



**CTU**

CZECH TECHNICAL  
UNIVERSITY  
IN PRAGUE

**F3**

**Faculty of Electrical Engineering  
Department of Cybernetics**

**Bachelor's Thesis**

# **Finite-horizon Approximation of Partially Observable Stochastic Games**

**Matěj Veselý**

**May 2024**

**Supervisor: doc. Mgr. Branislav Božanský, Ph.D.**





# BACHELOR'S THESIS ASSIGNMENT

## I. Personal and study details

Student's name: **Veselý Mat j** Personal ID number: **491989**  
Faculty / Institute: **Faculty of Electrical Engineering**  
Department / Institute: **Department of Cybernetics**  
Study program: **Open Informatics**  
Specialisation: **Artificial Intelligence and Computer Science**

## II. Bachelor's thesis details

Bachelor's thesis title in English:

**Finite-horizon Approximation of Partially Observable Stochastic Games**

Bachelor's thesis title in Czech:

**ešení áste n pozorovatelných stochastických her pomocí omezeného horizontu**

Guidelines:

Partially observable stochastic games (POSGs) are a general class of games that allow model dynamic strategic interactions without a strict limit on the length of the interaction. However, finding optimal strategies for POSGs is an undecidable problem. One possible approach is to approximate POSGs as games with finite horizon and solve this bounded game instead. The goal of the student is to:

- (1) Implement 3 different games based on POSGs into the OpenSpiel framework.
- (2) Evaluate the quality of strategies computed by different algorithms on horizon-bounded games compared to the quality of strategies computed on POSGs. Focus on sampling-based or reinforcement learning based algorithms that do not need to construct the whole game tree.
- (3) Evaluate the impact of different value estimations of states beyond the horizon on the quality of strategies and convergence of the algorithms.

Bibliography / sources:

- [1] Mar Lanctot et al. "OpenSpiel: A framework for reinforcement learning in games." arXiv preprint arXiv:1908.09453 (2019).
- [2] Shoham, Yoav, and Leyton-Brown, Kevin. "Multiagent systems." Cambridge Books (2009).
- [3] Horák, K., Bošanský, B., & P chou ek, M. (2017). Heuristic Search Value Iteration for One-Sided Partially Observable Stochastic Games. In AAI (pp. 558-564).

Name and workplace of bachelor's thesis supervisor:

**doc. Mgr. Branislav Bošanský, Ph.D. Artificial Intelligence Center FEE**

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **01.06.2023** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **19.02.2025**

doc. Mgr. Branislav Bošanský, Ph.D.  
Supervisor's signature

prof. Ing. Tomáš Svoboda, Ph.D.  
Head of department's signature

prof. Mgr. Petr Páta, Ph.D.  
Dean's signature

### III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

\_\_\_\_\_  
Date of assignment receipt

\_\_\_\_\_  
Student's signature

## Acknowledgement / Declaration

I extend my heartfelt gratitude to my supervisor, doc. Mgr. Branislav Božanský, Ph.D., for his invaluable guidance, support, and mentorship throughout the course of this work. I am grateful for his unwavering patience, encouragement, and dedication. His expertise and willingness to assist me enabled me to complete this work.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 24, 2024

.....

## Abstrakt / Abstract

V této práci implementujeme tři různé částečně pozorovatelné stochastické hry s možnými nekonečnými horizonty do frameworku OpenSpiel. Poté získáme strategie v implementovaných hrách, které mohou být nekonečné tím, že je aproximujeme jako hry konečné. Nakonec zhodnotíme kvalitu získaných strategií a dopad různých odhadů hodnot stavů za horizontem na jejich kvalitu a dopad odhadů na konvergenci použitých algoritmů. Implementované hry jsou Pursuit Evasion, Search Game a Patrolling Game. Všechny implementované hry představují možný bezpečnostně obraný scénář, kde se jeden agent snaží zabránit jinému agentovi ve vykonávání nějaké činnosti. V Pursuit Evasion se jeden hráč snaží chytit druhého hráče v dané oblasti. V Search Game se jeden hráč snaží zabránit druhému hráči v pohybu přes zóny. Ve hře Patrolling Game se snaží jeden hráč bránit graf před útokem druhého hráče. Pro odhadnutí nekonečné hry pomocí konečné hry, omezíme délku dané hry. Tím vzniknou nové koncové stavy, které odpovídají stavům, kde je dosažena maximální délka hry. Pokračování nekonečné hry za horizontem bude reprezentováno odměnami které hráči obdrží v nově vzniklých terminálních stavech. Tyto odměny odpovídají odhadům hodnot stavů za horizontem. K získání strategií budeme používat algoritmy MCCFR a IS-MCTS.

**Klíčová slova:** teorie her, aproximace hry s nekonečným horizontem, částečně pozorovatelná hra s omezeným horizontem, OpenSpiel

**Překlad titulu:** Řešení částečně pozorovatelných stochastických her pomocí omezeného horizontu

In this work, we will implement three different partially observable stochastic games with possibly infinite horizons into the OpenSpiel framework; then, we will compute strategies on the games by approximating them as finite games. Finally, we will evaluate the quality of computed strategies and the impact of different value estimations of states beyond the horizon on the quality of strategies and their impact on the convergence of used algorithms. The games are Pursuit Evasion, Search Game and Patrolling Game. All of the games simulate a possible defence scenario where one agent tries to prevent another agent from performing some activity. In Pursuit Evasion, one player tries to catch the other player in an area; in Search Game, one player tries to prevent the other player from moving through zones; and in Patrolling Game, one player tries to defend a graph from an attack of the other player. To approximate an infinite game as finite, we limit the length of the game; this will introduce new terminal states that correspond to the states where the maximum length is reached; the continuation of the infinite game beyond the horizon will be represented as rewards in the new terminal states. The rewards correspond to the value estimations of states beyond the horizon. To compute strategies, we will use MCCFR and IS-MCTS algorithms.

**Keywords:** game theory, infinite-horizon game approximation, finite-horizon partially observable game, OpenSpiel


## / Contents

<b>1 Introduction</b>	<b>1</b>
<b>2 General Game Theory</b>	<b>3</b>
2.1 Types of games . . . . .	3
2.2 Strategies . . . . .	6
<b>3 Used Algorithms</b>	<b>9</b>
3.1 Linear Programming for Normal-Form Games . . . . .	9
3.2 Value Iteration . . . . .	10
3.3 Best Response . . . . .	10
3.4 Exploitability . . . . .	10
3.5 IS-MCTS . . . . .	11
3.6 MCCFR . . . . .	11
<b>4 Games</b>	<b>13</b>
4.1 OpenSpiel . . . . .	13
4.2 Pursuit Evasion . . . . .	13
4.3 Search Game . . . . .	15
4.4 Patrolling Game . . . . .	16
<b>5 Games Implementation</b>	<b>17</b>
5.1 Pursuit Evasion Implemen- tation . . . . .	18
5.2 Search Game Implementation .	21
5.3 Patrolling Game Imple- mentation . . . . .	24
<b>6 Experimental Evaluation</b>	<b>28</b>
6.1 Approach Summary . . . . .	30
6.2 Pursuit Evasion . . . . .	31
6.3 Search Game . . . . .	34
6.4 Patrolling Game . . . . .	36
<b>7 Conclusion</b>	<b>40</b>
<b>References</b>	<b>42</b>
<b>A Domain configurations for experiments</b>	<b>45</b>
<b>B Code structure in OpenSpiel</b>	<b>48</b>

## Tables / Figures

<p><b>6.1</b> Pursuit Evasion game values using MCCFR..... 32</p> <p><b>6.2</b> MCCFR convergence on Pursuit Evasion games..... 32</p> <p><b>6.3</b> Pursuit Evasion game values using IS-MCTS..... 33</p> <p><b>6.4</b> IS-MCTS convergence on Pursuit Evasion games..... 34</p> <p><b>6.5</b> Search Game game values using MCCFR..... 35</p> <p><b>6.6</b> MCCFR convergence on Search Game games..... 35</p> <p><b>6.7</b> Patrolling Game game values using MCCFR..... 36</p> <p><b>6.8</b> MCCFR convergence on Patrolling Game games..... 37</p> <p><b>6.9</b> Patrolling Game game values using IS-MCTS..... 38</p> <p><b>6.10</b> IS-MCTS convergence on Patrolling Game games..... 39</p> <p><b>A.1</b> Configuration of Pursuit Evasion games used in experiments..... 45</p> <p><b>A.2</b> Configuration of Search Game games used in experiments..... 45</p> <p><b>A.3</b> Configuration of Patrolling Game games used in experiments..... 46</p>	<p><b>4.1</b> Pursuit Evasion example ..... 13</p> <p><b>4.2</b> Pursuit Evasion information set example..... 14</p> <p><b>4.3</b> Search Game example ..... 15</p> <p><b>4.4</b> Patrolling Game example..... 16</p> <p><b>5.1</b> Sequential game as simultaneous game..... 17</p> <p><b>5.2</b> Position indexes in Pursuit Evasion..... 18</p> <p><b>5.3</b> Pursuit Evasion action mapping example..... 19</p> <p><b>5.4</b> Pursuit Evasion rewards after reaching maximum length . 20</p> <p><b>5.5</b> Rewards file for Pursuit Evasion ..... 20</p> <p><b>5.6</b> Pursuit Evasion rewards after reaching maximum length with set rewards file..... 21</p> <p><b>5.7</b> Position indexes in Search Game ..... 22</p> <p><b>5.8</b> Search Game rewards after reaching maximum length ..... 22</p> <p><b>5.9</b> Rewards file for Search Game . 23</p> <p><b>5.10</b> Search Game rewards after reaching maximum length with set rewards file..... 23</p> <p><b>5.11</b> Search Game action mapping example..... 24</p> <p><b>5.12</b> Patrolling Game rewards after reaching maximum length . 25</p> <p><b>5.13</b> Rewards file for Patrolling Game ..... 25</p> <p><b>5.14</b> Patrolling Game rewards after reaching maximum length with set rewards file..... 26</p> <p><b>5.15</b> Graph configuration file for Patrolling Game..... 26</p> <p><b>5.16</b> Default graph for Patrolling Game ..... 27</p> <p><b>5.17</b> Patrolling Game action mapping example ..... 27</p> <p><b>6.1</b> Bounded game tree ..... 28</p> <p><b>6.2</b> Approach summary ..... 30</p> <p><b>A.1</b> Search Game games' initial states used in experiments..... 46</p>
--	---





<b>A.2</b>	Patrolling Game graphs used in experiments.....	47
<b>B.3</b>	Source code directory tree .....	48
<b>B.4</b>	Experiments data directory tree.....	49



# Chapter 1

## Introduction

When they hear the term game, most people imagine something like Chess, Ludo, Mau-Mau, Poker [8] or a video game. But in the eyes of Game Theory, a game can be so much more. Game theory can represent a wide variety of real-life scenarios and interactions between people [5], machines, animals or their interaction with different objects. For example, a scenario where an antelope faces a decision to stay in a place with limited food resources or to try to cross a dangerous river to reach more fertile lands can be represented as a game. Something more useful that can be represented as a game is how to distribute some limited resources to complete a task most efficiently [3] or how a night guard should patrol a building complex to prevent robbers from breaking in [1].

Some games can be played infinitely, and the players might not have information about some parts of the game; for example, the building complex can be patrolled by a night guard forever. Additionally, the guard does not know when someone will try to break in or if anyone will try to break in at all.

It is generally impossible to optimally solve an infinite game with imperfect information. However, many problems that we might want to solve using Game Theory can possibly require the game model to be infinite with imperfect information, so how can we solve such a game?

One possibility is to approximate the infinite game as a finite game because there are many algorithms for solving imperfect information finite games (for example, MCCFR [11] or IS-MCTS [4]); this finite game can be called a bounded game, and the length of the finite game is referred to as the horizon, meaning that everything in the finite game is before the horizon and everything that was in the infinite game but is not in the finite game is beyond the horizon.

The artificial horizon of the bound game requires assigning value estimations to the states (equal to the rewards for players if the game reaches that state) that are terminal in the bound game but were not terminal in the infinite game. Changing these value estimations can model game behaviour beyond the horizon. For example, if in some artificial terminal state of the bounded game, one player had a higher chance of succeeding if the game continued infinitely, the reward for that player in that state would be higher. Value estimation of the states can be evaluated heuristically, but in this work, we will obtain these value estimations by using an algorithm for solving perfect-information infinite games.

With large game domains and algorithms requiring large amounts of operations, it is necessary to implement games in some programming language so the computations can be done by a computer. To solve a game, we will need, besides the game's implementation, some algorithms and possibly support code to process the results. Implementing all of that is time-consuming, and there is a chance of errors in newly implemented features. For this reason, it is wise to use code already implemented and tested by someone else. This presents a problem with the compatibility of different parts of code written by different people. Different frameworks exist to solve these problems and make research in game theory more convenient.

One of the frameworks for research in Game Theory is OpenSpiel [12]. OpenSpiel provides a general interface for implementing new games and algorithms compatible with each other, with plenty of games and algorithms already implemented and systematically tested. OpenSpiel also provides additional support functionalities, for example, graphviz [6] for visualizing games. Additionally, the core of OpenSpiel is written in C++, making it more time-efficient. This makes OpenSpiel a suitable choice for implementing new game domains.

The goal of this work is to implement three partially observable, possibly infinite games, Pursuit Evasion, Search Game, and Patrolling Game, into the OpenSpiel framework. Then, compare strategies computed on the bounded games with strategies on infinite games, and then evaluate the impact of different state value estimations beyond the horizon on the quality of strategies and the convergence of the algorithms.

The structure of this work is following. Chapter 2 summarizes general game theory related to this work. Chapter 3 briefly introduces algorithms used for the computation. Chapter 4 describes the rules and mechanics of the implemented games. Chapter 5 presents details about the implementation of the games. Chapter 6 contains the game solution concept and presents the results of the use of different algorithms and value estimations of states beyond the horizon. Chapter 7 summarizes and concludes the results of this work. Additionally, Appendix A expends configurations of the games used in the experiments, and Appendix B displays the structure of the implemented code in the OpenSpiel framework.

# Chapter 2

## General Game Theory

Game theory is part of the mathematics that studies the behaviour, relations and strategies of agents in different environments. Game theory is a relatively young discipline that emerged in the 20th century with significant contributions by John von Neumann.

In game theory, one of the fundamental terms is game. A game is an entity representing some environment, some agents, relations between the agents and agents' interactions with the environment.

Games can be as simple as a game of rock-paper-scissors or as complicated as an entire stock market. Even the entire human society could be represented as a game. The main source for this chapter is [21].

### 2.1 Types of games

Games can be divided into multiple groups based on many properties, such as the number of players, distribution of rewards, length of a game, information available to players, and more. One of the properties used in this work to group games is the order in which agents or players take their actions, dividing games either as sequential or simultaneous.

**Definition 2.1. (Sequential game)** A sequential game is a game where players take turns to take an action.

In other words, in a sequential game, one player takes an action, and the current state of the game is updated, then another player takes an action, and the state of the game is updated again; this happens until the game ends. Players can take turns fairly, meaning every player takes one action before any player takes a second action, or they can take actions in any order regardless of the fairness of a game. Although sequential games can cover a lot of situations, they are impractical for representing the real world, where multiple things can happen at the same time. To represent these cases, simultaneous games come into play.

**Definition 2.2. (Simultaneous game)** A simultaneous game is a game where all players take an action at the same time.

In other words, in a simultaneous game, all players pick an action; after that, the current game state updates correspondingly to all picked actions; this repeats until the end of the game.

Games can also be differentiated based on their representation. One of the simpler representations is a game in a normal form.

**Definition 2.3. (Normal-form game)** A normal-form game is a tuple  $G = \{N, A, u\}$  where:

- $N$  is a finite set of  $n$  players indexed by  $i$ ;
- $A = A_1 \times A_2 \times \dots \times A_n$  is a finite set of actions where  $A_i$  is a set of actions available to player  $i$ ;
- $u = (u_1, u_2, \dots, u_n)$  where  $u_i : O \mapsto \mathbb{R}$  is the payoff function for player  $u_i$ , where  $O$  is a set of all possible outcomes of the game.

A game in a normal form can be rewritten into a matrix. The matrix has one dimension for each player, and the number of elements in a dimension corresponds to the number of actions of the player represented by that dimension. Each element of the matrix contains values of all payoff functions  $u_i(O)$ ,  $i \in N$  given the outcome  $O$  when the players play actions that are represented by the element coordinates. This representation is called a matrix-form game.

Normal-form representation is useful for representing simpler games, but it has one significant drawback. Because a set of actions available to a player must be finite, it is impossible to use the normal form to represent games with infinite horizons, e.g. games that can be infinitely long. To grasp such games, another representation is needed.

**Definition 2.4. (Perfect-information extensive-form game)** A perfect-information game in extensive form is a tuple  $G = \{N, A, H, Z, \chi, \rho, \sigma, u\}$  where:

- $N$  is a finite set of  $n$  players indexed by  $i$ ;
- $A$  is a single finite set of actions;
- $H$  is a possibly infinite set of nonterminal choice nodes;
- $Z$  is a possibly infinite set of terminal nodes where  $H \cap Z = \emptyset$ ;
- $\chi : H \mapsto 2^A$  is the action function which maps a set of actions  $a$  to each nonterminal choice node  $h \in H$ , where  $a \subseteq A$  and  $a \neq \emptyset$ ;
- $\rho : H \mapsto N$  is the player function, which maps to each nonterminal choice node a player that will choose an action there;
- $\sigma : H \times A \mapsto H \cup Z$  is the transition function, which maps a choice node and an action to a new choice or terminal node if  $h_1, h_2 \in H$  and  $a_1, a_2 \in A$  and if  $\sigma(h_1, a_1) = \sigma(h_2, a_2)$  for all  $h_1, h_2, a_1, a_2$ , then  $a_1 = a_2$  and  $h_1 = h_2$ ;
- $u = (u_1, u_2, \dots, u_n)$  where  $u_i : Z \mapsto \mathbb{R}$  is a payoff function for player  $i$  in terminal nodes  $Z$ .

