter out unplayable games. If any of the below criteria are satisfied, I discard the generated game:

1. Any win or loss condition is satisfied by the initial board state.
2. *Right* (i.e., the starting player), has no valid moves available in the initial board state.

3.3 Creating a Combined Complexity Metric

3.3.1 Estimating the Difficulty of a Game

To allow players of our games to have a sense of difficulty, I introduce a complexity metric for those generated with the framework introduced in Section 3.2.2. To begin with, I go back to remember how a game is played. The process of playing a game can be represented as its history, a sequence of state and action pairs. Playing multiple plays of the same game creates various histories. They can be combined to form a game tree, as introduced in Section 3.1.5, using the states as nodes and the actions as the edges between nodes. The notion of a game tree allows us to use heuristic search algorithms, such as Monte Carlo tree search (MCTS). To define our complexity metric, I will use an MCTS-based implementation of "intrinsic task difficulty" proposed in "The Measure of All Minds" [HO17]. In the following paragraphs, I summarize the notation and ideas introduced by [HO17].

Tasks

First, it is necessary to define a task: An abstract framework that describes the concept of something an agent can solve using a policy. A concrete implementation of a task is then referred to as an instance. In our case, the task would be the abstract framework described in Section 3.2.1, and an instance would be a game from this space, for example, as shown in the Appendix C.1.

Policies

A policy has three main components that contribute to the "difficulty" of a solution, namely the resources, search steps, and information required to reach a specific task. Since it is unrealistic to calculate this for arbitrary tasks, the definition of difficulty proposed by [HO17] will be an approximation. Nevertheless, it is useful to compare tasks against each other.

Difficulty

Using this view of policies, the difficulty of a task can now be defined as the effort required to build a policy that solves it. Solving a task is interpreted as accepting a solution over v trials with a tolerance ϵ:

$$\mathcal{A}^{[\epsilon,\mapsto v]}(\mu) \coloneqq \{\pi : \mathbb{RE}^{[\mapsto v]}(\pi,\mu) \geqslant 1-\epsilon\}.$$

where μ is the task, ϵ the tolerance, π the policy, and $\mathbb{RE}^{[\mapsto v]}(\pi,\mu)$ denotes the response after v-many trials. I refer to [HO17], Chapter 8.3 (I slightly changed the original notation $\mathbb{R}^{[\mapsto v]}(\pi,\mu)$, to avoid confusing it with the symbol $\mathbb{R}$ for real numbers). Using the acceptance $\mathcal{A}$, can define a measure of difficulty as:

$$\hbar^{[\epsilon,\delta,\mapsto v]}(\mu) \coloneqq \min_{\pi\in\mathcal{A}^{[\epsilon,\mapsto v]}(\mu)} \mathbb{F}^{[\epsilon,\delta,\mapsto v]}(\pi,\mu).$$

This can interpreted as the minimum acquisition effort $\mathbb{F}$ of any ϵ-acceptable policy μ for the task π in v trials with confidence δ. In this representation, $\mathbb{F}$ is a measure of effort that considers the following elements:

- L . . . Policy Length
- $\mathbb{S}(\pi,\mu)$ . . . Execution steps
- $\mathbb{W}(\pi,\mu)$ . . . Verification steps, depends on confidence δ
- ϵ . . . Tolerance
- v . . . Number of trials

For a full list of all features and dependencies of $\mathbb{F}$ read Table 8.2 in [HO17]. For the notion of task difficulty, [HO17] focus on a combination of the policy length L and the Execution steps $\mathbb{S}$. L is simply the length of a program that an agent can execute. This can be interpreted in how difficult a policy is to acquire. $\mathbb{S}$ are the execution steps per trial of π when performing task μ, which can be interpreted as how computationally complex a policy is. The combination of L and $\mathbb{S}$ yields the finding effort $\mathbb{LS}$, which is defined as follows:

$$\mathbb{LS}^{[\mapsto v]}(\pi,\mu) \coloneqq L(\pi) + \log \mathbb{S}^{[\mapsto v]}(\pi,\mu).$$

This function can be interpreted in how difficult finding a specific policy through search is. Using $\mathbb{LS}$ to instantiate the finding effort $\mathbb{F}$, the algorithmic difficulty for a given task can be given as

$$K_t^{[\epsilon,\mapsto v]}(\mu) \coloneqq \min_{\pi \in \mathcal{A}^{[\epsilon,\mapsto v]}(\mu)} \mathbb{LS}^{[\mapsto v]}(\pi, \mu).$$

Thus, algorithmic difficulty is defined as the optimal ϵ-acceptable policy in terms of the sum of its size and the logarithm of the number of its execution steps. It is important to note that, as mentioned before, this is an approximation that depends on the search algorithm (execution steps and program length), as well as the tolerance. For our use case, this is not an issue at all since I will use the same search algorithm and parameters for all generated games. As one may have noticed, this formulation is similar to that of Kolmogorov complexity and is used as a computable version of it [HO17].

Search Algorithm

A straightforward policy search algorithm that is commonly used to solve games [SHS+18], [SHM+16], [SB18] is Monte Carlo tree search (see Chapter 8.11 in [SB18]), MCTS for short. This search algorithm works by iteratively simulating games and updating a lookup table for a partial action-value function (e.g. learning a policy for a game). An advantage of MCTS is that it runs at decision time, requiring no prior training. Other reinforcement learning methods, like Q-Learning [SB18], require extensive training and fine-tuning to provide adequate results.

The disadvantage of operating at decision time comes in the form of the many simulations required by the algorithm. If an environment is expensive to simulate, MCTS will take a long time to find an adequate policy. For our use case, I do not have this problem, as the actions and checks that make up our game are cheap and not complicated to execute.

Another factor to consider with MCTS is the trade-off between exploration and exploitation, which is done by the action selection policy. For most modern versions of MCTS, like AlphaGo [SHM+16] or AlphaZero [SHS+18], this is implemented by a neural network trained via self-play ahead of time. This works well for algorithms designed to play a single type of game, as they do not need to generalize. Since I focus on a large number of possible games, this would require us to train a separate neural network (or a *very* general one) for each generated game, increasing computational effort drastically. Therefore, I will utilize a common action selection rule, the Upper Confidence Bound (UCB) [SB18].

Approximating Difficulty

To approximate the difficulty of our generated games, I combine ideas from the mathematical framework provided by [HO17] with MCTS.

Our basic idea is to run MCTS with UCB for a fixed number of simulations for each game. I then use the "policy", represented as a tree of board states as nodes and actions as edges, as the program as the basis for calculating the difficulty metric proposed earlier

in this section. To simulate the search process for a policy, I do the following steps for each game's initial state:

1. Define a list of fixed simulation numbers for MCTS, i.e. $(10, 50, 100, 500)$.
2. Take the lowest unused simulation budget from the list, i.e. 10.
3. Simulate v games of an MCTS agent with the given simulation budget against a random agent.
4. For each trial, record the outcome and the average number of games played.
5. If the win rate over v games exceeds $1-\epsilon$, the policy is accepted, and I can calculate our metric.
6. If the policy is not accepted, continue with an increased simulation budget with MCTS at Step 2.
7. If the maximum number of simulations is exceeded, the game has reached the maximum difficulty of our metric.

To form the components of the difficulty metric, I start by using the returned number of explored nodes as an approximation of the policy length L. This will always be the same number as the simulation budget, as each simulation step explores exactly one node. Furthermore, I can use the minimum game steps of the v trials to approximate the execution steps $\mathbb{S}$.

In the following, I refer to the search tree of policy π as G_π and its set of vertices (i.e., nodes) as V_π. Thus, our difficulty metric for games can be constructed as follows:

$$K_t(\mu)^{[\epsilon,\mapsto v]} \approx |V_\pi| \times \text{minimal game length.}$$

Of course, this metric has a dependency on the exploration-exploitation trade-off, which is unavoidable. However, since I use the same UCB parameters for all of our games, I expect that this will not drastically skew the resulting metric. Furthermore, the training process itself naturally depends on MCTS and the maximal number of simulation steps. Since I don't have unlimited resources, it is not feasible to run an extremely large amount of simulations.

However, since our goal is only to provide a comparable metric between our generated games, these factors should not become a problem, as the policy search for each game is restricted in the same way.

3.4 LLM Evaluation

To evaluate the reasoning capabilities of LMaaS I construct an automatic evaluation framework in Python code for the grid games introduced in Section 3.2. In addition to grid games, I also implement a few "static" games, like Tic-Tac-Toe or ConnectFour, although I focus solely on the former for LLM evaluation. Our framework consists of three main components, a (game) Generator, a Solver, and a Prompter. I will discuss the responsibilities of each in the following paragraphs. A visual representation is displayed in Figure 1.1.

Generator

The Generator is at the heart of our framework. Its purpose is to generate new states for a given game. These states represent the "reasoning step" to be solved by an LLM. For static games, each state is generated from a random playout from the game's initial state. With grid games, I first sample a random game as described in Section 3.2.2. Using this sampled game, I then generate a random playout and return one of the intermediate states. Each state generated by this module also provides a rule set, which represents the ins and outs of how the game is played in natural language. These rule sets will later be used to construct the prompts given to the LLMs. Finally, the Generator also calculates an approximate complexity for each state using the methods described in Section 3.3.

Prompter

The next of the three components is the Prompter. It handles all of the communication with an LLM. To make a game playable by text-based agents, like LLMs, the Prompter first converts each board state into a natural language representation. I choose a simple grid-based approach, using the "|" as the dividers between cells. The state in natural language is then placed into a prompt that consists of the related game rules, win and loss conditions, and a description of the required answer format. An example of complete prompts can be seen in Appendix B. Once a prompt is constructed, the Prompter communicates it to an LLM and parses the answer. To allow for the improvement of an initial answer, I allow each LLM a second chance by asking it to fix any prior mistakes. The final answer given by the LLM is parsed by the Prompter, which will be evaluated by our next module, the Solver.

Solver

Our final component is the Solver. As the name suggests, it is tasked with providing a heuristic for the best actions an agent could take for each given state. At this point, it is important to note that **I do not solve the generated games in the game theoretic sense**. This would require the Solver to find an optimal game plan for both players for all states, which is not feasible given the scope and aim of this work. As an approximation, I use MCTS with a simulation budget of 1000 steps for a given state to find the best valid actions. This suffices to find a better than random ranking of the next actions for

an agent. To ensure that I can compare each ranking, the action scores Q are normalized to a range of 0 to 1 (0 is the worst action, 1 is the best) using min-max scaling:

$$Q'_i = \frac{Q_i - \min(Q)}{\max(Q) - \min(Q)}.$$

Another task given to the Solver is the verification of the answers that the Prompter received from the LLMs. As I already generated rankings for each valid action, this will return either the score of the chosen move if it is valid, or a special value indicating an invalid move otherwise.

