Bachelor Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Comparing Exploration Methods in Partially Observable Stochastic Games

Jakub Rada

Supervisor: doc. Mgr. Branislav Bošanský, Ph.D.
Field of study: Open Informatics
Subfield: Artificial Intelligence and Computer Science
May 2022

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Rada Jakub**	Personal ID number:	**492291**
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:

Comparing Exploration Methods in Partially Observable Stochastic Games

Bachelor’s thesis title in Czech:

Porovnání metod explorace v částečně pozorovatelných stochastických hrách

Guidelines:

HSVI algorithm for solving subclasses of partially observable stochastic games approximates the value function of the game using a lower bound and an upper bound value function. Every iteration of the algorithm, the point-based Bellman-style updates are performed over these two approximate value functions. In HSVI, the belief points for the updates are selected based on the strategies of the players and the gap between the lower and upper bound. This heuristic however does not have to be the optimal method for exploring the space of belief points in POSGs. The goal of the student is to:
1. Get familiar with the algorithm HSVI for POSGs.
2. Survey the existing methods for solving exploration-exploitation problem in game-theoretic settings.
3. Select a subset of appropriate methods from the previous step and implement them as belief-points selection methods into the HSVI.
4. Compare these methods and analyze the impact of these different exploration techniques on the effectivity with which the space of the belief points is explored in POSGs.

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] Slivkins, Aleksandrs. "Introduction to multi-armed bandits." arXiv preprint arXiv:1904.07272 (2019).

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: **08.01.2022**   Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

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. Branislav Bošanský for his patience, willingness and support.

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA CZ LM2018140) supported by the Ministry of Education, Youth and Sports of the Czech Republic.

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, 20.5.2022

....

Signature

Abstract

The partially observable stochastic games model many situations consisting of two independent agents. Their one-sided subclass can be approximately solved by the HSVI algorithm, which estimates the optimal value function with lower and upper bound value functions. The approximation is refined by iteratively performing Bellman-style point-based updates on both bounding value functions in belief-points selected by a heuristic approach. However, this heuristic based on the strategies of both players and the gap between the bounding functions is not proven to be the optimal exploration method for searching the space of belief-points.

In reinforcement learning, multi-armed bandit algorithms are a tool for solving the exploration-exploitation problem. It is thus possible to use the bandits as an alternative approach for exploring the belief-point search space and refine the bounds in the HSVI algorithm. Additionally, the multi-armed bandits can provide similar alternative approach for solving stage games in the value iteration algorithm for the fully observable stochastic games. Moreover, the need of linear programming is thus eliminated, which could lead to improved scalability.

The goals of this thesis were the integration of this novel exploration method into the existing solving algorithms and comparing subset of the multi-armed bandit algorithms on both SGs and OS-POSGs.

Keywords: game theory, stochastic games, partially observable stochastic games, multi-armed bandits, value iteration, hsvi

Supervisor: doc. Mgr. Branislav Bošanský, Ph.D.
Department of Computer Science

Abstrakt

Částečně pozorovatelné stochastické hry modelují mnoho reálných situací skládající se ze dvou nezávislých agentů. Jejich podtřídu jednostranných her lze přibližně vyřešit algoritmem HSVI, který pomocí dvou value funkcí, jedné spodní a jedné horní meze, odhaduje optimální value funkci hry. V každé iteraci se aplikuje Bellmanův operátor na obě meze, který aktualizuje jejich hodnotu v bodech, které byly vybrány heuristickou funkcí. Nicméně, není dokázáno, že tento heuristický přístup, který je založený na strategiích obou hráčů a velikosti mezety mezi mezními funkcemi, je optimální metodou explorace pro prohledávání prostoru bodů beliefu.

Mnohorucí bandité jsou algoritmy používané v posilovaném učení, které řeší problém vyvažování explorace a exploitace. Je tedy možné použít tyto mnohoruké bandity jako alternativní způsob prohledávání prostoru bodů beliefu a tím zlepšovat meze HSVI algoritmu. Mnohorucí bandité mohou také zajistit podobný alternativní přístup k řešení fázových her v plně pozorovatelných stochastických hrách řešených metodou iterace hodnoty. Navíc, použití banditů eliminuje použití metod lineárního programování, které mohou způsobovat špatnou škálovatelnost původních algoritmů.

Cílem této práce byla integrace tohoto nového přístupu explorace do iterace hodnoty a HSVI a porovnání některých mnohorukých banditů na plně i částečně pozorovatelných stochastických hrách.

Klíčová slova: teorie her, stochastické hry, částečně pozorovatelné stochastické hry, problém mnohorukého bandity, value iteration, hsvi

Překlad názvu: Porovnání metod explorace v částečně pozorovatelných stochastických hrách

Contents

Chapter 1
Introduction

Many real-life problems can be modelled as some form of sequential reasoning. Even the single-agent models have usage in a wide spectrum of applications, the main areas being agriculture, adjusting production to demand, finances, etc. [1]. Other examples, where some information is hidden from some agents, include autonomous robotics, networking and marketing [2].

Even more interesting are the environments consisting of multiple rational independent agents competing or cooperating with each other to reach their own goals. The formalisms modelling these types of environments are called *games* and are studied by the *game theory*. However, due to the multiple agents, it is not as straightforward to define a solution of a game-theoretical problem and hence various different solution concepts are used. Equilibria are the most known solution concepts, the most famous being *Nash equilibrium*. Moreover, the agents often need to randomize their decisions to behave optimally, which is not necessary in the single-agent environments. This enlarges the size of the search space from finite to infinite. These two aspects make solving games much harder than sequential decision problems containing only a single agent.

The game theory was founded by John von Neumann as a method of economy, with the well-known *oligopoly game*, product pricing methods and various number of other means of studying behaviour of competing agents in an economic environment. A specific example is designing market trading strategies by using the tools of the game theory [3]. However, the game theory encompasses many more fields of application than economy. For example biology [4], machine (deep) learning [5], computer security, where a more concrete example being *Attacker Defender model for Intrusion Detection Systems in Cloud* [6] and even the recently popular blockchain [7].

The game theory already provides means and methods find optimal strategies or other solution concepts for the tasks mentioned above, but these methods usually have drawbacks which are listed in the next part.

1.1 Complications of existing methods

As it was said, there already exists some algorithm or method which is proven to be capable of finding the solution for each of the problems and models listed above. Unfortunately, these proofs work in the theoretical sense but in practice, many of these methods do not scale very well for very large instances as its demand for either space or time grows rapidly. In the case of multiagent sequential reasoning, which is the sole topic in this thesis, these methods often require linear programming.

For single-agent or even multiagent environments, where all agents have perfect information, this is not an issue as the linear programs do not grow in size as quickly and the problem is generally easier to solve. But when the perfect information is even partly removed, for example in *partially observable Markov decision processes* and *partially observable stochastic games*, these mathematical programs grow larger and take longer to solve.

In practice, the methods using solely the approach of linear programming are unusable for large problems, and thus approximative approaches had to be devised. One of these approximative methods, which is the cornerstone of this thesis, is the *Heuristic search value iteration* first introduced for POMDPs in [8] and then proposed for one-sided partially observable stochastic games in [9]. The latter is the main source of inspiration behind this thesis as it introduces the OS-POSG model and presents the solving HSVI algorithm with proof of convergence.

This presented method, however, also uses many linear programs, and thus it is possible to improve its scalability. Hence, some other method than linear programming is needed to drive the search in the correct direction. This thesis focuses on multi-armed bandit algorithms [10] as one of the most important reinforcement learning tools, which can provide this needed type of search by learning the properties of the environment by a repeated interaction.

1.2 Goals of the thesis

In this bachelor thesis, we introduce the reader into the problematics of sequential decisions with single or multiple agents, summarize the existing methods for solving such problems and discuss their properties. We state why some solution methods are not suitable for practical use and what are their main disadvantages. Then, we describe the concept of the multi-armed bandit algorithms and how they could be used to remove some problems in the standard algorithms.

We aim to incorporate the surveyed multi-armed bandit algorithms, which are meaningful in the context of games, particularly stochastic games and partially observable stochastic games, into the existing algorithms and investigate their behaviour. The goal is to realize, which of them possess properties sufficient to make the enwrapping algorithm find the optimal solution or at least a good approximation, and how quickly and reliably they discover the solution. The result of this thesis is a comparison of individual multi-armed bandits supported by experimental evaluation of the algorithms on domains of (partially observable) stochastic games. The best found bandit algorithms and their settings are highlighted and summarized. Also, the output is also a modification of an existing algorithm, which does not rely on linear programming as heavily as the original method.

In addition to the assignment of this thesis, where only the class of partially observable stochastic games is stated, we first focus more on the fully observable stochastic games and comparison of the bandits on this problem. The class of partially observable stochastic games is a harder problem than the stochastic games, mainly because of the uncertainty about the current state of the environment which makes the search space infinite rather than finite. Due to the infinite size, the so-called *belief* space needs to be discretized thus allowing us to approximate the space with large but finite number of elements, in this case finite number of bandits, but introducing another imprecision into the process.

Because of this increased difficulty it is reasonable to thoroughly compare the bandit algorithms on the easier domain and then proceed with investigating the differences which arise in the harder partially observable problem. Once we understand how to bandits behave in the finite dimension, the transition to the harder problem is smoother as some assumptions can be made in advance.

The comparisons are the most important part of the thesis and conclude the main body of this text. Last, we summarize the thesis and the results achieved by the experimental evaluation together with proposal of future work. In the appendices, a more detailed comparison of the multi-armed bandits is presented together with time analysis and implementation details of the used algorithms.

Chapter 2
Technical background

In this chapter, we provide a theoretical ground which is essential for the rest of the thesis. First, we describe basic models for single-agent environments as a foundation of sequential reasoning. Then, we use these notions to define fundamental concepts of game theory with a focus on game types studied in later chapters.

Definitions and terms in this chapter are based mainly on two textbooks, namely *Artificial Intelligence: A Modern Approach*[11] and *Multiagent Systems: Algorithmic, Game-Theoretic and Logical Foundations*[12].

In the next two sections, we consider environments consisting of a single agent. We define the agent's goals and discuss the agent's means to behave optimally and reach the goals.

2.1 MDP

The most basic tool for modelling sequential reasoning is the *Markov decision process*, abbreviated as MDPs. An MDP describes how a single agent moves through different states of the environment by choosing actions in discrete time steps. After each step, the agent receives a real *reward* from the environment and stores it. The agent aims to maximize this accumulated quantity over time.

MDPs generally model *stochastic* environments, which means that changes among states are (sometimes partly) randomized, so one action can possibly lead to more than one state with non-zero probability. In the special case, when all probabilities in the transition function are either 0 or 1, we say that the MDP is *deterministic*. The agent's information, however, is perfect, so he knows precisely his current state and actions that can be played. When properties of the MDP do not change in time, it is called *stationary*, otherwise *non-stationary*.

Definition 2.1. A Markov decision process is a tuple (S, A, T, R), where:

- S is a set of possible states,
- A is a set of actions,
- $T(s' \mid s, a)$ is a conditional probability of transitioning from state $s \in S$ to state $s' \in S$ when action $a \in A$ is played,
- $R(s)$ is an immediate reward function.

In an MDP, there is a very important notion of so-called *Markov property*. This property states, that the outcome depends only on the current state and the chosen action. In other words, the actions played before the current state have no effect on the next state.

In each time step, the agent, which is currently in state s, selects an action a and according to the transition function T moves to state s' and receives *immediate reward* r based on the reached state. MDPs can have either finite or infinite horizon. In the case of the finite horizon, we can use *additive rewards*, where all are summed together to form *utility*. When the time horizon is infinite, we typically use $0 < \gamma < 1$ as the discount factor and compute utility as *discounted rewards*. Universally, additive rewards can be defined as discounted rewards only with $\gamma = 1$. The discount factor is used to multiply the received reward in an increasing manner as the play continues to the next time steps. The resulting discounted reward is then computed as

$$R(\gamma) = \sum_{t=0}^{n} \gamma^t r_t. \tag{2.1}$$

A *policy* π describes how the agent selects actions. It is mapping $\pi : S \to A$, and $\pi(s)$ returns the preferred action to be played in state s. If the agent follows an optimal policy π^*, it will receive maximal possible *expected reward* or, in other words, utility. Based on the situation, it can be computed as the additive rewards, discounted rewards or even average rewards.

2.2 POMDP

The partially observable Markov decision process, abbreviated as POMDP, is a generalization of an MDP. In contrast with an MDP, the agent has *imperfect* information about the states of the environment, i.e. does not know in which state he currently is and also to which state he transitioned. To capture this modification, we have to add two new concepts to the definition of MDP (Definition 2.1).

Definition 2.2. A partially observable Markov decision process is a tuple (S, A, O, T, R, Z, b), where:

- S is a set of possible states,
- A is a set of actions,
- O is a set of observations,
- $T(s' \mid s, a)$ is a conditional probability of transitioning from state $s \in S$ to state $s' \in S$ when action $a \in A$ is played,
- $R(s)$ is an immediate reward function,
- $Z(o \mid s', a)$ is a conditional probability of generating observation $o \in O$ when the agent transitioned to state $s' \in S$ while playing action $a \in A$,
- $b \in \Delta(S)$ is a probability over states called initial belief.

The first new concept is *observations* $o \in O$, which are generated when transitioning to a new state and then published. The agent can deduce some information about the change that occurred and use it, but it can be inaccurate or wrong.

The second new element is *belief* $b \in \Delta(S)$. Because the current state is not revealed to the agent by the environment, he keeps a probability distribution over states $b \in \Delta(S)$. This distribution b represents the agent's knowledge about its current state. The agent starts from some *initial belief* and updates it with incoming observations as follows.

$$b'(s') = \mu Z(o \mid s', a) \sum_{s \in S} T(s' \mid s, a) b(s) \tag{2.2}$$

where μ is a normalizing factor so that b' is a distribution. Belief assigns a probability to each state $s \in S$ and thus forms a $|S|$-*simplex*, where $|S|$ is the number of states.

As opposed to the MDPs, here the optimal behaviour does not depend on the real state of the agent, but rather on his belief about his state. Thanks to this, we can formulate POMDP as an MDP over belief points instead of states. However, the number of these states would be infinite and thus intractable for solution methods for MDPs.

Even though the methods originally used for MDPs are unusable, there are some modifications that help to overcome these complications. We leverage the fact, that there is a finite number of possible actions in an POMDP and define *α-vectors*. An α-vector[13] is a linear function $\alpha : \Delta(S) \to \mathbb{R}$ and is conveniently represented as values $\alpha(s)$ in each of the vertices $s \in S$ of the belief simplex and thus forming a hyperplane in the belief space. Then value $\alpha(s)$, where $s \in S$, directly represents a value of an n-step policy conditioned by the starting state s. The conditionality follows from the agent's uncertainty about the current state, but the dependence on belief is linear. Thus, a value of the α-vector in a specific belief point $b \in \Delta(S)$ is then computed as a convex combination of the α-vector's corner values

$$\alpha(b) = \sum_{s \in S} b(s)\alpha(s). \tag{2.3}$$

This notion of α-vectors will be essential in the standard solution methods for POMDPs (Section 3.2) and more importantly for tackling the partially observable stochastic games (Section 3.4).

2.3 Game theory

MDPs and POMDPs provide formalisms for modelling and solving environments where only one agent is present. However, many real-world situations contain multiple independent agents. These problems are modelled by the *game theory*.

The game theory includes various game types. They differ in the number of players, time horizons, utility functions, whether the agents compete or cooperate and in other aspects. From now on, we will focus on strictly competitive games.

2.3.1 Basic concepts

In the next parts, the fundamental principles of the game theory are explained. After that we will look into a specific subset of games with greater detail.

Players

In the game theory, agents are called *players* and there can be an arbitrary number of them, as long as there is more than one. The convention is, that when we look at the game from the perspective of player i, then the other players are denoted together by $-i$ and are called *adversaries* or *opponents*. In the rest of this thesis, we will consider only two-player games and thus $-i$ denotes only one adversary.

Rewards

There exist many distinct types of games in terms of rewards or, as sometimes termed, payoffs or utilities. For example, *common-payoff* games have equal rewards for all players and action profiles. *Action profile* is a subset of the Cartesian product of the action sets of individual players, i.e. $a = (a_1, ..., a_n)$, where $a_i \in A_i$. We will focus on *zero-sum* games (only two-player), which have a property that the reward of player i means the same reward for the player $-i$ only multiplied by -1.

Strategies

In single-agent environments, we worked with policies to describe the agent's behaviour. In games, the term *strategies* is used. We distinguish two types of strategies, *pure* and *mixed*. A player following pure strategies selects just one action in each state and plays it with probability 1. In contrast, in mixed strategies a player follows a distribution over actions and plays each of them with probability ≤ 1. A strategy profile is a subset of the Cartesian product of the sets of all mixed strategies of each individual player

$$s = (s_1, \dots, s_n) \in \Delta(A_1) \times \dots \times \Delta(A_n). \tag{2.4}$$

Actions, that are played with probability > 0 in a strategy s, are called *support* of a mixed strategy s. The expected utility of player i under mixed strategy profile s is then computed as

$$u_i(s) = \sum_{a \in A} u_i(a) \prod_{j=1}^{n} s_j(a_j), \tag{2.5}$$

where A is the set of all actions and $u_i(a)$ is an assignment of utility to action $a \in A$.

Nash Equilibrium

In game theory, there are many notions of equilibria, but probably the most important is *Nash equilibrium*[14], which describes how players play optimally in an n-player game. Nash Equilibrium is a stable strategy profile for which it holds, that none of the players wants to deviate from his (possibly mixed) strategy because it would decrease its payoff. More formally, for every player holds, that his strategy s_i is the *best response* to the strategy profile of the remaining players s_{-i}. The overall strategy profile of all players in the game is then denoted as $s = (s_i, s_{-i})$. The best response to a strategy is such a strategy, that gains the same or better payoff, than any other player's strategy.

Definition 2.3. Player i's best response to strategy profile s_{-i} is a mixed strategy $s_i^* \in S_i$, for which holds that

$$u_i(s_i^*, s_{-i}) \geq u_i(s_i, s_{-i}) \qquad \forall s_i \in S_i. \tag{2.6}$$

Nash proved that in every game with a finite number of players and action profiles exists at least one Nash Equilibrium[14].

Minimax theorem

There are two special strategies and values, *minmax* and *maxmin*, which are then linked together with the *Minimax theorem*[15]. Here, we restrict only to two-player zero-sum games, even though maxmin is defined for a more general setting.

Maxmin strategy is such a strategy, that maximizes the worst possible reward for a player following a strategy profile s_i.

Definition 2.4. The maxmin value for a player i is

$$\max_{s_i} \min_{s_{-i}} u_i(s_i, s_{-i})$$

To obtain the strategy instead of value, the max changes to argmax. In other words, a player is guaranteed to obtain at least its maxmin value, when following strategy s_i, but the final payoff can be even higher.

On the other hand, minmax strategy wants to force the adversary to receive the worst possible value even when playing optimally. This can be viewed as trying to punish the other as much as possible without considering its own reward.

Definition 2.5. The minmax value for a player i is

$$\min_{s_i} \max_{s_{-i}} u_{-i}(s_i, s_{-i})$$

According to von Neumann's Minimax theorem, these two values coincide for two-player zero-sum games.

Theorem 2.6. (**von Neumann**[15]) In any finite two-player zero-sum game, in any Nash Equilibrium, each player receives a payoff that is equal to both its minmax and maxmin values.

The maxmin value is thus equal to the minmax value, we call this value *the value of the game*. Strategies forming these values are then automatically Nash equilibria.

Game forms

Games can be defined in many forms each representing different views. The most general and fundamental is *normal-form* and most other representations can be reduced to it.

Definition 2.7. A finite n-player normal-form game is a tuple (N, A, u), where

- N is a set of n players,
- $A = A_1 \times \cdots \times A_n$, where A_i is a finite set of actions available to player i,
- $u = (u_1, \ldots, u_n)$, where u_i is a real-valued mapping $u_i : A_i \to \mathbb{R}$ for player i.