The extensive form is much more flexible for representing different types of games. Additionally, the extensive form can be rewritten into a game tree. Leave nodes in the tree correspond to terminal nodes  $Z$ , and all other tree nodes correspond to nonterminal nodes  $H$ . The root of the tree is a node where the game starts. Edges in the tree correspond to actions, such as, from a nonterminal node  $h \in H$  leads one edge for each action  $a \in \chi(h)$  to an edge  $h' = \sigma(h, a)$ .

The definition of the extensive-form game above presumes that all players have perfect information about a game, e.g. every player knows everything there is to know about the game. To expand the range of possibly representable games, imperfect information (e.g. some players may have limited information about a game) needs to be introduced into the extensive-form representation.

**Definition 2.5. (Imperfect-information extensive-form game)** A imperfect-information game in extensive form is a tuple  $G = \{N, A, H, Z, \chi, \rho, \sigma, u, I\}$  where:

- $G = \{N, A, H, Z, \chi, \rho, \sigma, u\}$  is a perfect-information game in extensive form;
- $I = (I_1, I_2, \dots, I_n)$  where  $I_i = (I_{i,1}, I_{i,2}, \dots, I_{i,k_i})$  is a set of equivalence classes on  $\{h \in H: \rho(h) = i\}$  when any two nonterminal choice nodes  $h_1, h_2 \in H$  are in same equivalence class  $I_{i,j}$  so  $h_1, h_2 \in I_{i,j}$  then following equations must be true  $\rho(h_1) = \rho(h_2)$  and  $\chi(h_1) = \chi(h_2)$  additionally if game is of perfect recall (*Definition 2.6.*) histories for both choice nodes must be equal.

The imperfect-information extensive-form game introduces the idea of the information sets or information states; both terms can be used interchangeably. An information set is a set of nodes that are indistinguishable from a player's point of view, so all those nodes are the same for the player.

Information sets can change the game tree for the players. Each player has a unique tree with information sets as tree nodes instead of nodes from  $h$ .

The perfect-information game in extensive form can be considered a special case of the imperfect-information game in extensive form, where all information sets contain only one node.

The definition of an imperfect-information extensive form game from above can be used for simultaneous games with slight modifications. The actions function  $\chi$  returns a set of sets of actions available to each player at a node, and the transition function  $\sigma$  takes as parameters a node and actions of all players. The player function  $\rho$  is not needed since every player takes an action at every node.

A partially observable stochastic game with an infinite horizon can be defined as an imperfect information game in extensive form with an infinite number of states. Formal definition follows.

**Definition 2.6. (Partially observable stochastic game with infinite horizon)** Partially observable stochastic game with infinite horizon is an imperfect-information game in the extensive form  $G = \{N, A, H, Z, \chi, \rho, \sigma, u, I\}$  with  $H$  and  $Z$  being infinite sets.

If the horizon of the partially observable stochastic game is finite, then the  $H$  and  $Z$  are finite sets.

The special case of a partially observable stochastic game with two players is a one-sided partially observable stochastic game; in such a game, one player has imperfect information, and the other player has perfect information. The formal definition follows.

**Definition 2.7. (One-sided partially observable stochastic game)** One-sided partially observable stochastic game is an imperfect-information game in the extensive form  $G = \{N, A, H, Z, \chi, \rho, \sigma, u, I\}$  with  $|I_{i,j}| > 1$  for one  $I_{i,j} \in I_i$  and  $|I_{i,k}| \geq 1$  for every other  $I_{i,k} \in I_i$  where  $I_i \in I$  for the imperfect-information player  $i$ ; and with  $|I_{p,k}| = 1$  for every  $I_{p,k} \in I_p$  where  $I_p \in I$  for the perfect-information player  $p$ .

In other words, in a one-sided partially observable stochastic game, the perfect-information player has exactly one state in each of his information sets, while the imperfect-information player can have multiple states in his information sets, but at least one of his information sets must contain more than one state, otherwise the game becomes the perfect-information game.

In a game, players cannot be certain of having information about their previous actions. If the players have that information in a game, the game is of perfect recall.

**Definition 2.8. (Perfect recall)** Let there be an imperfect-information game  $G = \{N, A, H, Z, \chi, \rho, \sigma, u, I\}$ , any two nodes  $h, h' \in H \wedge h, h' \in I_{i,j}$  where  $I_{i,j} \in I_i$  where  $I_i \in I$  and any two paths  $P$  and  $P'$  from the root  $h_0$  to  $h, h'$  respectively,  $P = h_0, a_0, h_1, a_1, h_2, \dots, h_m, a_m, h$  and  $P' = h_0, a'_0, h'_1, a'_1, h'_2, \dots, h'_{m'}, a'_{m'}, h'$ . Player  $i$  has perfect recall in the game  $G$  if for  $h, h'$  and  $P, P'$  is the following true:

- $m = m'$
- for all  $0 \leq k \leq m$ , if  $\rho(h_k) = i$  ( $h_k$  is decision node for player  $i$ ) then  $h_k, h'_k \in I_{i,l}$
- for all  $0 \leq k \leq m$ , if  $\rho(h_k) = i$  then  $a_k = a'_k$

Imperfect-information game  $G$  is of perfect recall if every player has a perfect recall in it.

Games can also be grouped by special properties of their payoff functions. The most interesting group is the group containing zero-sum games.

**Definition 2.9. (Zero-sum game)** A game  $G$  is a zero-sum game if, in every state of the game, rewards for all players sum to zero. Specially imperfect-information extensive-form game  $G = \{N, A, H, Z, \chi, \rho, \sigma, u, I\}$  is zero-sum if  $\sum_i^N u_i(h) = 0$  for every terminal node  $h \in Z$ .

The zero-sum property of a game is most powerful if the game has exactly two players. In that case, the rewards of players are opposite values; for example, if the first player receives the reward of  $-1$ , the second must receive the reward of  $+1$ . If the game is two-player and zero-sum, it simplifies the computation of many algorithms in the game.

## 2.2 Strategies

With the basics of the game representations defined, the next step is to describe the behaviour of agents or players in games. For clarification, player and agent have basically the same meaning of someone or something that picks actions in a game, so that these terms can be used interchangeably.

Players pick their actions according to strategies. A strategy is a set of rules that tells a player what action to pick at what point. There are possibly an infinite number of strategies for a player in a game.

Some simple strategies are, for example, random strategies, where a player picks randomly among available actions or pure strategies, where a player decides always to play one specific action in case of a normal-form game. Pure strategies get a little more complicated in the case of extensive-form games.

**Definition 2.10. (Pure strategies in imperfect-information extensive-form game)** Let  $G = \{N, A, H, Z, \chi, \rho, \sigma, u, I\}$  be an imperfect-information extensive-form game. All pure strategies for player  $i$  in  $G$  are Cartesian product  $\prod_{I_{i,j} \in I_i} \chi(I_{i,j})$ .

The definition above implies that a pure strategy in an extensive-form game must pick an action in every information state regardless of the state's reachability. Pure strategy in a perfect-information extensive-form game can use the same definition if we consider the perfect-information extensive-form game as a special case of an imperfect-information extensive-form game where every information state contains exactly one real state.



Being limited to pure strategies significantly reduces the ability of players to pick actions. Therefore, there is a concept of picking actions based on some probability distributions.

**Definition 2.11. (Mixed strategy in a normal-form game)** A set of mixed strategies for player  $i$  in a normal-form game  $G = \{N, A, u\}$  is  $S_i = \Pi(A_i)$  where  $\Pi(A_i)$  is a set of all probability distributions over  $A_i$ .

In other words, if a player uses a mixed strategy, he picks actions according to their probability. For example, if a mixed strategy in a game tells a player to pick the action  $a_1$  with probability  $p(a_1) = 1/3$  and the action  $a_2$  with probability  $p(a_2) = 2/3$ , then every time the player should play an action he rolls a six-sided dice if he rolls 1 or 2 he will play action  $a_1$ . If he rolls a number greater than 2, he will play action  $a_2$ .

In extensive-form games, mixed strategies randomize over all pure strategies of a game instead of actions.

**Definition 2.12. (Mixed strategies in imperfect-information extensive-form game)** Mixed strategies for player  $i$  in an imperfect-information extensive-form game  $G = \{N, A, H, Z, \chi, \rho, \sigma, u, I\}$  are  $\Pi(\Pi_{I_{i,j} \in I_I} \chi(I_{i,j}))$  where  $\Pi(\Pi_{I_{i,j} \in I_I} \chi(I_{i,j}))$  is a set of all probability distributions over all pure strategies  $\Pi_{I_{i,j} \in I_I} \chi(I_{i,j})$  of player  $i$ .

For extensive-form games, there is another type of strategy called behavioural strategy that picks actions based on probability distributions in every single state of a game.

**Definition 2.13. (Behavioural strategies)** Behavioural strategies for player  $i$  in an imperfect-information extensive-form game  $G = \{N, A, H, Z, \chi, \rho, \sigma, u, I\}$  is a Cartesian product  $\Pi_{I_{i,j} \in I_i} (\Pi(\chi(I_{i,j})))$  where  $\Pi(\chi(I_{i,j}))$  is a set of all probability distributions over all actions  $\chi(I_{i,j})$ .

The key difference between behavioural and mixed strategies is that for mixed strategy, there is one probability distribution that picks pure strategy to play. For behavioural strategy, every node or information state has its own probability distribution over its actions.

Generally, mixed strategies and behavioural strategies do not cover the same outcomes in a game. But if the game is of perfect recall, mixed strategies and behavioural strategies produce the same outcomes and can replace each other.

In an extensive-form game of perfect recall, a mixed strategy can be understood as a probability distribution over terminal nodes. Each terminal node represents a path from the root to that terminal node.

To convert behavioural strategy to mixed strategy, all we need to do is multiply all the probabilities on the path from the root to the terminal node for every terminal node. That gives us probabilities for reaching all terminal nodes that correspond to a mixed strategy with the same outcomes as the original behavioural strategy.

To convert mixed strategy to behavioural strategy, we need to find probability distributions for every node so that if we multiply all probabilities along a path from the root to a terminal node, we get the same probability as was given by the original mixed strategy for every terminal node.

To describe the strategies of multiple players in a game, we use the term strategy profile. A strategy profile is a set containing at max one strategy from every player in a game. The formal definition of mixed strategy profile follows.

**Definition 2.14. (Mixed strategy profile)** A set of mixed strategy profiles in a game is the Cartesian product of mixed strategies of all individual players  $S_1 \times S_2 \times \dots \times S_n$ .

After defining strategies, it would be useful to be able to compare and rate the quality of different strategies. One of the instruments for that is the best response.

**Definition 2.15. (Best response)** Best response of player  $i$  to the given strategy profile  $s_{-i} = \{s_k | k \neq i \wedge k \leq n\}$  (strategies of all players other than  $i$ ) is a mixed strategy  $s_i^* \in S_i$  that satisfies  $u_i(s_i^*, s_{-i}) \geq u_i(s_i, s_{-i})$  for all  $s_i \in S_i$ .

And last but not least, we may want to solve games. A solution for one player could be to find the best response to other players' strategies. However, the players usually do not know the strategies of their opponents. Another question is what to consider as a solution for a game considering all the players? For that, we need to introduce Nash equilibrium.

**Definition 2.16. (Nash equilibrium)** A strategy profile  $s = (s_1, s_2, \dots, s_n)$  is a Nash equilibrium if  $s_i \in s$  is best response to  $s_{-i}$  for all players  $i = (1, 2, \dots, n)$ .

In other words, nash equilibrium is equality among players in the sense that no player can improve his outcome by diverting from his strategy.

To solve a game usually means to find some Nash equilibrium in the game and its corresponding strategies for all players and their payoffs.

# Chapter 3

## Used Algorithms

This chapter briefly introduces algorithms and methods used to obtain values and strategies for players in games. The following algorithms and methods will be introduced in the variants for two-player zero-sum games without chance nodes since this work does not use any other types of games.

### 3.1 Linear Programming for Normal-Form Games

A game in normal form can be solved (e.g. found Nash equilibrium) by constructing a linear program. For the normal-form game  $G = \{N, A, u\}$ , the value of the game and mixed strategy for player 2 can be obtained by solving the following linear program.

$$\begin{aligned}
 \min \quad & U_1^* \\
 \text{s.t.} \quad & \sum_{k \in A_2} u_1(a_1^j, a_2^k) \cdot s_2^k \leq U_1^* & \forall j \in A_1 \\
 & \sum_{k \in A_2} s_2^k = 1 \\
 & s_2^k \geq 0 & \forall k \in A_2
 \end{aligned}$$

Similarly, strategies for player 1 can be computed by solving the dual of the linear program above. The dual follows.

$$\begin{aligned}
 \max \quad & U_1^* \\
 \text{s.t.} \quad & \sum_{j \in A_1} u_1(a_1^j, a_2^k) \cdot s_1^j \geq U_1^* & \forall k \in A_2 \\
 & \sum_{j \in A_1} s_1^j = 1 \\
 & s_1^j \geq 0 & \forall j \in A_1
 \end{aligned}$$

After solving the equations, the value of the game for player 1 is equal to  $U_1^*$ , and because the game is zero-sum, the value for player 2 is equal to  $-U_1^*$ . A mixed strategy for player 1 is in variables  $s_1^j$  and for player 2 in variables  $s_2^k$ .

The value of the game for player 1, if player 2 picks actions randomly, can be computed as  $U_1^* = \max_{j \in A_1} \sum_{k \in A_2} u_1(a_1^j, a_2^k) / |A_2|$ , where  $|A_2|$  is the total number of actions available to player 2. Similarly, if player 1 plays randomly  $U_1^* = \min_{k \in A_2} \sum_{j \in A_1} u_1(a_1^j, a_2^k) / |A_1|$ , where  $|A_1|$  is number of actions available to player 1. It corresponds to solving the linear programs if  $s_2^k = 1/|A_2|$  for  $\forall k \in A_2$  or  $s_1^j = 1/|A_1|$  for  $\forall j \in A_1$ .

The main source for this section is [21].

### 3.2 Value Iteration

Value iteration is used to compute the values of individual states in games with perfect information. For the purposes of this work, the definition will be modified to work for simultaneous games.

Value iteration repeatedly iterates over all states and, for every state, updates the state's expected value  $Q_k^*(s)$  in the current iteration  $k$  with the formula  $Q_k^*(s) = LP(s)$ .

Where  $LP(s)$  returns a reward for player 1 in the state  $s$  if the state  $s$  is a terminal state; otherwise it constructs a game in the normal form  $G = \{\{1, 2\}, A = A_1 \times A_2, u\}$  with  $A_1$  and  $A_2$  containing all actions available to player 1 and player 2 in the game state  $s$  respectively. And with  $u(a_1, a_2) = c \cdot Q_{k-1}^*(s')$ , where  $a_1 \in A_1; a_2 \in A_2; Q_{k-1}^*(s')$  is an expected value in previous iteration of a state  $s'$  that was reached by applying actions  $a_1$  and  $a_2$  in the state  $s$ ;  $c$  is a constant called discount factor. After the normal-form game  $G$  is constructed, it is solved using linear programming and the value of the game  $U_1^*$  is returned. To compute values in case one player picks actions randomly, the game  $G$  will be solved with the appropriate variation of the linear program.

Value iteration ends when error  $e$  is smaller than a predefined threshold after finishing an interaction over all states. Error  $e$  is computed as  $e = \max_{s \in S} |(Q_k^*(s) - Q_{k-1}^*(s))|$ , where  $S$  is a set of all states of the game;  $(Q_k^*(s))$  is an expected value of a state  $s$  in current iteration; and  $Q_{k-1}^*(s)$  is an expected value of a state  $s$  in previous iteration.

### 3.3 Best Response

The value of the best response  $v_0$  can be obtained by solving the following linear program.

$$\begin{aligned} & \text{minimize} && v_0 \\ & \text{subject to} && v_{\mathcal{J}_1}(\alpha_1) - \sum_{I' \in \mathcal{J}_1(Ext_1(\alpha_1))} v_{I'} \geq \sum_{\alpha_2 \in \Gamma_2} g_1(\alpha_1, \alpha_2) r_2(\alpha_2) \quad \forall \alpha_1 \in \Gamma_1 \end{aligned}$$

Where  $v_0$  is an expected value if the player 1 plays the best response from the root information state,  $\alpha_1$  and  $\alpha_2$  are sequences of actions for player 1 and 2 respectively,  $\Gamma_1$  and  $\Gamma_2$  is a set of all action sequences for player 1 and 2 respectively,  $v_{\mathcal{J}_1}(\alpha_1)$  is a value of the player's 1 best response from an information set reach by action sequence  $\alpha_1$ ,  $\mathcal{J}_1(Ext_1(\alpha_1))$  is a set of player's 1 information sets reachable by playing one additional action after the action sequence  $\alpha_1$ ,  $v_{I'}$  is a value of best response from information set  $I'$ ,  $g_1(\alpha_1, \alpha_2)$  is a reward player 1 obtain if he plays action sequence  $\alpha_1$  and if the player 2 plays action sequence  $\alpha_2$ , and  $r_2(\alpha_2)$  is a probability of player 2 playing action sequence  $\alpha_2$ . The main source for this section is [21].

### 3.4 Exploitability

Exploitability is used to measure the closeness of the strategy to the Nash equilibrium. In our case, the closer the value of the Exploitability to the 0, the closer the strategy to the Nash equilibrium. The exploitability is computed with the following formula.  $NashConv(s) = \sum_i^n \max_{s_i' \in \Sigma_i} u_i(s_i', s_{-i}) - u_i(s_i)$ , where  $\max_{s_i' \in \Sigma_i} u_i(s_i', s_{-i})$  is a value of player  $i$  playing his best response on strategies of other players,  $u_i(s_i)$  is a value of player's  $i$  strategy and  $n$  in number of players. The definition is taken from [10].

### 3.5 IS-MCTS

Information set Monte Carlo tree search (IS-MCTS) is a variant of Monte Carlo tree search (MCTS) that is modified to be able to handle imperfect information games. IS-MCTS searches game trees of information states instead of normal states, as a classic variant of MCTS does. The main source for this section is [4].

The algorithm is given the information set  $I$  for which we want to obtain strategy; the algorithm then iterates a fixed number of times (= maximum iteration) over the information state  $I$ ; every iteration expands the game tree that the algorithm builds. The game tree is initialized with only the root node corresponding to the information set  $I$ .

