TU WIEN Informatics

Towards Improving Monte Carlo Tree Search for Games with Imperfect Information

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Lukas Grassauer, BSc.
Registration Number 01526001

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Ing. Mag.rer.soc.oec. Dr.rer.soc.oec. Horst Eidenberger

Vienna, February 23, 2024

Lukas Grassauer	Horst Eidenberger

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

Erklärung zur Verfassung der Arbeit

Lukas Grassauer, 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.

Wien, 23. Februar 2024

Lukas Grassauer

Acknowledgements

First and foremost, I want to thank my thesis advisor, Professor Horst Eidenberger, for the invaluable guidance, patience, and for the ongoing encouragement. His advice was pivotal in determining the course and execution of this work.

Furthermore, I'd like to express my appreciation to my parents, Dr. Andreas Grassauer, and Dr. Daniela Dörfler, for their continued support, and for believing in me.

My gratitude goes out to all those who listened to my ideas. You shaped this thesis, and your support has been invaluable.

Kurzfassung

Die Monte-Carlo-Baumsuche (MCTS, engl. Monte Carlo tree search) ist ein weit verbreiteter Suchalgorithmus für Spiele. Bei Spielen mit perfekter Information wurden bereits beeindruckende Ergebnisse erzielt, und der Algorithmus hat sich als bevorzugte Methode im anspruchsvolleren Bereich des allgemeinen Spielens herauskristallisiert. Beim allgemeinen Spielen müssen Agenten Spiele meistern, ohne deren Regeln vorher zu kennen. Viele Verbesserungen und Optimierungen für MCTS sind jedoch bei Spielen mit imperfekter (oder versteckter) Information oder beim allgemeinen Spielen nicht verfügbar. Daher waren Neuentwicklungen für MCTS mit dem Fokus auf diese Beschränkungen notwendig. Um die Spielstärke zu verbessern, schlug der Autor dieser Arbeit den neuartigen Ansatz vor, die Expansionstiefe des Suchbaums zu begrenzen. Der Ansatz erforderte die Identifizierung einer grundlegenden MCTS-Modifikation und ein Mittel zur Konfiguration der maximalen Expansionstiefe. Anschließend wurde der Ansatz in einem Experiment quantitativ bewertet. Dazu musste eine Engine implementiert werden, die in der Lage ist, allgemeine Spielpartien zu orchestrieren, Spiele mit imperfekter Information zu interpretieren und eine Reihe von Agenten zu entwickeln. In dieser Arbeit werden die wichtigsten Details der Engine- und Agentenimplementierungen vorgestellt. Sie präsentiert eine Analyse geeigneter MCTS-Modifikationen. Außerdem wird die Verwendung von Merkmalsextraktion als Mittel zur Konfiguration der maximalen Expansionstiefe untersucht. Abschließend wird über die Ergebnisse zweier Turniere berichtet, die durchgeführt wurden, um die Veränderung der Spielstärke bei Begrenzung der Expansionstiefe zu bewerten.

Für die betrachteten Fälle war die Multi-Beobachter-Informationsset MCTS (MO-ISMCTS, engl. multi observer-information set MCTS) die bevorzugte Modifikation. Die Merkmalsextraktion war hilfreich, aber nicht ausreichend, um den Agenten mit begrenzter Expansionstiefe zu konfigurieren. Bei diesem Agenten war im Vergleich zum Basisagenten ein deutlicher Unterschied in der Spielstärke. Überraschenderweise erreichte er in einer Variante von *Corridor* eine um $6,67\%$ höhere Gewinnrate (10 mehr, von 150 Spielen) im Vergleich zum Basisagenten. Infolgedessen sollte MO-ISMCTS mit einer begrenzten Expansionstiefe für Spiele mit imperfekter Information in Betracht gezogen werden. Es bleibt unklar, ob der Vorteil nur für ausgewählte Spiele gilt oder ob er auf eine breitere Klasse von Spielen anwendbar ist. Zukünftige Arbeiten werden sich auf die Klärung dieser Frage und die Weiterentwicklung der Engine konzentrieren.

Abstract

Monte Carlo tree search (MCTS) is a commonly used search algorithm for games. Impressive results have been achieved in perfect information games, and the algorithm emerged as the preferred method within the more challenging field of general game playing, where agents are required to play games without knowing their rules beforehand. However, many improvements and optimizations for MCTS are not available in imperfect (or hidden) information games or in general game playing. Therefore, new developments for MCTS with a focus on these constraints were necessary. To improve playing strength, the author of this thesis proposed the novel approach of limiting the search tree's expansion depth. The approach required to identify a baseline MCTS modification and a means to configure the maximum expansion depth. Subsequently, the approach was quantitatively evaluated with an experiment. This required implementing an engine capable of orchestrating general game playing matches, interpreting imperfect information games, and a set of agents. This thesis highlights the key details of the engine's and agent's implementations. It presents analysis of suitable MCTS modifications. Additionally, it examines the use of feature extraction as a means of configuring the maximum expansion depth. Finally, the thesis reports on the results of two tournaments, conducted to evaluate the change in playing strength when limiting the expansion depth.

For the considered cases, multi observer-information set MCTS (MO-ISMCTS) was the preferred modification. Feature extraction was helpful, but not sufficient for configuring the agent using limited expansion depth. This agent had a distinct playing strength compared to the baseline agent. Surprisingly, in the game of the *Corridor* family, it achieved a 6.67% higher win rate (10 more, out of 150 matches) compared to the baseline. As a consequence, MO-ISMCTS with a limited expansion depth should be considered for imperfect information games. It remains unclear whether the advantage is only applicable to the chosen games, or if it applies to a wider class of games. Future efforts will be focused on addressing this question and further developing the engine.

Contents

CHAPTER 1

Introduction

Games encapsulate complex decision-making processes. They are finite systems with well-defined metrics that have a long tradition in benchmarking agents. The advancements in the broader category of artificial intelligence (AI) resonate in the field of agent game playing, which have produced impressive results. Through the improvements made in hardware and development of new methods, games which deemed to be impossible to tackle, became accessible. Despite this, the class of games with imperfect (viz. hidden) information in the context of general game playing (GGP) remain insufficiently explored. This thesis aims to investigate the use of Monte Carlo tree search (MCTS) for playing games of this class. Thereby, providing insights in a novel modification of limiting the expansion depth, and its viability for imperfect information games. The ensuing sections will outline the motivation for the topic, define the problem statement, detail the objectives of this research and present the structure of this thesis.

1.1 Motivation

There is a fascination in letting agents play games that reaches back to the dawn of the computer itself. Alan Turing conceptualized a Chess engine before programming in a modern sense was possible (see [29]) and a few decades later the topic reached popular culture, after IBM's Deep-Blue beat the then-reigning Chess World Champion Garry Kasparov. Recently, interest peaked again after Lee Sedol lost to DeepMind's AlphaGo in the ancient game of Go, an achievement long thought to be impossible. While playing a specific game is certainly impressive, the field of GGP forces agents to be more adaptive, as they have to be able to play any game. The leading method for this task is MCTS, which has received significant attention after the result of the match between Lee Sedol and AlphaGo.

Games serve as the ideal proving ground for algorithms. They are approachable for a broad audience, yet rich complexity arises from them. They allow systematic exploration

and evaluation of search methods, pattern recognition and strategy learning. They have a clear metric of success— win, tie or loss, which is often lacking in other fields.

There is a tight conceptual link between game playing and AI-systems. Such systems act upon input received from their environment. Games provide low-stakes playground and benchmark environments. They challenge agents in planning, foresight and decision-making, features commonly associated with intelligence.

GGP confronts the agent with unknown environments. It requires agents to play games without human intervention, and without knowing the rules beforehand. Real-world problems require agents to adapt to the rules of the situation, while staying performant enough to be able to deal with new constraints. Specialization has to be done ad-hoc or not at all.

A fascinating class of games are those with imperfect information. These include elements of hidden information. Think of examples like Poker, where the opponent's hands are not known to a player, but also games like Kriegspiel (sometimes called Phantom Chess), where the moves are not communicated to the opponent. In contrast, are perfect information games, for example Go, where the whole state is known, and each move is clearly visible for all players.

Imperfect information games are analogous to problems where not every detail is available to the agent. Think of the field of medical diagnosis, where critical data about the patient might be missing. Another example would be financial forecasting, where it is sometimes in the best interest for some agents to hide their moves to gain an advantage. Or consider the discipline of autonomous driving, in which many uncertainties arise naturally— other vehicles and the weather conditions obscure the view of the agent.

The synthesis of playing imperfect information games in a GGP context is a true challenge for agents. It pushes the methods available today to their limits and serves as a proper benchmark for performance. It allows measuring if an algorithm is ready for the messy incomplete data encountered in the real world and can handle unseen as well as unknown rules.

In the long history of search methods, MCTS has emerged as the most promising method for both GGP and imperfect information games. It uses a sampling method to replace a heuristic function, which is often required for other methods to perform well enough. Additionally, because of the sampling nature of the algorithm, dealing with hidden information is conceptually only a short leap.

As the hardware improved, results grew became more notable and new avenues and methods opened up, such as machine learning (ML). As a result, agents using MCTS with neural nets guiding the search, quickly became the state-of-the-art for solving specific games, where a heuristic developed with expert-knowledge used to be the best available method. Many suggested methods remain unexplored and to be evaluated to this day.

Subsequently, the results MCTS produced were impressive. AlphaGo beat Lee Sedol 4:1 using MCTS with two deep neural networks (see Silver et al. [25]). The networks were

trained by reinforcement learning using self-play. The approach was further demonstrated to be viable for the games Chess and Shogi in Silver et al. [24]. In principle, the technique could be used for any game, given that the rules are known beforehand and sufficient resources are available. Note that these conditions are not met in a GGP context.

1.2 Problem Statement

In the discipline of GGP, games with imperfect information present a unique challenge. The combinatorial explosion due to the limited view and the partially observable moves, necessitates algorithms capable of effectively handling uncertainty, while staying applicable for all and any game. MCTS is a powerful tool for perfect information contexts, and impressive results have been produced using it as the primary method for game playing. Without the correct modifications, however, it is not effective when applied to games with imperfect information. Furthermore, many optimizations and approaches available normally, are no longer available in a GGP context.

The lack of knowledge in imperfect information games introduces a new quality of complexity. Think of the game Battleship. Battleship is a two-player game where each player has an arrangement of ships on a grid that are hidden from the opponent. In turn, each player shoots the enemies' grid and the other player announces if it was a hit or a miss. Consider a variant of this game where only one of the player's ships were hidden. Certainly, this player would have a significant advantage, as the other player would be forced to guess, while they themselves could pick off the opponent's ships without ever missing. The player with this advantage does not need to consider the uncertainty. Agents are no longer solely responsible for devising their strategies but are also required to consider the available data, the potential knowledge of their adversaries, and the potential adaptation of new information upon its availability.

Prior work has focused on improving MCTS for perfect information contexts, and the remaining research is divided between GGP and imperfect information games. There is a significant gap in performance between games with imperfect information and perfect information scenarios, where impressive results have been achieved. This is especially apparent in the discipline of GGP, where game-specific knowledge cannot be used to improve playing strength. However, many of the optimizations carry over and might be used in the joint context of GGP and imperfect information games. It remains to be revised and tested which of these optimizations are effective for the joint case.

When playing games with imperfect information, there is a significant increase in complexity, because of the uncertainty about the game's state. Predictions made about the state of the game decrease in value as the distance from the current state increases. The conditionals of the uncertainty compound. As the expansion depth of MCTS is unbounded, the procedure fails to deal with this efficiently. This is addressed as by a novel modification, proposed by the author. This modification sets a limit to the maximum expansion depth available to MCTS.

A fundamental limitation of playing games without knowing the rules beforehand is that aids like handcrafted heuristics are not available, or guiding the search with neural networks trained by reinforcement learning is infeasible. Nonetheless, the literature suggests different approaches for generating such improvements just-in-time, including statically analyzing the ruleset and dynamically sampling the games. Both methods allow extracting characteristic features of the game and can subsequently be used for ad-hoc heuristics. This approach will also be refered to as *feature extraction*. Previous work has not yet produced enough data to test if this approach is also viable for informing the value of the limit of the expansion depth. By contributing quantitative and qualitative evaluations, the author intents to firm the use of these approaches.

As GGP and MCTS have been a topic of discussion for a relatively long time, many implementations exist. There is no agreed standard, and the tooling surrounding is often outdated. Furthermore, the focus of these tools is more often than not on perfect information games. Thus, there is no apt implementation of an engine capable of GGP and imperfect information games. One of the contributions of this thesis is such an artifact.

1.3 Research Object

The contribution of this thesis is fourfold. Firstly, it surveys the available modifications for MCTS, while it provides reference implementations for the most important of those. Secondly, it studies the viability of limiting the expansion depth by experiment. In this experiment, GGP-agents played against each other using a custom implementation of an engine ready for imperfect information games. The data resulting from the experiment serves as the third contribution, while the engine used is the fourth and final contribution of the thesis.

The available literature presents many modifications of MCTS for games with imperfect information. Some of those modifications work better than others, and many share their fundamentals. The author of this thesis aimed to examine the large body of work and use qualitative analysis on two concrete examples to answer the question of which of the modifications for MCTS are effective for playing imperfect information games in a GGP context.

As MCTS is a general method, it can be configured to suite a specific problem. Similarly, the proposed modification of limiting the maximum expansion depth, can be adapted by changing where this cap actually is. During the quantitative evaluation of the validity of this novel approach, the author used static and dynamic feature extraction on two example games to inform the value of the limit. From this, a suggestion for a more general method for arbitrary games was extracted. This answered the question of which features can be extracted from descriptions of imperfect information games to construct a heuristic to set the maximum expansion depth for the MCTS algorithm.

The metric of success – frequency of wins – is clearly defined in GGP, thus quantitative

evaluation is relatively accessible. Often, previous work uses running tournaments of games as experiments to indicate how well a method does. To evaluate how a MCTS-agent with limited expansion depth compares in benchmarks, the author conducted such an experiment as a measure of quantitative evaluation. The results show that in special cases, the approach appears to be viable.

Besides the data produced during the benchmark of the modification, the engine used for the experiment serves as a workbench for research in playing games. It is written in Python using contemporary software engineering principles. Together with the implementation of agents with effective modifications of MCTS for playing games with imperfect information and a GGP-orchestrator, it is a comprehensive work environment for future experiments in this field.

1.4 Outline

This thesis is divided into seven chapters. Chapter 1 motivates the broader topic, states the fields' most recent problems, lists the research hypotheses and finally outlines the thesis. Chapter 2 provides background and reviews the available literature. In chapter 3 a discussion of scientific method is followed by a detailed report of the experiment setup. An overview of the engine's and agent's implementation is provided by chapter 4. The results of the research can be found in chapter 5, and they are discussed in chapter 6. Finally, a summary, outlook for future work, and most important contributions are presented in chapter 7.

Chapter 2 will lay out the necessary theoretical background while summarizing the prior art. Section 2.1 introduces the elementary principles and the protocol of GGP. Furthermore, it highlights the difference between perfect and imperfect information games from the context of GGP, and finally introducing the game description language (GDL). In section 2.2 the necessary definitions and concepts for viewing perfect and imperfect information games as trees, are introduced. These preliminaries are vital for section 2.3, which reviews MCTS and the most important modifications for playing imperfect information games found in literature.

The scientific methods and experiment setup used in this thesis are discussed in chapter 3. At first, the overall scientific approach is reviewed in section 3.1, and afterward the details for the qualitative and quantitative evaluations are laid out. Section 3.2 covers the comparison of the existing MCTS modifications, while section 3.3 discusses the qualitative approach for extracting features, that are used to set the maximum expansion depth. Finally, section 3.4 goes into detail for how the quantitative evaluation was planned and conducted.

The engine used for the experiment is the primary focus of chapter 4. This chapter touches upon the most interesting details, namely the overall architecture of the engine, the interpreter, and the match orchestration in section 4.1, while section 4.2 dives deep

into the tree data structure and section 4.3 outlines implementation details for the agents themselves.

Chapter 5 reports on the results of the research. The qualitative evaluation of the existing modifications for MCTS can be found in section 5.1, while the results for the static feature extraction may be read in section 5.2. The data produced by the experiment is in section 5.3.

A discussion of the results may be read in chapter 6. The chapter is sectioned according to the research questions.

Finally, chapter 7 summarizes the thesis, concludes the findings, gives perspectives for future work, and reviews the most important contributions.

CHAPTER 2

Background and Literature Review

This chapter introduces the required background for tracing the experiment setup (see section 3.4), to understand the implementation (see chapter 4), and reviews the prior-art. Section 2.1 gives an introduction to the field of general game playing (GGP). Section 2.2 presents the background and reviews the literature for viewing games as trees in the context of GGP. The search method Monte Carlo tree search (MCTS) is introduced in section 2.3.

2.1 General Game Playing

General game playing (GGP) is the challenge of playing games only by their definition, and no prior knowledge of them. It is a generalization of the field of autonomous computer playing. Through the generalization, the implementation of search algorithm is less prone to the bias of domain-knowledge. GGP encompasses both perfect and imperfect information games. This section of the thesis introduces the general concepts (see subsection 2.1.1), the commonly used protocol (see subsection 2.1.2), GGP in the context of perfect information (see subsection 2.1.3), as well as imperfect information (see subsection 2.1.4), and a standardized language for defining games (see subsection 2.1.5).

2.1.1 Elementary Principles

The terminology around games is highly overloaded. To avoid confusion, this subsection discusses the most important terms and introduces the elementary principles underlying GGP. The terms are *game*, *action*, *actor*, *role*, *move*, *turn*, *ply*, *history*, and *match*.

The word *game* is a synonym for its rules. The rules define a deterministic and finite state machine. A state machine has an internal state that is changed by *actions*. A state

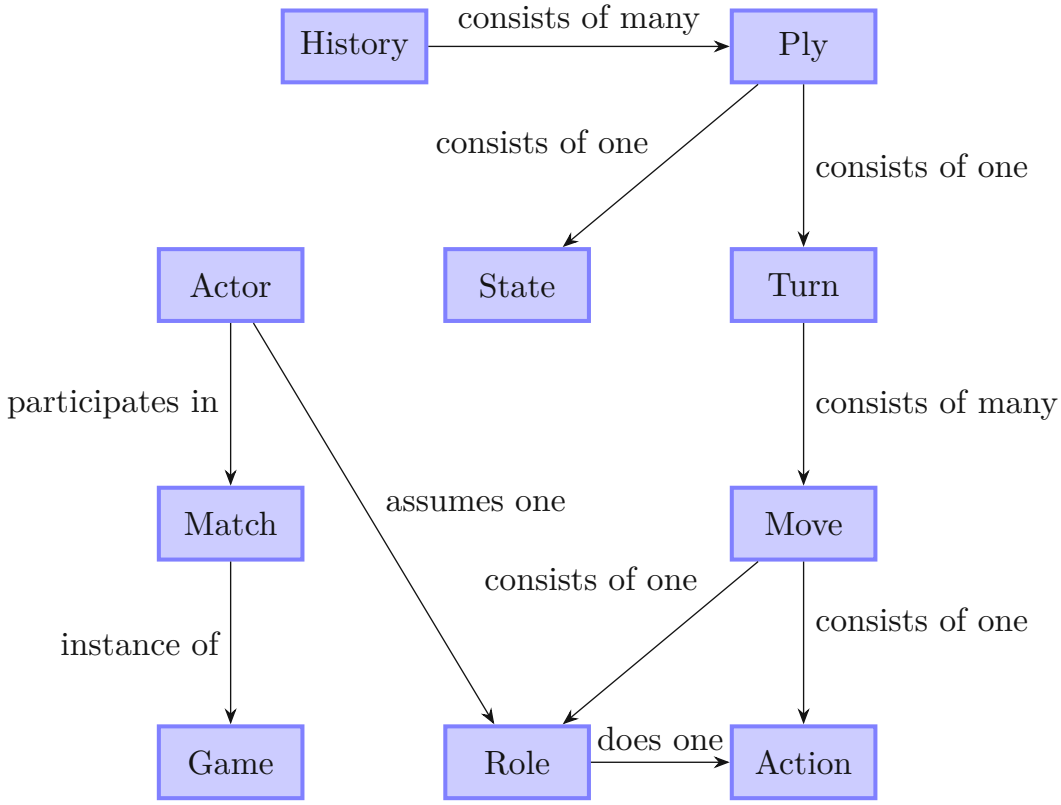

Figure 2.1: An abstract entity-relationship diagram of the term used for games.

is defined by a set of atoms, which are either true or false. *Actions* are performed by the *actors*.

A *role* is a distinct entity in a game, assumed by an *actor*. There can be any non-zero, finite number of roles. In the example of Chess, there are two roles, one role controlling the white pieces, and the other role controlling the black pieces. Poker may have three or more roles, a dealer and at least two players.

A *move* is a pair consisting of a *role* and an *action*. Given a state, a move can be legal or illegal. Not every move is legal in every state, and not all roles can do the same actions (and by extension the same moves).

The state contains the information which role has to do an action. As a convention, this paper uses, *a role is in control* instead of *a role has to do an action*.

Turns are groups of moves done at the same time. A turn consists of a move from every role that is in control. Depending on the roles that are in control, a turn can consist of a single move or multiple moves.

A *ply* is a state-turn pair. Plies are counted and ordered. The first ply contains the

initial state and the first turn. The second ply consists of the first turn applied to the initial state, and so forth.

A series of plies is called a *history* (see also [20]). This thesis discerns between *open* and *closed* histories. The last state of an open history is not terminal, while a *closed* history ends in a terminal state.

A *match* describes an instance of a game. Therefore, the sentence "Player A and Player B play a match of Chess", precisely means that two players A and B use the rules of the game Chess to play against each other once. This definition reliefs the word *game* of the same interpretation. Each match has a history. The history serves as the record.

2.1.2 Protocol

To conduct GGP matches Genesereth, Love, and Pell [10], suggest a protocol. The protocol tasks a *gamemaster* with the mediation of the match. Furthermore, it conceptually splits a match into four distinct phases and defines how the phases change.

The *gamemaster* is the matches' administrator and ensures that no actor has an advantage over other actors by assuming this role. The gamemaster has to keep the canonical copy of the state (separate to the copies of the actors) and progress it once all actors made their moves. Furthermore, it keeps the actors informed about the game. An actor cannot manipulate the state to its advantage, as the gamemaster only allows legal moves.

A match can be in one of four phases, the start-, play-, completed-, and aborted-phase (see fig. 2.2). During the start-phase, the actors are informed about the game they are playing. The actors submit their moves during the play phase, which leads to another iteration of the play-phase or is followed by either transitioning to completed or aborted. A match is completed if a terminal state is reached, should the match end in any other way, it is aborted.

Before the match starts, the gamemaster sends to the actors, the rules, the mapped roles and how much time the actor has to answer in the start phase and how much time per ply the actor has. Afterward, the actors acknowledge and the main game loop starts. The rules are usually defined via the game description language (GDL) (see subsection 2.1.5). The gamemaster initializes the state with the definition given by the rules.

To ensure fair conduct, the gamemaster keeps two clocks per actor, the start clock and the game clock. The start clock counts down as soon as the gamemaster sends the message to the agents. It is stopped once the gamemaster receives the acknowledgment. The game clocks count down during the plies of the main game loop if the player is in control. If a clock reaches zero, the match is aborted, and the associated actor is disqualified.