Using these three components, I construct the automatic generation and evaluation pipeline displayed in Figure 1.1. Summarizing, our approach works as follows: a random state for a sampled (or static game) is created by the game Generator. This state is then passed to the Prompter, which converts the state into a natural language prompt, passes it to an LLM, and then parses the answer. Finally, the Solver calculates an approximate ranking of the best actions for the state and scores the LLM's answer based on it.

All of the states (and games) generated in this way are persisted in a local database. This allows us to precisely control which samples are used for experiments by using specific subsets of the states saved in the database. When running an experiment, the outcome for each of the sampled states is recorded by a metrics module, which outputs both a chat log and a JSON document containing the complete game state and the LLM's solution to it, along with the average action ranking of the tested LLM. I use the following metric to calculate the average action ranking:

$$\bar{Q} = \frac{1}{n} \sum_{i=1}^{n} Q_i,$$

where Q are the action rankings of all tested samples. As all action rankings are normalized, $\bar{Q}$ is in the range of 0 to 1. For this metric, I treat invalid chosen actions and failures to reply in the correct format as an action ranking of 0.

3.4.1 OOD Evaluation

For our experiments, I hypothesize that the grid games I introduce in Section 3.2.1 are out-of-distribution (OOD) of LLMs. To test this, I analyze the distribution shift between commonly known games that are surely in distribution, such as Tic-Tac-Toe, and our generated grid games. This type of distribution shift is referred to as "background shift", which captures changes in the domain or style of the dataset content [YCC+23]. For this purpose, I will leverage the evaluation scheme proposed by [YCC+23] and compare the 4-shot learning abilities of LLMs using the average action rank as the performance metric. I opt not to fine-tune small LLMs, as evaluating LMaaS with undisclosed model weights and datasets is at the heart of our research. Therefore, I will also not use approaches to measuring distribution shifts, that use fine-tuning of small LLMs, like [YCC+23]

and [CPBD+23] suggest. Furthermore, testing models that are not "black-box" is not the focus of our work. I aim to provide a framework to test LMaaS with unavailable weights, training data, etc. as mentioned before!

LLMs

For our tests, I will conduct experiments using the following three LMaaS:

- OpenAI GPT-4o [Ope],
- Anthropic AI Claude 3 Sonnet [AI],
- Google Gemini 1.0 Pro [Dee].

I chose these models as they are the top end of the state-of-the-art LMaaS at the time of writing. For our experiments, I will rely on the responses generated by the API interfaces offered by each service.

2D Grid Games Samples

I begin our evaluation by constructing an out-of-distribution test set of $n_{\text{OOD}} = 500$ game states using the generation capabilities of our framework. Each generated sample consists of one playable state, ranked by the complexity metric introduced in Section 3.3 with approximate action rankings. During the generation process, any game states that do not have valid moves available (the game is won/lost already, all pieces can not move, etc.) are discarded. I call this dataset "Grid-Games".

Tic-Tac-Toe Default Samples

To evaluate the in-distribution game the well-known game Tic-Tac-Toe, which I can assume is in the distribution of all LLMs (all of the ones I test can explain the game's rules in great detail). Using our framework, I again sample a test set of $n_{ID} = 500$ game states. I call this dataset "Tic-Tac-Toe Default". Note however that this dataset uses a different type of prompts than the "Grid-Games" dataset, which more closely represents the common representation of the game Tic-Tac-Toe (board with X's and O's).

Tic-Tac-Toe Grid Game Samples

I also provide a test set of $n_{TIC} = 500$ samples of the game Tic-Tac-Toe recreated using our Grid Game framework. In contrast to the "Tic-Tac-Toe Default" dataset, it uses the same prompt template, and answer format as the "Grid-Games" dataset. I call this dataset "Tic-Tac-Toe Grid Game".

2D Grid Games

As is commonly done in current literature (see Section 2), I utilize few-shot prompting to test LMaaS, which has been shown to increase performance greatly. Few-shot in our case means I prompt the LMaaS n times and select only the best-performing answer. Due to technical reasons, I are limited to using 4-shot prompting, as elaborated on in Section 4. Using our framework, I evaluate the test set "Grid-Games" using 4-shot prompting. The results are displayed in the top third of Table 3.2.

For nearly all models, I can observe that they are not very good at choosing valid actions, which is understandable when playing new games that they are new to. I can also see, that the three models seem to be able to adhere to the answer format in most cases, with only Claude 3 Sonnet being noticeably worse than the other two.

Tic-Tac-Toe Grid Game

Again using 4-shot prompting, I now evaluate the in-distribution game Tic-Tac-Toe using the "Tic-Tac-Toe Grid Game", using the same prompt templates and representations as the "Grid-Games" experiment. An example of the prompt and answer templates is displayed in Example B. Due to the game being known by all of the LLMs I test, I expect to see an increase in performance in comparison to our out-of-distribution games. The results collected by our framework are shown in the middle third of Table 3.2.

Indeed, I see a large increase in performance for all tested models when compared to the "Grid-Games". Additionally, I can observe that the models take valid actions for nearly all samples, and none of them answered in an invalid format. However, it has to be said that choosing a valid action is easier for the "Tic-Tac-Toe Grid Game" than for other "Grid-Games", as there is only one action with no conditions for the models to take for this task.

Tic-Tac-Toe Default

Finally, I use 4-shot prompting again to evaluate the in-distribution game of Tic-Tac-Toe, represented in the "common form", using X's and O's as symbols in a 2D grid, with number labels for the LMaaS to choose from. An example prompt is displayed in Example B. I expect similarly good performance to the "Tic-Tac-Toe Grid Game" dataset, as the LLMs should be able to exploit their learned knowledge about the game. The results for this dataset are displayed in the lower third of Table 3.2.

Distribution Shift

From our experimental results, it is visible that the LMaaS I test exhibit a stark difference in performance when playing the out-of-distribution "Grid-Games" and the in-distribution games "Tic-Tac-Toe" and "Tic-Tac-Toe Grid Game". Below, in Figure 3.10, I provide an overview of the distribution shifts between the average overall rank between our proposed

LLM	Avg. Overall Rank	Avg. Valid Rank	Invalid Format	Invalid Action
GPT-4o	0.2694	0.9226	14%	70.80%
Gemini 1.0 Pro	0.2604	0.9365	5.2%	72.20%
Claude 3 Sonnet	0.2112	0.9261	21.46%	78.16%
GPT-4o	0.7103	0.7107	0%	2%
Gemini 1.0 Pro	0.6338	0.6830	0%	7.2%
Claude 3 Sonnet	0.7743	0.7104	0%	9.92%
GPT-4o	0.7103	0.7107	0%	2%
Gemini 1.0 Pro	0.6338	0.6830	0%	7.2%
Claude 3 Sonnet	0.7743	0.7104	0%	9.92%

Table 3.2: Results for the out-of-distribution evaluation

in- and out-of-distribution datasets, as well as between the normal and Grid-Game versions of Tic-Tac-Toe.

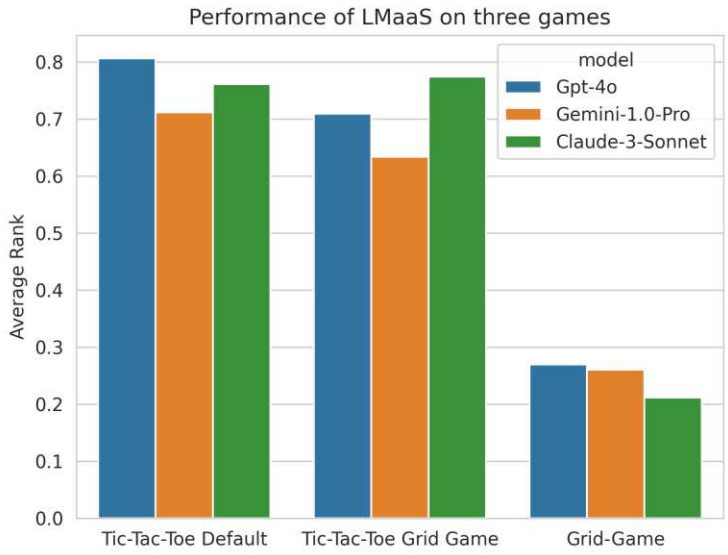

Figure 3.10: Performance comparison of three LMaaS the proposed datasets.

I can see that there is indeed a large distribution shift between the out-of-distribution dataset, "Grid-Games", and the two in-distribution datasets. For the two in-distribution games, the performance is roughly equal, with "Tic-Tac-Toe Grid Game" performing slightly worse for all models, except Claude 3 Sonnet. I attribute the slight performance difference to the representation form of our framework for the grid games, which elaborates the game mechanics in great detail and does not use the common X's and O's for the board pieces. For a comparison between the prompts, I refer to Examples B.

CHAPTER 4

Limitations & Future Work

I plan to conduct more experiments comparing known games to new and unknown games generated by our framework, as I did for "Tic-Tac-Toe". A further goal is to analyze individual games generated with our framework that, as well as LLM performance on it, to go beyond the performance I obtained on average, over the entire "Grid Games".

Furthermore, I would like to extend our framework by adding more functionality and control to allow users control testing in a more fine-grained manner. Lastly, I would like to conduct more experiments using variations of prompt styles, prompting strategies, and in-context learning, which was within the time and resource budget for this work. I summarize these below:

- **Improve game filtering**: With our current uniform random sampling, I often sample games with only a short or very pre-determined sequence of play. To allow users of our framework to generate more "playable" games, I would like to add a more elaborate filtering system to the space of games imposed by our mathematical framework from Section 3.2.1. Our current system solely filters out games that are "unplayable", which means they have no valid moves or immediately result in a game-ending state.
- **2D Grid Representation**: Our current implementations depend on the ability of LMaaS to read and understand the game state represented as a 2D grid. For future work, I would also like to experiment with adding different types of representations for the game state, such as textual description, cell-by-cell description, or other methods to see how they would affect performance.
- **Prompting style & Strategies**: In current literature (see Section 2), there are a lot of proposed prompting styles and strategies that are shown to increase reasoning performance. For future work, I would like to include them in our framework, to allow users to experiment with multiple approaches to LLM reasoning.