Players' utilities are written into an n-dimensional matrix, where n is the number of players. In each matrix cell representing one possible outcome, there is a vector of utility values for players. The vector has a length n and the cells in each dimension i are marked by possible actions for a player i. This representation is very convenient but quite ineffective as the size grows exponentially. Moreover, when reduced from other forms, the size grows even bigger.

Normal-form representation does not explicitly consider the concept of time. In normal-form games, all agents act *simultaneously* and independently on each other's actions. When time plays a major role in the environment, we use the *extensive form*.

The extensive form can be visualized as a tree, often called *the game tree* and this way implies the notion of time. For each node in the tree a player, which takes actions in this round, is assigned together with available actions. Depending on the selected action in each round, the state of the game transitions to one of the children of the current node and another player gets to act. The sequence of these selected actions is called a *history*. The game can be solved by solving subgames and applying backwards induction to find the Nash Equilibrium, one of the used algorithms being minimax with $\alpha\beta$ pruning for two-player zero-sum games.

If the extensive-form game is of *perfect information*, every player knows exactly in what state it is currently in and a pure strategy Nash Equilibrium exists[16]. A more general concept is a game with *imperfect information*, which better represents real applications and where the agent cannot distinguish between some states. Sets of these equivalent states are called *information sets* and available actions are not defined for individual states but rather for these information sets. When we consider imperfect information games, we assume that they are games of *perfect recall*, meaning that even though the player does not know in what state he is, he perfectly remembers all his previous taken actions.

Games can be played only once, multiple times or infinitely many times. These are called *repeated games*. In the case of infinitely repeated games, we have to employ either average rewards or discounted rewards. In the next section, we will introduce a generalization of repeated games, which is also a generalization of POMDPs.

2.4 Stochastic games

In previous section (Section 2.3), we laid out basic concepts of the game theory. In this section, we define a class of games that generalize the (partially observable) Markov decision process for more than one agent as well as repeated games. This class is known as *stochastic games*, *Markov games* or even shortly *SG*s. We discuss some important properties, which will be useful later in the partially observable variant. Last, methods that provide solutions for two-player, zero-sum stochastic games are presented and compared.

2.4.1 Definition

In a stochastic game, players repeatedly play normal form games drawn from some finite set. These games are usually called *stage games*. After each stage, players receive rewards based on the outcome of the normal form game and then proceed to the next stage by drawing a new game from the set. The probability of drawing this new game from the set depends solely on the previous game and the action profile selected by the players, which is analogic to the *Markov property*. It is clear, that with only one agent present the stochastic game reduces to an MDP.

Another perspective on stochastic games is that players move together through a finite set of states. In each state, all of them select an action to play and depending on these actions they move to another state while receiving rewards. Also, as stated before, we will restrict ourselves to the two-player, zero-sum case (2.3.1). As mentioned before, this is a strictly competitive type of game, where player i receives a reward r and player $-i$ receives a reward $-r$ Both players want to maximize their outcome. However, it is usually redefined that both players receive reward r and while player 1 wants to maximize this quantity (the *Max* player), player 2 minimizes it (the *Min* player).

Definition 2.8. A two-player zero-sum stochastic game[17] is a tuple $(S, A_1, A_2, T, R, \gamma)$, where

- S is a finite set of states,

- A_1, A_2 are finite sets of actions available to player 1, resp. player 2,
- $T(s' \mid s, a_1, a_2)$ is the *transition function* $T : S \times A_1 \times A_2 \times S \to [0, 1]$, which gives the probability of transitioning from state s to s' with selected action profile (a_1, a_2),
- $R(s, a_1, a_2)$ is the *reward function* $R : S \times A_1 \times A_2 \to \mathbb{R}$ specifying the reward gained by player 1 when action profile (a_1, a_2) was played in the state s and
- γ is a discount factor.

2.4.2 Properties

Even though the transitions between states are probabilistic, so one action profile can lead to multiple new states, stochastic games are games with perfect information. Both players can observe actions played by their opponent and can form play histories consisting of states and played actions. They can leverage this information to better reason about selecting the best action to play in every state.

Both players strive to obtain the best possible cumulative rewards throughout the whole play. The maximizing player wants to get the maximal possible overall outcome, while the minimizing player wants the minimal possible outcome. However, the play can be infinitely long, so other methods to accumulate rewards have to be used. We will focus on discounted rewards as they are less complicated and have better properties than for example average rewards. In discounted rewards case players collect rewards in a way of infinite discounted sum $\sum_{t=0}^{\infty} \gamma^t r_t$, where $0 < \gamma < 1$ is the *discount factor*.

For every two-player, zero-sum stochastic game with discounted rewards there exists a Nash equilibrium independent of the starting state. Moreover, in this type of game, the equilibrium strategies are *Markovian*, meaning that the strategies depend only on the current state. These two properties combined are called *Markov perfect equilibrium*. However, these stationary equilibrium strategies are not necessarily deterministic as in the single-agent MDP case. They might be stochastic, i.e. mixed, which means that a player doesn't have a single best action, but is given an optimal probability distribution over actions $\Delta(A_i)$ from which the played action is drawn. This causes some additional demands on the solving algorithm, which will be revealed in (Section 3.3).

2.5 Partially observable stochastic games

The next step from stochastic games described in previous sections towards a more general game model are *partially observable stochastic games*, or POSGs.

In Markov games, the current state s was accurately revealed to both players in every time step t. This fact allowed players to reason about quality of their individual actions more precisely. However, in partially observable stochastic games, the current state generally stays hidden to both players. To provide the agents with some information, they receive private *observations* similarly as in POMDPs (Definition 2.2). Each player i then keeps belief $b_i \in \Delta(S)$, which is a probability distribution over states in S representing their knowledge about the current state. They make their decisions based on this belief b_i.

In a POSG, both players keep their own belief initialized as $b^{\text{init}} \in \Delta(S)$, which is public. Based on transition function T they both move to the same new state $s' \in S$, but they do not know which state. Instead, they are each given a private observation and with this observation they adjust their beliefs. They do not know in which state they are located, the actions played by the adversary, or the observation received by the adversary. However, they have *perfect recall*, so they can remember history of their own actions and observations. From these pairs, they can construct *behavioural strategies* according to which they act, i.e. $\sigma_i : (A_i O_i) \to \Delta(A_i)$.

Even though, there exist theoretical results for solving POSGs with finite horizon, they are not applicable to the infinite horizon[18]. In their solution [18], they define a distribution over possible action-observation histories and by these distributions they define value function similar to *Bayesian games*. However, the number of action-observation histories grows exponentially with horizon, and thus it is impossible to use this approach on games with infinite horizon. To tackle this issue, there exist restrictions that simplify the problem and make practical solutions possible.

One of such restrictions are *one-sided partially observable stochastic games*, or *OS-POSGs*[9]. In this type of games, one of the player has perfect information while the other still receives observations. This type is focus of this thesis and will be discussed in more detail in the next section.

The second simplification of POSGs are *partially observable stochastic games with public observations*, or *PO-POSGs*[19]. Here, both players still receive observations, but the observations are public, meaning that the player 1 knows what the player 2 observed and vice versa. This allows the agents to reason about the belief of the adversary and thus better compare quality of the actions.

2.6 One-sided partially observable stochastic games

Contents of this section, most importantly the definitions and formulas, are taken from a doctoral thesis [9] with a slight change in notation to match the other definitions in this document. It focuses and describes a subclass of partially observable stochastic games, POSGs (Section 2.5), mentioned in the previous section. Because this model is a restricted version of the general model, it avoids some problems which make the general model intractable. By introducing asymmetry in the knowledge of the agents it reduces the high dimension of the problem.

Here, we present the model and basic concepts, in the next chapter (Chapter 3) is then described the standard methods for solving this class of games. In the chapter (Chapter 5) is then proposed alternative solving algorithm employing multi-armed bandits.

2.6.1 Model

Definition 2.9. A two-player zero-sum One sided partially observable stochastic game (or OS-POSG) is a tuple $G = (S, A_1, A_2, O, T, R, b^{\text{init}}, \gamma)$, where

- S is a finite set the game's states,
- A_1 and A_2 are finite sets of actions available to player 1, or player 2 respectively,
- O is a finite set of observations for the player 1,
- $T : S \times A_1 \times A_2 \to \Delta(O \times S)$ is a probabilistic transition function defining probability of transitioning from state $s \in S$ by playing $(a_1, a_2) \in A_1 \times A_2$ to a new state $s' \in S$ while generating observation $o \in O$,

- $R : S \times A_1 \times A_2 \to \mathbb{R}$ is a reward function of player 1,
- $b^{\text{init}} \in \Delta(S)$ is the initial belief of player 2,
- $\gamma \in (0, 1)$ is the discount factor.

As opposed to the POSG model, in OS-POSGs, only the player 1 receives observations, the opposing player 2 has perfect information about the current state and played actions by both players. To represent knowledge of the current state held by the oblivious player 1 we use the same notion of belief as was used in POMDPs and general POSGs. This said belief is a probability distribution over the states $\Delta(S)$.

Course of the game

The game begins by sampling the initial state s^1 from the initial belief b^{init}.

Then, for an infinite number of rounds, i.e. stages, the players simultaneously choose actions which move them through the game environment. After playing this pair of *joint* actions, both of the agents receive rewards from the reward function R potentially with some additional data and the game transitions into the next state s'. Note that the OS-POSG is a zero-sum game, thus the rewards for players differ only in the sign. The next state $s' \in S$ is chosen by the environment with respect to the transition function T, the current state s and the joint action profile (a_1, a_2).

Not only the transition function returns a probability distribution over the states, but it also provides an observation $o \in O$ for the first player. Player 2, on the other hand, is shown exactly the state of the game and the action played by the adversary. Thus, he has perfect information about the game and can observe the entire course of the game. The player 1 knows only his selected actions and the received observations, hence the information asymmetry between the two agents.

Strategies

Due to the repeated infinity nature of the game, two types of strategies are defined. The one known from the matrix games and stochastic games are stage strategies, which correspond to probability distributions over available actions in the particular stage. Due to the asymmetry in knowledge, each player has a different definition of a stage strategy.

Definition 2.10. Let $G = (S, A_1, A_2, O, T, R, b^{textinit}, \gamma)$ be a OS-POSG. Then

- a stage strategy of the player 1 is defined as a distribution $\pi_1 \in \Delta(A_1)$ and
- a stage strategy of the player 2 is a mapping $\pi_2 : S \to \Delta(A_2)$.

The sets of all stage strategies are denoted as Π_1 and Π_2.

The mapping π_2 can be understood as a probability distribution over actions conditioned by the current state s, which is perfectly observed by the player 2. Thus, as used in [9] and [17], we use the notation $\pi_2(a_2 \mid s)$ for the stage strategy mapping.

Player 1, however, cannot adjust his strategy for the current state, because it is unknown to him. Thus, the simple distribution over actions $\Delta(A_1)$.

These strategies can be used to play only a single round of the game. In OS-POSGs there are infinitely many stages, thus a more powerful notion of strategy is needed. For this serve *behavioural* strategies.

Definition 2.11. Let G be an OS-POSG. Then, mapping $\sigma_1 : (A_1 O)^* \to \Delta(A_1)$ is a behavioural strategy of player 1 and mapping $\sigma_2 : (SA_1A_2O)^*\, S \to \Delta(A_2)$ is a behavioural strategy of the player 2. The sets of all behavioural strategies are denoted as before as Σ_1 and Σ_2.

The *behavioural strategies* define a distribution over actions given the previously observed *history* of length t thus determining the way the game is played in specific stages at time t.

Moreover, composition of behavioural strategies and stage strategies is possible and results in new behavioural strategies assigning distributions to histories of length $t+1$ (one additional stage strategy to the histories of length t). This composition is described in more detail in [17] together with a derivation and a formula.

The infinite history is then called a *play*. To assign a value to a play, we consider discounted rewards in the same way as in sequential decisions 2.1 and the basic concepts of the game theory 2.3.1 and that is $\sum_{t=1}^{\infty} \gamma^{t-1} R(s^t, a_1^t, a_2^t)$

Belief

As mentioned before, the unobserving player 1 keeps a probability distribution $b \in \Delta(S)$ called belief, which represents his knowledge about the state of the game. Based on the observations received from the environment, he adjusts it during the course of the game to simplify choosing appropriate actions. However, an assumption is needed, that the player 1 knows some stage strategy π_2 of the adversary and based on this strategy he updates his current belief. If this premise is fulfilled, the new belief for the following stage is computed as follows.

Definition 2.12. Let $b \in \Delta(S)$ be current belief, $\pi_1 \in \Pi_1$ a strategy of the player 1, $\pi_2 \in \Pi_2$ a strategy of the opponent, a_1 be action played in the current round by the player 1 and o an observation generated by the transition function T when proceeding to the next state. An *updated belief* is then computed for every state $s' \in S$ as

$$\tau(b, \pi_2, a_1, o)(s') = \frac{1}{\mathbb{P}_{b,\pi_1,\pi_2}[a_1, o]} \sum_{(s,a_2)\in S\times A_2} b(s)\pi_1(a_1)\pi_2(a_2 \mid s)T(o, s' \mid s, a_1, a_2) \qquad (2.7)$$

The normalization constant $\frac{1}{\mathbb{P}_{b,\pi_1,\pi_2}[a_1,o]}$ ensures that the new belief is in fact a distribution over the states $\Delta(S)$.

2.7 Summary

In this chapter, we presented theoretical background for problems related to and used in this thesis. We discussed simple models of single-agent sequential reasoning as are MDPs and POMDPs. Then we defined basic principles and terms of the game theory, we focused on the description stochastic games and partially observable stochastic games. The one-sided subclass of POSGs was mentioned separately as its solving algorithms will be one of the main topics.

In the following chapter, standard methods to obtain solutions for these aforementioned models are presented. Then proceeds a discussion of basics of reinforcement learning, specifically multi-armed bandit algorithms, which play a crucial role in the main goals of this thesis. Results of these two chapters are then combined to define algorithms incorporating the bandits inside the standard method settings.

Chapter 3
Standard solution methods

In this chapter we mention and describe the solution methods that are commonly used to solve the models mentioned in the previous chapter (Chapter 2). These methods are proven to find the solution or at least an approximation but usually struggle from poor scalability. In the next chapter, we use ideas from these methods to design alternative solution methods.

As in the previous chapter, notions and theoretical properties are based on *Artificial Intelligence: A Modern Approach*[11], *Multiagent Systems: Algorithmic, Game-Theoretic and Logical Foundations*[12].

3.1 MDP

To solve an MDP means finding an optimal policy π^*, which will guarantee that the agent gains the best utility if he plays according to π^*. Due to the Markov property, the policy π^* need not be computed directly, but it is sufficient to find the optimal value function V^*. The optimal policy π^* can be extracted from V^*.

Algorithms designed to find the solution of an MDP are based on *value iteration* or *policy iteration*. These two algorithms are built around *Bellman equations*, different for both methods. We focus on the former because value iteration will be crucial in solving stochastic games later in this thesis.

Theorem 3.1. Let (S, A, T, R) be an MDP. Let $V : S \to \mathbb{R}$ be a value function assigning each state $s \in S$ a real value. An optimal value function V^* is a value function which is the single solution of the *Bellman equation*

$$V^*(s) = R(s) + \gamma \cdot \max_{a \in A} \sum_{s' \in S} T(s' \mid s, a) \cdot V^*(s') \quad \forall s \in S. \tag{3.1}$$

Value $V^*(s)$ then corresponds to the expected reward received if the agent started the play in state s.

3.1.1 Value iteration

An iterative variant of this system of non-linear equations is called the *Bellman operator* and the computation of a new value for state $s \in S$ by applying this operator on the old value is called the *Bellman update*.

$$V_{t+1}(s) = R(s) + \gamma \cdot \max_{a \in A} \sum_{s' \in S} T(s' \mid s, a) \cdot V_t(s') \quad \forall s \in S. \tag{3.2}$$

More specifically, this operator takes into account the utility values of all states adjacent to the current state and selects the neighbouring state with the maximal utility to compute the new value.

Definition 3.2. Let $f : X \to X$ be a mapping and $||.||$ be a norm defined on the vector space X. Let $0 \leq \delta < 1$. If the condition (Inequality 3.3) holds, the mapping f is a contraction.

$$||f(a) - f(b)|| \leq \delta ||a - b|| \qquad \forall a, b \in X \tag{3.3}$$

Very important fact is, that the Bellman operator is a contraction (Definition 3.2), which ensures that the algorithm converges.

Algorithm 3.1 Value iteration for MDPs

```
Input: an MDP $(S, A, T, R)$, $\epsilon$
Output: $\epsilon$-approximation of $V^*$
 1: Initialize $V_1(s) \leftarrow 0 \quad \forall s \in S$
 2: Set $t = 1$
 3: repeat
 4:     $\Delta \leftarrow 0$
 5:     for $s \in S$ do
 6:         $V_{t+1}(s) \leftarrow R(s) + \gamma \max_{a \in A} \sum_{s' \in S} T(s' \mid s, a) \cdot V_t(s')$
 7:         $\Delta \leftarrow \max \{\Delta, |V_t(s) - V_{t-1}(s)|\}$
 8:     end for
 9:     $t \leftarrow t + 1$
10: until $\Delta \leq \epsilon$
11: return $V$
```

The *value iteration* (Algorithm 3.1) then iterates through all states and applies the so-called Bellman update to each of them, which causes the immediate rewards to propagate to other states. When the new value is close enough to the current value for every state (the maximal difference of these values from all states is stored in variable Δ in (Algorithm 3.1)), the value iteration has converged to its optimal state. We say, that it reached an equilibrium, which is an important notion in games as well. The optimal policy is then extracted as simply selecting the state with the highest assigned value from value iteration.

3.1.2 Policy iteration

For the sake of completeness, we briefly mention another iterative algorithm for solving MDPs called the *policy iteration*. As opposed to value iteration, the policy iteration consists of two altering phases. First is policy evaluation, which evaluates current policy in each state. Second is policy improvement, which in each state uses a modified Bellman update and looks one step ahead and tries to find better action to play. If no action assignment was changed, the found policy optimal.

3.2 POMDP

In the previous chapter (Section 2.2), we defined a notion of an α-vector as function linear in belief b, which represents an n-step policy plan, where n is the time step of the α-vector's creation. The motivation behind this idea was to tackle the intractability of the infinite belief space, which would render the standard MDP solution methods useless. If we redefined POMDP as an MDP over the belief state, as mentioned in (Section 2.2), it would imply applying the Bellman operator infinitely many times in each iteration. The definition of α-vectors partly removes this problem as the number of α-vectors depends on the number of possible actions which is finite. Now, we can redefine value iteration from MDPs for POMDPs using this notion of α-vectors.

3.2.1 Value iteration

To perform the exact value iteration algorithm we can keep a set of α-vectors, denoted Γ. In each iteration t, new α-vectors are added and these correspond to all t-step policies for all states. Thus, each such iteration adds all combinations of previous $(t-1)$-step α-vectors and possible actions to the Γ set. Because α-vectors represent values for n-step policies, the value function in a belief point b is then computed as piecewise maximum over the set Γ

$$V(b) = \max_{\alpha \in \Gamma} \alpha(b) \tag{3.4}$$

and this way the best n-step plan is selected [17]. Even though, α-vectors solved the issue with infinite state-space, the size of Γ grows very rapidly and causes this method to be unscalable.

The size can be partly reduced be removing *dominated* α-vectors. An α-vector becomes dominated, when there always exist some other α-vector with higher value over the whole belief space. This means, that this vector is never optimal and will never be relevant to the solution of the game and thus can be removed from the Γ set. Nevertheless, this *pruning* of Γ is not enough to completely solve the poor scalability. Hence, another approach must be used.

