Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Using Fast Upper-Bound Approximation in Heuristic Search Value Iteration

Jakub Brož

Supervisor: doc. Mgr. Branislav Bošanský, Ph.D.
Field of study: Open Informatics
Subfield: Artificial Intelligence and Computer Science
May 2021

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Brož Jakub**	Personal ID number:	**483529**
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:

Using Fast Upper-Bound Approximation in Heuristic Search Value Iteration

Bachelor’s thesis title in Czech:

Využití rychlé aproximace funkcí v heuristickém algoritmu iterace hodnot

Guidelines:

One-Sided Partially Observable Stochastic Games are dynamic games with infinite horizon where only one player has imperfect information and the opponent has full information. Despite the numerous possible application domains (e.g., in security), the applicability of the existing algorithm PG-HSVI [1] is limited due to insufficient scalability. Computing and querying the upper-bound of the value function is one of the key bottlenecks of the algorithm. This upper-bound value function is represented as a lower convex envelope of a set of points in a high-dimensional space. The goal of the student is to:
1. Get familiar with the algorithm PG-HSVI.
2. Identify and implement an appropriate method for fast approximation of the upper-bound value function (e.g., using neural networks).
3. Modify the exploration phase of the PG-HSVI algorithm to use approximate upper-bound value function (e.g., by computing bounded-rational game-theoretic strategies).
4. Experimentally compare the convergence of proposed modifications with the original algorithm.

Bibliography / sources:

[1] Horák, K., Bošanský, B., & Pěchouček, M. (2017). Heuristic Search Value Iteration for One-Sided Partially Observable Stochastic Games. In AAAI (pp. 558-564).
[2] McKelvey, Richard; Palfrey, Thomas (1995). "Quantal Response Equilibria for Normal Form Games". Games and Economic Behavior. 10: 6–38.

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: **09.01.2021** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **30.09.2022**

doc. Mgr. Branislav Bošanský, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	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

Acknowledgements

I would like to thank my supervisor, doc. Mgr. Branislav Bošanský, Ph.D., for his guidance, invaluable advice, welcoming approach and a good dose of patience. I would also like to thank him for helping me choose a project which I enjoyed working on, I had an interest in and allowed me to grow in a lot of areas.

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

Declaration

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 21, 2021

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 21. května 2021

Abstract

Many security problems, such as pursuit-evasion games or patrolling games, can be modeled as one-sided partially observable stochastic games (OS-POSGs). In these problems, two sides compete against each other, although only one side has full information about the current state. Due to the similarities with partially observable Markov decision processes (POMDPs), a modification of the heuristic search value iteration (HSVI) algorithm can be used to solve these kinds of problems. However, the HSVI for OS-POSGs unfortunately does not scale well and therefore is not applicable to larger problems. The main bottleneck of the HSVI for OS-POSGs algorithm is the frequent usage of linear programs that slow down the computation. We provide a solution to this performance issue in the form of two modifications. The first modification replaces the computation of the upper bound value function by a method that utilizes neural networks to approximate the true value of the upper bound. Our second contribution is the replacement of the exact Nash equilibrium stage game solution with a bounded-rational quantal response equilibrium (QRE). The usage of QRE allows the incorporation of the approximative upper bound method into the original algorithm.

Keywords: game theory, partially observable stochastic games, one-sided information, heuristic search value iteration, neural networks, quantal response equilibrium, approximation

Supervisor: doc. Mgr. Branislav Bošanský, Ph.D.
Praha 2, Karlovo náměstí 13, E-407

Abstrakt

Řada bezpečnostních problémů, jako jsou pronásledovací hry nebo hlídkovací hry, lze modelovat jako jednostranné, částečně pozorovatelné stochastické hry (OS-POSG). V těchto problémech proti sobě soupeří dvě strany, ale pouze jedna ze stran má úplnou informaci o současném stavu. Vzhledem k podobnostem s částečně pozorovatelnými Markovovými rozhodovacími procesy (POMDP) lze k řešení těchto problémů použít modifikaci heuristického algoritmu iterace hodnot (HSVI). HSVI pro OS-POSG bohužel špatně škáluje, a proto není použitelný na větší problémy. Hlavní překážkou algoritmu HSVI pro OS-POSGs je časté využívání lineárních programů, které zpomaluje výpočet. Poskytujeme řešení tohoto problému s rychlostí ve formě dvou modifikací. První modifikace nahrazuje výpočet horní meze hodnotové funkce metodou, která využívá neuronové sítě k aproximaci skutečné hodnoty horní meze. Naším druhým příspěvkem je nahrazení přesného řešení Nashovy rovnováhy v jednotlivých fázích pomocí omezeně racionální rovnováhy kvantových odpovědí (QRE). Použití QRE umožňuje zakomponovat přibližnou metodu pro horní mez do původního algoritmu.

Klíčová slova: teorie her, částečně pozorovatelné stochastické hry, jednostranná informace, heuristický algoritmus iterace hodnot, neuronové sítě, rovnováha kvantových odpovědí, aproximace

Překlad názvu: Využití rychlé aproximace funkcí v heuristickém algoritmu iterace hodnot

Contents

Figures

Tables

Chapter 1
Introduction

Games, in general, are often used to model real-life security problems. They can have the form of defender-attacker games, where the defending side protects some critical targets against attackers, or the pursuit-evasion form, where the defenders are trying to capture the evading target. The real-life problems modeled can be, for example, the allocation of limited security resources (such as patrolling units) to protect vulnerable targets, protecting computer networks against cyberattacks or protecting wildlife against poachers [SFA$^+$18, Tam09].

The one-sided partially observable stochastic games (OS-POSGs) are non-deterministic dynamic games with infinite horizon, in which only one side has full information about the current state. Because in most security problems, the information the defending side has available is not complete, the OS-POSGs can be used to model them. In particular, the OS-POSGs can be used, for example, to solve problems such as the strategic allocation of honeypots in computer networks [HBT$^+$19] or protecting critical infrastructures against sequential attacks [TBN20].

An existing heuristic search value iteration (HSVI) algorithm [HBP17, HBKK20] can be used to solve OS-POSGs games. It is an extension of the HSVI algorithm for solving partially observable Markov decision processes (POMDPs) [SS12b]. However, the existing HSVI algorithm does not scale sufficiently, which makes it inapplicable to larger games.

One of the key bottlenecks of the HSVI algorithm is the querying of the upper bound value function. The upper bound value function is represented as the lower convex hull of a set of points in high-dimensional space. Obtaining the value of the upper bound at an arbitrary point requires solving a linear program.

Our goal is to find a suitable approximative method that could replace the computationally heavy querying of the upper bound value function and incorporate it into the existing algorithm. This also requires replacing the exact stage game solution computation with an approximate method compatible with the modified upper bound value function.

In Chapter 2, we define some basic concepts from game theory that the HSVI algorithm utilizes, and we also introduce the quantal response equilibrium [MP96], which will be used as the stage game solution concept in our modifications. In Chapter 3, we first introduce Markov decision processes (MDPs) and their partially observable extensions and then describe the version of HSVI algorithm for solving POMDPs. In the following Chapter 4, we define the OS-POSGs and describe the modifications that make the usage of HSVI for OS-POSGs possible. In Chapter 5, we introduce our two main contributions: approximate computation of the upper bound value function using neural networks and approximate stage game solution using quantal response equilibrium. In Chapter 6, we evaluate our modifications and compare them with the original algorithm.

Chapter 2

Game Theory

In this chapter, we will introduce the most fundamental representation of games in game theory and some concepts to help us reason about games. We will also describe two solution concepts that can be used to solve games.

Games, in general, can be described as a multiagent environment where each of the agents acts to maximize his expected utility. When choosing an action in a multiagent environment, one needs to consider the actions of other agents. The other agents also act in their best self-interest and, by doing so, affect other agents by changing their utility gain.

2.1 Normal-form games

The most fundamental representation of games in game theory is the normal form. This representation is very straightforward. All agents in the environment choose one action and play it simultaneously. For every possible outcome of the game, utility rewards are given. We will further use the terms agent and player interchangeably.

Definition 2.1 (Normal-form game [SLB08])**.** A (finite, n-person) *normal-form game* is a tuple (N, A, u), where:

- N is a finite set of n *players*, indexed by i;
- $A = A_1 \times \cdots \times A_n$, where A_i is a finite set of *actions* available to player i. Each vector $a = (a_1, \ldots, a_n) \in A$ is called an *action profile*;
- $u = (u_1, \ldots, u_n)$ where $u_i : A \mapsto \mathbb{R}$ is a real-valued *utility* (or *payoff*) *function* for player i.

The normal-form games are usually represented as an n-dimensional payoff matrix. The available actions of each player are listed in the corresponding axis of the matrix. The matrix cells represent the possible outcomes with each player's utility given in the order of players.

The famous *Prisoner's dilemma* is an example of a normal-form game. In this game, both players have two available actions - cooperate and defect. It can be represented as a 2×2 payoff matrix, as shown in Figure 2.1.

$$\begin{array}{c|c|c|} & & \text{Player 2} & \\ & & \text{C} & \text{D} \\ \hline \text{Player 1} & \text{C} & -1,-1 & -3,0 \\ \hline & \text{D} & 0,-3 & -2,-2 \\ \hline \end{array}$$

Figure 2.1: Payoff matrix of the normal-form game Prisoner's dilemma

We will be focusing on noncooperative normal-form games. In noncooperative branch of game theory, the agents compete independently against each other in a competition. The goal of an agent is to maximize his own utility while taking into account that all other agents are doing exactly the same.

2.1.1 Zero-sum games

We will focus specifically on zero-sum games. These are two-player games, in which for every possible outcome, the sum of utilities of the two players is equal to zero:

$$u_1(a_1, a_2) + u_2(a_1, a_2) = 0 \qquad \forall (a_1, a_2) \in A_1 \times A_2 \,. \tag{2.1}$$

This can be expressed also as

$$u_2(a_1, a_2) = -u_1(a_1, a_2) \qquad \forall (a_1, a_2) \in A_1 \times A_2 \,. \tag{2.2}$$

As a result, the goal of maximizing own utility is actually exactly the same goal as minimizing the utility of the opponent. This also significantly simplifies the representation of the game because the outcome rewards have to be specified only for one player. Rewards of the second player can be computed simply by negating the rewards of the first one.

For the payoff matrix representation of normal-form games, this means that the sum in each outcome cell is equal to zero. An example of such a game is a simple game of *Matching pennies*, shown in Figure 2.2. In this game, both players choose one side of a coin, either heads or tails, and reveal their choice simultaneously. If the top side of the coins matches, the first player wins; otherwise, the second player wins.

$$\begin{array}{c|c|c|} & & \text{Player 2} & \\ & & \text{H} & \text{T} \\ \hline \text{Player 1} & \text{H} & 1,-1 & -1,1 \\ \hline & \text{T} & -1,1 & 1,-1 \\ \hline \end{array}$$

Figure 2.2: Payoff matrix of the zero-sum game Matching pennies

2.1.2 Strategies

Players in normal-form games follow strategies. The simplest strategy a player can play is a single action. This is called a pure strategy. If we choose a single action for each player, we get a pure-strategy profile.

However, these are not the only strategies the player has available. He can choose randomly from a set of his available actions according to a probability distribution. This strategy combined from multiple actions is called a mixed strategy, and we denote it s_i. If we choose a mixed strategy for each player, we get a mixed-strategy profile, denoted $s = (s_i, \ldots, s_n)$.

Definition 2.2 (Mixed strategy [SLB08])**.** Let (N, A, u) be a normal-form game, and for any set X, let $\Pi(X)$ be the set of all probability distributions over X. Then the set of *mixed strategies* for player i is $S_i = \Pi(A_i)$.

For a given mixed-strategy $s_i \in S_i$, $s_i(a_i)$ denotes the probability that player i plays an action a_i under mixed-strategy s_i. The elements of S_i are probability distributions. Therefore, it must hold that $\forall a_i \in A_i : s_i(a_i) \geq 0$ and $\sum_{a_i \in A_i} s_i(a_i) = 1$.

The set of mixed-strategy profiles is a Cartesian product of individual mixed-strategy sets $S = S_i \times \cdots \times S_n$.

We extend the domain of the utility function u from the set of action profiles A to the set of mixed-strategy profiles S. For a given mixed-strategy profile s, the expected utility of player i is

$$u_i(s) = \sum_{a \in A} u_i(a) \prod_{j=1}^{n} s_j(a_j)\,. \tag{2.3}$$

2.1.3 Nash equilibrium

With the terms strategy and strategy profile defined, we can now introduce one of the most important solution concepts in game theory - Nash equilibrium. For this, we will need to know what is a player's best response to a strategy profile.

If we leave other agents' actions fixed and only focus on one player i, we can define his best response to a strategy profile. Assume we have a strategy profile s_{-i} with all actions except for the one player i defined. Now the strategy chosen from his set of mixed strategies that would give him the greatest utility reward is the best response to the strategy profile s_{-i}.

Definition 2.3 (Best response [SLB08])**.** Player i's *best response* to the strategy profile s_{-i} is a mixed strategy $s_i^* \in S_i$ such that $u_i(s_i^*, s_{-i}) \geq u_i(s_i, s_{-i})$ for all strategies $s_i \in S_i$.

Now we can use the notion of best response to define Nash equilibrium, a mixed strategy profile in which every player plays his best response. Assume we have a mixed-strategy profile s. If for every player i, his strategy in this profile is his best response to the same profile with his strategy left out s_{-i}, then this mixed-strategy profile s is a Nash equilibrium.

Definition 2.4 (Nash equilibrium [SLB08])**.** A strategy profile $s = (s_1, \ldots, s_n)$ is a *Nash equilibrium* if, for all agents i, s_i is a best response to s_{-i}.

In Nash equilibrium, no player would change his strategy because he cannot gain any utility by doing so. He maximizes his utility by playing the best response. Because of this, Nash equilibria are stable strategy profiles and can be considered a solution to a noncooperative game.

For the game of Prisoner's dilemma introduced in Figure 2.1, there exists only one Nash equilibrium. It occurs when both players choose to defect - a pure strategy.

On the other hand, the game of Matching pennies from Figure 2.2 does not have a pure strategy Nash equilibrium. A mixed strategy profile, in which both players play heads or tails with the probability of 50 %, is the unique Nash equilibrium of this game.

Finding such equilibria is our goal. Luckily, the Nash theorem tells us that there exists at least one Nash equilibrium for every game.

Theorem 2.5 (Nash theorem [Nas51])**.** *Every game with a finite number of players and action profiles has at least one Nash equilibrium.*

2.2 Minimax

Finding Nash equilibria of two-player, general-sum games is a very hard problem - it belongs to a PPAD-complete (Polynomial Parity Arguments on Directed graphs) complexity class [SLB08]. We are, however, interested only in two-player, zero-sum games. These can be solved using maximin and minimax strategies.

2.2.1 Maximin and minimax strategies

For two-player games, we can define a strategy for player 1 that maximizes his worst-case payoff. He assumes the opponent will play the action that causes him the most harm and plays the action that maximizes his utility, given this assumption.

Definition 2.6 (Maximin strategy, two-player [SLB08])**.** In a two-player game, the *maximin strategy* for player 1 against player 2 is

$$\arg\max_{s_1 \in S_1} \min_{s_2 \in S_2} u_1(s_1, s_2)\,,$$

and player 1's *maximin value* is

$$\max_{s_1 \in S_1} \min_{s_2 \in S_2} u_1(s_1, s_2)\,.$$

Analogously, we can define a strategy that minimizes the best outcome of the opponent:

Definition 2.7 (Minimax strategy, two-player [SLB08])**.** In a two-player game, the *minimax strategy* for player 1 against player 2 is

$$\arg\min_{s_1 \in S_1} \max_{s_2 \in S_2} u_2(s_1, s_2)\,,$$

and player 2's *minimax value* is

$$\min_{s_1 \in S_1} \max_{s_2 \in S_2} u_2(s_1, s_2)\,.$$

We also defined the maximin and minimax value of a player. The player's maximin and minimax values in two-player games are actually the same:

$$\max_{s_1 \in S_1} \min_{s_2 \in S_2} u_1(s_1, s_2) = \min_{s_2 \in S_2} \max_{s_1 \in S_1} u_1(s_1, s_2)\,. \tag{2.4}$$

2.2.2 Minimax theorem

Because of the property of two-player, zero-sum games described in Equation (2.2), the maximin and minimax values of both players in these types of games actually coincide with Nash equilibrium:

Theorem 2.8 (Minimax theorem [vN28]). *In any finite, two-player, zero-sum game, in any Nash equilibrium each player receives a payoff that is equal to both his maximin value and his minimax value.*

The Nash equilibrium value of two-player, zero-sum, normal-form game, corresponding to maximin and minimax strategies, can be computed using linear program:

$$\begin{aligned}
\max_{V_1, s_1} \quad & V_1 && && \text{(2.5a)}\\
\text{s.t.} \quad & V_1 \le \sum_{a_1 \in A_1} s_1(a_1) u(a_1, a_2) && \forall a_2 \in A_2 && \text{(2.5b)}\\
& \sum_{a_1 \in A_1} s_1(a_1) = 1 && && \text{(2.5c)}\\
& s_1(a_1) \ge 0 && \forall a_1 \in A_1 && \text{(2.5d)}
\end{aligned}$$

This will prove useful for us when solving stage games in the original HSVI for OS-POGSs algorithm.

2.3 Quantal response equilibrium

Nash equilibrium is an exact solution concept. However, finding the exact solution to a game might not be necessary for some situations. Sometimes, an approximative method that is less computationally demanding and offers a reasonable solution is sufficient. Therefore, we will introduce an approximative method for solving normal-form games called quantal response equilibrium (QRE) [MP96].

QRE is a solution concept based on statistical models for quantal choice. The players choose their strategies based on an estimated expected utility. Because of this, better responses have a higher probability of being played than the worse ones. However, the best responses are not certain as with Nash equilibrium.

The method does not approximate Nash equilibrium. Instead, it finds an equilibrium with bounded rationality - meaning the equilibrium is not perfectly rational. The equilibrium is based on estimations and therefore is noisy or imperfect.

The method of finding QRE is an iterative process. We start with some default estimation of strategies and expected utilities and refine them during each iteration. We do this until we converge to a fixed point - a quantal response equilibrium.

2.3.1 Expected utility and error

First, we define some new notations. We will use J_i to denote $|A_i|$, the cardinality of a set of available actions for player i. $X_i = \mathbb{R}^{J_i}$ will denote the space of possible payoffs for the actions player i might adopt and $X = \prod_{i=1}^{n} X_i$.

We define the function $\overline{u} : S \mapsto X$ as

$$\overline{u}(s) = (\overline{u}_1(s), \ldots, \overline{u}_n(s)) \,, \tag{2.6}$$

where

$$\overline{u}_{ij}(s) = u_i(a_{ij}, s_{-i}) \tag{2.7}$$

is the expected utility of player i playing action $a_{ij} \in A_i$ against the opponents' mixed strategies from s.