CHAPTER 5

Conclusion

Based on an extensive literature review from Section 2 and Appendix A, I introduced, to our knowledge, the first fully automatic generation and evaluation framework for testing the reasoning capabilities of black-box LMaaS, such as GPT-4o, Claude 3 Sonnet, and Google Gemini 1.0 Pro. The (typically new, and thus unknown) games generated by our approach are supported by a mathematical framework for two-dimensional games played on a grid, which I refer to as "2D Grid Games", from which games can easily be sampled using statistical distributions over each component space. To calibrate game difficulty, I introduced a complexity measure based on "Intrinsic Task Difficulty" by [HO17].

Generating games not in the training distribution of LMaaS solves a problem with the current reasoning benchmarks: The benchmarks are usually included in the training data, as they are offered publicly. Thus, model training can memorize and optimize for these types of benchmarks, rendering the benchmarks, once released, invalid for further evaluation of reasoning capabilities. Our generative approach allows a clearer separation of LLMs' memorization capabilities from their reasoning capabilities.

I note that [YKG+23] have presented, among other findings, evidence that LLMs have reasoning abilities that go beyond merely being a "stochastic parrot". Their argument has a non-constructive character, by appealing to a pigeon-hole principle argument to show that there must be reasoning abilities an LLMs has, that were not covered by training data – but the question is left unanswered what the specific reasoning performance of an LLM is, and how LLMs compare against each other. Our approach closes this gap by allowing concrete assessment of specific LLMs.

I believe our generative dataset approach has a certain degree of immunity against its games being scraped and incorporated into model training data, thus rendering it invalid: Since the generated games are diverse and non-repeating, and individual games can possess a level of complexity sufficiently large that a lot of effort needs to be expended to train a model to a degree that allows it to master that game well [RDM+24], it will

not be of use to train on a selection of our generated dataset, as they do not cover play patterns needed to help the models improve.

The framework for this thesis was implemented in Python and is largely split into one module for the grid game generation, and one that handles the three parts of the generation pipeline, see Section 3.4. The software was designed to be future-proof, anticipating further extensions, to easy extension to new Prompters, Solvers, and Games. The codebase spans approximately 4600 lines of code and was a large effort during the thesis.

Our experiments running this framework show, that, as expected, there is a clear distribution shift between the proposed in- and out-of-distribution games I test. Furthermore, the LMaaS seem to be able to leverage their learned knowledge about "Tic-Tac-Toe" in both representation forms.

Finally, our research clearly highlights the need for more comprehensive generative, out-of-distribution benchmarks, which developers of LLMs or other textual AI models, as well as end-users, can apply to accurately monitor reasoning performance without the risk of incorrect performance assessment due to test sets leaking into training data.

APPENDIX A

Comparison To Similar Work

The following listed papers presented some overlap with our approach. The discussion below summarizes the essential content of each article and highlights in bullet points the differences and similarities to our setup.

A.1 Comparison to [CWF+22]

Summary: In [CWF+22] the authors propose a planning and reasoning benchmark for out-of-distribution tasks. The novel dataset is generated by collecting human annotations to several planning tasks (for example, "Clean the dirty dishes") and reasoning tasks (providing explanations to real-world scenarios and their outcomes). The authors create prompts with three levels of difficulty by adding the most common answers from the human annotators as constraints to the base prompts. They also propose OOD prompts by adding constraints designed to force humans and LLMs to think "out of the box". They evaluate their results in two stages, once against a plain LLM (GPT-3) and once against a combination of an LLM (GPT-Neo) and a symbolic planning module. Their results are screened by human annotators again to filter out any degenerate generations.

- One of our main goals is to create a dynamic, generative dataset for formal games without requiring any human annotations. In [CWF+22] the authors make use of extensive human annotations for their experiments, both during benchmark generation, as well as in the screening of LLM responses.

- They focus on more planning-oriented tasks, like asking an LLM to propose ways to "wash the dishes", which does not have a clearly defined, unique solution. I solely focus on the domain of formal games, where a solution to a problem can be clearly defined.

- I aim to test the reasoning capabilities on games the LLM has not been trained on and has memorized a method to generate a solution. In Part I of their experiments, [CWF+22] focus on forcing LLMs to make use of OOD language via constraints added to their prompts. In Part II of their paper, they introduce a more constrained planning domain, aimed at being solvable using a planning solver in the PDDL language, which has more in common with our approach.

A.2 Comparison to [VOSK23]

Summary: In [VOSK23] the authors propose an extensible framework that automatically generates random problems for a given planning/reasoning domain, which can be used to evaluate the capabilities of LLMs. Their assessment architecture consists of a domain-dependent component and a domain-independent component, each responsible for different parts of the problem generation. The domain-dependent component holds a problem generator, which uses a human-annotated domain model (e.g., a description of what actions objects in the domain can take and how they are constrained. A translator module then converts the problem generated from natural language to PDDL. Receiving this PDDL as input, a planner and a planner validator, can verify the solutions generated by the LLM using test cases. The authors evaluate their work solely on the simple Blocksworld domain, which consists of colored blocks that have to be stacked in a certain order under some constraints. The models they evaluate are GPT-3 and BLOOM. Finally, they also include a human baseline from a preliminary study on Blocksworld as a reference for their benchmark.

- In [VOSK23] a benchmarking framework is developed that, as claimed, can be extended to other domains. It is however limited to domains that can easily be described using the PDDL language, as this is required by the domain-dependent components of their architecture. This limits the use cases of their work to domains that consist of objects that have relations to each other (as is common for planning problems). Furthermore, they also only test their work on the very simple Blocksworld domain. Problems therefore consist only of random arrangements of colored blocks on top of each other. I aim to provide a framework for formal games, that are not limited to the planning domain of problems defined by objects and their relations.

- I aim to provide a generative dataset that can be used to evaluate the OOD reasoning capabilities of a model. [VOSK23] solely use their framework to automatically evaluate LLMs on generated problems in the Blocksworld domain, without any generalization testing.

- Finally, to test OOD reasoning I want to fine-tune small models using samples of our generative dataset. [VOSK23] rely on baseline LLMs without any fine-tuning on samples generated by their proposed architecture.

A.3 Comparison to [CRW+23]

Summary: In [CRW+23], the GLAM method for functional grounding for LLMs is proposed. The authors to use methods from online RL to fine-tune a Flan-T5 instance with Proximal Policy Optimization (PPO) on a textual version of the BabyAI environment. Their environment is a version of GridWorld, where agents have to pick up or navigate to objects that are placed in it while only observing parts of the grid. To enable PPO in their environment, the authors choose to add an MLP on top of their Encoder blocks, which learns to approximate the Value function during training. Doing so severely increases the computational complexity due to the action space determining the size of the MLP, which in turn hinders experiments on larger state-of-the-art LLMs. Using their proposed GLAM method, the authors aim to answer four questions in their work based on sample efficiency, generalization to new tasks and objects, and the difference in using Behaviour Cloning (BC).

- They finetune their model by adding a separate value head to estimate a Value function (from RL, predicting the Value of a current state given the goal), which they train using Proximal Policy Optimization (PPO). Thus, their work is limited to environments that can be formalized as a partially observable MDP (PMDP) (environment needs to define rewards, state, transitions, etc.) which our approach does not require. Using an RL environment also comes with the task of defining a reward function, which is hard in itself.
- I aim to provide a rigid solution for each problem generated by our approach, where each step the LLM takes can be verified and assessed by our framework. As with all RL problems, [CRW+23] are interested in verifying the agent having reached the goal within a given timeframe to validate success, guiding it only by receiving intermediate rewards for each action.
- Experiments are constrained to one small LLM (FLAN-T5) due to the nature of their action space setup increasing computational complexity severely. I are interested in experimenting with multiple large state-of-the-art language models like GPT-4, Claude, and others.

A.4 Comparison to [PCOT23]

Summary: In [PCOT23] a programming-puzzle generation system is introduced that is called *Autotelic Code Exploration via Semantic Descriptors* (ACES). The authors contribute a novel approach to diversity maximization in programming puzzle generation by utilizing a specific version of GPT-3.5-Turbo as the generator, solver, and labeler of new puzzles. They propose the use of semantic descriptors (10-dimensional encoding of classical areas of programming puzzles) as a goal for the few-shot in-context learning LLM tasked to generate new pairs of puzzles and solutions. As the base for their work they rely on a collection of programming puzzles, P3, introduced by [SKPK21]. To

verify their approach the authors conduct multiple experiments and ablation studies, comparing ACES to other approaches that aim to maximize diversity in programming puzzle generation.

- I focus on the domain of formal games, where [PCOT23] only evaluates ACES on programming puzzles with the same structure as those in the P3 dataset (e.g., problems have to be defined as two functions f and g).
- Our approach aims to analytically analyze the diversity of generated games, whereas they focus on the distribution inside the matrix generated by semantic descriptors.
- In [PCOT23] only one version of ChatGPT is used for their studies, whereas I want to explore the OOD reasoning capabilities of multiple LLMs, such as GPT-4o, Claude, and others.
- The main goals of our works differ: [PCOT23] propose a diversity-maximization algorithm for programming puzzles, whereas I are interested in creating an OOD reasoning benchmark for LLMs in the domain of formal games.

A.5 Comparison to [LSS+24]

Summary: The authors of [LSS+24] introduce a Transformer architecture and training regime that can produce optimal search paths, using next token prediction, for various configurations of the grid games Maze and Sokoban. To train an initial Transformer encoder-decoder model, the authors produce a training and test set of search paths and execution traces from an A* search algorithm. To allow for seamless next token prediction, a novel prompt format for the search process of A* is introduced. Using the training set, the authors train two initial models: one using solely the task and the optimal search path (solution-only) and another using the task, execution trace, and optimal search path (search-augmented). To further improve the search performed by the search-augmented model, they use non-deterministic A* samples generated by its decoder module to create a second training set. A third model, which the authors call "SearchFormer", is then trained to optimize for the shortest optimal search path. The results indicate that the SearchFormer outperforms both the search-augmented and solution-only models on the test sets for both games.

- The authors of [LSS+24] aim to provide a solution for each instance of their games. They train their Transformers to provide an optimal search path, even improving upon A* when experimenting with their SearchFormer. I, on the other hand, do not seek to provide optimal solutions to the games I generate. Our goal is to provide a heuristic that can be used to judge the responses made by LLMs.
- Our focus lies in generating a vast number of OOD games to evaluate the reasoning capabilities of LLMs. [LSS+24] only aim to demonstrate that Transformers can solve planning tasks, for which they rely on two well-known games.