3.2.2 Point based value iteration

The problem in the scalability comes mainly from the size of the belief space which is infinite. The idea behind the PBVI method is to represent the entire belief space with a finite set of belief points. Then, the Bellman update is performed only on elements of this finite set. Moreover, an α-vector is added to Γ only if it is the best for any of those belief points. This way, we have a coarser approximation of the real value, but now the number of α-vectors is bounded by the size of the set of belief points. On the other hand, the quality of the found solution depends on the selected belief points, and it is not a simple task to select which belief points lead to a good approximation. Similar idea about belief points is used in the HSVI algorithm described in the next part (Section 3.2.3).

3.2.3 Heuristic search value iteration

There are methods to approximate the value function without the need to keep all α-vectors and thus scale better. One of these methods is *Heuristic search value iteration*[8]. Another algorithm with the same name and similar idea will be presented in later sections to solve partially observable games.

This method keeps two piecewise linear and convex functions, *lower* and *upper bound*, where the lower bound is formed by a set of α-vectors, while the upper bound consists of values in belief points, from which is then computed a lower convex hull. During the course of the algorithm, new points and α-vectors are added to these sets and making the bounds closer to each other and thus making the approximation more precise. This operation is called *point-based update*.

Points to be updated are selected using a heuristic function, for example, selecting the best action based on the upper bound. We want to make the *gap* between the bounds tighter, so it makes sense to select the action with the highest value in the upper bound and try to push it lower. Then, both bounds are updated at the same point. When some α-vector or point in the upper bound becomes dominated, it can be removed and thus the size of those two sets can be reduced during the search.

After the distance between the two bounds is smaller than required threshold ϵ, the algorithm terminates and the ϵ-approximation is found.

3.3 Stochastic games

Similarly to the *Markov decision processes*, there exist versions of well-known iterative algorithms to find the approximate solution of the game, namely versions of *value iteration* and *policy iteration*. In this section, we will focus on the former. Also, employing some methods from *reinforcement learning* and *multi-armed bandits* can lead to good approximations, as well. Those will be discussed in the next section.

3.3.1 Value iteration

The value iteration is a well-described method used to solve *Markov decision processes*. It can be modified to solve stochastic games, too.

Value function

Let $G = (S, A_1, A_2, T, R, \gamma)$ be a stochastic game. As mentioned in (Section 2.4.2), the potentially mixed equilibrium strategies depend only on the current state s. Therefore, we can define a strategy of a player i as $\pi_i : S \to \Delta(A_i)$, which maps a probability distribution over possible actions A_i of the player i to each state $s \in S$.

We define a mapping $V : S \to \mathbb{R}$, which assigns a real value to every state $s \in S$. Then, the *value function* V^* is a mapping V, where the value $V^*(s)$ is the overall discounted reward received by player 1 if the game started from the state s. This definition of value function indicates how good it is for players to move into each state and can be used later to derive optimal strategies. Especially useful is that the mapping V^* can be expressed by equation [17]:

$$V^*(s) = [HV^*](s) = \max_{\pi_1 \in \Delta(A_1)} \min_{\pi_2 \in \Delta(A_2)} \mathbb{E}_{a_1 \sim \pi_1(s), a_2 \sim \pi_2(s)} [R(s, a_1, a_2) + \\ + \gamma \sum_{s' \in S} T(s' \mid s, a_1, a_2) \cdot V^*(s')] \quad (3.5)$$

The backup operator H is a contraction, therefore by iteratively applying it on an arbitrarily initialized V, it provably converges to its fixed point V^*[20]. Moreover, applying the backup operator H in a state s is the same as determining the value of the game of a corresponding *matrix game*, game in normal form. The description of the matrix game follows.

Stage game

Definition 3.3. Let $(S, A_1, A_2, T, R, \gamma)$ be a stochastic game and V be some value function. A matrix game u for a state $s \in S$ is then defined as:

$$u(a_1, a_2) = R(s, a_1, a_2) + \gamma \sum_{s' \in S} T(s' \mid s, a_1, a_2) \cdot V(s') \quad \forall a_1 \in A_1, \forall a_2 \in A_2 \tag{3.6}$$

Suppose we have a game in normal form for state $s \in S$ of the stochastic game G defined as above. Then, the value in the game table for an action profile $a = (a_1, a_2)$ is equal to the sum of the immediate reward received by playing action profile a in state s and the expected discounted reward received in all the next moves.

In the context of stochastic games, this matrix game is often called a *stage game* and can be solved by linear programming. Here is the linear program solving the game from the perspective of the maximizing player (LP 3.1). It maximizes over all mixed strategies π_1 of

$$\begin{aligned}
\text{maximize} \quad & v && && \text{(3.7a)}\\
\text{subject to} \quad & \sum_{a_1 \in A_1} u(a_1, a_2) \cdot \pi_1(a_1) \geq v && \forall a_2 \in A_2 && \text{(3.7b)}\\
& \sum_{a_1 \in A_1} \pi_1(a_1) = 1 && && \text{(3.7c)}\\
& \pi_1(a_1) \geq 0 && \forall a_1 \in A_1 && \text{(3.7d)}
\end{aligned}$$

Figure 3.1: LPSGMAX(u)

player 1 and value of the game v. Because we defined strategy as a distribution over actions $\Delta(A_i)$, the constraints (3.7c) and (3.7d) ensure that π_1 is in fact a probability distribution. Constraint (3.7b) finds those pure strategies of the opponent for which the expected utility against the player's mixed strategy π_1 is at least v. Those pure strategies for which the expected utility is equal to v form the support of the opponent's best response. Moreover, the value v is maximized, which influences the choice of the player's mixed strategy π_1.

A linear program LPSGMIN(u) solving the game from the perspective of the minimizing player can be constructed in a similar sense.

Algorithm

Now, we are ready to present the used *value iteration* algorithm (Algorithm 3.2).

Input to the algorithm is the stochastic game G and precision parameter ϵ. The precision parameter limits the allowed gap (Definition 3.4) between the last two approximations V_{t-1} and V_t so that $d(V_{t-1}, V_t) < \epsilon$.

Definition 3.4. Let V, V' be value functions. Then, the gap between V and V' is defined as

$$d(V, V') = \max \left\{ |V(s) - V'(s)| \;\middle|\; \forall s \in S \right\}. \tag{3.8}$$

Informally, this implies that the distance between approximate and optimal values in each state is at most ϵ. The output is then the ϵ-estimate of the real value function V^*.

Algorithm 3.2 Value iteration for stochastic games

```
Input: $G = (S, A_1, A_2, T, R, \gamma)$, $\epsilon$
Output: $\epsilon$-approximation of $V^*$
 1: Initialize $V(s) \leftarrow 0 \quad \forall s \in S$
 2: Initialize $\Delta \leftarrow \infty$
 3: while $\Delta \geq \epsilon$ do
 4:     $\Delta \leftarrow 0$
 5:     $V_{\text{prev}} \leftarrow V$
 6:     for $s \in S$ do
 7:         $V(s) \leftarrow$ solve LPSGMAX($u$) in $s$ (LP 3.1)        ▷ for $u$ see (Definition 3.3)
 8:         $\Delta \leftarrow \max\{\Delta, |V(s) - V_{\text{prev}}(s)|\}$
 9:     end for
10: end while
11: return $V$
```

Any value function can be used as an initial estimate. For simplicity, we use 0, but it could be initialized to any other value, for example, to maximal/minimal discounted reward. The algorithm always converges, but some initializations will converge faster than others. The initial gap Δ is initialized to an arbitrary value bigger than ϵ, only to first enter the main while loop.

In each iteration, the current gap is set to 0 and the approximation of V from the previous iteration is saved for comparison. Then, we perform a value update for each state $s \in S$. The update consists of building a stage game u in current state s and with the last approximation V according to definition (Definition 3.3), getting the optimal value from the linear program LPSGMAX(u) (LP 3.1) for the previously built stage game u and updating the current maximal gap Δ. After the algorithm updates value for every state, it checks the maximal gap Δ. If $\Delta < \epsilon$ holds, the algorithm terminates and returns the approximation of V from the last iteration, otherwise, it continues with the next iteration.

Optimal strategies. After the approximation is returned, it can be used to extract the strategies of both players. It is sufficient to solve both linear programs LPSGMAX(u), resp. LPSGMIN(u), for each state s with the returned value function V. This time, however, the values of variables $\pi_1(a_1)\ \forall a_1 \in A_1$, resp. $\pi_2(a_2)\ \forall a_2 \in A_2$, are used instead of the value of the game v.

Performance

Even though the algorithm is very simple and provably converges to an equilibrium, it is not as efficient. As opposed to Markov decision processes, where the update is performed only as finding the currently best action (pure strategy) in each state, here we need to solve as many linear programs as states in the game. Moreover, this procedure is repeated until the desired precision is met.

Linear programs can be solved in polynomial time, but the necessary number of them can grow rapidly, which makes Markov games more difficult to solve than Markov decision processes. And while in stochastic games it is not a real issue, in the partially observable stochastic games this becomes a crucial cause of unscalability.

In (Chapter 5), we propose other solution methods which do not require linear programs and compare them with the value iteration.

3.4 OS-POSGs

Methods to solve OS-POSGs combine the practices from *stochastic games* together with *partially observable Markov decision processes.* There exists an exact algorithm to solve this class of games, but similarly as in POMDPs, due to the dimension of the belief space it is practically intractable. Thus, an approximate approach is used to bypass this intractability and obtain at least a rough solution.

3.4.1 Value function

The value of the game corresponds with the discounted utility which is guaranteed to the unobserving player 1 by playing against a best-responding opponent 1, when starting in the initial belief b^{init}.

Definition 3.5. The optimal value function for an OS-POSG is a mapping $V^* : \Delta(S) \to \mathbb{R}$ assigning a real value to every belief point (a distribution over states) defined as follows [17]

$$V^*(b) = \sup_{\sigma_1 \in \Sigma_1} \inf_{\sigma_2 \in \Sigma_2} \mathbb{E}_{b,\sigma_1,\sigma_2} \text{Disc}^\gamma,$$

where Disc^γ are the discounted rewards sum.

In other words, it corresponds to the expected utility received by player 1 when starting in the initial belief $b \in \Delta(S)$ and both players employ their respective optimal behavioural strategies.

Because the value function V^* is defined as an expectation over a discounted sum with discount factor $0 < \gamma < 1$, which converges to a single value, it is bounded. Thus, we can define two values, the upper bound U and lower bound L, for which holds $L \leq V^*(b) \leq U \,\forall b \in \Delta(S)$. The bounds are defined as

$$L = \min_{s \in S, a_1 \in A_1, a_2 \in A_2} \frac{R(s, a_1, a_2)}{1 - \gamma} \tag{3.9}$$

$$U = \max_{s \in S, a_1 \in A_1, a_2 \in A_2} \frac{R(s, a_1, a_2)}{1 - \gamma} \tag{3.10}$$

α-vectors

Similarly to POMDPs, the belief state space is infinite and thus the value iteration technique for applying backup operators for each state as in MDPs or stochastic games cannot be used. Instead, we again use the notion of α-vectors in the same fashion as in POMDPs. Since the value of a first player's strategy σ_1 is linear in $b \in \Delta(S)$, which is proven in [17], we can use linear functions $\alpha : \Delta(S) \to \mathbb{R}$ to represent them. Moreover, a linear function defined in this way can be characterized by the values in the vertices of the belief simplex so from now on we represent the α-vector as values for each corner $s \in S$. The value of the α-vector in a belief $b \in \Delta(S)$ is thus computed

$$\alpha(b) = \sum_{s \in S} \alpha(s) \cdot b(s). \tag{3.11}$$

Be reminded, that an α-vector represents a n-th step plan, in this case a behavioural strategy.

Because of this representation and the fact, that the optimal value function V^* is convex and continuous, it can be approximated as a piecewise linear convex functions V. These are then created as a pointwise maximum

$$V(b) = \max_{\alpha\in\Gamma} \alpha(b) \quad \forall b \in \Delta(S) \tag{3.12}$$

where Γ is a finite set of α-vectors

$$\Gamma \subset \{\alpha : \Delta(S) \to \mathbb{R} \mid \alpha \text{ is linear}\}\,. \tag{3.13}$$

Information sets

In many domains, the dimension of belief can be reduced by introducing the *information sets*. We allow the player 1 to know his location in the environment but the position and actions of the opponent remain hidden to him. Thus, the space of space is split into partitions whose inner states are indistinguishable for the player 1, he knows only the current partition. This reduction of the size of the belief space helps to improve scalability and solve bigger instances, where applicable.

3.4.2 Exact algorithm

As in previous problems, we use the iterative approach to obtain the optimal value function V^*, because finding the optimal value in a belief point exactly is a hard problem. The main idea is again to start from some initial approximation and repeatedly apply a backup operator on it to get a refined more accurate approximation. For this purpose serve the so-called *Bellman operators* already known from the past sections. Here, we present the Bellman operator $[HV](b)$ for OS-POSGs as it was derived in [17].

In short, the Bellman operator is a special strategy composition, where an optimal stage strategy π_1 of player 1 is sought to maximally improve the current value function by adding another step.

Definition 3.6. Let $V : \Delta(S) \to \mathbb{R}$ be a convex continuous function and Γ be a convex set of α-vectors such as $V(b) = \sup_{\alpha\in\Gamma} \alpha(b) \quad \forall b \in \Delta(S)$. Let τ be the *belief update* as defined in (Definition 2.12). Then,

$$\begin{aligned}[HV](b) = \max_{\pi_1\in\Pi_1} \min_{\pi_2\in\Pi_2} \Bigg[& \mathbb{E}_{b,\pi_1,\pi_2}\left[R(s,a_1,a_2)\right] + \\ & +\gamma \sum_{(a_1,o)\in A_1\times O} \mathbb{P}\left[a_1,o\right] V(\tau(b,\pi_2,a_1,o)) \Bigg]\end{aligned} \tag{3.14}$$

Note that, thanks to the *minimax* theorem, the max and min can be switched to form equivalent formulations [15].

Similarly, as for Bellman operators for MDPs, POMDPs, etc., it can be proven, that it is a contraction (Definition 3.2) and thus converge to a unique fixed point, which is the optimal value function V^*. The equality $V^*(b) = [HV^*](b)$ holds $\forall b \in \Delta(S)$. Also, the operator does not depend on the set of α-vectors Γ and thus can be used for updates at any time and on arbitrarily initialized approximation.

Stage game

In stochastic games, applying the Bellman operator on the current approximation of the value function was equivalent to finding a Nash equilibria of a matrix game, called the *stage game*. This applies to the OS-POSGs too, and the corresponding stage game is defined as listed in

Definition 3.7. A stage game with a value function V, which is convex and continuous, and a belief $b \in \Delta(S)$ is a two-player zero-sum game. Let Π_1 be the strategy set of the maximizing player 1 and Π_2 of the minimizing player 2. The utility function is then

$$u(\pi_1, \pi_2) = \mathbb{E}_{b,\pi_1,\pi_2}\left[R(s, a_1, a_2)\right] + \gamma \sum_{a_1,o} \mathbb{P}_{b,\pi_1,\pi_2}[a_1, o] \cdot V(\tau(b, \pi_2, a_1, o)) \tag{3.15}$$

This game can be solved by a linear program thoroughly described in [17] together with its dual formulation. We exclude it because it is not essential for the topic of this thesis as we will focus on the approximate HSVI algorithm.

Value iteration

The exact value iteration algorithm works similarly as the one mentioned in context of stochastic games. It starts from an initial piecewise linear and convex value function V_0 and then by iteratively solving stage games and creating new piecewise linear and convex value functions $V_i = HV_{i-1}$ the algorithm converges to the optimal V^*.

It is known that the exact algorithm designed for POMDPs can solve only very small instances and since OS-POSGs are a more complex problem it can be estimated that this would hold also for this domain. Thus, we focus more on the non-exact Heuristic search value iteration described in the next section.

3.4.3 Heuristic search value iteration

This approximate method is inspired by the algorithm for POMDPs with the same name and is described in [9]. It improves the scalability of the exact value iteration method while sacrificing precision of the found result. Here, we shortly describe the original algorithm from [9] and later in (Chapter 5) we introduce a modification of this algorithm employing the multi-armed bandit framework.

Main idea

The basic concept of HSVI is keeping two value functions, lower bound V_{LB}^{Γ} and upper bound ub, which both bound the unknown optimal value function V^*. In other words, the inequality $V_{LB}^{\Gamma} \leq V^* \leq V_{UB}^{\Upsilon}$ holds for every belief point $b \in \Delta(S)$.

Then, in each step of the algorithm the bounds are improved until some desired precision in the initial belief b^{init} is reached. The improvement lies in recursively solving stage games, reaching new belief points and using this information to create better stage strategies and thus finding better behavioural strategies. After some recursion depth, the search is restarted from the initial belief b^{init}.

The precision is specified by $\epsilon \in (0, +\infty)$ forcing the value function $V(b^{\text{init}})$ returned by the algorithm to be inside the ϵ-neighbourhood around $V^*(b^{\text{init}})$.

Lower and upper bound

Both bounds are represented by finite sets of elements, from which can be constructed the corresponding piecewise linear convex value function. However, type of the elements are different for each bound.

Lower bound $V_{LB}^{\Gamma} : \Delta(S) \to \mathbb{R}$ is represented in the same way as discussed before in this section and as was used for lower bound in HSVI for POMDPs. That is by a set of α-vectors denoted Γ from which is selected a point-wise maximum as $V_{LB}^{\Gamma}(b) = \max_{\alpha \in \Gamma} \alpha(b)$. More details on selecting the maximum is in (Section 3.4.1). To understand the initialization and point-based updates be reminded that α-vector corresponds to a n-th step policy of an agent.

The point-based update of the lower bound is performed by adding new α-vectors into Γ. These are created from behavioural strategies prolonged by one stage strategy of the players. This way, when a strategy with higher value is found, the corresponding α-vector is selected in the point-wise maximum over Γ. That pushes the lower bound V_{LB}^{Γ} upwards.

Initialization of the lower bound V_{LB}^{Γ} is done in a very straightforward way and that is by creating an α-vector corresponding to uniform strategy of the player, meaning that each action $a_1 \in A_1$ is played with probability $\frac{1}{|A_1|}$ and conversely for player 2.

Upper bound $V_{UB}^{\Upsilon} : \Delta(S) \to \mathbb{R}$ is represented as a lower convex envelope of a set of points $\Upsilon = \{(b_i, y_i) \mid 1 \leq i \leq k,\ b_i \in \Delta(S),\ y_i \in \mathbb{R}\}$. Every $(b_i, y_i) \in \Upsilon$ bounds the optimal V^* from above in the belief point b_i. In [17] is shown, that this representation of the upper bound follows the requirements on convexity. The convex envelope can be also computed by linear programming.

The refinement of the upper bound V_{UB}^{Υ} is again done by adding new points (b, y) into the Υ set. The new point is retrieved by solving the stage game as well as in the case of lower bound, but this time with simpler linear program as composition of values is not necessary. Similarly, due to computing only the lower convex hull of the points, the value in a given belief point b can only decrease and so the upper bound is pushed downwards.

The initialization of the upper bound V_{UB}^{Υ} is more complex. The original OS-POSG game is transformed into a stochastic game by revealing all the information to the player 1 too. All the other parts remain the same as in the OS-POSG. Then, the value iteration algorithm can be used to compute the optimal values $V^*(s) \forall s \in S$, which then create one point $(b_s, V^*(s))$, where b_s is a pure belief in state $s \in S$. By removing the full information, the player surely cannot receive higher reward and so this is a good initial value function.

From the above follows that by refining the bounds by point-based updates, the lower bound increases while the upper bound decreases and thus the gap gets tighter.

The algorithm

In this subsection, we present the HSVI algorithm (Algorithm 3.3) for OS-POSGs. The correctness and termination is described in [17], here we provide only a description on which we will build in next sections.