We assume that instead of observing $\overline{u}_i$, each player observes $\overline{u}_i$ with some error ε_i. Specifically, player i observes

$$\hat{u}_{ij} = \overline{u}_{ij} + \varepsilon_{ij} \,, \tag{2.8}$$

where $\varepsilon_i = (\varepsilon_{i1}, \ldots, \varepsilon_{iJ_i})$ is player i's error vector. Each ε_i is distributed according to a joint distribution with a density function $f_i(\varepsilon_i)$, $f = (f_i, \ldots, f_n)$. f is admissible if the marginal distribution of f_i exists for each ε_{ij} and $E(\varepsilon_i) = 0$. We assume that for a given mixed-strategy profile s, player i will choose an action $a_{ij} \in A_i$, for which $\hat{u}_{ij}(s) \geq \hat{u}_{ik}(s) \, \forall k = 1, \ldots, J_i$.

2.3.2 Definition

For any expected payoff $x = (x_1, \ldots, x_n)$, where $x_i \in X_i$, and f, we can for each player i and each his action $a_{ij} \in A_i$ define the ij-response set $R_{ij} \subseteq X_i$:

$$R_{ij}(x) = \{\varepsilon_i \in X_i \mid x_{ij} + \varepsilon_{ij} \geq x_{ik} + \varepsilon_{ik} \, \forall k = 1, \ldots, J_i\} \,. \tag{2.9}$$

Given a mixed-strategy profile s, $R_{ij}(\overline{u}(s))$ specifies a region of errors, for which player i chooses the action a_{ij} against the profile s. For any $\varepsilon_i \in R_{ij}(\overline{u}(s))$, $\hat{u}_{ij}(s)$ is maximal among his available actions and therefore, he chooses to play the action a_{ij}. Figure 2.3 shows R_{ij} response sets of player 1 for $x_i = (2, 1)$ in a game where he has two available actions. Because he expects more utility from action a_{i1} (i.e, $x_{i1} > x_{i2}$), the boundary between the two response sets is shifted in favor of action a_{i1}.

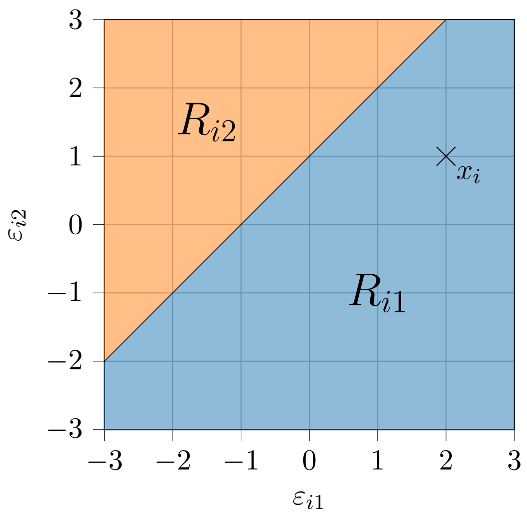

Figure 2.3: QRE ij-response sets R_{i1} and R_{i2} of player i for $x_i = (2, 1)$

We can then define

$$\sigma_{ij}(x) = \int_{R_{ij}(x)} f(\varepsilon) d\varepsilon \tag{2.10}$$

as the probability of player i playing the action a_{ij} given x. We compute the integral of errors' relative likelihood (i.e., $f(\varepsilon)$) over the errors for which a_{ij} is the best available action (given the noisy information).

Finally, we can define QRE as a vector $\pi \in S$, for which

$$\pi_{ij} = \int_{R_{ij}(\overline{u}(\pi))} f(\varepsilon) d\varepsilon \tag{2.11}$$

holds for every player i and every action a_{ij}.

Definition 2.9 (Quantal response equilibrium [MP96])**.** Let (N, A, u) be a normal-form game, and let f be admissible. A *Quantal response equilibrium* (*QRE*) is any $\pi \in S$ such that for all players i and $j = 1, \ldots, J_i$:

$$\pi_{ij} = \sigma_{ij}(\overline{u}(\pi)) \, .$$

The function $\sigma_i : X_i \mapsto S_i$ is called a quantal response function. It gives us a probability distribution over actions (i.e., a mixed strategy) given the expected (noisy) utilities of these actions. The probability of playing an action is positively related to the payoff from that action. QRE is actually a fixed point of $\sigma \circ \overline{u}$. Moreover, the existence of QRE is guaranteed for every normal-form game.

Theorem 2.10 ([MP96])**.** *For any normal-form game (N, A, u) and admissible f, there exists a QRE.*

2.3.3 Logit equilibrium

To be able to compute QRE, we must first specify a density function f for errors. The most common specification of QRE, and the one proposed in the original paper [MP96], is the logit equilibrium (LQRE).

LQRE is a parametric class QRE parametrized by $\lambda \geq 0$. We define the logit quantal response function $\sigma_i : X_i \mapsto S_i$ as

$$\sigma_{ij}(x_i) = \frac{e^{\lambda x_{ij}}}{\sum_{k=1}^{J_i} e^{\lambda x_{ik}}} \,. \tag{2.12}$$

This corresponds to f_i having an extreme value distribution, where the cumulative density function $F_i(\varepsilon_{ij}) = e^{-e^{-\lambda \varepsilon_{ij}}}$ and ε_{ij}'s are independent. If we use this distribution f_i for each player, we get

$$\pi_{ij} = \frac{e^{\lambda x_{ij}}}{\sum_{k=1}^{J_i} e^{\lambda x_{ik}}} \,, \tag{2.13}$$

where $x_{ij} = \overline{u}_{ij}(\pi)$.

2.3.4 Influence of λ parameter in logit equilibrium

The parameter $\lambda \geq 0$ in LQRE controls the level of error. $\lambda = 0$ means that the observations are full error. This gives us

$$\pi_{ij} = \frac{e^0}{\sum_{k=1}^{J_i} e^0} = \frac{1}{J_i} \,, \tag{2.14}$$

which means that the optimal strategy to play is the uniform strategy, where each action has the same probability of being played.

The higher we set λ, the smaller errors will be used. When $\lambda \to \infty$, the logit equilibrium approaches Nash equilibrium. We must, however, be careful with setting high λs, because $e^{\lambda x_{ij}}$ can easily overflow.

We define a simple two-player, zero-sum game to demonstrate the effect λ has on the LQRE value V. Both players have 4 available actions A, B, C and D. The utilities are shown in Figure 2.4.

		Player 2			
		A	B	C	D
Player 1	A	3,-3	-3, 3	-3, 3	-3, 3
	B	-3, 3	-3, 3	3,-3	3,-3
	C	-3, 3	3,-3	-3, 3	3,-3
	D	-3, 3	3,-3	3,-3	-3, 3

Figure 2.4: Payoff matrix of the zero-sum game for the demonstration of the influence of λ on the logit equilibrium

For comparison, we compute the Nash equilibrium strategies and value V using the maximin linear program from Equation (2.5). The game is symmetrical. Therefore, both players have the same optimal mixed strategy

$(0.4, 0.2, 0.2, 0.2)$. The player 1's value of the game in Nash equilibrium is -0.6. If both players adopted uniform strategy $(0.25, 0.25, 0.25, 0.25)$, player 1's utility would be -0.375.

We also solve this game using logit equilibrium for different values of λ. The results are shown in Figure 2.5. We can see that for $\lambda \geq 10$, the LQRE value is very close to Nash equilibrium value, and for very small λ, the value is close to -0.375.

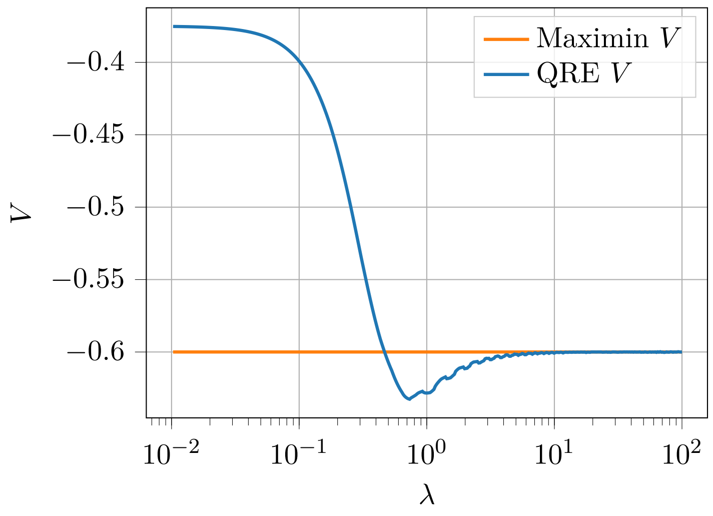

Figure 2.5: Influence of λ on logit equilibrium value of a game

2.3.5 Iterative method

As we said, QRE is a fixed point of $\sigma \circ \overline{u}$ and can be therefore found iteratively. We simply compute $\overline{u}$ followed by σ during each iteration until convergence. We will use the logit equilibrium definition of σ from the previous section.

We start by initializing $\pi_i^{(0)}$ with the uniform strategy (i.e., $\pi_{ij}^{(0)} = \frac{1}{J_i}$ for $i = 1, \ldots, J_i$) for each player i. Each iteration t consists of two steps. First, for every player i, the expected utilities of his actions $x_{ij}^{(t)}$ are computed against the strategies from the previous iteration $\pi_{-i}^{(t-1)}$:

$$x_{ij}^{(t)} = \overline{u}_{ij}(\pi^{(t-1)}) = u_i(a_{ij}, \pi_{-i}^{(t-1)})\,. \tag{2.15}$$

In the second step, for every player i, the new strategies $\pi_i'^{(t)}$ are computed from $x_i^{(t)}$ using σ:

$$\pi_{ij}'^{(t)} = \sigma_{ij}(x_i^{(t)}) = \frac{e^{\lambda x_{ij}^{(t)}}}{\sum_{k=1}^{J_i} e^{\lambda x_{ik}^{(t)}}}\,. \tag{2.16}$$

To guarantee a convergence, the new strategy is computed as the cumulative moving average of all previously seen strategies and the new strategy $\pi_i'^{(t)}$:

$$\pi_i^{(t)} = \frac{t \cdot \pi_i^{(t-1)} + \pi_i'^{(t)}}{t+1} \,. \tag{2.17}$$

We repeat these alternate steps until the strategies converge for all players, i.e., until

$$\pi_i^{(t)} \approx \pi_i^{(t-1)} \qquad \forall i = 1, \ldots, n \,. \tag{2.18}$$

We use some small precision constant ε_{qre} (e.g. 10^{-3}) to test for the approximate equality of the strategies. When the algorithm converges after t iterations, the logit equilibrium strategy for player i is $\pi_i^* = \pi_i^{(t)}$ and the corresponding QRE value of the game is $u(\pi^*)$.

Chapter 3

HSVI for POMDPs

The heuristic search value iteration algorithm (HSVI) for one-sided partially observable stochastic games (OS-POSGs) is a modification of HSVI for partially observable Markov decision processes (POMDPs). It is thus reasonable to first describe the original algorithm.

This chapter introduces Markov decision processes (MDPs) and how we can solve them using value iteration. Then we are going to describe the partially observable extension of MDPs - POMDPs. Solving POMDPs only by the plain value iteration is very inefficient. The solution to this performance issue is HSVI algorithm, which will be introduced at the end of this chapter.

The definitions of MDPs, POMDPs and value iteration algorithms in this chapter were taken from [RN09].

3.1 MDPs

Markov decision processes are used for representing and solving sequential decision problems. These are very similar to games. An agent takes actions in a sequence in a defined environment, and his goal is to maximize his expected utility. However, there is one main difference. The environment is not multiagent. It is not a game, so the only one who can act on the environment is the single agent.

Moreover, an MDP environment is nondeterministic (i.e, stochastic). This means that there may be multiple outcomes to a single action taken by the agent. These outcomes have predefined probabilities given by probability distributions.

For now, we assume that this environment is fully observable, meaning the agent always knows in which state he currently is.

Definition 3.1 (Markov decision process [RN09])**.** A *Markov decision process* is a tuple (S, A, T, R), where:

- S is a finite set of *states*, with initial state $s_0 \in S$;
- A is a finite set of *actions* the agent can take;
- $T : S \times A \times S \mapsto [0, 1]$ is a *transition probability function*, where $T(s' \mid s, a)$ is the probability of transitioning from state s to state s' by action a;

- $R : S \mapsto \mathbb{R}$ is a *reward function* assigning every state an utility reward.

The agent starts at time $t = 0$ from the initial state s_0. At a given time t, the agent is in a state s_t and can choose an action from the finite set of actions A. After he takes an action a_t, he gets a reward $r_t = R(s_t)$ and ends up in a state s_{t+1} sampled from the probability distribution given by $T(s_{t+1} \mid s_t, a_t)$. By choosing the correct sequence of actions, he is trying to maximize his accumulated reward. In finite-horizon MDPs, this goes on for a fixed number of stages. In infinite-horizon MDPs, this goes on forever.

We can also look at MDPs as search trees rooted at the initial state s_0. The agent navigates downward by choosing an action in each node. The transition probability function T determines the next node (i.e., state) after taking an action $a \in A$.

3.1.1 Discount factor

We will focus on MDPs with infinite horizon. This means that there is no limit on the number of stages the process goes through. Because of this, we need a way of comparing infinite sums of accumulated rewards.

The solution is to use a discount factor $\gamma \in (0, 1)$ that will make future rewards smaller. Instead of maximizing the simple sum of the rewards, we will maximize the discounted accumulated reward $\sum_{t=0}^{\infty} \gamma^t r_t$. As a result, the agent will prefer the current rewards over the ones he gets in the future.

3.1.2 Policy

Because the environment does not change in time, we can simplify the task of finding the optimal sequence of actions by finding only the optimal action for each state $s \in S$. If we know the optimal action for each state, we can easily use this information to find the optimal sequence.

To describe what action to take in a particular state, we will introduce the notion of policy. Policy π defines an action to take for each state $s \in S$. $\pi(s) = a$ tells us we should take the action a every time we find ourselves in the state s.

The expected utility of policy π starting in state s is given by

$$V^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right], \tag{3.1}$$

where rewards r_t were gained when following the policy.

An optimal policy π_s^* starting in state s is a policy that maximizes expected utility:

$$\pi_s^* = \arg\max_\pi V^\pi(s)\,. \tag{3.2}$$

Because the environment does not change in time, the optimal policy π^* is actually independent of s. It gives us the optimal action $\pi^*(s) = a$ for each state s.

3.1.3 Value Iteration

One of the algorithms for finding the optimal policy π^* is called the value iteration algorithm. We can rewrite the expected utility from previous section like this:

$$V(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} T(s' \mid s, a) V(s') \,. \tag{3.3}$$

This is called a Bellman equation. Given that $\max_{a \in A(s)}$ chooses the best available action a (similarly to an optimal policy π^*), the expected utility $V(s)$ of state s is the sum of the immediate reward $R(s)$ and the discounted expected utilities of neighboring states s' weighted by the probability $T(s'|s,a)$ that we end up in them.

Putting together all Bellman equations for n states gives us a system of n equations with n unknowns. Because of the $\max_{a \in A(s)}$ operation in the Bellman equations, we have to use iterative methods for solving the system. The iterative method, in this case, is called the value iteration.

We initialize the values of $V(s)$ to arbitrary numbers. Then in each step, we compute the left-hand sides $V(s)$ of the equations from the right-hand sides. After each step, we update the right-hand side of each equation by plugging in the newly computed value. This step is called the Bellman update, is applied simultaneously to all states $s \in S$ and can be written like this:

$$V_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A} \sum_{s' \in S} T(s' \mid s, a) V_i(s') \,. \tag{3.4}$$

The value iteration algorithm eventually converges and gives us a unique solution for the optimal policy π^*. For each state s, the optimal action to take is the maximal argument of the $\max_{a \in A(s)}$ operation in the final system of Bellman equations.

3.2 POMDPs

The extension of MDP, where the agent does not know the state he is currently in, is called the partially observable MDP. In this stochastic environment, instead of getting the information in which state s' the agent ends up after taking an action a, he gets only an observation o from a finite set of observations O. He can then use this information to deduce the true state (or at least its distribution).

Definition 3.2 (Partially observable markov decision process [RN09, HBKK20]). A *Partially observable markov decision process* is a tuple (S, A, O, T, R), where:

- S is a finite set of *states*;
- A is a finite set of *actions* the agent can take;
- O is a finite set of *observations* the agent can observe;

- $T : S \times A \times O \times S \mapsto [0, 1]$ is a *transition probability function*, where $T(o, s' \mid s, a)$ is the probability of transitioning from state s to state s' by taking action a while observing observation o;
- $R : S \times A \mapsto \mathbb{R}$ is a *reward function*, where $R(s, a)$ is an immediate reward for taking action a in state s.

As we said, the agent does not have to know the true state in each stage of the game. Instead, the agent keeps internal information telling him which states he might be in and which are more likely to be the true state. This estimation concept is called the belief state.

A belief state is a probability distribution over the set of states S. Given a belief state b, $b(s)$ is the probability that the agent is in state s. Furthermore, in POMDPs, the agent does not know the initial state. Instead, the agent starts with an initial belief state b_0 information, which gives us the probability $b_0(s)$ that s is an initial state.

The visualization of POMDPs as search trees can still hold as with MDPs. However, the agent does not know the true state. Instead of representing states, the nodes will represent belief states, starting with the initial belief b_0 at the tree's root. At each node, the agent chooses an action $a \in A$, receives an observation $o \in O$, sampled from the transition probability function T, and ends up in a new node (i.e., belief state) $\tau(b, a, o)$.

Given belief b, action a and observation o, the agent can deduce the next belief $\tau(b, a, o)$ after taking a in s and observing o as

$$\tau(b, a, o)(s') = \eta \sum_{s \in S} b(s) T(o, s' \mid s, a)\,, \tag{3.5}$$

where η is a normalizing constant such that $\sum_{s' \in S} \tau(b, a, o)(s')$ is equal to 1. While taking action a_t in belief state b_t at time t he also gets a reward $r_t = \sum_{s \in S} b_t(s) R(s, a_t)$.

3.2.1 Policy

Similarly to MDPs, there exists a policy concept for POMDPs, which recommends when to take which action. However, in POMDPs, the agent does not know the true state. He only knows the probability distribution given by the belief state b. Because of this, in POMDPs, the policy $\pi(b)$ maps from a belief state to an action.

We can define the expected utility of policy π, starting from a belief state b, in a similar manner as in MDPs:

$$V^{\pi}(b) = E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]\,. \tag{3.6}$$

As in MDPs, the optimal policy π^* is a policy that maximizes the expected utility and chooses an optimal action for each belief state.