Games may be classified into two types, simultaneous- and non-simultaneous moving games. Simultaneous moving games include those where more than one player at a time is in control. On the contrary, non-simultaneous moving games only have exactly one

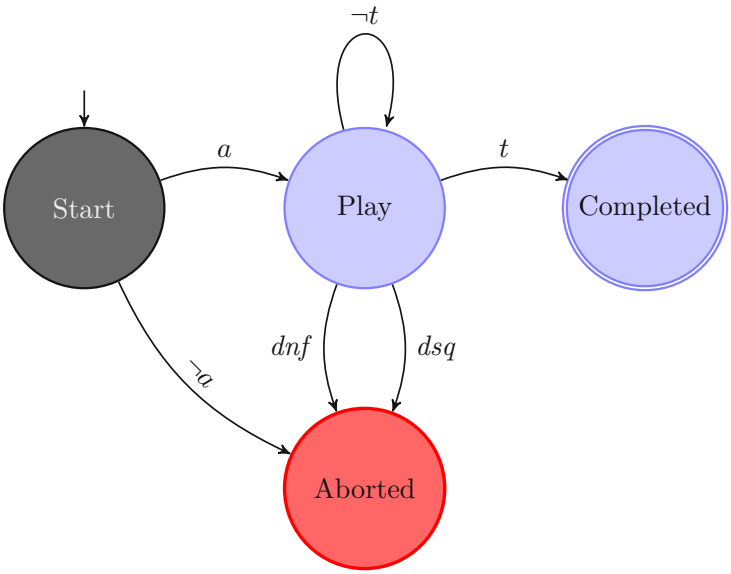

Figure 2.2: The phases of a match. The filled black state is the entry point, the state with double borders is the final phase and the red state indicates an error state. The transation a, requires all actors acknowledge in time, t is applicable if the state is terminal, *dnf* means that some actor failed to submit move in time, while if some actor submitted an illegal move *dsq* is applicable.

player in control. In general, every non-simultaneous moving game can be expressed as a simultaneous moving game by modeling the inaction of the opponents with a null or no-operation move. The inverse direction is only possible under certain conditions (for more information see subsection 2.1.3 and subsection 2.1.4).

During the play loop, each actor whose role is in control, must submit a move. If the move is deemed illegal, the match is aborted. Otherwise, the gamemaster combines all moves to a turn and applies it to the current state, advancing to the next state. Until a terminal state is reached, the play loop repeats.

Each role has a goal value associated with it, for every state in the game. The goal value is either an integer or a null value. The goal values of terminal states have to be integers. They are synonymous with the *utility* of the roles. The actor with the highest utility wins the match.

The match switches to the *completed* phase if it reaches a terminal state. The gamemaster uses the rules and the terminal state to calculate the utility for each role. Finally, it submits the utility to the actors. Afterward, the match has ended.

2.1.3 Perfect Information Games

Perfect information games describe a class of games without hidden information. Therefore, every entity involved knows everything that can be known. Each match starts with a defined initial state. The gamemaster informs every actor about the moves of the other roles. As the rules are deterministic, the actors may synchronize their state by applying the turns submitted by the gamemaster. Popular games with perfect information include Tic-tac-toe, Chess and Go.

The formal specification of an n-player perfect information game, requires the following definitions (adapted from [10]):

$$
\begin{aligned}
S & \quad \text{set of all game states} \\
s_i \in S & \quad \text{initial state} \\
S_t \subseteq S & \quad \text{terminal states} \\
R = \{r_1, \ldots, r_n\} & \quad \text{set of the } n \text{ roles} \\
A & \quad \text{set of all actions} \\
T = \{\{(r,a) \mid r \in R', a \in A\} \mid R' \in 2^R, R' \neq \emptyset\} & \quad \text{set of all turns} \\
M = R \times A & \quad \text{set of all moves} \\
\ell \subseteq S \times M & \quad \text{legal relation} \\
n \subseteq S \times T \times S & \quad \text{next state relation} \\
g \subseteq S \times R \times (\mathbb{Z} \cup \{\texttt{null}\}) & \quad \text{goal relation}
\end{aligned}
\tag{2.1}
$$

Elements of chance, such as dice throws or card draws, are modeled with an additional role *random*. This role selects its move randomly and has the lowest possible goal value. Hidden information cannot be modeled, this requires an extension (see subsection 2.1.4).

Any non-simultaneous game can be translated to a simultaneous one. Without a formal proof: Add a null or no-operation move to A (the set of all actions). Let this move be called `null`. The legal relation ℓ is extended by pairs of all roles and `null` for every state where the given role is not in control. Furthermore, the turns of the next state relation n, require `null` from every role that is not in control.

Generally, simultaneous perfect information games cannot be converted to non-simultaneous games. Simultaneous turns would need to be broken up into sequential turns. Thus, one player would have the advantage of seeing the opponent's moves before having to make their move.

Another example of a simple perfect information game is Nim. Nim was named and extensively studied by Bouton [3]. It is often used, as it is simple to understand and is rich in tools for analysis. Nim is a basic alternating two-player game. At the start, there are a defined number of straws and the goal is to take the last straw. A player may take

one, two, or three straws from the heap, thereby reducing the number of straws. There are many versions known, with different sized heaps and additional rules.

To illustrate, consider a variant of Nim with a single heap of five straws. Let this variant be called 5-Nim. Definition 2.2 characterizes the game, adhering to definition 2.1.

The number of straws is represented as **n** or n depending on the number of straws n and **bold** if **first** is in control or *italic* if *second* is in control. The actions are $\mathbb{1}, 2, \mathbb{3}$ taking $1, 2$ or 3 straws respectively.

$$
\begin{aligned}
S &= \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{5}\} \cup \\
&\quad \{\mathit{0}, \ldots, \mathit{4}\} \\
s_i &= \mathbf{5} \\
S_t &= \{\mathbf{0}, \mathit{0}\} \\
R &= \{\mathbf{first}, \mathit{second}\} \\
A &= \{\mathbb{1}, 2, \mathbb{3}\} \\
\ell &= (S \setminus \{\mathbf{0}, \mathit{0}\}) \times (R \times \{\mathbb{1}\}) \cup \\
&\quad (S \setminus \{\mathbf{0}, \mathbf{1}, \mathit{0}, \mathit{1}\}) \times (R \times \{2\}) \cup \\
&\quad (S \setminus \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathit{0}, \mathit{1}, \mathit{2}\}) \times (R \times \{\mathbb{3}\}) \\
n &= \{(\mathbf{5}, \{(\mathbf{first}, \mathbb{1})\}, \mathit{4}),\ (\mathbf{5}, \{(\mathbf{first}, 2)\}, \mathit{3}),\ (\mathbf{5}, \{(\mathbf{first}, \mathbb{3})\}, \mathit{2}), \\
&\quad (\mathbf{4}, \{(\mathbf{first}, \mathbb{1})\}, \mathit{3}),\ (\mathbf{4}, \{(\mathbf{first}, 2)\}, \mathit{2}),\ (\mathbf{4}, \{(\mathbf{first}, \mathbb{3})\}, \mathit{1}), \\
&\quad (\mathbf{3}, \{(\mathbf{first}, \mathbb{1})\}, \mathit{2}),\ (\mathbf{3}, \{(\mathbf{first}, 2)\}, \mathit{1}),\ (\mathbf{3}, \{(\mathbf{first}, \mathbb{3})\}, \mathit{0}), \\
&\quad (\mathbf{2}, \{(\mathbf{first}, \mathbb{1})\}, \mathit{1}),\ (\mathbf{2}, \{(\mathbf{first}, 2)\}, \mathit{0}), \\
&\quad (\mathbf{1}, \{(\mathbf{first}, \mathbb{1})\}, \mathit{0}), \\
&\quad (\mathit{4}, \{(\mathit{second}, \mathbb{1})\}, \mathbf{3}),\ (\mathit{4}, \{(\mathit{second}, 2)\}, \mathbf{2}),\ (\mathit{4}, \{(\mathit{second}, \mathbb{3})\}, \mathbf{1}), \\
&\quad (\mathit{3}, \{(\mathit{second}, \mathbb{1})\}, \mathbf{2}),\ (\mathit{3}, \{(\mathit{second}, 2)\}, \mathbf{1}),\ (\mathit{3}, \{(\mathit{second}, \mathbb{3})\}, \mathbf{0}), \\
&\quad (\mathit{2}, \{(\mathit{second}, \mathbb{1})\}, \mathbf{1}),\ (\mathit{2}, \{(\mathit{second}, 2)\}, \mathbf{0}), \\
&\quad (\mathit{1}, \{(\mathit{second}, \mathbb{1})\}, \mathbf{0}), \} \\
g &= \{(\mathit{0}, \mathbf{first}, 1), (\mathit{0}, \mathit{second}, -1), \\
&\quad (\mathbf{0}, \mathbf{first}, -1), (\mathbf{0}, \mathit{second}, 1)\} \cup \\
&\quad (S \setminus S_t) \times R \times 0
\end{aligned}
\tag{2.2}
$$

To reason about matches, the author of this thesis adapts a notation for histories proposed by Schiffel and Thielscher [20]. Histories are prefixed with a unique identifier, features states, which are separated by arrows. These arrows represent the turns, labeled with moves.

For example, consider a history h starting from the state $\square$ and the roles r_1 and r_2 doing the action a and b respectively resulting in the state $\bigcirc$. From this state r_1 does

the action a again and this leads to a terminal state $\triangle$ with the goal values 1 for r_1 and 0 for r_2. Therefore,

$$h : \square \xrightarrow{r_1:a, r_2:b} \bigcirc \xrightarrow{r_1:a} \triangle \mid r_1 : 1, r_2 : 0$$

would compactly represent this history.

As a more concrete example, consider a match of 5-Nim. In this match, **first** takes one straw from the heap and *second* answers by taking another straw from the heap. Finally, **first** wins the game by taking the final three straws. Compactly:

$$h_{\mathbb{1},\mathbb{1},\mathbb{3}} : \mathbf{5} \xrightarrow{\textbf{first}:\mathbb{1}} \mathit{4} \xrightarrow{\mathit{second}:\mathbb{1}} \mathbf{3} \xrightarrow{\textbf{first}:\mathbb{3}} \mathit{0} \mid \textbf{first} : 1, \mathit{second} : -1$$

2.1.4 Imperfect Information Games

Opposed to perfect information games, are imperfect information games, which describe a class of games with hidden information. Therefore, at least one actor, does not know everything that can be known. The gamemaster is a clairvoyant observer. The information visible to the actors, is defined by the rules. Moves of their opponents are no longer submitted to the actors. Instead, only the visible parts of the state are shared. An actor may reconstruct the possible states by applying all legal moves and subsequently comparing which states are compatible with the visible information. Popular games with hidden information include Battleship and Poker.

Formally, a *view* is a partial state. A partial state only defines an assignment for visible atoms, all non-visible atoms may be true or false. The union of all partial states may be total or partial in respect to the actual state.

The formal specification of imperfect information games expands the definition 2.1. The expansion includes the representation of views. It includes a new relation v. The relation maps a partial state to each role and state.

$$v \subseteq R \times S \times V \qquad \text{view relation} \tag{2.3}$$

V is the set of all partial states $V = \{2^s \mid s \in S\}$. It is the power set of every state in S.

Imperfect information games introduce a shift of focus from *the state* to a set of *possible states*. In general, the number of possible states is exponential. Therefore, the computational effort is increased.

Games with imperfect information can be freely translated between simultaneous- to non-simultaneous-moving formulations. Without a formal proof, the non-simultaneous moving formulation can be translated akin to perfect information games, where the non-action is explicitly replaced by a null or no-operation move. Simultaneous moving formulations may break up turns with multiple substates, where the effect of each move is not visible to the involved actors.

To illustrate the formal definition of imperfect information games, consider the game Phantom-Nim, a variant of Nim where the participants cannot directly see which size the heap has. Taking more straws than available, leads to a loss. The changes required for 5-Phantom-Nim, a variant with five initial straws, are listed in definition 2.4. When a term is omitted, definition 2.2 still applies.

$$
\begin{aligned}
V &= \{\mathbf{h}, h\} \\
v &= (R \times \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{5}\} \times \{\mathbf{h}\}) \cup \\
&\quad (R \times \{0, \dots, 4\} \times \{h\}) \\
\ell &= R \times A \\
n &= n' \cup \{ \\
&\quad (\mathbf{2}, \{(\mathbf{first}, 3)\}), \mathbf{0}), \\
&\quad (\mathbf{1}, \{(\mathbf{first}, 3)\}), \mathbf{0}), (\mathbf{1}, \{(\mathbf{first}, 2)\}), \mathbf{0}), \\
&\quad (2, \{(second, 3)\}), 0), \\
&\quad (1, \{(second, 3)\}), 0), (1, \{(second, 2)\}), 0)\}
\end{aligned}
\tag{2.4}
$$

Where n' is the next state relation of definition 2.2. A state where **first** is in control has the view **h** and a state where *second* is in control has the view h.

To illustrate, consider a match of Phantom-5-Nim. A clairvoyant observer sees that at the first ply **first** takes one straw, afterward *second* also only takes one straw, which **first** answers with again taking one straw. Finally, *second* wins the game by taking the last straws. Therefore, the canonical history for the match is:

$$
h_{\mathbb{1},\mathbb{1},\mathbb{1},2} : \mathbf{5} \xrightarrow{\mathbf{first}:\mathbb{1}} 4 \xrightarrow{second:\mathbb{1}} \mathbf{3} \xrightarrow{\mathbf{first}:\mathbb{1}} 2 \xrightarrow{second:2} \mathbf{0} \mid \mathbf{first} : -1, second : 1
$$

However, after the first ply, the possible histories for *second* are:

$$
\begin{aligned}
h_{\mathbb{1}} &: \mathbf{5} \xrightarrow{\mathbf{first}:\mathbb{1}} 4 \\
h_{2} &: \mathbf{5} \xrightarrow{\mathbf{first}:2} 3 \\
h_{\mathbb{3}} &: \mathbf{5} \xrightarrow{\mathbf{first}:\mathbb{3}} 2
\end{aligned}
$$

After *second* made their move, one of three histories may be possible for **first**.

$$
h_{\mathbb{1},\mathbb{1}} : \mathbf{5} \xrightarrow{\mathbf{first}:\mathbb{1}} 4 \xrightarrow{second:\mathbb{1}} \mathbf{3}
$$

$$h_{\mathbb{1},2} : \mathbf{5} \xrightarrow{\textbf{first}:\mathbb{1}} \mathit{4} \xrightarrow{\mathit{second}:2} \mathbf{2}$$
$$h_{\mathbb{1},3} : \mathbf{5} \xrightarrow{\textbf{first}:\mathbb{1}} \mathit{4} \xrightarrow{\mathit{second}:3} \mathbf{1}$$

On the third ply, only three histories are possible for *second*.

$$h_{\mathbb{1},\mathbb{1},\mathbb{1}} : \mathbf{5} \xrightarrow{\textbf{first}:\mathbb{1}} \mathit{4} \xrightarrow{\mathit{second}:\mathbb{1}} \mathbf{3} \xrightarrow{\textbf{first}:\mathbb{1}} \mathit{2}$$
$$h_{\mathbb{1},\mathbb{1},2} : \mathbf{5} \xrightarrow{\textbf{first}:\mathbb{1}} \mathit{4} \xrightarrow{\mathit{second}:\mathbb{1}} \mathbf{3} \xrightarrow{\textbf{first}:2} \mathit{1}$$
$$h_{2,\mathbb{1},\mathbb{1}} : \mathbf{5} \xrightarrow{\textbf{first}:2} \mathit{3} \xrightarrow{\mathit{second}:\mathbb{1}} \mathbf{2} \xrightarrow{\textbf{first}:\mathbb{1}} \mathit{1}$$

From the information *second* received, they can infer that first did not take three straws from the heap; Otherwise, the match would already be over. The match ends after *second* takes the last straws. From the perspective of **first**, two closed histories are possible,

$$h_{\mathbb{1},\mathbb{1},\mathbb{1},2} : \mathbf{5} \xrightarrow{\textbf{first}:\mathbb{1}} \mathit{4} \xrightarrow{\mathit{second}:\mathbb{1}} \mathbf{3} \xrightarrow{\textbf{first}:\mathbb{1}} \mathit{2} \xrightarrow{\mathit{second}:2} \mathbf{0} \mid \textbf{first} : -1, \mathit{second} : 1$$
$$h_{\mathbb{1},2,\mathbb{1},\mathbb{1}} : \mathbf{5} \xrightarrow{\textbf{first}:\mathbb{1}} \mathit{4} \xrightarrow{\mathit{second}:2} \mathbf{2} \xrightarrow{\textbf{first}:\mathbb{1}} \mathit{1} \xrightarrow{\mathit{second}:\mathbb{1}} \mathbf{0} \mid \textbf{first} : -1, \mathit{second} : 1$$

where **first** cannot know which of those two is the canonical history.

2.1.5 Game Description Language

Game description language (GDL) or sometimes game definition language is a formal language to describe a deterministic finite state machine. It is used to define a game more consisely (compare to definition 2.1 and 2.3). It was introduced by Genesereth, Love, and Pell [10] and subsequently extended by Thielscher [27] to allow for games with imperfect information. This subsection adapts the definitions of the cited sources. It reviews the syntax, and gives possible definitions of 5-Nim and Phantom-5-Nim. It discusses how it is evaluated, and what the advantages of formulating games in GDL are.

The language is logic-based, making its syntax similar to first-order logic. A definition of a game consists of facts and rules. The facts build the base. They are relations that hold statically. A rule consists of a head, which is also a relation and a body. The head holds if the body— a conjunctive clause of relations, holds. Rules can depend on facts, or on other rules.

The language is a subset of Datalog, which itself is a subset of Prolog [2, 10, 14]. The syntax is commonly in infix (`head :- body(1), body(2).`) or prefix

((=> head ((body 1) (body 2)))) notation. This thesis uses the infix notation. Relations are denoted in the `name/arity` notation, where `name` is the name of the relation and `arity` the number of arguments of that relation.

Some relations are reserved to describe the components of the game. The relation `role/1` defines which entities can do actions. The state has three keywords associated with it, `init/1`, `true/1` and `next/1`. The initial state is described by `init/1`, while the current state is described by `true/1`. Valid GDL-definitions require `next/1` to only be used in the head of rules. It indicates which atoms become true in the next state. Frequently, these rules depend on `true/1` and `does/2` relations. The moves of the roles during a ply are considered facts and are added as `does/2` relations. The resulting `next/1` relations are the next state's `true/1` relations. The special `true(control/1)` relation indicates which players are in control. Accordingly, `next(control/1)` describes who is in control in the next ply, and `init(control/1)` describes which players are in control at the initial state.

The three relations `legal/2`, `terminal/0` and `goal/2`, are also reserved, however, unrelated to the next state. A move is legal, precisely if the first argument of the relation `legal/2` contains the role, and the second the action. The atom `terminal/0` is only true if the state is terminal. For these states, `goal/2` indicates the utility of a role, where the first argument is the role, and the second an integer value (by convention in the range of $0 - 100$).

In comparison to programming languages, GDL is low-level in scope. The scope explicitly excludes arithmetic operations. Should arithmetic operations be needed, they have to be defined via the rules. Mittelmann and Perrussel [18] proposed an extension called GdlZ, that expands GDL with integer arithmetic. The only higher level relation available is `distinct/2`. The relation is true if the two arguments do not unify (viz. are different).

Intentionally, GDL is a compact representation of definition 2.1. The notation is still transparent while allowing evaluation by an interpreter. Moreover, it is succinct, as all states do not need to be enumerated.

To illustrate, consider the game of 5-Nim (see definition 2.2). Listing 2.1 defines 5-Nim in GDL. The roles are defined in line 1. The initial state is defined by lines 3 and 4. The actions are enumerated in line 26 and the legal actions in lines 38 to 41. The state transition is handled in lines 28 to 36, where 28 to 31 defines that the moves are alternating and 33 to 36 the effects of taking from the heap. An empty heap leads to a terminal state, which is defined on line 43. The goal values for non-terminal states are defined in 45 to 47. The losing conditions are listed in 49 to 51, while the winning conditions are stated on lines 53-57. Lines 6 to 24 define the arithmetic needed for taking from the heap.

The style of programming in GDL is as opinionated as any other programming language. For example, in the lines 46, 50 and 54, it is possible to substitute `true(heap(0))` with `terminal`. The author of this thesis argues that it should not be, as the goal value does not actually depend on the completion of the game but the size of the heap. Another

```
role(first). role(second).

init(control(first)).
init(heap(5)).

succ(0,1). succ(1,2). succ(2,3).
succ(3,4). succ(4,5).

num(N) :- succ(N,N').
num(N') :- succ(N,N').

plus(S,0,S) :- num(A).
plus(A,1,S) :- succ(A,S).
plus(A,B',S') :-
  succ(B,B'),
  succ(S,S'),
  plus(A,B,S).
plus(B,A,S) :- plus(A,B,S).

le(A,A) :- num(A).
le(A,B) :- lt(A,B).

lt(A,B) :- succ(A,B).
lt(A,C) :- succ(B,C), lt(A,B).

action(1). action(2). action(3).

next(control(R)) :-
  true(control(R')),
  role(R), role(R'),
  distinct(R,R').

next(heap(N)) :-
  does(R,A),
  true(heap(N')),
  plus(N,A,N').

legal(R,A) :-
  true(heap(N)),
  role(R), action(A),
  le(A,N).

terminal :- true(heap(0)).

goal(R,0) :-
  not true(heap(0)),
  role(R).

goal(R,-1) :-
  true(heap(0)),
  true(control(R)).

goal(R,1) :-
  true(heap(0)),
  true(control(R')),
  role(R), role(R'),
  distinct(R,R').
```

Listing 2.1: A possible definition of 5-Nim using GDL.

example is the commonly seen redundant pattern to define that the legality of an action is conditional on the role being in control. The gamemaster only allows roles in control to submit a move, which makes the inclusion of the clause redunant.

The relation of GDL-definitions and logic programs leads to parallels in how they are evaluated. Two distinct approaches are possible, forward evaluation and evaluation via backtracking. Forward evaluation, or sometimes Herbrand interpretation, gathers a set of true relations, the facts. The set of true relations is expanded by all rules that are true based on the current true relations. This is done iteratively until a fixpoint is reached. Backtracking-based evaluation, starts from a relation, the query. All rules compatible with the query are checked. If a rule's head unifies with a query, it is compatible. A relation is supported if there is a compatible rule and the body of this rule is supported. The backtracking-evaluation picks an untried arbitrary rule, with a compatible head, and adds it to the query. If everything is properly supported, the initial query is true.

Should a contradiction occur, the query backtracks until the contradiction resolves. If a query cannot be supported, it is false.

Both forward and backtracking-based evaluation have relevant use cases. The performance is highly dependent on the game. Backtracking is usually better if only a few relations need to be checked, while it suffers compared to forward evaluation in the contrary case. Both can be abstracted and used interchangeably, depending on the usecase. In fact, successful agents decide which technique they use, based on the task [2, 22, 28].

Defining games in a formal language, allows them to be executed, reasoned about, and compared to each other. Executing (or playing) the game does no longer need a separate implementation, a GDL-interpreter can use the game's definition to do that. This generality comes at the cost of computational overhead. Another advantage of this representation of games is the formal reasoning that is possible. The definition can be checked if it satisfies certain requirements. For example, playability, which is the notion if there is a state where a role cannot move. Another example would be winability, it requires the existence of a series of moves that results in the winning condition of a role [11]. Similarly, the differences between games can be highlighted by the differences in their definitions. If a game is more difficult to play, its definition might include more relations and rules than a simpler example, or more legal moves are possible in a given state, which increases the size of the search space.

The formal description of GDL does not allow for games with elements of chance, such as dice throws or hidden information akin to the hand of other players in a card game. To address this shortcoming, Thielscher [27] proposed game description language with imperfect information (GDL-II) (sometimes *incomplete* instead of *imperfect*). Since then, Schiffel and Thielscher [20] expanded the GGP-protocol to accommodate the changes. GDL-II is evaluated just the same as standard GDL.

GDL-II has only one new relation, `sees/2` and a special role called `random`. A role perceives a relation, if the relation is the second argument of `sees/2` and the role is the first argument. This relation is the equivalent to the view relation v of definition 2.3. To incorporate elements of chance, the role `random` chooses arbitrarily from the possible moves.

Listing 2.2 defines the additional (or changed) rules for Phantom-5-Nim.