In each iteration, a random state  $s$  that is included in the information set  $I$  is picked, and then four phases *SELECTION*, *EXPANSION*, *SIMULATION* and *BACKPROPAGATION* are run in the exact order with state  $s$  as a starting point.

In the *SELECTION* phase, a path from the state  $s$  is traversed down the actual game tree until the reached state  $s'$  is not terminal or an information set that contains it is not part of the game tree that the algorithm builds. Actions during the traversal are picked according to the formula:

$$a = \operatorname{argmax}_{a \in \chi(s')} Q^*(s', a) + c \cdot \sqrt{\ln(v_{s'})/v_{s',a}}$$

Where  $Q^*(s', a)$  is the average expected value of the information state reachable from  $s'$  using action  $a$ ,  $c$  is an exploration constant,  $v_{s'}$  is equal to the total number of times information set containing state  $s'$  was visited during the run of the algorithm and  $v_{s',a}$  is equal to the total number of times the information set reachable from  $s'$  using the action  $a$  was visited.

The *EXPANSION* phase happens when during the *SELECTION* phase a state  $s'$  is reached such that for any action  $a \in \chi(s')$  the  $v_{s',a} = 0$ , in that case of all actions that satisfy  $v_{s',a} = 0$  one action  $a_w$  is picked randomly. Then, an information set reachable from  $s'$  using action  $a_w$  is added to the game tree the algorithm builds with both the total number of visits and the expected value initialized to 0.

The *SIMULATION* phase runs a game simulation from the lastly expanded state  $s_e$  that is reached from the state  $s'$  using action  $a_w$ . In the game simulation, all actions are picked randomly until a terminal state is reached and reward  $r$  is obtained from it.

The *BACKPROPAGATION* phase starts when reward  $r$  is obtained. During this phase, for each visited information set from information set containing  $s$  to information set containing  $s_e$ , increment the total number of visits by one and update an information state's average expected value according to  $r$ . The average expected value is equal to the sum of all rewards propagated back through the information set divided by the total number of the information set visits.

After all iterations are done, the behavioural strategy for the initial information set  $I$  is obtained by averaging the number of visits of information sets reachable from  $I$ . To obtain a behavioural strategy for an entire game, the algorithm is run for every state of the game.

### 3.6 MCCFR

In this work, we will use the Monte Carlo counterfactual regret minimization (MCCFR) variant called external-sampling MCCFR; in the rest of the work, we will refer to this

variant as MCCFR. MCCFR is an iterative algorithm that does not traverse the entire game tree in each iteration. The main source for this section is [11].

The game tree is divided for each player into blocks. There is one block for each pure strategy of the opposite player; each block contains all terminal histories  $z \in Z$  (equal to terminal states in games with perfect recall) that are reachable if the opposite player plays the corresponding pure strategy. In each iteration of the algorithm, a sub-iteration is performed for each player. At the beginning of the sub-iteration, one block  $Q$  is picked. The probability of picking block  $Q$  is equal to  $\prod_{I \in I_{-i}} \sigma_{-i}(\tau(I)|I)$ , where  $I_{-i}$  are information sets of the opposite player,  $\tau(I)$  is an action, picked in information set  $I$  according to the corresponding pure strategy, and  $\sigma_{-i}(\tau(I)|I)$  is a probability the opposite player will pick action  $\tau(I)$  in information set  $I$  if he follows his behavioural strategy (equal to the policy  $\sigma_{-i}$ ) for the current iteration.

In a sub-iteration, the algorithm traverses the game tree sampling actions for each history  $h$  (equal to non-terminal states in games with perfect recall) where  $\rho(h) \neq i$  and for such visited information set  $I$  computes sampled counterfactual regrets according to the following formula.

$$r^*(I, a) = (1 - \sigma(a|I)) \sum_{z \in Q \cap Z_I} u_i(z) \pi_i^\sigma(z[I]a, z)$$

Where  $\sigma(a|I)$  is a probability of playing action  $a$  in an information set  $I$  if all players play accordingly to their behavioural strategies for the current iteration,  $Z_I$  is a set of terminal histories reachable from histories included in  $I$ ,  $u_i(z)$  is a reward for player  $i$  in terminal history  $z$ ,  $z[I]a$  is a prefix of the terminal history  $z$  from root to information set  $I$  followed by action  $a$ , and  $\pi_i^\sigma(z[I]a, z) = \pi_i^\sigma(z) / \pi_i^\sigma(z[I]a)$  if  $z[I]a$  is a prefix of  $z$  or zero otherwise, with  $\pi_i^\sigma(z)$  being the probability of reaching  $z$  if player  $i$  plays according to the policy  $\sigma$  and  $\pi_i^\sigma(z[I]a)$  being the probability of reaching  $z[I]$  and picking action  $a$  if player  $i$  plays according to the policy  $\sigma$ .

The sampled counterfactual regrets  $r^*(I, a)$  are then added to the total regrets for each pair  $(I, a)$ . The new policy (same as a behavioural strategy)  $\sigma$  for the next iterations is computed by averaging the total regrets of action in each information set formula follows.

$$\sigma_i^{t+1}(a, I) = R(I, a) / \sum_{a' \in \chi(I)} R(I, a')$$

Where  $R(I, a)$  is a total regret for action  $a$  in information set  $I$  and  $\chi(I)$  is a set of all the available actions in  $I$ . After the algorithm ends, a final strategy is created in the same way.

# Chapter 4

## Games

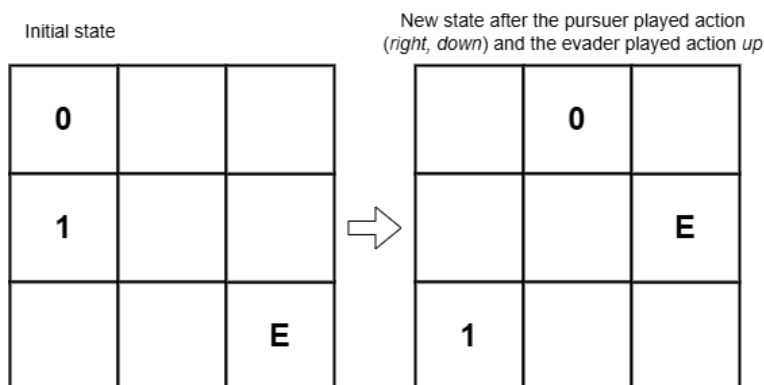
Games used for purposes of this work are Pursuit Evasion, Search Game and Patrolling Game; the game definitions were taken from [9]. All listed games are two-player, simultaneous move, imperfect information, and zero-sum games of perfect recall with infinite horizon. The bounded variants of these infinite games are implemented in OpenSpiel [12].

### 4.1 OpenSpiel

OpenSpiel [12] is a framework developed under DeepMind. It is a collection of algorithms and game environments for research in game theory, especially in reinforcement learning. OpenSpiel supports a wide variety of features. There is support for single-player and multi-player games, perfect and imperfect information games, zero-sum, general-sum and cooperative games, and sequential and simultaneous games. The main focus is on sequential games represented as extensive-form games, which is a preferred form for implemented games, although games can be transformed into matrix-form games. Simultaneous games are missing some features compared to sequential games, mainly support for imperfect information. OpenSpiel core is implemented in C++, including the games, but some games are implemented in Python as well. Most of the algorithms are implemented in both C++ and Python. Parts implemented in C++ are available in Python using pybind11. Games can be visualised using graphviz [6].

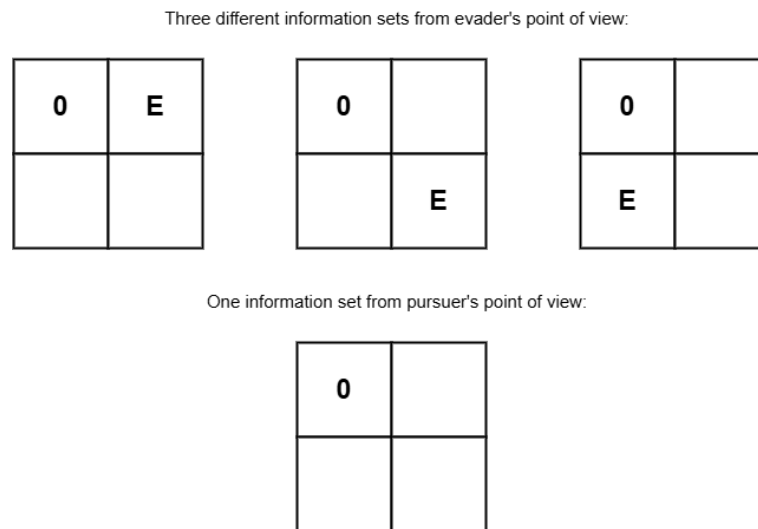
### 4.2 Pursuit Evasion

Pursuit Evasion is a two-player game played on a chess board-like grid. Where one player controls a group of pursuers, and the other player controls an evader. The group of pursuers consist of  $k$  game pieces indexed from 0 to  $k - 1$ . The evader game piece is indexed with 'E'.



**Figure 4.1.** Pursuit Evasion example with width and height equal to 3 and with 2 pursuer game pieces.

All game pieces can move *up*, *down*, *left*, *right* or possibly *wait*. Waiting means that a game piece will not change its position. The player controlling the pursuers must control all pursuer game pieces with one action, so action for this player is a cartesian product of all available moves of pursuer game pieces. For example, if there are two pursuer game pieces, one with available actions *up* and *right* and the other with actions *down* and *right* pursuer player will be able to choose from four actions:  $(up, down)$ ,  $(up, right)$ ,  $(right, down)$ ,  $(right, right)$  with the first action in the action pair corresponding to game piece with index 0 and the second action corresponding to game piece with index 1. Pursuer game pieces can not stand in the same position but can switch positions with each other.



**Figure 4.2.** Pursuit Evasion information set example with width and height equal to 2 and with 1 pursuer game pieces.

The pursuer player does not have information about the position of the evader game piece. The pursuer player only knows the initial position of the evader game piece. The evader player has perfect information about the game.

Game pieces always start at the same positions regardless of game configurations. The evader game piece starts in the bottom left corner. The pursuer game pieces start at the left side of the game grid, with a game piece with index 0 being in the top left corner and all other pursuer game pieces below it in order corresponding to their index. The starting position example can be seen in Figure 4.1.

The goal of the game is for the pursuer player to catch the evader game piece. The evader game piece is caught if a pursuer game piece steps on the same field in the game grid or if the evader game piece switches position with a pursuer game piece, this occurs if the pieces stand directly next to each other and both play action contrary to each other in the correct direction. The goal of the evader player is to avoid being caught for as long as possible. The game is played until the evader game piece is caught.

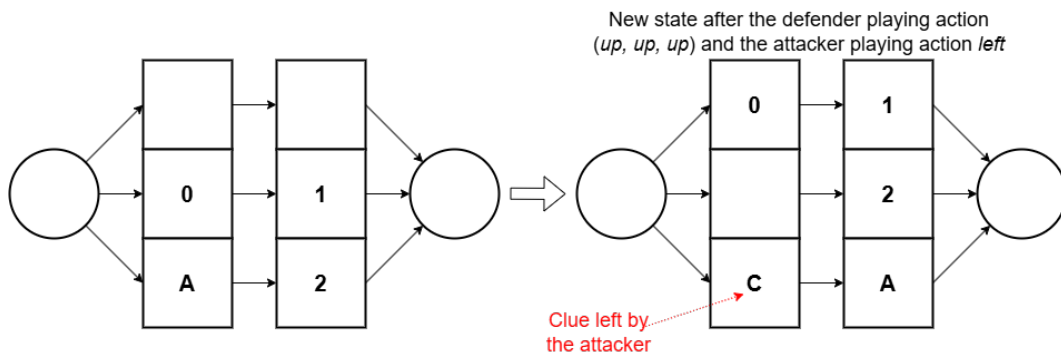
Pursuit Evasion game has multiple changeable configurations. Width and height for changing the size of the game grid, the number of pursuer game pieces in the grid, and lastly, whether game pieces can *wait* instead of changing position. Configuration can not be changed during a game.



### 4.3 Search Game

Search Game is a two-player game. One player controls an attacker game piece, and the other player controls a group of  $k$  defender game pieces, each with an assigned index from 0 to  $k - 1$ . The game is played on a game field composed of zones stacked next to each other in a row. A zone is a column of nodes where game pieces can stand. All zones are of the same size except for two special zones, the first one (start node for the attacker) and the last one (goal node for the attacker), which always contain only one node.

The attacker game piece starts in the first special zone. Until the attacker game piece stays in this zone, it can decide to move in any node in the following zone in the row (which is the first normal zone) or *wait* and does not change its position. Once the attacker game piece leaves the first special zone by entering the first normal zone, it can move *up* or *down* in the current zone or move *left* into the following zone. By moving *left*, the attacker game piece will end up at the same height in the new zone as it was in the zone before; for example, if the attacker game piece was in the top node in the old zone, after moving *left*, it will be in the top node in the new zone. The only exception is when the attacker game piece is in the last normal zone; then, by moving *left* from any node, it enters the second special zone and the game ends.



**Figure 4.3.** Search Game example with 2 zones and height equal to 3 and with 3 defender game pieces (Figure does not show an initial state).

When the attacker game piece enters any node in a normal zone, it leaves a clue in that node. The clue stays in the node until the attacker game piece *waits* in that node for at least one turn. The *wait* move removes the clue from the node until the attacker game piece reenters the node again.

The defender game pieces start anywhere in the normal zones. They can only move *up*, *down* or *wait* in the same node. This means that a defender game piece can not leave the zone it started in. Two defender game pieces can not stand in the same node, but they can switch places.

The goal of the attacker is to move the attacker game piece through all the zones, i.e. reach the last special zone, without getting caught by a defender game piece. The goal of the defender is to catch the attacker game piece. The defender catches the attacker game piece if he moves one of the defender game pieces to the node where the attacker game piece stands.

The defender does not have information about the position of the attacker game piece, but if one of the defender game pieces moves to a node where the attacker game piece left a clue, it discovers the clue, which gives the defender information that the

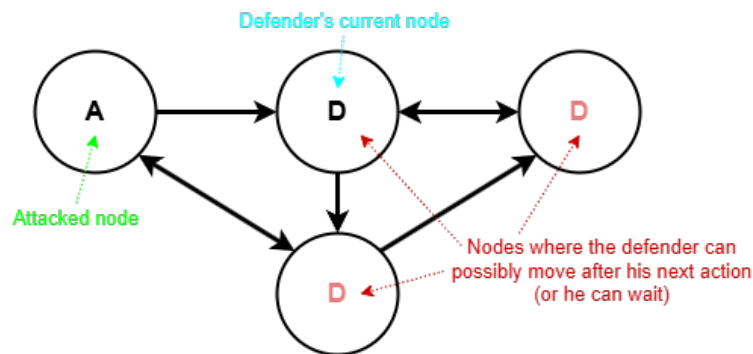
attacker game piece visited the node in the past. The attacker has perfect information about the game.

The Search Game configuration is defined by the number of normal zones, the number of nodes in a normal zone, the number of defender game pieces and their initial positions.

## 4.4 Patrolling Game

Patrolling Game is a two-player game. One player controls an attacker game piece, and the other player controls a defender game piece. The game is played in a graph. The graph is composed of nodes and edges. Node is a place where game pieces can stand. Edge is a representation of the one-way path from one node to another. Each node has a set of edges that start in it and go to other nodes.

The defender game piece starts in a node, and each turn can move from a node  $A$  to the node  $B$  if there is an edge starting in the node  $A$  that goes to the node  $B$ . Alternatively, the defender game piece can wait in its current node.



**Figure 4.4.** Patrolling Game example (Figure does not show an initial state).

The attacker game piece starts outside the game graph. The attacker game piece can wait and not change its position or move to any node of its choosing as long as it remains outside of the graph. Once the attacker game piece moves to a node, it starts an attack on that node. When the attack starts, the attacker game piece can not change its position, meaning it has to wait for the attack to finish. The attack finishes after a set number of turns.

The goal of the defender is to catch an attacker game piece. The attacker game piece is caught if the defender game piece moves to a node that is under unfinished attack. This means the attacker game piece can not be caught if it has not started an attack yet. The goal of the attacker is to successfully complete an attack on any node.

The defender does not have information about the position of the attacker game piece, so the defender does not know whether an attack started and when the attack starts, the defender does not know which node is targeted. The attacker has perfect information about the game.

The Patrolling Game configuration is defined by the graph, the starting position of the defender game piece and the length of an attack. For the game to be meaningful, every node in the graph should be reachable from every node in the length of the attack, and the attack should take fewer turns than it takes the defender game piece to go through all nodes. The conditions listed before are not necessary, but the game then becomes trivial.

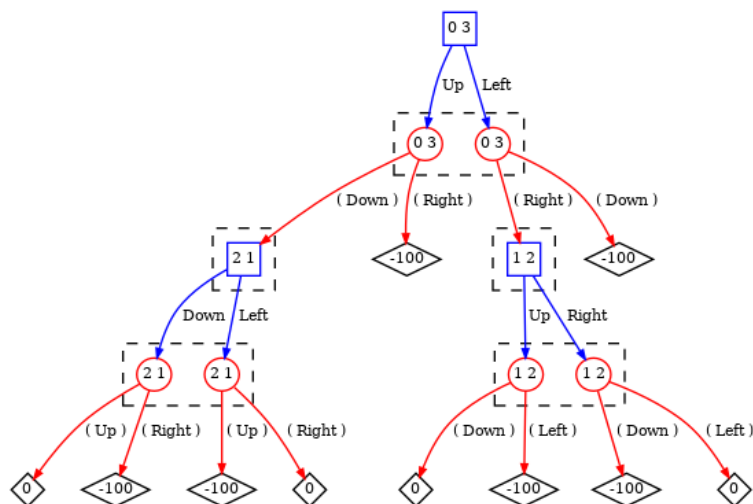
# Chapter 5

## Games Implementation

In OpenSpiel, all game implementations are derived from the *Game* superclass for sequential games and from the *SimMoveGame* superclass for simultaneous move games. The superclasses provide generic interfaces for any specific game implementation. Games are accessed only through the superclass interface. Therefore, there is not much space for custom game methods that are visible outside the game implementation. Furthermore, games cannot be initialized directly; game initialization is handled by *LoadGame* methods. Alternatively, a game can be initialized via shared pointer *shared\_ptr<const Game>*. All game states also derive from the *State* superclass, which provides a common interface for them, similar to the superclasses for games [12].