3.2.2 Value Iteration

Like with MDPs, we can rewrite the expected utility in the form of a Bellman equation - in terms of immediate reward and discounted expected utility of the subsequent beliefs. This gives us the Bellman update equivalent for POMDPs. We will use the operator H for the Bellman update:

$$HV(b) = \max_{a \in A} \left[\sum_{s \in S} b(s) R(s,a) + \gamma \sum_{o \in O} \mathbb{P}_b[o \mid a] V(\tau(b,a,o)) \right], \tag{3.7}$$

where

$$\mathbb{P}_b[o \mid a] = \sum_{s \in S} b(s) \sum_{s' \in S} T(o, s' \mid s, a) \tag{3.8}$$

is the probability of observing observation o after taking action a in belief state b.

This can be also rewritten in terms of the value of taking action a in belief state b:

$$Q^V(b,a) = \sum_{s \in S} b(s) R(s,a) + \gamma \sum_{o \in O} \mathbb{P}_b[o \mid a] V(\tau(b,a,o)) \,. \tag{3.9}$$

The Bellman update H is then the maximum of $Q^V(b,a)$ over $a \in A$:

$$HV(b) = \max_{a \in A} Q^V(b,a) \,. \tag{3.10}$$

Because there is an infinite number of belief states, we cannot use this representation for value iteration as we did with MDPs. Instead, we will represent the value function V as a piecewise linear and convex (PWLC) function composed of a finite set of α-vectors and replace Bellman update with the operation of computing a new set of α-vectors.

3.2.3 Value function representation

Given n states, the space of all belief states can be seen as n-simplex, with the n states being its vertices. We denote this simplex by $\Delta(S)$. An α-vector is a linear (affine) function $\alpha : \Delta(S) \mapsto \mathbb{R}$ that assigns a real value to each belief state $b \in \Delta(S)$. It is a hyperplane in a $(|S|+1)$-dimensional space. Because it is linear, it can be defined only by values in the vertices of the $\Delta(S)$ simplex (i.e., states). Value in arbitrary belief $b \in \Delta(S)$ can be then computed as $\alpha(b) = \sum_{s \in S} b(s)\alpha(s)$ or simply as dot product $\alpha \cdot b$.

This α-vector can represent an arbitrary fixed strategy. Here, strategy has the same meaning as in game theory. It can be a pure strategy (i.e., single action) or a mixed strategy (i.e., probability distribution over available actions). However, the strategy is always the same independent of the current belief b.

The PWLC value function V is then defined by a finite set Γ of α-vectors (i.e., strategies). For each belief b, we choose the best available fixed strategy represented by some α. In other words, the value of the value function V in an arbitrary belief state b is a maximal projection of b onto the set Γ:

$$V(b) = \max_{\alpha \in \Gamma} \alpha(b) . \tag{3.11}$$

This divides the belief space simplex $\Delta(S)$ into regions, for which the optimal strategy is the same. Below, in Figure 3.1, is an example of a value function in one-dimensional belief space (i.e., with only two states) composed of α-vectors. Each α-vector is fully defined by its values in these two states. The final value in each belief state, i.e., the maximal projection onto the set of α-vectors, is shown in bold blue line.

We can see that some α-vectors do not contribute to the value function V (shown in orange dashed line). These α-vectors are said to be dominated. If more than one vector is needed to dominate a strategy, it is weakly dominated. If one vector is enough to dominate a strategy, it is called strongly dominated. Because dominated strategies do not contribute to the value function V, they can be removed from the set Γ.

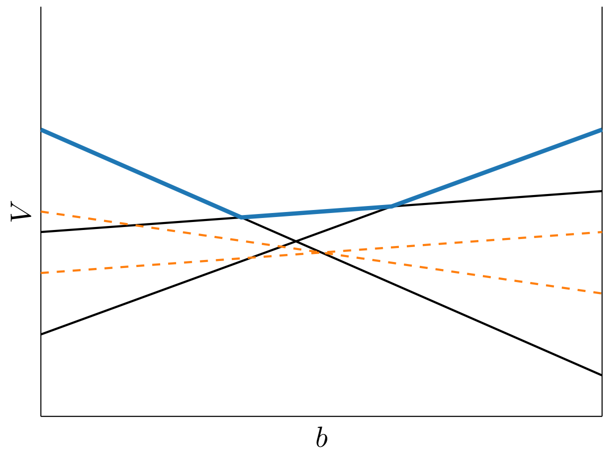

Figure 3.1: Representation of value function composed of α-vectors

The value function can be updated by computing a new set of α-vectors Γ^{t+1} composed of

$$\alpha : \Delta(S) \mapsto \mathbb{R} \mid \alpha(s) = R(s,a) + \gamma \sum_{(o,s') \in O \times S} T(o, s' \mid s, a) \alpha^o(s') , \tag{3.12}$$

for some $a \in A$ and $\alpha^o \in \Gamma^t$, $o \in O$, where α^o is a strategy for observation o. This approach, unfortunately, does not scale well. The number of combinations between A, Γ^t and O is too high and grows exponentially. Because of this, we need to use a different approach that will approximate the PWLC value function V.

3.3 HSVI

Due to the time complexity of the exact value iteration, we cannot compute the optimal value function V exactly. Instead, we will approximate its value by two PWLC functions V_{LB}^{Γ} and V_{UB}^{Υ}. These two functions will be the lower bound and the upper bound of the optimal value function V.

We are going to recursively explore the search tree driven by a heuristic until the gap between the bounds in the initial belief state b_0 is smaller than the predefined precision ε. In each visited belief state, the bounds will be refined by a point-based update that is derived from the Bellman update H. These updates will bring the bounds closer together, effectively shrinking the gap and approximating V.

The original HSVI for POMDPs was introduced by [SS12b]. However, we are going to present a variant from [HBKK20] with a slightly altered notation.

3.3.1 Bounds' representation

The α-vector representation works fine for the lower bound V_{LB}^{Γ}. However, this representation does not make sense for the upper bound V_{UB}^{Υ}. We need the function to be PWLC and to decrease with each point-based update. Instead, we will use the point set representation.

The upper bound will be represented by a finite set Υ of belief-value pairs (b, v), where $b \in \Delta(S)$ and $v \in \mathbb{R}$. We can imagine these pairs as points above (or below for a negative value v) the simplex $\Delta(S)$ in $(|S|+1)$-dimensional space. The value of $V_{UB}^{\Upsilon}(b)$ in belief state b is computed as a projection of b onto the convex hull of Υ:

$$V_{UB}^{\Upsilon}(b) = \min \left\{ \sum_{i=1}^{m} \lambda_i v_i \mid \lambda \in \mathbb{R}_{\geq 0}^{m} : \sum_{i=1}^{m} \lambda_i = 1 \wedge \sum_{i=1}^{m} \lambda_i b_i = b \right\}, \tag{3.13}$$

where $m = |\Upsilon|$. This requires solving a linear program.

We also define the width of the gap between $V_{LB}^{\Gamma}(b)$ and $V_{UB}^{\Upsilon}(b)$ in belief b simply as

$$\text{width}(V(b)) = V_{UB}^{\Upsilon}(b) - V_{LB}^{\Gamma}(b)\,. \tag{3.14}$$

There are multiple approaches to the initialization of the bounds. It plays an important role in the performance of the algorithm. The smaller the initial gap is, the faster the algorithm will converge to the desired precision.

The easiest way how to initialize the bounds is to use the discounted sum of the smallest and the largest possible immediate reward:

$$R_{min} = \min_{(s,a)\in S\times A} R(s,a) \tag{3.15}$$

$$R_{max} = \max_{(s,a)\in S\times A} R(s,a) \tag{3.16}$$

$$L = \sum_{t=1}^{\infty} \gamma^t R_{min} = \frac{R_{min}}{1-\gamma} \tag{3.17}$$

$$U = \sum_{t=1}^{\infty} \gamma^t R_{max} = \frac{R_{max}}{1-\gamma}\,. \tag{3.18}$$

These values are then used for the whole belief simplex $\Delta(S)$. This can be achieved by initializing Γ with one α-vector and Υ with a point for each state s:

$$\begin{aligned}\Gamma &= \{\alpha \mid \alpha(s) = L, \forall s \in S\}\\ \Upsilon &= \{(s,U) \mid \forall s \in S\}\,.\end{aligned} \tag{3.19}$$

This sure is a viable initialization of the bounds, but it is not very good. The L and U are actually the smallest and largest possible value of the game, respectively. The gap between them is very large in general.

A more suitable way of initializing the upper bound is to initialize it with the value obtained by solving the same problem as MDP (i.e., with perfect information). The lower bound can be initialized, for example, by constructing a lower bound of the strategy '*always play the same single action*':

$$\underline{R_a} = \min_{s\in S} \frac{R(s,a)}{1-\gamma}\,, \tag{3.20}$$

for each action a. We can then easily choose the action for which this lower bound is the largest:

$$L' = \max_{a\in A} \underline{R_a}\,. \tag{3.21}$$

3.3.2 Point-based updates

We can use the equivalent of Bellman update H for POMDPs from Equation (3.7) to derive the point-based updates for V^{Γ}_{LB} and V^{Υ}_{UB}.

To update the upper bound V^{Υ}_{UB} in belief state b, we simply compute $HV^{\Gamma}_{LB}(b)$ and add the new value with the belief b into Υ:

$$\Upsilon = \Upsilon \cup (b, HV^{\Upsilon}_{UB}(b))\,. \tag{3.22}$$

The procedure of updating V^{Γ}_{LB} is more complex. We have to find a new α-vector that contains the point $(b, HV^{\Gamma}_{LB}(b))$ by Bellman backup operator with the use of gradient information. The point-based `update` procedure for both bounds is shown in Algorithm 1.

```
Algorithm 1: Point-based update(b) of $V_{LB}^{\Gamma}$ and $V_{UB}^{\Upsilon}$
1 $\alpha^{a,o} \leftarrow \arg\max_{\alpha \in \Gamma} \sum_{s' \in S} \tau(b,a,o)(s')\alpha(s')$ for all $a \in A, o \in O$
2 $\alpha^{a}(s) \leftarrow R(s,a) + \gamma \sum_{o,s'} T(o,s' \mid s,a)\alpha^{a,o}(s')$ for all $s \in S, a \in A$
3 $\Gamma \leftarrow \Gamma \cup \{\arg\max_{\alpha^a} \sum_{s \in S} b(s)\alpha^a(s)\}$
4 $v^a \leftarrow \sum_{s \in S} b(s)R(s,a) + \gamma \sum_{o \in O} \mathbb{P}_b[o \mid a] V_{UB}^{\Upsilon}(\tau(b,a,o))$
5 $\Upsilon \leftarrow \Upsilon \cup \{(b, \max_{a \in A} v^a)\}$
```

The point-based update is also demonstrated in Figure 3.2.

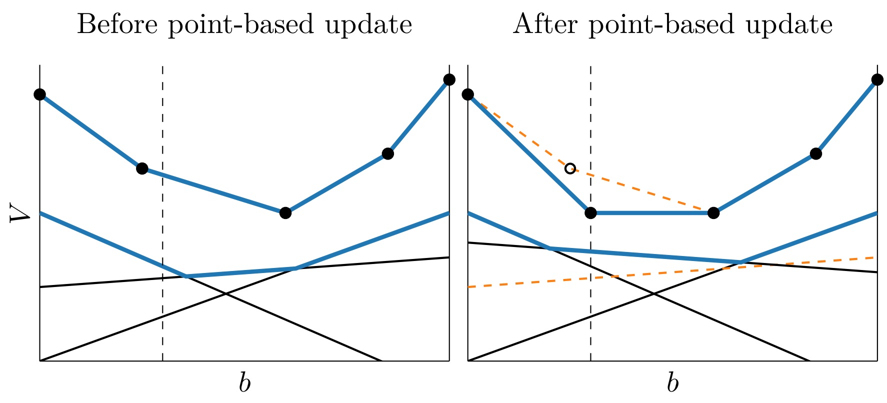

Figure 3.2: Point-based update of V_{LB}^{Γ} and V_{UB}^{Υ}

3.3.3 Excess gap

For given belief s and action a, we can combine the definition of $Q^V(b,a)$ from Equation (3.9) with the definition of width from Equation (3.14) and get

$$\text{width}(Q^V(b,a)) = \gamma \sum_{o \in O} \mathbb{P}_b[o \mid a] \text{width}(V(\tau(b,a,o))) \,. \tag{3.23}$$

We can see that after an update, the width at b is γ-times the sum of probability-weighted widths at child nodes. Because of this, the termination condition for exploration depends on the depth of the currently explored belief b. The deeper we dive into the search tree, the more can we loosen the requirement on width. We use the inverse of the discount factor γ to loosen the requirement on the gap each time we progress further in the search tree. We can express this concept by an excess gap in belief state b at depth t:

$$\text{excess}(b,t) = \text{width}(V(b)) - \varepsilon\gamma^{-t} \,. \tag{3.24}$$

In the initial belief b_0 at depth $t = 0$, this is simply the difference between the current width and the desired precision:

$$\text{excess}(b_0,0) = \text{width}(V(b_0)) - \varepsilon \,. \tag{3.25}$$

We restart the exploration when the excess gap in the currently explored belief b is smaller than or equal to zero:

$$\operatorname{excess}(b,t) \leq 0\,, \tag{3.26}$$

or equivalently

$$\operatorname{width}(V(b)) \leq \varepsilon\gamma^{-t}\,. \tag{3.27}$$

3.3.4 Exploration

As we already said, the algorithm recursively explores one path of the search tree. It does so starting from the initial belief state s_0. The nodes to explore are selected by a heuristic. The heuristic selects an optimal action a^* and observation o^* with an attempt to close the gap between $V^{\Gamma}_{LB}(b_0)$ and $V^{\Upsilon}_{UB}(b_0)$ in the initial belief state b_0 as fast as possible. When the exploration reaches a point, where the condition on the width of the gap between the lower bound and the upper bound is met, it restarts the search from the initial belief s_0.

The value of $HV^{\Upsilon}_{UB}(b)$ is determined by the maximum of $Q^{V^{\Upsilon}_{UB}}(b,a)$ over $a \in A$. Similarly, the value of $HV^{\Gamma}_{LB}(b)$ is determined by the maximum of $Q^{V^{\Gamma}_{LB}}(b,a)$ over $a \in A$. By point-based update, the lower bound can only grow, and the value of the upper bound can only decrease. It is thus reasonable to choose as action a^* the action with maximal $Q^{V^{\Upsilon}_{UB}}(b,a)$.

If we instead choose a suboptimal action a^* for which $Q^{V^{\Gamma}_{LB}}(b,a)$ is maximal, the value of $Q^{V^{\Gamma}_{LB}}(b,a)$ could only grow. This would make us select the same action a^* again the next time we found ourselves in the belief b, never finding out its suboptimality. On the other hand, choosing a suboptimal action $a^* = \arg\max_a Q^{V^{\Upsilon}_{UB}}(b,a)$ does not suffer from this problem. The value of $Q^{V^{\Upsilon}_{UB}}(b,a)$ will decrease, eventually making us discover the suboptimality of a^*.

Similar to Equation (3.23), the excess gap can also be rewritten in terms of the excess gaps of child nodes:

$$\operatorname{excess}(b,t) \leq \sum_{o\in O} \mathbb{P}_b[o \mid a^*]\operatorname{excess}(\tau(b,a^*,o),t+1)\,. \tag{3.28}$$

To shrink the excess gap at b as fast as possible, we can choose as o^* the observation that contributes the most to this sum.

The choice of a^* and o^* can be summarized as

$$\begin{aligned} a^* &\leftarrow \arg\max_{a\in A}\left[\sum_{s\in S} b(s)R(s,a) + \gamma\sum_{o\in O}\mathbb{P}_b[o \mid a]V^{\Upsilon}_{UB}(\tau(b,a,o))\right] \\ o^* &\leftarrow \arg\max_{o\in O}\mathbb{P}_b[o \mid a^*]\operatorname{excess}(\tau(b,a^*,o),t+1)\,. \end{aligned} \tag{3.29}$$

3.3.5 Algorithm

We can now summarize the whole HSVI algorithm. The algorithm starts by initializing the bounds. We repeatedly call the recursive `explore` procedure in the initial belief b_0. At each depth, we update the bounds using the `update` procedure. We also find the optimal action a^* and observation o^* with the exploration heuristic. The algorithm then proceeds with a recursive call to `explore` with updated belief $\tau(b, a^*, o^*)$. The exploration terminates when the condition on the excess gap at depth t is met. When the algorithm returns from the recursive call and folds the exploration, it calls the `update` procedure again. The whole algorithm terminates when the excess gap condition is met in the initial belief b_0 at depth 0.

Algorithm 2: HSVI for POMDPs

```
1 Initialize $V_{LB}^{\Gamma}$ and $V_{UB}^{\Upsilon}$
2 while excess$(b_0, 0) > 0$ do explore$(b_0, \varepsilon, 0)$
3 Procedure explore$(b, \varepsilon, t)$
4     if excess$(b, t) \leq 0$ then return
5     update(b)
6     $a^* \leftarrow$
        $\arg\max_{a \in A} \left[\sum_{s \in S} b(s) R(s,a) + \gamma \sum_{o \in O} \mathbb{P}_b[o \mid a] V_{UB}^{\Upsilon}(\tau(b,a,o))\right]$
7     $o^* \leftarrow \arg\max_{o \in O} \mathbb{P}_b[o \mid a^*] \text{excess}(\tau(b, a^*, o), t+1)$
8     explore$(\tau(b, a^*, o^*), \varepsilon, t+1)$
9     update(b)
```

Chapter 4
HSVI for OS-POSGs

As we mentioned, HSVI for OS-POSGs is heavily based on the POMDP variant. This is possible because we can think of partially observable stochastic games (POSGs) as a multiagent generalization of POMDPs. We will focus specifically on one-sided POSGs (OS-POSGs), which are two-player games where only one side has full information about the game's current state.

This chapter describes HSVI for OS-POSGs algorithm, which was introduced in [HBP17, HBKK20].

4.1 OS-POSGs

We start by defining the OS-POSGSs and comparing them with the POMDPs.

Definition 4.1 (One-sided partially observable stochastic game [HBKK20]). A *one-sided partially observable stochastic game* is a tuple
$G = (S, A_1, A_2, O, T, R)$ where:

- S is a finite set of *states*;
- A_1 and A_2 are finite set of *actions* of player 1 and player 2, respectively;
- O is a finite set of *observations* player 1 can observe;
- $T : S \times A_1 \times A_2 \times O \times S \mapsto [0, 1]$ is a *transition probability function*, where $T(o, s' \mid s, a_1, a_2)$ is the probability of transitioning from state s to state s' by actions a_1 and a_2 of player 1 and player 2, respectively, while observing observation o;
- $R : S \times A_1 \times A_2 \mapsto \mathbb{R}$ is a *reward function* of player 1, where $R(s, a_1, a_2)$ is an immediate reward for taking action a_1 in state s while player 2 takes action a_2.