The definitions for the two games share most of their rules. Lines 4 to 16 define that if a role takes more than is available on the heap, they are still in control in the next round, otherwise the control alternates. Lines 20 to 23 specifies that taking more than is available from the heap leads to an empty heap. Lines 25 and 26 define which roles see who is in control (if they are in control). Finally, line 28 expands the legal actions to all actions.

```
% Lines 1-26 from 5-Nim

next(control(R)) :-
  does(R',A),
  true(control(R')),
  true(heap(N)),
  role(R), role(R'),
  distinct(R,R')
  le(A,N).

next(control(R)) :-
  does(R,A),
  true(control(R)),
  true(heap(N)),
  lt(N,A).

% Lines 33-36 from 5-Nim

next(heap(0)) :-
  does(R,A),
  true(heap(N)),
  lt(N,A).

sees(Everyone,control(R)) :-
  role(R), role(Everyone).

legal(R,A) :- role(R), action(A).

% Lines 43-57 from 5-Nim
```

Listing 2.2: A possible definition of Phantom-5-Nim using GDL.

2.2 Games as Trees

Games can be viewed as trees. This representation lends itself to using tree search methods. Tree search methods are studied in many fields of computer science. This section discusses trees in the context of perfect information games (see subsection 2.2.1) imperfect information games (see subsection 2.2.2) and a common pitfall of tree search methods called the *horizon effect* (see subsection 2.2.3).

2.2.1 Perfect Information Games

The following paragraphs review how trees can be used to define games and how they are used to find the best move. As complete search is infeasible for large enough games, this subsection also discusses remedies for this limitation, and quantifies a good agent.

The nodes of a perfect information game's tree, hold the state and the edges are labeled with the turns. The tree is rooted in the initial state. From this root node, each possible move leads to a child node holding the resulting state. The leafs of the game tree hold the terminal states. This encodes definition 2.1. For an example of a game tree for the game 5-Nim, see fig. 2.3.

The tree may be used by an agent to calculate the best move. The utility of a move is equivalent to the role's utility of the resulting state. Usually, only terminal states have a defined utility. For non-terminal states, the utility is equivalent to the value of the best move. Crucially, the value of a move depends on which player is in control. Furthermore, if a tree is too large, enumerating all non-terminal states might be infeasible. In these cases, heuristics can be used to approximate the utility.

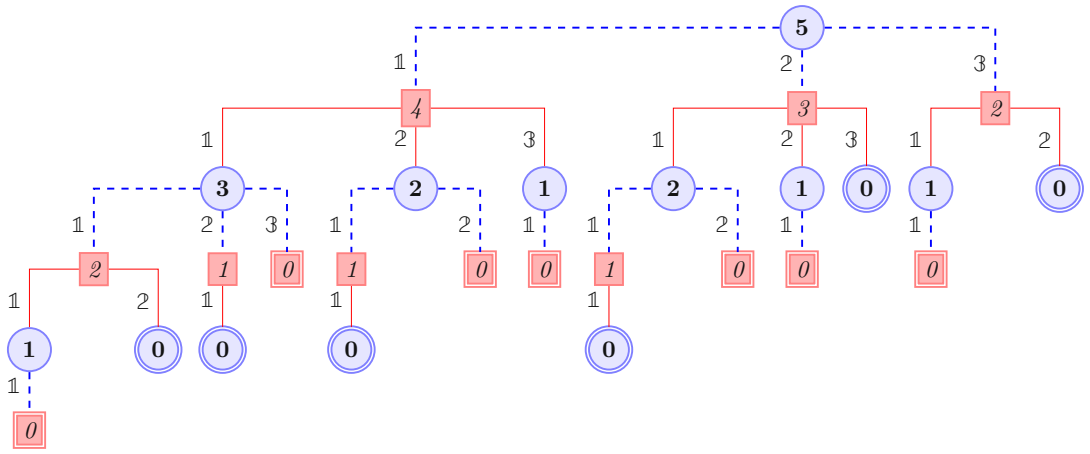

Figure 2.3: The game tree of 5-Nim. The round and blue nodes denote states where **first** is in control and rectangular and red nodes denote states where *second* is in control. The top-most node is the initial state and the states with a double border are terminal states. The edges represent the moves. Fat, blue and dashed edges are actions done by **first**. Thin and red edges are actions done by *second*. The label of the edge represents the arithmetic operation on the heap.

In a match of a game with perfect information, every player views the same game tree. Each player may use this tree to find the best move. However, for most games, searching for the optimal move becomes akin to a search for the needle in the haystack, as game trees are prohibitively large for non-trivial games. Each state has at least one node, and each possible move results in a branch. There are too many nodes and branches to consider for a complete search.

To remedy this limitation, agents build their own tree, called a search tree. Starting from the root node, the tree is built by expanding nodes. The expansion is done by calculating every possible follow-up state from the combinations of legal moves. The states correspond to the nodes and the moves correspond to the edges. Step-by-step, a subtree of the game tree is built.

A good agent does not only consider more nodes (viz. handles a larger search-tree), but is also cautious about what parts of the tree are considered. Traditionally, this is done via heuristics, handcrafted by human-expert knowledge. Recently, these were replaced by better performing neural networks. They are trained by reinforcement learning, where the data is generated via self-play [26].

After a ply, the agents never need to ascend the game tree again. The tree is rerooted after each ply to the current node. As the current node holds all the available information, nothing is lost by pruning the tree above the current ply level.

2.2.2 Imperfect Information Games

The trees for perfect and imperfect information games fundamentally differ. The hidden information introduces the knowledge of a state. Encoding knowledge into the tree complicates the data structure. This subsection introduces the difference between perfect and imperfect information game trees, using the running example of Nim. Furthermore, it discusses the theoretical underpinnings for these differences and how this affects agents playing against each other.

For games with imperfect information, agents can no longer accurately determine in which branch of the game tree they are. They cannot observe their opponent's moves, and they do not view the full state, therefore, multiple nodes (and by extension states) need to be considered. Their states are indistinguishable for this player. See this visualized for the game Phantom-5-Nim in fig. 2.4.

A set of states that is indistinguishable by a player is called an *information set*. This concept forms the basis of the search tree for games with imperfect information. Depending on the assumed state, a move of a player might have different effects. Therefore, a single move might point to different information sets. To differentiate, they are only ever meaningful together with the assumed state.

In imperfect information games, multiple different histories are possible. By contrast, in perfect information games, there is only one history. The current state can be reconstructed by applying the moves in order. In imperfect information games, moves are only partially observable, and only a partial state is visible. Therefore, multiple states may be consistent with the available information.

Histories implicitly describe information sets [23]. An information set is the set of all states of all indistinguishable histories. Histories are important for navigating the search tree. They describe which branches need to be taken to reach the current node of the tree.

Searching games with hidden information requires a new kind of data structure for search trees. The search tree's nodes no longer hold one single state, but a set of indistinguishable possible states— the information set. The edges are labeled with the assumed state and the moves, as the effect of a turn can differ depending on the actual state.

In the case some players have different views, their respective search tree differs, contrary to perfect information games, where full knowledge is assumed. In perfect information games, a node holds the actual state. While, in imperfect information games, the actual state is some state of the intersection of each player's information set.

2.2.3 Horizon Effect in Games

Tree search algorithms are susceptible to problems, that surface as anomalies in playing strength. One such difficulty is known as the *horizon effect*. This subsection reviews

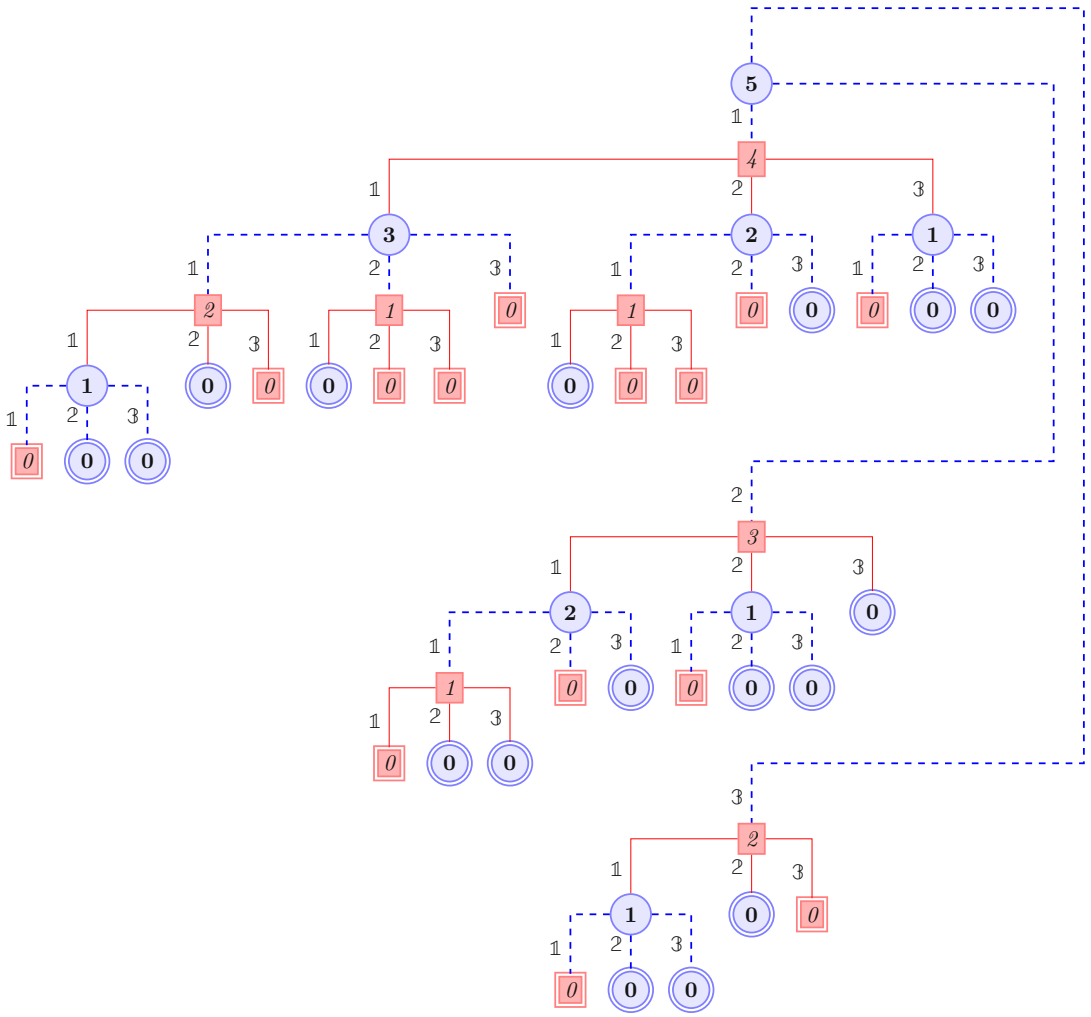

Figure 2.4: The game tree of Phantom-5-Nim. The round and blue nodes denote states where **first** is in control and rectangular and red nodes denote states where *second* is in control. The top-most node is the initial state and the states with a double border are terminal states. The edges represent the moves. Fat, blue and dashed edges are actions done by **first**. Thin and red edges are actions done by *second*. The label of the edge represents the arithmetic operation on the heap.

what the horizon effect is, where the term was coined, and discusses why it is an issue to keep in mind for games with imperfect information.

The search tree is only a part of the full game tree. At best, it is still sufficient for finding the optimal move, in the worst case, it produces inaccurate results. These inaccurate results are caused by the missing information the search is blind to. In some games, these blind spots can be mitigated by strong heuristics. Still, however accurate the heuristic is, it is an approximation of the actual utility.

Naturally, an agent using a search tree (that is partial in respect to the full game tree) has a *horizon*. Whatever lies beyond this horizon— a good move, a trap laid by the opponent, might ultimately be opaque for the agent. This phenomenon was first named by Berliner [1], and is called the *horizon effect*.

Berliner outlines a negative and a positive horizon effect. Even though the naming suggests that sometimes the effect is beneficial for playing strength, it usually is not. The negative horizon effect broadly describes cases where a bad move initially appears deceptively strong. The move has high heuristic value but due to some kind of downside that typically only appears after some plies, it is of low utility to the player. In contrast, the positive horizon effect includes moves that have an upside, that after searching deeper, reveals an alternative with even greater upside.

While the initial definition of the horizon effect was meant for Chess programs, it describes a phenomenon found in many fields of artificial intelligence. If the search depth is limited, unpredictable discrepancies between the actual utility and a heuristic can occur.

The author of this thesis argues that games with imperfect information and high branching factors have a greater tendency to suffer from the horizon effect. While it has not been a major focus of previous work, the literature generally agrees [5]. One possible explanation is that the heuristic has even less information to base their evaluation on, and anything beyond the horizon may be hidden among an exponential number of possible states.

2.3 Monte Carlo Tree Search

The Monte Carlo tree search (MCTS)-algorithm is a procedure to efficiently generate a search tree from the rules of a game. It is used for games, and other fields such as planning, Chemical synthesis and Security [26]. Due to properties of the algorithm, it is particularly interesting for GGP. Namely, it does not need a heuristic and only needs the rules of a game to be applicable. In the following subsections, the author of this thesis reviews the core algorithm and four extensions for games with imperfect information. Subsection 2.3.1 gives the principles of operation for all MCTS variants. Determinizing MCTS is the first discussed simple extension for imperfect information games (see subsection 2.3.2). Cowling, Powley, and Whitehouse [8] introduce three variants of information set Monte Carlo tree search (ISMCTS). The first of those three is reviewed in subsection 2.3.3, the second in subsection 2.3.4, and the last in subsection 2.3.5.

2.3.1 Principle of Operation

While there are many resources [4, 26] covering the details of MCTS and its modifications, the implementation of the core principles differ from author to author. This subsection discusses the most important aspects for implementing MCTS variants and how to configure it for different kinds of games. It spans the four characteristic phases, and goes into detail for UCT— the primary algorithm for the selection phase. Lastly, it discusses the susceptibility towards the horizon effect.

MCTS is divided into the four phases, selection, expansion, simulation and backpropagation. During selection, the procedure walks down the tree until a leaf node is met. The children of this node are linked in the expansion phase, turning the leaf into a branch. The simulation phase evaluates the node, usually by simulating random plays starting from the current state. By propagating upwards until the root is reached, the backpropagation phase updates the tree, thus completing one step of the procedure.

The goal of the selection phase is to determine, which of the available leaf nodes is the most worthwhile to expand. A leaf node is considered worthwhile to expand, either if the state of the node has a high valuation or if there is yet high uncertainty about the state. How these are quantified is dependent on configuration. Selecting nodes with high valuation is called *exploitation*, while choosing nodes with high uncertainty is considered *exploration*.

The procedure starts at the root node and observes all children. If the current node is a leaf node, the selection phase ends. Otherwise, each child is assigned a weight. Considering these weights, a selector decides on a node. Then the procedure iteratively repeats until a leaf node is reached.

Depending on the game, the selector and the assignment of the weights may be configured, which in turn configures the algorithm's preference for exploitation or exploration. Simple selectors include minimum or maximum selectors that always picks the node with the lowest or highest weight, or random selectors where nodes with higher weight have a higher probability of selection [7]. A common way to assign weights is the UCT algorithm introduced by Kocsis and Szepesvári [13].

The UCT algorithm quantifies the degree of exploration and exploitation, by assigning a value u to each node that depends on the valuation and the number of times it has been valuated in comparison to its siblings and selecting the node with the highest score [13, 26]. The value for u is given by evaluating the function ucb

$$\mathrm{ucb}(n) = Q(n) + C \cdot \sqrt{\frac{\ln N(n_{\mathrm{pred}})}{N(n)}}$$

where $Q(n)$ is the valuation of the node n, C is the exploration weight— usually decided empirically, $N(n)$ is the number of iterations done in n and n_{pred} denotes the parent of node n (see also [26, 6, eq. 2]).

In the expansion phase, the procedure creates new nodes from the selected leaf node's state and links them as children. Thereby, transforming the leaf node to a branch node. Due to the selection phase, the expansion of the tree is asymmetric. If the weights and selector of the algorithm prefer exploration, the resulting tree is more shallow and symmetric, where a high preference for exploitation produces deep and asymmetric trees.

During the simulation phase, the algorithm repeatedly applies random turns to the state of a node until a terminal state is reached. The terminal state's utility is then used as a valuation for the initially selected node. While a single simulation might not be meaningful, the algorithm repeatedly simulates, thus increasing confidence.

Using simulations as valuations for the nodes divorces the search algorithm from the need for a heuristic. Heuristics usually estimate the utility for a given state, while simulations quantify the possibilities of a win. If a state has a high valuation, there exist enough sequences of moves that result in a win, which indicates a high utility.

In the backpropagation phase, the procedure updates the tree with the information gained during the simulation phase. It starts at the node chosen during the selection phase, and uses the link with the parent to walk up the tree. Each node is updated until the current root is reached. At this point, one iteration of the procedure is completed.

While MCTS is not immune to the horizon effect (see Section 2.2.3), it is far less susceptible to it. The horizon effect is more likely to occur if the heuristic valuation and the actual utility of a node differ greatly [1]. As MCTS uses simulations, the heuristic is gradually improved, mitigating the horizon effect.

2.3.2 Determinizing Monte Carlo Tree Search

On its own, MCTS cannot handle imperfect information games. It assumes the search to be on a tree with a state per node and a move per edge. The agent, however, has to handle a multi-edge search tree with multiple possible states in each node and a state-turn pair as edge label in imperfect information games. A simple extension to handle this shortcoming is determinizing MCTS. This subsection conceptually reviews the algorithm, why it ultimately is not suited for imperfect information games in the context of GGP, and why it is still a considerable alternative for certain games.

Determinizing MCTS dissects an imperfect information game tree into several perfect information game trees. Starting from the root node, each possible state is assigned as separate tree. Before an iteration, one of those trees is selected. The results of each tree are then aggregated. The move with the most visits over all trees is chosen as the selected move.

As alluringly simple this procedure is, it suffers from a grave weakness, referred to in the literature as *strategy fusion* [8]. Strategy fusion describes the phenomenon that averaging over all possibilities leads to weak playing performance. The weakness surfaces, when the optimal strategy depends on the fact that the hidden information cannot be known.

To illustrate, consider the game Mini-Single-Call. Mini-Single-Call is a single-player game with randomness and hidden information. First, the player has to decide whether they want to place a bet or fold. If they fold, the utility is +5. If the player bets, they are dealt a card with one of two equally likely suits, red or black. Without looking at the card, the caller must decide if they call red or black. Both moves end the game. If the player calls the same suite as the dealt card, they receive a utility of +10, if they guess wrongly −10. See fig. 2.5 as an illustration for a game tree.

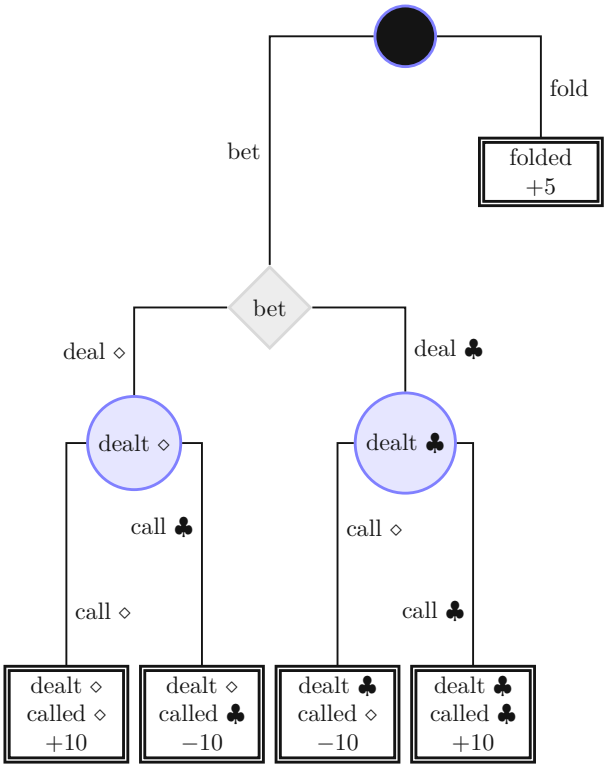

Figure 2.5: The game tree of Mini-Single-Call.

The best move is clearly not placing the bet at the beginning, as this results in a sure +5 utility. Placing a bet on the other hand forces the outcome to be either +10 half of the time and −10 the other half of the time. The expected utility is 0.

Nonetheless, determinizing MCTS would select to bet, due to strategy fusion. Strategy fusion occurs because the two determinized trees have the optimal strategy of betting and then selecting the correct call. When combining the strategies, the branch of betting seems to be promising a utility of +10. However, as the agent cannot see what card has been dealt, it is unable to call the correct suit with certainty; therefore the strategy is suboptimal.

Experiments indicate that determinizing MCTS is still competitive in games where strategy fusion is not a major factor [8]. Determining if a game susceptible to strategy fusion

is difficult to gauge a-priori. There are methods of indirectly measuring it [16], but this is impractical for non-trivial games.

A major benefit of determinizing MCTS is the relatively low runtime cost. In comparison to more sophisticated algorithms (see the subsequent sections) Cowling, Powley, and Whitehouse [8] report up to more than double the number of iterations per time frame for certain games. If strategy fusion is not a factor, this may result in better playing performances if the calculation time is fixed.

2.3.3 Single Observer-Information Set Monte Carlo Tree Search

Single observer-information set Monte Carlo tree search (SO-ISMCTS) is part of the family of algorithms referred to as ISMCTS [8]. These algorithms abstract the ideas of MCTS to build information set trees, instead of game trees. This subsection reviews the changes required to the core MCTS algorithm, namely how the four phases are affected. It then reviews the advantages and drawbacks, on the topics of playing strength and runtime cost.

Instead of splitting the information set's states into the roots of multiple trees, SO-ISMCTS directly samples the possible states. Starting at the root, the procedure selects one possible state and only considers edges of the multi-edge tree, where the label is compatible with the selected state.

A node no longer only has a single state which can be used to valuate it. There are multiple ways of dealing with this. Cowling, Powley, and Whitehouse [8] suggest determinizing the tree at the root note before each iteration. In this way, the four phases of MCTS do not need to be altered conceptually.

The agent in control defeminizes the tree by selecting one of the possible states of the root node. From the root, the edges labeled with pairs of states and moves lead to the children. Every edge, where the label does not contain this determinization, is incompatible. Those children with a compatible in-going edge are considered. The selection process then continues as in the perfect information MCTS.

The selected node is then expanded. In perfect information MCTS this is straight-forward, as each node only contained a single possible state, the edges corresponded to the turns and each follow-up state was assigned its child node. The expansion in the new search tree for imperfect information games is different. As the information set may contain a prohibitively large number of possible states, it is not tractable to enumerate them all and consider each state. Instead, only the determinized state is used. When expanding a node via a determinized state, existing children are either populated or new nodes are created. A node may be a leaf node in one determinization, but not in another.

The determinization approach ensures that after selecting the appropriate leaf node, expansion, simulation and backpropagation are done as in perfect information MCTS.

While SO-ISMCTS improves the drawback of not being directly applicable to imperfect information games, it still suffers from strategy fusion. Fundamentally, SO-ISMCTS does

not differ greatly from determinizing MCTS. The difference is the accumulation of data. Determinizing MCTS first splits the information sets into perfect information trees and later recombines the resulting search trees, while SO-ISMCTS directly operates on the information set tree. They both produce similar results, however SO-ISMCTS is more efficient in allocating iterations.

In the experiments done by Cowling, Powley, and Whitehouse [8], SO-ISMCTS is comparable to determinized MCTS. In the same number of iterations, it is slightly more competitive. Its obvious downside is the weak opponent model, which is why it is weaker than multiple observer-information set Monte Carlo tree search (MO-ISMCTS) (see subsection 2.3.5).