Actions in OpenSpiel are represented as integers ranging from zero to the number of all actions available to all players. Every integer in that interval must represent an action. For example, in a two-player game, if one player can play three different actions and the second player can play two different actions, the actions will be represented by integers from zero to four. So, every action must be consistently mapped to an integer. Furthermore, every instance of the same game should have exactly the same mapping between actual actions and their integer representations.

Due to OpenSpiel's limited support for simultaneous games, Pursuit Evasion, Search Game and Patrolling Game are implemented as sequential games. To satisfy the simultaneous nature of the games, a game state changes after both players have their turn. This means that if the first player plays an action, the game state will remain unchanged. After the second player plays an action, both actions are applied simultaneously, and the game state progresses (example tree is shown in Figure 5.1).



**Figure 5.1.** Example game tree of the sequential game representing the simultaneous game.

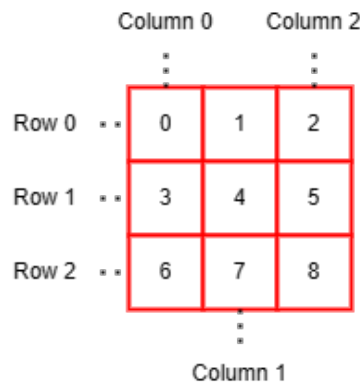
## 5.1 Pursuit Evasion Implementation

A Pursuit Evasion game is represented by parameters defined during the initialization of the game. After the game is created, these parameters can not be changed; if any parameter is not defined during the creation process, the default parameter value will be used.

The parameters with values equal to integers are *width*, *height*, *num\_pursuers*, *pursuer\_reward* and *max\_game\_lenght* with default values: 3 for *width* and *height*, 2 for *num\_pursuers*, 100 for *pursuer\_reward* and with 1000 for *max\_game\_lenght*. Then, there are parameters with boolean values *agent\_can\_wait* and *short\_form* with default values *true* and *false*, respectively. The last parameter is called *rewards\_file* and is represented by a string with an empty string as a default value.

The first two parameters, *width* and *height*, modify the width and the height of the game board; in other words, *width* specifies how many columns will the game board have, and *height* specifies how many rows the game board will have.

A game piece position in the game board is specified by an integer ranging from 0 to  $width * height - 1$ . The position in the top right corner (0-th row and 0-th column) is assigned to integer 0. By moving right, the integer increases by one; by moving down, the integer increases by *width*. The position in the bottom right corner is indexed with  $width * height - 1$ . By moving left, the integer decreases by 1; by moving up, the integer decreases by *width*. The formula for calculating the position index in a board is  $currentRow * width + currentColumn$ , assuming rows and columns are indexed from zero. An example of position indexes for default game configuration is shown in Figure 5.2.



**Figure 5.2.** Position indexes for the default configuration of the Pursuit Evasion game.

To set the number of pursuer game pieces, there is a parameter called *num\_pursuers*. Pursuer game pieces are indexed from 0 to  $num\_pursuers - 1$  and are placed on positions in a game board corresponding to the formula  $startIndex_i = width * i$  where  $i$  is an index of a pursuer game piece. An evader game piece always starts at position  $startIndex_e = width * height - 1$ .

By default, game pieces can move up, down, left, right and wait if there is no restriction by environment (side of the board, the desired position is already occupied). By setting *agent\_can\_wait* = *false*, the wait move is removed from the possible move list, prohibiting waiting for the game pieces. All moves are represented as integers [*up* = 0, *down* = 1, *left* = 2, *right* = 3, *wait* = 4].

Actions for the players are generated from moves available for their game pieces. Available actions for the evader player are  $[0, 1, 2, 3, 4]$  if waiting is permitted or  $[0, 1, 2, 3]$  if waiting is prohibited; the action number directly corresponds to the move number available to the evader game piece. Actions for the pursuer player are generated from permutations of moves available for pursuer game pieces; for example, action  $[up, down]$  means that the 0-th pursuer game piece will move *up* and the 1-st pursuer game piece will move *down*.

Action numbers for pursuer start from 5 if waiting is permitted or 4 if waiting is not permitted. An action number is then calculated with the following formula:  $actionNumber = m + \sum_{j=0}^k m_j * m^{k-j}$ , where  $k = num\_pursuer - 1$ ,  $m_j$  is a move of a  $j$ -th game piece and  $m = 4$  if waiting is prohibited or  $m = 5$  if waiting is allowed. For example, in a game with two pursuers game pieces and waiting prohibited, the action number for action  $[left, down]$  will be  $4 + \sum_{j=0}^1 m_j * 4^{1-j} = 4 + (2 * 4^1) + (1 * 4^0) = 13$ . Action mapping for default game configuration can be seen in Figure 5.3.

Actions:			
0	Up	15	( Left Up )
1	Down	16	( Left Down )
2	Left	17	( Left Left )
3	Right	18	( Left Right )
4	Wait	19	( Left Wait )
5	( Up Up )	20	( Right Up )
6	( Up Down )	21	( Right Down )
7	( Up Left )	22	( Right Left )
8	( Up Right )	23	( Right Right )
9	( Up Wait )	24	( Right Wait )
10	( Down Up )	25	( Wait Up )
11	( Down Down )	26	( Wait Down )
12	( Down Left )	27	( Wait Left )
13	( Down Right )	28	( Wait Right )
14	( Down Wait )	29	( Wait Wait )

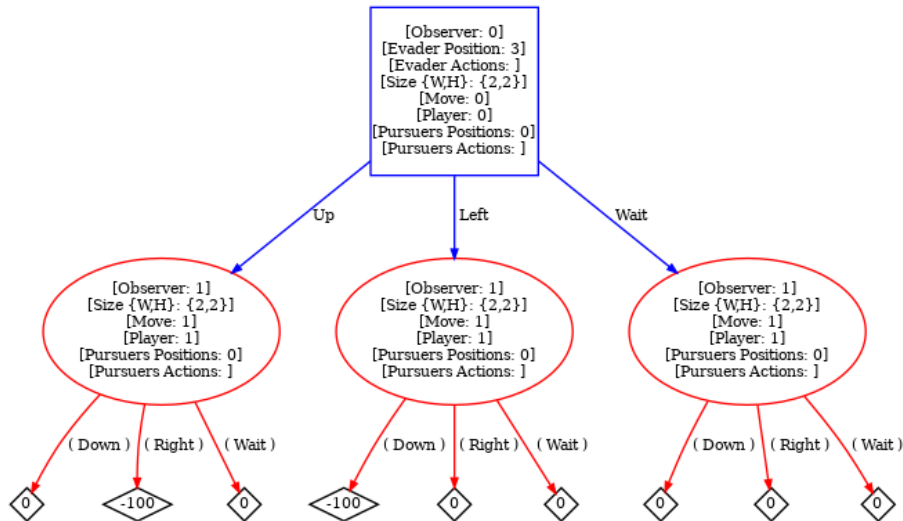
**Figure 5.3.** Mapping of action numbers for Pursuit Evasion game with waiting permitted.

The reward for catching the evader game piece can be set with *pursuer\_reward* parameter. It is the reward that the pursuer player will get. The evader player will be rewarded with  $-pursuer\_reward$ .

The maximal length of a game must be known from the moment the game was created. Parameter *max\_game\_lenght* sets the maximum number of actions players can take before the game ends. Each player can take  $max\_game\_lenght/2$  actions in total. After *max\_game\_lenght* is reached, all states become terminal. In that case, rewards in the states where the pursuer did not catch the evader game piece will be 0 for both players. Rewards in the states where the evader game piece was caught will stay the same (example in Figure 5.4).

To modify rewards for players in all terminal states, including the ones where the evader game piece was not caught, a parameter *rewards\_file* can be used. The parameter is a path to a file with specified rewards for states (example in Figure 5.6). The file format is the following: The first line contains a state in short form, and the next line contains the reward of the evader in that state. Then, on the third line, follow the next state and the reward in that state. This continues for every state of the game. Figure 5.5 shows an example of the file format.

The *short\_form* parameter switches the game state's *toString()* method to print state representation in a short form. The short form has the format of  $k + 1$  integers



**Figure 5.4.** Example Pursuit Evasion game tree with rewards set to 0 after the game reaches maximum length.

```

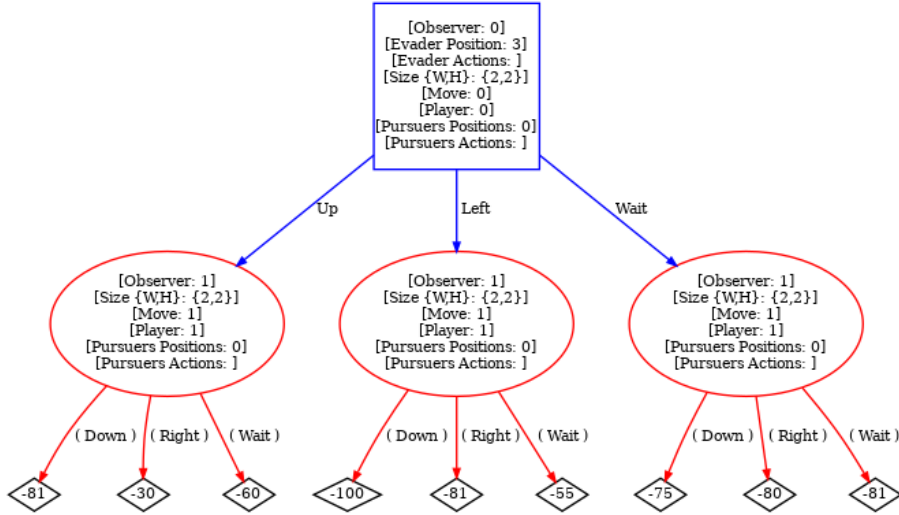
1  0 3  ————— State in short form
2  -40.864708283623536 — Reward in state 0 3
3  2 1
4  -40.88911745805866
5  1 1
6  -100.0
7  2 2
8  -100.0
9  1 2
10 -40.889119091311045

```

**Figure 5.5.** Example of Pursuit Evasion rewards file format.

separated by white spaces, where the first  $k$  integers correspond to positions of pursuer game pieces according to their index, with  $k$  being the number of pursuers. The last integer in the short form is the position of an evader game piece. For example, the short form '0 3 5' tells that there are pursuer game pieces on positions 0 and 3, and there is an evader game piece on position 5. The short form is perfect information representation without perfect recall.

Representation of Pursuit Evasion's information state for pursuer contains information about a current player, actions played, positions of pursuer game pieces and a history of pursuer actions. Information state for evader contains the same properties as for pursuer with the addition of evader game piece position and history of evader actions.



**Figure 5.6.** Example Pursuit Evasion game tree with rewards set by reward file after the game reaches maximum length.

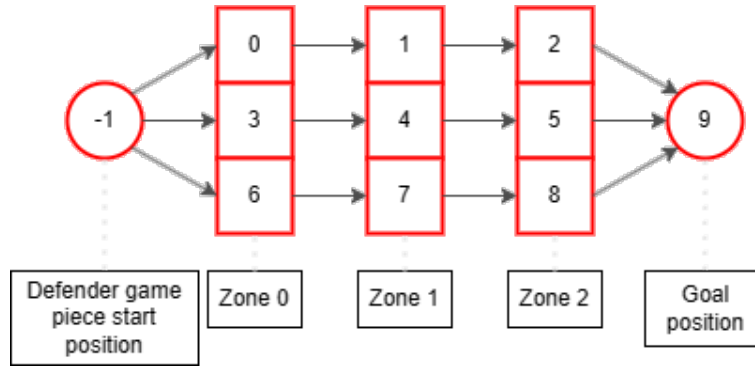
## 5.2 Search Game Implementation

An instance of a Search Game is represented by its parameters. The parameters must be defined during the initialization of the game instance. If any parameters are not defined, their default values will be used. Once the game is initialized, the parameters cannot be changed.

The Parameters *width*, *height*, *reward\_attacker\_win*, *reward\_attacker\_loss* and *max\_game\_length* are integers with default values 3 for *width* and *height*, 100 for *reward\_attacker\_win*,  $-100$  for *reward\_attacker\_loss* and with 1000 for *max\_game\_length*. The Parameters *defender\_init\_positions* and *rewards\_file* are strings with default values '0:1\_2:1' for *defender\_init\_positions* and an empty string for *rewards\_file*. The last parameter, *short\_form*, is a boolean with the default value of *false*.

The parameters *width* and *height* modify the size of the game board. The *width* sets the number of zones in a game, and the *height* sets the height of the zones. The position in a game board is represented as a single integer. The integer is calculated as  $height_{zone} * width + idx_{zone}$ , where  $height_{zone}$  is a position in a zone with the top position in the zone being 0, the second position from the top in the zone is 1 and so on, the most bottom position in the zone is  $height - 1$ . The  $idx_{zone}$  is an index of a zone; the most left zone has index 0 the next zone to right has index 1 and so on, the most right zone has index  $width - 1$ . This applies to the normal zones where defender game pieces can operate. There are two special zones: the zone where the attacker starts and the goal zone. The integer for the attacker start zone is always  $-1$ . The integer for the goal zone is equal to  $width * height$ . For example, in a game with  $width = 3$  and  $height = 3$ , the integer of position in the bottom of the second zone (index 1) is  $2 * 3 + 1 = 7$ . An example of position indexes for default game configuration is shown in Figure 5.7.

The parameter *defender\_init\_positions* sets the initial positions of the defender game pieces. It's format is  $d_0^{zone_{idx}}:d_0^{zone_{height}} \dots d_{k-1}^{zone_{idx}}:d_{k-1}^{zone_{height}}$ , where  $k$  is a number of defender game pieces,  $d_j^{zone_{idx}}$  is an index of a zone where  $j$ -th defender should start and  $d_j^{zone_{height}}$  is a position in the zone. For example,

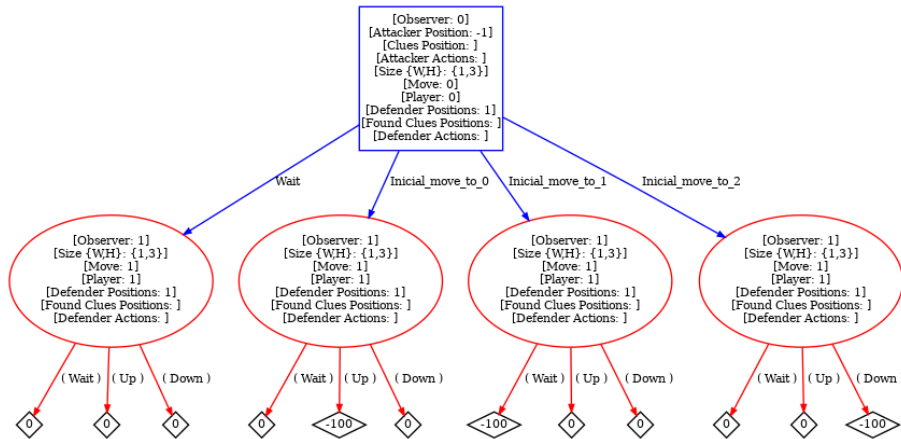


**Figure 5.7.** Position indexes for the default configuration of the Search Game game.

$defender\_init\_positions = '0:1\_2:0'$  means that there will be two defenders. The defender game piece with index 0 will start in the first zone (the most left zone, zones are indexed from 0) in the second position from the top (also indexed from 0). The defender game piece with an index of 1 will start in the top position of the third zone.

The parameter  $reward\_attacker\_win$  is a reward an attacker will receive if the attacker game piece reaches the goal zone,  $reward\_attacker\_loss$  is a reward the attacker will receive if the defender catches the attacker game piece. The defender will receive the opposite value of the attacker's rewards.

The parameter  $max\_game\_length$  sets an upper limit at the number of actions that can be played by players. Each player can take  $max\_game\_length/2$  actions in total. After this limit is reached, all game states become terminal regardless of the position of the attacker game piece. In that case, players in all states where the attacker game piece was not caught nor reached the goal zone will receive a reward equal to 0 (example in Figure 5.8).



**Figure 5.8.** Example Search Game game tree with rewards set to 0 after the game reaches maximum length.

A rewards file can be used to modify rewards in all terminal states, including the ones that originated from reaching the maximum length of a game (example in Figure 5.10). The file is set via the  $rewards\_file$  parameter that contains the path to the file. The format of the reward file is as follows: The first line is a game state in short form, followed by the reward in that state. On the next line is another state with its reward on the following line. This repeats for all states. Figure 5.9 shows an example of the file format.



```

1  -1; 0;; ————— State in short form
2  20.250000016251118 — Reward in state -1; 0;;
3  1; 0; 1;
4  44.9999999703412
5  1; 1;; 1
6  0.0
7  1; 1; 0 1;
8  0.0
9  0; 0; 1; 0
10 0.0

```

Figure 5.9. Example of Search Game rewards file format.