Similarly, as with POMDPs, the game starts in an initial state s_0 that is sampled from the initial belief state b_0. However, in each stage of the game, both players choose their actions and play them simultaneously. The following state, along with the observation, is sampled from the transition probability function T.

If we look at OS-POSGs from the side of player 1, the one which has imperfect information, we can see the similarities they share with POMDPs. Player 1 does not know the true state of the game. Instead, he uses a belief state b, a probability distribution over states $s \in S$, to reason about the game. He chooses an action in each stage of the game $a_1 \in A_1$, plays it, and obtains a reward r_t. We will also discount this reward obtained at depth t by γ^t. After the game progresses to the next stage, he observes only an observation $o \in O$ and uses this information to update his belief state.

On the other hand, player 2 knows the true state of the game and his own action $a_2 \in A_2$ in addition to the information player 1 has. Because we consider zero-sum games, player 2's goal is to minimize the reward of player 1.

4.2 HSVI

Because there are now two players, instead of simply choosing the optimal action as in POMDPs, we will have to solve a two-player game in each stage. We will now introduce the necessary concepts that are needed for solving a stage game.

4.2.1 Strategies

For reasoning about actions in a stage game of OS-POSGs, we will use the same concept of strategy as we introduced for normal-form games in Section 2.1.2. Even though we use π to denote a stage strategy, it is a bit different concept than policy in POMDPs. While the policy π in POMDPs maps from belief to a single action, in OS-POSGS, the stage strategy π is a mixed strategy for a two-player stage game.

The stage strategy of player 1 is a distribution over all his available actions $\pi_1 \in \Pi(A_1)$. Because player 2 knows the true state, his stage strategy is a mapping from state to a distribution over his actions $\pi_2 : S \mapsto \Pi(A_2)$. Due to the strategy of player 2 being a conditional probability, we will use $\pi_2(a_2 \mid s)$ notation for it. We can denote Π_1 and Π_2 the sets of all strategies of player 1 and player 2, respectively.

Using the strategies of both players, the belief state b and the transition probability function T, we can derive probabilities of different events. We can express what is the probability of a stage being in state s, players playing actions a_1 and a_2, and proceeding to next state s' while observing o:

$$\mathbb{P}_{b,\pi_1,\pi_2}[s, a_1, a_2, o, s'] = b(s)\pi_1(a_1)\pi_2(a_2 \mid s)T(o, s' \mid s, a_1, a_2)\,. \tag{4.1}$$

By marginalizing this probability, we can obtain the probability of player 1 playing action a_1 and observing o:

$$\mathbb{P}_{b,\pi_1,\pi_2}[a_1, o] = \sum_{(s,a_2,s') \in S \times A_2 \times S} \mathbb{P}_{b,\pi_1,\pi_2}[s, a_1, a_2, o, s']\,. \tag{4.2}$$

4.2.2 Belief update

During the exploration phase of HSVI algorithm, we will need to be able to deduce the new belief in a subsequent stage. Player 1 has the information about the current belief state b, strategies π_1, π_2 of both players, his action a_1 and outcome in the form of an observation o. Using this knowledge, the next belief can be deduced in the following way:

$$\begin{aligned}\tau(b,\pi_1,\pi_2,a_1,o)(s') &= \mathbb{P}_{b,\pi_1,\pi_2}[s' \mid a_1,o] \\ &= \frac{1}{\mathbb{P}_{b,\pi_1,\pi_2}[a_1,o]} \sum_{(s,a_2)\in S\times A_2} \mathbb{P}_{b,\pi_1,\pi_2}[s,a_1,a_2,o,s'] \,.\end{aligned} \tag{4.3}$$

4.2.3 Stage game

Solving a stage game in OS-POSGs is the equivalent of Bellman update H in POMDPs. Each stage game can be thought of as a separate zero-sum game. Because of this, we can use the minimax or maximin value introduced in Section 2.2 to solve them.

Given stage game strategies π_1 and π_2, we can construct the Bellman equation for belief b by summing the expected immediate reward and the discounted future reward:

$$V_{\pi_1,\pi_2}(b) = \mathbb{E}_{b,\pi_1,\pi_2}[R(s,a_1,a_2)] + \gamma \sum_{(a_1,o)\in A_1\times O} \mathbb{P}_{b,\pi_1,\pi_2}[a_1,o] V(\tau(b,\pi_1,\pi_2,a_1,o)) \,, \tag{4.4}$$

where

$$\mathbb{E}_{b,\pi_1,\pi_2}[R(s,a_1,a_2)] = \sum_{(s,a_1,a_2)\in S\times A_1\times A_2} b(s)\pi_1(a_1)\pi_2(a_2 \mid s)R(s,a_1,a_2) \,. \tag{4.5}$$

Finding the minimax or maximin value of this Bellman equation over all strategies π_1, π_2 solves the game:

$$\begin{aligned}HV(b) &= \max_{\pi_1\in\Pi_1} \min_{\pi_2\in\Pi_2} V_{\pi_1,\pi_2}(b) \\ &= \min_{\pi_2\in\Pi_2} \max_{\pi_1\in\Pi_1} V_{\pi_1,\pi_2}(b).\end{aligned} \tag{4.6}$$

If the value function V is represented by α-vectors, the Bellman update H can be solved by a linear program:

$$\max_{V,\pi_1,\hat{\alpha},\hat{\lambda}} \quad \sum_{s\in S} b(s)V(s) \tag{4.7a}$$

$$\text{s.t.} \quad V(s) \le \sum_{a_1\in A_1} \pi_1(a_1)R(s,a_1,a_2) + \gamma \sum_{(a,o,s')\in A_1\times O\times S} T(o,s' \mid s,a_1,a_2)\hat{\alpha}^{a_1,o}(s') \qquad \forall (s,a_2)\in S\times A_2 \tag{4.7b}$$

$$\hat{\alpha}^{a_1,o}(s') = \sum_{i=1}^{|\Gamma|} \hat{\lambda}_i^{a_1,o}\alpha_i(s') \qquad \forall (a_1,o,s')\in A_1\times O\times S \tag{4.7c}$$

$$\sum_{i=1}^{|\Gamma|} \hat{\lambda}_i^{a_1,o} = \pi_1(a_1) \qquad \forall (a_1,o)\in A_1\times O \tag{4.7d}$$

$$\sum_{a_1\in A_1} \pi_1(a_1) = 1 \tag{4.7e}$$

$$\pi_1(a_1)\ge 0 \qquad \forall a_1\in A_1 \tag{4.7f}$$

$$\hat{\lambda}_i^{a_1,o}\ge 0 \qquad \forall (a_1,o)\in A_1\times O, 1\le i\le |\Gamma| \tag{4.7g}$$

The dual formulation of this linear program is

$$\min_{V,\pi_2,\hat{V},\hat{\tau}} \quad V \tag{4.8a}$$

$$\text{s.t.} \quad V \ge \sum_{(s,a_2)\in S\times A_2} \pi_2(s\wedge a_2)R(s,a_1,a_2) + \gamma \sum_{o\in O} \hat{V}(a1,o) \qquad \forall a_1 \tag{4.8b}$$

$$\hat{V}(a_1,o) \ge \sum_{s'\in S} \hat{\tau}(b,\pi_1,\pi_2,a_1,o)(s')\alpha_i(s') \qquad \forall (a_1,o), 1\le i\le |\Upsilon| \tag{4.8c}$$

$$\hat{\tau}(b,\pi_1,\pi_2,a_1,o)(s') = \sum_{(s,a_2)\in S\times A_2} T(o,s' \mid s,a_1,a_2)\pi_2(s\wedge a_2) \qquad \forall (a_1,o,s') \tag{4.8d}$$

$$\sum_{a_2\in A_2} \pi_2(s\wedge a_2) = b(s) \qquad \forall s \tag{4.8e}$$

$$\pi_2(s\wedge a_2)\ge 0 \qquad \forall (s,a_2) \tag{4.8f}$$

To improve the readability of the dual formulation, the strategy of player 2 is used in the form of joint probability. The conditional probability can be obtained like this:

$$\pi_2(a_2 \mid s) = \frac{\pi_2(s\wedge a_2)}{b(s)}\,. \tag{4.9}$$

These versions of linear programs are fine for the α-vector representation of the value function. For the representation by a set Υ of points, we have to replace the constraint (4.8c) in the dual formulation, which is responsible for the value of subgame, by these equations:

$$\hat{V}(a_1, o) = \sum_{i=1}^{|\Upsilon|} \lambda_i^{a_1,o} v_i + \delta \sum_{s' \in S} \Delta_{a_1,o}^{s'} \qquad \forall (a_1, o) \in A_1 \times O \tag{4.10a}$$

$$\sum_{i=1}^{|\Upsilon|} \lambda_i^{a_1,o} b_i(s') = b'_{a_1,o}(s') \qquad \forall (a_1, o, s') \in A_1 \times O \times S \tag{4.10b}$$

$$\Delta_{a_1,o}^{s'} \geq b'_{a_1,o}(s') - \hat{\tau}(b, \pi_1, \pi_2, a_1, o)(s') \qquad \forall (a_1, o, s') \in A_1 \times O \times S \tag{4.10c}$$

$$\Delta_{a_1,o}^{s'} \geq \hat{\tau}(b, \pi_1, \pi_2, a_1, o)(s') - b'_{a_1,o}(s') \qquad \forall (a_1, o, s') \in A_1 \times O \times S \tag{4.10d}$$

$$\sum_{i=1}^{|\Upsilon|} \lambda_i^{a_1,o} = \sum_{s' \in S} \hat{\tau}(b, \pi_1, \pi_2, a_1, o)(s') \qquad \forall (a_1, o) \in A_1 \times O \tag{4.10e}$$

$$\lambda_i^{a_1,o} \geq 0 \qquad \forall (a_1, o) \in A_1 \times O, 1 \leq i \leq |\Upsilon| \tag{4.10f}$$

These equations are based on the linear program for computing the value of the upper bound function (Equation (3.13)). Here, $\delta := (U - L)/2$ denotes a Lipschitz constant, which makes sure that V_{UB}^{Υ} is δ-*Lipschitz* continuous (both V_{LB}^{Γ} and V_{UB}^{Υ} are required to be δ-*Lipschitz* continuous for the correctness of the algorithm).

4.2.4 Bounds initialization

Same as with POMDPs, the lower bound V_{LB}^{Γ} is represented by a finite set Γ of α-vectors. The upper bound V_{UB}^{Υ} is represented by a finite set Υ of belief-value pairs (b, v).

The bounds can be initialized similarly as in the POMDP variant of the algorithm. Easy, but not very good, initialization is to use the discounted sum of minimal and maximal immediate rewards L and U for the lower bound and the upper bound, respectively:

$$R_{min} = \min_{(s,a_1,a_2) \in S \times A_1 \times A_2} R(s, a_1, a_2) \tag{4.11}$$

$$R_{max} = \max_{(s,a_1,a_2) \in S \times A_1 \times A_2} R(s, a_1, a_2) \tag{4.12}$$

$$L = \sum_{t=1}^{\infty} \gamma^t R_{min} = \frac{R_{min}}{1 - \gamma} \tag{4.13}$$

$$U = \sum_{t=1}^{\infty} \gamma^t R_{max} = \frac{R_{max}}{1 - \gamma} \, . \tag{4.14}$$

Better initialization for the upper bound is to solve the perfect information variant of the game. The lower bound can be initialized by playing a uniform strategy (i.e., all actions with the same probability) by player 1.

4.2.5 Excess gap

We also have to change the way how the excess gap is computed, because of the influence of player 2:

$$\rho(0) = \varepsilon \tag{4.15}$$

$$\rho(t+1) = \frac{\rho(t) - 2\delta D}{\gamma} \tag{4.16}$$

$$\text{excess}(b,t) = \text{width}(V(b)) - \rho(t)\,, \tag{4.17}$$

where $D \in (0, \frac{(1-\gamma)\varepsilon}{2\delta})$ is a neighborhood parameter. The role of D is to ensure that the sequence of widths $\rho(t)$ is monotonically increasing and unbounded.

4.2.6 Exploration

The exploration heuristic that drives the algorithm through the search tree is also very similar to the POMDP variant. One difference is that the Bellman update H cannot select the optimal action a^* for us. It only selects the equilibrium strategies π_1^{UB} and π_2^{LB}. We thus select the optimal action a^* and observation o^* as the pair that contributes the most to the excess gap at the initial belief b_0:

$$(a_1^*, o^*) \leftarrow \operatorname*{arg\,max}_{(a_1,o)\in A_1\times O} \mathbb{P}_{b,\pi_1^{UB},\pi_2^{LB}}[a_1,o]\text{excess}_{t+1}(\tau(b,\pi_1^{UB},\pi_2^{LB},a_1,o))\,. \tag{4.18}$$

If the excess gap at next belief $\text{excess}_{t+1}(\tau(b,\pi_1^{UB},\pi_2^{LB},a_1^*,o^*))$ is smaller or equal to zero, we restart the exploration.

4.2.7 Algorithm

The linear programs (4.7) and (4.8) (with (4.10) replacement) solve the stage game at lines 4 and 5 in the Algorithm 3. They give us the equilibrium strategy for player 1 π_1^{UB}, the equilibrium strategy for player 2 π_2^{LB}, and at the same time compute the new α-vector and pair (b,v) for the bounds' update. The $V(s)$ in the primal (4.7) gives us the values for the new α-vector and the V in the dual (4.8) is the value v in new pair. The `update`(b) procedure adds those values to Γ and Υ, respectively.

When we return from the recursive call to `explore`, the point-based update is done once more. Because of this, we must compute new α-vector and pair (b,v) for the `update` procedure using the linear programs. This time, however, we are not interested in the strategies.

The whole algorithm is summarized below.

Algorithm 3: HSVI for OS-POSGs [HBKK20]

```
1  Initialize $V_{LB}^{\Gamma}$ and $V_{UB}^{\Upsilon}$
2  while excess$(b_0, 0) > 0$ do explore$(b_0, \varepsilon, 0)$
3  Procedure explore$(b, \varepsilon, t)$
4      $(\pi_1^{LB}, \pi_2^{LB}) \leftarrow$ equilibrium strategy profile in $HV_{LB}^{\Gamma}(b)$
5      $(\pi_1^{UB}, \pi_2^{UB}) \leftarrow$ equilibrium strategy profile in $HV_{UB}^{\Upsilon}(b)$
6      update(b)
7      $(a^*, o^*) \leftarrow$ select according to forward exploration heuristic
8      if $\mathbb{P}_{b,\pi_1^{UB},\pi_2^{LB}}[a_1^*, o^*] \cdot \text{excess}_{t+1}(\tau(b, \pi_1^{UB}, \pi_2^{LB}, a_1^*, o^*)) > 0$ then
9          explore$(\tau(b, \pi_1^{UB}, \pi_2^{LB}, a_1^*, o^*), \varepsilon, t+1)$
10         update(b)
```

The algorithm has guaranteed convergence:

Theorem 4.2 ([HBKK20])**.** *For any $\varepsilon > 0$ and $0 < D < (1-\gamma)\varepsilon/2\delta$, Algorithm 3 terminates with $V_{UB}^{\Upsilon}(b_0) - V_{LB}^{\Gamma}(b_0) \leq \varepsilon$.*

4.2.8 Partitions

Although player 1 does not have perfect information about the game's current state, he generally has some sort of limited information. For example, in pursuit-evasion games we consider, the player always knows his position. Because of this, it is unnecessary to represent the uncertainty about his position in the belief state.

This allows us to split the states into disjoint information sets called partitions. These partitions group together states that player 1 cannot distinguish. Because he knows the current partition, we can replace the global belief state b, a probability distribution over all states $s \in S$, with smaller belief states for each partition. If we are in partition p, belief $b \in \Delta(S_p)$ is a probability distribution only over the states of partition p, denoted S_p. We can also deduce the next partition $p' = T(p, a_1, o)$ using only player 1's action a_1 and observation o.

This representation significantly improves the performance of the algorithm. It allows us to split the computation of the optimal value function V to individual partitions. Each partition holds its own representation of the lower bound and the upper bound in the form of sets Γ and Υ, respectively.

When computing the linear programs, we do not need to go through all game states. We can focus only on the immediate rewards of the current partition and the values in the partitions we might end up in. This dramatically reduces the size of the linear programs. The quantifications $s \in S$ can be replaced by $s \in S_p$, where S_p is set of states of the current partition p. Furthermore, since we can deduce the next partition from a_1 and o, we can replace the quantifications $(a_1, o, s') \in A_1 \times O \times S$ by $(a_1, o, s') \in A_1 \times O \times S_{p'}$, where $S_{p'}$ is the set of states of next partition $p' = T(p, a_1, o)$.

Chapter 5

Approximation methods

Performance and scalability are the main drawbacks of HSVI for OS-POSGs algorithm. In its current state, the algorithm is inefficient and cannot be applied to larger games. This chapter introduces two main contributions of our work that try to solve the performance issue: approximate computation of the upper bound value function using neural networks and approximate stage game solution using quantal response equilibrium.

The neural network substitutes the linear program for computing the value of V_{UB}^{Υ} and the quantal response equilibrium is used inplace of the minimax and maximin linear programs that solve the stage games of HSVI. Together these changes completely eliminate the usage of linear programming from the exploration phase of the algorithm (it might still be used for bounds initialization). Constructing and solving these linear programs is a computationally heavy task that slows the original algorithm.

5.1 Upper bound approximation

We start with the approximate computation of the upper bound value function V_{UB}^{Υ}. The computation and querying of V_{UB}^{Υ} is the main performance bottleneck of HSVI algorithm.

5.1.1 Original representation

As described in Section 3.3.1, the upper bound is represented by a finite set Υ of belief-value pairs (b, v). The upper bound value in an arbitrary belief b is computed as a projection onto the lower part of Υ's convex hull. This requires solving a linear program corresponding to Equation (3.13). The size of this linear program depends on the dimensionality of the belief space (i.e., number of states) and also on the number of (b, v) pairs in set Υ.

The querying of $V_{UB}^{\Upsilon}(b)$ is very frequent in the algorithm. It is not computed directly using the linear program from Equation (3.13). Instead, it is part of a larger linear program - the dual formulation for solving a stage game using values from V_{UB}^{Υ} (Equation (4.8) with replacement from Equation (4.10)). Stage game is being solved two times at each depth - first, on the way down the search tree and once more on the way up.

Because the bounds are updated after each stage game, upper bound changes very frequently in different belief states. It is thus not possible to cache the computed values between individual stage games.

5.1.2 Available alternatives

All these issues mentioned in the previous section call for a faster approximative representation of the upper bound value function. Some alternatives to the convex hull approach were introduced for POMDPs and can be also applied to our problem.