After the initialization of the bounds (Section 3.4.3), the algorithm repeatedly calls procedure EXPLORE($b^{\text{init}}, 0$) until the *excess gap* in b^{init} is zero or below. The excess gap is computed as

$$\text{excess}_t(b_t) = V_{UB}^{\Upsilon}(b_t) - V_{LB}^{\Gamma}(b_t) - \rho(t). \tag{3.16}$$

Algorithm 3.3 HSVI algorithm for OS-POSGs

```
Input: game $G$, initial belief $b^{\text{init}}$, precision requirement $\epsilon > 0$, discount factor $\gamma$, neighbouring
    parameter $D$
Output: bounds $V_{LB}^{\Gamma}$ and $V_{UB}^{\Upsilon}$ satisfying $V_{UB}^{\Upsilon}(b^{\text{init}}) - V_{LB}^{\Gamma}(b^{\text{init}}) \leq \epsilon$ and the corresponding
    sets $\Gamma$ and $\Upsilon$
 1: initialize $V_{LB}^{\Gamma}$ and $V_{UB}^{\Upsilon}$                                    ▷ see (Section 3.4.3)
 2: while $\text{excess}_0(b^{\text{init}}) > 0$ do
 3:     EXPLORE($b^{\text{init}}, 0$)
 4: end while
 5: return $V_{LB}^{\Gamma}$, $V_{UB}^{\Upsilon}$, $\Gamma$, $\Upsilon$
 6: procedure EXPLORE($b_t, t$)
 7:     $(\pi_1^{\text{LB}}, \pi_2^{\text{LB}}) \leftarrow$ strategies of the Nash equilibrium of $[HV_{LB}^{\Gamma}](b_t)$
 8:     $(\pi_1^{\text{UB}}, \pi_2^{\text{UB}}) \leftarrow$ strategies of the Nash equilibrium of $[HV_{UB}^{\Upsilon}](b_t)$
 9:     point-based updates of $V_{LB}^{\Gamma}$ and $V_{UB}^{\Upsilon}$                       ▷ see (Section 3.4.3)
10:     $(a_1^*, o^*) \leftarrow$ select best pair according to the forward exploration heuristic
11:     if $\mathbb{P}_{b,\pi_1^{\text{UB}},\pi_2^{\text{LB}}}[a_1^*, o^*] \cdot \text{excess}_t(\tau(b_t, \pi_2^{\text{LB}}, a_1^*, o^*)) > 0$ then
12:         explore($\tau(b_t, \pi_2^{\text{LB}}, a_1^*, o^*), t+1$)
13:         point-based updates of $V_{LB}^{\Gamma}$ and $V_{UB}^{\Upsilon}$                   ▷ see (Section 3.4.3)
14:     end if
15: end procedure
```

Hence, when the excess_t is positive, the approximation error is bigger than the desired precision computed by $\rho(t)$ from ϵ. The function $\rho : \mathbb{N} \to \mathbb{R}_+$ generates an increasing sequence starting from ϵ and thus allows for higher gap between the bounds as the given trial goes in farther stages of the game. The sequence ρ is defined as

$$\rho(0) = \epsilon \qquad \rho(t+1) = \frac{\rho(t) - 2\delta D}{\gamma}, \tag{3.17}$$

where $\delta = \frac{U-L}{2}$, D is an arbitrary neighbouring constant satisfying $0 < D < \frac{(1-\gamma)\epsilon}{2\delta}$ and γ is the discount factor. Even though the gap in farther stages is allowed to be higher than ϵ, in [17] is shown, that the algorithm converges to the ϵ approximation in the initial belief, when the above conditions hold. Note that for the initial call of EXPLORE procedure, the $\text{excess}_0(b^{\text{init}})$ is equivalent to a condition $V_{UB}^{\Upsilon}(b^{\text{init}}) - V_{LB}^{\Gamma}(b^{\text{init}}) \leq \epsilon$.

The EXPLORE($b^{\text{init}}, 0$) procedure recursively searches for behavioural strategies and one call of this procedure corresponds to a single trial of playing the game. The recursive call moves through a sequence of belief points starting from b^{init} and for every one tries to improve the value bounds in the selected belief point b_t.

First, it finds the stage strategies (π_1, π_2), i.e. Nash equilibrium, of the stage game (Section 3.4.2) for both the upper and lower bounds in the belief b_t of current recursive call resulting in two strategy profiles. With these stage strategies, the point-based update of both bounds is performed, i.e. a new α-vector and a new point (b_t, y) are added into their respective set Γ or Υ.

After the update a new belief point $b_{t+1} = \tau(b_t, \pi_2, a_1^*, o^*)$ is derived as described in

(Section 2.12). The pair (a_1^*, o^*) is selected by a *forward exploration heuristic*.

$$(a_1^*, o^*) = \underset{(a_1,o)\in A_1\times O}{\operatorname{argmax}} \mathbb{P}_{b,\pi_1^{\mathrm{UB}},\pi_2^{\mathrm{LB}}}[a_1, o] \cdot \mathrm{excess}_{t+1}(\tau(b_t, \pi_2^{\mathrm{LB}}, a_1, o)) \tag{3.18}$$

This heuristic approach aims to find the best action and observation of the player 1, so that the next belief point has the biggest *excess* gap between bounds and thus promises the biggest improvement. For this purpose serves the *weighted excess gap*, where the excess in give belief point is multiplied by a probability of occurrence of the given pair (a_1, o).

If the excess in the next belief given the (a_1^*, o^*) is non-positive, the algorithm stops searching new stages and returns to the previous belief again performing point-based updates for the belief points with the new improved bounds. Otherwise, it proceeds to the next stage.

If the gap between bounds in b^{init} is wider than ϵ after emerging from the recursion, the search is repeated with a new game play trial.

Performance

Even though, the algorithm is proven to converge and is an improvement from the exact value iteration approach, it still uses linear programs which can become large and thus slow the algorithm down. In the (Section 5.3) we propose the usage of multi-armed bandits as an alternative exploration method to avoid linear programming and possibly scale better for larger problems.

Chapter 4
Multi-armed bandits

At the beginning of the thesis (Section 2.1), we discussed the Markov decision process as one of the models used for describing a single-agent environment. In MDPs, an agent continuously chooses actions by which he moves through many states of the environment and receives rewards throughout the process. Moreover, this model can be used for learning agent's optimal actions and policies in each state.

This section is about examining MDPs where the set of all states S is a singleton, i.e. consists of only one state $s \in S$. This subset of all MDPs is often referred to as *Multi-armed bandits* and many algorithms based on this model are very useful for solving more complex problems. The bandits are a well-known tool of reinforcement learning enabling a single agent to gain information about the environment and leverage it for its advantage. Here, we describe the basic concepts of bandit learning and provide multiple specific algorithms designed for this purpose.

This section and the standard versions of the bandit algorithms are based on a book *Introduction to Multi-Armed Bandits*[10].

4.1 Basics

As stated above, a Multi-armed bandit problem is essentially a Markov decision process of a single state $s \in S$ with many, possibly infinitely many, different available actions $a_i \in A$. In the context of bandit algorithms, these actions are called 'arms' and hence the adjective *multi-armed*. We will use 'arms' and 'actions' interchangeably within this chapter. There

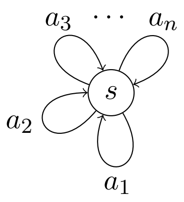


Figure 4.1: Model of a multi-armed bandit

exists a variety of different bandit algorithms, which can be used in many contexts. By using them, one can decompose a complex problem into many smaller bandit problems, which can learn independently on the others. The means and methods of bandit learning are presented in the next part.

4.2 Bandit learning

Let's suppose that the rewards of actions depend solely on the current state. In other words, the rewards are identically independently distributed. Thus, a bandit learning a particular state can learn independently on the other bandits.

The bandit learning takes place in discrete time steps $t = 1, \ldots$. In each step t, the bandit selects and plays an arm $a^t \in A$ according to some decision rule and receives a reward r^t. This decision rule, or sometimes selection rule, is the main distinction point among the many multi-armed bandit algorithms. The bandit stores the received rewards, updates its inner state and uses them for the next arm selections.

As a basis for the arm selection process, it is convenient to have some measure of quality of a function. For this purpose, we define so-called action function $q^* : A \to \mathbb{R}$

$$q^*(a) = \mathbb{E}[r^t \mid a^t = a] \quad \forall a \in A. \tag{4.1}$$

However, neither the optimal values $q^*(a)$, nor the distribution the rewards are drawn from, is not known by the algorithm. Thus, instead of the true values for actions, it estimates these values by averaging received rewards over time. For this purpose we define an action function estimate $Q^t : A \to \mathbb{R}$ in time step t, corresponding to the average reward gained by playing some action $a \in A$ so far.

Definition 4.1. Let A be a set of all arms of a bandit algorithm and $t \in \mathbb{N}$ be the current time step. Then, $Q : A \to \mathbb{R}$ is defined as

$$Q^t(a) = \frac{\sum_{i=1}^t r^i \cdot [\![a^i = a]\!]}{\sum_{i=1}^t [\![a^i = a]\!]} \quad \forall a \in A, \tag{4.2}$$

the indication function $[\![.]\!]$ is defined as

$$[\![a^i = a]\!] = \begin{cases} 1 & a^i = a \\ 0 & a^i \neq a \end{cases} \tag{4.3}$$

and where a^i is an action taken in time step $i \in 1, \ldots, t$ and r^i is reward gained by playing this action a^i.

Even though this formula is clear and simple it is not very computationally efficient, because it would require evaluating two sums every time.

First, we need to derive an incremental computation of an arithmetic average of values. Let X_m be an arithmetic average of first $m-1$ values $x_1, \ldots, x_{m-1}$, where

$$X_m = \frac{x_1 + x_2 + \cdots + x_{m-1}}{m-1}. \tag{4.4}$$

Then, the arithmetic average of values received before time step $m+1$ can be computed as

$$\begin{aligned} X_{m+1} &= \frac{1}{m}\sum_{i=1}^{m} x_i = \frac{1}{m}\sum_{i=1}^{m-1} x_i + \frac{1}{m}x_m = \frac{m-1}{m-1}\frac{1}{m}\sum_{i=1}^{m-1} x_i + \frac{1}{m}x_m = \\ &= \frac{m-1}{m}\frac{1}{m-1}\sum_{i=1}^{m-1} x_i + \frac{1}{m}x_m = \frac{m-1}{m}X_m + \frac{1}{m}x_m = \\ &= \frac{m}{m}X_m - \frac{1}{m}X_m + \frac{1}{m}x_m = X_m + \frac{x_m - X_m}{m} \end{aligned} \tag{4.5}$$

Now, we can describe the computation of the action function estimate Q^t. Instead of keeping a full action-reward history, a bandit algorithm keeps track of the current average of received values so far in $Q^t(a)$, $\forall a \in A$ and also how many times have been each action selected in a vector $n(a) \in \mathbb{N}^{|A|}$. On receiving reward r^t after playing action a^t, the trial function is trivially updated as

$$n^{t+1}(a^t) = n^t(a^t) + 1. \tag{4.6}$$

The update $Q^{t+1}(a^t)$ can be computed by employing the previously derived incremental averaging rule for the action selected at this time round.

$$Q^{t+1}(a) = \begin{cases} Q^t(a) + \frac{r^t - Q^t(a)}{n^{t+1}(a)} & a^i = a \\ Q^t(a) & \text{otherwise} \end{cases} \quad \forall a \in A \tag{4.7}$$

Notice that except the changed notation, the update is the same as the result of (Equation 4.4).

4.2.1 Exploration-exploitation trade-off

Action a^t for which it holds

$$a^t \in \operatorname*{argmax}_{a \in A} Q^t(a), \tag{4.8}$$

is usually called the *greedy action*. In some sense, it is the best of all possible actions with respect to the received rewards up to the time step t.

However, always selecting the greedy action, which could be then called a *greedy bandit*, can cause problems or slow down the convergence of the function Q^t to the optimal action function q^* and thus making it harder for the bandit algorithm to find the optimal actions to play. To improve convergence to the real values and prevent getting stuck in some local optima, the *exploration-exploitation trade-off* has to be taken into account.

Exploration is a mechanism when a multi-armed bandit explores possibilities, i.e. actions, that either have not yet been selected or the so far collected average reward is much lower than for other actions and thus are not selected by the greedy method. *Exploitation*, on the other hand, is abusing so far learned values to take as much profit as possible. In other words, exploitation is selecting the action, which is thought to provide the highest reward. It is essential for a well-performing bandit algorithm to balance these two aspects in some way. If it was too much exploration, by selecting actions with values very different from the real value, it would take a long time to converge and find the optimum. If it was too much exploitation, it would get likely stuck in some local optima and it would struggle to find the best actions to select.

Now we have everything ready to present some common bandit algorithms with several arm selection rules, which handle the exploration-exploitation trade-off differently.

4.3 Stochastic bandits

Stochastic bandits are the simplest multi-armed bandit algorithms mentioned in this thesis. As mentioned before, the rewards are assumed independent and identically distributed as well as bounded.

At first, the Q function and function counting action trials n are initialized.

Algorithm 4.1 Initialize a bandit

```
Input: action set $A$
1: $Q(a) \leftarrow 0 \quad \forall a \in A$
2: $n(a) \leftarrow 0 \quad \forall a \in A$
```

In each time step t the algorithm selects an arm a^t. After, the standard versions of these bandit algorithms observe only the received reward r^t, which is then called *bandit feedback*. However, in the context of games, we later construct *observable* variants (Section 5.1.1) for

Algorithm 4.2 The **receive** function for *bandit feedback*

```
Input: $n$, $Q$, action $a^t$, reward $r^t$
1: $n(a^t) \leftarrow n(a^t) + 1$
2: $Q(a^t) \leftarrow Q(a^t) + \frac{r^t - Q(a^t)}{n(a^t)}$
```

each mentioned stochastic bandit algorithm, which also observes the action taken by the opponent a'_t. Later in this thesis, we will compare the observable and standard alternatives and see if the information about the opponent's selections can be leveraged and will lead to faster convergence.

Now, we define the standard stochastic bandit algorithms. The first two mentioned bandits are often termed as *non-adaptive* because they do not adjust the amount of exploration during the course of learning.

4.3.1 Best of N

Algorithm 4.3 Best of N bandit

```
Input: action set $A$, $N \in \mathbb{N}$
1: try arms uniformly at random until each one was tried exactly $N$ times
2: select the action greedily as $a_{\text{greedy}} \leftarrow \text{argmax}_{a \in A} Q(a)$
3: play $a_{\text{greedy}}$ forever
```

The first and simplest bandit algorithm described in this thesis is a *Best of N* bandit (Algorithm 4.3), sometimes known as an *uniform* bandit. It has a single parameter $N \in \mathbb{N}$ which influences the degree of exploration of each action. In short, the bandit uniformly tries actions and accumulates the average received rewards until each action was tried exactly N times. After all actions has been tried N times, exploration ends and from that point, the algorithm only exploits. The exploitation phase is very simple as the agent selects the greedy action a_{greedy} (Definition 4.8) and plays it forever.

4.3.2 ϵ-greedy

The second bandit algorithm, which we will discuss here is an ϵ-*greedy* bandit (Algorithm 4.4). In contrast to the Best of N bandit, ϵ-greedy does not separate exploration and exploitation phases so strictly but does it in a more clever way founded on a single parameter $\epsilon \in [0, 1]$. At the beginning of every action selection process, the algorithm probabilistically chooses

Algorithm 4.4 **ϵ-greedy** bandit

```
Input: action set $A$, $\epsilon \in [0,1]$
1: with probability $\epsilon$ sample $a^t$ uniformly from $A$
2: with probability $(1-\epsilon)$ select the greedy action $a^t \leftarrow \text{argmax}_{a \in A} Q(a)$
```

between two possibilities. With probability ϵ, it chooses exploration and samples action a^t from uniform distribution over the set A. Otherwise, with probability $(1-\epsilon)$, it selects greedy action based on saved function Q^t as in (Definition 4.8).

The parameter ϵ thus controls how often the algorithm explores and how often it exploits. A typical value of ϵ is 0.1, which means that exploration is expected to occur every 10^{th} round. However, this value can be tuned to a specific problem. If $\epsilon = 0$, then the bandit degrades to a *greedy* bandit, which never explores and only chooses the action with highest accumulated rewards Q^t.

As opposed to Best of N and ϵ-greedy stochastic bandits, both non-adaptive multi-armed bandits, the following two algorithms are *adaptive*, because exploration of an arm depends on a *confidence* that the accumulated average rewards are a good approximation of the real values. For the purpose of this and the next bandit, we need to define *confidence interval* and *confidence bounds*.

A confidence interval is a range around the estimated value and it holds that the true value lies in this range with high probability. The two limit points of this said interval are *confidence bounds*, *lower* and *upper*. We will use such confidence bounds as were derived in [10] using *Hoeffding Inequality*.

Definition 4.2. For each arm a at fixed time step t let $n^t(a)$ represent the number of selections of the arm a before time t and $Q^t(a)$ is the accumulated mean reward until t. We then define a *confidence radius* $r^t(a)$ as derived in [10]:

$$r^t(a) = \sqrt{\frac{2 \log t}{n^t(a)}}. \tag{4.9}$$

Let $\alpha \in \mathbb{R}, \alpha \geq 0$ be a parameter. The upper confidence bound, resp. lower confidence bound is thus defined as

$$\text{UCB}^t(a, \alpha) = Q^t(a) + \alpha \cdot r^t(a) \tag{4.10}$$

$$\text{LCB}^t(a, \alpha) = Q^t(a) - \alpha \cdot r^t(a) \tag{4.11}$$

Hence, the confidence interval of an estimate $Q^t(a)$ is $[\text{LCB}^t(a,\alpha), \text{UCB}^t(a,\alpha)]$.

Now we have everything ready to present the two *adaptive* stochastic bandit algorithms.

4.3.3 Successive elimination

Successive elimination bandit (Algorithm 4.5) tries all actions until it is confident, based on the computed confident intervals, that some action is the best. From that point, it plays only this best action forever. The bandit keeps a set of *active* arms, which is initialized to contain all possible actions. It iteratively selects actions from this active set one by one until it tries every arm. After each active arm was tried, it computes the confidence bounds $\text{LCB}(a, \alpha)$

Algorithm 4.5 Successive elimination bandit

```
Input: action set $A$, $\alpha \in \mathbb{R}_+$
1: set all actions as *active* arms

2: try each *active* arm once
3: deactivate every active arm $a$ for which $\text{UCB}^t(a, \alpha) \leq \text{LCB}^t(a', \alpha)$ for some other active
   arm $a'$
```

and $\text{UCB}(a, \alpha)$ for each arm a in current time step t. Then, if for some active arm a and some other active arm a' holds

$$\text{UCB}^t(a, \alpha) \leq \text{LCB}^t(a', \alpha) \tag{4.12}$$

the action a is *deactivated*, i.e. removed from the active set. The condition (Inequality 4.12) is often called the *deactivation rule*. After this rule is applied to each previously active arm, the algorithm starts again iteratively selecting arms from this newly constructed active set. This continues until only a single action remains. Then, the intervals need not be computed and only the remaining action is selected until the end.

The deactivation rule can be visualized on the confidence intervals. When the intervals of two different actions overlap, it cannot be said, that one is better than the other, with enough confidence. However, if the intervals are disjoint, the arm with a lower average reward simply cannot be better with high enough probability and thus can be deactivated.

The exploration parameter α in the computation of upper confidence bound values, resp. lower confidence bound values, influences how much one trial improves the confidence of the estimate. Higher values mean more trials are needed to gain enough confidence about the average of received values.

4.3.4 UCB

Algorithm 4.6 UCB bandit

```
Input: action set $A$, $\alpha \in \mathbb{R}_+$
1: always select $a^t \in \text{argmax}_{a \in A} \text{UCB}(a, \alpha)$
```

The idea behind *UCB* bandit, or fully *Upper Confidence Bound* bandit, is simpler than behind Successive elimination as it does not consider lower confidence bound at all, but it often performs better and is more widely used than Successive elimination. It does not deactivate arms, but every time step t it directly selects the action a with the highest $\text{UCB}(a, \alpha)$ value. In other words, UCB bandit selects the action, which is optimistically the best, or that has the best potential outcome based on the rewards received so far. As in successive elimination bandit, it has a single parameter $\alpha \in \mathbb{R}, \alpha \geq 0$, which controls the amount of exploration, specifically, how fast the confidence bounds get tighter with one trial.