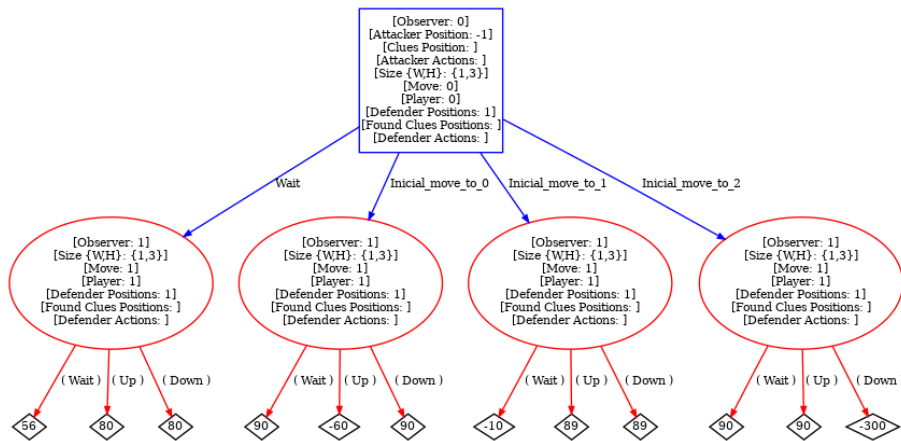


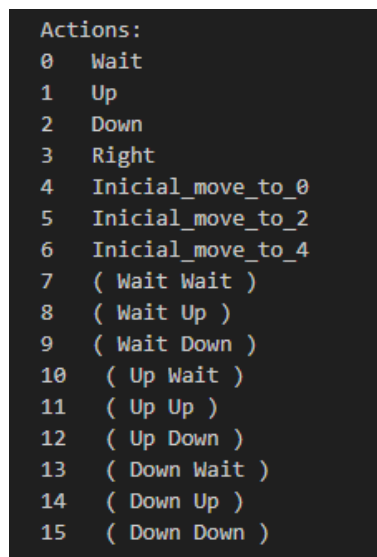
Figure 5.10. Example Search Game game tree with rewards set by reward file after the game reaches maximum length.

The parameter `short_form = true` switches game state method `toString()` to return state representation in a short form. The short form is a sequence of integer white spaces and semicolons. The short form starts with the position of the attacker game piece followed by a semicolon, then follows the positions of all defender game pieces in order corresponding to their index, then a semicolon followed by positions of placed clues and then a semicolon followed by found clues. If no clue is placed or found, there will be only semicolons without integers. For example, in a game with `width = 3` and `height = 3` short form `'-1; 0 8;'` the attacker game piece is in the start zone, and there are two defender game pieces, one on the top of the first zone, the other on the bottom of the second zone and no clues were placed nor found. If the short form was `'1; 0 8; 0; 1'`, then the defender game pieces would be at the same positions, the attacker game piece would be in the top position of the second zone, and there would be an unfound clue on the position of the attacker game piece and a found clue at the position of the 0-th defender game piece. The short form is a perfect information view of a state.

Information state for the defender contains the information about the positions of the defender game pieces, the history of the defender's actions, the positions of found clues and the number of actions taken. An information state for the attacker contains the same information as for the defender; additionally, it contains information about the

attacker's position, the attacker's action history and information about the positions of unfound clues.

Game piece moves are represented as integers [ $wait = 0, up = 1, down = 2, right = 3$ ], defender game pieces do not have access to the *right* move. Actions for the attacker are mapped into integers as follows: actions from 0 to 3 correspond to the moves of the attacker game piece. Actions from 4 to  $height + 3$  correspond to initial moves into the first zone; action 4 will move the attacker game piece to the top position in the first zone, action 5 will move it to the position under it and so on. Actions for defender are mapped to integers from  $height + 4$  to the total number of actions. The action number is calculated from the permutation of all defender game pieces moves; the formula is  $actionNumber = height + 4 + \sum_{j=0}^k m_j * 3^{k-j}$ , where  $k$  is the number of defender game pieces minus one (indexed from 0) and  $m_j$  is move number of  $j$ -th defender game piece. For example, in a game with  $height = 3$  for action [ $up, wait, down$ ] (three defender game pieces), the action number would be  $3+4+(1*3^2)+(0*3^1)+(2*3^0) = 7+9+0+2 = 18$ . Action mapping for default game configuration can be seen in Figure 5.11.



Action Number	Move Description
0	Wait
1	Up
2	Down
3	Right
4	Inicial_move_to_0
5	Inicial_move_to_2
6	Inicial_move_to_4
7	( Wait Wait )
8	( Wait Up )
9	( Wait Down )
10	( Up Wait )
11	( Up Up )
12	( Up Down )
13	( Down Wait )
14	( Down Up )
15	( Down Down )

**Figure 5.11.** Mapping of action numbers for Search Game game with height three and with two defenders.

### 5.3 Patrolling Game Implementation

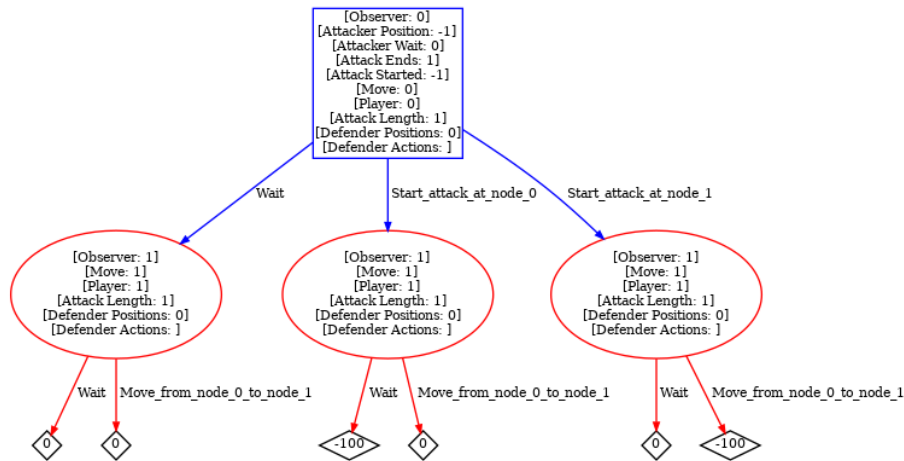
An instance of a Patrolling Game is represented by its parameters. The parameters define key properties of a game, so they must be defined during game initialization and can not be changed after that. The default value will be used if any of the parameters is not specified.

The integer parameters for the game are *max\_game\_length*, *attacker\_reward*, *attack\_lenght*, the boolean parameters are *short\_form* and *finish\_attack*, and the string parameters are *graph\_file* and *rewards\_file*. The default value for *max\_game\_length* is 1000, for *attacker\_reward* is 100, for *attack\_lenght* is 4, for both *short\_form* and *finish\_attack* is *false* and for both *graph\_file* and *rewards\_file* is an empty string.

The *attacker\_reward* parameter sets the reward the attacker receives when he successfully finishes an attack; in that case, the defender receives  $-attacker\_reward$ . If

the attack is finished unsuccessfully (the defender catches the attacker game piece), the attacker will receive a reward equal to  $-attacker\_reward$ , and the defender will receive  $attacker\_reward$ .

The *max\_game\_length* parameter sets the maximum number of actions players can take in total, meaning each player can take up to  $max\_game\_length/2$  actions. After a game reaches its maximum length, all states become terminal regardless of the attacker game piece's state; if that happens and the attack did not end, both players receive a reward of 0; there is an exception: if parameter *finish\_attack* = *true* and the attacker has already started an attack, the game will continue till the end of the attack (example in Figure 5.12).



**Figure 5.12.** Example Patrolling Game game tree with rewards set to 0 after the game reaches maximum length.

The parameter *rewards\_file* can be used to change rewards in all terminal states, including the ones that originated from reaching maximum game length (example in Figure 5.14). The parameter specifies a path to the rewards file. In the file, the first line contains the game state in short form, followed by the reward in that state. On the third line, there is another state in a short form, followed by its rewards on the next line; this repeats for all states. Figure 5.13 shows an example of the file format.

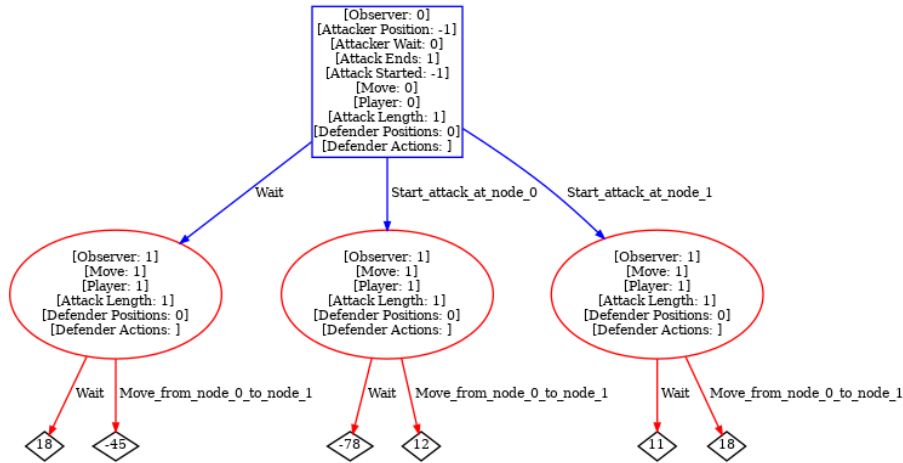
```

1   -1 0 2   _____ State in short form
2   12.149999983188913 — Reward in state -1 0 2
3   -1 1 2
4   12.149999983189078
5   -1 2 2
6   12.149999983188902
7   0 0 2
8   0.0
9   0 1 2
10  1.7980824249697472e-10

```

**Figure 5.13.** Example of Patrolling Game rewards file format.

The parameter *short\_form* switches state method *toString()* to return state representation in a short form. The short form format for a state is the following:  $'p_a p_d$

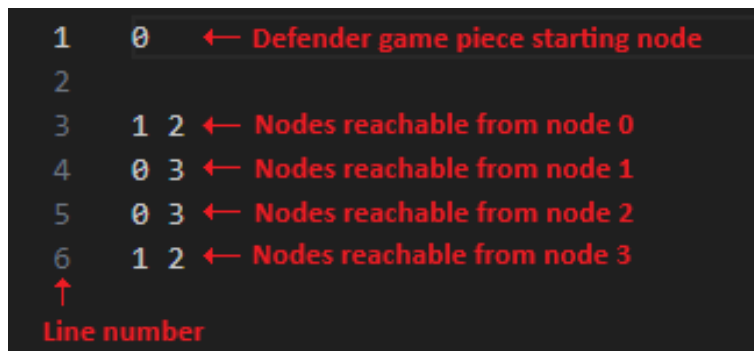


**Figure 5.14.** Example Patrolling Game game tree with rewards set by reward file after the game reaches maximum length.

$attack_{end}$ , where  $p_a$  is the position of an attacker game piece,  $p_d$  is the position of a defender game piece and  $attack_{end}$  is the number of the attacker turns to finish an attack. The short form is the perfect information view of a state without perfect recall.

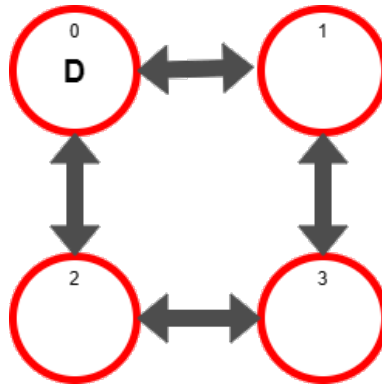
The parameter  $attack\_length$  sets how many turns it takes for an attacker to finish an attack on a node. The attack's remaining turns are decreased during the turn of the attacker, so the attack lasts for  $attack\_length$  turns of the attacker, so it is not influenced by the defender's turns. For example, if  $attack\_length = 3$ , it will take 3 attacker turns to finish an attack; this means it will take 6 turns in a game (3 for attacker and 3 for defender).

The parameter  $graph\_file$  specifies a path to the file containing the definition of a graph to use for a game. The graph file is in the following format: the first line of the file is the starting position of a defender game piece represented as an index of a node. The second line is empty. The third line contains indexes of nodes reachable from the node with index 0. The fourth line contains indexes of nodes reachable from the node with index 1. Every node in the graph has a line specifying nodes that are reachable from it; the index of a node is equal to the number of lines preceding its line minus two. Example of graph file can be seen in Figure 5.15. The default graph will be used if the  $graph\_file$  parameter is not set (see Figure 5.16).



**Figure 5.15.** Graph file format for default graph in Patrolling Game.

The integer representation of actions is the following: the action *Wait* is assigned 0 for both players. The attacker's actions to start an attack are assigned



**Figure 5.16.** Default graph for Patrolling Game, where D is the position of the defender game piece and the numbers are indexes of nodes.

integers from 1 to the number of nodes in a graph in order of the node index, meaning action *Start\_attack\_at\_node\_0* will have assigned number 1, action *Start\_attack\_at\_node\_1* will have assigned number 2 and so on. The defender player's actions to move his game piece in a graph are assigned integers from  $1 + t$  to  $1 + t + e$ , where  $k = \text{atotalnumberofnodes}$  and  $e = \text{atotalnumberofedges}$ . Action numbers are bound to the edges they represent and are assigned when the graph is parsed from a graph file. The parsing of edges starts from the top left of the file so that the action numbers are assigned according to the order of the edges in the file. Action mapping for default game configuration can be seen in Figure 5.17.

```

Actions:
0  Wait
1  Start_attack_at_node_0
2  Start_attack_at_node_1
3  Start_attack_at_node_2
4  Start_attack_at_node_3
5  Move_from_node_0_to_node_1
6  Move_from_node_0_to_node_2
7  Move_from_node_1_to_node_0
8  Move_from_node_1_to_node_3
9  Move_from_node_2_to_node_0
10 Move_from_node_2_to_node_3
11 Move_from_node_3_to_node_1
12 Move_from_node_3_to_node_2

```

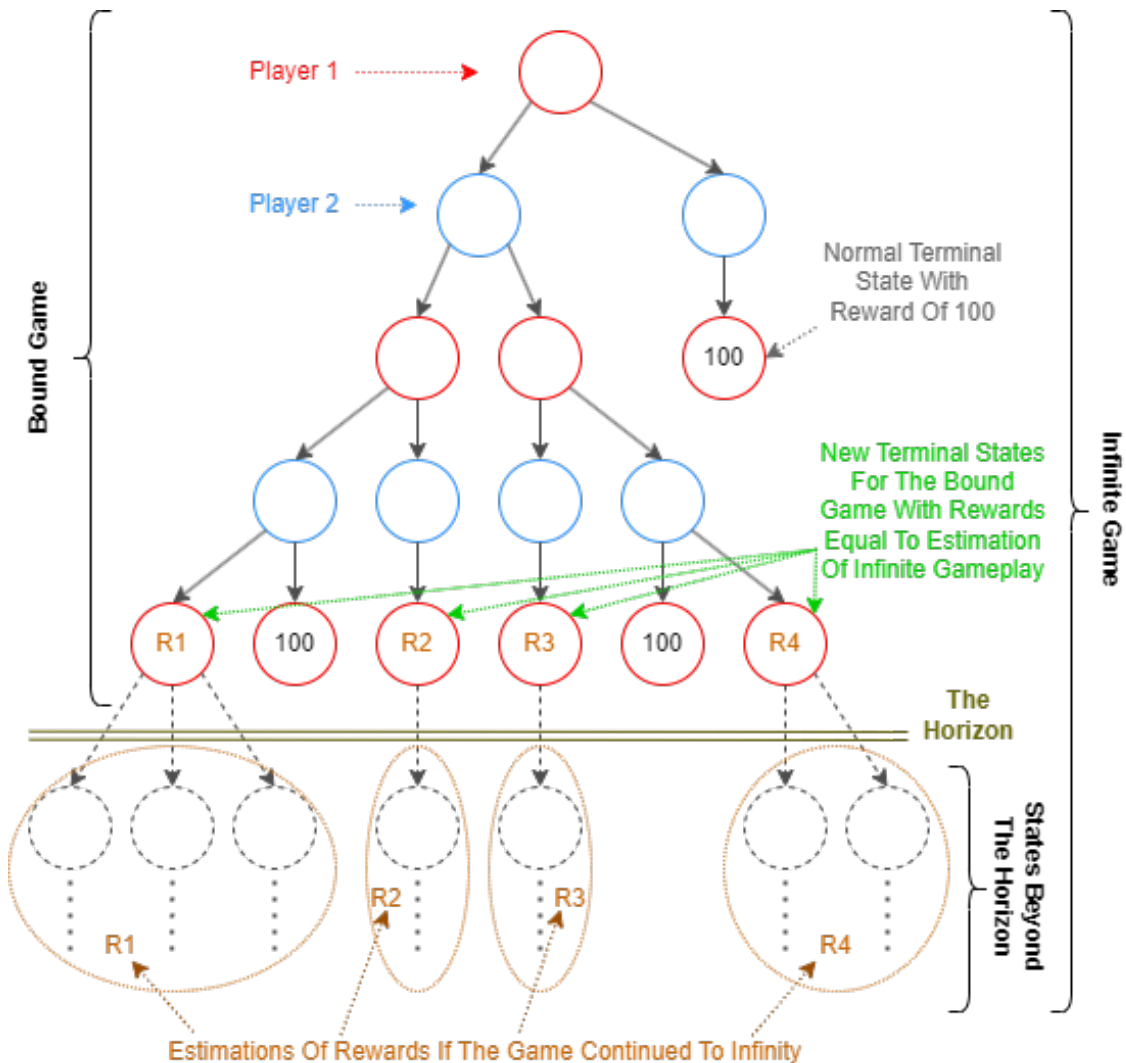
**Figure 5.17.** Mapping of action numbers for Patrolling Game game with default graph.

Information state for a defender contains information about the defender game piece's position, history of the defender's actions and number of turns from the beginning of a game. The information state for the attacker contains the same information as for the defender, with additional information such as the attacker's position, the number of turns the attacker waited for an attack, the turn the attacker started the attack and the remaining time to finish the attack.

# Chapter 6

## Experimental Evaluation

It is impossible to directly evaluate strategies in imperfect-information extensive-form games with an infinite horizon. However, such games can be approximately solved using their finite horizon estimation. To estimate an infinite game as a finite game, the infinite game must be bound to some maximum length. After the bounded game reaches its end, players will receive rewards equal to the estimated reward they would receive if the bounded game continued into infinity from a state in which the bounded game ended.



**Figure 6.1.** Game tree of infinite game estimated as a bounded game. Since games are zero-sum, rewards in the tree are represented as rewards for player 1.

One way to obtain the estimation of the values the game would have if it were played into infinity is by using value iteration; it can be done because value iteration ignores perfect recall, which makes a number of states of the game (at least for games used in this work) finite and thus makes game solvable. One disadvantage of using value iteration is that it does not work for imperfect information games; on the other hand, using the perfect information variant of the game negates the problem of losing the perfect recall.