2.3.4 Single Observer-Information Set Monte Carlo Tree Search with Partially Observable Moves

Single observer-information set Monte Carlo tree search with partially observable moves (SO-ISMCTS+POM) is a evolution of the ISMCTS variants introduced by Cowling, Powley, and Whitehouse [8]. This subsection discusses the differences of SO-ISMCTS+POM and SO-ISMCTS. Furthermore, it lists the advantages and disadvantages in comparison to other reviewed MCTS variants, including the performance considerations.

The continuation SO-ISMCTS+POM alleviates some shortcomings of its predecessors. It no longer differentiates between moves that result in outcomes with the same view. Thereby, removing the assumption that the opponents moves are fully observable. It no longer suffers from strategy fusion, while significantly weakening the model of the opponents.

To accurately model partially observable moves, the search tree is extended by distinguishing between two kinds of nodes, visible and hidden nodes. Visible nodes represent plies where the observer is in control, and is updated with the current view of the game state. Because the observer can distinguish its moves, the edges can be differentiated. On the other hand, are hidden nodes, where it is opaque to the observer which moves were made.

The player in control observes the current partial game state and determinizes from the possible states. Each incompatible edge remains unconsidered. One of the children is selected. Should this child be a visible node, the selection procedure continues in the same way. If this child is a hidden node, the selection procedure randomly selects a compatible edge.

Expansion, simulation and backpropagation are done as in SO-ISMCTS and perfect information MCTS. Expanding the in the final step selected leaf node is done lazily by the determinization. Simulation is carried out by random playouts. Backpropagation aggregates the already present data and amends it with the new result of the simulation.

The advantages of SO-ISMCTS+POM over SO-ISMCTS are minimal. Their playing strength is similar, where SO-ISMCTS+POM has a tendency to be slightly less competitive at the

same number of iterations [8]. In games where there is limited interaction between opponents, the playing strength should be closer to MO-ISMCTS (see subsection 2.3.5).

2.3.5 Multiple Observer-Information Set Monte Carlo Tree Search

The last of three ISMCTS algorithms introduced by Cowling, Powley, and Whitehouse [8] is multiple observer-information set Monte Carlo tree search (MO-ISMCTS). MO-ISMCTS solves both the strategy fusion problem of determinizing MCTS and SO-ISMCTS, as well as the weak opponent model of SO-ISMCTS+POM. This subsection highlights the differences to the other variants of ISMCTS and reviews the required extensions to the four phases of MCTS. Lastly, it discusses the performance in comparison to other three reviewed MCTS variants for imperfect information.

MO-ISMCTS uses an independent search tree, for each observer in the game. Wherever SO-ISMCTS+POM would assume a random move in the selection phase, MO-ISMCTS instead consults the respective tree, informing the decision. As a consequence, the models of the opponents are significantly stronger, at the cost of higher complexity.

The phases of selection, expansion and backpropagation need to be carried out concurrently on all trees, while simulation can be used for all trees. Where the previous iterations and perfect information MCTS implicitly chose moves by successive selection of favorable states, MO-ISMCTS constructs turns from the role's moves, ply by ply. Expansion is still done on a specific determinization. Each tree grows lazily. The utility is backpropagated in each tree depending on the role.

In the selection phase, at the start of the iteration, a possible state is determinized by selecting one from the information set of the current root. This determinization may be in many indistinguishable histories and seperate branches for each tree, therefore all non-main trees need to be rerooted. Rerooting requires backtracking up the tree until the last known state is reached and generating a development that is compatible with this last known state and the determinization.

After all trees are rooted in the common determinization, the algorithm queries which roles are in control. Using UCT the most eligible child node for each tree is determined. Potentially, many edges point to that child. By arbitrarily choosing an edge, the procedure selects a move for that role. All those moves are collected and combined into a turn. The turn is used to progress the determinized state. In each tree, the edge that is compatible with the respective move and the progressed state is used to select the next node.

After iterating enough times until a leaf node in respect to the determinized state in the main tree was found, the algorithm transitions into the expansion phase. In the expansion phase, for every follow-up state of the determinized state is either added to the information set of an already existing node, or it is the basis of a new child node. This needs to be done for every tree.

Simulation via random playouts can be done just as in perfect information MCTS. After it reaches the terminal state, the utility for every role is extracted (instead of only the actor's role).

The utility of each role is backpropagated in each tree. This ensures that the moves in the selection phase mimic those of strategic opponents.

In the experiments done by Cowling, Powley, and Whitehouse [8], MO-ISMCTS is the best option in the family of ISMCTS. While its runtime performance is comparatively poor, the stronger opponent model and the ability to search the tree deeply without strategy fusion offsets its limitations. In some games, the margin is bigger than others, depending on branching factor, susceptibility to strategy fusion and player interaction.

CHAPTER 3

Methods

Autonomous play of imperfect information games in the context of general game playing (GGP) remains insufficiently explored. A promising family of methods are the Monte Carlo tree search (MCTS)-algorithms. One objective of this body of research was to survey the literature on MCTS, and gather the most important variants for imperfect information games, and GGP. Additionally, this thesis describes a new method in this family, based on the found variants. When applying MCTS, there are multiple configuration options available, to accommodate to different use cases. Using methods found in the literature, the author of this thesis suggests ways of configuring the proposed method, similar to the known options. To evaluate the method, this thesis describes a tournament, and reports on a quantitative experiment designed according to this tournament. A dependency of the tournament is an engine to run games in, and agents to play those games. The author of this thesis designed, and wrote an engine, and agents informed by similar programs described by the literature.

Section 3.1 reviews the overall scientific approach, of these objectives. The methodology of the comparison of existing MCTS modifications is discussed in section 3.2. In section 3.3 the focus lies on the path towards configuring the proposed method. Section 3.4 offers the details of the design of the experiment. Finally, chapter 4 is devoted to the implementation of the engine, and the agents.

3.1 Scientific Approach

The thesis had three research questions. First: Which of the modifications for MCTS are effective for playing imperfect information games in a GGP context? Second: Which features can be extracted from descriptions of imperfect information games to construct a heuristic to set the maximum expansion depth for the MCTS algorithm? Third: How does a MCTS-agent with limited expansion depth compare in benchmarks?

The first research question, led to a literature research, and to a comparative and qualitative evaluation of two concrete games. The literature research started with the extensive survey papers by Browne et al. [4] and Świechowski et al. [26]. Both papers aimed to gather and review all available methods, modifications, and applications. From there, the most relevant publications on the topics of imperfect information games and GGP became the subject of the literature review. After selecting two games for the experiment (see section 3.4), the author of this thesis used those games for applying the methods described in the publications, and comparing them qualitatively.

The survey of the chosen literature focused on the approach that allowed to play imperfect information games, and if there are any specific optimizations that ruled the usage in a GGP context. More specifically for playing imperfect information games, check how the method mitigates strategy fusion (see subsection 2.3.2), if there have been experiments comparing to other methods, and if there is code available. Some methods are not usable in a GGP context. These were identified by investigating if the variant relies on a game specific optimization, and machine learning techniques.

The most promising methods were qualitatively compared using the two example games `Phantom_Connect(4,4,4)` and `Dark_Split_Corridor(3,4)`. The two games were also used for the experiment. The author of this thesis chose them for the qualitative analysis, because of two reasons. Firstly, they were small enough for manual analysis, but still rich enough in complexity such that they were not trivial. Secondly, after having implementing them, he was intimately familiar with the games, and thus being able to quickly falsify hypotheses by running examples.

The methodology of the qualitative analysis itself, was to use a characteristic state of each game, then analyze it manually, and finally, infer how the MCTS variants would evaluate the game state. The variants considered were determinizing MCTS (see subsection 2.3.2), single observer-information set Monte Carlo tree search (SO-ISMCTS) (see subsection 2.3.3), single observer-information set Monte Carlo tree search with partially observable moves (SO-ISMCTS+POM) (see subsection 2.3.4), and multiple observer-information set Monte Carlo tree search (MO-ISMCTS) (see subsection 2.3.5). Focusing on specific games loses generality of analysis, but greatly simplifies the scope, while being relatively insightful. Furthermore, it may serve as a stepping stone for more general analysis in future work.

When configuring the devised method of limiting the expansion depth for MO-ISMCTS, the choice for the concrete depth is subject to a range of parameters. The author of this thesis proposes a set of parameters and methods, based on the approaches by Kuhlmann and Stone [15], Schiffel and Thielscher [21], Clune [6], and Mańdziuk and Świechowski [17].

The experiment was closely modelled after the quantitative evaluation done by Cowling, Powley, and Whitehouse [8]. It is a double-round-robin tournament including self-play with three contestants, a clairvoyant MCTS-agent, an agent using unlimited depth-MO-ISMCTS, and a limited depth-MO-ISMCTS agent. Each contestant played as

the first-moving player and the second-moving player, against each other contestant, including themselves.

The design process of the engine followed an iterative prototype model. Starting from the elementary data structures, the author aimed to identify the minimal required changes, such that the features required to conduct the experiment. Before implementing, adding unit test cases, whenever possible. This is also known as test-driven development.

A goal during the implementation was to reuse as much of existing libraries and projects as possible to minimize how much code the author had to write, and thus start the experiment as early as possible. However, due to the scope of the project, the code base was still sizeable. Naturally, while implementing one part of the engine, insights accrued during this process, revealed design mistakes, and better alternatives for implementation details. Thus, a balance was to be kept, between iteratively improving, and striving to finish.

3.2 Comparison of Existing Modifications

Four variants of MCTS (see section 2.3) are subject to comparison. These are the suggested primary methods for playing imperfect information games [26]. Naturally, their relative playing strengths differs from game to game. To compare the playing strength, the author of this thesis used a method of qualitative evaluation, adapted from previous work [8].

The survey paper by Świechowski et al. [26] identified improvements for MCTS used for playing imperfect information games. The primary suggested methods were determinizing MCTS, and three variants of information set Monte Carlo tree search (ISMCTS). All of them were extensively covered in the publication by Cowling, Powley, and Whitehouse [8]. Furthermore, the paper features several experiments directly comparing the methods.

The authors use a small example game to showcase how the different the MCTS variants work. In the showcase, they chose a particular state, and visualized the search tree for every algorithm. As a consequence, Cowling, Powley, and Whitehouse [8] highlight the differences between the search trees, and emphasize the differences in expected outcome.

For this thesis, the author expanded upon the idea of directly comparing the algorithms with specific examples. Specifically, by considering more complex games. The intended outcome was a richer analysis of the differences, losing simplicity. However, the goal was to compare, and not instruct on how the algorithms work.

The chosen games were `Phantom_Connect(4,4,4)`, and `Dark_Split_Corridor(3,4)`. These were the games used for the experiment. Thus, technical feasibility was the main consideration. Nonetheless, among the choices of games, the author selected the games for additional reasons. For a detailed explanation of the games see subsection subsection 5.1.1 and subsection 5.1.2.

By choosing `Phantom_Connect(4,4,4)`, the results of the experiment could be compared directly to those by Cowling, Powley, and Whitehouse [8]. The game is a hidden information version of `Connect` that is well understood, while not being trivial.

The author picked `Dark_Split_Corridor(3,4)` because it features a comparatively wide decision tree, with a complex state, while still being technically feasible. Furthermore, the moves are partially visible, leading to indirect player interaction, and incentivizing knowledge management. These properties indicated an interesting game to analyze the algorithms on.

The choices for characteristic game states, follow from the histories produced by the experiment. The matches for `Phantom_Connect(4,4,4)` ran chronologically before those of `Dark_Split_Corridor(3,4)`. Unfortunately, 53 of the 450 matches did not record the history in an easily extractable format. Thus, only 397 histories were available for `Phantom_Connect(4,4,4)` for analysis. For `Dark_Split_Corridor(3,4)` this mistake was rectified, and the analysis included all 450 histories.

The states were produced by methodical selection of actions, informed by the number of occurrences in all histories. The number of occurrences is aggregated further upon the outcome of the match. From this follows for each action a count for all histories for each ply and for each outcome (first player wins, first player loses, or a tie).

The selection of the moves corresponds to (one of) the most often used move, that ultimately leads to the most wins for the role in control. This process of selection results in a history. The history is cut off one ply before the shortest recorded history. The shortest recorded history for `Phantom_Connect(4,4,4)` was 7 plies long. The shortest recorded history for `Dark_Split_Corridor(3,4)` was 5 plies long. The last state of the generated history is the characteristic state for that game.

3.3 Feature Extraction for Setting the UCT-Border

The proposed modification of the MCTS is characterized by limiting the maximum expansion depth. The maximum expansion depth is also called the UCT-border. While there was no formal analysis on the effect of where exactly the UCT-border is, there is prior work on the feature extraction in the context of GGP. These features ideally inform the decision for setting the maximum expansion depth. Furthermore, after sufficient quantitative evidence, this may also pave the way towards a formal investigation in future work.

The literature includes two approaches, static analysis (sometimes also offline analysis), and dynamic sampling (sometime also online analysis). The former is discussed in Schiffel and Thielscher [21], and Kuhlmann and Stone [15]. The latter is featured in the publications by Clune [6], and Mańdziuk and Świechowski [17].

Static analysis concentrates on the static information of a game, which may be extracted via the description. These features are characteristic for a game, and indicative of

properties relevant to search. For example, a game with a narrow tree, may only have a small set of actions. As another example, serves a game with a high number of atoms. This feature may indicate a complicated state. A more thorough analysis yields more expressive results. When resources are limited, such as a competitive environment, the expense may not be justifiable.

Dynamic sampling – another approach – includes sampling states, and doing statistical analysis upon the recorded states. This is less general, but nonetheless useful in most cases. For example, counting the number of possible states, given a role's perspective. This indicates how much information is actually hidden from this role.

For the value of the maximum expansion depth, two static-, and two dynamic features were considered. The static features were the size of the state space, and the size of the action space. The size of the state space may be given an upper bound by counting the atoms defined by the game description language (GDL) description. The state space is less or equal to the powerset of atoms. The tightness of the upper-bound varies, and can be improved by incorporating rules. The size of the action space is the number of actions defined by the GDL description. The dynamic features were the mean arity (viz. number of branches of a node) of the game tree, and the average distance to the terminal nodes.

To determine the state- and the action space's sizes, ground the logic program described by the GDL description. The set of relations that are arguments of `init/1`, `true/1`, and `next/1`, form the state space. The set of relations that are the second argument of `legal/2` constitute the action space.

The mean arity of the game tree is approximated by generating random histories, and counting the number of possible turns at each ply. Starting from the initial state, apply random turns to the current state until reaching a terminal state. During each step, record previously unseen non-terminal states into a set. For each state in the set, calculate the number of unique turns.

Similarly, the average depth is calculated by sampling closed histories, and counting their length. Starting from the initial state until reaching a terminal state corresponds to a path from the root- to a terminal node in the game tree. Thus, the number of plies is equivalent to the depth.

3.4 Experiment Setup

The conducted experiment was a tournament, playing two games, with the implemented agents as contestants. The tournament was double-round-robin with self-play. Self-play includes games where two instances of the same agent play against each other. The selected games were `Phantom_Connect(4,4,4)`, and `Dark_Split_Corridor(3,4)`. Both games are alternating two player zero-sum games with hidden information. Each agent played fifty games as first moving-, and fifty games as second moving role, against each contestant. The number of MCTS iterations were fixed, with an unlimited game clock time, to remove any influence of implementation details. The contestants were a

clairvoyant MCTS agent, a MO-ISMCTS agent without a limited expansion depth, and an agent using MO-ISMCTS with a fixed expansion depth. The design of the tournament was informed by Cowling, Powley, and Whitehouse [8].

The mode of double-round-robin with self-play, ensures an equal number of matches for each agent, and for each role. By balancing the number of matches, statistical comparison is more representative. Alternatives to double-round-robin tournaments include elimination tournaments. Elimination tournaments are well suited for finding the best agent among the contestants, using a smaller number of games. However, the smaller number of games makes it more difficult to analyze why the agent is better.

The author of this thesis chose `Phantom_Connect(4,4,4)` as the first game because it was also used as a game in the tournament conducted by Cowling, Powley, and Whitehouse [8]. The conclusions of the paper, served as a point of reference, and verification. Additionally, the small state- and action space, classify it is a computationally tractable game.

The second game `Dark_Split_Corridor(3,4)`, is a computationally more intense game. Furthermore, there seems to be less available literature on it. In contrast to the first game, it does not feature ties, has a more complicated game state, and features a more immediate player interaction.

The number of matches played was fifty per game, per role, and per agent. In total, for each game this equates to 450 matches, and 250 matches per agent (not double counting the games they played against themselves). The number of games was limited by the high computational load, and consequentially the required time to run the experiment.

The number of iterations was fixed, while there was no time limit per ply. This enabled the possibility to run the experiment distributely on multiple machines, while keeping the results representative.

For both games, the number of MCTS iterations, were chosen as high as possible, while keeping the total runtime feasible. With `Phantom_Connect(4,4,4)` the agents applied 7500 iterations of MCTS per ply. `Dark_Split_Corridor(3,4)` due to its more complex state, and the resulting longer individual iterations, was reduced to 2500 iterations.

The implemented agents were a clairvoyant (viz. cheating) MCTS agent, an unlimited-, and limited MO-ISMCTS agent. The clairvoyant agent plays with the full game state in view, instead of a partial view. Therefore, there is no information hidden to it. Obviously, this leads to a significant advantage. The clairvoyant agent serves as an upper bound of playing strength. Furthermore, an agent winning, tieing, or losing more often against the clairvoyant agent is an indication of an advantage or disadvantage in playing strength. The unlimited agent implemented MO-ISMCTS, and would be directly compared to the limited counterpart. The limited agent had a set expansion depth of 4 for `Phantom_Connect(4,4,4)`, and 5 for `Dark_Split_Corridor(3,4)`.

The expansion depth depended on the static analysis, dynamic sampling, and most importantly on the histories of the matches conducted by the unlimited agent. These

matches were conducted chronologically before the limited MO-ISMCTS agent. The histories showed that the values chosen resulted in lower maximum expansion depth, while not being too restricted.

CHAPTER 4

Implementation

The implementation consists of three modules, the engine, the tree (or node) data structure, and the agents. The engine is the foundational layer of the implementation. It covers the elementary data structures, the game description language (GDL)-interpreter and the match orchestration. The tree data structure is a representation of the dynamic search tree discussed in section 2.2. The implementation of the three available agents, apply Monte Carlo tree search (MCTS) onto the tree data structure.

The programming language of choice was Python with type-hints. The targeted versions included 3.8 up to 3.12. The choice of Python was based on the readability, usage, and because it's interpreted rather than compiled, ease of debugging, and introspection. Furthermore, by using libraries for solving logic programs, writing parsers, and orchestrating multithreading, the scope of the project was more manageable.

4.1 Engine

The engine covers three parts, the elementary data structures, the interpreter, and the match orchestration. The elementary data structures are modeled after the definition of GDL. The interpreter uses the elementary data structures to receive and answer requests. The match orchestration module uses the interpreter to conduct matches. This section briefly summarizes the most important details for the implementation of the elementary data structures, the interpreter, and the match orchestration.

4.1.1 Elementary Data Structures

The elementary data structures are organized into a hierarchy of classes (see fig. 4.1), which closely follow the definitions of GDL. `Symbol` forms the bottom of the hierarchy. It is a disjoint union of `Primitive`, and `Relation`. One level up is `Sentence`. A sentence is either a fact or a rule. It consists of a *head* and a *body*. The head is a relation. The

body is a sequence of `Literal` objects. A literal is a relation and a *sign*. A sequence of sentences forms a `Ruleset`. The design and architecture was informed by the Python API of the answer set programming (ASP)-solver `clingo` [12].

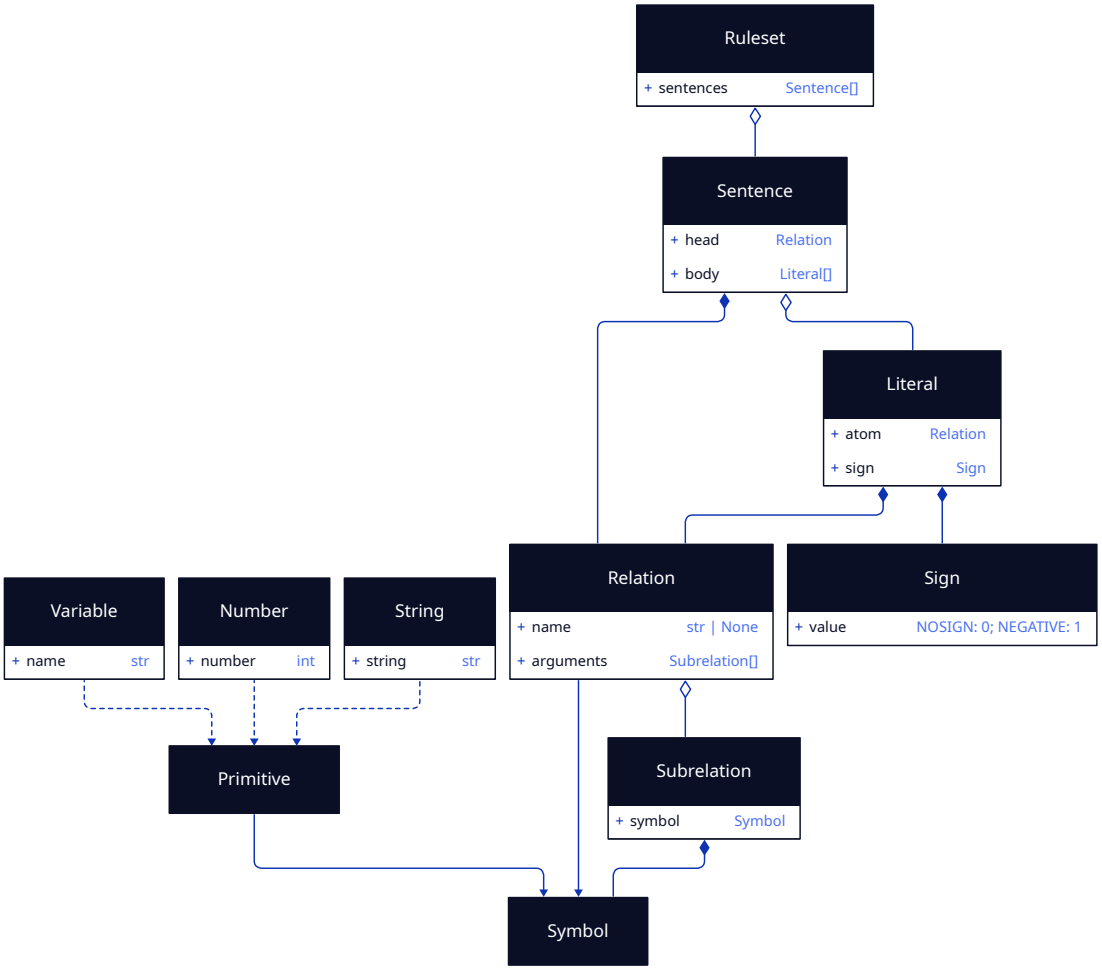

Figure 4.1: The class hierarchy of the elementary data structures.

The `Primitive` class is the abstract base class, of `Variable`, `Number` and `String`. A *variable* is only a name that may be used instead of any other symbol. A *number* is a zero-ary relation, and an integer as name. Finally, a string is also a zero-ary relation, with any characters as name. Strings are denoted inside quotes to distinguish between relations and variables. These primitives are the bedrock of the hierarchy. They do not have arguments, and therefore no children.

The `Relation` class consists of a name, that is either a `str` or `None` and a sequence of arguments, with the type `Subrelation`. A relation with no name is a tuple. A *subrelation* is a wrapper for either a relation or a primitive. The wrapper class allows a

type-safe total ordering between all symbols, and to distinguish a top-level relation and a relation as argument.

Sentences are constructed with the `Sentence` class, consisting of a head with type `Relation` and a body, a sequence of elements with type `Literal`. If the body is empty, the sentence is considered to be a fact. If the body is non-empty, the sentence is a rule. Literals are a `Relations` together with a `Sign`, which is either positive (i.e., `NOSIGN`) or negative (i.e., `NEGATIVE`).