4.4 Adversarial bandits

So far, all stochastic bandits were using so-called *bandit feedback*. In other words, they received only the reward gained by playing the selected action, but the potential rewards for other actions remained unknown. *Adversarial* bandits not only receive the reward for the chosen action, they also observe all the other rewards. This is called *full feedback*. However, the applications of bandits do not always enable receiving rewards for all actions, so later, a way to use bandit feedback to fit into the full feedback model will be shown.

Moreover, adversarial bandits do not assume i.i.d. rewards, instead they are arbitrarily chosen by an unknown virtual adversary, thus the name adversarial. For example, while UCB is designed to learn the best pure strategy, i.e. always play the single best action, the adversarial bandits strive to find suitable probability distributions over possible actions and thus learn mixed strategies. This phenomenon is discussed on examples in a later chapter.

4.4.1 Hedge algorithm

The main idea from this category of bandits, which is used in further introduced bandits, is expert advice. An expert can be viewed simply as a function, which gives its prediction, or advice, about the action to play. The algorithm has a set of experts E and in each time step decides which expert's advice to follow. For this purpose serves the *Hedge* algorithm.

The purpose of this algorithm is to evaluate performance of individual experts in a sense whether their advice was fulfilled or not and based on this adjust how often the particular expert is followed. To ensure some amount of exploration, the expert selection is done in a way of assigning weights to the experts and sampling the current expert from a distribution proportional to these assigned weights. In this setting, it is not a full bandit algorithm, it is more of a core algorithm for some other adversarial bandit which enwraps it and uses its weighting system to handle the experts.

Algorithm 4.7 Hedge algorithm

```
Input: the set of experts $E$, parameter $\epsilon \in \left(0, \frac{1}{2}\right)$
1: Initialize weights $w^1(e) \leftarrow 1 \quad \forall e \in E$
2: for $t \leftarrow 1, \dots$ do
3:     construct probability distribution $p^t(e) = \frac{w^t(e)}{\sum_{e' \in E} w^t(e')} \quad \forall e \in E$
4:     sample expert: $e \sim p^t(.)$
5:     observe rewards: $r^t(e) \quad \forall e \in E$
6:     adjust weights: $w^{t+1}(e) \leftarrow w^t(e) \cdot (1-\epsilon)^{-r^t(e)} \quad \forall e \in E$
7: end for
```

Input to this algorithm is the set of experts E and a single parameter $\epsilon \in \left(0, \frac{1}{2}\right)$, which determines how much do received rewards change the weights. The initial weight of every expert is set to $w^1(e) = 1$, $\forall e \in E$, so at the beginning, all bandits have the same probability of selection.

In each time step t a probability distribution p over the set of experts E is constructed as

$$p^t(e) = \frac{w^t(e)}{\sum\limits_{e' \in E} w^t(e')} \quad \forall e \in E \tag{4.13}$$

From this distribution p^t the algorithm samples an expert, whose advice will be used in this time step t. Based on the observed rewards for every expert $r^t(e)$, new weights are computed based on the arm they recommended.

$$w^{t+1}(e) = w^t \cdot (1-\epsilon)^{-r^t(e)} \quad \forall e \in E \tag{4.14}$$

In the cited textbook [10], this algorithm is defined for costs rather than for rewards, hence the extra minus sign before $r^t(e)$ in the exponent of weight update.

Now we present a bandit algorithm using Hedge to achieve balance between exploration and exploitation and provide good arm selection, while requiring bandit feedback, which is natural to many bandit use cases.

4.4.2 Exp3

First, we need to discuss how to adapt bandit feedback, so that is accepted by the Hedge algorithm, which requires full feedback. It is necessary because bandit feedback is more common in bandit algorithms as the environment rarely returns rewards for all possible actions, but only the actual outcome. This obstacle can be tackled by using *fake costs*.

Algorithm 4.8 Exp3 bandit

```
Input: action set $A$, $\epsilon \in \left(0, \frac{1}{2}\right)$, $\gamma \in \left[0, \frac{1}{2}\right)$
1: create set of experts $E \leftarrow \{e : e_a = a' \mid \forall a' \in A\}$
2: for $t \leftarrow 1, \ldots$ do
3:     retrieve expert probabilities $p^t$ from hedge
4:     sample an expert $e^t \sim \{p^t\}$
5:     with probability $(1-\gamma)$ use action $a^t = e^t_a$ recommended by sampled expert $e^t$,
       otherwise sample action $a^t$ uniformly at random
6:     observe immediate reward $r^t(a^t)$
7:     compute fake costs $\hat{r}^t(.)$ according to (Formula 4.15)
8:     return fake costs $\hat{r}^t(.)$ to hedge
9: end for
```

As said before, the *Exp3* adversarial bandit is built on top of the Hedge algorithm (Algorithm 4.7) and provides a transition from bandit feedback to fake costs. Experts in Exp3 are constructed trivially as for one possible action $a_i \in A$ exists exactly one expert $e_i \in E$, which always recommends said action a_i. No other experts are present in the set of all experts, except those mentioned above. In addition to the Hedge parameter $\epsilon \in \left(0, \frac{1}{2}\right)$, Exp3 takes a parameter $\gamma \in \left[0, \frac{1}{2}\right)$, which regulates additional exploration.

In each time step t, the algorithm receives probability p^t over experts from Hedge. Exploration is provided by a procedure similar to ϵ-greedy bandit. With probability γ the algorithm samples a random action uniformly from the set of all possible actions. Otherwise, it samples an expert e^t from the Hedge probability p^t and follows its advice e^t_a. Let the selected action be denoted a^t independent of how it was selected. From the received reward for the chosen (sampled or recommended) action the fake rewards $\hat{r}^t$ are computed by a formula

$$\hat{r}^t(e) = \begin{cases} \frac{r^t(a)}{\Pr[a^t = e^t_a | p^t]} & a^t = e^t_a, \\ 0 & \text{otherwise} \end{cases}. \tag{4.15}$$

These rewards are then returned to Hedge as real rewards to update its inner expert weights.

This time, exploration is regulated by two parameters. γ influences the more aggressive exploration with random actions, while ϵ manages selection of experts, which do not predict so well very often.

There exists an extension of the Exp3 algorithm, which is called Exp4 and differs in used experts. The user can define arbitrary experts to predict future actions and the number of experts does not even need to coincide with the number of possible actions. However, this bandit algorithm was not used in this thesis.

4.5 Summary

In this chapter, we focused on learning in an environment using multi-armed bandit algorithms. The most simple stochastic bandits were described in addition to the more complex adversarial bandit algorithm Exp3 (Section 4.4.2). Although, these methods are designed for a single agent, they can be used to play games as well. In the next section, we propose modifications to make them fit more into algorithms solving games, stochastic games specifically.

Chapter 5
New solution methods

In previous chapter (Chapter 3), solution methods for (partially observable) stochastic games were presented. These algorithms are proven to converge and find a solution or its approximation, however, they can be slow and thus unusable for very large domains.

In addition, in one of the first sections (Chapter 4), we examined the multi-armed bandit problem and some examples of algorithms solving this problem. These so-called bandits serve to decompose a big learning problem to smaller ones and learn more granularly.

In this chapter, we propose adjustments of the common definitions of some multi-armed bandit problems, as in [10], so they are more suitable for the game-theoretic settings and so could potentially be more effective. Then, we use these modified bandits to modify the standard algorithms for both observable and partially observable stochastic games mentioned in (Section 3.3) and (Section 3.4).

5.1 Preparation of multi-armed bandits

This section describes modifications made to the multi-armed bandits to be more fit for usage in the context of stochastic games. By definition, the bandits presented so far (Chapter 4), especially stochastic bandits (Section 4.3), do not consider the actions of the opponent at all. They were created from MDPs and thus are focused on single-agent learning. However, in the case of the fully observable stochastic games, the knowledge about opponent's played action could be leveraged and lead to improved learning.

With this in mind, we propose a new alternative bandit for each standard algorithm and we will call this category *observable stochastic bandit algorithms*.

5.1.1 Observable stochastic bandits

As said before, we design derivations of stochastic bandit algorithms that also observe actions played by the opponent. These derivations differ only in construction of the Q^t function, but everything else is the same as in the standard variants. Thus, we here describe only the general principle behind these variants.

For the next part, suppose that A is a set of bandit's available actions and B is the set of opponent's actions and let $a_N = |A|$ and $b_N = |B|$. An observable bandit does not hold Q^t function directly but rather keeps reward matrix $R^t \in \mathbb{R}^{a_N, b_N}$. Then, an entry $R^t_{a,b}$, where $a \in \{1, \ldots, a_N\}$ and $b \in \{1, \ldots, b_N\}$, corresponds to the average reward received by playing action profile (a, b) up to time step $t \in 1, \ldots$. Similarly, the action trial function is now represented by a matrix $n^t \in \mathbb{N}^{a_N, b_N}$. Moreover, the bandit holds a vector $m^t \in \mathbb{N}^{b_N}$ where entry m^t_b is a number of times an action b was played by the opponent until time t.

Algorithm 5.1 Initialize an *observable* bandit

```
Input: action set $A$, action set $B$
1: $R(a,b) \leftarrow 0 \quad \forall a \in A, \forall b \in B$
2: $n(a,b) \leftarrow 0 \quad \forall a \in A, \forall b \in B$
3: $m(b) \leftarrow 0 \quad \forall b \in B$
```

Then, in the arm selection part, a bandit first computes its Q^t function from the reward matrix based on the opponent's *average play* according to the formula

$$Q^t(a) = R^t_a \cdot \frac{1}{t} m^t \quad \forall a \in A \tag{5.1}$$

where R^t_a is the a-th row of the matrix R^t. This way, the Q^t takes into account how likely is the opponent to play some action. This allows the bandit to avoid selecting promising actions with potentially high average reward, but whose corresponding counterpart is not selected often by the opponent. From this state, the selection of an arm continues as in the standard variant, but with Q^t computed by this method.

The accumulation of gained rewards also differs from the standard variant as well as initialization and is carried out as follows in (Pseudocode 5.2). Later, in experimental

Algorithm 5.2 The **receive** function for *bandit feedback* in observable variant

```
Input: player's action $a^t_i$, opponent's action $a^t_{-i}$, reward $r^t$ for joint actions $(a^t_i, a^t_{-i})$
1: $m(a^t_{-i}) \leftarrow m(a^t_{-i}) + 1$
2: $n(a^t_i, a^t_{-i}) \leftarrow n(a^t_i, a^t_{-i}) + 1$
3: $R(a^t_i, a^t_{-i}) \leftarrow R(a^t_i, a^t_{-i}) + \frac{r^t - R(a^t_i, a^t_{-i})}{n(a^t_i, a^t_{-i})}$
```

evaluation chapter, we compare this extension to the standard bandit algorithms. Note that these bandits can be used only in settings, where the player has access to action history of the adversary, and thus will be useful mainly in fully observable stochastic games.

5.1.2 Stochasticity

Ideally, every multi-armed bandit algorithm should have some stochastic element in it. Either to differentiate among runs of the algorithms or to prevent a situation, where the structure of a particular instance unintentionally benefited the algorithm thus leading to misinterpreted results.

For example, suppose we have a vector of accumulated average rewards for actions over time and we want to select the action with the highest value. Moreover, suppose that there are multiple actions with the same value, which is equal to the maximal value of the vector, present in the reward vector. Due to the way the operator argmax is implemented in programming languages, the action selection always results in an action which is on the first position of those with the same value in the vector. It is desirable to avoid this situation.

Many of the bandits already use such stochasticity somewhere in their design, we, however, add one additional element into each algorithm just so they behave similarly. Namely, before using argmax, argmin or some other position-dependent operation, we change the order of the statistics in the vectors to prevent the example mentioned above. In detail, we generate a random permutation of indices of actions and reorganize the structure of elements according to the created permutation. On this reordered vector we perform the desired operation.

This procedure for the basic case with only Q^t function is documented in (Pseudocode 5.3). If other deciding statistics are used (i.e., UCB(a, α) value), the procedure works similarly.

Algorithm 5.3 Shuffling of Q values before order-dependent selection

```
Input: function $Q^t$ represented as a vector, i.e. $[Q^t(a_1), Q^t(a_2), \dots, Q^t(a_n)]$
1: generate a random permutation $I$ of the set of indices $\{1, \dots, n\}$, where $n$ is the number
   of actions
2: create new action-function $O^t \leftarrow [Q^t(a_{I_1}), Q^t(a_{I_2}), \dots, Q^t(a_{I_n})]$
3: return action $a_{I_i} \in \operatorname{argmax} O^t$
```

Note that the order is not changed in-place of the original values, but rather copied with the new ordering. The inner state of the bandit algorithm stays unaltered, the only effect is on the action selection in the current round.

Moreover, for every instance of the picked bandit algorithm the set of actions A is initially randomly permuted, so the bandits are not exactly the same.

5.1.3 Numerical instability

This section is focused on the Exp3 bandit algorithm as it is the only one gravely affected. Even though its definition serves well for theoretical analysis, if programmed exactly in the way described in (Algorithm 4.8) and in [10], it becomes unusable after few iterations. Due to the formula selected for updating weights w and computing probabilities p for experts, the floating point representation quickly overflows and compromises the selection.

This can be solved by adapting the implementation of Exp3 algorithm from [21]. Here, they provide equivalent formula for computing selecting probabilities but with more numerical stability.

$$p(a) = \frac{\gamma}{|A|} + \frac{1-\gamma}{\sum_{a' \in A} e^{\eta(s(b)-s(a))}} \quad \forall a \in A \tag{5.2}$$

where $\gamma \in (0, 1]$ and $\eta \in \mathbb{R}_+$ are constant parameters. The expression $s(a)$, where $a \in A$, is a sum of rewards for all selections of the action a, each divided by the probability of selecting that action in the specific round.

Both γ and η can be chosen arbitrarily and for different problems may have better results for different pair of these parameters. We choose values according to description in [21]

$$\gamma = \min \left\{ 1, \sqrt{\frac{|A| \cdot \log |A|}{n(e-1)}} \right\}, \tag{5.3}$$

where n is the number of trials and e is the *Euler's number*, and

$$\eta = \frac{\gamma}{|A|}. \tag{5.4}$$

Moreover, for simplicity, we rewrite the Exp3 algorithm (Algorithm 5.4) in a way, that it does not use Hedge as a separate independent component, but all the functionality is implemented inside Exp3. For this, we took inspiration in [22].

Algorithm 5.4 Numerically stable Exp3 with incorporated Hedge

```
Input: set of actions $A$, $\gamma \in (0, 1]$, $\eta \in \mathbb{R}_+$
 1: initialize $s(a) \leftarrow 0 \quad \forall a \in A$
 2: for $t = 1, \dots, n$ do
 3:     $p^t(a) \leftarrow$ compute probabilities according to (Formula 5.2)
 4:     draw $a^t \sim p^t(a)$                ▷ probability of drawing $a_i$ is $p^t(a_i)$
 5:     receive reward $r^t \in [0, 1]$
 6:     for $b \in A$ do
 7:         compute fake cost $\hat{r}^t(b) \leftarrow \begin{cases} \frac{r^t}{p^t(b)} & b = a^t \\ 0 & \text{otherwise} \end{cases}$
 8:         $s(b) \leftarrow s(b) + \hat{r}^t(b)$
 9:     end for
10: end for
```

In this section we made some adjustments to the presented standard versions of multi-armed bandit problems. We provided a way to take into account observations from the opponent, we ensured that the algorithms do not exploit some undesired structure of the problem by increasing their stochasticity. Last, we presented a variant of the Exp3 algorithm, which is numerically stable and can be better used for experiments.

Now, we are ready to use these adjusted bandits with the standard methods described in (Chapter 3) to create new algorithms.

5.2 Bandit iteration

In this section, we outline an alternative to the value iteration algorithm, which does not require solving matrix games and thus linear programming is not necessary. Instead of the linear programs, the previously mentioned multi-armed bandit algorithms are essential to this new algorithm. The general algorithms are described in section (Chapter 4) and their versions adapted for use in games in section (Section 5.1). From the use of bandit algorithms is derived the name of the next algorithm *bandit iteration*.

5.2.1 Algorithm

The main algorithm is the same for all multi-armed bandits as they use the same interface. This interface involves function **select**, which returns the bandit's intended action to be played, and function **receive**, which processes the received reward r^t for the last action a^t. Thus, only a type of the bandit B_{type} and its extra parameters B_{params} must be passed as input to the bandit iteration algorithm.

Other inputs are, of course, the stochastic game to be solved, a number of iterations to be performed $t_{\max}$ and an averaging function step : $T \to \mathbb{R}$. The purpose of said averaging function will be specified later in (Section 5.2.2). The pseudocode for the algorithm is listed in (Algorithm 5.2.1).

Bandit iteration uses the same definition of value function as the value iteration algorithm for stochastic games and aims to gradually improve the estimate of the value function in individual states as well. Also, as in value iteration, we can initialize this value function to an

Algorithm 5.5 *Bandit iteration* for stochastic games

```
Input: $G = (S, A_1, A_2, T, R, \gamma)$, $t_{\max}$, $B_{\text{type}}$, $B_{\text{params}}$, step$(t)$
Output: approximation of $V^*$ after $t_{\max}$ iterations
 1: Initialize $V(s) \leftarrow 0 \quad \forall s \in S$
 2: Initialize $B_1(s) \leftarrow$ new $B_{\text{type}}$ for actions $A_1$ and $B_{\text{params}} \quad \forall s \in S$
 3: Initialize $B_2(s) \leftarrow$ new $B_{\text{type}}$ for actions $A_2$ and $B_{\text{params}} \quad \forall s \in S$
 4: for $t = 1, \dots, t_{\max}$ do
 5:     $V_{\text{prev}} \leftarrow V$
 6:     for $s \in S$ do
 7:         $a_1 \leftarrow$ select action from $B_1(s)$
 8:         $a_2 \leftarrow$ select action from $B_2(s)$

 9:         $r \leftarrow R(s, a_1, a_2) + \gamma \cdot \sum_{s' \in S} T(s' \mid s, a_1, a_2) \cdot V(s')$
10:         $V(s) \leftarrow V_{\text{prev}}(s) + \frac{r - V_{\text{prev}}(s)}{\text{step}(t)}$          ▷ see (Section 5.2.2)

11:         $B_1(s) \leftarrow$ receive $(a_1, r)$          ▷ for observable bandits see (Section 5.1.1)
12:         $B_2(s) \leftarrow$ receive $(a_2, r)$          ▷ for observable bandits see (Section 5.1.1)
13:     end for
14: end for
15: return $V$
```

arbitrary value without breaking the convergence to the real values. The speed of convergence, however, is influenced by choosing the starting values.

In contrast to the value iteration, in *bandit iteration*, a bandit of type B_{type} has to be created for both players in every state $s \in S$ with the set A_i of actions available to the corresponding player i. Also, additional parameters required for each respective bandit algorithm need to be passed to initialized bandits as well (for details see (Chapter 4) and (Section 5.1)).

Note that in the pseudocode (Algorithm 5.2.1), it is assumed that all actions can be played in all states. At the expense of a more complicated notation, the restriction to only available actions could be made, but the overall algorithm would remain the same. The only difference would be that only these playable actions would be given to the bandit in the state s.

In each round of the value iteration, a matrix game u (Definition 3.3) is solved and thus an optimal value of the game is found. This newly retrieved value then replaces the old one in the value function for that solved state and in this way, the representation becomes more accurate.

In this algorithm, on the other hand, no game is solved, but instead, bandits belonging to the current stage's state provide actions a_1 for player 1, a_2 for player 2 respectively, to be played. Then a value representing the expected outcome if the players decided to play the joint action profile (a_1, a_2) is computed and used to construct a new value function V. Also, bandits that provided a_1 and a_2 collect this value by the **receive** method (Pseudocode 4.2), which updates their inner state depending on their type (more in (Chapter 4) and (Section 5.1)). By this update, the bandits learn quality of their selected actions and as the algorithm continues choosing the optimal actions more frequently and on average find the optimal strategy. In this way, the algorithm gets closer to the optimal value. To get the complete value function, we perform the recently mentioned procedure for every state $s \in S$.