The RTDP-Bel algorithm, presented in [GB98], tries to discretize the belief space into regions with the same value. This makes the upper bound value function behave like a piecewise constant function.

Another approach, presented in [Hau00, SS12a], approximates upper bound using "sawtooth"-shaped function. This method approximates the convex hull computation by computing a set of simpler convex hulls. For each interior point (b, v) from Υ, the convex hull of this point and the points belonging to the vertices of the belief simplex $\Delta(S)$ is added to the set. The upper bound value in an arbitrary belief b is then computed as a minimal projection of b onto this set.

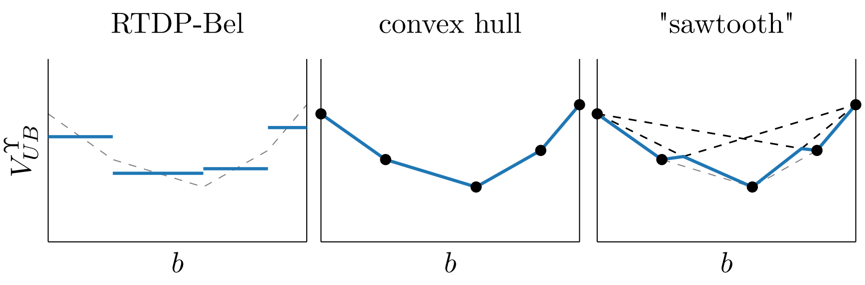

Figure 5.1: Comparison of upper bound value function representations

None of these methods seem like a suitable replacement for the convex hull method. Although they are fast, they are not very accurate approximations. The comparison of the original convex hull approach with the mentioned alternatives can be seen in Figure 5.1.

One other approach, presented in [Ša19], is trying to solve the upper bound approximation issue specifically for HSVI for OS-POSGs algorithm. It uses the Approximate convex hull algorithm to identify extreme points (i.e., ones that are crucial for the convex hull) of Υ. Once the subset of extreme points is identified, the upper bound can be approximated using only this subset, and nonextreme points can be pruned from Υ. However, this algorithm uses quadratic programs to identify the extreme points, which is a computationally demanding task. It has been shown that the performance gain from smaller set Υ is not enough to compensate for this and the modification actually worsens the performance of the algorithm.

5.1.3 Approximation using neural network

Because we are using partitions to divide the belief space, the domains of individual partitions' value functions are significantly smaller. The value function can be simply seen as a real-valued function in a $|S_p|$-dimensional space $V : |S_p| \mapsto \mathbb{R}$, where S_p is the set of partition p's states. By definition, it also has a convenient shape - it is a piecewise linear and convex (PWLC) function. On top of that, it is also δ-*Lipschitz* continuous.

All these properties make the value function, and also its upper bound V_{UB}^{Υ}, easy to interpolate. To make the querying of the V_{UB}^{Υ} fast and at the same time reasonably precise, we will use an artificial neural network to approximate its value, i.e., the lower convex hull of Υ. This shifts the primary computation time from querying to updating because the neural network will have to be retrained after each point-based update of upper bound. On the other hand, the querying will be almost instant (only one pass of inputs through the neural network).

Artificial neural networks are capable of solving large and complex problems in various fields of artificial intelligence. However, we will use the term in its purest form - as a simple approximation of a $|S_p|$-dimensional real-valued function. Although specific neural network architectures exist that focus on approximating convex functions (introduced for example in [AXK17]), there was no need for them so far in our task. Nevertheless, they may considered in future work as a room for improvement.

A simple neural network composed of densely connected layers, with two or three hidden ones, and tens of units should be enough to approximate the upper bound sufficiently. The neural network will be trained on the belief-value pairs from Υ and will be capable of interpolating the points in between them.

5.1.4 Pruning

The V_{UB}^{Υ} is composed of belief-value pairs that represent the value in given belief. New pairs are added to the set Υ after each update and the bound improves. This means the size of Υ grows but not all of its elements are contributing to the value of V_{UB}^{Υ}. These elements are said to be dominated and can be pruned from Υ.

Similarly, Γ also contains some dominated α-vectors. Pruning can be used, in general, for both bounds to reduce the size of sets Γ and Υ and the improve the performance of the algorithm. However, leaving dominated elements in the sets does not spoil the correctness of the original HSVI algorithm.

On the other hand, the pruning of upper bound is crucial for the approximation using the neural network. If, for example, we would leave in Υ two belief-value pairs with the same belief b but different values, an accurate approximation using the neural network would be impossible. In this situation, the neural network would, in general, learn to return mean of the two values. However, the correct value is the lower one from those two. This makes the pruning of Υ a must for the neural network approximation.

There are many ways how can we determine which points in Υ are dominated and which are not. However, the pruning must not be computationally demanding, otherwise the performance gain from the approximation would be diminished. Therefore, we chose a simple method that, for each point from Υ, prunes its neighborhood of a predefined size ε_{prune} induced by the maximum norm $\|\cdot\|_\infty$ (i.e., a hypercube).

Before adding a new belief-value pair (b, v) into Υ, the ε_{prune}-neighborhood of b is checked. If

$$\exists (b_i, v_i) \in \Upsilon : \|b_i - b\|_\infty < \varepsilon_{prune} \wedge v_i \geq v \,, \tag{5.1}$$

then the new pair (b, v) is not added to Υ. If, for some pair $(b_i, v_i) \in \Upsilon$, $\|b_i - b\|_\infty < \varepsilon_{prune}$ but the new value is larger, then the new pair (b, v) is added and the old one (b_i, v_i) is removed. If no point from Υ shares neighborhood with the new one, it is also added. This procedure makes sure that the situation of having two pairs with the same belief and different values cannot occur.

5.1.5 Implementation details

Each partition p has its own neural network to approximate V_{UB}^{Υ} in its belief space $\Delta(S_p)$. This neural network is composed of densely connected layers with a small number of units and a sigmoid nonlinear activation function σ. The input layer has $|S_p|$ units and it takes a belief $b \in \Delta(S_p)$ as an input. The last dense layer functions as an output layer and consists only of one unit with no activation function. The output of the neural network is approximation of upper bound value in belief b, i.e., $V_{UB}^{\Upsilon}(b)$.

The neural network is trained on batches sampled uniformly with replacement from the set Υ. Beliefs from Υ are the training inputs and the values are the gold data. Mean squared error (MSE) is used as a loss function, and it is minimized by the Adam optimizer. We do not train for a fixed number of epochs. Instead, the training continues until the MSE evaluated on the Υ is lower than a predefined target loss. The training runs after the upper bound initialization and also after each point-based update (assuming the Υ has changed, see the previous Section 5.1.4).

Various hyperparameters' settings are discussed and evaluated in the Experimental evaluation Chapter 6.

5.1.6 Method drawback

One main drawback of this approach is that the function learned by the neural network does not have to be convex. If there are not enough points in the Υ, the resulting approximation might not follow the convex hull precisely. It may happen that the neural network will learn to include points that do no longer belong to the convex hull of Υ but were not removed by the simple pruning method.

The situation is demonstrated in one-dimensional belief space in Figure 5.2. The convex hull of upper bound is shown in blue and its neural network approximation is shown in orange. The individual beliefs of Υ are far away from each other, so they do not share an ε_{prune}-neighborhood. This means that no point is pruned from the Υ. The point in the middle is used for the training of the neural network, even though it does not belong to the convex hull of Υ (i.e., it is dominated). The gray area shows where the neural network approximation deviates from the convex hull. The difference is most noticeable just in the close neighborhood of the dominated point.

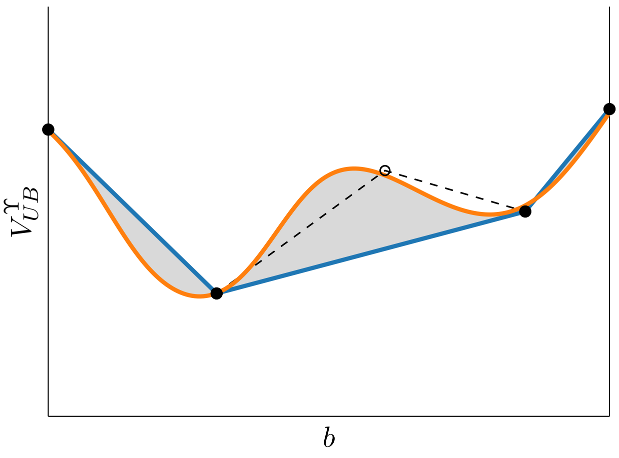

Figure 5.2: Neural network approximating convex hull imprecisely

However, we do not believe that this issue would make this approximation method unusable. It might slow the convergence, but it should not skew the results of the algorithm. Given how the heuristic of HSVI algorithm works, the exploration should be navigated to such problematic beliefs because they will have, in general, a larger excess gap. This means that such concave regions will be smoothed over time, and the shape of the approximation will be corrected.

5.2 Stage game equilibrium approximation

Our second contribution and modification of HSVI algorithm is approximating stage game solution using a bounded-rational equilibrium. The modification of stage game solution goes hand-in-hand with the upper bound approximation introduced in previous section. Integrating these changes together is easier and might even give better results.

5.2.1 Problem of using neural network with minimax

Solving a stage game in the original HSVI for OS-POSGs (i.e., making a Bellman update H) is done by finding the minimax and maximin strategies introduced in Section 2.2. These strategies correspond to a Nash equilibrium and are found using linear programming (Section 4.2.3). Because these strategies are computed as exact best responses, they might be a strict optima for the given values of bounds.

By using a neural network to approximate the upper bound, the upper bound is no longer guaranteed to have the properties of a value function. The approximate value function obtained from the neural network is not piecewise linear and convex, in general. Moreover, it does not have to be δ-*Lipschitz* continuous. We must consider that, in general, value in an arbitrary belief is imprecise by a nonnegligible value.

This prevents us from using exact Nash equilibrium computed using linear programming because it relies on the bounds to be exact. A small imprecision in the upper bound value approximation might cause great deviation of the best-response strategies. Because of this, we will use a different approach to stage game solving in HSVI, which will not be affected by the imprecise approximation of the upper bound.

5.2.2 Stage game quantal response equilibrium

The solution to this problem is to use a solution concept with bounded rationality for the stage games. We will use the quantal response equilibrium introduced in Section 2.3. It is a solution concept that uses estimations of utilities and strategies to find a bounded-rational equilibrium. Therefore, it will work even with the imprecise upper bound approximation. The lower bound stage game will be also solved by the QRE.

We will use the logit equilibrium parametric class, with $\sigma_i : X_i \mapsto \Pi(A_i)$ defined as

$$\sigma_{ij}(x_i) = \frac{e^{\lambda x_{ij}}}{\sum_{k=1}^{J_i} e^{\lambda x_{ik}}} \,. \tag{5.2}$$

This will allow us to control the outcome of stage game solution. The higher λ we use, the closer to Nash equilibrium will be the found strategies. By setting lower λ, we can make the resulting strategies more randomized. This can help us to introduce some nondeterminism to the exploration heuristic. This randomization can help the algorithm to unstuck from a dead end, caused by the imprecision in approximation.

Furthermore, we will be able to control the convergence of QRE to a fixed point by limiting the number of iterations or setting the desired convergence precision. This can be used to make a trade-off between performance and accuraccy.

5.2.3 Expected utility of an action

The implementation of logit equilibrium is not so straightforward in our case. Two major obstacles make the adaptation of QRE to HSVI stage game a harder task than applying it to normal-form game.

First, the strategy of player 2 is conditioned by the true state (which player 2 knows). While player 1 has strategy π_1 for the belief state $b \in \Delta(S_p)$, player 2 has a different strategy $\pi_2(\cdot \mid s)$ for each state $s \in S_p$.

The second obstacle is that the computation of expected utility $\overline{u}$ of some action given opponent's strategy is much more complex. The expected utility of an action is composed of the immediate reward and the discounted value of subsequent beliefs. Both of these values must be further weighted by the belief and the probability of opponent's action. The subsequent values are also weighted by the transition's probability.

We will use $\overline{u}_{1,a_1}(\pi_2)$ to denote the expected utility of player 1's action a_1 against player 2's strategy π_2. $\overline{u}_{2,a_2}(\pi_1 \mid s)$ denotes the expected utility of player 2 playing action a_2 in state s against strategy π_1 of player 1.

The immediate reward part of $\overline{u}$ can be computed as

$$\overline{u}^{imm}_{1,a_1}(\pi_2) = \sum_{(s,a_2) \in S_p \times A_2} b(s)\pi_2(a_2 \mid s)R(s,a_1,a_2) \tag{5.3a}$$

$$\overline{u}^{imm}_{2,a_2}(\pi_1 \mid s) = -\sum_{a_1 \in A_1} \pi_1(a_1)R(s,a_1,a_2)\,. \tag{5.3b}$$

The part of $\overline{u}$ with subsequent values is more complex. As mentioned in Section 4.2.8, given a partition p, we can deduce the the next partition p' using action a_1 and observation o: $p' = T(p,a_1,o)$. Therefore, for each possible (a_1,o) pair, we compute the subsequent belief in partition p' and its value. Then we sum those values weighted by the probabilities that the game progresses in corresponding partition:

$$\overline{u}^{sub}_{1,a_1}(\pi_2) = \sum_{o \in O} \mathbb{P}_{b,a_1,\pi_2}[a_1,o]V(\tau(b,a_1,\pi_2,a_1,o)) \tag{5.4a}$$

$$\overline{u}^{sub}_{2,a_2}(\pi_1 \mid s) = -\sum_{(a_1,o) \in A_1 \times O} \mathbb{P}_{s,\pi_1,a_2}[a_1,o]V(\tau(s,\pi_1,a_2,a_1,o))\,, \tag{5.4b}$$

where

$$\mathbb{P}_{b,a_1,\pi_2}[a_1,o] = \sum_{(s,a_2,s') \in S_p \times A_2 \times S_{p'}} b(s)\pi_2(a_2 \mid s)T(o,s' \mid s,a_1,a_2) \tag{5.5a}$$

$$\mathbb{P}_{s,\pi_1,a_2}[a_1,o] = \sum_{s' \in S_{p'}} \pi_1(a_1)T(o,s' \mid s,a_1,a_2) \tag{5.5b}$$

and

$$\tau(b,a_1,\pi_2,a_1,o)(s') = \\ = \frac{1}{\mathbb{P}_{b,a_1,\pi_2}[a_1,o]} \sum_{(s,a_2) \in S_p \times A_2} b(s)\pi_2(a_2 \mid s)T(o,s' \mid s,a_1,a_2) \tag{5.6a}$$

$$\tau(s,\pi_1,a_2,a_1,o)(s') = \frac{1}{\mathbb{P}_{s,\pi_1,a_2}[a_1,o]}\pi_1(a_1)T(o,s' \mid s,a_1,a_2)\,. \tag{5.6b}$$

These equations are very similar to Equations (4.5), (4.2) and (4.3). However, the probabilities $\pi_1(a_1)$ for player 1 and $b(s)\pi_2(a_2 \mid s)$ for player 2 were left out. This can be done because they are equal to 1 due to the fact that we compute $\overline{u}_{1,a_1}(\pi_2)$ for specific a_1 and $\overline{u}_{2,s,a_2}(\pi_1)$ for specific s and s_2.

The expected utilities $\overline{u}$ are then simply

$$\overline{u}_{1,a_1}(\pi_2) = \overline{u}^{imm}_{1,a_1}(\pi_2) + \gamma\overline{u}^{sub}_{1,a_1}(\pi_2) \tag{5.7a}$$

$$\overline{u}_{2,a_2}(\pi_1 \mid s) = \overline{u}^{imm}_{2,a_2}(\pi_1 \mid s) + \gamma\overline{u}^{sub}_{2,a_2}(\pi_1 \mid s)\,. \tag{5.7b}$$

5.2.4 Implementation details

Having defined the expected utility $\overline{u}$ for OS-POSGs, we can compute stage game QRE similarly to how we did for normal-form games in Section 2.3.5.

The major difference is that the computation of player 2's strategy π_2 in the second step of the QRE iterative method must be done for each state separately. This is due to the fact that π_2 is a conditional probability and $\pi_2(\cdot \mid s)$ for each $s \in S_p$ must be a probability distribution.

$$x^{(t)}_{2,a_2}(s) = \overline{u}_{2,a_2}(\pi_1^{(t-1)} \mid s) \qquad \forall s \in S_p \tag{5.8a}$$

$$\pi_2'^{(t)}(a_2 \mid s) = \frac{e^{\lambda x^{(t)}_{2,a_2}(s)}}{\sum_{a_2 \in A_2} e^{\lambda x^{(t)}_{2,a_2}(s)}} \qquad \forall s \in S_p \tag{5.8b}$$

$$\pi_2^{(t)}(a_2 \mid s) = \frac{t \cdot \pi_2^{(t-1)}(a_2 \mid s) + \pi_2'^{(t)}(a_2 \mid s)}{t+1} \qquad \forall s \in S_p \tag{5.8c}$$

The convergence to a fixed-point of strategies is tested using a small precision constant ε_{qre} (e.g. 10^{-3}). Strategies are considered equal if their maximum norm $\| \cdot \|_\infty$ is smaller than ε_{qre}. The strategies of player 2 are tested for each state separately.

This iterative method is used for both lower bound V^{Γ}_{LB} and upper bound V^{Υ}_{UB}. For the upper bound point-based update, the value of stage game $HV^{\Upsilon}_{UB}(b)$ corresponding to the QRE strategies is computed using the Bellman Equation (4.4). The computation of corresponding α-vector for lower bound point-based update is based on the Bellman equation. Since the α-vector is defined by its values in the vertices of the belief simplex $\Delta(S_p)$, we can construct it by computing

$$V_{\pi_1,\pi_2}(s) = \mathbb{E}_{s,\pi_1,\pi_2}[R(s,a_1,a_2)] + \gamma \sum_{(a_1,o)\in A_1 \times O} \mathbb{P}_{s,\pi_1,\pi_2}[a_1,o]V(\tau(s,\pi_1,\pi_2,a_1,o)) \tag{5.9}$$

for each state $s \in S_p$.

5.2.5 Modification of exploration heuristic

The original algorithm has guaranteed convergence (see Theorem 4.2). Because of imprecision, caused both by upper bound approximation and stage game equilibrium approximation, this guarantee no longer holds. The modified algorithm can (especially in advanced stages of the algorithm) reach a point, where it stagnates and stops converging. In such situation, the exploration heuristic navigates the algorithm in such a way that the explorations made do not improve the bounds.

We believe that by introducing some nondeterminism to the exploration heuristic, we can avoid such convergence plateaus. We do this by modifying the original exploration heuristic from Equation (4.18). Instead of choosing (a_1^*, o^*) as the action and observation, for which the weighted excess gap at next stage is the largest, we will exploit a property of QRE strategies.