Finally, an ordered collection of sentences forms a *ruleset* which is instantiated with the `Ruleset` class. In the implementation, `Ruleset` builds the rule-graph, and from that, extracts the strongly connected components.

4.1.2 Interpreter

The interpreter uses the definition of a game to fulfill requests. The definition of a game is given in GDL (see subsection 2.1.5). GDL is a dialect of Datalog, and therefore a stratified logic program. The interpreter transforms the logic program, depending on the request, and delegates it to a solver. Then the interpreter processes the solution, extracts and returns the answer, and thus carries out the request.

The architecture of the interpreter is inspired by Kowalski and Szykuła [14]. The interpreter uses the ASP-solver `clingo` [12], more specifically its Python-API, to solve logic programs. This approach was first described by Thielscher [28] and later by Möller et al. [19].

The interpreter has to fulfill static, dynamic, and temporal requests. Static requests are neither dependent on the state, nor sensitive to actions. These include what roles exist in the game and what the initial state is. Dynamic requests either depend on the state, and for one method the actions of the actors. The dynamic requests include, what the next state is, which role sees what, what the legal moves are, what the goal values for each role are, whether the state is terminal. Finally, there are two temporal requests, enumerating all possible histories, and listing all possible states. See fig. 4.2 for the methods of an interpreter.

The interpreter only considers relevant rules when fulfilling requests, by slicing the original GDL definition. The relevant rules are grouped in the strongly connected components of the rule graph. The rule graph is a directed graph, where nodes correspond to the signature of a relation, and an edge represents the inclusion in the body of a rule.

Some rules depend on the `distinct/2` relation, which holds if its arguments do not unify. Literals containing the relation are substituted with the ASP binary comparison `(!=)/2` (viz. does-not-equal operator). A backtracking-solver would need to delegate to the `(/=)/2` (viz. does-not-unify operator).

The method for the role request is called `get_roles`. It answers which roles exist in a game. The method extracts the rules where the heads are in the same strongly connected

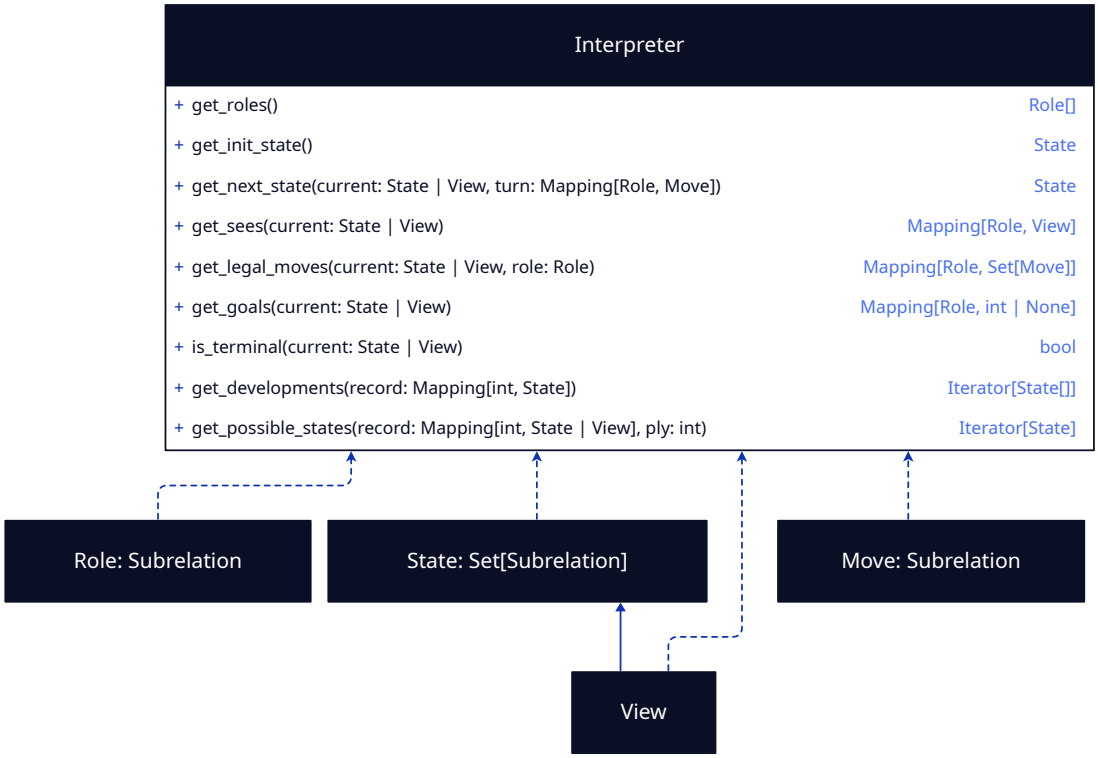

Figure 4.2: The class diagram of the interpreter.

component as the `role/1` relation. The solution to the corresponding logic program contains `role/1` relations. Every argument of such a relation is a role. The solvers answer may be cached.

The request for the initial state, is the interpreter's `get_init_state` method. The solver receives all rules which are relevant in respect to the `init/1` relation. In the solution, all arguments of the `init/1` relations form the true atoms of the initial state. The solvers answer may be cached.

Dynamic requests include an encoding of the state into the logic program. The state is a set of atoms, representing the atoms that are true in this state. All other atoms are assumed to be false. The encoding of a state is a series of facts. These facts match the signature `true/1`.

The method `get_next_state`, represents the request for the next state. It takes two inputs, the current state, and the turn. The solver receives all rules which have the relation `next/1` as head, and their relevant rules. Additionally, the current state is encoded, and the turn is encoded with `does/2` relations. A turn consists of moves, which are pairs of roles and actions. The role is the first argument of the `does/2` relation, while the action is the second. The solution to the logic program contains the next state encoded with `next/1` relations. All other relations in the solution may be

ignored.

Given a state, the method `get_sees` returns the views for each role in the game. The derived logic program includes all rules with the `sees/2` relation as head, their relevant rules, and the encoded state. The solution consists of `sees/2` relations where the first argument corresponds to the roles, and the second to the atom that is visible to the role. All other relations in the solution are irrelevant.

Similarly, the method `get_legal_moves` represents the request for the legal moves for each role in the game, given a particular state. By slicing the GDL definition, such that, only the relevant rules for all rules with `legal/2` are included, and the state is encoded, the interpreter derives the appropriate logic program. The solution of the logic program encodes the legal actions of each role in the given state via the `legal/2` relation.

To determine if a state is terminal, the interpreter offers the `is_terminal` method. The method takes the state as input. The logic program derived from the GDL definition includes all relevant rules to the `terminal/0` relation, as well as the encoded state. If the solution to the program includes `terminal/0` the state is terminal.

Temporal requests model the match as a series of states, as a logic program. This logic program is called the *temporal extension* [28]. The temporal extension of a GDL definition swaps out the `init/1`, `true/1`, and `next/1` relations for a `holds_at/2` relation. The `holds_at/2` relation's first argument is the atom, while the second argument is the ply number. It also transforms relations that are relevant in a temporal context by adding the `_at` suffix, and adding the ply number as another argument.

The `get_developments` method is a temporal request, and can enumerate all possible histories. It takes any number of states paired with their ply number, and returns all histories that are compatible with the states. The interpreter transforms the original GDL definition into the temporal version, and encodes the states at their respective ply with `holds_at/2` relations. The resulting logic program has multiple solution. Each solution represents a history. The state at a given ply of a history is encoded with `holds_at/2`.

Similarly, the `get_possible_states` method may be used to enumerate all possible states given a collection of views and their ply numbers. Optionally, the method also takes states with ply numbers. The resulting temporal logic program consists of `holds_at/2` and `sees_at/3` rules. Each solution encodes a possible history. The interpreter extracts the possible solutions by filtering the `holds_at/2` relations by ply number.

4.1.3 Match orchestration

Match orchestration is divided into two modules, the actors, and the general game playing (GGP)-gamemaster (see subsection 2.1.2 and [10]). The actors are objects defined by the `Actor` class. They serve as the interface between the agents and the gamemaster.

The gamemaster is represented by the `Match` class. It centrally mediates the actors, and manages the match's history.

The `Actor` class serves as the bridge between the gamemaster, and the agent. As a bridge, there is nearly no business logic, except enforcing disqualification when the game clocks run out. The gamemaster sends the signals discussed in subsection 2.1.2 via the methods `send_start`, `send_play`, `send_stop`, and `send_abort` to notify the agent of transitions in the match's phase. The return values of those methods are the respective answers.

The `Match` class is the representation of the gamemaster. It has a copy of the ruleset, an interpreter, a mapping of roles to their actor, the start- and game clocks, and the history. Once the match is over, the gamemaster also holds a mapping of roles to utilities.

4.2 Tree Data Structure

All implemented agents (see section 4.3) use a specialized tree data structure as a search tree. The tree data structure is a root node that links to its children. Each node may link to its parent (except the root node), and to its children (except the terminal nodes). The implementation consists of two types of nodes, perfect information nodes and information set nodes. There are no subtypes to perfect information nodes. There exist two subtypes of information set nodes, hidden information nodes, and visible information nodes. Hidden information nodes are for plies where the agent is not in control, while at visible information nodes the agent makes a move.

4.2.1 Perfect Information Nodes

Perfect information nodes (see subsection 2.2.1) are implemented as a class. The class' name is `PerfectInformationNode` and it specifies a mutable stateful object with five attributes, and four methods. The attributes are named `parent`, `children`, `state`, `turn`, and `valuation`. The methods are called `expand`, `evaluate`, `develop` and `trim`.

As the name suggests, the `parent` attribute is a link to the parent of a node. If the parent is `None` the node is the (current) root node. The distance from the root node, also implicitly defines the `depth` of a node. Generally, the depth of a node is defined to be `parent.depth+1` where the depth of the root is `0`.

In the implementation of search trees, the edges correspond to the mutable mapping `children`. The mapping's keys are turns, while its values are the child nodes. The keys represent the edge's labels. In perfect information nodes, the mapping is assumed to be bijective. Therefore, each turn maps to exactly one node, and there is an edge for every child node.

Each node represents the state of a game, stored in the attribute `state`. The root node holds the initial state. The state of a node cannot change— it is immutable.

After a ply, an agent may record the turn in the attribute `turn`. While not strictly necessary, it is useful for pruning branches that are no longer relevant. Namely, after `turn` is set, all entries in the mapping `children` with a different key are obsolete.

The attribute `valuation` stores the current valuation. It is mutable, as the MCTS-algorithm updates it during its iterations.

The method `expand` creates the child nodes, links them, and stores the links into `children`. It takes one argument, an interpreter. Using the interpreter, it enumerates each follow-up state, and the corresponding turn. It creates a `PerfectInformationNode` with the followup state as `state`, for each follow-up state. These nodes are mapped, with the turns as keys. The expand operation is idempotent.

With the `evaluate` method, the node is evaluated, thereby setting or updating the `valuation` attribute. The method takes three arguments, an interpreter, an evaluator, and a valuation factory. The evaluator uses the interpreter to produce a utility value from the `state` of the node. If the node already has an `valuation` that is not `None`, the `propagate` method aggregates the utility to the existing valuation. In the other case, the valuation factory produces a valuation and stores it in the `valuation` attribute.

During the progression of the game, the agents need to keep the tree up-to-date. The update process is fairly simple in perfect information games. As the moves are observable, updating the tree consists of setting the `turn` value for each node and walking the edges with the correct turns.

Nodes are updated with the `develop` method. Calling `develop` returns the node on a depth that is equal to the ply number, and for perfect information games, a state that equals the view— the current state. The method takes three arguments, an interpreter, the ply number, and the view. With the interpreter's `get_developments` method, the method iterates through all possible histories. For perfect information games, there is (usually) only one possible history. With the turns generated by the history, the method walks down the tree until reaching the correct node.

Finally, `trim` deletes all links to obsolete children. Without the links, the garbage collector is free to collect the children.

4.2.2 Hidden Information Nodes

Hidden Information Nodes are implemented via the `HiddenInformationNode`, which is informed by the concepts discussed in subsection 2.2.2. It is used for plies where the role of the actor is not in control. The class defines stateful and mutable objects. It has six attributes `role`, `possible_states` `fully_enumerated`, `fully_expanded`, `parent`, and `children`. It has three methods: `branch`, `evaluate`, and `develop`.

The attributes of `HiddenInformationNode` encode the current knowledge of the match. In `role` is the role from which perspective the tree is. It is the same value for all nodes of the tree. The information set of a node is stored in `possible_states`. It is a set

of states that share the same view. The attribute `fully_enumerated` is true if the information set is complete. The attribute `fully_expanded` is true if, for every possible state, all possible turns are linked in `children`. Both `parent`, and `children` fulfill the same function as in the implementation for perfect information nodes.

The mapping `children` represents the edges to its children, similar to `PerfectInformationNode`, however, the mapping is no longer guaranteed to be bijective. Furthermore, a hidden information node may only have up to two children, a visible child or another hidden child. As the moves are unobservable, all keys in the mapping point to one of those two children. The key of an entry consists of the assumed state, paired with the turn.

To add edges, the method `branch` takes two arguments, an interpreter and a state of `possible_states`. Using the interpreter, the method expands the `children` map with an edge per turn and the follow-up state, only for the state in the arguments. If the role in `role` is in control in the follow-up state, the visible node is the target. In the other case, the hidden child is the target.

As there are only two children for hidden information nodes, the `develop` method for information set nodes walks down the hidden information children until the depth matches the last ply. Then, it chooses the visible child node. Should a link not be branched yet, the method uses arbitrary states from `possible_states` to reach its target.

4.2.3 Visible Information Nodes

The implementation of the second type of information set node, the class `VisibleInformationNode`, is also informed by the concepts discussed in subsection 2.2.2. The class describes a stateful mutable object. Compared to `HiddenInformationNode` it has two additional attributes `view`, and `move`. Additionally, the `children` attribute differs. The class has one additional method, `trim`.

The attribute `view` is an optional view during the ply. The attribute is set to `None`, until the agent receives the view from the gamemaster. Once it is set, all incompatible states in `possible_states` are obsolete. The trim method will later remove these states.

The attribute `move` is an optional move. It is set after the agent has committed to a move. All edges that are incompatible become obsolete, once it is set.

The mapping `children` points from pairs of states (the assumed state), and moves to children. It is neither guaranteed to be injective, nor surjective. Therefore, two edges may point to the same node, and not all follow-up states are covered. Other than `HiddenInformationNode`, an instance of the `VisibleInformationNode` class may have more than two children. Visible children are grouped by the view of the follow-up states. Therefore, two follow-up states with the same view, are part of the same child's `possible_state` attribute. Hidden nodes are not grouped. Therefore, each move leads to a different hidden node. Thus, if two edges point to the same hidden child, their states differ, but not their move.

As with `PerfectInformationNode`, `trim` deletes all links to obsolete children. Without the links, the garbage collector is free to collect the children.

4.3 Agents

The agents are autonomous players, using MCTS to play games. The agents use the data structures and algorithms from the previous sections to search for the best move. The author implemented three types of agents: an agent using MCTS for games with perfect information, called `MCTSAgent`; an agent using single observer-information set Monte Carlo tree search with partially observable moves (SO-ISMCTS+POM), called `SO-ISMCTSAgent`; and an agent using multiple observer-information set Monte Carlo tree search (MO-ISMCTS) called `MOISMCTSAgent`. This section discusses the architecture used to implement the agents, as well as each agent separately. The design and implementation of the agents is informed by Cowling, Powley, and Whitehouse [8].

4.3.1 Architecture

The architecture of the agent's implementation is based on the *composition over inheritance* design principle. The design principle recommends to use composition over inheritance, if it can be used instead of inheritance. Consequently, the implementation is separated into multiple classes that could be aggregated. This decision led to a more modular code base, and was easier to understand for the author. For an overview, consult the class diagram in fig. 4.3.

The entry point for an agent is the `calculate_move` method. The method receives the numbered ply, the current view and information about the game clock. As the name suggests, it calculates the next move. How it calculates the move is based on the search algorithm. The search algorithm for the implemented agents was an adapted form of MCTS.

Any of the three MCTS-agents is also a tree-agent. Tree agents characteristically contain two minor- and two major operations. The minor operations are `descend` and `get_key_to_evaluation`. The major operations are `update` and `search`.

The method `descend` receives a *key* as input, and is called at the very end of the `calculate_move`. The key is the label of an edge, prunes all incompatible edges, and if possible, descends the (main) search tree. The key corresponds to the best move.

The `get_key_to_evaluation` operation builds a map that points each edge to an evaluation. Depending on the agent and type of game, an edge is not necessarily correlated to only one move. Thus, to evaluate the moves, a calling method has to further process this mapping.

With the `update` method, the agent ensures that the tree(s) are up-to-date before they start the search. The `update` method receives as input the current *ply* number and a

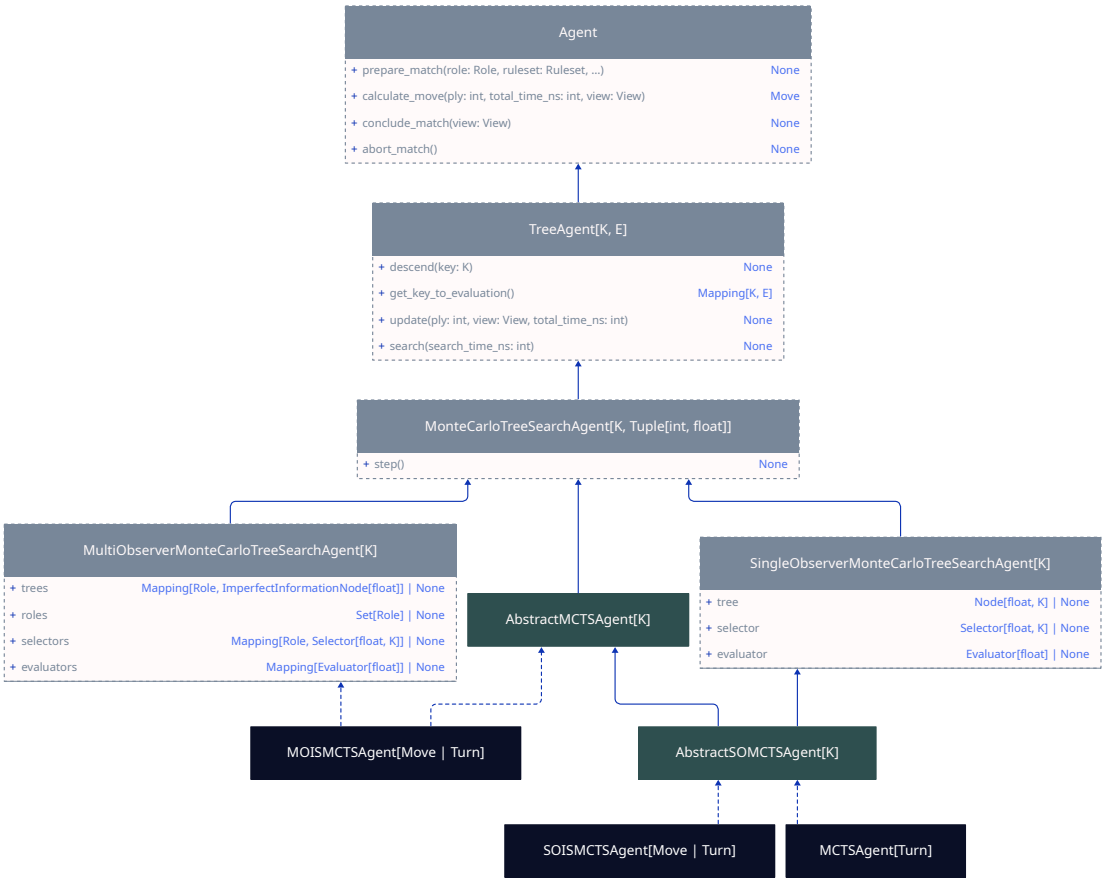

Figure 4.3: The class diagram for the agents. Protocols (viz. interfaces) are in a light shade of gray, and with dashed borders. Abstract classes have a darker shade of gray. Concrete implementation classes are in the darkest shade of gray.

view. The *view* in perfect information games is the full state. In imperfect information games, it is only the actual view of the agent.

Finally, in `search` the agent uses the MCTS-algorithm to progress the search tree. As input, the method only takes the remaining available time. It repeats iterations of MCTS until the available time expires.

The MCTS-agent's `calculate_move` is shared among all implemented agents. First, `calculate_move` calls `update` to get the tree(s) to the latest version. Then, it invokes `search`. Afterward, it uses `get_key_to_evaluation` to map the edges labels to their evaluation. The mapping is again processed by grouping, and aggregating by move. The resulting mapping points from a move to an evaluation. From this mapping it is possible to pick a move, by picking the move with the maximum evaluation. Finally, it calls `descend` to trim the tree(s), of all branches that are incompatible with that move.

Both the `MCTSAgent`-class and `SOISMCTSAgent`-class inherit from the abstract class `Ab-`

`stractSOMCTSAgent`. An object of type `AbstractSOMCTSAgent` is defined by a tree, a selector, and an evaluator. The tree is the search tree (see section 4.2) which MCTS is applied to. The selector picks a child node from the available children of a node. In this implementation, it uses the UCT algorithm for selection. The evaluator returns the utility of a state. In this implementation, it uses random playouts to evaluate a given state.

The `MOISMCTSAgent` is characterized by the attributes `trees`, `selectors` and `evaluators`. These three attributes are maps, that point from the role to their respective data structure. Trees are search trees (see section 4.2). As the UCT value depends on the valuation of a node, which in turn depends on the role, each selector uses UCT to pick a child node from a node for their associated tree. Finally, the evaluators do random playouts from the perspective of the role, to evaluate a given state.

4.3.2 Single Observer-Monte Carlo Tree Search

The abstract base-class `AbstractSOMCTSAgent` is the foundation of both the `MCTSAgent` (for perfect information games) and `SOISMCTSAgent` (for imperfect information games). The base-class applies the four phases of MCTS in `step`. The selector handles the select phase, the tree itself handles the expand phase, the agent determines the valuation of a node using the evaluator, and, finally, the tree handles the select phase. `MCTSAgent` assumes the tree is a perfect information tree (see subsection 4.2.1), while `SOISMCTSAgent` assumes an information set tree (see subsection 4.2.2 and 4.2.3).

The method `update` of `AbstractSOMCTSAgent` ensures that the tree is up-to-date. It uses the tree's `develop` method to walk to the node that has a depth that corresponds to ply number. For `SOISMCTSAgent`, the method uses some time to sample from the possible states, and add them to the node. This allows more possible states to be available for the selection phase.

During `step`, `AbstractSOMCTSAgent` iterates MCTS-algorithm once. The MCTS-algorithm (see section 2.3) is divided into the four phases select, expand, simulate and backpropagate. With the agent's selector, the `step` routine iteratively picks child nodes until it reaches an unexpanded node, which concludes the select phase. The expand phase uses the state of the node to add children nodes and link them. The same state is used for the simulation phase. The `step` routine uses the agent's evaluator to estimate a utility for the state. In the backpropagate phase, starting from the expanded node, the routine updates each node's evaluation with the utility, and walks up to the parent. The phase ends after reaching the current root node.

The `MCTSAgent` overrides `descend` and `get_key_to_evaluation`. The method `descend` receives a key. The key is the edge's label. The label for perfect information trees are turns. Thus, the agent can already walk the tree on non-simultaneous plies. The operation `get_key_to_evaluation` maps each possible turn to the evaluation of a node. The turn is associated with the node n, if from the current node c, there is an edge with the turn as a label on it, that leads to the node n.

The implementation for `step` of `SOISMCTSAgent` slightly differs from `MCTSAgent`'s. Before the select phase starts, it picks one of the available possible states. Based on this state, the selector picks a child. From this child, the method selects another possible state, and so forth. The method, upon reaching an unexpanded node, expands the tree based on the last determinized possible state. From there, the method uses the same implementation as its super class, for simulation and backpropagation.