APPENDIX B

Examples of LLM Gameplay

Gameplay example of a Grid-Game with Gemini 1.0 Pro

System

You are a helpful assistant, tasked with playing games that require reasoning.

This game is played on a 3x4 grid.
Two players take turns moving their game pieces on the grid.
The game pieces on the grid are represented by (Type,Player),as example (1,0) for a game piece of type 1 belonging to player 0.

The following is a description of the basic rules that other rules are constructed from:
up: moves a piece up one row
down: moves a piece down one row
left: moves a piece left one column
right: moves a piece right one column
nw: moves a piece up one row and left one column
sw: moves a piece down one row and left one column
ne: moves a piece up one row and right one column
se: move a piece down one row and right one column
Moves with a '_c' appended capture another piece in its target location.
Moves can be combined with arrows '->', as example 'up->down'.
Each type of game piece can only be moved using a subset of rules.
In the following, I list the rules for each type of game piece.
To use a rule on a game piece, it has to satisfy the CONDITION:

RULES for Type 1:
1.: left_c->down_c->se_c->nw->se_c;CONDITION: None
2.: sw_c->up_c->left;CONDITION: a piece at row 0 and column 2 belonging to any player
RULES for Type 2:
3.: left_c->down_c->se_c->nw->se_c;CONDITION: None
RULES for Type 3:
4.: ne->sw_c->se_c;CONDITION: OR(OR(a piece at any row and column 1 belonging to player 1 , OR(a piece at row 0 and any column belonging to player 0 , a piece at row 2 and any column belonging to any player)) , OR(a piece at row 1 and any column belonging to player 0 , NOT(a piece at row 0 and column 3 belonging to any player)))
5.: nw->left_c->nw_c->ne_c;CONDITION: OR(NOT(OR(a piece at row 0 and column 1 belonging to any player , a piece at row 0 and any column belonging to player 0)) , NOT(NOT(a piece at row 1 and any column belonging to player 1)))
6.: left_c->ne->ne->up->right;CONDITION: OR(OR(OR(a piece at row 1 and any column belonging to player 0 , a piece at row 2 and column 2 belonging to player 1) , OR(a piece at row 2 and any column belonging to any player , a piece at any row and column 0 belonging to player 0)) , OR(OR(a piece at row 2 and column 0 belonging to player 1 , a piece at any row and column 2 belonging to player 1) , OR(a piece at any row and column 2 belonging to player 1 , a piece at any row and column 3 belonging to player 0)))

WIN CONDITIONS:
The following are the win conditions that a player must reach to win the game:
1.: OR(a piece at any row and any column with type 1 , a piece at row 0 and column 2 with type 3)

LOSS CONDITIONS:
Players will lose the game by satisfying the following conditions :
1.: None

The following is a description of a game:
You are playing as 0
The current board is:
3x4,Types: [1, 2, 3]
(3,1) | (-,-) | (-,-) | (-,-)
(-,-) | (3,0) | (2,0) | (-,-)
(3,1) | (2,0) | (2,1) | (2,1)

You are playing as 0.
You have the following pieces on the board:
0.: (1,1) with type 3
6.: (1,2) with type 2
7.: (2,1) with type 2
You must choose a movement rule and a game piece from the list above to move a game piece on the board.
Reply only in the following format by choosing a number that represents the index of the rule from above
and a game piece in the form of (row,column):
ACTION: $< 1, 2, 3, 4, 5, 6 >$,(row,column)
Reply with no additional text!
row is in the range of 0,1,2 and column is in the range of 0,1,2,3:
Take great care to reply exactly in the answer format shown above!

Gemini 1.0 Pro

ACTION: $< 4 >$,(0,1)

Gameplay example of Tic-Tac-Toe-Grid with Gemini 1.0 Pro

System

You are a helpful assistant, tasked with playing games that require reasoning.

This game is played on a 3x3 grid.
Two players take turns moving their game pieces on the grid.
The game pieces on the grid are represented by (Type,Player),as example (1,0) for a game piece of type 1 belonging to player 0.

The following is a description of the basic rules that other rules are constructed from:
up: moves a piece up one row
down: moves a piece down one row
left: moves a piece left one column
right: moves a piece right one column
nw: moves a piece up one row and left one column
sw: moves a piece down one row and left one column
ne: moves a piece up one row and right one column
se: move a piece down one row and right one column
place: places a piece at an EMPTY cell at row and column (only if there is not a

piece there already!)
Moves with a '_c' appended capture another piece in its target location.
Moves can be combined with arrows '- ', as example 'up- down'.
Each type of game piece can only be moved using a subset of rules.
In the following, I list the rules for each type of game piece.
To use a rule on a game piece, it has to satisfy the CONDITION:
RULES for Type 1:
1.: place;CONDITION: None

WIN CONDITIONS:
The following are the win conditions that a player must reach to win the game:
1.: AND(a piece at row 0 and column 0 with type 1 , AND(a piece at row 0 and column 1 with type 1 , a piece at row 0 and column 2 with type 1))
2.: AND(a piece at row 1 and column 0 with type 1 , AND(a piece at row 1 and column 1 with type 1 , a piece at row 1 and column 2 with type 1))
3.: AND(a piece at row 2 and column 0 with type 1 , AND(a piece at row 2 and column 1 with type 1 , a piece at row 2 and column 2 with type 1))
4.: AND(a piece at row 0 and column 0 with type 1 , AND(a piece at row 1 and column 0 with type 1 , a piece at row 2 and column 0 with type 1))
5.: AND(a piece at row 0 and column 1 with type 1 , AND(a piece at row 1 and column 1 with type 1 , a piece at row 2 and column 1 with type 1))
6.: AND(a piece at row 0 and column 2 with type 1 , AND(a piece at row 1 and column 2 with type 1 , a piece at row 2 and column 2 with type 1))
7.: AND(a piece at row 0 and column 0 with type 1 , AND(a piece at row 1 and column 1 with type 1 , a piece at row 2 and column 2 with type 1))
8.: AND(a piece at row 0 and column 2 with type 1 , AND(a piece at row 1 and column 1 with type 1 , a piece at row 0 and column 2 with type 1))

LOSS CONDITIONS:
Players will lose the game by satisfying the following conditions :

The following is a description of a game:
You are playing as 0
The current board is:
3x3,Types: [1]
(-,-) | (-,-) | (-,-)
(-,-) | (-,-) | (-,-)
(-,-) | (-,-) | (-,-)

You are playing as 0.
You have the following pieces on the board:

You must choose a movement rule and a game piece from the list above to move a game piece on the board, or choose an empty cell if placing a piece.
Reply only in the following format by choosing a number that represents the index of the rule from above
and a game piece or empty cell in the form of (row,column):
ACTION: < 1 >,(row,column)
Reply with no additional text!
row is in the range of 0,1,2 and column is in the range of 0,1,2:
Take great care to reply exactly in the answer format shown above!

Gemini 1.0 Pro

Action: 1,(0,0)

Gameplay example of Tic-Tac-Toe with Gemini 1.0 Pro

System

You are a helpful assistant, tasked with playing games that require reasoning.

User

Players take turns placing symbols on a 3x3 grid.
The starting players uses 'X', the other 'O' to mark their fields.
It is not allowed to place symbols on fields that already have one assigned to them.
The first player to have three of their symbols next to each other wins the game (can be diagonal or straight lines).

The following is a description of a game:
You are playing as X
X | O | X
| |
X | | O

—

You must place the next symbol on the board.
The cells of the board are labeled from 0 to 8, starting in the top left corner.
Reply using only one of the following numbers: 0,1,2,3,4,5,6,7,8
Reply with exactly one character that is the chosen number in the following format: ACTION:<1,etc.,8>
Make sure that you don't include any additional text!

Gemini 1.0 Pro

ACTION: 4

APPENDIX C

A Full Example of Game Generation and Gameplay

C.1 Creating a Game Deterministically

The Grid

I begin by defining a positive integer n and a positive integer k. For this example assume that $n = 6$ and $k = 8$. Next, I create an empty $n \times k = 6 \times 8$ matrix that represents our grid, as shown in Figure C.1.

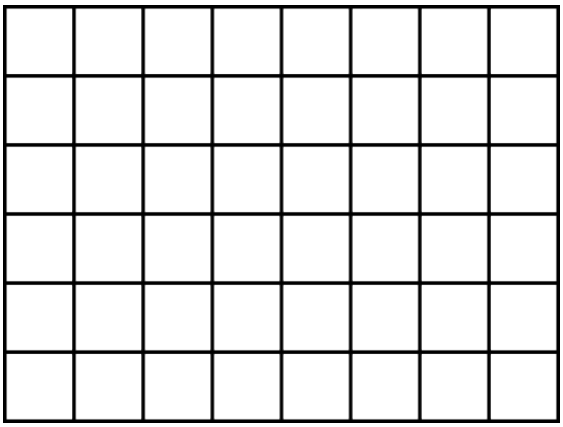

Figure C.1: Initial empty grid

Creating Types

Next, I create a number n_t to define the amount of types of game pieces. I assume $n_t = 3$ for this example, which means that our available types are $\{1, 2, 3\}$.

In the following, I are going to represent game pieces of type 1 by a circle, type 2 by a square, and type 3 by a star. Player 0's pieces will be colored blue, while player 1's pieces will be orange. An illustration of this can be seen in Figure C.2.

Figure C.2: Symbols used for different game-pieces

Creating Rules

As the next step, I create n_r-many (rule, condition) pairs from the space of $\mathrm{MCR}(c, j)$. For this example, let us assume $n_r = 6$. To make the example easier to follow, I restrict ourselves to rules with rule-complexity of $c \in \{1, 2\}$ and condition-complexity of $j \in \{0, 1\}$. Using these parameters, I define r_1 to r_8:

$$
\begin{aligned}
r_1 &= (\mathrm{up}, \varnothing),\\
r_2 &= (\mathrm{down}, \varnothing),\\
r_3 &= (\mathrm{exchange}(\mathrm{up}, 3), \mathrm{eq}_{1,-1,1}),\\
r_4 &= (\mathrm{exchange}(\mathrm{down}, 3), \mathrm{eq}_{5,-1,0}),
\end{aligned}
$$

$$
\begin{aligned}
r_5 &= (\mathrm{left}, \varnothing),\\
r_6 &= (\mathrm{right}, \varnothing),\\
r_7 &= (\mathrm{left}_C, \varnothing),\\
r_8 &= (\mathrm{right}_C, \varnothing),
\end{aligned}
$$

The conditions on r_3 and r_4 allow each player to only use those rules for game pieces in either the 5th row (for player 0) or the 1st row (for player 1). The conditions are illustrated in Figure C.3.