The QRE strategies correspond to a bounded-rational equilibrium and are computed as cumulative moving average, starting from uniform strategy. Because of this, QRE strategy asssigns a nonzero probability to each action. We use this to sample the action a_1^* randomly according to categorical probability distribution induced by player 1's QRE strategy π_1^{UB} from upper bound stage game. The observation o^* is then selected according to the weighted excess gap.

$$a_1^* \leftarrow p(a_1^* = a_1) = \pi_1^{UB}(a_1) \tag{5.10a}$$

$$o_1^* \leftarrow \arg\max_{o \in O} \mathbb{P}_{b,\pi_1^{UB},\pi_2^{LB}}[a_1^*, o] \cdot \text{excess}_{t+1}(\tau(b, \pi_1^{UB}, \pi_2^{LB}, a_1^*, o)) \tag{5.10b}$$

Using this modified exploration heuristic, each action has a nontrivial probability of being selected for next stage.

Chapter 6

Experimental evaluation

We have described the original HSVI for OS-POSGs algorithm thoroughly and also introduced our modifications of the algorithm. In this chapter, we are going to compare the original algorithm with our modified version on number of experimental games. The methods we used to modify the algorithm require setting some number of parameters. Therefore, we will also try to find out, which parameters work the best for the upper bound neural network approximation and also the stage game quantal response equilibrium.

6.1 Games

The class of OS-POSGs can be used to model many real life security and cybersecurity problems. Therefore, we will test and compare versions of the HSVI algorithm on such problems. All games presented were taken from [HBP17].

6.1.1 Representation

The games are defined using Definition 4.1 with some minor alterations. First, for each state, a partition, to which the state belongs, is given. This allows us to split the belief space into smaller parts and improve dramatically the performance of the algorithm. Furthermore, each game has initial partition p_0 and initial belief b_0 specified. Because we solve the games with infinite horizon, we also need a discount factor $\gamma \in (0, 1)$.

We also limit the available actions a player can take according to current state of the game. For each state, we define a subset of actions available to player 2. Player 1 has a subset of available actions specified for each partition. This can be done by defining the transition function T only for combinations of (s, a_1, a_2) that are possible given these subsets. Similarly, the reward function R is specified only for possible combinations.

Each game can be described using number of states $|S|$, number of partitions $|P|$, number of player 1's actions $|A_1|$, number of player 2's actions $|A_2|$, number of observations $|O|$, number of transitions $|T|$, number of rewards $|R|$ and discount factor γ.

6.1.2 Pursuit-evasion games

Our main set of games for evaluating the performance of the algorithm consists of 6 pursuit-evasion games. In pursuit-evasion games, agents navigate in a grid environment, being able to move into adjacent cells. However, moves are not defined for every pair of adjacent cells.

A team of pursuers, controlled by player 1, is trying to find and capture the single evader, which is controlled by player 2. The evader has a full information about the state of the game, while the pursuers do not know the location of the evader. The game ends when the evader is located on the same cell as one of the pursuers (i.e., he is captured). In such case, player 1 receives utility of 95 and player 2 corresponding penalty of -95. Other transitions have zero utility for both players.

game	$\|S\|$	$\|P\|$	$\|A_1\|$	$\|A_2\|$	$\|O\|$	$\|T\|$	$\|R\|$	γ
peg03.posg	143	21	145	13	2	2 671	2 671	0.95
peg04.posg	363	37	290	18	2	8 123	8 123	0.95
peg05.posg	731	57	485	23	2	18 335	18 335	0.95
peg06.posg	1 299	82	730	28	2	34 807	34 807	0.95
peg07.posg	2 093	111	1 025	33	2	59 039	59 039	0.95
peg08.posg	3 171	145	1 370	38	2	92 531	92 531	0.95

Table 6.1: Set of pursuit-evasion games

The descriptions of the 6 pursuit-evasion games of gradually increasing sizes are shown in Table 6.1. We consider only games where the team of pursuers has 2 members. The number in the name of pursuit-evasion games corresponds to the width of the grid. For all games presented, the grid is 3 cells high. An initial grid environment for the peg05.posg game is show in Figure 6.1. Pursuers always start in bottom-left corner and the evader starts in the top-right corner.

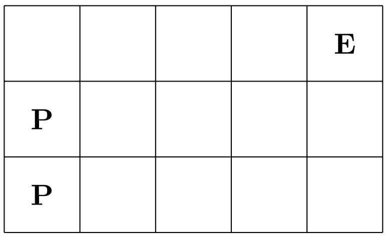

Figure 6.1: Initial state of the peg05.posg game

6.2 Implementation

The original HSVI for OS-POSGs algorithm from [HBP17] was written in C++. The implementation was heavily optimized and very hard to extend or modify. Furthermore, C++ does not have a suitable user-friendly machine learning library that was needed for the upper bound approximation method. Therefore, we decided to rewrite the algorithm on our own in more higher-level language with better package ecosystem.

6.2.1 Julia

Julia [BEKS17] was chosen as the language of choice for our new implementation. Although Julia is relatively young language, it has great ecosystem of packages and it is easy to work with, while still being highly performant.

First we have rewritten the original algorithm, including the minimax and maximin linear programs for HSVI stage game, bounds representation, exploration heuristic and the described bounds' initialization methods. We have also written our own parser for the game definition files.

Then we added our contributions to the module: the approximation of upper bound using neural network and the approximation of HSVI stage game using quantal response equilibrium.

6.2.2 Libraries

In our implementation, we utilized couple of libraries from the Julia package ecosystem.

First, we needed a performant library that would be able to solve stage game linear programs from the original algorithm. For this, we used JuMP [DHL17], a modeling language for mathematical optimization that supports many backend solvers. We chose the commercial CPLEX [IBM20] as the solver of linear programs, which was tied together with JuMP using CPLEX.jl wrapper package.

Furthermore, we needed a machine learning library capable of constructing and training neural networks for upper bound approximation. Flux [ISF$^+$18] was used for this purpose. It provides all the machine learning tools we needed.

6.3 Parameters

The original HSVI algorithm has several parameters that influence its behavior. Both our modifications also introduced some new parameters that have impact on the results. Some of them are required to be set to reasonably in order for the algorithm to converge to correct results and do not have much impact on the speed. Other parameters are used to make a trade-off between performance and precision.

6.3.1 Shared parameters

Some parameters are shared by both the original HSVI and our modified version. These come mostly from the original algorithm and methods used for bounds initialization. We set them to the same value for all algorithm configurations.

ε_{target} - target precision of the algorithm

The desired precision with which we would like to solve the game. It is also used for the excess gap calculations. We used $\varepsilon_{target} = 0.01$ for all experiments. Because we use normalization of utilities (see normalize utilities below) for every game, the resulting values of those games are relatively low. Therefore, the required precision of 0.01 seemed as a reasonable goal that would be achievable for most of the games.

$\delta_{presolve}$ - initialization convergence delta

Initialization methods of both upper bound and lower bound are iterative algorithms. The initialization phase is terminated when change of value in all states is smaller than $\delta_{presolve}$ between following iteartions. The used initialization methods are quite simple and fast. Because of this, we can use rather small value for $\delta_{presolve}$. We used $\delta_{presolve} = 0.0001$ for both upper bound and lower bound initialization.

$T_{presolve}$ - initialization time limit

The initialization phase might not convergence to the desired precision or it might take a lot of iterations. $T_{presolve}$ is used to make sure that the initialization of bounds does not last too long. We used time limit of 5 min for both bounds individually. This is still reasonably small value compared to the running time of the whole algorithm and the starting position plays large role in the performance of the algorithm.

R_{norm} - normalize utilities

The codomain of reward function $R(s, a_1, a_2)$ is not constrained in any way. This means that transitions can have arbitrary utility values. Therefore, we normalized the utilities of all games so that they all fall into interval $[0, 1]$.

D - neighborhood parameter

The neighborhood parameter D ensures that the sequence of widths $\rho(t)$ is monotonically increasing and unbounded. It is required to be between values 0 and $\frac{(1-\gamma)\varepsilon}{2\delta}$, in order for the convergence guarantees of the original algorithm to hold. The value of neighborhood parameter D is more or less given by its definition. We set $D = 10^{-6}$ for all games.

6.3.2 Neural network parameters

Our first modification to the original algorithm has five parameteres. They define the shape of neural networks for upper bound approximation and also the way we are training them.

Some of the parameters of the neural networks were set to the same values across the different configurations. The choice of values for these parameters was based on the values generally used in the machine learning community.

B - batch size

The size of the uniformly sampled set of Υ points on which the neural network is trained during one epoch. We did not see much effect when changing values of batch size and set it to 128 for all configurations.

η - learning rate

The learning rate used for the Adam optimizer during gradient descent. We tried couple of values for the learning rate of Adam optimizer. For value of 0.01 the training of neural networks seemed to be faster than when using Adam's default value of 0.001.

Below is the rest of the neural network upper bound approximation parameters. These were used to create different configurations of the algorithm.

MSE_{target} - target MSE loss

The target mean squared error loss of the neural network on Υ. We do not train the neural networks for fixed number of epochs. Instead, we train them until they reach the desired loss. Setting small target loss may extend significantly the learning time and in some situations extremely small target loss might not be achievable. We used 0.0001 as a default and also tried values of 0.01 and 10^{-6}.

NN_{layers} - neural network layers' configuration

The number of neural network's dense layers and their units. We used only densely connected layers with sigmoid activation function σ. We tested two configurations with two hidden layers and one with three hidden layers (32-16, 16-8 or 32-16-8). The 32-16 configuration was used as a default.

The belief space of partitions is not large. If we divide the number of states by the number of partitions, we can see that the typical dimensionality of partition's belief space ranges roughly from 7 for peg03.posg to 22 for peg08.posg. Therefore, the configuration of the neural network does not have to be much complicated. Similarly to target loss, setting small number of layers and units might make the neural network unable to learn the upper bound shape.

ε_{prune} - upper bound pruning precision

Precision used during the upper bound pruning when searching for close beliefs. The neighborhood is induced by the maximum norm, i.e., $\|\cdot\|_\infty < \varepsilon_{prune}$. By setting different values for the upper bound pruning neighborhood, we can control the precision of the approximation and also the size of Υ. Setting higher values should prune Υ more dramatically. We started with default value of 0.01 and also experimented with 0.1 and 0.001.

6.3.3 Quantal response equilibrium parameters

Our modification to the stage game solving phase of HSVI also has parameters that affect its behavior. They control how close will be the bounded-rational equilibrium to the strict Nash equilibrium.

λ_{qre} - logit equilibrium parameter

Logit equilibrium parameter that controls the level of error. The higher λ is set, the closer is computed LQRE to the exact Nash equilibrium. However, setting it to large values might cause the logit equilibrium computation to overflow. We used value 100 as a default and also tried values of 10 and 500.

ε_{qre} and T_{qre} - quantal response equilibrium convergence parameters

These parameters control how many iterations it takes to compute the quantal response equilibrium. We change them together, because achieving better precision takes more iterations. By setting these parameters we control the trade-off between performance and precision of the equilibrium approximation. We tried values 0.1, 0.01 and 0.001 for ε_{qre} and 10, 100 and 1000 for T_{qre}, respectively. The $\epsilon_{qre} = 0.01$ and corresponding limit of 100 iterations were used as a default.

6.4 Experiments

In this section, we present the experiments we made to test our modifications to the HSVI algorithm. First, we try to find values for the mentioned parameters that work reasonably well. Then, we compare the performance with both C++ and Julia implementations of the original algorithm. We also test for robustness and scalability of the approximation methods.

6.4.1 Environment

Both implementations are single-threaded. Majority of experiments, with the exception of C++ ones, were run on MetaCentrum computational grid on nodes with Intel® Xeon® Gold 6130 CPU @ 2.10GHz. The C++ experiments were run on our own machine utilizing AMD Ryzen™ 5 1600 (AF) @ 3.20GHz.

We use the C++ implementation only for reference and also to make sure that our reimplementation of the original algorithm works correctly. Given that the implementations are written in different languages and the C++ one is heavily optimized, we do not make any conclusion from comparison of their running times.

The convergence of the algorithm tends to slow down dramatically when the gap between bounds approaches smaller values and single run might take hours for larger games. Because of this, we run the individual experiments with a limit of one hour. If the algorithm was not able to solve the game with desired precision under one hour, we terminate it and record the achieved gap.

6.4.2 Recorded values

To be able to compare the algorithms and their configurations, we record several values as a result of each algorithm run. The most important values are the values of lower bound and upper bound in initial belief, i.e., $V_{LB}^{\Gamma}(b_0)$ and $V_{UB}^{\Upsilon}(b_0)$. Together these values also give us the final excess gap in initial belief $width(V(b_0))$, which tells us if the algorithm reached desired precision ε_{target}. We are also interested in the running time of the algorithm, which we use to evaluate its performance. If the algorithm reaches the one-hour time limit, we record the achieved excess gap after one hour.

Furthermore, we store sizes of sets Γ and Υ. Because we use partitions, the sizes are computed for each partition individually and then summed. These values can give us an comparison of algorithms in terms of computational memory consumption. They also tell us how effective is the pruning method we used. Smaller sets should yield faster algorithm, in general.

In addition to these final values recorded at the end of the run, we also store interim results after each exploration. We are interested in the same values as mentioned above, i.e., $V_{LB}^{\Gamma}(b_0)$, $V_{UB}^{\Upsilon}(b_0)$, $width(V(b_0))$, $|\Gamma|$ and $|\Upsilon|$. The depth of the exploration is also added to those values, because together with the total number of explorations it can be used to evaluate the exploration heuristic. With each set of interim results, we also include the timestamp at which the values were achieved. This can be used to see how the algorithm behaves in time.

6.4.3 Parameters evaluation

Our modifications to the HSVI algorithm have several parameters that have an impact on how the algorithm behaves. Because there is a larger number of them, running a grid search above them and testing all possible combinations is impossible. Therefore, we manually tested number of values for the parameters, observing the behavior of the algorithm, and defined a reasonable default value for each parameter. For some parameters, we also defined a set of possible alternative values. The choice of both default and alternative values is described in Section 6.3.

We then ran the experiments on all 6 pursuit-evasion games, changing values of these parameters one by one using the default parameters as a starting point. This gave us 11 different configurations. The individual configurations and the results of those experiments can be seen in Appendix A. From these 11 configurations, we have chosen three for further detailed evaluation, in which each configuration was tested multiple times on each game. The process of selecting the top three configurations is summarized below.

Some configurations had negative results and therefore were not considered for the detailed evaluation. They either converged much slower than other configurations or arrived at incorrect results. One of them is the $\textsc{Hsvi}_1$ configuration with $\lambda_{qre} = 10$, which had trouble converging even for the small games and the resulting value of the game was very imprecise. Other configuration that we dropped was $\textsc{Hsvi}_9$ with $MSE_{target} = 0.01$. This configuration was fast, but unable to learn the correct value of upper bound due to the small target loss. The $\textsc{Hsvi}_{10}$ configuration with $\varepsilon_{qre} = 0.1$ and $T_{qre} = 10$ also approximated the value imprecisely and therefore was not selected. The smaller number of iterations and high convergence constant made the QRE iterative method not convergence to reasonable strategies.

Setting the target loss MSE_{target} to 10^{-6} in $\textsc{Hsvi}_3$ configuration did not make the approximation significantly more precision than other configurations. For larger games, the configuration was not fast enough because it had to train for a longer time to achieve the small target loss. Therefore, we did not use it for the detailed evaluation either. Similarly, the $\textsc{Hsvi}_8$ with one added layer was a bit slower. Its value approximation, however, was not more precise and so it was not used either. Increasing the λ_{qre} to 500 in $\textsc{Hsvi}_{11}$ configuration made the resulting approximated value significantly higher than the correct one in all games. For that reason, we did not include it in further experiments.

All other configurations had good results. Some of them behaved very similarly, so we choose only three of them for the detailed experimental evaluation. The first variant, $\textsc{Hsvi}_{6=\mathrm{A}}$, is the one with default parameters. The second variant, $\textsc{Hsvi}_{2=\mathrm{B}}$, has $\varepsilon_{qre} = 0.001$ and $Tqre = 1000$. It was slower on all games compared to the default configuration. However, on games, where it managed to converge in time limit, it approximated the value of the game more precisely. The third and last variant we considered, $\textsc{Hsvi}_{7=\mathrm{C}}$, has ε_{prune} set to 0.1. This makes the pruning of the upper bound Υ more aggressive. As a result, $|\Upsilon|$ is smaller and the upper bound might be approximated in a slightly different way.

6.4.4 Robustness, performance and scalability evaluation

Having defined a smaller set of configurations to run, we were able to test them more thoroughly. Because the modifications we made are nondeterministic, each run on the same game is different (assuming different seed for the random number generator). Especially the modification of the exploration heuristic, in which the action of player 1 for the next stage is sampled randomly according to QRE strategy π_1^{UB}, introduces a lot of randomization to the algorithm.

To evaluate the robustness of the modified algorithm, we have run each of the three configurations 10 times for each game. The values reported in tables below are then computed as a mean over these 10 runs. We also include the standard error of the mean (SEM) for each value. The values acquired from the original algorithm for both C++ and Julia implementation are also shown in the results for comparison.

peg03.posg

The smallest of the pursuit-evasion games takes only tens of seconds to solve. The first important observation to mention is that our reimplementation of the original algorithm in Julia works correctly. Even though it takes more time to reach the desired precision, the final computed values of bounds are almost the same as for the C++ implementation. This also applies to larger games, if the algorithm manages to converge under one hour.

	time[s]	V_{LB}^{Γ}	V_{UB}^{Υ}	width	$\lvert\Gamma\rvert$	$\lvert\Upsilon\rvert$	expl.
HSVI$_{\text{C++}}$	5	0.8319	0.8410	0.0092	189	322	22
HSVI$_{\text{Julia}}$	33	0.8317	0.8404	0.0087	529	651	12
HSVI$_{\text{A}}$	37	0.8254	0.8316	0.0062	356	300	10
	±2	±0.0008	±0.0012	±0.0010	±32	±14	±2
HSVI$_{\text{B}}$	55	0.8298	0.8363	0.0065	337	284	12
	±3	±0.0010	±0.0016	±0.0011	±27	±12	±2
HSVI$_{\text{C}}$	38	0.8249	0.8322	0.0073	394	249	41
	±2	±0.0011	±0.0014	±0.0007	±78	±11	±29

Table 6.2: peg03.posg results

After the initialization of bounds, the neural networks of individual partitions have to learn the shape of the upper bound. For smaller game, such as peg03.posg, this stage may took larger part of the whole running time of the algorithm. This means that the original algorithm, which starts the explorations right after the initialization, has a slight head start over the modified algorithm. This may be the reason why the original algorithm converges faster for this small game.