After the algorithm performs all $t_{\max}$ rounds, it returns the final approximation V of the real value function V^*. Value in $V(s)$, $s \in S$ approximates the value of the game $V^*(s)$, i.e. total expected reward of the maximizing player 1, if the game started in the state s. The equilibrial strategies can be then extracted from the value function by using the two linear programs described in one of previous sections about the value iteration algorithm (LP 3.1), resp. the ommited LPSGMIN. As an alternative, they could be retrieved from the individual bandits as the normalized vector constructed from values in the Q^t function.

5.2.2 Averaging of value functions

In a single-agent environment, this newly constructed value function can be taken as an updated more accurate one, due to the fact that there always exists a single action, which is better than some randomized decision, and thus every optimal strategy can be regarded as a pure strategy. This is not possible in stochastic games, because, as mentioned earlier in this thesis, the equilibrial strategies are possibly mixed. Therefore, the value functions from previous rounds must be taken into account together with the new one as the bandits explore new possibilities and better actions. Considering only the new value function would mean taking the last pure actions recommended by the bandits.

To tackle this problem, we need to aggregate all past value functions in some way. The most natural and easiest way of achieving is a simple arithmetic averaging of values $V(s) \forall s \in S$. However, as is later showcased in the experimental chapter (Chapter 6), the convergence of this aggregate can be really slow, although functional.

This can be caused by the fact, that the arithmetic average takes all such collected values equally and each of these values have the same impact on the final average. Nevertheless, it would be practical to consider the later values with higher weights than the early ones, as they are presumably closer to the real values. At the beginning, the bandits have not yet accumulated enough trials to strongly prefer better actions and are likely to explore more aggressively and thus the values tend to fluctuate more.

For better scaling of the algorithm, we once again use the incremental average as devised in (Derivation 4.5). This formula can be viewed as the old value added to the difference between the reward and the old value multiplied by a decreasing function. In the case of the arithmetic average, this decreasing function is the *reciprocal function* $\frac{1}{t}$, where t is the current time step. Henceforth, we will identify this decreasing function with $\text{step}(t)$. Also, we will name the reciprocal function $\text{LIN}(t)$.

To increase weights to more recent values, we need to select such a decreasing function, which decreases slower than the reciprocal function. We choose one more and that is a function $\text{SQRT}(t) = \frac{1}{\sqrt{t}}$.

As can be seen from the figure (Figure 5.1), the $\text{SQRT}(t)$ step function decreases slower than $\text{LIN}(t)$ and thus can potentially lead to a faster convergence, because it will consider more recent values with higher weights than the older ones. Please note, that the x axis of the plot is displayed in the logarithmic scale. However, we can no longer speak of the aggregation as of an arithmetic average. These two step functions will be later compared on the bandit algorithms.

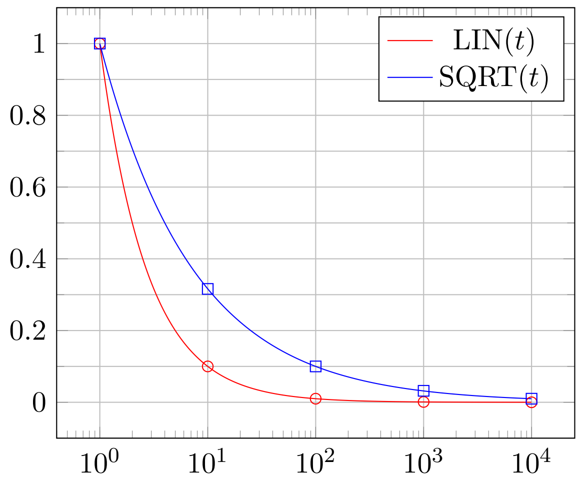


Figure 5.1: Comparison of decrease in value depending on time steps of two used step functions. These functions are used to compute the incremental average as described in (Section 5.2.2)

5.2.3 Observable bandits

When an observable variant of any stochastic multi-armed bandit algorithm is used in the *bandit iteration* algorithm (Algorithm 5.2.1), the **receive** function looks a bit different. These bandits need to receive both action a_i selected by the owner and action a_{-i} selected by the opponent together with the gained reward. A more detailed description of observable bandit alternatives is located in section (Section 5.1.1).

5.3 B-HSVI

In (Section 3.4.3), a POMDP heuristic search value iteration modified for OS-POSGs (Section 2.6) was presented. Even though this algorithm scales better in comparison to the exact method, it still uses linear programs as a method for solving stage games, which can lead to poor scalability. With the use of multi-armed bandits presented in this and previous chapters (Chapter 4), we introduce an alternative method to solve the stages and find the equilibrial strategies.

5.3.1 The algorithm

From the original algorithm in (Section 3.4.3) the modified B-HSVI algorithm differs only in finding the equilibrial strategies of $[HV_{LB}^{\Gamma}](b_t)$, resp. $[HV_{UB}^{\Upsilon}](b_t)$. Thus, the only change in (Algorithm 3.3) is on the lines 7, resp. 8.

In (Algorithm 5.6) is shown the extraction of a stage strategy profile from the lower bound $(\pi_1^{\text{LB}}, \pi_2^{\text{LB}})$. The algorithm for the upper bound is analogous except for a few minor differences. Instead of an α-vector, a point (b, y) is selected from Υ and then new point (b', y') is created and inserted to Υ.

In each call of solving the stage game, i.e. lines 7, 8 in the original algorithm, the procedure (Algorithm 5.6) is performed. First, an α-vector, or a point (b, y) is selected from its respective set. In the case of the set Γ, the selection is partly randomized. The elements of Γ are sorted in decreasing order by their value in the current belief point b_t. With probability $\epsilon = 0.1$

Algorithm 5.6 Extract equilibrial strategies from $[HV_{LB}^{\Gamma}](b_t)$

```
Input: OS-POSG $G$, $V_{LB}^{\Gamma}$, $\Gamma$, $b_t$
Output: stage strategies $\pi_1^{\mathrm{LB}}$, $\pi_2^{\mathrm{LB}}$
1: $\alpha \leftarrow$ select from $\Gamma$
2: $B_1, B_2 \leftarrow$ bandit algorithms belonging to $\alpha$
3: $a_1, a_2, \pi_1, \pi_2 \leftarrow$ retrieve current actions and strategies from $B_1$, $B_2$
4: $\mathrm{val}_{a_1}, \mathrm{val}_{a_2}(s) \leftarrow$ compute bandit feedback $\forall s \in S$
5: $B_1, B_2 \leftarrow a_1, \mathrm{val}_{a_2}$ update state of the bandits
6: $\Gamma \leftarrow \Gamma \cup \{$construct new $\alpha'$ from $(\pi_1, \pi_2)$ in $b_t\}$
7: $B'_1, B'_2 \leftarrow$ copy bandits $B_1, B_2$ belonging to the new $\alpha'$
8: return $\pi_1, \pi_2$
```

a random α-vector from the first 10 elements of the ordered Γ is selected, with probability $(1-\epsilon)$ the single best α-vector. This randomization prevents the algorithm from choosing a single α-vector every time and creating new compositions which provide no improvement in value. The point (b, y) is selected by the belief point b.

Then we retrieve the bandit algorithms which belong to this selected α-vector, whose strategy is meant to be refined. Player 1 owns only a single bandit, while the player 2 has one bandit algorithm for each state $s \in S$. From the bandits, the current actions are selected as well as the strategies played so far by these bandits. Note, that while the bandit for the player 1 returns a single action a_1 and a single stage strategy π_1, the bandits of the player 2 are conditioned by the state s and thus they provide mappings $a_1 : S \to A_1$ and $a_2 : S \to \Delta(A_2)$, which can conveniently be represented by vectors of length $|S|$. The returned strategies correspond to relative frequencies of selections made by the bandit so far including this round, i.e. average play.

With these things prepared, the bandit feedback for the lower bound update is computed according to the following formula

$$\mathrm{val}_{a_2}(s) = R(s, a_1, a_2(s)) + \gamma \sum_{(o,s')\in O\times S} T(o, s' \mid s, a_1, a_2(s)) \cdot \alpha^{a_1,o}(s') \quad \forall s \in S, \tag{5.5}$$

where a_1, a_2 are fixed by the selection from the bandits and $\alpha^{a_1,o}(s')$ is a convex combination of the α-vectors for the current pair (a_1, o) evaluated in the simplex vertex $s' \in S$. The upper bound formula is again analogous except that instead the $\alpha^{a_1,o}(s')$, the lower convex hull of the points is computed and then evaluated in s'. The value val is then computed as

$$\mathrm{val}_{a_1} = \sum_{s\in S} b_t(s) \cdot \mathrm{val}_{a_2}(s). \tag{5.6}$$

This feedback is the passed to the bandits, however, the val_{a_2} is multiplied by -1 to follow the maximization nature of the bandit algorithms.

Last, new α-vector α' or (b', y') is created based on the strategies returned by the bandits in this round. These two operations are still conducted by a linear program. The bandits B_1 and B_2 are then copied and assigned to this new object to allow independent continuation in case it is better than the former.

Presolve

For this procedure to work as described before, we need to alter the *presolve* process in a similar fashion. New bandits need to be initialized and assigned for every α-vector and (b, y) point. These empty bandits then correspond to uniform strategies over A_1, resp. A_2.

5.3.2 Performance

This modification of the HSVI algorithm removes the need for solving a linear program to obtain the stage strategies in each step. Instead, it uses strategies learned by the bandit algorithms corresponding to individual lower/upper bound elements. However, the update of these strategies is changed only by a single action selection each round and thus can also change slowly in later phases of the learning. In the next section, we compare different bandit algorithms and how they handle this slow changes.

The bandits are being copied because if their single selection was not in a good direction, in terms of creating α-vector with lower value than the original, it would not compromise the original bandits and could be tried again. This is also improved by the randomized element selection from Γ.

5.4 Conclusion

In this chapter, we summarized modifications of the bandit algorithms presented in chapter (Chapter 4) to fit them more into game theory. Next, we used these bandit algorithms to define an algorithm similar to value iteration called *bandit iteration* to find solution of stochastic games without the need of computing linear programs. Last, we incorporated the multi-armed bandits to the *heuristic search value iteration* algorithm to create algorithm *B-HSVI* solving OS-POSGs again to omit solving too many linear programs.

In the following chapter, we compare the specified bandit algorithms on SGs and OS-POSGs and how they influence convergence to the true values.

Chapter 6
Experimental evaluation

In the preceding chapters, we have built up the theory around games and the methods to discover their solution. Then, modifications of these algorithms, which do not necessarily require linear programming formalism to determine the approximate results, were proposed. This chapter consists of comparing different multi-armed bandit algorithms within these modified methods on two distinct models.

Firstly, we study the performance and the convergence of distinct bandits, used in the bandit iteration algorithm proposed in (Section 5.2), to the optimal value found by the value iteration method on the model of fully observable stochastic games. We will mostly focus on comparing the observable and standard variants as well as the two different accumulation steps (Section 5.1), LIN(t) and SQRT(t).

Secondly, we compare convergence of the B-HSVI (Section 5.3) algorithm also using multi-armed bandits with the standard HSVI (Section 3.4) which uses mathematical programming. In this case, we focus on partially observable stochastic games and specifically on the OS-POSGs subset, even though this algorithm can solve perfect information stochastic games, too.

6.1 Technical details

The experiments were carried out through the *Metacentrum* platform acknowledged at the preface of the thesis, specifically on the *halmir* cluster [23]. For a single run of an algorithm was used a single AMD EPYC 7543 core with 2.8GHz base frequency and 16GB of RAM. The algorithms were implemented in a single-threaded program in the programming language Julia, version (1.7.0) [24]. For the linear programs, the *Coin-or linear programming* [25] and *IBM CPLEX v. 20.1* optimization frameworks were used. The implementation details for solution methods of both evaluated algorithms are listed in (Appendix C).

6.2 SGs and bandit iteration

In (Section 2.4) a formal model of a stochastic game was introduced and an algorithm provided to solve any instance declared in this formalism in (Section 3.3). It was modified to create *bandit iteration* algorithm in (Section 5.2). For comparison purposes of this thesis, we used two specific game types, each consisting of multiple different instances. Before proceeding with the experimental testing of the algorithm, we define the two types of games. The games are quite simple and the particular used instances are small so many results of the experiments can be at least partly verified by intuition.

6.2.1 Game types

In both games, there are two players placed in a maze-like environment. The players simultaneously choose actions and move through the maze or carry out extra manoeuvres and while player 1 tries to defeat player 2 as quickly as possible, player 2 tries to stay alive as long as he can.

Both players can observe positions and formerly chosen actions of the adversary, the only thing unknown to them is the action selected in the current round. Because these are types of *stochastic* games, every selected action can have multiple outcomes with different probabilities. For example, if a player decides to go forward, he can end up somewhere else than before him with some non-zero probability. However, these outcomes and probabilities depend on the particular instance of a game type and are not essential for the understanding of the game. Henceforth, for the sake of a simpler description, we will assume that each action has only a single outcome, i.e. the player always does what was intended.

For both types, it is possible, that the game will take infinitely long. As any player can perfectly observe its adversary, it just depends on the specific instance. We try to avoid this situation by posing a restriction that there always exists some path between both players.

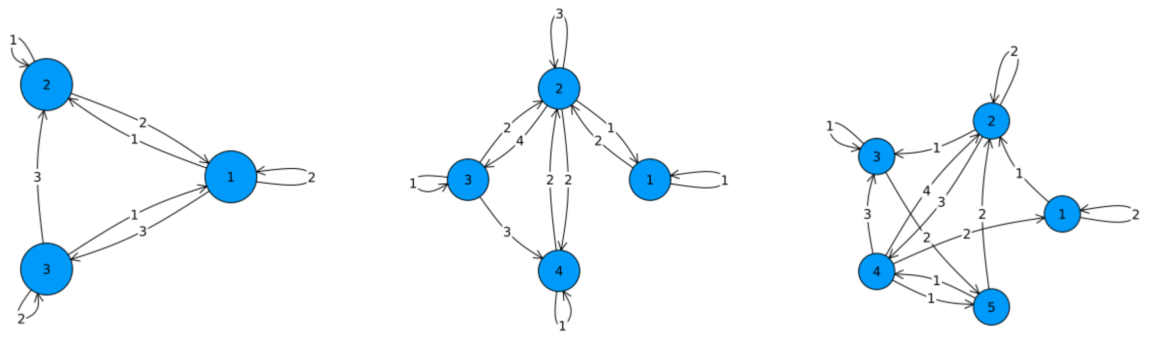
Chase

The game we call *Chase* is the simpler of the two types. It takes place in a randomly generated directed graph $g = (V, E)$ with loops allowed, where V is a set of vertices of size n and E is a set of directed edges. The edges connect the n vertices in such a way, that there always exists a path between any two different nodes in the graph.

Player 1, i.e. *chaser*, and player 2, i.e. *runner*, are then located in random vertices v_1, v_2 in g. Actions available to both players in a vertex v correspond exactly to outgoing edges from the vertex v. An action identified with edge (v, v') means that a player can move from vertex v to the adjacent vertex v'.

Players use these actions to move through the graph g and the goal for player 1 is to *catch* player 2, which happens when they end up in the same vertex. When this phenomenon occurs, they are both moved to a special absorbing state $n + 1$, which is not shown on the visualizations (Figure 6.1), and the game ends. The only exception to this rule, when both can be present in the same vertex and the game does not end, is the initial round 0, because both can be placed in the same vertex $v_1 = v_2$. In this case, the game continues and ends only if they immediately go to a same vertex again.

To motivate the chaser to catch the runner the rewards are defined as follows. Every chaser's action, which does not lead to victory, is penalized with -1. When the runner is caught, the chaser receives a reward $+10$. The chaser is trying to maximize the total received reward over all rounds and thus trying to win soon. On the contrary, from the definition of a two-player zero-sum game, the runner is trying to minimize this quantity and avoid being caught for as long as possible.



(a) : Chase3 with $n = 3$ **(b) : Chase4** with $n = 4$ **(c) : Chase5** with $n = 5$

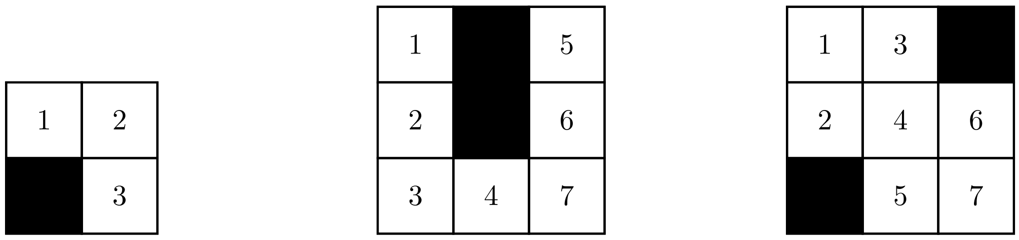
Figure 6.1: Illustration of graphs representing the environment of the **Chase** stochastic game for different number of vertices. The detailed description of **Chase** is in (Section 6.2.1). Numbers in vertices of the graphs represent identifiers of the states of the game and the integers near the edges are labels for corresponding actions.

Tag

As opposed to *Chase*, *Tag* is closer to more conventional games. Player 1 is called tagger, while player 2 is named evader. Players are randomly placed into a square $n \times n$ board with some obstacles, which do not permit the player to pass through them. The only restriction to the placement of obstacles is, as mentioned before, that there must exist a path between the two players so that solving the game even makes sense.

Both players have actions to move to adjacent squares if there is no obstacle present on the target square. In addition, the tagger can shoot a laser beam either vertically or horizontally in every state. In this case, player 2 stays on the square he was before shooting and the beam hits all squares in the chosen direction until an obstacle or end of the map is hit.

The goal of the tagger is not to get on the same square as the evader, as it was in *Chase*, but to hit the evader with the laser beam. Again, he tries to succeed as fast as possible and player 2 tries to avoid getting hit for the longest. Rewards for moving and completing the objective remain the same as in *Chase*. However, to discourage the tagger from shooting unnecessarily, blasting a laser beam without hitting the target results in a bigger penalization by reward -10. As well as in *Chase*, player 1 tries to maximize the overall accumulated reward and player 2 tries to minimize it.



(a) : *Tag* 2×2 **(b) :** `tag_3_01` 3×3 **(c) :** `tag_3_02` 3×3

Figure 6.2: Illustration of the maze, where the tagger tries to shoot the evader in the **Tag** game described fully in (Section 6.2.1). Both players cannot step on the black fields. The states available for them are labelled with unique integer identifiers used in later comparisons.

When the evader is hit, the game ends and both players are again moved into an absorbing terminal state, which is again not shown in the pictures (Figure 6.2).

6.2.2 Environment and parameters

The performance of an algorithm depends not only on the algorithm itself but also on its parameters and the settings of the environment where the algorithm is put. Here, the most important parameters are the bandit ones as they drive the search. These details are briefly described in this section.

Reference

To argument about convergence of a given bandit algorithm used inside the bandit iteration framework (Algorithm 5.2.1) some referential value is needed. The values returned by the algorithm are then supposed to get as close as possible to this reference value. Ideally, they should be equal.

The best possible reference value is the optimal value, but this value is unknown. Thus, the value returned by the value iteration algorithm, after it stopped with some precision parameter Δ. From the definition of value iteration (Algorithm 3.2), the returned value function $V_{\text{iter}}(s)$ is a good approximation of the optimal $V^*(s)$ in every state $s \in S$. For the purpose of the following comparisons, we pose $V^* = V_{\text{iter}}$ and the values returned from the bandit iteration algorithm are compared to this value. The precision parameter was set to $\Delta = 1 \times 10^{-6}$.

For comparing the results of the bandit algorithm with different bandits, we use quantities $V(s)$ based on this formula

$$V(s) = V_{\text{bandit}}(s) - V^* \qquad \forall s \in S \tag{6.1}$$

When $V(s) = 0 \quad \forall s \in S$, we say that the bandit iteration has converged.