Solving the perfect information variant of the imperfect information game can be used to obtain estimations, but the estimations will probably be inclined in favour of a player who gained additional information. Let us refer to the estimations obtained this way by the symbol  $V$ .

In this work, the problem above can be negated by using the fact that in the used games, only one player has imperfect information, while the other has perfect information. If the imperfect information player is forced to pick actions randomly, it does not matter what information he gains. By solving a game where an imperfect information player picks actions randomly, we obtain value estimations that are close to the values of the infinite game where the imperfect information player plays randomly as well.

This approach, unfortunately, manifests the opposite problem, where the value estimations are more inclined to favour the perfect information player. Let us refer to the estimations obtained this way by the symbol  $RN$ .

By solving the perfect-information variant of the infinite game with value iteration, we have obtained the estimated values of states in such a game. Now, we substitute those state value estimations into the terminal states of the corresponding imperfect-information bounded game. The state value estimation will be substituted in the following way. When the imperfect-information bounded game ends in some information state, the player 1 will receive a reward equal to the value estimation of a state that corresponds to the state the bounded game is currently in, which are basically the same states except their histories. For example, in Pursuit-Evasion, the states correspond to each other if the positions of the defender game pieces and the positions of the evader game pieces are the same for both states, the histories of action that lead to these states are irrelevant. The player 2 will receive an opposite reward than player 1 since the games are zero-sum.

After substituting estimated values into a bounded game's terminal states, we compute the strategy of the imperfect information player in the bounded game. By doing so, we obtain a strategy of the imperfect information player for the first  $k$  moves in the infinite game that corresponds to the bounded game, where  $k$  is the number of actions the imperfect information player can play before the game reaches the depth of the bound, for deeper states (states after the length bound) the player need another strategy, in this work we will assume that the player will play randomly.

The strategy beyond the bound can be represented by substituting values into the terminal states of the bounded game in the same way as before. To represent a strategy where the imperfect-information player plays actions randomly after the bound, we substitute the  $RN$  state value estimations into the terminal states.

Strategies computed for bounded games with different value estimations in terminal states and their quality for the infinite game if the imperfect-information player played randomly in the states beyond the bound can be compared by computing the best response of the perfect-information player on the computed strategies in the bounded games with  $RN$  value estimations in terminal states. This way, we can obtain the value estimation of the infinite game where the imperfect-information player followed the

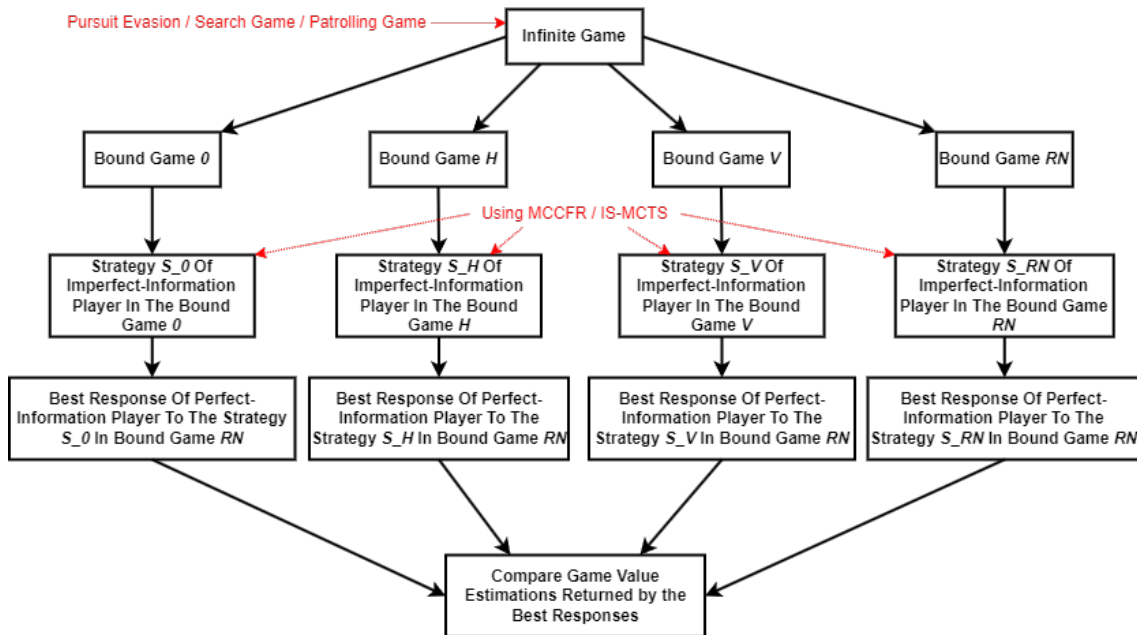
computed strategy and then played randomly. The obtained values can be compared to evaluate the quality of the strategies.

The diagram with all the steps from an infinite game to the value estimations of the different strategies in the infinite game can be seen in Figure 6.2.

## 6.1 Approach Summary

In this work, for each domain (Pursuit Evasion, Search Game, Patrolling Game), four bounded games will be used with the following estimations for states beyond bound:

- 0 - with this estimation, all states have a value equal to 0; this represents games where all players draw when the bound is reached
- $V$  - values of the states are computed using value iteration for perfect information variation of a game; this represents games where players play with perfect information beyond the bound
- $H$  - values of the states are equal to values for  $V$  multiplied by  $1/2$  except normal terminal states, which have unchanged values; this does not represent any specific scenario, but it is a middle ground between 0 and  $V$
- $RN$  - values of the states are computed using value iteration for perfect information variation of a game where imperfect information player picks actions randomly; this represents games where players play randomly after the bound



**Figure 6.2.** Summary of approach in experiments.

Strategies of imperfect information players (the algorithms produce strategies for both players, but the produced strategies of perfect information players are used only for computing the exploitability [= closes to Nash equilibrium] of the strategies) on bounded games will be computed using MCCFR and IS-MCTS. The convergence of the algorithms will be evaluated by comparing the exploitability of the strategies in their corresponding bounded game. For example, if there is a strategy computed by



MCCFR in the  $V$  game variant, the exploitability of that strategy will be computed in exactly the same  $V$  game variant.

Even though both MCCFR and IS-MCTS algorithms are not required to explore the entire game tree, their implementation in OpenSpiel creates the entire game tree and loads it into memory to create a strategy in every possible information set. This was the limiting factor for using bigger game configurations in this work's experiments. Implementing new variants of the algorithms is beyond the scope of this work.

The best responses of the perfect information player and value estimations of the infinite games will be computed on bounded game  $RN$ , meaning that the imperfect information player will pick actions randomly beyond the length bound.

In all domains, the perfect information player is the first player, and the imperfect information player is the second player.

## 6.2 Pursuit Evasion

Game configurations used for Pursuit Evasion experiments are shown in the format  $Wx_1_Hx_2_Px_3_Cx_4_Lx_5$ , where the capital letter represents a parameter, and  $x_i$  represents a value of the parameter with  $i \in \{1, 2, 3, 4, 5\}$ . With  $W$  represents *width*,  $H$  represents *height*,  $P$  represents *num\_pursuers*,  $C$  represents *agent\_can\_wait* with values  $0 = false$  and  $1 = true$ , and finally  $L$  represents *max\_game\_lenght* which is the length of the bounded game. The rest of the parameters are set to default values. More details regarding the configuration can be seen in Appendix A.

Value estimations of the states beyond the horizon were computed with value iterations with discount factor  $c = 0.9$  for  $V$  and  $H$  in smaller cases of  $R$  and with  $c = 0.99$  for larger cases of  $R$  to move the values of the states further from 0 because the action space is relatively large so defender picking actions randomly has a relatively low chance of catching the evader game piece. The maximum error for the value iteration was set to  $e = 1$ .

Table 6.1 shows values for different Pursuit Evasion game configurations with strategies computed using MCCFR. The values of the games are negative because the table shows the values of the player 1, who is the evader and who receives a reward of  $-100$  when his game piece is caught. This means that the pursuer (player 2) had a better strategy and was able to catch a defender game piece more reliably in cases with lower game values.

The first two lines in a table are special cases. The game  $W2_H2_P1_C0_L8$  represents a case where the optimal strategy for both players is to pick actions randomly; in that case, the evader game piece is caught with probability  $p = 1 - (1/2)^{L/2}$ , where  $L$  is a length of the game, this means that value of the game in the infinity should convert to  $-100$ . The results show that all game variants came close to that number; the small difference is a result of the value iteration terminating after the error is lower than 1. The game  $W2_H2_P1_C1_L8$  is a case where when the evader plays an optimal strategy, he can never be caught. Thus, the value of the game with any length is 0. Results show that all game variants reached the correct value.

The rest of the game configurations are more general cases that cannot be solved trivially. As expected, the best results were obtained with the  $RN$  game variants because the best response is computed on the same game variant, so the  $RN$  column is more of a reference column to show the most optimal results. The worst-performing game variants were  $V$ , which were outperformed even with 0 game variants. This result was slightly unexpected, but a possible explanation is that the value estimations of

Game configuration	0	H	V	RN
W2_H2_P1_C0_L8	-99.39	-99.30	-99.39	-99.39
W2_H2_P1_C1_L8	0	0	0	0
W3_H3_P2_C0_L6	-82.89	-83.47	-82.41	-84.67
W3_H3_P2_C0_L8	-86.90	-87.98	-84.22	-89.19
W3_H3_P2_C1_L6	-48.88	-49.85	-42.09	-54.12
W3_H3_P2_C1_L8	-65.51	-66.44	-61.30	-68.88
W3_H4_P2_C0_L8	-64.42	-65.92	-52.71	-67.36
W3_H4_P2_C0_L10	-77.32	-77.28	-71.38	-77.84
W3_H4_P2_C1_L8	-36.98	-39.76	-27.30	-42.57
W4_H3_P2_C0_L10	-77.70	-77.62	-71.24	-77.93
W4_H4_P2_C0_L10	-35.87	-35.67	-30.16	-40.27

**Table 6.1.** Pursuit Evasion game values for different value estimations of states beyond the horizon, with strategies computed using MCCFR. (Table shows estimated values of the first player = player with perfect information)

states in  $V$  are significantly overvalued, so a strategy computed on such a game tends to lean more towards such states. Because the best response is computed on the  $RN$  variant of the game, where the state estimations have lower values, the strategy will have a lower expected value because it will lean towards the now low-value states instead of the actual terminal states with high values meanwhile the  $0$  variants rely on the terminal states. This idea can be reinforced by the results of the  $H$  game variants, which, in some cases, outperformed the  $0$  variants. This means that  $H$  game variants do not have their value estimations overvalued as  $V$  variants do.

Game configuration	0 - iter   expl	H - iter   expl	V - iter   expl	RN - iter   expl
W2_H2_P1_C0_L8	1   0.00	4001   0.84	1   0.01	1   0.60
W2_H2_P1_C1_L8	5001   0.89	5001   0.81	5001   0.98	5001   0.86
W3_H3_P2_C0_L6	10001   0.97	8001   0.99	2001   0.74	5001   0.94
W3_H3_P2_C0_L8	30000   1.34	15001   0.92	4001   0.89	13001   0.70
W3_H3_P2_C1_L6	30000   2.06	28001   0.82	4001   0.96	29001   0.84
W3_H3_P2_C1_L8	30000   2.41	30000   1.93	15001   0.90	30000   1.91
W3_H4_P2_C0_L8	30000   1.67	24001   0.98	7001   0.95	29001   0.97
W3_H4_P2_C0_L10	30000   2.76	30000   2.04	24001   0.99	30000   2.03
W3_H4_P2_C1_L8	30000   2.18	30000   1.54	21001   0.99	30000   2.73
W4_H3_P2_C0_L10	30000   2.06	30000   1.61	28001   0.93	30000   1.83
W4_H4_P2_C0_L10	30000   4.19	30000   4.01	30000   1.77	30000   3.87

**Table 6.2.** Convergence of MCCFR algorithm for different value estimations of states beyond the horizon for Pursuit Evasion games (iter = iterations, expl = exploitability in the respective bounded games). The closer an exploitability is to zero, the closer a strategy is to Nash equilibrium.

MCCFR for all game configurations was limited to 30000 iterations, or it would end when it reached exploitability lower than 1. Exploitability was calculated after the first iteration and then after each 1000 iterations. Table 6.2 shows convergence and the number of run iterations to reach the convergence for every game configuration. As can be seen in the table,  $V$  game variants outperformed all other game variants by

reaching exploitability lower than 1 in all game configurations except the biggest one, where it still managed to converge closest to exploitability 1 of all the other variants.  $H$  and  $RN$  game variants performed more or less similarly. The worst performance convergence occurred in 0 game variants, converging below 1 exploitability only in the three smallest game configurations. A possible reason could be the fact that 0 game variants have sparse non-zero values, making it harder for MCCFR to find the optimal strategies. The opposite applies to the  $V$  game variants.

IS-MCTS algorithm for experiments on Pursuit Evasion game configuration was limited to 10000 iterations in each information set, and with exploration constant  $c = 100$ . High exploration constant proved to enable the algorithm to convert closer to equilibrium than lower values of exploration constant between 0 and 1. Despite testing different values of exploration constant and limit of iterations, the algorithm converged poorly to a Nash equilibrium in most cases. The strategies computed by IS-MCTS leaned heavily toward the best possible action in each information state, even with a high exploration constant, which was easily exploitable by the best response algorithm. That led to MCCFR producing significantly better results.

Game configuration	0	H	V	RN
W2_H2_P1_C0_L8	-98.91	-98.41	-99.18	-99.29
W2_H2_P1_C1_L8	0	0	0	0
W3_H3_P2_C0_L6	-51.25	-54.68	-63.99	-57.33
W3_H3_P2_C0_L8	-60.44	-65.23	-68.48	-68.29
W3_H3_P2_C1_L6	-19.39	-24.09	-24.89	-23.42
W3_H3_P2_C1_L8	-25.34	-27.29	-27.33	-29.25
W3_H4_P2_C0_L8	-29.05	-32.82	-36.37	-33.12
W3_H4_P2_C0_L10	-28.92	-29.89	-37.15	-32.72
W3_H4_P2_C1_L8	-4.62	-6.16	-5.08	-5.60
W4_H3_P2_C0_L10	-27.02	-34.71	-41.57	-34.23

**Table 6.3.** Pursuit Evasion game values for different value estimations of states beyond the horizon, with strategies computed using IS-MCTS. (Table shows estimated values of the first player = player with perfect information)

Table 6.3 shows the values of the Pursuit Evasion games for all game configurations and game variants with strategies computed with the IS-MCTS algorithm. The best results had the  $V$  game variants in most cases, but the success of the  $V$  variant is possible only because of the best convergence. As can be seen in Table 6.4, all of the strategies for the  $V$  game variants converged closest to the Nash equilibrium, making them more robust against exploitation by the best response of the evader player.

The 0 game variants performed the worst of all game variants in both game values and convergence. This shows that even with a poorly converging algorithm, providing non-zero state value estimations beyond the horizon yields better-performing strategies.

The  $H$  game variants and the  $RN$  game variants, in most cases, performed better than the 0 variants and worse than  $V$  variants, with  $RN$  game variants performing better in values of most game settings than  $H$  variants.  $RN$  variants also had better convergence in smaller game settings than  $H$  game variants, which had better convergence in bigger game settings.

Game configuration	0	H	V	RN
W2_H2_P1_C0_L8	5.32	6.80	0.59	0.14
W2_H2_P1_C1_L8	3.33	4.35	3.65	3.56
W3_H3_P2_C0_L6	76.56	47.01	14.08	36.86
W3_H3_P2_C0_L8	77.25	42.35	11.11	28.59
W3_H3_P2_C1_L6	66.43	42.21	16.83	56.86
W3_H3_P2_C1_L8	74.38	46.12	17.08	60.22
W3_H4_P2_C0_L8	64.75	38.39	16.18	46.25
W3_H4_P2_C0_L10	87.47	58.93	21.28	64.05
W3_H4_P2_C1_L8	72.62	47.64	20.78	69.05
W4_H3_P2_C0_L10	90.87	54.86	19.36	63.49

**Table 6.4.** Convergence of IS-MCTS algorithm for different value estimations of states beyond the horizon for Pursuit Evasion games. Values in the table are equal to exploitability in the respective bounded games. The closer an exploitability is to zero, the closer a strategy is to Nash equilibrium.

### 6.3 Search Game

Game configurations used for Search Game experiments are shown in the format  $Wx_1Hx_2Lx_3dp$ , where the capital letter represents a parameter,  $x_i$  represents a value of the parameter with  $i \in \{1, 2, 3\}$ , and  $dp$  represents positions of defender game pieces, it corresponds to the format of the parameter *defender\_init\_positions* with omitted ':' (the default starting positions of defender game pieces can be seen in Figure A.1).  $W$  represents *width*,  $H$  represents *height*, and finally  $L$  represents *max\_game\_lenght*, which is the length of the bounded game. The rest of the parameters are set to default values except *reward\_attacker\_loss*, which was set to 0 for all game configurations. The reason for this modification is to motivate the attacker to move into the zones instead of waiting in the start zone. More details regarding the configuration can be seen in Appendix A.

Value estimations of the states beyond the horizon were computed with value iterations with discount factor  $c = 0.9$  for all game variants. The maximum error for the value iteration was set to  $e = 1$ .

Table 6.5 shows values for different Search Game game configurations with strategies computed using MCCFR. The values of the games are positive because the table shows the values of the player 1, who is the attacker and who receives a reward of 100 when his game piece reaches the goal zone. This means that the defender (player 2) had a better strategy and was able to catch an attacker game piece more reliably in cases with lower game values.

The first row of the table represents a special case where the attacker has a 50% chance to win when both players play the optimal strategy; values for all game variants, in this case, are close to the correct value, which is 50. For the other rows, the *RN* game variants produced the best results, which is expected since the best response is computed on the same variant of the games, so the *RN* column can be viewed as a reference of the most optimal value.