Assigning Rules to Types

To assign rules to each of the piece types, I will now define the function $\mathcal{A}$. I do this by assigning a subset of rules to each type. Assume now that $\mathcal{A}$ is the following:

$$
\mathcal{A}(t) \coloneqq (1 \mapsto \{r_1, r_2, r_3, r_4\}, 2 \mapsto \{r_5, r_6, r_7, r_8\}, 3 \mapsto \varnothing)\,.
$$

Game pieces of type 1 can now move up and down. They also have a special move that allows them to exchange the piece for one of type 3 after moving up or down from the first to the last row, depending on the player. Pieces of type 2 can move left and right and can capture pieces that belong to the opponent in those directions.

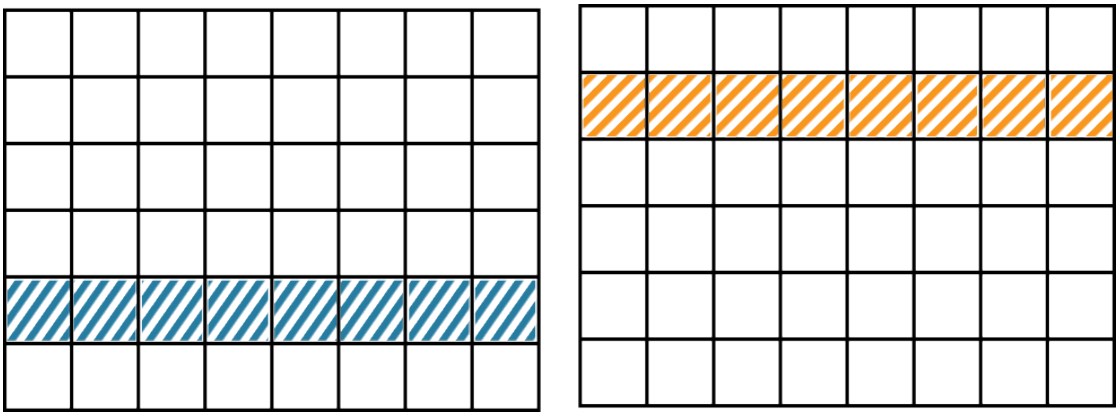

Figure C.3: Conditions for r_3 and r_4

Creating Win Conditions

To create a goal for our game, I will now define n_{wp} win conditions. For this example, I assume $n_{\text{wp}} = 1$. I define the following win condition w_1 using a complexity of $j = 2$ from the space of WP(2):

$$w_1 = \text{or}(\text{hp}(0, -1, 3, -1), \text{hp}(6, -1, 3, -1)).$$

The win condition covers the first row and last row of the grid. Each player can win by reaching it with one piece of type 3 (which can only be achieved by using r_3 and r_4). An illustration of win condition w_1 is displayed in Figure C.4.

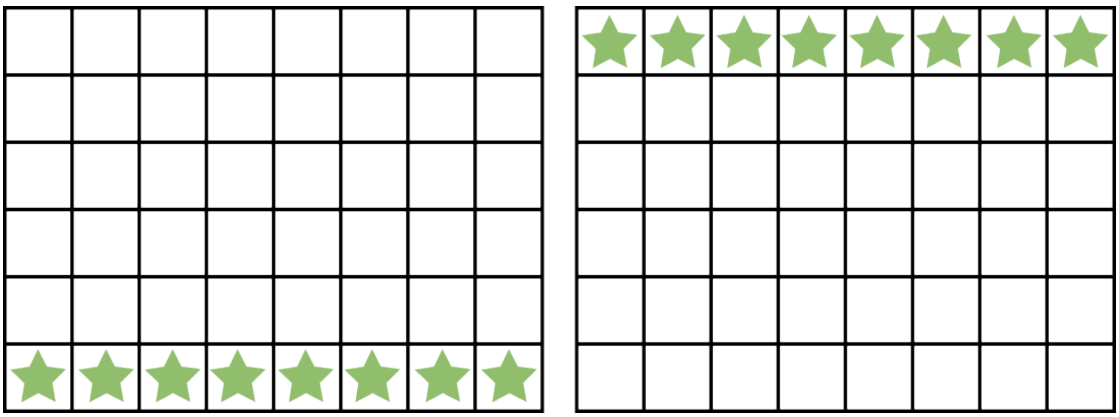

Figure C.4: Positional Win-Condition w_1

Creating the Initial Board State

I now proceed to create an initial board state, consisting of n_{p} game pieces for each player. I assume $n_{\text{p}} = 5$, thus creating 5 pieces for each player. I define the following five game pieces for both players, creating $\mathcal{P}$:

$$\mathcal{P} = \{(6,1,1,0),(6,3,1,0),(6,5,1,0),(6,7,1,0),(5,4,2,0)\} \cup \{(1,2,1,1),(1,4,1,1),(1,6,1,1),(1,8,1,1),(2,5,2,1)\}.$$

The new pieces are now placed on the grid, creating the starting position of the game, as shown in Figure C.5.

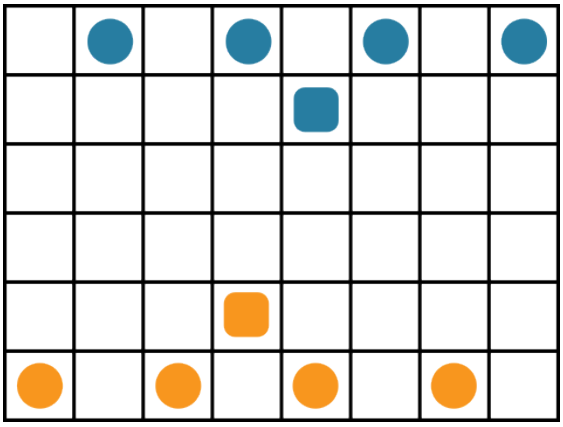

Figure C.5: Initial state of the game

Creating a Loss Condition

For the loss condition, I will rely on the default contents of $\mathcal{L}$, which contains both of the all conditions:

$$\mathcal{L} = \{\text{all}(\mathit{Left}), \text{all}(\mathit{Right})\}.$$

The loss condition makes each player lose the game if they have no more moves available for any of their pieces.

C.2 Recreating the Game by Sampling

I will now reconstruct the game from the previous Section C.1. However, I will rely on the probability distributions introduced in Section 3.2.2 to sample the game's components, instead of choosing them directly.

Sampling the Grid

I define the distribution $P(\mathbf{N} \cap \mathbf{K})$ as deterministic. Sampling from this distribution, I receive the following:

$$P(\mathbf{N} = 6 \cap \mathbf{K} = 8) = 1,$$
$$(6,8) \sim P(\mathbf{N} \cap \mathbf{K}).$$

As in the previous section, our grid is now a $n = 6 \times k = 8$ matrix.

Sampling Types

To sample the types used in game created in Section C.1, I construct a deterministic distribution for $P(\mathbf{T})$ and sample from it to create $\mathcal{T}$:

$$P(\mathbf{T} = [3]) = 1,$$
$$\mathcal{T} = [3] \sim P(\mathbf{T}).$$

Sampling Rules & Conditions

Recreating the rules and conditions will require a bit more effort. Before going over the distributions, it is important to note that $\mathcal{R}$ is a set, thus it does not contain any duplicates. As the first step, I need to define the distribution of the parameters:

$$P(\mathbf{C} = 1 \cap \mathbf{J} = 0) = 0.5,$$
$$P(\mathbf{C} = 2 \cap \mathbf{J} = 1) = 0.5.$$

I continue by defining the distribution for $P(\mathbf{R}_1 \cap \mathbf{B}_0)$:

$$P(\mathbf{R}_1 = \text{up} \cap \mathbf{B}_0 = \varnothing) = 0.1\overline{6},$$
$$P(\mathbf{R}_1 = \text{down} \cap \mathbf{B}_0 = \varnothing) = 0.1\overline{6},$$
$$P(\mathbf{R}_1 = \text{left} \cap \mathbf{B}_0 = \varnothing) = 0.1\overline{6},$$
$$P(\mathbf{R}_1 = \text{right} \cap \mathbf{B}_0 = \varnothing) = 0.1\overline{6},$$
$$P(\mathbf{R}_1 = \text{left}_\text{C} \cap \mathbf{B}_0 = \varnothing) = 0.1\overline{6},$$
$$P(\mathbf{R}_1 = \text{right}_\text{C} \cap \mathbf{B}_0 = \varnothing) = 0.1\overline{6}.$$

I now also define the distribution for $P(\mathbf{R}_2 \cap \mathbf{B}_1)$:

$$P(\mathbf{R}_2 = \text{exchange}(\text{up}, 3) \cap \mathbf{B}_1 = \text{eq}_{1,-1,1}) = 0.5,$$
$$P(\mathbf{R}_2 = \text{exchange}(\text{down}, 3) \cap \mathbf{B}_1 = \text{eq}_{5,-1,0}) = 0.5.$$

As before, remember that the rule and condition pairs that are put into the set $\mathcal{R}$ must be unique! Therefore, if I sample values of the fixed parameters c, j from $P_{C,J}$ and rule & condition pairs from $P(\mathbf{R} \cap \mathbf{B})$ until $|\mathcal{R}| = 8$, I receive the same rule and conditions as in Section C.1. For the readers' sake, I have omitted the sampling of all 8 rules here. They can be seen in the previous section. I show the sampling of one of the rules as an example:

$$(1, 0) \sim P(\mathbf{C} \cap \mathbf{J}),$$
$$(\text{up}, \varnothing) \sim P(\mathbf{R}_1 \cap \mathbf{B}_0).$$

Sampling Actions

To recreate the assignments made by the $\mathcal{A}$ function, I will first define the probability distribution $P(\mathbf{R} \mid \mathbf{T} = t)$ as deterministic:

$$P(\mathbf{R} = \{r_1, r_2, r_3, r_4\} \mid \mathbf{T} = 1) = 1,$$
$$P(\mathbf{R} = \{r_5, r_6, r_7, r_8\} \mid \mathbf{T} = 2) = 1,$$
$$P(\mathbf{R} = \varnothing \mid \mathbf{T} = 3) = 1.$$

With the now deterministic distribution, I can create the helper tuple $\mathcal{H}$ by sampling the three elements:

$$\mathcal{H} = (\{r_1, r_2, r_3, r_4\} \sim P(\mathbf{R} \mid \mathbf{T} = 1), \{r_5, r_6, r_7, r_8\} \sim P(\mathbf{R} \mid \mathbf{T} = 2), \varnothing \sim P(\mathbf{R} \mid \mathbf{T} = 3)) .$$

As I have sampled $\mathcal{H}$, the function $\mathcal{A}$ is now defined as follows:

$$\mathcal{A}(t) \coloneqq (1 \mapsto \{r_1, r_2, r_3, r_4\}, 2 \mapsto \{r_5, r_6, r_7, r_8\}, 3 \mapsto \varnothing) .$$

The replicates the $\mathcal{A}$ function that was created in Section C.1.