After a `SOISMCTSAgent` picked a move (during `calculate_move`), it calls the `descend` method. The method can no longer walk down the tree, as it does not know the actual state. Therefore, it can only prune incompatible branches. Incompatible branches include those which do not have the picked move on the label of the edges.

4.3.3 Multiple Observer-Monte Carlo Tree Search

The `MOISMCTSAgent` uses MO-ISMCTS to find the best move. As the name suggests, it uses multiple trees, one for each role in the game, to accomplish this goal (see also subsection 2.3.5). Every role's tree is inherently linked to their perspective. The tree that matches the role of the actor's is synonymously called the main tree.

The `MOISMCTSAgent` describes a stateful object with three maps, each pointing the roles in the game to a tree, a selector, and an evaluator. The tree consists of hidden and visible information nodes. Modelling the available information, and how it is updated, necessitates this distinction. At hidden information nodes, the agent did (or rather will) not receive new information. Visible information nodes support incorporating new information. The former are used for plies where the agent is not in control, while the latter are used if the agent has to move. Thus, if a role is not in control, the corresponding node is a hidden node, and it is a visible node if the role is in control. As a consequence, the trees of each role are shaped differently and contain different information.

When incorporating new information with `update`, it only updates the main tree. During the update, the tree's `develop` method ensures the current node is at the same depth as the ply's number. Furthermore, it guarantees the current node's possible states are compatible with the view, discarding any that are incompatible. In the second phase of `update`, the method spends some time sampling the space of compatible states and adding them to the possible states of the node. As with `SOISMCTSAgent`, this enables more possible states to be picked during the selection phase.

Conceptually, `step` describes the four phases of MCTS, more specifically MO-ISMCTS. It starts by picking one of the possible states, called the determinization. The determinization may only originate from the main tree. Afterward, the interpreter generates a matching history for the determinization. The history enables synchronization of the trees to compatible branches. From there, the method applies the role-specific selector to its matching tree for all roles that are in control. As there is a determinization, and each role is in control, there is a unique edge in each tree. By extension, there exists a unique move for each role. These moves are combined into a turn. The turn is applied to the state. The state together with the moves determine the edges of the trees. This

is repeated, until the method reaches an unexpanded node, concluding the select phase. During the expand phase, the procedure expands the main tree's current node, assuming the determinization. As a consequence, only edges where the label contains the determinization are added. Each selector simulates a separate playout which is propagated back to their respective tree.

Each call of `step` during `calculate_move`, is significantly pricier, in comparison to `MCTSAgent` or `SOISMCTSAgent`. Therefore, `MOISMCTSAgent` takes significantly more time to reach the same number of iterations. However, the procedure overall takes fewer iterations to provide good moves.

Nonetheless, the implemented agents all feature a common base of implementation. The common base of implementation, was useful in debugging and for comparison during the experiment. The results of the experiment are featured in the next chapter, chapter 5.

CHAPTER 5

Results

The results include insights due to the analysis and the outcome of the experiment. First, is a qualitative evaluation of the effectiveness of the four studied Monte Carlo tree search (MCTS) modifications. Second, follows a review of characteristic features found for the games `Phantom_Connect(4,4,4)` and `Dark_Split_Corridor(3,4)`. Third and final, reports on the results of the experiment run on the same games.

5.1 Effectiveness of Existing Monte Carlo Tree Search Modifications

To evaluate the effectiveness of determinizing MCTS and of information set Monte Carlo tree search (ISMCTS), this section contains the case studies for `Phantom_Connect(4,4,4)` and `Dark_Split_Corridor(3,4)`. First, the case studies outline the most important rules and properties of the games. Then, given a characteristic game state, the different algorithms are considered. For each algorithm, the author of this thesis discusses the process of calculating the next move, and highlights the differences. Commonly, determinizing MCTS, single observer-information set Monte Carlo tree search (SO-ISMCTS) and single observer-information set Monte Carlo tree search with partially observable moves (SO-ISMCTS+POM) end up reaching similar conclusions. However, the convergence is mostly incidental, and due to the structure of the games.

5.1.1 Case Study `Phantom_Connect(4,4,4)`

The game `Phantom_Connect(4,4,4)` is an alternating two-player zero-sum game with hidden information. It is an instance of the n, m, k-connect family. Those games feature an $n \times m$ board. Each player marks cells with their initial (usually `x` and `o`), where the first player to mark k in a row, column, or diagonal wins the game. The game ends in a draw if the win-condition is not yet met, but all cells are already marked. Arguably

the most well known of this family is *Tic-tac-toe* (`Connect(3,3,3)`). Some *Connect* games feature modifications to their rules, like *gravity* and *phantom*. In *gravity*, a player only chooses the column, and the mark lands on the lowest not-already marked row. A famous game in this category is known as 4-Connect (`Gravity_Connect(7,6,4)`).

The *phantom* modifier, hides the status of a cell for all players, until they try to mark it. Once they mark a cell, they either know that it was not occupied before, resulting in their mark being present, or revealing the other player's mark. Should a player attempt to mark an already occupied cell, they may move again. Thus, the variant is no longer alternating.

The notation for the actions is (C, R), and the moves are denoted with $Role : (C, R)$. The column of the marked cell is denoted with C and the row is denoted with R. The roles are `x` (the first-moving player) and `o` the second-moving player.

The characteristic game state for the case study is in its seventh ply, after the sixth turn has been played. Eq. 5.1 lists its history. See fig. 5.1 for a visual representation of the characteristic state.

$$s_0 \xrightarrow{\texttt{x}:(1,4)} s_1 \xrightarrow{\texttt{o}:(4,4)} s_2 \xrightarrow{\texttt{x}:(4,1)} s_3 \xrightarrow{\texttt{o}:(1,4)} s_4 \xrightarrow{\texttt{o}:(3,3)} s_5 \xrightarrow{\texttt{x}:(3,2)} s_6 \tag{5.1}$$

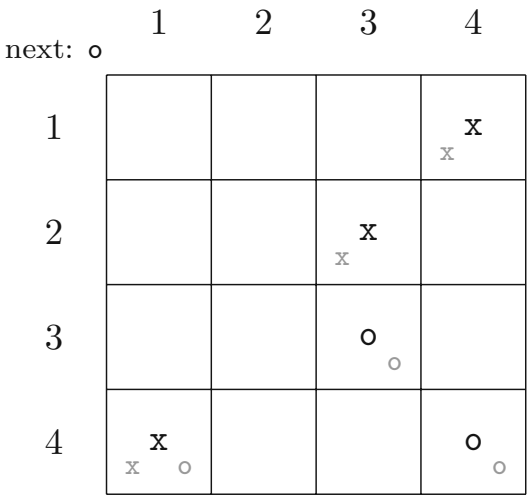

Figure 5.1: The characteristic game state s_6 for the case study of `Phantom_Connect(4,4,4)`. A cell is either blank, or features up to three marks. The marking player's symbol is seen in the center of the cell. The marks below show which player knows the mark of a cell.

The state s_6 favors x, as o is no longer permitted to make mistakes, to avoid a potential loss. Acceptable actions include $(2,3)$, $(3,2)$, and $(4,1)$. Both $(3,2)$, and $(4,1)$ allow immediate subsequent moves. Therefore, gaining more information in the process. Thus, they are both optimal moves.

From the perspective of o, 78 unique states are possible, where 5 (one of which is s_6) of these feature situations, where a wrong move from o might result in a loss. The board has 16 cells, 3 cells are known to o, and 5 cells are marked in total, thus $78 = \binom{13}{2}$ states are possible. A player cannot distinguish the order in which the opponent marked the cells, thus $158 = \frac{13!}{(13-2)!}$ histories lead to the 78 states.

Determinizing MCTS has at least 78 determinized trees, one for each unique state. Until the algorithm runs out of time or iterations, it chooses one of the trees and applies MCTS. From each root node, of every tree, the procedure counts the number of iterations of the child-nodes. The number of iterations is aggregated by the move that lead to the child. The algorithm returns the move with the highest number of iterations.

Given that in only 5 of the 78 determinized trees— less than 7%, the optimal moves converge to the acceptable moves, determinized MCTS would probably pick $(1,1)$ or $(2,2)$. The overwhelming number of determinizations are positions where x cannot secure a win in the next ply. In those positions, $(1,1)$ or $(2,2)$ would indeed be beneficial, as o would be able to secure a win with one more move. Determinized MCTS fails to weight the threat of losing accordingly.

SO-ISMCTS only features one tree, holding 78 unique possible states. Each history leads to a separate branch of the tree. The procedure cannot distinguish the correct branch and thus has to either create a new tree, and discard the previous iterations, or iterate from the last certain ancestor of the current node. At the start of an iteration, the procedure assumes a possible state. Next, the algorithm applies the four phases of MCTS, until the break condition. Until this point, SO-ISMCTS and determinizing MCTS are similar. However, as there is only one tree, SO-ISMCTS does not need to combine the data into a single tree. This allows the procedure to potentially reuse the previous iterations.

Nonetheless, despite the gained efficiency, SO-ISMCTS likely performs poorly, as it still suffers from strategy fusion. Only 5 out of the 78 possible states are similar enough to the actual state, such that their optimal move coincides with the acceptable moves. In most of the possible states $(1,1)$ and $(2,2)$ is far more likely to lead to a win, than defending on $(2,3)$, $(3,2)$, or $(4,1)$. Even if the probability is low, SO-ISMCTS fails to consider the importance of not losing.

The improved SO-ISMCTS+POM also uses only one tree, but addresses a fundamental weakness over the previous attempts, by removing the assumption that the opponent's moves can be observed. Without the assumption, the algorithm differentiates between hidden and visible nodes. Every non-terminal hidden node only has one or two children (see subsection 2.3.4). Ultimately, SO-ISMCTS+POM never needs to throw away the tree. Starting from 78 possible states, the procedure determinizes a possible state. On

visible nodes, the usual MCTS selection step ensues. At hidden nodes, the algorithm determinizes again.

Even though, the impending loss is given algorithmically adequate weight, by aggregating all possible states in hidden nodes, the algorithm still likely fails to address the threat because the opponent's model is reduced to random moves. Given the overwhelming number of bad moves `x` could do i.e., not picking $(2, 3)$ to win, it still fails to acknowledge the threat. However, this is not due to strategy fusion, but due to the high number of available options for `x`.

Finally, multiple observer-information set Monte Carlo tree search (MO-ISMCTS) improves the opponent's model, by once again using multiple trees. Each role (i.e., `x` and `o`), has their dedicated tree. Each tree has visible and hidden nodes, and thus does not assume observable moves. Instead of sampling a random state at hidden nodes, the selection continues at the other role's tree. The benefit is two-fold. Not only is the opponent's model more strategic, but also the iterations are efficiently allocated and can be reused for later plies.

The improved MO-ISMCTS will rank the actions $(2, 3)$, $(3, 2)$, and $(4, 1)$ high. Any other actions either lead to losses because `x` will win the diagonal, or `x` to block `o`'s attempt to get the other diagonal.

The analysis shows that MO-ISMCTS has a decisive advantage in a characteristic state for this game. Additionally, Cowling, Powley, and Whitehouse [8] empirically came to a similar result. Their explanation is that `Phantom_Connect(4,4,4)` triggers strategy fusion, which leads to overly optimistic and pessimistic assumptions. The optimistic assumption is that the agent can respond to the opponent's actions perfectly. The pessimistic assumption is that the opponent can also perfectly respond to the agent's move.

All four methods stand to benefit from weighting the possible states. Cowling, Powley, and Whitehouse [8] agree, and together with opponent strategy modelling, suggest such an improvement in their outlook.

5.1.2 Case Study `Dark_Split_Corridor(3,4)`

The game `Dark_Split_Corridor(3,4)` is an alternating two-player zero-sum game with hidden information. It is an instance of the n, m-corridor family. An n, m-split corridor game is played on two $n \times m$ boards, where each player controls the sole pawn of that board. The pawn starts in the column with the index $\lfloor (n+1)/2 \rfloor$ of the first row. The pawn is allowed to move to any orthogonally adjacent cell. The player, whose pawn reaches the last row first, wins. Besides moving the pawn, a player may choose to construct a barrier between two orthogonally adjacent cells on the opponent's board. A pawn cannot cross a barrier. The player may only construct a barrier, if it is still possible to reach the last row.

The *dark* variant, hides the barriers from the players' views, while the pawns' positions, are still visible to both players. Should a player attempt to move a pawn such that it crosses a barrier, the pawn stays put, and the barrier is revealed. Players see revealed barriers.

There are two classes of actions in *dark split corridor*, moving and blocking. All moving-actions are fully observable to the opponent. Actions that block, cannot be differentiated from one another. As a corollary, both families of actions are distinguishable. Therefore, a player can judge if the other player moved or blocked.

The notation for the actions are *south*, *west*, *north*, *east*, denoting the movement, and $(C1, R1), (C2, R2)$ denoting the blocks. The coordinates $(C1, R1)$ and $(C2, R2)$ ought to be orthogonally adjacent, and as a convention $(C1, R1)$ is the cell that is more northwest than $(C2, R2)$. The columns are denoted as a, b, and c. Rows are enumerated 1 to 4.

The characteristic game state for the case study is in its fourth ply, after the third turn has been played. The history is listed in eq. 5.2. Fig. 5.2 offers a visual representation of the characteristic state.

$$s_0 \xrightarrow{\texttt{left}:south} s_1 \xrightarrow{\texttt{right}:south} s_2 \xrightarrow{\texttt{left}:south} s_3 \xrightarrow{\texttt{right}:(b,3),(b,4)} s_4 \tag{5.2}$$

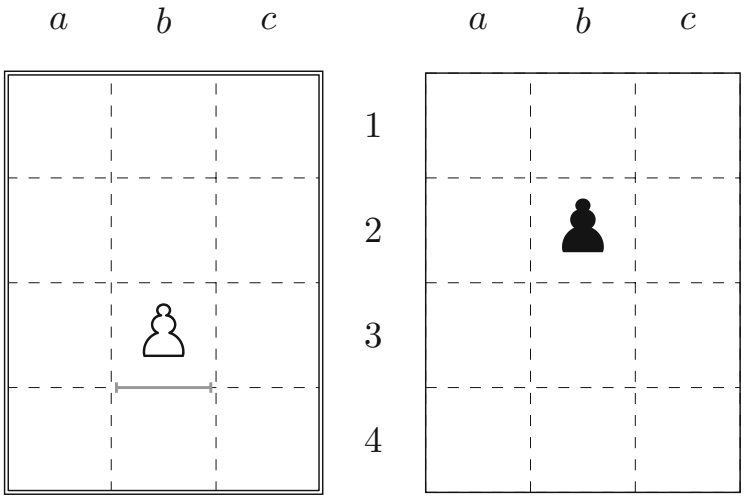

Figure 5.2: The characteristic game state s_4 for the case study of `Dark_Split_Corridor(3,4)`. `left` is in control, indicated by the double border. The pawns are visible to both players. The barrier is still hidden to `left` indicated by the gray color.

The state s_4 favors `left` over `right`. The second-moving role `right` is forced to respond, at least until `left` makes a mistake, then `left` falls back to the more reactive role. In an advantageous position, victory is assured, should the disadvantaged role not block.

From the available actions, both moving and blocking are admissible. Given the power of clairvoyance, moving *west*, or *east* is desirable. Realistically, `left` cannot be sure which of the available crossings is blocked. Either `right` bluffed and set the barrier somewhere unexpected, or it set the barrier at the expected crossing (which is the case in s_4). Thus, there is a slight chance that calling the bluff and moving *south* wins the game. However, in s_4 the move is suboptimal. While it does not lose the game, it forces `left` to be the reactive player.

Given the uncertainty, blocking actions could be considered. If `right` did not bluff, then moving *west* or *east* can be defended with the appropriate $(a,3),(a,4)$ or $(c,3),(c,4)$ answer. From there, `left` needs at least three plies for a victory, while `right` can win in two. Eventually, `left` has to block some crossing as well to increase the distance for `right`. If `right` bluffed, then moving increases the number of plies until victory by one in the worst case, and `right` can increase this distance by blocking again. Therefore, `left` will need to block to still be able to win.

The available blocking actions can be categorized into non-bluffs and bluffs. The non-bluffs are $(b,2),(b,3)$— blocking the immediate forward movement, and $(b,3),(b,4)$— blocking the straightforward path. Therefore, all other options could be included. However, blocks concerning the first row are riskier, as they may only be useful later in the game. Thus, as bluffs $(a,2),(a,3)$; $(a,2),(b,2)$; $(a,3),(a,4)$; $(a,3),(b,3)$; $(b,2),(c,2)$; $(b,3),(c,3)$; $(c,2),(c,3)$; and $(c,3),(c,4)$ are acceptable moves.

As the moves in plies 1, 2 and 3 are fully observable, there are only 15 possible moves, and by extension only 15 possible histories, one for each potentially blocked crossing. Due to the symmetry of the game, 12 states have a mirrored equivalence.

An agent using determinizing MCTS considers the 15 possible states as 15 separate trees. In only one of those *south* is not the optimal move. Therefore, it is likely that after aggregating all the iterations from the trees, the procedure picks *south*. Unfortunately, the one tree, where the chosen move is suboptimal, corresponds to the real state.

While, the move is suboptimal, not all is lost. Furthermore, *south* reduces the number of possible states to 1, which significantly improves the performance of determinizing MCTS for subsequent turns.

SO-ISMCTS also tracks 15 possible states, managed in a single tree. Due to the assumption, that all moves are fully observable, the procedure is either forced to rebuild the tree from scratch, or start from above the current root. At this point, the algorithm randomly determinizes from the possible states. Like with determinizing MCTS only one of the possible states has *south* as suboptimal move. The possibility that `right` blocked, is judged equal to all other states. As *south* leads to wins most of the time, it is the preferred move.

In the case of SO-ISMCTS+POM, the same 15 possible states also remain in a single tree. The model never assumes that a move is visible, allowing to reuse the previous iterations, from before plies. The algorithm assumes `right` chose their moves randomly. Therefore, the algorithm likely will come to the same conclusion as its predecessors. Most of the moves `right` could have made, have *south* as the optimal answer.

This limitation is only partially overcome with MO-ISMCTS. Due to the procedure determinizing at the root node, all states 15 states are equally likely. Thus, *south* stays the most promising action. However, unlike the previous methods, MO-ISMCTS spend more iterations in this part of the tree.

Given the analysis, all methods select the same move *south*. This move is judged to be suboptimal, when clairvoyant. However, each method takes different paths to reach the same conclusion because the actual state is outnumbered by alternatives. Due to the equal weighting, they hide the more relevant game states. Frank and Basin [9] call this phenomenon *non-locality* (also described in [8]).

For this game, all methods would benefit from a more sophisticated weighting of the possible states. Especially SO-ISMCTS+POM and MO-ISMCTS stand to gain the most, as they could effectively use the win-ratio of the determinized playouts as weight (or loss-ratio should the opponent be in control).

5.2 Characteristic Features of Games

The UCT-border is constrained by two boundaries, zero and the lowest depth of the game. Zero is the minimum, as it is not possible to expand to a depth less than zero. It is equally impossible to expand beyond terminal nodes. Usually, during a match, an agent using MCTS will only encounter terminal depth, when the match has progressed and is close to the end. In the plies before that, the UCT-border is constrained by the limited resources available to the algorithm. If a proposed limit for the UCT-border is too low, the effect will be noticeable but likely prohibitively negative. If the limit is set too high, the tree might not ever be constrained. Finding the correct value is especially difficult in the context of general game playing (GGP), where the players cannot be fine-tuned to the games. The only available information remains the description of the game (static analysis), and sampled states (dynamic sampling).

Four characteristic features of games are the cardinality of the state space, the cardinality of the action space, the average arity of the game tree, and the average depth of the terminal nodes. All four features are closely related to the size of the game tree and by extension the search tree.

5.2.1 Cardinality of State Space

The state space is the set of all possible spaces. The cardinality of the state space serves as a lower bound for the total number of nodes in the game tree. In the best case, each state

is mapped to exactly one node. For complex games, it is often impractical to calculate the exact number of states. In service to brevity, the analysis for both `Phantom_Connect(4,4,4)`, and `Dark_Split_Corridor(3,4)` only attempts to find approximate upper bounds.

A state of `Phantom_Connect(4,4,4)` may have a cardinality up to 66. Enumerated by 2 `control/1` atoms, 32 `cell/3` atoms, and 32 `revealed/3` atoms. Each of the 66 atoms that may be included or excluded in a state. Therefore, a first upper bound for the size of the state space is 2^{66}.

A tighter upper bound for the cardinality of the state space is $2 \cdot 5^{16} \approx 3.05 \times 10^{11}$— significantly less than the naive $2^{66} \approx 7.37 \cdot 10^{19}$. Of the 66 atoms, 2 atoms describe which role is in control. The rules dictate that exactly one role is in control. Therefore, there are only 2 classes of states. The board has 16 cells. For each cell, there are two atoms. One atom for each role that could potentially mark the cell. There can either be no mark or one mark of a role (but not of both). Therefore, a cell may have 3 different states. Finally, a marked cell can be revealed to the opponent, or remain hidden, two states per marked cell. Unmarked cells cannot be revealed to either player. Thus, a cell can be in 5 different states. First, not marked, second marked by `x` and not revealed, third marked by `x` and revealed, fourth marked by `o` and not revealed, and fifth marked by `o` and revealed. In total, resulting in 5^{16} board configurations.

The game `Dark_Split_Corridor(3,4)` has a state consisting of up to 86 atoms. The sum consists of 2 `control/1` atoms, 24 `at/2` atoms, 30 `border/2` atoms, and 30 `revealed/2` atoms. A naive upper bound for the cardinality of the state space is 2^{86}.

This upper bound can be reduced to at least $2 \cdot (6 \cdot 3^{14} + 3 \cdot 3^{13} + 3 \cdot 3^{12})^2 \approx 2.46 \times 10^{15}$. The board of the game is split into two separate sub-boards, one sub-board for each player. On a board, there is exactly one pawn, therefore a sub-board can be in 12 different states. The number of barriers is harder to count exactly, as it is dependent on the position of the pawn. A crossing can be in three states, the barrier not constructed, the barrier constructed and not revealed, and the barrier constructed and revealed. Per sub-board there are 15 crossings. On the last two rows, there has to be at least one crossing where no barrier is constructed. On the second row, there have to be at least two empty crossings. Finally, on the first row, at least three crossings remain empty. In total, each sub-board has $6 \cdot 3^{14} + 3 \cdot 3^{13} + 3 \cdot 3^{12}$ unique states.

5.2.2 Cardinality of Action Space

The action space is the set of all actions. The cardinality of the action space is an upper bound for the arity of a single node, and by corollary a bound for the average arity of the tree.

The action space of `Phantom_Connect(4,4,4)` is 16. All 16 have the signature `cell/2`. One argument for each cell. Each cell may be marked by either player.

The action space of `Dark_Split_Corridor(3,4)` is 19. A pawn may move in four directions, and there are 15 crossings that may be constructed.

5.2.3 Average Arity

The arity of a state is the number of unique turns. The number of turns depends on the number of actions. Furthermore, a turn consists only of legal actions from roles in control. In non-simultaneous moving games, the number of turns is the number of legal actions. Both `Phantom_Connect(4,4,4)` and `Dark_Split_Corridor(3,4)` are non-simultaneous moving games.

For `Phantom_Connect(4,4,4)` the average arity is approximately 8.37. The average arity was estimated experimentally. For the estimation 486,000 histories were generated. During generation 8,002,496 unique states were recorded. The estimation for the average arity is the mean number of legal actions.

The average arity of `Dark_Split_Corridor(3,4)` is approximately 6.61. Similarly to `Phantom_Connect(4,4,4)`, the average arity was estimated by experiment. The estimation consisted of 354,000 histories, resulting in 13,342,062 unique states. The average arity was estimated with the mean number of legal actions of these states.