Means of evaluation

In stochastic games, there is defined a starting state, where both players begin their plays. Solving the game then means finding value of the game in this particular starting state. However, in our experiments, we always perform the learning backup for every state $s \in S$ and thus we end with value of the game for every state. To acquire the value of the game for the starting point, it only needs to be picked from the value function V_{bandit}. This way is convenient for comparison of the bandit algorithms and how they behave in different types of states.

Setting

Experiments were run on all three *Chase* instances (Figure 6.1), for one *Tag* instance with dimensions 2×2 (Figure 6.2a) and for 7 instance of *Tag* with dimensions 3×3. Two of those latter instances are displayed in (Figure 6.2b), respectively (Figure 6.2c).

The versions of multi-armed bandits from (Section 5.1) were used in the bandit iteration algorithm for evaluation, which are those with shuffled greedy action selection, the observable variants and the numerically stable Exp3. Every bandit was tried inside the bandit iteration

bandit	Best of N	ϵ-greedy	Successive elimination	UCB	Exp3
parameter	$N = 100$	$\epsilon = 0.1$	$\alpha = 20$		$n = 10^5$

Table 6.1: The best found parameter values for each individual bandit algorithm tested on stochastic games in the bandit iteration framework.

for 10^5 iterations 10 times for each instance for both steps, LIN(t) and SQRT(t) (Definition 5.1), with different seeds for each single run.

The runtime of these bandit iteration runs for different instances of *Tag* and *Chase* is analysed in (Appendix A).

Parameters

The parameters were fixed to those used standardly with each respective bandit. Both the observable and non-observable variants were parametrized with the same value, so the comparison is more accurate. The individual chosen parameters for all non-observable bandits are listed in table (Table 6.1).

For the Best of N bandit, the N should be much smaller than the intended number of iterations, otherwise it would degrade into simply random selection as the first $N * |A|$ actions are selected uniformly at random. The chosen number is shown in (Table 6.1)

The value $\epsilon = 0.1$ is used usually for the ϵ-greedy as it provides sufficient exploration while exploiting very frequently.

The exploration parameter α has the same purpose for both Successive elimination and UCB bandits as it represents how fast do the confidence bounds tighten around the average received reward. A good rule of thumb for selecting this value is the size of an interval, from which the rewards are sampled. In the case of *Catch*, the interval is $[-1, 10]$, so the parameter should be $\alpha_{\text{catch}} = 11$, for *Tag* it is $[-10, 10]$, and thus $\alpha_{\text{tag}} = 20$. Naturally, the larger one was selected as α.

The Exp3 is given the number of iterations n as a single parameter and then the appropriate γ and η are computed as declared in (Formula 5.3).

The experiments were conducted for different parameter settings with the above in mind. The values listed in (Table 6.1) showed the best results.

Graphs

Graphs are the most important part of the following examination of the algorithm's behaviour. To prevent confusion, we briefly describe the properties of the used graphs.

Each graph shows how selected bandits behave in a single state of the examined stochastic game. Values on the x axis correspond to rounds $t = 1, \dots, T$ of the bandit iteration and due to the large $T = 10^5$, this axis is scaled by the decimal logarithm $\log_{10}$. The values on the y axis then displays the deviation of the result computed by the bandit iteration and the optimal value found by value iteration for each round. From these differences $V(t) = V_{\text{bandit}}(t) - V^*$ is computed *mean* $\mu(t)$ and the *standard deviation* $\sigma(t)$. The graph is thus shown as a line $\mu(t)$ surrounded by an interval $[\mu(t) - \sigma(t), \mu(t) + \sigma(t)]$ $\forall t = 1, \dots, T$.

The deviation of the optimal value from itself is always 0 and thus is displayed as a constant function $y = 0$ in black.

Now we have everything ready for the actual comparison of the bandit algorithms and their convergence.

6.2.3 Individual bandits

In this short section, we summarize the analysis of the individual bandit algorithms. We focus on their performance, comparison of the standard and observable variants and investigation of the effects of the two step functions on the course of convergence. This summary is based on a detailed analysis located in (Appendix B).

From the comparisons in (Appendix B), it clearly follows that the observable variants converge faster and closer to the optimal value than the standard bandits. We demonstrate this on the example of UCB. This algorithm does not converge to the optimal value in states where mixed strategies are optimal, because it was designed to find the best single action. However, the observable UCB is able to leverage the average play of the opponent and correctly find the mixed strategy and converge to the optimal value.

Similarly, the SQRT(t) step function is superior to the LIN(t) function, but not as definitely as the observable variants are superior to the standard ones. The SQRT(t) causes much faster convergence to the value, but when the value is close to the optimal value it produces high fluctuations in value. On the other hand, LIN(t) has a smooth course but at the expense of slow convergence.

From the experiments, the best performing algorithms are UCB, ϵ-greedy with their respective observable variants and the Exp3 adversarial bandit. These algorithms often find the optimal value with either of the step functions. In contrast, the Best of N bandit and the Successive elimination do not perform well due to fixating a single best action at some point, which happens with these bandits with no means of recovery.

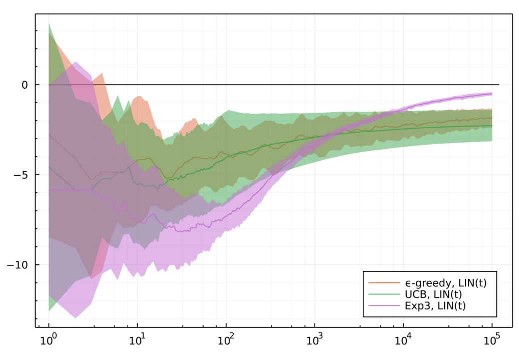
We proceed with comparison of these selected well-performing multi-armed bandits among themselves to analyse their speed and resistance to changes in value.

6.2.4 Bulk comparison

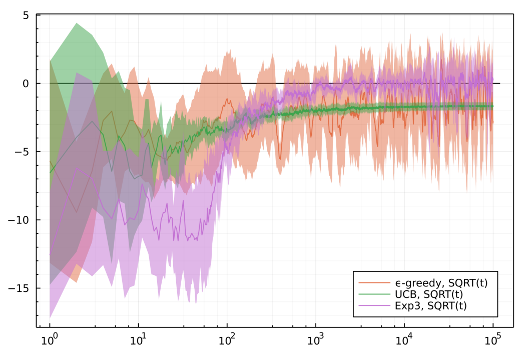
Previously, we showcased how the proposed algorithms behave inside the bandit iteration framework and how they converge to the value found by the value iteration. Also, we compared the standard single-agent versions with the observable versions meant for games. Now, we focus on the three most promising multi-armed bandits and compare them together on selected states. Generally, the bandits which stop even considering some action should be avoided, as this action can later be important in the mixed strategy and just more exploration is necessary.

The comparison is conducted on 4 states, where are many possible actions and where the optimal strategies are mixed which is confirmed by the value iteration. These selected bandit algorithms usually have no problem with convergence to an optimum in states with pure strategies, so these are more appropriate to showcase the advantages and disadvantages of each individual bandit. We separate the set-ups by used step function and the version of the algorithm to prevent confusion and increase readability of the charts.

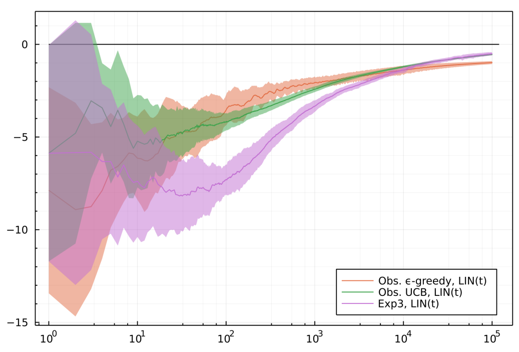
The first state is $s = (3, 3)$ of the instance `tag_3_01` (Figure 6.2b). In this state (Figure 6.3), the Exp3 algorithm is superior to the standard versions without adversary's average play. Even though it falls behind at the beginning and despite the large fluctuations with the SQRT(t) step, it eventually overcomes the other two in convergence. Standard UCB and



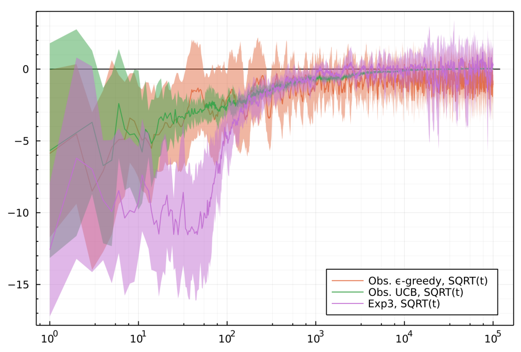
(a) : Standard bandits with LIN(t) step



(b) : Standard bandits with SQRT(t) step



(c) : Observable bandits with LIN(t) step



(d) : Observable bandits with SQRT(t) step

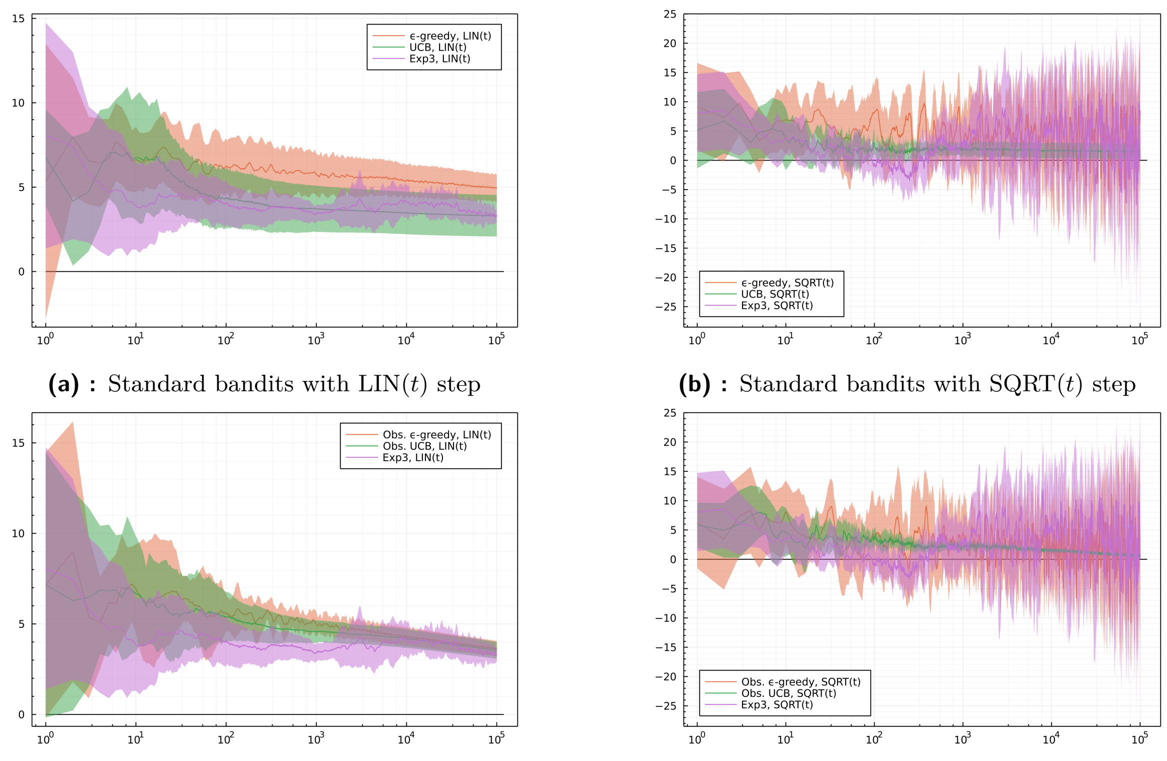
Figure 6.3: This quartet of figures studies development of deviation of learned values from the value iteration result over time. The comparison is made on a **Tag** instance `tag_3_01` in a mixed-strategy state $s = (3, 3)$ for the three best bandits from previous comparisons, ϵ-greedy, UCB and Exp3, in combination with the two step functions, LIN(t) and SQRT(t). The top half compares the standard bandits together, the bottom half their observable complements.

ϵ-greedy perform approximately the same, but UCB is more resistant to the fluctuations caused by the weighted step. The observable UCB, however, catches up to the Exp3 algorithm and does not oscillate afterwards, observable ϵ-greedy follow close behind.

The second state is $s = (4, 4)$ of the instance `tag_3_02` (Figure 6.2c). Here (Figure 6.4), all the bandits converge very slowly with the LIN(t) step. Moreover, the non-observable variants seem to converge to some suboptimal values, even though, after many more iterations, the Exp3 algorithm appears that it would approach the optimum closer. The observable bandits converge as slowly as the standard ones, but the trend suggests further decrease of the deviation from the value of the game for more iterations. With this step function, these three algorithms are comparable.

When it comes to SQRT(t), the situation dramatically improves. Especially the average play UCB performs very well as it converges close to the real value and does not fluctuate afterwards. The other standard bandits converge to another value and oscillate around it, while the observable ones converge to the optimum. The fluctuations are enormous, when intersected by a line.

The situation (Figure 6.5) for the third discussed state $s = (1, 4)$ of the instance `tag_3_02` (Figure 6.2c) is very similar to the second example with few exceptions which will be now discussed. The LIN(t) step performs a bit worse than in the previous case, because it



(a) : Standard bandits with LIN(t) step

(b) : Standard bandits with SQRT(t) step

(c) : Observable bandits with LIN(t) step

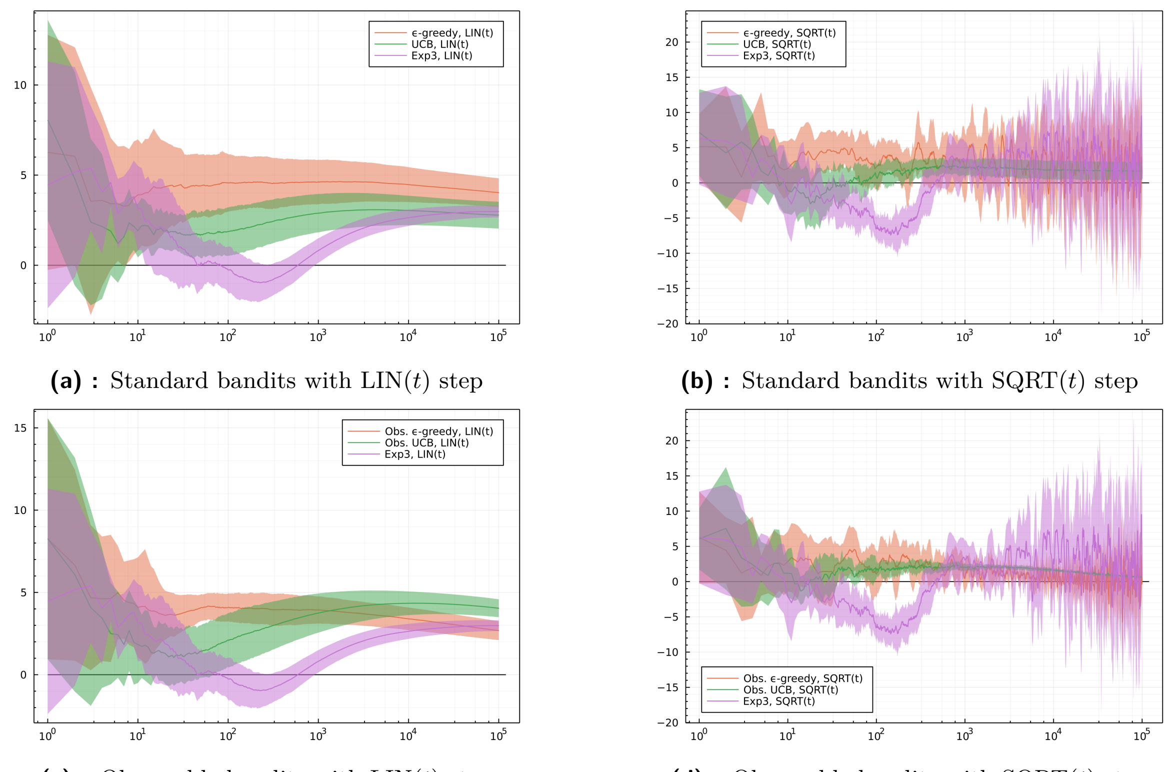
(d) : Observable bandits with SQRT(t) step

Figure 6.4: This quartet of figures studies development of deviation of learned values from the value iteration result over time. The comparison is made on a **Tag** instance `tag_3_02` in a mixed-strategy state $s = (4, 4)$ for the three best bandits from previous comparisons, ϵ-greedy, UCB and Exp3, in combination with the two step functions, LIN(t) and SQRT(t). The top half compares the standard bandits together, the bottom half their observable complements.

approaches towards the optimal value for a while and then it diverges to another value before turning to improve again. From the graphs it seems, that more iterations would bring the deviations even closer to 0. But again, the observable UCB with SQRT(t) step dominates the others especially in close to no fluctuations and tight standard deviation intervals.

Interestingly, the Exp3 algorithm always goes below the 0 deviation line and then again above rather than going just in one direction. This could be tied to the two most probable joint actions which can occur in this state. The evader can go left or down, while the tagger can hit the evader with vertical or horizontal beam. The Exp3 thus can first explore these two pure strategies before settling to the randomized solution.

The last state is $s = (3, 3)$ from the chase instance (Figure 6.1c). Here (Figure 6.6), the situation can be divided into two categories. First, the observable variants converge for both step function, only the SQRT(t) causing more oscillation at the end. Second, the non-observable ϵ-greedy approaches the optimal value but not as quickly and precisely as Exp3. The non-observable UCB converges to another value, because it found a pure strategy in a state when mixed one is required.



(a) : Standard bandits with LIN(t) step

(b) : Standard bandits with SQRT(t) step

(c) : Observable bandits with LIN(t) step

(d) : Observable bandits with SQRT(t) step

Figure 6.5: This quartet of figures studies development of deviation of learned values from the value iteration result over time. The comparison is made on a **Tag** instance `tag_3_02` in a mixed-strategy state $s = (1, 4)$ for the three best bandits from previous comparisons, ϵ-greedy, UCB and Exp3, in combination with the two step functions, LIN(t) and SQRT(t). The top half compares the standard bandits together, the bottom half their observable complements.

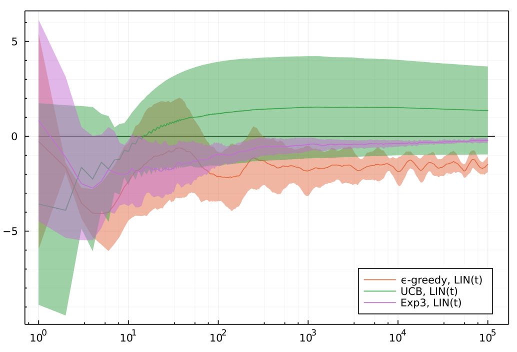
6.2.5 Strategies

Here we shortly list and compare strategies acquired by the bandit algorithms in an example state. The strategy is computed as the average use of that said arm. In terminology from (Section 4.2) the strategy in state s, which is being learned by the bandit, is computed as

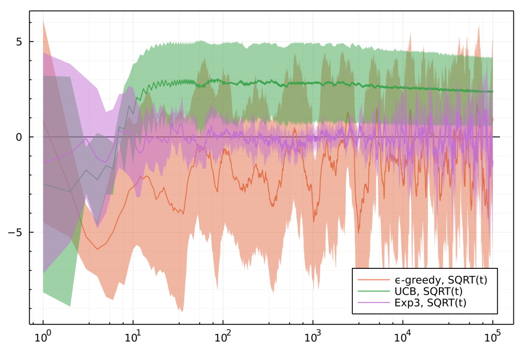
$$\pi(a) = \frac{n^t(a)}{\sum_{a \in A} n^t(a)} \qquad \forall a \in A. \tag{6.2}$$

In the table (Table 6.2), the average policies over the 10 runs are listed. For simplicity, we include only the runs with SQRT(t) step and only the strategies of the Tagger. The data in the table confirm what was already clear in the graph comparison before. The best of N simply selects the best action after the first N trials, while successive elimination removes the not so good arms. But both of them do not reach near the optimal strategies and thus could not converge to the optimal 0 deviation in the experiments before.