Sampling Win Conditions

To recreate the win conditions, I begin by defining the distributions for $P(\mathbf{J})$ and $P(\mathbf{W}_j)$ as:

$$2P(\mathbf{J} = 2) = 1$$
$$f \coloneqq \mathrm{or}(\mathrm{hp}(0, -1, 3, -1), \mathrm{hp}(6, -1, 3, -1))$$
$$P(\mathbf{W}_2 = f) = 1$$

By first sampling a parameter j from $P(\mathbf{J})$ and then sampling a win condition from $P(\mathbf{W}_j)$, I receive the same win condition as introduced in Section C.1:

$$2 \sim P(\mathbf{J}),$$
$$\mathrm{or}(\mathrm{hp}(0, -1, 3, -1), \mathrm{hp}(6, -1, 3, -1)) \sim P(\mathbf{W}_j).$$

Sampling Loss Conditions

As the game in Section C.1 example relied on the default set of loss conditions, I can omit this section and also use the default contents of $\mathcal{L}$.

Sampling the Initial Board State

To finalize our game, I will also sample the same set of initial game pieces as in Section C.1. To achieve this, first define the distribution $P(\mathbf{P})$ as follows:

$$
\begin{aligned}
P(\mathbf{P} = (6,1,1,1)) &= 0.1, & P(\mathbf{P} = (1,2,1,0)) &= 0.1,\\
P(\mathbf{P} = (6,3,1,1)) &= 0.1, & P(\mathbf{P} = (1,4,1,0)) &= 0.1,\\
P(\mathbf{P} = (6,5,1,1)) &= 0.1, & P(\mathbf{P} = (1,6,1,0)) &= 0.1,\\
P(\mathbf{P} = (6,7,1,1)) &= 0.1, & P(\mathbf{P} = (1,8,1,0)) &= 0.1,\\
P(\mathbf{P} = (5,4,2,1)) &= 0.1, & P(\mathbf{P} = (2,5,2,0)) &= 0.1.
\end{aligned}
$$

Sampling from this distribution until the size of the initial set of game pieces $|\mathcal{P}| = 10$, I recreate the same set as in the previous section. I only show one of the sampled pieces here as an example:

$$(6,1,1,0) \sim P(\mathbf{P}).$$

C.3 Gameplay

Below I display players taking turns of the game introduced in the previous sections of this chapter. I provide a visual representation of the board state as needed and aggregate turns to avoid unnecessary length.

Turn 1

The game begins in state s_0, which is depicted in Figure C.5, with *Left* choosing a valid action from the set A_{Left}. Their choice lands on the tuple $((1,8,1,0), r_2)$ as action a_1. This would move the chosen piece down. By applying the transition function, the game piece is moved and the board state $T(s_0, a_1) = s_1$ is created. The board at the end of Turn 1 is shown in Figure C.6.

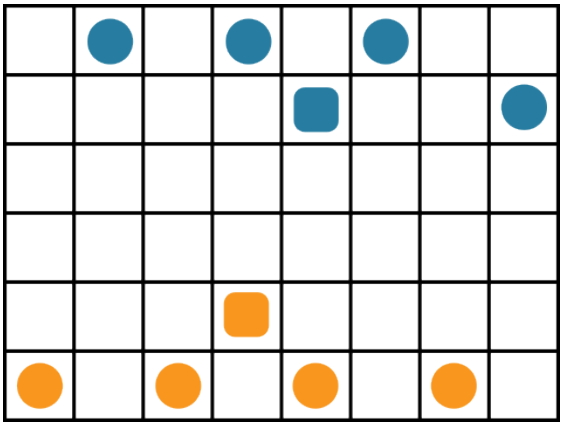

Figure C.6: Board state s_0 in Turn 1

Before moving on the win and loss conditions of the game are checked. To avoid repeating the same text multiple times, I will omit writing out this detail until an end state is reached.

Turn 2

Now in board state s_1, it's *Right*'s turn to choose an action from the set A_{Right}. Their choice lands on the tuple $((5, 4, 2, 1), r_6)$ as a_2. This action would move their "goalkeeper" piece to the right. Again, by applying the transition function the piece is moved and state $T(s_1, a_2) = s_2$ is created. The board state is shown in Figure C.7.

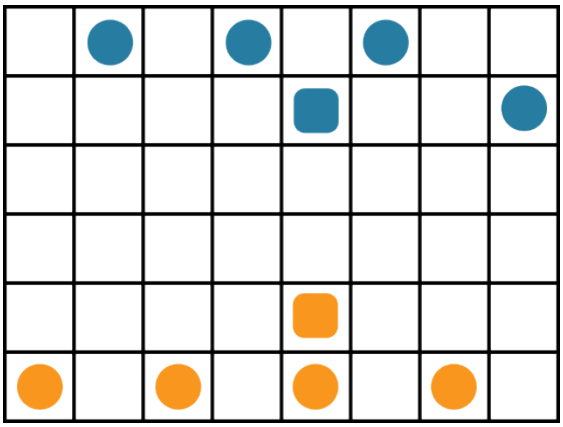

Figure C.7: Board state s_2 in Turn 2

Turn 3 - 10

To keep the example short, I aggregate the moves both players take in turns 3 to 10. I also do not explicitly mention the selection of actions and the transition to new states. *Left* uses 3 of their turns to try and move his piece from Turn 1 into the win condition. *Right* uses 3 of their moves to counter this and capture the piece with r_8 before it enters the goal line. Both players use their other two turns to move the leftmost pieces up/down 2, creating the board state s_{10} shown in Figure C.8.

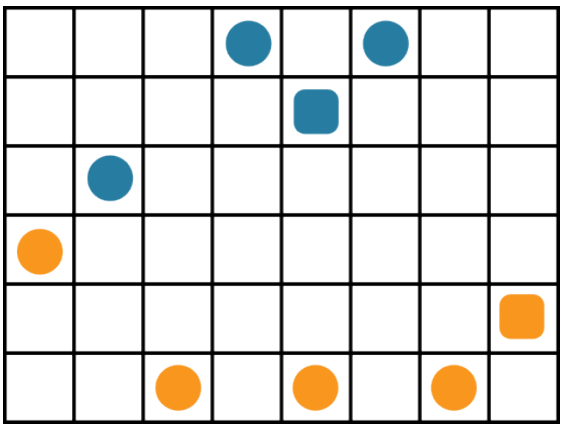

Figure C.8: Board state s_{10} in Turn 10

Turn 11 - 16

As is apparent from the board state s_{10} after Turn 10, *Left* can now proceed to move their leftmost piece down, and then finally use r_4 to move down into the "goal-line" and exchange their piece for one of type 3 in s_{16}. *Right* can't do much but match moving their leftmost piece up on each of their turns but has no more chance to win. The board state s_{16} is shown in Figure C.9.

As mentioned at the beginning of this section, I will now proceed to check the win conditions for state s_{16}. As this game only has one of them, w_1, this is quite easy. Before I begin, let us recall how w_1 was defined:

$$w_1 = \text{or}(\text{hp}(0, -1, 3, -1), \text{hp}(6, -1, 3, -1)).$$

The function hp evaluates to True if there is a piece in P that matches the parameters. Let us check it for *Left*'s pieces first. Their leftmost piece is at $(6, 2, 3, 0)$, which does not satisfy $\text{hp}(0, -1, 3, -1)$. However, it does satisfy $\text{hp}(6, -1, 3, -1)$, which causes the whole win condition to be satisfied. Thus, *Left* has won the game and a terminal state is reached!

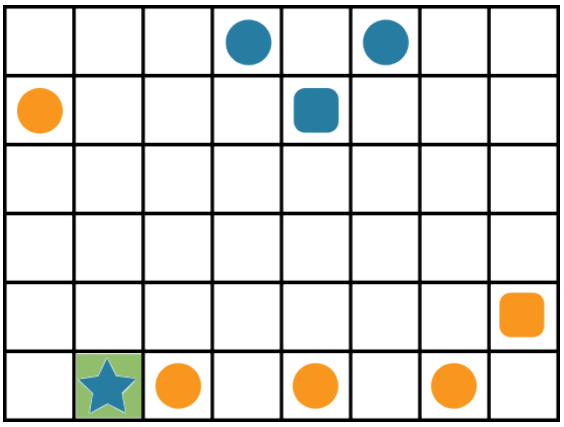

Figure C.9: End state s_{16} at Turn 16

Overview of Generative AI Tools Used

ChatGPT was used to connect the already written summaries of research papers from a text document during the literature review into paragraphs. No additional information was added by the AI Assistant and no research was done by it (e.g., finding papers, summarizing, etc.), it was solely used to connect the independent sentences in order of date. The following prompt was used to connect the sentences:

Prompt for connecting paper summaries for related work

You are a scientist working on a machine learning paper on Large Language Models. You are currently writing the 'Related Work' section for your paper focused on Automatic Evaluation for LLMs.
Using the following sources, write a short paragraph for the 'Related Work' section of your paper:
You are a scientist working on a machine learning paper on Large Language Models. You are currently writing the 'Related Work' section for your paper focused on Automatic Evaluation for LLMs.
Using the following sources, write a short paragraph for the 'Related Work' section of your paper:

[8. Jun 2023] "PandaLM: An Automatic Evaluation Benchmark for LLM Instruction Tuning Optimization" [arxiv-lite: 'automatic evaluation', p1-5 end of rel.res.] [autoeval] (Authors introduce a benchmark for automatic evaluation and introduce a finetuned LLama instance using a custom dataset designed for human-aligned evaluation as a cost-effective alternative to GPT-4)

...
[17. Oct 2023] "Exploring Automatic Evaluation Methods based on a Decoder-based LLM for Text Generation" [arxiv-lite: 'automatic evaluation', p1-5 end of rel.res.] [autoeval] (Authors introduce a number of Decoder-based LLMs finetuned on evaluation of machine translation and semantic similarity tasks and compare them on a number of benchmarks)

List of Figures

List of Tables

Bibliography

[AGS+23] Sahar Abdelnabi, Amr Gomaa, Sarath Sivaprasad, Lea Schönherr, and Mario Fritz. LLM-Deliberation: Evaluating LLMs with interactive multi-agent negotiation games, 2023.