The *V* game variants performed better than 0 and *H* game variants because the value estimations of the *V* states beyond the horizon are closest to the *RN* state estimations despite the *V* and *RN* variants benefiting the opposite players. The *V* state value estimations contain non-zero values either in states where the defender can no longer

stop the attacker from reaching the goal zone no matter what action he plays or in states in which reaching the goal zone depends on random probability; the dependency on random probability occurs only when the attacker entering the first zone, and the value estimations of such states will be slightly lower or equal to the  $RN$  estimations depending on the game configuration. The  $V$  value estimations of the states where the attacker cannot be stopped are equal to the value estimation in  $RN$  because if the attacker cannot be caught, the actions of the defender are irrelevant. The defender will try to avoid entering the information sets that have a high probability of being those states and, as a result, will produce better strategies.

The  $H$  game variants had the worst performance despite having non-zero value estimation in the same states as  $V$ . The reason might be that the ratio between the values of the states where the attacker won and the other non-zero states is very different from the  $RN$  variants, which makes the strategies incorrectly prioritize which information sets they should try to avoid.

Game configuration	0	H	V	RN
W1_H2_L6_00	50.44	50.46	50.13	50.56
W1_H3_L6_01_02	42.82	42.84	38.64	38.13
W2_H2_L6_00_10	22.49	20.25	20.25	20.59
W2_H3_L6_01_11	54.20	64.37	47.07	43.25
W2_H3_L8_01_11	54.47	59.94	47.87	43.31
W3_H3_L8_01_11_21	36.21	32.64	32.97	28.25
W3_H3_L8_01_20_22	36.63	40.01	40.17	29.48

**Table 6.5.** Search Game game values for different value estimations of states beyond the horizon, with strategies computed using MCCFR. (Table shows estimated values of the first player = player with perfect information)

The iteration limit for MCCFR was variable for different game configurations ranging from 40000 to 100000; the algorithm was also stopped when exploitability reached a value lower than 1. Exploitability was calculated after the first iteration and then after each 1000 iterations. Table 6.6 shows convergence and the number of run iterations to reach the convergence for every game configuration. The big impact on convergence was the configuration of games. The game variant impacted the convergence as well, with 0 and  $H$  variants converging better than  $V$  and  $RN$  variants.

Game configuration	0 - iter   expl	H - iter   expl	V - iter   expl	RN - iter   expl
W1_H2_L6_00	7001   0.81	5001   0.70	13001   0.89	8001   0.86
W1_H3_L6_01_02	34001   0.95	35001   0.74	61001   0.99	34001   0.99
W2_H2_L6_00_10	1001   0.39	1001   0.32	1001   0.80	13001   0.96
W2_H3_L6_01_11	13001   0.95	10001   0.77	26001   0.97	15001   0.96
W2_H3_L8_01_11	84001   0.92	71001   0.99	89001   0.98	86001   0.99
W3_H3_L8_01_11_21	3001   0.90	5001   0.88	6001   0.94	100000   3.68
W3_H3_L8_01_20_22	2001   0.75	3001   0.88	4001   0.87	40000   2.50

**Table 6.6.** Convergence of MCCFR algorithm for different value estimations of states beyond the horizon for Search Game games (iter = iterations, expl = exploitability in the respective bounded games). The closer an exploitability is to zero, the closer a strategy is to Nash equilibrium.

Computing strategies on the Search Game games using IS-MCTS was omitted because IS-MCTS implementation requires a way to resample states from the same information set as a given state according to a probability distribution over actions. This requirement is not trivial in Search Game because of the clue mechanic. When resampling a new state, all placed and found clues would have to correspond to the state from which resampling started and creating a heuristic for this functionality would not be optimal for iterative algorithm such as IS-MCTS.

## 6.4 Patrolling Game

Game configurations used for Patrolling Game experiments are shown in the format  $Gx_1_Ax_2_Lx_3$ , where the capital letter represents a parameter, and  $x_i$  represents a value of the parameter with  $i \in \{1, 2, 3\}$ .  $G$  represents the index of a used graph (graphs can be seen in Figure A.2),  $A$  represents *attack\_lenght*, and finally  $L$  represents *max\_game\_lenght*, which is the length of the bounded game. The rest of the parameters are set to default values. More details regarding the configuration can be seen in Appendix A.

The game implementation was also modified to give rewards equal to 0 for both players if an attack fails. The reason for this modification is to motivate the attacker to start an attack instead of waiting for a draw.

Value estimations of the states beyond the horizon were computed with value iterations with discount factor  $c = 0.9$  for all game variants. The maximum error for the value iteration was set to  $e = 1$ .

Table 6.7 shows values for different Patrolling Game game configurations with strategies computed using MCCFR. The values of the games are positive because the table shows the values of the player 1, who is the attacker and who receives a reward of 100 when he successfully completes an attack. This means that the defender (player 2) had a better strategy and was able to catch an attacker game piece more reliably in cases with lower game values.

Game configuration	0	H	V	RN
G1_A2_L8	63.12	63.39	56.70	56.70
G1_A3_L10	85.26	72.30	54.09	42.51
G2_A2_L8	67.46	72.88	60.71	57.96
G2_A3_L10	72.67	73.57	63.46	44.23
G3_A2_L8	67.76	67.50	60.80	57.85
G3_A3_L10	77.53	71.07	60.47	44.28
G4_A3_L8	89.99	89.99	70.72	59.93
G4_A4_L10	88.74	70.66	68.88	47.10
G5_A3_L8	63.75	65.13	64.93	54.44
G5_A4_L10	67.51	63.02	62.79	44.72
G6_A4_L10	87.52	88.67	68.08	63.12
G6_A5_L12	80.44	72.00	68.11	54.42

**Table 6.7.** Patrolling Game game values for different value estimations of states beyond the horizon, with strategies computed using MCCFR. (Table shows estimated values of the first player = player with perfect information)

When strategies were computed using MCCFR, the defender was most successful in *RN* game variants, which was expected because the best response of the attacker is

computed on the same games. So, the *RN* column is more of a reference to the most optimal values. The second-best results can be seen in *V* game variants. In *V*, value estimations of states are non-zero only in cases where the defender has no chance of disrupting an attack; all other states have the value of 0 because if the defender has information about the attacked node position, he will always get there if has enough time. This means that the strategy of the defender in the *V* game variants should try to minimize the chance of reaching the information sets that contain states with non-zero value estimations, or if it is impossible, the defender should try to reach the information sets with the lowest probability of being the state with high-value estimation. Such strategies performed well against the best response in *RN* variants of the games because all non-zero state value estimations in *V* are equal to the corresponding state value estimations in *RN*. Additionally, these states have the highest value because they guarantee a successful attack, no matter the defender's actions, so the rest of the non-zero states in *RN* that are zero in *V* would have a lower impact on the final strategy.

The *H* game variants performed slightly better than the 0 variants. In the 0 variants, only states where an attack finishes before the bound have a non-zero value, so the defender's strategies computed on such games will try to prevent these, completely ignoring other states, which will have higher values in *RN* variants and thus will be exploited by the best response of the attacker. The problem with *H* variants is that even though they consider the same states as *V*, the values in these states are lower, so the final strategies will put a higher priority on preventing the states where the attack ends before the bound, which will make them more vulnerable to exploitation by the best response of the attacker. There are cases where 0 variants outperformed *H* variants; the reasons might be that strategies in 0 variants got lucky and accidentally prevented undesirable states or that the *H* variants strategies ended up in between trying to prevent the states where the attack succeeded and the states with high-value estimations with ratio not corresponding to the *RN* variants or combination of both.

Game configuration	0 - iter   expl	H - iter   expl	V - iter   expl	RN - iter   expl
G1_A2_L8	30001   0.93	26001   0.78	24001   0.98	29001   0.96
G1_A3_L10	22001   0.68	29001   0.88	70000   1.73	70000   1.62
G2_A2_L8	59001   0.99	36001   0.96	57001   0.98	53001   0.97
G2_A3_L10	25001   0.92	37001   0.71	70000   1.33	70000   1.74
G3_A2_L8	23001   0.99	29001   0.90	38001   0.92	42001   0.99
G3_A3_L10	23001   0.68	14001   0.95	50001   0.88	70000   1.39
G4_A3_L8	12001   0.71	8001   0.92	14001   0.60	43001   0.93
G4_A4_L10	8001   0.82	7001   0.48	13001   0.84	70000   1.29
G5_A3_L8	6001   0.68	9001   0.13	4001   0.79	9001   0.94
G5_A4_L10	1001   0.24	1001   0.23	1001   0.54	13001   0.98
G6_A4_L10	10001   0.47	18001   0.89	47001   0.94	25001   0.89
G6_A5_L12	10001   0.96	10001   0.95	11001   0.61	34001   0.97

**Table 6.8.** Convergence of MCCFR algorithm for different value estimations of states beyond the horizon for Patrolling Game games (iter = iterations, expl = exploitability in the respective bounded games). The closer an exploitability is to zero, the closer a strategy is to Nash equilibrium.

MCCFR for all game configurations was limited to 70000 iterations, or it would end when it reached exploitability lower than 1. Exploitability was calculated after the

first iteration and then after each 1000 iterations. Table 6.8 shows convergence and the number of run iterations to reach the convergence for every game configuration. As can be seen in the table, the convergence of the algorithm was very dependent on the configuration of games, especially on the graphs. The algorithm, in the majority of cases, converged faster on 0 and  $H$  game variants compared to  $V$  and  $RN$  game variants.

For computing strategies for Patrolling Game games with IS-MCTS, the IS-MCTS was limited to a maximum of 10000 simulation per information state and the exploration constant  $c$  was set to 70 for every game configuration and variant. The higher exploration constant was chosen because the IS-MCTS algorithm heavily prioritizes a single best action in the produced strategies, and a higher exploration constant makes the algorithm produce more balanced strategies. This helps the strategies to be closer to the Nash equilibria in games where there is no one best optimal action in information states. But even with a high exploration constant, the produced strategies leaned towards the one best action and thus were exploited by the best response of the attacker player. Making the strategies computed with MCCFR much more reliable.

Game configuration	0	H	V	RN
G1_A2_L8	82.51	72.32	62.78	78.27
G1_A3_L10	99.27	99.23	99.04	99.19
G2_A2_L8	79.14	69.06	71.55	71.63
G2_A3_L10	76.23	72.23	75.06	80.27
G3_A2_L8	74.29	79.53	78.09	79.10
G3_A3_L10	72.57	69.56	62.77	70.30
G4_A3_L8	90.16	89.26	89.02	88.90
G4_A4_L10	99.13	89.49	89.50	89.50
G5_A3_L8	99.83	66.94	65.48	69.24
G5_A4_L10	69.55	66.46	66.69	62.79
G6_A4_L10	99.81	99.84	99.99	99.98
G6_A5_L12	78.95	81.06	89.54	99.34

**Table 6.9.** Patrolling Game game values for different value estimations of states beyond the horizon, with strategies computed using IS-MCTS. (Table shows estimated values of the first player = player with perfect information)

Table 6.9 shows the values of the Patrolling Game games for all game configurations and game variants with strategies computed with the IS-MCTS algorithm. The best results had the game variants where IS-MCTS converged closest to the Nash equilibrium because the produced strategies were harder to exploit by the best response of the attacker. As can be seen in Table 6.10, almost no strategies converged close to equilibria with an exception in the case of  $G5\_A4\_L10$  game configuration for the 0,  $H$  and  $V$  game variant, where the exploitability was close to 1. The reason is that the optimal strategy in the graph (see Figure A.2 graph 5) the game configuration uses is closer to picking one best action, which is ideal for the use of IS-MCTS. The strategy for the  $RN$  game variant nor the strategies for the  $G5\_A3\_L8$  configuration did not converge because the optimal strategies are no longer close to picking one action.

Using the game variants with non-zero state value estimations beyond the horizon is reasonable even with an algorithm that converges poorly to the Nash equilibria because algorithms computed on such game had slightly better results than the 0 game variants, as can be seen in Table 6.9.



Game configuration	0	H	V	RN
G1_A2_L8	35.41	27.58	29.14	31.89
G1_A3_L10	81.00	86.90	76.05	76.94
G2_A2_L8	32.54	21.04	27.31	25.40
G2_A3_L10	42.39	38.89	56.29	55.63
G3_A2_L8	26.38	32.85	34.14	31.66
G3_A3_L10	54.98	39.42	39.45	53.74
G4_A3_L8	41.35	25.95	44.41	41.62
G4_A4_L10	78.95	48.02	66.39	65.37
G5_A3_L8	50.65	8.82	17.99	23.34
G5_A4_L10	1.41	1.99	0.50	32.79
G6_A4_L10	52.54	53.18	52.68	53.42
G6_A5_L12	53.57	74.12	71.97	68.34

**Table 6.10.** Convergence of IS-MCTS algorithm for different value estimations of states beyond the horizon for Patrolling Game games. Values in the table are equal to exploitability in the respective bounded games. The closer an exploitability is to zero, the closer a strategy is to Nash equilibrium.

## Chapter 7

### Conclusion

A partially observable stochastic game model is a versatile tool that is useful for representing a wide variety of scenarios. We implemented three games based on a partially observable stochastic game model into the OpenSpiel framework. The implemented games are Pursuit Evasion, Search Game and Patrolling Game. During the usage of the games' implementations, all discovered errors were fixed; after that, all functionalities behaved as expected, and no additional errors were discovered. In conclusion, the games can be declared as functional. The games could potentially be better optimised performance-wise because the string representations of information states are unnecessarily long; however, the current implementation prioritizes human readability.

Then, we created four bounded variants for each of the implemented games:  $0$ ,  $V$ ,  $H$ , and  $RN$ . The  $0$  variant with state value estimations of states beyond the horizon equal to zeros. The  $V$  variant with state value estimations of states beyond the horizon equal to the values of states in the infinite perfect information game. The  $H$  variant with values from  $V$  divided by two. Lastly, the  $RN$  variant with state value estimations of states beyond the horizon equal to the values of states in the infinite perfect information game where the imperfect information player plays actions randomly.

We computed strategies for the imperfect information player on different configurations of all bounded game variants using MCCFR and IS-MCTS algorithms. We compared the quality of computed strategies by obtaining the value of the perfect information player playing the best response to the tested strategies in the  $RN$  variants of the games. Finally, we discussed the convergence of the used algorithms.

In the case of strategies computed with MCCFR, the best-performing strategies were the ones computed on  $RN$  game variants, which was expected since the best response was computed on the same variants. For Search Game and Patrolling game, the strategies computed on  $V$  game variants outperformed the strategies computed on  $H$  and  $0$  game variants. In Pursuit Evasion, the strategy computed on  $H$  game variants outperformed strategies computed on  $V$  and  $0$  game variants. The reason strategies computed on the  $V$  variants did not perform that well was the  $V$  variants' overestimation of values of states beyond the horizon.

In Pursuit Evasion, the strategies computed with the MCCFR algorithm clearly converged to the Nash equilibrium the best on the  $V$  game variant for all the game configurations. In the case of Search Game and Patrolling Game, there was no clear evidence of a variant converging better than the others, but in general,  $0$  and  $H$  game variants converge slightly better than the  $RN$  and  $V$  variants.

The strategies computed by IS-MCTS algorithm generally converged poorly to the Nash equilibria for all game configurations and variants because the strategies produced by IS-MCTS lean heavily toward playing the one most optimal action, but the used games required for optimal strategies to be more spread among the picked actions. This led to the best response of the perfect information player exploiting such strategies, thus making them perform significantly worse than strategies computed using MCCFR.

---

A possible future extension of this work could be to compare the result of this work with strategies and game values obtained by algorithms specialized in solving one-sided, partially observable stochastic games, for example, algorithm [9]. Since these algorithms are not part of OpenSpiel, there is a problem with the compatibility, meaning that these algorithms would have to be implemented into OpenSpiel or some interlayer would have to be introduced, which is beyond the scope of this work.