We can see that all three variants of our modified algorithm returned values very close to the ones of the original algorithm. It is important to mention that the exact value of the game can be anywhere in the interval determined by the bounds of the original HSVI. Variant HSVI$_{\text{B}}$ took little more time to arrive at the target precision, but also produced better results than the other alternatives. This is probably given by the fact that the QRE computation for individual stage games takes more iterations to converge, but produces better values and strategies. Also, the standard error of value for both bounds is relatively small for all variants. This means that the modifications are robust enough for this small game.

The larger number of explorations and also larger SEM of exploration count for variant HSVI$_{\text{C}}$ is caused by one single run that took 302 explorations to converge. It got stuck at small gap size of roughly 0.013 because of imprecision in the upper bound approximation. The algorithm was waiting for the randomized exploration heuristic to pick an action with small assigned probability to unstuck the upper bound approximation and progress further.

The size of Υ did not increase dramatically in those explorations because of the pruning method. On the other hand, we do not use pruning for Γ, so it accumulated 1020 α-vectors, which in turn caused the larger SEM in Γ.

peg04.posg

For slightly larger game, we can already see the speed up from our modifications. Even the slowest from the three variants is faster than the Julia implementation. The returned values of bounds in initial belief are again very close to the correct ones and standard error is also relatively small.

	time[s]	V_{LB}^{Γ}	V_{UB}^{Υ}	width	$\|\Gamma\|$	$\|\Upsilon\|$	expl.
$\text{HSVI}_{\text{C++}}$	30	0.7769	0.7868	0.0099	2 348	2 551	150
$\text{HSVI}_{\text{Julia}}$	318	0.7768	0.7867	0.0098	5 131	5 457	66
HSVI_{A}	146	0.7712	0.7787	0.0075	1 448	988	64
	±13	±0.0006	±0.0011	±0.0009	±218	±69	±29
HSVI_{B}	213	0.7778	0.7829	0.0051	797	728	16
	±9	±0.0009	±0.0016	±0.0014	±77	±36	±2
HSVI_{C}	140	0.7703	0.7774	0.0071	1 478	890	54
	±11	±0.0005	±0.0010	±0.0008	±163	±35	±16

Table 6.3: peg04.posg results

Similarly as with the smaller game, variant HSVI_{B} has the most precise results, but takes a bit longer to converge. It also makes smaller number of explorations compared to the other variants and as a result it also has smaller sets Γ and Υ. The small number of explorations is probably caused by the fact that the QRE computation for variant HSVI_{B} is more precise, because of the smaller ε_{qre}. This makes this variant shrink the gap by a larger value in each exploration compared to the alternatives (even if they take longer to compute). Therefore, less explorations is needed, in general, to converge.

The larger SEM for both the size of Υ and the number of explorations in variants HSVI_{A} and HSVI_{C} is again caused by one single run that deviated from the rest and took more explorations to converge.

peg05.posg

Here we can observe the impact of pruning on the sizes of Γ and Υ. The C++ implementation includes pruning methods for both lower bound and upper bound. We can see that even if it has almost three times the number of explorations compared to the Julia implementation, the sizes of lower bound and upper bound sets are almost half in size. This impacts dramatically the performance of the Julia implementation and it takes almost half an hour for it to converge. All our three variants are significantly faster compared to the Julia implementation of the original algorithm.

The results for variants HSVI_{A} and HSVI_{C} are again close to the correct one and the intervals even overlap a bit. Variant HSVI_{B} with $\varepsilon_{qre} = 0.001$

	time[s]	V_{LB}^{Γ}	V_{UB}^{Υ}	width	$\|\Gamma\|$	$\|\Upsilon\|$	expl.
HSVI$_{\text{C++}}$	153	0.7190	0.7289	0.0100	7 845	9 735	424
HSVI$_{\text{Julia}}$	1 740	0.7188	0.7288	0.0100	15 604	16 278	155
HSVI$_{\text{A}}$	309	0.7144	0.7220	0.0076	2 828	1 982	86
	±21	±0.0005	±0.0011	±0.0006	±272	±93	±24
HSVI$_{\text{B}}$	641	0.7214	0.7277	0.0064	2 104	1 674	57
	±56	±0.0014	±0.0012	±0.0008	±216	±84	±15
HSVI$_{\text{C}}$	315	0.7142	0.7224	0.0083	2 932	1 813	84
	±23	±0.0007	±0.0005	±0.0005	±187	±47	±13

Table 6.4: peg05.posg results

has very precise result. It took more than twice as long for it to converge compared to the alternative variants, but the whole interval returned fits completely into the one from C++ implementation.

Small SEM values for each method show again that the modifications are very robust.

peg06.posg

For this size of game, the Julia implementation failed to converge under one hour. It managed to get to gap of roughly 0.03. We can see that the value of lower bound is roughly the same as for the C++ implementation. So it is the upper bound, which did not fall fast enough, and caused the larger gap.

	time[s]	V_{LB}^{Γ}	V_{UB}^{Υ}	width	$\|\Gamma\|$	$\|\Upsilon\|$	expl.
HSVI$_{\text{C++}}$	1 130	0.6574	0.6674	0.0100	27 303	33 005	1 566
HSVI$_{\text{Julia}}$	3 617	0.6564	0.6859	0.0295	24 351	25 568	209
HSVI$_{\text{A}}$	1 293	0.6674	0.6771	0.0098	10 616	5 457	438
	±271	±0.0006	±0.0016	±0.0017	±1 992	±503	±185
HSVI$_{\text{B}}$	2 117	0.6739	0.6813	0.0075	6 087	3 940	174
	±175	±0.0006	±0.0009	±0.0007	±522	±187	±29
HSVI$_{\text{C}}$	1 548	0.6692	0.6794	0.0102	13 536	4 669	646
	±285	±0.0008	±0.0014	±0.0013	±1 931	±257	±179

Table 6.5: peg06.posg results

Variant HSVI$_{\text{A}}$ is close to the correct results as its lower bound has the same value as the upper bound of C++ implementation. Variant HSVI$_{\text{C}}$ performs slightly worse. We can see that the $\varepsilon_{prune} = 0.1$ of variant HSVI$_{\text{C}}$, indeed, prunes the upper bound more aggressively. Compared to variant HSVI$_{\text{A}}$, variant HSVI$_{\text{B}}$ has higher number of explorations and as an effect also larger $|\Gamma|$, but the size of Υ is smaller.

As opposed to smaller games, variant HSVI$_{\text{B}}$ has the most imprecise result. This trend continues also for larger games. This is an important observation. It looks like the variant HSVI$_{\text{B}}$ returns larger value of the game, in general, compared to variants HSVI_A and HSVI_B. We found out, that for smaller games, the modified algorithm tends to underestimate the value of the game and it caused the variant HSVI_B to be the most precise. For peg05.posg its result was, on average, almost the same as the correct one. However, for larger games, the trend reverses and the modified version underestimates the true value of the game. This causes variant HSVI_B to perform worse compared to the alternatives.

Both variants HSVI_A and HSVI_C had a single run that did not converge for peg06.posg. For variant HSVI_A, the run was terminated after one hour at gap of roughly 0.025. For variant HSVI_C, the run arrived at gap of roughly 0.02 in one hour.

The standard error of the running times of the modified algorithm is significantly larger compared to the smaller games. The fastest run among the variants of the modified algorithm was for variant HSVI_C and managed to converge in about 560 seconds. The fastest runs for both variants HSVI_A and HSVI_B took roughly 660 seconds to converge. This means that some of the runs of the modified algorithm were even faster than the C++ implementation of the original algorithm.

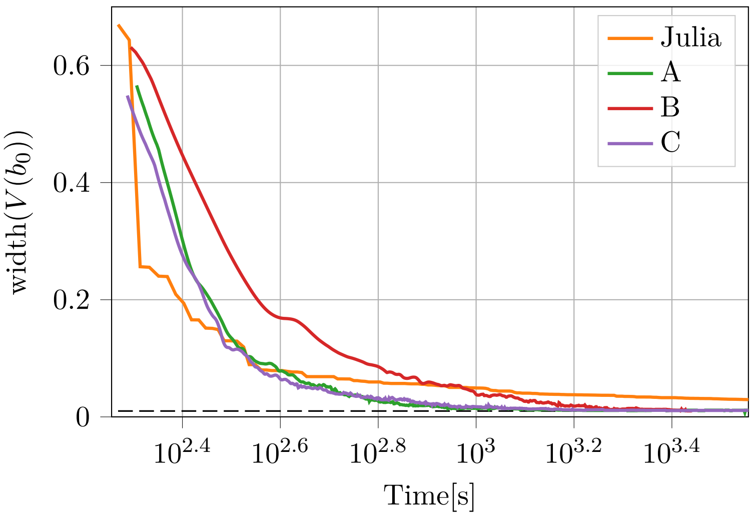

Figure 6.2: peg06.posg gap convergence graph

The convergence graph for peg06.posg is shown in Figure 6.2. The individual runs of the modified algorithm have different number of explorations and also record the exploration data at different timestamps. To be able to visualize the progress of gap in time, we had to combine them somehow. We interpolated the original time series to the same range and time steps using cubic splines. For runs of different lengths, the missing values after the convergence were padded by 0.01 (i.e., ε_{target}). While the convergence graph of the original algorithm is monotonically decreasing, this does not hold for the modified version. The retraining of the neural networks can cause the upper bound value to increase between following explorations.

We also do not show the convergence graph of C++ implementation. Its bounds' initialization phase is much faster than the one of the Julia implementation. The largest part of the gap is reduced in the early explorations and because we use logarithmic scale for the time axis, the interesting part of the C++ graph is shifted towards zero by a large amount and does not provide much insight.

We can see that the original algorithm has smaller gap in the early stages of the algorithm. However, the convergence slows down dramatically towards the end. The convergence of the modified algorithm also slows down, but not so fast. Variants $\textsc{Hsvi}_{\mathrm{A}}$ and $\textsc{Hsvi}_{\mathrm{C}}$ got ahead of the Julia implementation of the original algorithm at around 400 seconds, on average.

We can also observe the impact of more precise QRE computation in variant $\textsc{Hsvi}_{\mathrm{B}}$. It is behind the other variants in terms of convergence speed throughout the whole run. It got ahead of the original algorithm at around 1000 seconds, on average.

peg07.posg

Increasing the size of the game even further makes even the C++ implementation of the original algorithm not converge in time. Nevertheless, it managed to get very close to the target gap. The implementation of the original algorithm in Julia had final gap after one hour comparable to peg06.posg.

	time[s]	V_{LB}^{Γ}	V_{UB}^{Υ}	width	$\lvert\Gamma\rvert$	$\lvert\Upsilon\rvert$	expl.
$\textsc{Hsvi}_{\mathrm{C++}}$	3 602	0.6141	0.6253	0.0111	42 846	70 478	2 706
$\textsc{Hsvi}_{\mathrm{Julia}}$	3 613	0.6122	0.6443	0.0321	23 694	25 676	181
$\textsc{Hsvi}_{\mathrm{A}}$	1 977	0.6268	0.6367	0.0099	13 953	8 334	282
	±214	±0.0005	±0.0017	±0.0015	±1 101	±422	±41
$\textsc{Hsvi}_{\mathrm{B}}$	3 606	0.6302	0.6518	0.0217	8 281	5 891	149
	±2	±0.0008	±0.0020	±0.0020	±392	±161	±16
$\textsc{Hsvi}_{\mathrm{C}}$	3 441	0.6273	0.6426	0.0153	22 909	7 698	648
	±110	±0.0003	±0.0020	±0.0021	±509	±73	±27

Table 6.6: peg07.posg results

For a game of this size, all runs of variant $\textsc{Hsvi}_{\mathrm{C}}$ and most runs of variant $\textsc{Hsvi}_{\mathrm{B}}$ did not converge under one hour. This also causes the larger standard error of the final upper bound value. However, the gap achieved by those variants was smaller, in general, than the gap of original algorithm written in Julia and is still relatively close to the gap produced by C++ implementation.

9 out of 10 runs of variant $\textsc{Hsvi}_{\mathrm{A}}$ converged in time. The fastest took only 1250 seconds. In general, they returned values very close to the true values produced by C++ implementation.

As with previous game, we can observe the effect of the larger value of ε_{prune} parameter in variant $\textsc{Hsvi}_{\mathrm{C}}$. It has smaller size of Υ than variant $\textsc{Hsvi}_{\mathrm{A}}$ even though it has more than two times the number of explorations and also larger $|\Gamma|$.

peg08.posg

For the largest game, no run managed to converge under one hour. The C++ implementation reached gap of roughly 0.03. The Julia implementation of the original algorithm and also all variant of the modified algorithms reached much larger gaps.

	time[s]	V_{LB}^{Γ}	V_{UB}^{Υ}	width	$\lvert\Gamma\rvert$	$\lvert\Upsilon\rvert$	expl.
$\text{HSVI}_{\text{C++}}$	3 602	0.5621	0.5909	0.0288	27 245	77 551	2 895
$\text{HSVI}_{\text{Julia}}$	3 629	0.5544	0.6315	0.0770	22 861	25 887	162
HSVI_{A}	3 612	0.5797	0.6655	0.0858	21 112	12 925	219
	±2	±0.0011	±0.0101	±0.0101	±400	±201	±12
HSVI_{B}	3 617	0.5710	0.6560	0.0850	6 664	6 343	64
	±3	±0.0018	±0.0063	±0.0075	±281	±133	±5
HSVI_{C}	3 610	0.5786	0.6486	0.0700	21 098	9 338	234
	±1	±0.0009	±0.0050	±0.0053	±552	±101	±17

Table 6.7: peg08.posg results

The value intervals of all variants of the modified algorithm overlap with the true interval of C++ implementation. However, because the algorithms did not converge, this does not tell us much information about the precision of the modifications for game of this size. Terminating the individual runs before converging also caused larger standard error of the resulting values of upper bound.

Similarly to smaller games, variant HSVI_{B} has notably smaller number of explorations (and as a consequence also $|\Gamma|$ and $|\Upsilon|$) than variants HSVI_{A} and HSVI_{C}. Nevertheless, it managed to achieve, on average, the same width of gap as variant HSVI_{A}. Variant HSVI_{C} achieved the smallest gap, compared to the alterative variants, and also slightly smaller than the Julia implementation of the original algorithm.

Note that difference between the sizes of Υ for variants HSVI_{A} and HSVI_{C} is not so significant, compared to the smaller games. What we are observing here is probably the curse of dimensionality phenomenon. For larger games, the dimensionality of belief space of individual partitions increases and pruning using constant ε_{prune} does not have such an impact.

In Figure 6.3, we can see the convergence graph for peg08.posg. We can see that here, similarly as with peg06.posg, the Julia implementation of the original algorithm converges faster at the beginning, compared to the modified variants. However, as it slows down, the variants of the modified version of the algorithm eventually catch up. Variant HSVI_{C} even got to slightly smaller gap towards the end.

Variant HSVI_{B} with higher number of QRE iterations converges slower at beginning, compared to alternative variants HSVI_{A} and HSVI_{C}. Nevertheless, after one hour, it achieves gap of similar width as variant HSVI_{A}.

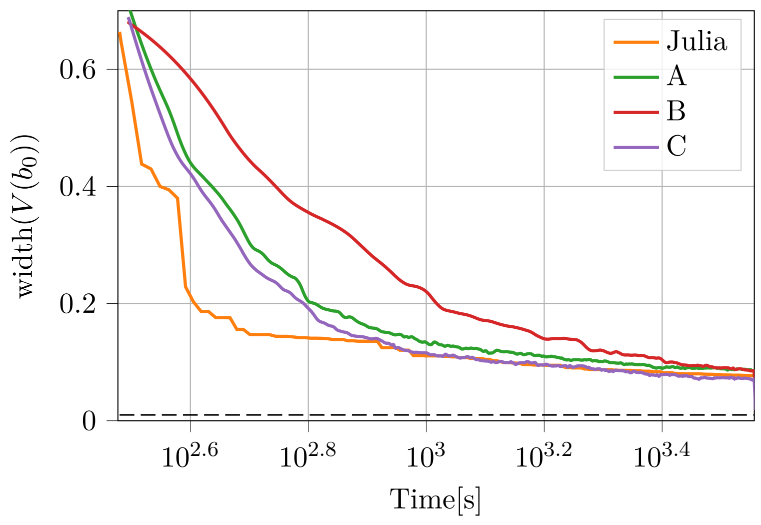

Figure 6.3: peg08.posg gap convergence graph

Scalability

The results presented above have told us a lot about the performance of the original algorithm and its modifications on individual games. By observing them, we can find out how fast and precise are the variants of the algorithm and also how much do the results deviate between individual runs.

However, as we mentioned, the main problem of the original HSVI algorithm is its application to larger games. It does not scale well enough. As the size of the game increases, the convergence of the algorithm slows down dramatically. To evaluate the scalability of our modifications, and compare them with the original algorithm, we need to evaluate the performance on all games as a whole sequence.

We show the evaluation in Figure 6.4. The figure contains both implementations of the original algorithm and also all three variants of our modified version. The x axis of the figure contains the 6 pursuit-evasion games sorted in order of increasing size of the game.

Upper part of the figure shows a plot of running time of the algorithm on individual games in a semi-logarithmic scale. The running times of the modified variants are computed as a mean over the 10 runs and the stripes denote the confidence interval in the form of standard error of the mean.

The running time of the algorithm, however, is limited to one hour. To be able to compare the variants even when this time limit is saturated, we include the bottom part of the figure. It denotes the excess gap by which the algorithm exceeded the desired precision of $\varepsilon_{prune} = 0.01$. For the variants HSVI_A, HSVI_B and HSVI_C, the bars represent the mean over the 10 runs. The errors bars indicate the standard error of the mean. We can see that for the runs, where the algorithm managed to converged under one hour, the excess gap is zero (i.e., width$(V(b_0)) \leq 0.01$).

The implementation of the original algorithm in Julia converged only for the first half of games. On the second half of games, it did not converge and the bar plot shows the excess gap it managed to achieve. The C++ implementation converged on four smaller games. On the fifth game, it did not converge under one hour, but the excess gap is very small. We can observe that the lines, denoting their running times on the first three games, have similar slope.