[AI] Anthropic AI. Claude 3. https://docs.anthropic.com/en/docs/intro-to-claude. Retrieved: 2024-05-16.

[Ant] Antrophic AI. Claude 2.1. https://www.anthropic.com/index/claude-2. Retrieved: 2023-12-29.

[BCE+23] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with GPT-4, 2023.

[BCL+23] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. A multitask, multilingual, multimodal evaluation of ChatGPT on reasoning, hallucination, and interactivity, 2023.

[BGG21] Bart Bogaerts, Emilio Gamba, and Tias Guns. A framework for step-wise explaining how to solve constraint satisfaction problems. *Artificial Intelligence*, 300:103550, nov 2021.

[BGP+23] Qiming Bao, Gael Gendron, Alex Yuxuan Peng, Wanjun Zhong, Neset Tan, Yang Chen, Michael Witbrock, and Jiamou Liu. A systematic evaluation of large language models on out-of-distribution logical reasoning tasks, 2023.

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan et al. Language models are few-shot learners, 2020.

[BYC+23] Yushi Bai, Jiahao Ying, Yixin Cao, Xin Lv, Yuze He, Xiaozhi Wang, Jifan Yu, Kaisheng Zeng, Yijia Xiao, Haozhe Lyu, Jiayin Zhang, Juanzi Li, and Lei Hou. Benchmarking foundation models with language-model-as-an-examiner, 2023.

[CGT+20] Wojciech Marian Czarnecki, Gauthier Gidel, Brendan Tracey, Karl Tuyls, Shayegan Omidshafiei, David Balduzzi, and Max Jaderberg. Real world games look like spinning tops, 2020.

[CH23] Cheng-Han Chiang and Hung-yi Lee. Can large language models be an alternative to human evaluations?, 2023.

[Cho19] François Chollet. On the measure of intelligence, 2019.

[CL23] Cheng-Han Chiang and Hung-yi Lee. A closer look into automatic evaluation using large language models, 2023.

[CMS+23] Meiqi Chen, Yubo Ma, Kaitao Song, Yixin Cao, Yan Zhang, and Dongsheng Li. Learning to teach large language models logical reasoning, 2023.

[CPBD+23] Nitay Calderon, Naveh Porat, Eyal Ben-David, Zorik Gekhman, Nadav Oved, and Roi Reichart. Measuring the robustness of natural language processing models to domain shifts, 2023.

[CRW+23] Thomas Carta, Clément Romac, Thomas Wolf, Sylvain Lamprier, Olivier Sigaud, and Pierre-Yves Oudeyer. Grounding large language models in interactive environments with online reinforcement learning, 2023.

[CSZ+23] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors, 2023.

[CWF+22] Katherine M. Collins, Catherine Wong, Jiahai Feng, Megan Wei, and Joshua B. Tenenbaum. Structured, flexible, and robust: benchmarking and improving large language models towards more human-like behavior in out-of-distribution reasoning tasks, 2022.

[Dee] Google DeepMind. Gemini Pro 1.5. https://deepmind.google/technologies/gemini/pro/. Retrieved: 2024-05-16.

[DF23] Maksym Del and Mark Fishel. True detective: A deep abductive reasoning benchmark undoable for GPT-3 and challenging for GPT-4, 2023.

[dMAC] Alejandro de Miquel, Yuji Ariyasu, and Roderic Guigo Corominas. 2nd place solution. https://www.kaggle.com/competitions/abstraction-and-reasoning-challenge/discussion/154391. Retrieved: 2024-05-16.

[EGH+23] Joan Espasa, Ian P. Gent, Ruth Hoffmann, Christopher Jefferson, Alice M. Lynch, András Salamon, and Matthew J. McIlree. Using small MUSes to explain how to solve pen and paper puzzles, 2023.

[Elo78] Arpad E. Elo. *The Rating of Chess Players, Past and Present*. Arco Pub., New York, 1978.

[Fav18] Megan Fava. What is game theory? https://math.osu.edu/sites/math.osu.edu/files/What_is_2018_Game_Theory.pdf, 2018. Retrieved: 2024-02-07.

[FNJL23] Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. GPTScore: Evaluate as you desire, 2023.

[FPC+23] Simon Frieder, Luca Pinchetti, Alexis Chevalier, Ryan-Rhys Griffiths, Tommaso Salvatori, Thomas Lukasiewicz, Philipp Christian Petersen, and Julius Berner. Mathematical capabilities of ChatGPT, 2023.

[GBWD23] Gaël Gendron, Qiming Bao, Michael Witbrock, and Gillian Dobbie. Large language models are not strong abstract reasoners, 2023.

[GN21] Adrian Groza and Cristian Nitu. Natural language understanding for logical games, 2021.

[Gooa] Google. Google Bard. https://bard.google.com/. Retrieved: 2023-12-29.

[Goob] Google DeepMind. Google Gemini-AI. https://blog.google/technology/ai/google-gemini-ai/. Retrieved: 2023-12-29.

[Gro21] Adrian Groza. *Modelling Puzzles in First Order Logic*. 01 2021.

[Gro23] Adrian Groza. Measuring reasoning capabilities of ChatGPT, 2023.

[GSG23] Kanishk Gandhi, Dorsa Sadigh, and Noah D. Goodman. Strategic reasoning with language models, 2023.

[GYY+23] Jiaxian Guo, Bo Yang, Paul Yoo, Bill Yuchen Lin, Yusuke Iwasawa, and Yutaka Matsuo. Suspicion-agent: Playing imperfect information games with theory of mind aware gpt-4, 2023.

[HACY20] Matthew Hausknecht, Prithviraj Ammanabrolu, Marc-Alexandre Côté, and Xingdi Yuan. Interactive fiction games: A colossal adventure. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):7903–7910, Apr. 2020.

[HGM+23] 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, 2023.

[HMR23] Rishi Hazra, Pedro Zuidberg Dos Martires, and Luc De Raedt. SayCanPay: Heuristic planning with large language models using learnable domain knowledge, 2023.

[HO17] José Hernández-Orallo. *The Measure of All Minds: Evaluating Natural and Artificial Intelligence*. Cambridge University Press, 2017.

[Ice] Icecuber. 1st place solution + code and official documentation. https://www.kaggle.com/competitions/abstraction-and-reasoning-challenge/discussion/154597. Retrieved: 2024-05-16.

[IYL23] Adam Ishay, Zhun Yang, and Joohyung Lee. Leveraging large language models to generate answer set programs, 2023.

[JJL+23] Sumit Kumar Jha, Susmit Jha, Patrick Lincoln, Nathaniel D. Bastian, Alvaro Velasquez, Rickard Ewetz, and Sandeep Neema. Neuro symbolic reasoning for planning: Counterexample guided inductive synthesis using large language models and satisfiability solving, 2023.

[JS85] F. Forgó J. Szép. *Introduction to the Theory of Games*. Mathematics and its Applications. Springer Dordrecht, 1 edition, 1985.

[JvN04] Oskar Morgenstern John von Neumann. *Theory of Games and Economic Behavior*. Princeton Classic Editions. Princeton University Press, 60th anniv. edition, 2004.

[JWJ23] Ana Jojic, Zhen Wang, and Nebojsa Jojic. GPT is becoming a turing machine: Here are some ways to program it, 2023.

[KK23] Tomohito Kasahara and Daisuke Kawahara. Exploring automatic evaluation methods based on a decoder-based LLM for text generation, 2023.

[KKSR22] Saurabh Kulshreshtha, Olga Kovaleva, Namrata Shivagunde, and Anna Rumshisky. Down and across: Introducing Crossword-Solving as a new NLP benchmark, 2022.

[KT53] H.W. Kuhn and A.W. Tucker. *Contributions to the Theory of Games*. Annals of Mathematics Studies. Princeton University Press, 1953.

[LC23] Yen-Ting Lin and Yun-Nung Chen. LLM-Eval: Unified multi-dimensional automatic evaluation for open-domain conversations with large language models, 2023.

[LCL+20] Jian Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. LogiQA: A challenge dataset for machine reading comprehension with logical reasoning, 2020.

[LCSH23] Jonathan Light, Min Cai, Sheng Shen, and Ziniu Hu. AvalonBench: Evaluating LLMs playing the game of Avalon, 2023.

[LG] Ilia Larchenko and Vlad Golubev. 3rd place. https://www.kaggle.com/c/abstraction-and-reasoning-challenge/discussion/154305. Retrieved: 2024-05-16.

[LHZ+23] Zhihan Liu, Hao Hu, Shenao Zhang, Hongyi Guo, Shuqi Ke, Boyi Liu, and Zhaoran Wang. Reason for future, act for now: A principled framework for autonomous LLM agents with provable sample efficiency, 2023.

[LIX+23] Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-Eval: NLG evaluation using GPT-4 with better human alignment, 2023.

[LJZ+23] Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. LLM+P: Empowering large language models with optimal planning proficiency, 2023.

[LKs+23] Man Luo, Shrinidhi Kumbhar, Ming shen, Mihir Parmar, Neeraj Varshney, Pratyay Banerjee, Somak Aditya, and Chitta Baral. Towards LogiGLUE: A brief survey and a benchmark for analyzing logical reasoning capabilities of language models, 2023.

[LLL23] Jiatong Li, Rui Li, and Qi Liu. Beyond static datasets: A deep interaction approach to LLM evaluation, 2023.

[LMG23] Adian Liusie, Potsawee Manakul, and Mark J. F. Gales. LLM comparative assessment: Zero-shot NLG evaluation through pairwise comparisons using large language models, 2023.

[LMPF+23] Emanuele La Malfa, Aleksandar Petrov, Simon Frieder, Christoph Weinhuber, Ryan Burnell, Anthony G Cohn, Nigel Shadbolt, and Michael Wooldridge. Language models as a service: Overview of a new paradigm and its challenges. *arXiv preprint arXiv:2309.16573*, 2023.