While choosing an action for `Phantom_Connect(4,4,4)` removes the option for later plies, the same is not true for `Dark_Split_Corridor(3,4)`. Namely, moving the pawn is (typically) reversible. Once all barriers are constructed, the legal actions reduce to a minimum of 1 (moving the pawn in one direction) up to a maximum of 4 (moving the pawn in all four directions).

5.2.4 Average Depth of Terminal Nodes

The depth of a node is defined by the distance to the root node. The leaf nodes of a game tree correspond to the terminal states. The distance between the root and a leaf node is the length of the associated history. Therefore, the average length of histories corresponds to the average depth of the terminal nodes.

The average depth of terminal nodes for `Phantom_Connect(4,4,4)` is approximately 21.86. The estimate is the arithmetic mean length of 619 801 histories. The theoretical minimum depth is 7. A game with this depth features `x` marking a line, `o` not prohibiting the line, and `o` not marking any of the line's cells. The theoretical maximum depth is 32, a draw where every cell is attempted to be marked twice, revealing the whole board for both roles.

The terminal nodes of the game `Dark_Split_Corridor(3,4)` have an average depth of 43.25. The estimation considered 127 601 histories. The theoretical minimum is 5. This is a game where `left` moves down the board in every ply, and `right` does not block `left` in doing so. The theoretical maximum is not finite, as both `left` and `right` could just repeatedly move back and forth, never reaching the last row.

	Phantom_Connect(4,4,4)	Dark_Split_Corridor(3,4)
$\|S\|$	$\leq 3.05 \times 10^{11}$	$\leq 2.46 \times 10^{15}$
$\|A\|$	16	19
$\bar{b}$	≈ 8.37	≈ 6.61
$\bar{d}$	≈ 21.86	≈ 43.25
l	4	5

Table 5.1: The extracted numeric features compared for the games `Phantom_Connect(4,4,4)` and `Dark_Split_Corridor(3,4)`. $|S|$ is the cardinality of the state space. $|A|$ is the cardinality of the action space. $\bar{b}$ is the average arity. $\bar{d}$ is the average depth of terminal nodes. l is the chosen maximum depth for the experiment (see section 5.3)

5.2.5 Choice of Maximum Depth

The feature evaluation of `Phantom_Connect(4,4,4)` shows a significantly smaller state space, and a game tree with slightly more branches, but not as deep, in comparison to `Dark_Split_Corridor(3,4)` (see table 5.1). The direct comparison, influenced the decision for the limited agent's maximum expansion depth in the experiment (see section 5.3).

A tree with a smaller average arity can usually be searched deeper in the same amount of MCTS iterations. The matchup in the experiment of the unlimited agent against itself confirmed that. Specifically, the unlimited agent searched deeper for `Dark_Split_Corridor(3,4)` than for `Phantom_Connect(4,4,4)`.

A match's outcome typically is not decided in the very last move. In most games, after a key ply, the match's outcome cannot be changed anymore. The nature of the game dictates when that ply is, and how long the game continues after that. For example, in Chess, winning with only a bishop and a knight takes up to 20 plies (i.e., 40 moves), often times much longer. However, the outcome is inevitable (given no mistakes occur).

Matches of `Phantom_Connect(4,4,4)` are typically shorter than `Dark_Split_Corridor(3,4)`. This indicates that the tail of moves, where the outcome is inevitable, is shorter than with `Dark_Split_Corridor(3,4)`. If victory is guaranteed, the match usually ends soon in `Phantom_Connect(4,4,4)`. By contrast, a match of `Dark_Split_Corridor(3,4)` can already be decided, but 9 plies are still left to be played (given no player makes mistakes).

Nonetheless, the length of a typical game, together with the average arity (bound by the cardinality of the action space), is a benchmark for how deep an agent should calculate. Given the loose connection between size of the state space and the number of the tree's nodes, this metric is less meaningful.

The number of iterations the MCTS procedure makes, indirectly limits the depth. Together, with the other metrics, the choice for the maximum expansion depth may be

done comparatively. This allows to estimate the influence of a changed maximum expansion depth. For example, to increase the maximum expansion depth, without also increasing the number of iterations, will likely lead to deeper trees, that are more similar to a tree searched by unconstrained MCTS. Similarly, decreasing the expansion depth in comparison to another game, where the average arity was higher, will lead to more balanced tree, with high average expansion depth.

5.3 Experiment Results

The experiment consisted of two double-round robin tournaments with self-play, where three MCTS-agents played 450 matches in total. The number of iterations was fixed, while there was no time-limit. The played games were `Phantom_Connect(4,4,4)` and `Dark_Split_Corridor(3,4)`. All percentages in this section are rounded to two significant digits.

5.3.1 Results of the *Phantom Connect* Tournament

The `Phantom_Connect(4,4,4)` tournament had 450 matches in total, where each of the three agents played 250 matches. Each agent has 300 results. This is due to the double-counting of the games with self-play. The three agents were a clairvoyant agent using MCTS (agent C), an unlimited MO-ISMCTS agent (agent U), and an agent using MO-ISMCTS with a maximum expansion depth of 4 (agent $L(4)$). All agents did 7500 iterations per ply. The clairvoyant agent capitalizes upon its advantage, and did not lose a single game. The unlimited agent is a significantly better player for the role `x` compared to the limited agent. For the role `o` their playing strengths are closer, with a slight advantage towards the unlimited agent. Table 5.2 shows the result matrix in absolute numbers. Fig. 5.3 displays the win-, tie-, and loss-rate for both roles. Furthermore, the figure displays the total number of points, if a win is worth one point, a tie equal to a half point, and loss is worth no points.

The clairvoyant agent played 250 matches, where in 50 of those matches it played against itself. It was the strongest player. In total, it won 82 (27.33%), lost none, and drew 218 (72.67%) of them. As the player moving first, agent C won 59 (39.33%) of the matches, lost none, and tied 91 (60.67%). As the player moving second, it won 23 (15.33%), lost none, and drew 127 (84.67%) of the matches. In a direct comparison, both the unlimited, and the limited agent fared similarly against the clairvoyant agent. Against U, it won 42 (42%), and tied 58 (58%) matches. With $L(4)$, it won 40 (40%), and tied 60 (60%) matches.

Of the 250 matches the unlimited agent played, 200 were against other players, and 50 against itself. U's playing strength was in between C and $L(4)$. Winning 71 (23.67%), losing 96 (32.00%), and drawing 133 (44.33%) of those 300 results. As the first-moving player, agent U managed to win 62 (41.33%) matches, while losing 18 (12%), and tieing 70 (46.67%). Going second, resulted in 9 (6%) wins, 78 (52%) losses, and 63 (42%)

o-role \ x-role	C			U			$L(4)$		
	x	o	½	x	o	½	x	o	½
C	0	0	50	0	12	38	0	11	39
U	30	0	20	26	3	21	22	6	22
$L(4)$	29	0	21	36	3	11	26	3	21

Table 5.2: The result matrix of the `Phantom_Connect(4,4,4)` tournament. The cells describe the total number of distinct outcomes. The horizontal axis denotes the agent of the x-role (the player moving first). The vertical axis denotes the agent of the o-role (the player moving second). The subcolumns of the cell's matrix describe the outcome. They are written as x (the player moving first won), o (the player moving second won), and ½ (the match ended in a draw).

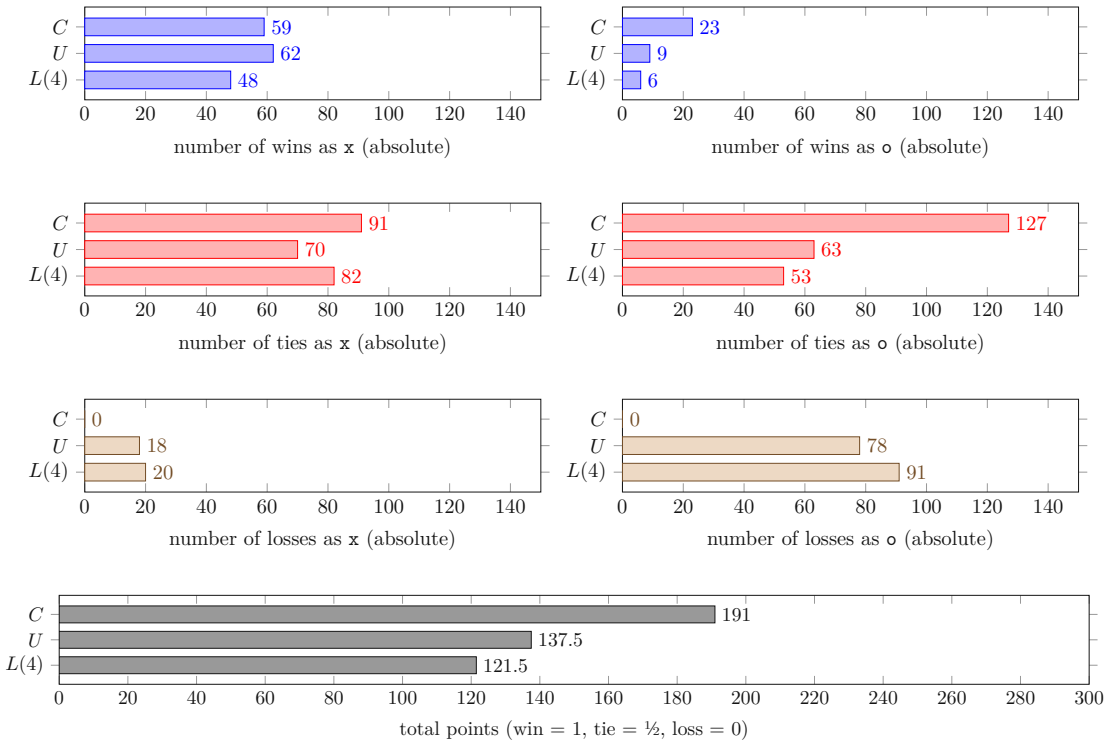

Figure 5.3: The results for the `Phantom_Connect(4,4,4)` tournament.

draws. The direct matchup U versus $L(4)$ favors the former, winning 42 (42%), losing 25 (25%), and tieing 33 (33%), out of 100 matches.

The limited agent participated in 250 matches, giving 300 results, due to the 50 matches against itself. It was the weakest player. Of the 300 results, 54 (18%) are wins, 111 (37%) are losses, and 135 (45%) are draws. As the player moving first, $L(4)$ managed to accrue 48 (32%) wins, 20 (13.33%) loses, and 82 (54.67%) ties. The results as second-moving player were 6 (4%) wins, 91 (60.67%) losses, and 53 (35.33%) ties.

The clairvoyant agent seems to converge to optimal play, as in the self-matchup all played matches resulted in ties. The game `Connect(4,4,4)` is a known draw [30]. Two clairvoyant agents playing `Phantom_Connect(4,4,4)` is isomorphic to two agents playing `Connect(4,4,4)`. At suboptimal play, spurious non-draws would occur. Furthermore, both U, and $L(4)$ have similar results against C, even though their playing strengths differ in the direct comparison. This suggests, that C is as strong as possible. Otherwise, it would fare distinctly against each of its opponents.

Notably, U has a higher win-rate than C as `x`. U could capitalize $L(4)$'s weakness better than C. One possible explanation is the wrongly made pessimistic assumption. The clairvoyant agent assumes that any move can be countered, while the unlimited agent models the lack of knowledge the limited agent has.

The direct comparison of the unlimited agent U and the limited agent $L(4)$ shows that the former is stronger as the `x`-role. When U assumed the `x`-role, it won 36 (72%), lost 3 (6%), and drew 11 (22%) out of 50 matches. Against the same opponent, $L(4)$ achieved only 26 (52%) wins, the same number of losses— 3 (6%), and 21 (42%) ties.

The playing strength of U and $L(4)$ for the `o` role is relatively similar, with an advantage for the unlimited agent. U managed to win twice as often, totaling 6 (12%) matches, losing only 22 (44%) matches, and tieing 22 (44%) matches, in comparison to the mirrored case. While playing against itself, the outcome is identical to the $L(4)$ versus $L(4)$ matchup. A possible explanation for the identical outcome, is the relatively weak playing performance of $L(4)$ as `x`.

The experiments done by Cowling, Powley, and Whitehouse [8] bear similar results. However, the paper omitted exact numbers of the experiment, [8, fig. 9] shows that the *cheating UCT* (equivalent to clairvoyant MCTS) is equally matched against itself, and winning most of the games against *MO-ISMCTS* (equivalent to unlimited MO-ISMCTS). Furthermore, when weighting ties as half-wins, the win percentage suggests about 60%, which is slightly more than indicated in [8, fig. 10]. The win percentage of U with about 45% is also slightly lower than the indicated value in the same figure. In comparison to the experiment done in this thesis, the number of games was significantly higher, with a total number of matches at 37 880.

5.3.2 Results of the *Dark Split Corridor* Tournament

The `Dark_Split_Corridor(3,4)` tournament featured 450 matches. As with the `Phantom_Connect(4,4,4)` tournament, each agent played 250 matches, which leads to 300 results, due to double counting the self-play matches. The contestants were the clairvoyant MCTS agent (denoted as C), an unlimited agent using MO-ISMCTS (abbreviated with U), and a limited expansion depth MO-ISMCTS (denoted as $L(5)$). The maximum expansion depth of $L(5)$ was 5. The agents did 2500 iterations per ply. The clairvoyant agent was the strongest player, as expected. However, the advantage was not as clear-cut as with `Phantom_Connect(4,4,4)`. Surprisingly, the limited agent won the direct comparison against the unlimited agent, assuming the `left`-role (the first-moving role). Furthermore, $L(5)$ was able to accrue more wins moving first, compared to U, when playing against C. Playing as the `right`-role, the advantage turns the other way around, and U beat $L(5)$. The result matrix is shown in table 5.3. The win and loss-rate for both roles is visualized in fig. 5.4, as well as the total number of points, if a win is worth one point, and loss is worth no points.

The clairvoyant agent was featured in 250 matches, resulting in 300 results, due to the double counting in self-play matches. Of the three, it was the strongest agent. The total number of wins was 219 (73%), while it lost 81 (27%) of the times. Moving first, it won 134 (89.33%) matches, while losing 16 (10.67%) out of 150 matches. Of the three agents, it was the strongest as `left`. Moving second, it won 85 (56.67%) and lost 65 (43.33%) matches, making it the only agent with a positive win rate for this role. As expected, this agent was the strongest when assuming the `right`-role. Going first, in the direct matchups, the most formidable opponent was itself, with 40 (80%) wins and 10 (20%) losses. Both against U and $L(5)$ it achieved identical results, winning 47 (94%) games and losing only 3 (6%).

The unlimited agent contested in 250 matches, totaling 300 results. It won 114 (38%) matches and lost 186 (62%). As the first-moving player, it was the sole agent that lost more often than not. With 73 (48.67%) wins and 77 (51.33%) losses, it was the weakest first-moving player. Going second, U manages to clutch 41 (27.33%) of the wins, while losing 109 (72.67%). This slots it as the second best of the agents, considering only the `right`-role. In direct matchups, when moving first, C was the strongest opponent, with a win-lose ratio of $10 : 40$ (20%). Against itself, it won 29 (58%) and lost 21 (42%) matches. Finally, against $L(5)$ the tally was 34 (68%) wins and 16 (32%) losses. Going second, it was beat substantially by C, with only 3 (6%) wins and 47 (94%) losses for U. In the matchup against itself, it won 21 (42%) and lost 29 (58%) matches. Surprisingly, $L(5)$ was a stronger opponent, with 17 (34%) wins and 33 (66%) losses out of 50 matches for U.

The limited agent also contested in 250 matches, totaling 300 results. Marginally, $L(5)$'s win rate was the second best, with 117 (39%) wins versus 183 (61%) losses. When assuming the `left` role, 83 (55.33%) of the played matches were wins, and 67 (44.67) losses, making it the second-best agent when moving first. As the second-moving player,

		left-role					
		C		U		$L(5)$	
		left	right	left	right	left	right
right-role	C	40	10	10	40	15	35
	U	47	3	29	21	33	17
	$L(5)$	47	3	34	16	35	15

Table 5.3: The result matrix of the `Dark_Split_Corridor(3,4)` tournament. The cells describe the total number of distinct outcomes. The horizontal axis denotes the agent of the `left`-role (the player moving first). The vertical axis denotes the agent of the `right`-role (the player moving second). The subcolumns of the cell's matrix describe the outcome. They are written as `left` (the player moving first won), and `right` (the player moving second won),

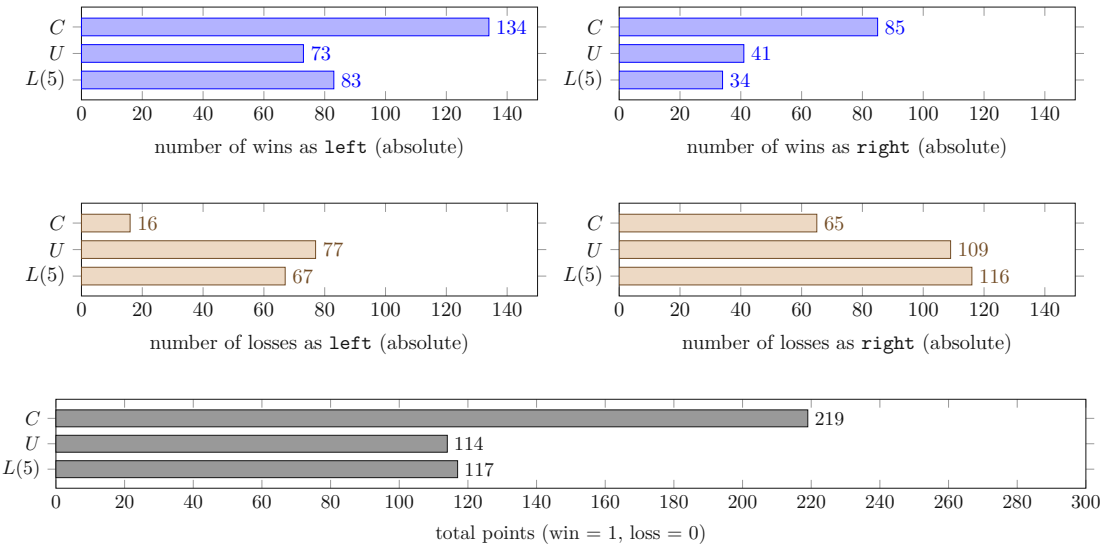

Figure 5.4: The results for the `Dark_Split_Corridor(3,4)` tournament.

it loses to both C and U with a win-loss ratio of 34 : 116 (22.67%). The direct comparison shows it to be an apt agent for the `left` role, winning 15 (30%) and losing 35 (70%) matches against C. Against U it won 33 (66%) times and lost 17 (34%) matches. Playing against itself yielded similar results; $L(5)$ won 35 (70%) matches and lost 15 (30%).

The data suggests, that in contrast to `Phantom_Connect(4,4,4)`, the clairvoyant agent did not yet converge towards optimal play. Currently, the game of `Dark_Split_Corridor(3,4)` lacks formal analysis. However, assuming the first-moving player has the advantage, at optimal play, C should have a perfect win rate. Especially in the case against U and $L(5)$, where C solely has perfect information.

With the parameters set, $L(5)$ is surprisingly a better first-moving player than U. Not only does it win the direct matchup, but also win more often against the superior clairvoyant agent. While it was able to eke out an advantage as first-moving, it is approximately by the same margin weaker in the second-moving case.

The results do not provide an obvious explanation for why $L(5)$ is better when moving first. A discussion featuring an interpretation and the implications continues in the next chapter, specifically section 6.3.

CHAPTER 6

Discussion

The contributions of this thesis cover three topics, that coincide with the three research questions. First is a comparison of the extensions for Monte Carlo tree search (MCTS) for games with imperfect information in the context of general game playing (GGP) available in the literature. Second is a suggestion of what features of a game may be useful for determining the depth of a limited expansion depth MCTS algorithm, and how those features may be quantified. Third and final is an evaluation of a limited expansion depth algorithm using an experiment.

6.1 Monte Carlo Tree Search Extensions

Of the considered MCTS modifications for imperfect information games in the context of GGP, multiple observer-information set Monte Carlo tree search (MO-ISMCTS) was the most apt extension. MO-ISMCTS is a part of the information set Monte Carlo tree search (ISMCTS) family. It mitigates the issues of strategy fusion and non-locality. Furthermore, both weighting the possible states, and more sophisticated modelling of the opponent, are powerful suggestions for improving the method further.

The results show, in line with Cowling, Powley, and Whitehouse [8], MO-ISMCTS is an effective extension for playing imperfect information games in the context of GGP. MO-ISMCTS is a designated algorithm for games with hidden information. Furthermore, it does not require game-specific heuristics to approximate the next best move. Yet, it is not perfect. The case study of `Dark_Split_Corridor(3,4)` showed, that without further modifications, the best move does not always coincide with the likely chosen move. Furthermore, while the opponent model of MO-ISMCTS is certainly stronger than that of single observer-information set Monte Carlo tree search with partially observable moves (SO-ISMCTS+POM), it is an area of improvement. Nevertheless, MO-ISMCTS suffers less from strategy fusion and non-locality, while keeping a strong enough opponent model.

When building an agent capable of GGP with imperfect information, MO-ISMCTS should be considered as the primary search algorithm. Previous research focused on either GGP (with perfect information) or games with hidden information. Work that assumes both is sparse. These results demonstrate that MO-ISMCTS is the most effective expansion of MCTS for a wide class of applications.

Due to the evaluation on specific examples, the comparison of the existing MCTS methods usable for imperfect information games in the context of GGP is lacking generalizability. The chosen games are both relatively simple two-player games. The differences in the algorithms are possibly not as pronounced as with more complex examples. Certainly, the differences would be greater in games with more than two roles (especially for MO-ISMCTS). However, the chosen games are important representatives. Furthermore, their relative simplicity allows for a clear analysis.

The list of considered MCTS methods was not exhaustive. There are alternative approaches available. Most can be categorized by determinization and information set approaches. Many resort to multiple paradigms—hybrid approaches. The goal was to represent their base variants by focusing on determinizing MCTS, and the three instances of ISMCTS.

Finally, the aspect of computational cost viability was left out of scope entirely. Cowling, Powley, and Whitehouse [8] report that MO-ISMCTS is computationally pricier, while producing better results. Analysis covering such a scope would have hindered the brevity of this thesis. Overall, the issue can and should be considered separately, and does not at all affect the playing strength.

While MO-ISMCTS is indeed a good base-expansion of the MCTS algorithm fit for games with imperfect information in the context of GGP, it is still not ideal. Avenues for future research include the improvement of MO-ISMCTS with better opponent modelling, and more sophisticated probabilistic analysis in respect to the possible states. Moreover, future research may aim to further develop the other available approaches.

Furthermore, in future research, these methods should be additionally validated and their difference quantified. As a first suggestion, the author of this thesis recommends tournaments. Tournaments lend themselves to direct comparison, even though they are computationally intensive endeavors. They are easy to replicate, and emphasize differences in playing strength. Quantifying playing strength is difficult qualitatively.

6.2 Feature Extraction

The features obtained by dynamic sampling, and static analysis, as applied in the prior art, characterize a game. This characterization lends itself to heuristically determining where the UCT-border at a given number of MCTS iterations might be. The discussed features were the cardinality of the state space, the cardinality of the action space, estimate the average arity, and estimate the average depth of terminal nodes.

The underlying assumption for extracting features directly from game descriptions was that these are characteristic enough to determine a value for the maximum expansion depth for a limited MCTS agent. This assumption was confirmed by applying two methods from the prior art, static analysis and dynamic sampling. Static analysis was useful in providing lower and upper bounds for features related to the game tree's size. Their result was less meaningful, but computationally cheap. Dynamic sampling is an easy to apply method, but computationally intensive. Furthermore, while the values are more directly applicable, they might be sensitive to biases in the sampling method, and may require a substantial sample size to be meaningful.