## References

- [1] ALPERN, Steve, Alec MORTON, and Katerina PAPADAKI. Patrolling Games. *Operations Research*. 10, 2011, Vol. 59. Available from DOI 10.2307/41316027.
- [2] BROWN, Noam, and Tuomas SANDHOLM. Reduced Space and Faster Convergence in Imperfect-Information Games via Regret-Based Pruning. 2016.
- [3] CHANG, Shu-Lin, Kun-Chang LEE, Ruey-Rong HUANG, and Yu-Hsien LIAO. Resource-Allocation Mechanism: Game-Theory Analysis. *Symmetry*. 05, 2021, Vol. 13, pp. 799. Available from DOI 10.3390/sym13050799.
- [4] COWLING, Peter I., Edward J. POWLEY, and Daniel WHITEHOUSE. Information Set Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*. 2012, Vol. 4, No. 2, pp. 120-143. Available from DOI 10.1109/TCIAIG.2012.2200894.
- [5] DIXIT, Avinash, and Barry NALEBUFF. *The Art of Strategy: A Game Theorist's Guide to Success in Business and Life*. 2010. Available from <https://api.semanticscholar.org/CorpusID:106854475>.
- [6] ELLSON, John, Emden R. GANSNER, and Eleftherios KOUTSOFIOS. Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools. 2003. Available from <https://graphviz.org/documentation/EGKNW03.pdf>.
- [7] FARINA, Gabriele, Christian KROER, and Tuomas SANDHOLM. Stochastic Regret Minimization in Extensive-Form Games. 2020.
- [8] GANZFRIED, Sam, and Max CHISWICK. Most Important Fundamental Rule of Poker Strategy. 2020.
- [9] HORÁK, Karel, Branislav BOŠANSKÝ, and Michal PĚCHOUČEK. Heuristic Search Value Iteration for One-Sided Partially Observable Stochastic Games. *Proceedings of the AAAI Conference on Artificial Intelligence*. Feb., 2017, Vol. 31, No. 1. Available from DOI 10.1609/aaai.v31i1.10597. Available from <https://ojs.aaai.org/index.php/AAAI/article/view/10597>.
- [10] LANCTOT, Marc, Vinicius ZAMBALDI, Audrunas GRUSLYS, Angeliki LAZARIDOU, Karl TUYLES, Julien PEROLAT, David SILVER, and Thore GRAEPEL. A Unified Game-Theoretic Approach to Multiagent Reinforcement Learning. 2017.
- [11] LANCTOT, Marc, Kevin WAUGH, Martin ZINKEVICH, and Michael BOWLING. Monte Carlo Sampling for Regret Minimization in Extensive Games. In: Y. BENGIO, D. SCHUURMANS, J. LAFFERTY, C. WILLIAMS, and A. CULOTTA, eds. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2009. Available from [https://proceedings.neurips.cc/paper\\_files/paper/2009/file/00411460f7c92d2124a67ea0f4cb5f85-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2009/file/00411460f7c92d2124a67ea0f4cb5f85-Paper.pdf).
- [12] LANCTOT, Marc, Edward LOCKHART, Jean-Baptiste LESPIAU, Vinicius ZAMBALDI, Satyaki UPADHYAY, Julien PEROLAT, Sriram SRINIVASAN, Finbarr TIMBERS, Karl TUYLES, Shayegan OMIDSHAFIEI, Daniel HENNES, Dustin MORRILL, Paul MULLER, Timo EWALDS, Ryan FAULKNER, Janos KRAMAR,

- Bart De VYLDER, Brennan SAETA, James BRADBURY, David DING, Sebastian BERGEAUD, Matthew LAI, Julian SCHRITTWIESER, Thomas ANTHONY, Edward HUGHES, Ivo DANIHELKA, and Jonah RYAN-DAVIS. OpenSpiel: A Framework for Reinforcement Learning in Games. *CoRR*. 2019, Vol. abs/1908.09453. Available from <http://arxiv.org/abs/1908.09453>.
- [13] LESLIE, David, Steven PERKINS, and Zibo XU. Best-response Dynamics in Zero-sum Stochastic Games. *Journal of Economic Theory*. 07, 2020, Vol. 189, pp. 105095. Available from DOI 10.1016/j.jet.2020.105095.
- [14] LISÝ, Viliam, Vojtěch KOVAŘÍK, Marc LANCTOT, and Branislav BOŠANSKÝ. Convergence of Monte Carlo Tree Search in Simultaneous Move Games. 2013.
- [15] NISAN, Noam, Michael SCHAPIRA, Gregory VALIANT, and Aviv ZOHAR. Best Response Mechanisms. In: *ICS-11: Proceedings of the Conference on Innovations in Computer Science 2011*. ICS-11: Proceedings of the Conference on Innovations in Computer Science 2011ed. Tsinghua University Press, 2011. Available from <https://www.microsoft.com/en-us/research/publication/best-response-mechanisms/>. Presented at INFORMS '07.
- [16] OSBORNE, Martin J.. *An introduction to game theory*. New York: Oxford University Press, c2004. ISBN 0195128958.
- [17] PEROLAT, Julien, Bart DE VYLDER, Daniel HENNES, Eugene TARASSOV, Florian STRUB, Vincent de BOER, Paul MULLER, Jerome T. CONNOR, Neil BURCH, Thomas ANTHONY, Stephen MCALEER, Romuald ELIE, Sarah H. CEN, Zhe WANG, Audrunas GRUSLYS, Aleksandra MALYSHEVA, Mina KHAN, Sherjil OZAIR, Finbarr TIMBERS, Toby POHLEN, Tom ECCLES, Mark ROWLAND, Marc LANCTOT, Jean-Baptiste LESPIAU, Bilal PIOT, Shayegan OMIDSHAFIEI, Edward LOCKHART, Laurent SIFRE, Nathalie BEAUGUERLANGE, Remi MUNOS, David SILVER, Satinder SINGH, Demis HASSABIS, and Karl TUYLS. Mastering the game of Stratego with model-free multiagent reinforcement learning. *Science*. American Association for the Advancement of Science (AAAS), dec, 2022, Vol. 378, No. 6623, pp. 990–996. ISSN 1095-9203. Available from DOI 10.1126/science.add4679. Available from <http://dx.doi.org/10.1126/science.add4679>.
- [18] REEVES, Daniel, and Michael P. WELLMAN. Computing Best-Response Strategies in Infinite Games of Incomplete Information. 2012.
- [19] RENAULT, Jérôme. *A tutorial on Zero-sum Stochastic Games*. 2019.
- [20] SCHMID, Martin, Neil BURCH, Marc LANCTOT, Matej MORAVCIK, Rudolf KADLEC, and Michael BOWLING. Variance Reduction in Monte Carlo Counterfactual Regret Minimization (VR-MCCFR) for Extensive Form Games using Baselines. 2018.
- [21] SHOHAM, Yoav, and Kevin LEYTON-BROWN. *Multiagent systems: algorithmic, game-theoretic, and logical foundations*. Cambridge: Cambridge University Press, 2009. ISBN 978-0-521-89943-7.
- [22] WANG, Zifan, Yi SHEN, Michael M. ZAVLANOS, and Karl H. JOHANSSON. Convergence Analysis of the Best Response Algorithm for Time-Varying Games. 2023.
- [23] ZHANG, Li, Wei WANG, Shijian LI, and Gang PAN. Monte Carlo Neural Fictitious Self-Play: Approach to Approximate Nash equilibrium of Imperfect-Information Games. 2019.

- [24] ZINKEVICH, Martin, Michael JOHANSON, Michael BOWLING, and Carmelo PICCIONE. Regret Minimization in Games with Incomplete Information. In: J. PLATT, D. KOLLER, Y. SINGER, and S. ROWEIS, eds. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2007. Available from [https://proceedings.neurips.cc/paper\\_files/paper/2007/file/08d98638c6fcd194a4b1e6992063e944-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2007/file/08d98638c6fcd194a4b1e6992063e944-Paper.pdf).

# Appendix A

## Domain configurations for experiments

This appendix expands on the game configuration abbreviation used in experiments.

List of abbreviations used in the Table A.1:

- n-p = *num\_pursuers*
- a-c-w = *agent\_can\_wait*
- m-g-l = *max\_game\_length*

Game configuration	width	height	n-p	a-c-w	m-g-l
W2_H2_P1_C0_L8	2	2	1	false	8
W2_H2_P1_C1_L8	2	2	1	true	8
W3_H3_P2_C0_L6	3	3	2	false	6
W3_H3_P2_C0_L8	3	3	2	false	8
W3_H3_P2_C1_L6	3	3	-2	true	6
W3_H3_P2_C1_L8	3	3	2	true	8
W3_H4_P2_C0_L8	3	4	2	false	8
W3_H4_P2_C0_L10	3	4	2	false	10
W3_H4_P2_C1_L8	3	4	2	true	8
W4_H3_P2_C0_L10	4	3	2	false	10
W4_H4_P2_C0_L10	4	4	2	false	10

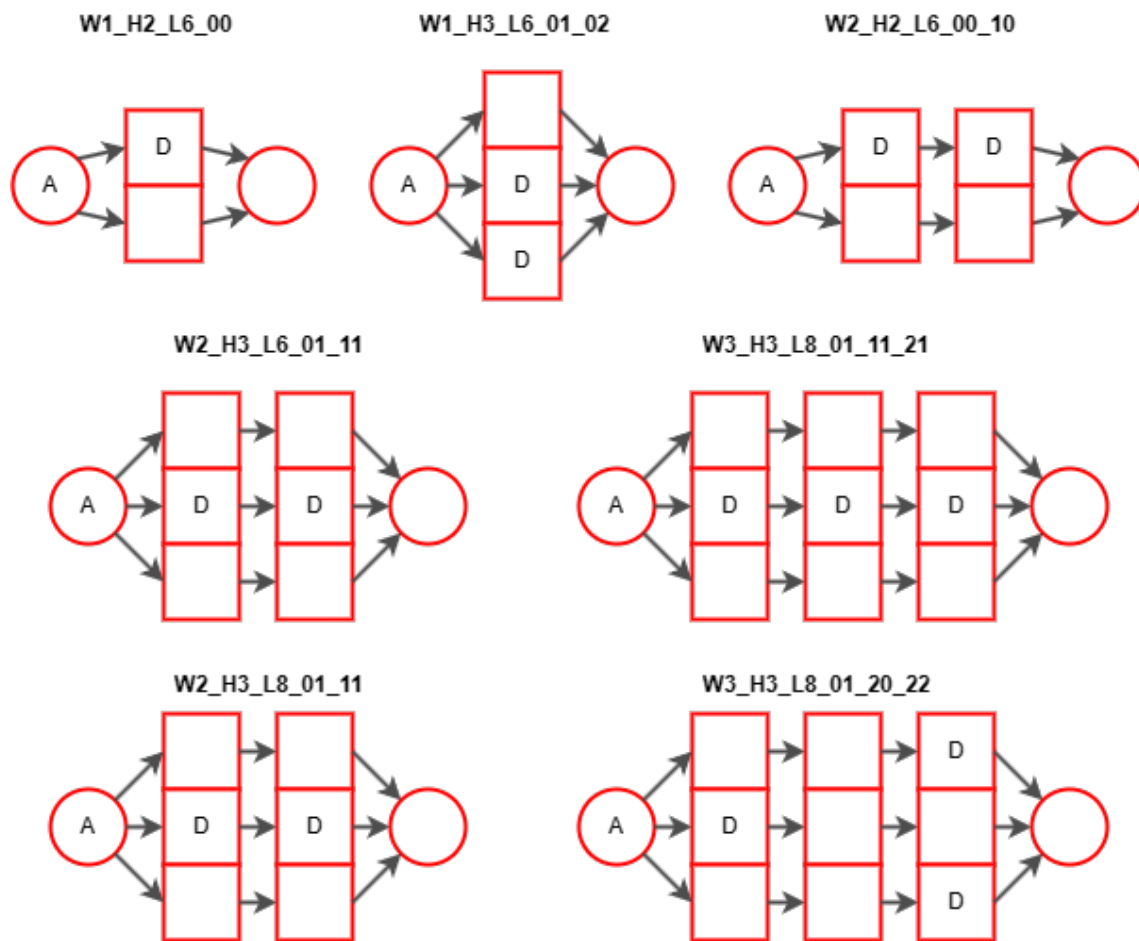
**Table A.1.** Configuration of Pursuit Evasion games used in experiments.

List of abbreviations used in the Table A.2:

- m-g-l = *max\_game\_length*
- d-i-p = *defender\_init\_position*

Game configuration	width	height	m-g-l	d-i-p
W1_H2_L6_00	1	2	6	00
W1_H3_L6_01_02	1	3	6	01_02
W2_H2_L6_00_10	2	2	6	00_10
W2_H3_L6_01_11	2	3	6	01_11
W2_H3_L8_01_11	2	3	8	01_11
W3_H3_L8_01_11_21	3	3	8	01_11_21
W3_H3_L8_01_20_22	3	3	8	01_20_22

**Table A.2.** Configuration of Search Game games used in experiments.

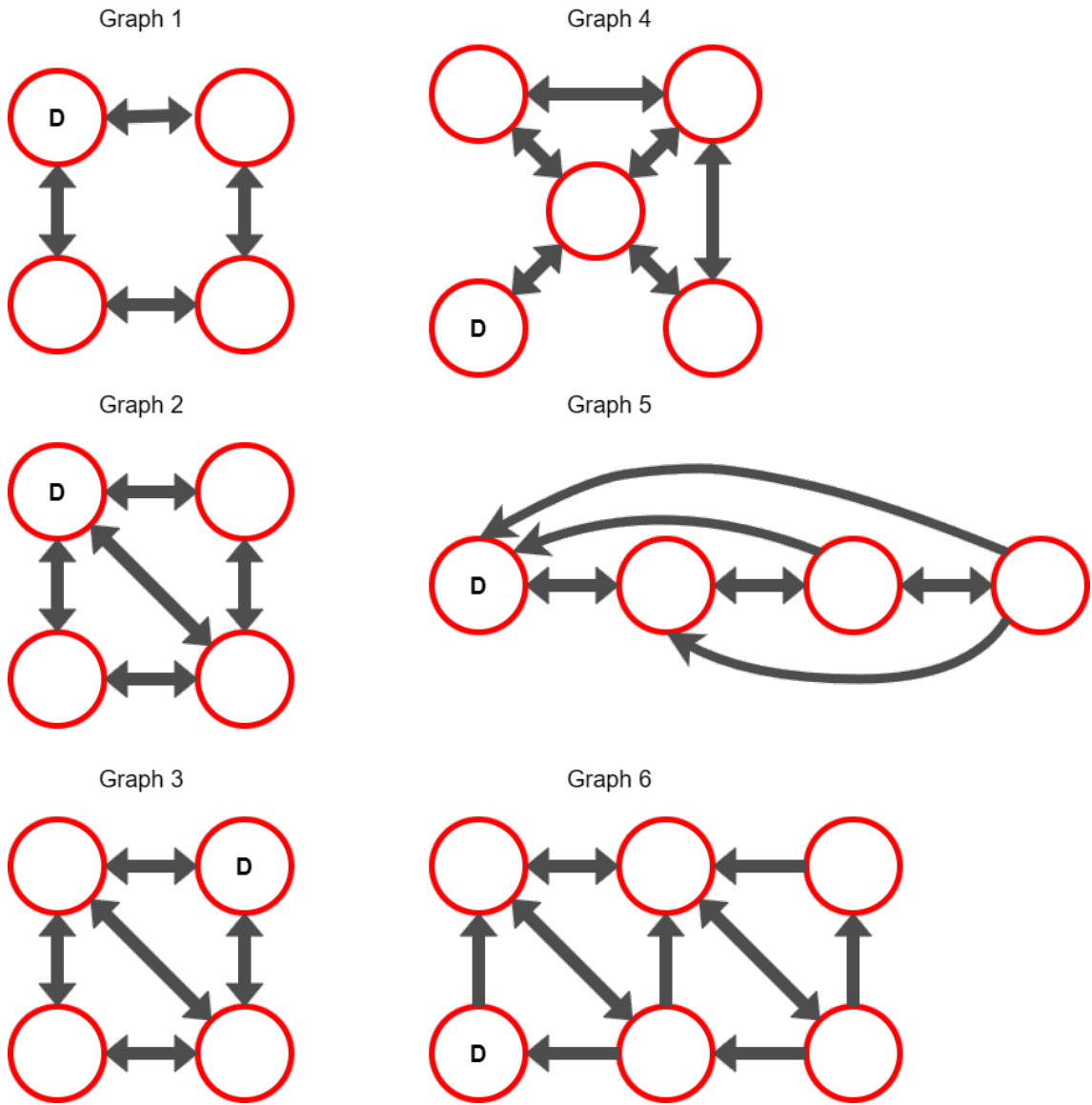


**Figure A.1.** Search Game games' initial states used in experiments.

Game configuration	graph	attack-length	max-game-length
G1_A2_L8	1	2	8
G1_A3_L10	1	3	10
G2_A2_L8	2	2	8
G2_A3_L10	2	3	10
G3_A2_L8	3	2	8
G3_A3_L10	3	3	10
G4_A3_L8	4	3	8
G4_A4_L10	4	4	10
G5_A3_L8	5	3	8
G5_A4_L10	5	4	10
G6_A4_L10	6	4	10
G6_A5_L12	6	5	12

**Table A.3.** Configuration of Patrolling Game games used in experiments.





**Figure A.2.** List of Patrolling Game graphs used in experiments with assigned numbers from 1 to 6.

## Appendix B

### Code structure in OpenSpiel

Figure B.3 shows the newly added files in the *open\_spiel-master* folder. The folder *bound\_games\_experiments\_data* was included in *open\_spiel-master* because it contains code for generating the experimental data that is dependent on the OpenSpiel. The folders *patrolling\_game*, *pursuit\_evasion*, and *search\_game* contain the implementations of the games. The file *value\_iteration\_simultaneous\_as\_sequential.py* contains the variation of value iteration for simultaneous games that takes a sequential game as input. A detailed structure of the *bound\_games\_experiments\_data* folder can be seen in Figure B.4.

```
open_spiel-master
├── bound_games_experiments_data
│   └── ...
└── open_spiel
    ├── games
    │   ├── patrolling_game
    │   │   ├── patrolling_game_graph.txt
    │   │   ├── patrolling_game_test.cc
    │   │   ├── patrolling_game.cc
    │   │   └── patrolling_game.h
    │   ├── pursuit_evasion
    │   │   ├── pursuit_evasion_test.cc
    │   │   ├── pursuit_evasion.cc
    │   │   └── pursuit_evasion.h
    │   ├── search_game
    │   │   ├── search_game_test.cc
    │   │   ├── search_game.cc
    │   │   └── search_game.h
    └── python
        └── algorithms
            └── value_iteration_simultaneous_as_sequential.py
```

**Figure B.3.** Source code directory tree

The *bound\_games\_experiments\_data* folder contains one subdirectory for each game. Each subdirectory contains similar files and folders except for the

*patrolling\_game* folder, which has in *input\_data* folder folder *graphs*, that contains the definitions of used graphs. Folders *input\_data* contain text files with state value estimations for different variants of games. Folders *ismcts\_output* contain text files with results with IS-MCTS as a used algorithm; this folder does not exist for Search Game. Folders *mccfr\_output* contain text files with results using MCCFR as a used algorithm. Files *generate\_input.py* generate files with state value estimations in *input\_data* folder. Files *generate\_ismcts\_output.py* and *generate\_mccfr\_output.py* generates files with results in folders *ismcts\_output* and *mccfr\_output* using IS-MCTS and MCCFR respectively.

```

bound_games_experiments_data
├── patrolling_game
│   ├── input_data
│   │   ├── graphs
│   │   │   └── *.txt (input graphs)
│   │   └── *.txt (input data)
│   ├── ismcts_output
│   │   └── *.txt (ismcts output data)
│   ├── mccfr_output
│   │   └── *.txt (mccfr output data)
│   ├── generate_input.py
│   ├── generate_ismcts_output.py
│   └── generate_mccfr_output.py
├── pursuit_evasion
│   ├── input_data
│   │   └── *.txt (input data)
│   ├── ismcts_output
│   │   └── *.txt (ismcts output data)
│   ├── mccfr_output
│   │   └── *.txt (mccfr output data)
│   ├── generate_input.py
│   ├── generate_ismcts_output.py
│   └── generate_mccfr_output.py
└── search_game
    ├── input_data
    │   └── *.txt (input data)
    ├── ismcts_output
    │   └── *.txt (ismcts output data)
    ├── mccfr_output
    │   └── *.txt (mccfr output data)
    ├── generate_input.py
    ├── generate_ismcts_output.py
    └── generate_mccfr_output.py

```

**Figure B.4.** Experiments data directory tree