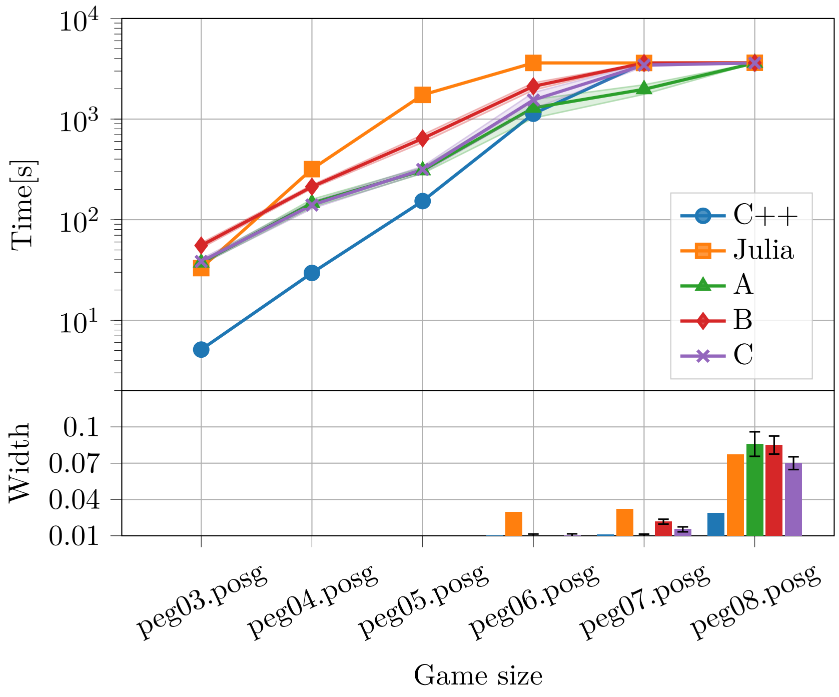

Figure 6.4: Scalability of the algorithm variants

While the variant $\mathrm{H{\scriptsize SVI}}_{\mathrm{A}}$ of the modified algorithm converged on five games, the variants $\mathrm{H{\scriptsize SVI}}_{\mathrm{B}}$ and $\mathrm{H{\scriptsize SVI}}_{\mathrm{C}}$ converged only on the first four. However, their excess gap for peg05.posg is smaller than the excess gap of the original algorithm written in Julia. We can see that, for all the games where the time limit of one hour was not exceeded, the $\mathrm{H{\scriptsize SVI}}_{\mathrm{B}}$ was slower than the other two variants.

The lines for all variants of the modified algorithm also have similar slope, analogously to C++ and Julia implementations of the original algorithm. Nevertheless, the slope is slightly smaller than for the original algorithm. This means that the modified version of HSVI might scale better than the original one with increasing size of the game.

Chapter 7
Conclusion

In this work, we have decided to study and improve the scalability of the existing HSVI algorithm for OS-POSGs. The main bottleneck of the original algorithm is the querying of the upper bound value, which is computed using linear programming. Our goal was to use approximative methods that would make the algorithm faster but still give reasonable results.

We chose to replace the upper bound computation method with neural networks and used quantal response equilibrium in stage games to incorporate this change into the original algorithm. By using those methods, we have completely eliminated the usage of linear programming from the exploration phase of the algorithm.

First, we introduced necessary concepts from game theory. Then we explained how the HSVI algorithm solves POMDPs and how it can be extended to solve OS-POSGs. After that, we introduced our two contributions.

Finally, we tested and evaluated the modifications on the set of pursuit-evasion games. We tuned the parameters, observed the behavior of the algorithm and found reasonable configurations of parameters that were used for detailed experimental evaluation. In the detailed evaluation, we compared the modified algorithm with the original one and tested its robustness and scalability.

The results are, in general, encouraging. The modified algorithm is robust as there is not much deviation to be seen between individual runs. Furthermore, the computed value of the game returned by the modified algorithm is relatively close to the true value, even for larger games. The speed of the modified algorithm is also good. In general, it is faster than our own implementation of the original algorithm. The trend seen when evaluating the algorithm on a set of games of increasing sizes gives us hope that the modifications made might make the algorithm scale better.

7.1 Future work

The performance of the modified algorithm could be further improved by using fine-tuned parameters. A detailed study of some of the results could bring valuable insight into the behavior of the algorithm and help us find a better-suited set of parameters. The modifications could be further evaluated on different domains of OS-POSGs (e.g., patrolling games or blocking games), which would also tell us new information about the workings of the modifications.

We used only very simple neural networks for the upper bound approximation. One possible modification that could improve the performance of our contributions would be to use specialized architectures of neural networks, which specialize in approximating convex functions. This change could improve the precision of the upper bound approximation and also bring back some guarantees on the shape of the upper bound that we lost by using simple neural networks.

Another possibility is to change the used upper bound pruning method. We have chosen a very simple one because of the ease of implementation. However, a more complex method could prune the upper bound in a better way, which could help the neural network with learning the desired shape. Pruning of lower bound could also improve the performance of the algorithm even further.

Appendix A

Parameters evaluation

	UB	SG	λ_{qre}	ε_{qre}	T_{qre}	MSE_{target}	NN_{layers}	ε_{prune}
HSVI$_1$	nn	qre	10.0	0.01	100	0.0001	32-16	0.01
HSVI$_{2=B}$	nn	qre	100.0	0.001	1000	0.0001	32-16	0.01
HSVI$_3$	nn	qre	100.0	0.01	100	1e-06	32-16	0.01
HSVI$_4$	nn	qre	100.0	0.01	100	0.0001	16-8	0.01
HSVI$_5$	nn	qre	100.0	0.01	100	0.0001	32-16	0.001
HSVI$_{6=A}$	nn	qre	100.0	0.01	100	0.0001	32-16	0.01
HSVI$_{7=C}$	nn	qre	100.0	0.01	100	0.0001	32-16	0.1
HSVI$_8$	nn	qre	100.0	0.01	100	0.0001	32-16-8	0.01
HSVI$_9$	nn	qre	100.0	0.01	100	0.01	32-16	0.01
HSVI$_{10}$	nn	qre	100.0	0.1	10	0.0001	32-16	0.01
HSVI$_{11}$	nn	qre	500.0	0.01	100	0.0001	32-16	0.01

Table A.1: Configurations

	time[s]	V_{LB}^{Γ}	V_{UB}^{Υ}	width	$\|\Gamma\|$	$\|\Upsilon\|$	expl.
HSVI$_{C++}$	5	0.8319	0.8410	0.0092	189	322	22
HSVI$_{Julia}$	35	0.8317	0.8404	0.0087	529	651	12
HSVI$_1$	2 812	0.7718	0.7810	0.0093	42 815	8 626	2 710
HSVI$_{2=B}$	56	0.8292	0.8385	0.0092	303	265	18
HSVI$_3$	228	0.8280	0.8276	−0.0005	401	285	72
HSVI$_4$	33	0.8275	0.8351	0.0076	289	272	6
HSVI$_5$	41	0.8248	0.8347	0.0099	391	319	12
HSVI$_{6=A}$	35	0.8276	0.8374	0.0099	284	261	11
HSVI$_{7=C}$	39	0.8262	0.8295	0.0033	357	248	12
HSVI$_8$	45	0.8272	0.8357	0.0085	395	319	14
HSVI$_9$	26	0.7987	0.7753	−0.0234	172	217	3
HSVI$_{10}$	42	0.8129	0.8158	0.0029	617	411	22
HSVI$_{11}$	44	0.8409	0.8449	0.0040	322	278	13

Table A.2: peg03.posg parameters evaluation results

	time[s]	V_{LB}^{Γ}	V_{UB}^{Υ}	width	$\|\Gamma\|$	$\|\Upsilon\|$	expl.
HSVI$_{C++}$	30	0.7769	0.7868	0.0099	2 348	2 551	150
HSVI$_{Julia}$	344	0.7768	0.7867	0.0098	5 131	5 457	66
HSVI$_1$	3 607	0.6848	0.7450	0.0602	45 906	19 623	651
HSVI$_{2=B}$	235	0.7774	0.7866	0.0092	891	773	16
HSVI$_3$	760	0.7730	0.7819	0.0090	2 712	1 232	673
HSVI$_4$	127	0.7678	0.7769	0.0091	1 237	946	24
HSVI$_5$	156	0.7723	0.7812	0.0089	1 490	1 063	33
HSVI$_{6=A}$	172	0.7724	0.7813	0.0089	1 780	1 155	65
HSVI$_{7=C}$	129	0.7745	0.7817	0.0073	1 038	775	25
HSVI$_8$	157	0.7717	0.7812	0.0095	1 427	977	54
HSVI$_9$	127	0.7245	0.7341	0.0095	1 542	1 097	21
HSVI$_{10}$	142	0.7609	0.7653	0.0044	4 244	2 164	163
HSVI$_{11}$	157	0.7891	0.7967	0.0076	1 087	818	106

Table A.3: peg04.posg parameters evaluation results

	time[s]	V_{LB}^{Γ}	V_{UB}^{Υ}	width	$\|\Gamma\|$	$\|\Upsilon\|$	expl.
HSVI$_{C++}$	153	0.7190	0.7289	0.0100	7 845	9 735	424
HSVI$_{Julia}$	1 823	0.7188	0.7288	0.0100	15 604	16 278	155
HSVI$_1$	3 601	0.5796	0.6874	0.1078	46 014	21 079	493
HSVI$_{2=B}$	976	0.7194	0.7248	0.0054	3 349	2 147	124
HSVI$_3$	3 600	0.7178	0.7300	0.0122	15 595	3 921	1 914
HSVI$_4$	352	0.7168	0.7212	0.0044	3 859	2 317	162
HSVI$_5$	386	0.7136	0.7224	0.0088	3 399	2 336	72
HSVI$_{6=A}$	449	0.7156	0.7220	0.0064	4 004	2 548	105
HSVI$_{7=C}$	405	0.7156	0.7234	0.0078	3 794	2 051	121
HSVI$_8$	494	0.7141	0.7213	0.0072	4 493	2 564	188
HSVI$_9$	475	0.6754	0.6654	−0.0100	4 337	2 751	76
HSVI$_{10}$	389	0.6987	0.7067	0.0080	9 647	4 917	238
HSVI$_{11}$	954	0.7302	0.7400	0.0098	6 142	1 953	1 171

Table A.4: peg05.posg parameters evaluation results

	time[s]	V_{LB}^{Γ}	V_{UB}^{Υ}	width	$\lvert\Gamma\rvert$	$\lvert\Upsilon\rvert$	expl.
$\textsc{Hsvi}_{\text{C++}}$	1 130	0.6574	0.6674	0.0100	27 303	33 005	1 566
$\textsc{Hsvi}_{\text{Julia}}$	3 619	0.6563	0.6864	0.0301	23 182	24 399	198
$\textsc{Hsvi}_{1}$	3 609	0.4851	0.6234	0.1383	47 885	22 490	435
$\textsc{Hsvi}_{2=\text{B}}$	3 607	0.6738	0.6942	0.0204	10 353	5 179	443
$\textsc{Hsvi}_{3}$	3 601	0.6718	0.7043	0.0325	7 969	4 690	227
$\textsc{Hsvi}_{4}$	578	0.6655	0.6738	0.0083	4 958	3 614	82
$\textsc{Hsvi}_{5}$	1 051	0.6647	0.6738	0.0091	9 427	5 611	275
$\textsc{Hsvi}_{6=\text{A}}$	837	0.6658	0.6752	0.0094	6 934	4 452	144
$\textsc{Hsvi}_{7=\text{C}}$	1 327	0.6694	0.6792	0.0097	11 362	4 577	358
$\textsc{Hsvi}_{8}$	1 347	0.6746	0.6811	0.0066	11 778	6 290	342
$\textsc{Hsvi}_{9}$	3 601	0.6099	0.6980	0.0881	22 827	7 812	523
$\textsc{Hsvi}_{10}$	1 598	0.6531	0.6605	0.0074	31 925	14 874	701
$\textsc{Hsvi}_{11}$	935	0.6899	0.6990	0.0091	4 607	3 008	245

Table A.5: peg06.posg parameters evaluation results

	time[s]	V_{LB}^{Γ}	V_{UB}^{Υ}	width	$\lvert\Gamma\rvert$	$\lvert\Upsilon\rvert$	expl.
$\textsc{Hsvi}_{\text{C++}}$	3 602	0.6141	0.6253	0.0111	42 846	70 478	2 706
$\textsc{Hsvi}_{\text{Julia}}$	3 617	0.6119	0.6506	0.0387	22 515	24 497	170
$\textsc{Hsvi}_{1}$	3 602	0.3950	0.5741	0.1791	44 054	21 058	357
$\textsc{Hsvi}_{2=\text{B}}$	3 601	0.6328	0.6432	0.0105	7 613	5 646	118
$\textsc{Hsvi}_{3}$	2 885	0.6216	0.6311	0.0095	6 714	5 145	307
$\textsc{Hsvi}_{4}$	1 623	0.6228	0.6327	0.0100	11 968	7 670	175
$\textsc{Hsvi}_{5}$	1 994	0.6274	0.6369	0.0095	14 032	8 613	293
$\textsc{Hsvi}_{6=\text{A}}$	1 428	0.6256	0.6339	0.0083	9 610	6 516	165
$\textsc{Hsvi}_{7=\text{C}}$	3 605	0.6280	0.6453	0.0173	22 633	7 596	688
$\textsc{Hsvi}_{8}$	3 604	0.6295	0.6565	0.0271	21 168	11 394	453
$\textsc{Hsvi}_{9}$	2 170	0.5856	0.5954	0.0098	12 956	7 575	145
$\textsc{Hsvi}_{10}$	3 601	0.5994	0.6419	0.0425	50 979	23 796	894
$\textsc{Hsvi}_{11}$	1 714	0.6450	0.6487	0.0037	6 945	4 876	278

Table A.6: peg07.posg parameters evaluation results

	time[s]	V_{LB}^{Γ}	V_{UB}^{Υ}	width	$\lvert\Gamma\rvert$	$\lvert\Upsilon\rvert$	expl.
$\textsc{Hsvi}_{\text{C++}}$	3 602	0.5621	0.5909	0.0288	27 245	77 551	2 895
$\textsc{Hsvi}_{\text{Julia}}$	3 609	0.5474	0.6593	0.1119	20 757	23 783	142
$\textsc{Hsvi}_{1}$	3 602	0.3102	0.5740	0.2638	41 051	20 670	298
$\textsc{Hsvi}_{2=\text{B}}$	3 608	0.5696	0.6754	0.1058	6 248	6 155	55
$\textsc{Hsvi}_{3}$	3 634	0.5661	0.6446	0.0785	7 998	6 945	157
$\textsc{Hsvi}_{4}$	3 606	0.5785	0.6301	0.0516	21 946	13 398	253
$\textsc{Hsvi}_{5}$	3 611	0.5835	0.6434	0.0599	20 318	12 808	217
$\textsc{Hsvi}_{6=\text{A}}$	3 612	0.5816	0.6462	0.0646	21 033	12 974	220
$\textsc{Hsvi}_{7=\text{C}}$	3 615	0.5832	0.6406	0.0574	20 766	9 279	223
$\textsc{Hsvi}_{8}$	3 613	0.5820	0.6582	0.0762	20 130	12 677	217
$\textsc{Hsvi}_{9}$	3 602	0.5447	0.5971	0.0524	16 800	10 917	153
$\textsc{Hsvi}_{10}$	3 608	0.5610	0.6305	0.0695	49 314	25 662	523
$\textsc{Hsvi}_{11}$	3 606	0.6145	0.6705	0.0560	12 282	8 725	145

Table A.7: peg08.posg parameters evaluation results

Appendix B

Attachment content structure

```
./
└─ HSVIforOneSidedPOSGs/ - root folder of the module
   ├─ games/
   │  ├─ game-format.txt - specification of game files
   │  └─ pursuit-evasion/ - folder containing 6 pursuit-evasion games
   │     peg{03..08}.posg
   ├─ Project.toml - module description, defines dependencies
   ├─ README.md - README file, contains detailed description of the
   │  module, installation instructions and instructions on how to run
   │  the module on attached game files
   └─ src/ - source code directory
      ├─ HSVIforOneSidedPOSGs.jl - file defining the module
      └─ ...
```

Appendix C

Bibliography

[AXK17] Brandon Amos, Lei Xu, and J. Zico Kolter, *Input convex neural networks*, 2017.

[BEKS17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah, *Julia: A fresh approach to numerical computing*, SIAM Review **59** (2017), no. 1, 65–98.

[DHL17] Iain Dunning, Joey Huchette, and Miles Lubin, *Jump: A modeling language for mathematical optimization*, SIAM Review **59** (2017), no. 2, 295–320.

[GB98] Héctor Geffner and Blai Bonet, *Solving large pomdps using real time dynamic programming*, In Proc. AAAI Fall Symp. on POMDPs, 1998.

[Hau00] M. Hauskrecht, *Value-function approximations for partially observable markov decision processes*, Journal of Artificial Intelligence Research **13** (2000), 33–94.

[HBKK20] Karel Horák, Branislav Bošanský, Vojtěch Kovařík, and Christopher Kiekintveld, *Solving zero-sum one-sided partially observable stochastic games*, 2020.

[HBP17] Karel Horák, Branislav Bošanský, and Michal Pěchouček, *Heuristic search value iteration for one-sided partially observable stochastic games*, 2017.

[HBT$^+$19] Karel Horák, Branislav Bošanský, Petr Tomášek, Christopher Kiekintveld, and Charles Kamhoua, *Optimizing honeypot strategies against dynamic lateral movement using partially observable stochastic games*, Computers & Security **87** (2019), 101579.

[IBM20] IBM, *V20.1.0: User's manual for cplex*, 2020.

[ISF$^+$18] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah, *Fashionable modelling with flux*, CoRR **abs/1811.01457** (2018).

[MP96] Richard D. McKelvey and Thomas R. Palfrey, *Quantal response equilibria for normal form games*, 1996.

[Nas51] John Nash, *Non-cooperative games*, The Annals of Mathematics **54** (1951), no. 2, 286.

[RN09] Stuart Russell and Peter Norvig, *Artificial intelligence: A modern approach*, 3rd ed., Prentice Hall Press, USA, 2009.

[SFA$^+$18] Arunesh Sinha, Fei Fang, Bo An, Christopher Kiekintveld, and Milind Tambe, *Stackelberg security games: Looking beyond a decade of success*, Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence Organization, July 2018.

[SLB08] Yoav Shoham and Kevin Leyton-Brown, *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*, Cambridge University Press, 2008.

[SS12a] Trey Smith and Reid Simmons, *Point-based pomdp algorithms: Improved analysis and implementation*, 2012.

[SS12b] Trey Smith and Reid G. Simmons, *Heuristic search value iteration for pomdps*, CoRR **abs/1207.4166** (2012).

[Tam09] Milind Tambe, *Security and game theory*, Cambridge University Press, 2009.

[TBN20] Petr Tomášek, Branislav Bošanský, and Thanh H. Nguyen, *Using one-sided partially observable stochastic games for solving zero-sum security games with sequential attacks*, Decision and Game Theory for Security (Cham) (Quanyan Zhu, John S. Baras, Radha Poovendran, and Juntao Chen, eds.), Springer International Publishing, 2020, pp. 385–404.

[vN28] J. v. Neumann, *Zur theorie der gesellschaftsspiele*, Mathematische Annalen **100** (1928), no. 1, 295–320.

[Ša19] Jaroslav Šafář, *Approximation of bound functions in algorithms for solving stochastic games*, Bachelor's thesis, CTU FEE, Department of Cybernetics, 2019.