[LNT+23] Hanmeng Liu, Ruoxi Ning, Zhiyang Teng, Jian Liu, Qiji Zhou, and Yue Zhang. Evaluating the logical reasoning ability of ChatGPT and GPT-4, 2023.

[LSS+24] Lucas Lehnert, Sainbayar Sukhbaatar, DiJia Su, Qinqing Zheng, Paul Mcvay, Michael Rabbat, and Yuandong Tian. Beyond A*: Better planning with Transformers via Search Dynamics Bootstrapping, 2024.

[LSY+23] Junlong Li, Shichao Sun, Weizhe Yuan, Run-Ze Fan, Hai Zhao, and Pengfei Liu. Generative judge for evaluating alignment, 2023.

[lTN+23] Hanmeng liu, Zhiyang Teng, Ruoxi Ning, Jian Liu, Qiji Zhou, and Yue Zhang. GLoRE: Evaluating logical reasoning of large language models, 2023.

[MOS+23] Michaël Mathieu, Sherjil Ozair, Srivatsan Srinivasan, Caglar Gulcehre, Shangtong Zhang, Ray Jiang, Tom Le Paine, Richard Powell, Konrad Żołna, Julian Schrittwieser, David Choi, Petko Georgiev, Daniel Toyama, Aja Huang, Roman Ring, Igor Babuschkin, Timo Ewalds, Mahyar Bordbar, Sarah Henderson, Sergio Gómez Colmenarejo, Aäron van den Oord, Wojciech Marian Czarnecki, Nando de Freitas, and Oriol Vinyals. AlphaStar Unplugged: Large-scale offline reinforcement learning, 2023.

[MPF+24] Emanuele La Malfa, Aleksandar Petrov, Simon Frieder, Christoph Weinhuber, Ryan Burnell, Raza Nazar, Anthony Cohn, Nigel Shadbolt, and Michael Wooldridge. Language-models-as-a-service: Overview of a new paradigm and its challenges. *Journal of Artificial Intelligence Research*, 80:1–30, 2024.

[NCY+23] Ranjita Naik, Varun Chandrasekaran, Mert Yuksekgonul, Hamid Palangi, and Besmira Nushi. Diversity of thought improves reasoning abilities of large language models, 2023.

[OGL+23] Theo X. Olausson, Alex Gu, Benjamin Lipkin, Cedegao E. Zhang, Armando Solar-Lezama, Joshua B. Tenenbaum, and Roger Levy. LINC: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers, 2023.

[Ope] OpenAI. OpenAI models documentation. https://platform.openai.com/docs/models/gpt-4o. Retrieved: 2024-05-16.

[Ope23] OpenAI. GPT-4 technical report, 2023.

[OSG+23] Odhran O'Donoghue, Aleksandar Shtedritski, John Ginger, Ralph Abboud, Ali Essa Ghareeb, Justin Booth, and Samuel G Rodriques. BioPlanner: Automatic evaluation of LLMs on protocol planning in biology, 2023.

[PCOT23] Julien Pourcel, Cédric Colas, Pierre-Yves Oudeyer, and Laetitia Teodorescu. ACES: Generating diverse programming puzzles with autotelic language models and semantic descriptors, 2023.

[PPR23] Vagelis Plevris, George Papazafeiropoulos, and Alejandro Jiménez Rios. Chatbots put to the test in math and logic problems: A preliminary comparison and assessment of ChatGPT-3.5, ChatGPT-4, and Google Bard, 2023.

[QWL+23] Dan Qiao, Chenfei Wu, Yaobo Liang, Juntao Li, and Nan Duan. GameEval: Evaluating LLMs on conversational games, 2023.

[RDM+24] Anian Ruoss, Grégoire Delétang, Sourabh Medapati, Jordi Grau-Moya, Li Kevin Wenliang, Elliot Catt, John Reid, and Tim Genewein. Grandmaster-level chess without search. *arXiv preprint arXiv:2402.04494*, 2024.

[S+23] Aarohi Srivastava et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models, 2023.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[SBT23] Justin Stevens, Vadim Bulitko, and David Thue. Solving Witness-type triangle puzzles faster with an automatically learned human-explainable predicate, 2023.

[SDS+23] Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B. Tenenbaum, Leslie Pack Kaelbling, and Michael Katz. Generalized planning in PDDL domains with pretrained large language models, 2023.

[SFLG23] Ying Su, Xiaojin Fu, Mingwen Liu, and Zhijiang Guo. Are LLMs rigorous logical reasoner? empowering natural language proof generation with Contrastive Stepwise Decoding, 2023.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[Sie] Aaron Siegel. CGSuite. https://www.cgsuite.org/. Retrieved: 2024-05-10.

[Sie13] Aaron N. Siegel. *Combinatorial Game Theory*. Graduate Studies in Mathematics, Vol. 146. American Mathematical Society, 2013.

[SKPK21] Tal Schuster, Ashwin Kalyan, Oleksandr Polozov, and Adam Tauman Kalai. Programming puzzles, 2021.

[SL23] Damien Sileo and Antoine Lernould. MindGames: Targeting theory of mind in large language models with dynamic epistemic modal logic, 2023.

[Spa] William Spaniel. The Game Theory of Soccer Penalty Kicks. https://williamspaniel.com/2014/06/12/the-game-theory-of-soccer-penalty-kicks/. Retrieved: 2024-02-29.

[SPH+23] Tomohiro Sawada, Daniel Paleka, Alexander Havrilla, Pranav Tadepalli, Paula Vidas, Alexander Kranias, John J. Nay, Kshitij Gupta, and Aran Komatsuzaki. ARB: Advanced reasoning benchmark for large language models, 2023.

[SPP+23] Abulhair Saparov, Richard Yuanzhe Pang, Vishakh Padmakumar, Nitish Joshi, Seyed Mehran Kazemi, Najoung Kim, and He He. Testing the general deductive reasoning capacity of large language models using OOD examples, 2023.

[TLI+23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models, 2023.

[TWL+23] Yongqi Tong, Yifan Wang, Dawei Li, Sizhe Wang, Zi Lin, Simeng Han, and Jingbo Shang. Eliminating reasoning via inferring with planning: A new framework to guide LLMs' non-linear thinking, 2023.

[VOSK23] Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Large language models still can't plan (a benchmark for LLMs on planning and reasoning about change), 2023.

[WCO+23] Xinyi Wang, Lucas Caccia, Oleksiy Ostapenko, Xingdi Yuan, and Alessandro Sordoni. Guiding language model reasoning with planning tokens, 2023.

[Wik] Wikipedia. Castling in chess. https://en.wikipedia.org/wiki/Castling. Retrieved: 2024-04-04.

[WLC+23] Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Qi Liu, Tianyu Liu, and Zhifang Sui. Large language models are not fair evaluators, 2023.

[WLM+23] Jiaan Wang, Yunlong Liang, Fandong Meng, Zengkui Sun, Haoxiang Shi, Zhixu Li, Jinan Xu, Jianfeng Qu, and Jie Zhou. Is ChatGPT a good NLG evaluator? a preliminary study, 2023.

[WLZ+21] Siyuan Wang, Zhongkun Liu, Wanjun Zhong, Ming Zhou, Zhongyu Wei, Zhumin Chen, and Nan Duan. From LSAT: The progress and challenges of complex reasoning, 2021.

[WMW+23a] Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. Unleashing cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration, 2023.

[WMW+23b] Taylor Webb, Shanka Subhra Mondal, Chi Wang, Brian Krabach, and Ida Momennejad. A prefrontal cortex-inspired architecture for planning in large language models, 2023.

[WTB+22] 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, 2022.

[WTX+22] Eric Wallace, Nicholas Tomlin, Albert Xu, Kevin Yang, Eshaan Pathak, Matthew Ginsberg, and Dan Klein. Automated crossword solving, 2022.

[WTY+23] Ruoyao Wang, Graham Todd, Eric Yuan, Ziang Xiao, Marc-Alexandre Côté, and Peter Jansen. ByteSized32: A corpus and challenge task for generating task-specific world models expressed as text games, 2023.

[WWS+23] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[WYT+23] Tianlu Wang, Ping Yu, Xiaoqing Ellen Tan, Sean O'Brien, Ramakanth Pasunuru, Jane Dwivedi-Yu, Olga Golovneva, Luke Zettlemoyer, Maryam Fazel-Zarandi, and Asli Celikyilmaz. Shepherd: A critic for language model generation, 2023.

[WYZ+23] Yidong Wang, Zhuohao Yu, Zhengran Zeng, Linyi Yang, Cunxiang Wang, Hao Chen, Chaoya Jiang, Rui Xie, Jindong Wang, Xing Xie, Wei Ye, Shikun Zhang, and Yue Zhang. PandaLM: An automatic evaluation benchmark for LLM instruction tuning optimization, 2023.

[XLV+24] Yudong Xu, Wenhao Li, Pashootan Vaezipoor, Scott Sanner, and Elias B. Khalil. LLMs and the abstraction and reasoning corpus: Successes, failures, and the importance of object-based representations, 2024.

[XYZ+23] Yaqi Xie, Chen Yu, Tongyao Zhu, Jinbin Bai, Ze Gong, and Harold Soh. Translating natural language to planning goals with large-language models, 2023.

[YCC+23] Lifan Yuan, Yangyi Chen, Ganqu Cui, Hongcheng Gao, Fangyuan Zou, Xingyi Cheng, Heng Ji, Zhiyuan Liu, and Maosong Sun. Revisiting out-of-distribution robustness in NLP: Benchmark, analysis, and LLMs evaluations, 2023.

[YCDD23] Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. SatLM: Satisfiability-aided language models using declarative prompting, 2023.

[YGW23] Fei Yu, Anningzhe Gao, and Benyou Wang. Outcome-supervised verifiers for planning in mathematical reasoning, 2023.

[YIL23] Zhun Yang, Adam Ishay, and Joohyung Lee. Coupling large language models with logic programming for robust and general reasoning from text, 2023.

[YJDF20] Weihao Yu, Zihang Jiang, Yanfei Dong, and Jiashi Feng. ReClor: A reading comprehension dataset requiring logical reasoning, 2020.

[YKG+23] Dingli Yu, Simran Kaur, Arushi Gupta, Jonah Brown-Cohen, Anirudh Goyal, and Sanjeev Arora. Skill-Mix: a flexible and expandable family of evaluations for AI models, 2023.

[YYZ+23] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023.

[ZWS+23] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena, 2023.

[ZYSR+23] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models, 2023.