Static analysis and dynamic sampling are methods previously used by Kuhlmann and Stone [15], Schiffel and Thielscher [21], Clune [6] and Mańdziuk and Świechowski [17], to generate ad-hoc heuristics for playing games in a GGP context. The qualitative evaluation of using these methods to set the maximum expansion depth, built on the existing use-case and extended the idea. The methods can be used for configuring search methods, especially in the context of GGP. The configuration may span not only the maximum expansion depth in the case of the limited UCT-border agent, but also the exploration parameter in a general MCTS agent.

Beyond the scope of this thesis was to quantify how good the features extracted from the game were at heuristically suggesting the value for the limit of the expansion depth for the limited agent. Given that the ultimiate decision for the limit in the experiment was not only based on the extracted features, there was no direct quantitative evaluation of the method. Nonetheless, as the scope was restricted to the initial viability of the method, this limitation was not confounding.

Feature extraction via static analysis and dynamic sampling seems promising. Further research is needed to establish the viability of the approach in an ad-hoc scenario. An ad-hoc scenario might be a tournament with an unlimited agent and a limited expansion depth agent that can adjust the UCT-border according to the game and the phase of the game.

Another area of interest is to explore other features extractable by static analysis or dynamic sampling. A promising feature, not yet considered, is the number of possible states. The number of possible states is an important factor for ISMCTS. More possible states likely require more iterations to accurately estimate the expected utility of a node.

In addition, more qualitative analysis on the feature extraction is necessary. Feature extraction was proposed as a mechanism for generating heuristics for GGP. Thus, it is mainly discussed in a quantitative setting and sufficient theoretical evaluation is still lacking.

6.3 Limited Expansion Depth Agent

The experimental data shows that limiting the maximum expansion depth for games with imperfect information, in the context of GGP, can have a negative and surprisingly

also a positive effect. Specifically, in the example of `Phantom_Connect(4,4,4)` the effect was decisively negative, whereas with `Dark_Split_Corridor(3,4)` there was a positive effect when moving first. The positive effect was not present when assuming the second-moving role.

The outcome of the `Phantom_Connect(4,4,4)` tournament contradicts the hypothesis that limiting the expansion depth has a positive effect on playing strength. The agent with a limited UCT-border was severely disadvantaged. This was measurable in the tally of the direct matchup, as well as the aggregated points.

The `Dark_Split_Corridor(3,4)`-tournament's results show that in certain conditions, it can be beneficial to reduce the maximum expansion depth of an agent using MO-ISMCTS. Surprisingly, the positive effect was only present when the agent was assuming the first-moving role. The results indicate that as the second-moving role, the limited expansion depth had a negative effect.

The results of the experiment do not fit with the theory that a deeper search tree is strictly better. One of the reasons why MCTS is lauded, is that it refrains from exploring parts of the tree that are less promising. This allows to produce deeper trees, while remaining accurate. Previous research has focused on increasing the depth MCTS can foray into. The results of the experiment demonstrate that in certain conditions this might be counterproductive.

There are multiple possible explanations for why the limited agent surpassed the unconstrained agent in `Dark_Split_Corridor(3,4)`. One such explanation would be, that due to the relatively low number of MCTS iterations, the search tree of the unconstrained agent was too optimistic, placing too much weight on parts of the search tree which are still highly uncertain due to the hidden information. Due to the fixed UCT-border, the uncertainty was better accounted for, as each expansion step compounds the unpredictability of the future (what move will my opponent make) with the ambiguity of the hidden game state.

Perhaps, each game has an associated characteristic curve, parameterized by the exploitation parameter and the number of MCTS iterations. The curve indicates in how many cases the algorithm converged to the actual best move, given a maximum expansion depth. Three possible families of curves might exist. The simplest case is a monotonically increasing curve; The farther the algorithm can look ahead, the better the playing strength. In fig. 6.1 an example of such a curve is labeled *Monotonically Increasing*. The global optimum for the expansion depth is the maximum height of the game tree. The second case describes all curves with peaks and troughs, until there is again a global optimum at the maximum height of the game tree. An example curve is labeled *Non-Monotonically Increasing* in fig. 6.1. The third case would be all curves where the global optimum does not reside with the maximum height of the game tree. See fig. 6.1, for an example; The curve is labeled *Early Peak*.

Possibly, `Dark_Split_Corridor(3,4)` with the chosen iterations and exploration parameter has a characteristic curve belonging to the second or third family. More data

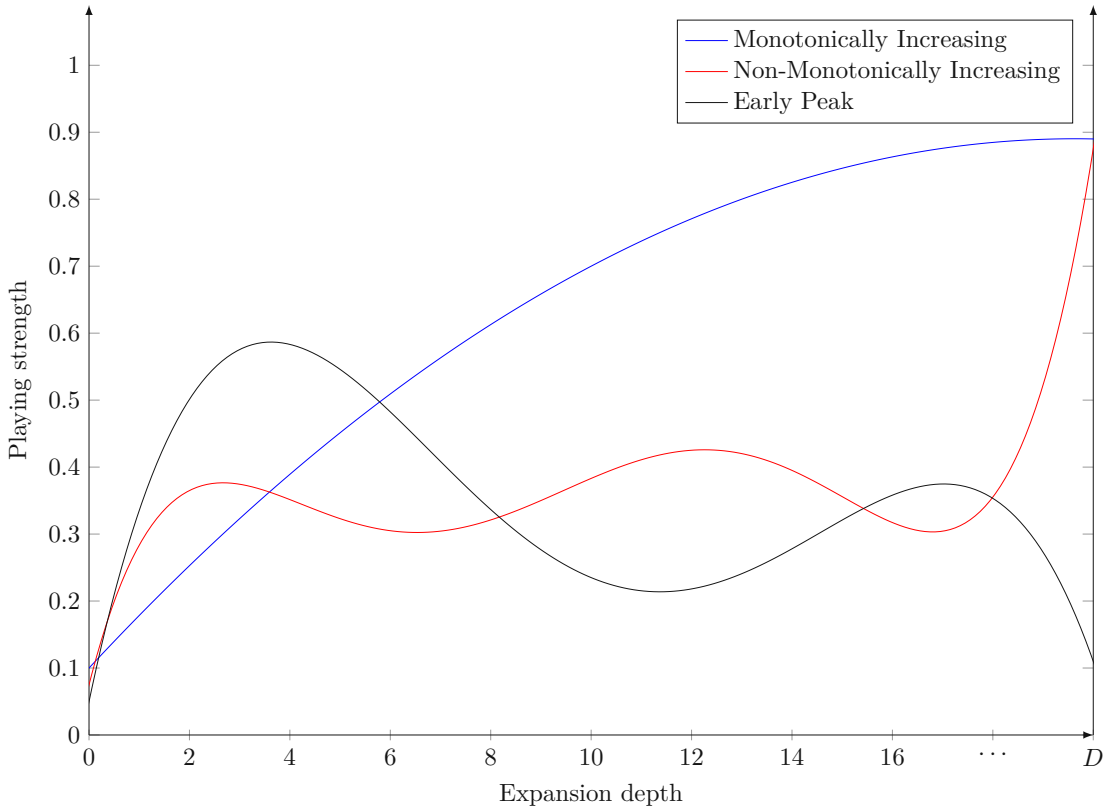

Figure 6.1: The hypothesized three families of game characteristic curves. A curve is parameterized by the exploration parameter, and the number of MCTS iterations. The x-axis quantifies the expansion depth. Left is zero, on the very right is the depth of the game tree D. The y-axis quantifies an abstract notion of playing strength—the probability that the chosen move coincides with the best move.

is required to confirm the hypothesis of game characteristic curves is applicable. Furthermore, if this hypothesis is not rejected, additional evidence if `Dark_Split_Corridor(3,4)` indeed has a game characteristic curve of the family *Non-Monotonically Increasing* or *Early Peak*.

One of the tournaments' limiting factor was the number of matches per matchup. With 50 matches per matchup, is certainly on the lower end for permitting a correlation between playing strength and performance in a tournament. This was mitigated by including the clairvoyant agent as an overwhelmingly stronger opponent. In both tournaments, the non-clairvoyant agent which achieved overall better results also fared better against the clairvoyant agent. This permits drawing the conclusions with a lower number of matches.

The restricted number of iterations for the `Dark_Split_Corridor(3,4)` limits the generalizability of the results. Without more data it is impossible to confirm if the limited

expansion depth agent has an advantage because of the game, the number of iterations, or other factors such as a better rate of convergence at the chosen exploitation parameter. However, in tournaments and real-world scenarios, the number of iterations is most likely suboptimal. In these situations, it may then be beneficial to limit the expansion depth.

Future studies should not only include the tournaments with different games and more matches per matchup. Ideally, these tournaments would vary the exploration parameter among agents, as well as their number of iterations and the maximum expansion depth. Therefore, allowing to quantify the impact of these parameters.

CHAPTER 7

Conclusion

This research aimed to examine Monte Carlo tree search (MCTS) as a search method for games with imperfect information in the context of general game playing (GGP). The thesis pursued three goals. First was to identify effective modifications for MCTS for GGP in hidden information scenarios. Second, was to suggest a set of features that could inform the maximum depth of a limited expansion depth MCTS variant, as well as examining a method for evaluating these features. Third and final, was to conduct a quantitative evaluation of an agent using said MCTS variant. The quantitative evaluation required the implementation of an engine, capable of orchestrating GGP matches and interpreting the standard description language, game description language with imperfect information (GDL-II).

7.1 Summary

Drawing upon a large body of work in a literature review, and a qualitative analysis using concrete examples, this thesis provided the relevant background for imperfect information games and GGP. Furthermore, it compared the variants determinization MCTS, and information set Monte Carlo tree search (ISMCTS). As MCTS is a commonly used method for GGP, the goal was to find and verify the most promising variant of MCTS for imperfect information games in the context of GGP. In accordance with the hypothesis, the results indicated that determinizing MCTS is not an ideal extension, while from the family of ISMCTS, multiple observer-information set Monte Carlo tree search (MO-ISMCTS) was the most promising method for imperfect information games in the context of GGP.

By reusing a method found in literature, this thesis provided a concrete way of determining a set of features, useful for characterizing games, without prior knowledge about the game. The characterization of the game hinted at the size of the game tree. The size of the game tree subsequently influenced the value for the maximum depth of the limited

expansion depth MO-ISMCTS algorithm used by an agent in the experiment. Priorly, the method was used for generating heuristics for search methods, without any game-specific knowledge, which was why it was also applicable for this use case.

The limited expansion depth MO-ISMCTS variant was shown to have an advantage only when moving first in a game with a more complex state, but fewer possible moves. The agent was evaluated against an overwhelmingly stronger clairvoyant MCTS agent, and a baseline unconstrained expansion depth MO-ISMCTS agent using two tournaments. In the tournament for the game `Phantom_Connect(4,4,4)`—a variant of the game *Tic-tac-toe* on a 4×4-board, where the players do not see the marks of their opponents, the limited expansion depth agent was disadvantaged. In the other tournament for the game `Dark_Split_Corridor(3,4)`, where both players race towards a finish line, and may be obstructed by invisible barriers placed by the opponent, the limited expansion depth agent achieved an about 6.67% (10 out of 150 matches) higher win rate as the first-moving player.

In an effort to support GDL-II and orchestrate matches of GGP-agents, one of the contributions of this thesis is an integrated game engine capable of these features. The engine supports loading GDL-II definitions, and provides an interpreter interface for agents. Included in the engine are agents using MCTS. The agents use the interpreter to manage their search trees, and calculate the next best move. This thesis reviews the implementation of the engine, to make it more accessible to learn from. Ultimately, the goal was to shape future revisions and iterations.

7.2 Future Work

While this thesis provides an introduction into the usage of MCTS for imperfect information games in the context of GGP, there remain many insufficiently explored topics. These topics include the combination of modifications, and application of extension-specific optimizations. Specifically for MO-ISMCTS, its authors hypothesize a benefit when applying opponent modelling, and probability weighting of the possible states (see [8]). Experiments are needed to quantify the effects of these suggested improvements.

After the promising initial evaluation of static analysis and dynamic sampling for feature extraction, further research is necessary. Future studies could address other features (e.g. the number of possible states). Furthermore, to determine the viability of the approach, a more in-depth analysis of the computational cost remains open. Likewise, finding specific counter examples where the characterization via feature extraction is misleading, may help to improve and refine the method.

This thesis showed that the limited expansion depth MCTS variant can play better in specific games. This ultimately leads to the question for which class(es) of games this is also true. Furthermore, future research could consider why there is such a big difference between moving first and moving second. Moreover, it remains to be validated whether

there is an ideal limit given a range of parameters, such as characterizing information about the game, the number of MCTS iterations, and data about the opponent.

The engine serves as an iterative prototype influenced by previous similar efforts. While the performance was acceptable, it is still far from optimal. The literature provides a plethora of unexplored optimizations for evaluating GDL, and improving the computational speed of the tree data structures. Future versions may improve the throughput of the engine to greatly increase the number of matches that are economical to run.

7.3 Contributions

The engine and its Python interface, implemented as part of the experiment, pave the way towards an improved ecosystem in the field of GGP. Similar predecessors use Java as an implementation language, making it difficult to embed with commonly used artificial intelligence (AI) packages. Most of those packages are meant to be used via Python, one of the most widely used languages in AI and machine learning (ML). Therefore, it was chosen as the implementation language of the engine, facilitating its usage in the context of AI and ML.

This thesis offers a novel approach of limiting the expansion depth of MO-ISMCTS. Moreover, the data provides initial evidence that the approach is indeed viable. An experiment revealed an unexpected conditional advantage in a specific game. This challenges the often times implicit made assumption, that MCTS always benefits from a deeper search tree. The approach together with the engine form the main contributions to the research topic of GGP, and the broader field of AI.

List of Figures

List of Tables

List of Code Listings

Acronyms

GGP general game playing

MCTS Monte Carlo tree search

AI artificial intelligence

ML machine learning

GDL game description language

GDL-II game description language with imperfect information

ISMCTS information set Monte Carlo tree search

SO-ISMCTS single observer-information set Monte Carlo tree search

SO-ISMCTS+POM single observer-information set Monte Carlo tree search with partially observable moves

MO-ISMCTS multiple observer-information set Monte Carlo tree search

ASP answer set programming

Bibliography

[1] Hans Berliner. "Some Necessary Conditions for a Master Chess Program". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. International Joint Conference on Artificial Intelligence. Vol. 3. Aug. 20, 1973, pp. 77–85.

[2] Y. Bjornsson and H. Finnsson. "CadiaPlayer: A Simulation-Based General Game Player". In: *IEEE Transactions on Computational Intelligence and AI in Games* 1.1 (Mar. 2009), pp. 4–15. ISSN: 1943-068X, 1943-0698. DOI: 10.1109/TCIAIG.2009.2018702. URL: http://ieeexplore.ieee.org/document/4804731/ (visited on 01/10/2023).

[3] Charles L. Bouton. "Nim, A Game with a Complete Mathematical Theory". In: *The Annals of Mathematics* 3.1/4 (1901), p. 35. ISSN: 0003486X. DOI: 10.2307/1967631. JSTOR: 1967631. URL: https://www.jstor.org/stable/1967631?origin=crossref (visited on 12/07/2023).

[4] Cameron B. Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (Mar. 2012), pp. 1–43. ISSN: 1943-068X, 1943-0698. DOI: 10.1109/TCIAIG.2012.2186810. URL: http://ieeexplore.ieee.org/document/6145622/ (visited on 01/11/2023).

[5] Paolo Ciancarini and Andrea Gasparro. "Priority Level Planning in Kriegspiel". In: *Entertainment Computing - ICEC 2012*. Ed. by Marc Herrlich, Rainer Malaka, and Maic Masuch. Vol. 7522. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 333–340. ISBN: 978-3-642-33541-9 978-3-642-33542-6. DOI: 10.1007/978-3-642-33542-6_29. URL: http://link.springer.com/10.1007/978-3-642-33542-6_29 (visited on 11/30/2023).

[6] James Clune. "Heuristic Evaluation Functions for General Game Playing". In: *AAAI*. AAAI. Vol. 7. 2007.

[7] Rémi Coulom. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". In: *Computers and Games*. Ed. by H. Jaap Van Den Herik, Paolo Ciancarini, and H. H. L. M. Donkers. Red. by David Hutchison et al. Vol. 4630. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 72–83. ISBN: 978-3-540-75537-1 978-3-540-75538-8. DOI: 10.1007/978-3-540-75538-8_7. URL: http://link.springer.com/10.1007/978-3-540-75538-8_7 (visited on 11/20/2023).

[8] P. I. Cowling, E. J. Powley, and D. Whitehouse. “Information Set Monte Carlo Tree Search”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.2 (June 2012), pp. 120–143. ISSN: 1943-068X, 1943-0698. DOI: 10.1109/TCIAIG.2012.2200894. URL: http://ieeexplore.ieee.org/document/6203567/ (visited on 12/21/2022).

[9] Ian Frank and David Basin. “Search in Games with Incomplete Information: A Case Study Using Bridge Card Play”. In: *Artificial Intelligence* 100.1-2 (Apr. 1998), pp. 87–123. ISSN: 00043702. DOI: 10.1016/S0004-3702(97)00082-9. URL: https://linkinghub.elsevier.com/retrieve/pii/S0004370297000829 (visited on 12/09/2022).

[10] Michael Genesereth, Nathaniel Love, and Barney Pell. “General Game Playing: Overview of the AAAI Competition”. In: *AI Magazine* 26.2 (2005), pp. 62–72. DOI: 10.1609/aimag.v26i2.1813.

[11] Giuseppe De Giacomo, Yves Lesperance, and Adrian R Pearce. “Situation Calculus Game Structures and GDL”. In: (2016), p. 9.

[12] Roland Kaminski et al. *How to Build Your Own ASP-based System?!* Nov. 5, 2021. arXiv: 2008.06692 [cs]. URL: http://arxiv.org/abs/2008.06692 (visited on 09/29/2022). preprint.

[13] Levente Kocsis and Csaba Szepesvári. “Bandit Based Monte-Carlo Planning”. In: *Machine Learning: ECML 2006*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Red. by David Hutchison et al. Vol. 4212. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293. ISBN: 978-3-540-45375-8 978-3-540-46056-5. DOI: 10.1007/11871842_29. URL: http://link.springer.com/10.1007/11871842_29 (visited on 11/16/2023).

[14] Jakub Kowalski and Marek Szykuła. “Game Description Language Compiler Construction”. In: *AI 2013: Advances in Artificial Intelligence*. Ed. by Stephen Cranefield and Abhaya Nayak. Red. by David Hutchison et al. Vol. 8272. Cham: Springer International Publishing, 2013, pp. 234–245. ISBN: 978-3-319-03679-3 978-3-319-03680-9. DOI: 10.1007/978-3-319-03680-9_26. URL: http://link.springer.com/10.1007/978-3-319-03680-9_26 (visited on 01/02/2023).

[15] Gregory Kuhlmann and Peter Stone. “Automatic Heuristic Construction in a Complete General Game Player”. In: *AAAI*. Vol. 6. 2006, pp. 1457–62.

[16] Jeffrey Long et al. “Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 24.1 (July 3, 2010), pp. 134–140. ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v24i1.7562. URL: https://ojs.aaai.org/index.php/AAAI/article/view/7562 (visited on 01/02/2023).

[17] Jacek Mańdziuk and Maciej Świechowski. “Generic Heuristic Approach to General Game Playing”. In: *SOFSEM 2012: Theory and Practice of Computer Science*. Ed. by Mária Bieliková et al. Vol. 7147. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 649–660. ISBN: 978-3-642-27659-0 978-3-642-27660-6. DOI: 10.1007/978-

3-642-27660-6_53. URL: http://link.springer.com/10.1007/978-3-642-27660-6_53 (visited on 01/10/2023).

[18] Munyque Mittelmann and Laurent Perrussel. "Game Description Logic with Integers: A GDL Numerical Extension". In: *Foundations of Information and Knowledge Systems*. Ed. by Andreas Herzig and Juha Kontinen. Vol. 12012. Cham: Springer International Publishing, 2020, pp. 191–210. ISBN: 978-3-030-39950-4 978-3-030-39951-1. DOI: 10.1007/978-3-030-39951-1_12. URL: http://link.springer.com/10.1007/978-3-030-39951-1_12 (visited on 01/06/2023).

[19] Maximilian Möller et al. "Centurio, a General Game Player: Parallel, Java- and ASP-based". In: *KI - Künstliche Intelligenz* 25.1 (Mar. 2011), pp. 17–24. ISSN: 0933-1875, 1610-1987. DOI: 10.1007/s13218-010-0077-4. URL: http://link.springer.com/10.1007/s13218-010-0077-4 (visited on 11/08/2023).

[20] S. Schiffel and M. Thielscher. "Representing and Reasoning About the Rules of General Games With Imperfect Information". In: *Journal of Artificial Intelligence Research* 49 (Feb. 14, 2014), pp. 171–206. ISSN: 1076-9757. DOI: 10.1613/jair.4115. URL: https://www.jair.org/index.php/jair/article/view/10862 (visited on 01/02/2023).

[21] Stephan Schiffel and Michael Thielscher. "Automatic Construction of a Heuristic Search Function for General Game Playing". In: *Department of Computer Science* (2006), pp. 16–17.

[22] Stephan Schiffel and Michael Thielscher. *Fluxplayer: A Successful General Game Player*. 2007. URL: https://www.aaai.org/Papers/AAAI/2007/AAAI07-189.pdf (visited on 12/29/2022).

[23] Michael Schofield and Michael Thielscher. "General Game Playing with Imperfect Information". In: *Journal of Artificial Intelligence Research* 66 (Dec. 13, 2019), pp. 901–935. ISSN: 1076-9757. DOI: 10.1613/jair.1.11844. URL: https://jair.org/index.php/jair/article/view/11844 (visited on 12/30/2022).

[24] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. Dec. 5, 2017. arXiv: 1712.01815 [cs]. URL: http://arxiv.org/abs/1712.01815 (visited on 11/01/2023). preprint.

[25] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (Jan. 28, 2016), pp. 484–489. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature16961. URL: http://www.nature.com/articles/nature16961 (visited on 01/09/2023).

[26] Maciej Świechowski et al. "Monte Carlo Tree Search: A Review of Recent Modifications and Applications". In: *Artificial Intelligence Review* (July 19, 2022). ISSN: 0269-2821, 1573-7462. DOI: 10.1007/s10462-022-10228-y. URL: https://link.springer.com/10.1007/s10462-022-10228-y (visited on 12/01/2022).

[27] Michael Thielscher. “A General Game Description Language for Incomplete Information Games”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 24.1 (July 4, 2010), pp. 994–999. ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v24i1.7647. URL: https://ojs.aaai.org/index.php/AAAI/article/view/7647 (visited on 12/30/2022).

[28] Michael Thielscher. “Answer Set Programming for Single-Player Games in General Game Playing”. In: *Logic Programming*. Ed. by Patricia M. Hill and David S. Warren. Vol. 5649. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 327–341. ISBN: 978-3-642-02845-8 978-3-642-02846-5. DOI: 10.1007/978-3-642-02846-5_28. URL: http://link.springer.com/10.1007/978-3-642-02846-5_28 (visited on 01/09/2023).

[29] Alan M. Turing. “Chess”. In: *Computer Chess Compendium*. Ed. by David Levy. New York, NY: Springer New York, 1988, pp. 14–17. ISBN: 978-1-4757-1970-3 978-1-4757-1968-0. DOI: 10.1007/978-1-4757-1968-0_2. URL: http://link.springer.com/10.1007/978-1-4757-1968-0_2 (visited on 10/30/2023).

[30] J.W.H.M. Uiterwijk and H.J. Van Den Herik. “The Advantage of the Initiative”. In: *Information Sciences* 122.1 (Jan. 2000), pp. 43–58. ISSN: 00200255. DOI: 10.1016/S0020-0255(99)00095-X. URL: https://linkinghub.elsevier.com/retrieve/pii/S002002559900095X (visited on 01/18/2024).