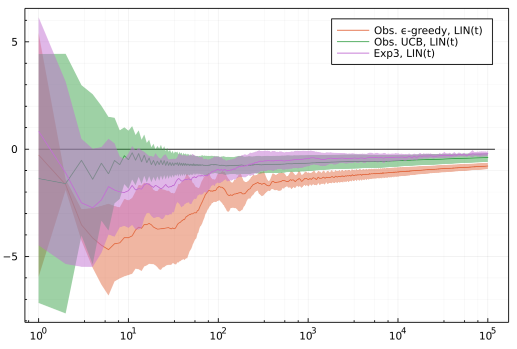
The rows for ϵ-greedy show that the bandit truly explores as it has a bit bigger usage of actions $\Updownarrow$ (shoot laser beam vertically) and $\Leftrightarrow$ (shoot laser beam horizontally) which are not used in the optimal strategy at all. Moreover, it can be said that the other bandits explore a lot less.



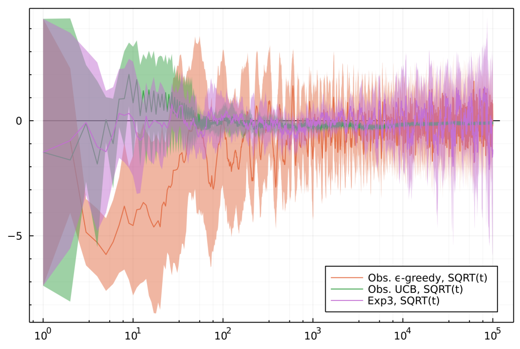
(a) : Standard bandits with LIN(t) step



(b) : Standard bandits with SQRT(t) step



(c) : Observable bandits with LIN(t) step



(d) : Observable bandits with SQRT(t) step

Figure 6.6: This quartet of figures studies development of deviation of learned values from the value iteration result over time. The comparison is made on a **Chase** instance `Chase5` in a mixed-strategy state $s = (3, 3)$ for the three best bandits from previous comparisons, ϵ-greedy, UCB and Exp3, in combination with the two step functions, LIN(t) and SQRT(t). The top half compares the standard bandits together, the bottom half their observable complements.

The UCB together with the observable variant gets quite close to the values found by the value iteration. It should be noted, that most of these strategies could turn out better if the beginning was not included. If the probabilities were computed from let's say second half, it could filter the wrong choices in the beginning and thus get closer to the optimal strategies.

The Exp3 bandit results in very similar average strategy as the UCB and observable UCB algorithms and thus is again close to the optimum.

6.3 OS-POSGs and B-HSVI algorithm

In (Section 2.6) we presented the One sided partially observable stochastic game formalism used to model scenarios, where one of the adversarial agents does not have full information about the environment state. Then, in (Section 3.4) we showed an algorithm invented by [9], which is proven to approximately solve problems modelled as OS-POSGs with the use of linear programming. To avoid the poor scalability of such linear program, which can grow very large for partially observable settings, we altered the HSVI algorithm to form a new method called B-HSVI in (Section 5.3).

Here, we experimentally evaluate the B-HSVI algorithm and compare the different multi-armed bandit type selections on a selected instance of an OS-POSG game. In contrast to the

	$\Updownarrow$	$\Leftrightarrow$	$\uparrow$	$\rightarrow$	$\downarrow$	$\leftarrow$
value iteration	0	0	0.4	0.6	0	0
Best of N	0.997	0.001	0.001	0.001	0	0
Obs. Best of N	0.997	0.001	0.001	0.001	0	0
ϵ-greedy	0.07	0.21	0.18	0.54	0	0
Obs. ϵ-greedy	0.03	0.27	0.26	0.69	0	0
Successive elim.	0.02	0.02	0.02	0.94	0	0
Obs. Successive elim.	0.02	0.03	0.02	0.93	0	0
UCB	0.01	0.16	0.21	0.62	0	0
Obs. UCB	0.0	0.10	0.23	0.66	0	0
Exp3	0.01	0.08	0.27	0.64	0	0

Table 6.2: This table compares strategies learned by individual bandits used in the bandit iteration framework to the optimal mixed strategy found by the value iteration algorithm. These depicted Tagger's strategies correspond to distributions over actions: $\Updownarrow$ = "shoot beam vertically"; $\Leftrightarrow$ = "shoot beam horizontally"; $\uparrow$, $\rightarrow$, $\downarrow$, $\leftarrow$ = "move to the adjacent square determined by the direction of the arrow". This strategy was learned on **Tag** instance `tag_3_01` in a state $s = (3, 3)$.

SGs, the observable variants cannot be applied to this setting. Therefore, we consider only the standard bandit algorithms receiving feedback only for actions selected by them.

Note, that we will focus mainly on the results which are different from the previous comparisons The similarities will be mentioned but not thoroughly analysed.

6.3.1 Pursuit-Evasion

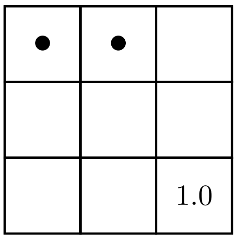
For comparison purposes we selected a game type similar to the *Chase* and *Tag* game types called *Pursuit-evasion*, or shortly **PEG**. This game type is taken from [17] and was inspired by [26].

The game is played on a grid with a fixed number of rows and instance-dependent number of columns. The two players are then placed in the environment and their actions correspond to moving to an adjacent square. The player 1, called the *Pursuer* and represented by n independent units, aims to get one of these units on the same grid cell as the *Evader*'s single unit and this way terminate the game. The Evader, on the other hand tries to escape for as long as possible. For catching the Evader, the Pursuer receives reward 1.0, otherwise 0.0.

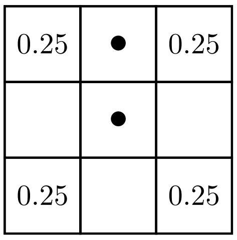
As player 1, the Pursuer does not have full information, thus the position of the Evader is unknown to him. To simplify the problem, we employ the information sets (3.4.1), hence allowing the Pursuer to know perfectly his own positions on the grid, effectively reducing the dimension of the belief search space. The Evader has access to full information and knows his and the adversary's states and the actions played by both of them.

For the comparison purposes, we set the parameters of the game to a grid 3×3 and fix the number of Pursuer unit to $n = 2$. However, for the sake of clarity, the transition probabilities are either 1 or 0, thus prohibiting failed effects of actions.

For the comparisons, we consider two initial settings as shown on (Figure 6.7). The setup on the right should be a bit easier for the algorithm to solve as the Evader is with some probability very close to getting caught.



(a) : **PEG 1** with initial belief b_1^{init}



(b) : **PEG 2** with initial belief b_2^{init}

Figure 6.7: Illustration of the environment, where the game of **PEG** (Section 6.3.1) is played. In the figures, the • symbol represents the units of the Pursuer and the numbers, represent the initial belief b^{init} about the position of the Evader.

6.3.2 Environment and parameters

As in the previous comparison, the convergence of the individual bandit algorithms inside the B-HSVI algorithms depends on the parameters and the means of evaluation. These details are briefly described in this section.

Reference

As well as in the previous experiments, we need some referential solution to which all the bandit algorithms will be compared. Ideally, we would take the exact optimal solution, but for this class of problems, this value is not available. Thus, we use the result of the original HSVI algorithm, which is proven to converge to an ϵ-neighbourhood of the true solution.

Specifically, we solved the two possible initial settings of the **PEG** game instance (Figure 6.7) to a precision $\epsilon = 10^{-4}$. The retrieved solution is a tuple (l, u) where $l \in \mathbb{R}$ is the lower bound and $u \in \mathbb{R}$ is the upper bound and where $l \leq u$. The value used as a reference then corresponds to the number exactly between these two bounds, i.e. the arithmetic average of l and u.

In the experiments we observe how the bandits and the B-HSVI approach this referential value.

Means of evaluation

As opposed to the value iteration and bandit iteration for stochastic games, the HSVI-based algorithms only focus on finding the close approximation of the initial state and the value of the others is not guaranteed to be inside the ϵ-neighbourhood. To provide two different situations for comparison we choose the two initial starting settings as described in (Section 6.3.1). On the other hand, with these bounds-focused algorithms there are two possible means of comparison of the individual runs, each convenient for different type of analysis. The first is tracking the approach of the two bounds to the optimal value somewhere between them. We focus on this to better distinguish between behaviour of the bandits on the lower bound and the upper bound. The second and least important one is observing the gap between the two bounds going to zero, which provides a more general idea about the convergence of the algorithm.

Setting

Experiments were conducted on the **PEG** instance described in (Section 6.3.1) on both initial starting points as shown in (Figure 6.7). Moreover, each run was repeated both with the default discount factor $\gamma = 0.95$ and with a smaller one $\gamma = 0.9$.

Each of these types of problems were executed 5 times for each non-observable bandit algorithm with multiple different parameter values. The runs were set up for $5 \cdot 10^4$ discrete time steps, where one time step corresponds to a single update of both the lower and upper bound. This number of time steps corresponds to approximately 8-10 hours of solve time on the *Metacentrum* machine mentioned at the beginning of the chapter. All the runs had a unique random seed for the pseudo-random number generator.

The parameters selected for comparisons were decided by similar rules of thumb as in the previous experiments. The best found values for individual algorithms are compared later in this chapter in (Section 6.3.3).

Graphs

As in the experiments for stochastic games, the analysis includes demonstration on the graphs, even though, there are a few differences from the previous ones. Similarly, the y axis corresponds to a mean value over the 5 runs highlighted in full colour with the band of standard deviation around it. However, here the number of evaluations is not as big and the graphs are synoptic so there is no need for logarithmic scale on the x axis. For more technical details see (Section 6.2.2).

The least important type of graphs is the plot displaying convergence of the gap between bounding functions to zero. These graphs are very similar as those in (Section 6.2).

The second and more important graph type depicts how both bounds approach to the optimal value found by HSVI with linear programs. These graphs differ as two lines highlighted by the same colour belong to a single specific bandit algorithm with given parameter value. These are more descriptive of the course of optimization because they distinguish between the improvements in each respective bound and thus a deeper analysis can be made.

6.3.3 Individual bandits

In this short section we discuss the bandit algorithms individually with respect to the evaluated parameters. The interesting realizations are demonstrated on images, while the less important ones are only mentioned.

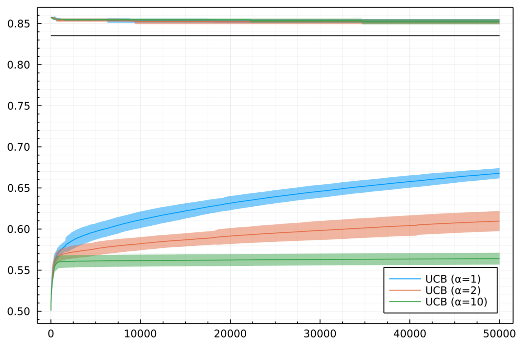
Surprisingly, most of the bandits do not change their behaviour much based on the tested parameters even though chosen really different. The Exp3 bandit algorithm, for example, for all the selections of the explorative parameter $\gamma \in \{0.1, 0.25, 0.5, 1.0\}$ behaved almost exactly the same for all the chosen runs with only slight changes within the standard deviation interval. The same thing can be said about the Best of N and even about the Successive elimination bandit for its exploration parameter $\alpha \in \{1, 2, 10\}$. Thus, the parameters for the bulk comparison of the bandit algorithms were chosen as those with the best mean value even though it was never out of the interval of standard deviations of the other runs.

Conversely, the very important realization is that the UCB multi-armed bandit is not suitable for the HSVI algorithm at all. While it and its observable complement dominated all other bandits on most instances, here it is one of the poorly performing bandits with only the Successive elimination being incomparably worse. And it seems that the parameter which

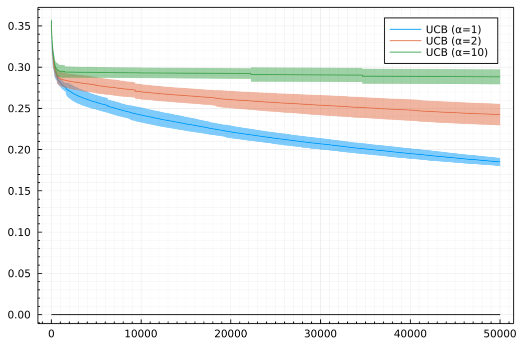
bandit	Best of N	ϵ-greedy	Successive elimination	UCB	Exp3
parameter	$N = 100$	$\epsilon = 0.3$	$\alpha = 1$		$\gamma = 0.1$

Table 6.3: The best found parameter values for each individual bandit algorithm tested on **PEG** OS-POSG within the B-HSVI framework. The argumentation about these values is presented in (Section 6.3.3).

controls tightening of the confidence bounds around average reward, set similarly as for the Successive elimination, $\alpha \in \{1, 2, 10\}$ has the opposite effect than the intuitive idea. The



(a) : The bounds of HSVI approaching the true value



(b) : The gap between bounds converging to zero

Figure 6.8: The behaviour of the UCB bandit algorithm on the **PEG** instance with initial settings as in 6.7a with the discount factor $\gamma = 0.95$. On the left figure is show dependence of values found by the upper and lower bound on the discrete time steps representing one point-based update and how the bounds approach the true value. The right graph depicts convergence of the gap between bounds to zero depending on the same discrete time steps.

parameter setting $\alpha = 1$ shows the best results while the $\alpha = 10$ does not converge almost at all when considering the lower bound. This goes against the conclusions from the previous experiments, where bigger exploration meant much better convergence. It could be caused by the fact that with smaller α the UCB gradually degrades to the greedy bandit and as will be shown in the next comparison the ϵ-greedy works well with the HSVI algorithm.

On the upper bound, the different parameters for the UCB do not show significant differences in convergence.

The best parameter over the all trials for ϵ-greedy bandit algorithm is $\epsilon = 0.3$, even though not by a large margin. This holds mainly for the lower bound, because in the upper bound the more aggressive exploration parameter value $\epsilon = 0.5$ converges slightly closer to the real value. However, when compared by the convergence of the gap to zero, the lower bound improvements outweigh the upper bound, thus making the $\epsilon = 0.3$ a better choice overall.

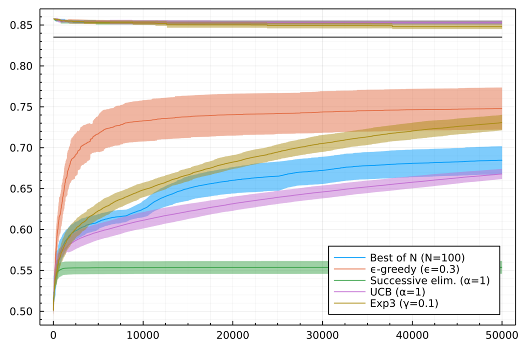
With these best parameters selected, we can compare all the bandit algorithms together.

6.3.4 Bulk comparison

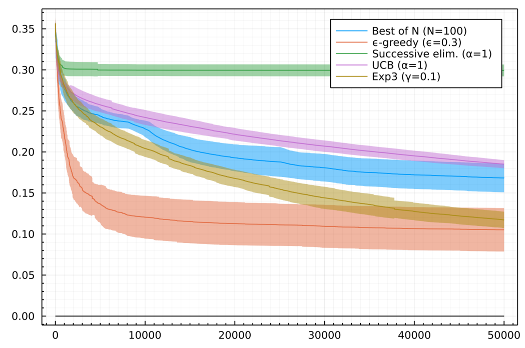
The table (Table 6.3) lists the parameters deemed best from the previous analysis in (Section 6.3.3). These parameters are used for the comparison of all bandit algorithms together in this section.

The change of learning rate slightly changes the optimal solution found by the linear programming utilizing HSVI algorithm. It would be expected, that solving the game with the lower discount factor $\gamma = 0.9$, the game would become easier for the approximate algorithms to solve. However, during testing of the B-HSVI algorithm, it converged slightly farther from the optimal value than when the default $\gamma = 0.95$ was employed. On the other hand, the standard deviation intervals surrounding the plotted means are noticeably tighter for the smaller discount factor than for the default one and thus the values were learned by the bandits with higher confidence.

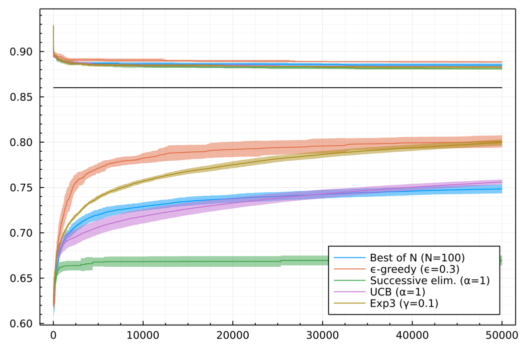
All the bandits evaluated on the **PEG** instance with both initial settings and with the discount factor 0.95 is shown in the (Figure 6.9). The courses of convergence are very similar



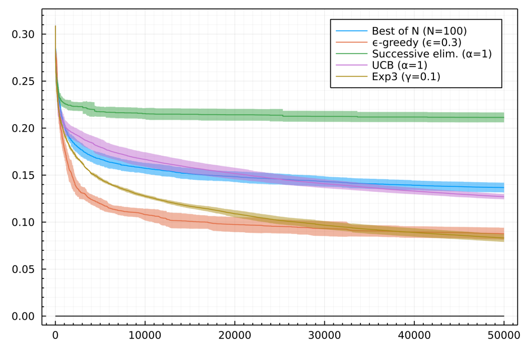
(a) : Value bounds for the original setting (Figure 6.7a)



(b) : Gap between the bounds for the original setting (Figure 6.7a)



(c) : Value bounds for the modified setting (Figure 6.7b)



(d) : Gap between the bounds for the modified setting (Figure 6.7b)

Figure 6.9: This figure depicts the convergence course of the multi-armed bandits with the best found parameters on the **PEG** instance with the use of discount factor $\gamma = 0.95$. The upper two graphs belong to the initial state in (Figure 6.7a), where the Pursuer is located in the upper left corner and initial belief b^{init} is pure in the bottom right corner. The bottom two charts then belong to the initial state in (Figure 6.7b), with the Pursuer in the middle and the Evader with uniform probability in each corner. The left graphs show dependence of both lower and upper bounds on discrete time steps corresponding to bandit updates and the right graphs show decrease of the gap between bounds for the same time steps.

for all the runs, only the absolute values differ, thus it suffices to discuss a single case.

The Successive elimination bandit algorithm performs by far the worst on the lower bound. The upper bound is by a tiny margin better, but it does not suffice to make it a suitable algorithm as the lower bound slowly converges to a completely different value.

Surprisingly, the Best of N algorithm is for many bandit updates better than the UCB. On the easier initial setting in the lower half of the (Figure 6.9) the UCB at last overcomes the Best of N, so this can be expected even for the harder problem in later iterations after the predefined horizon. This can be caused by the setting of the parameter N. Until all actions are tried, the search is conducted as uniform sampling over the actions and thus provides enough exploration power to improve quickly, but after the best action is fixed, no exploration is conducted by the bandit. The only exploration at that moment is the randomized selection of α-vectors, but evidently it is not enough and the more explorative UCB continues slowly improving. Note, however, that this holds only for the selected α for UCB, as for higher values it does not converge to the true value at all.

The best two algorithms are ϵ-greedy and Exp3 which both get closer to the real value, for the easier example in the bottom half of (Figure 6.9) the gap between bounds gets even under 0.1. Even though, the ϵ-greedy with $\epsilon = 0.3$ improves very quickly at the beginning, in later iterations the improvements slows down as are more often chosen random actions which do not move the values in the correct direction on average. Again, an adaptive annealing approach could help overcome this issue. In contrast, the Exp3 improves more steadily and does not slow down as much and based on the trends in the chart, it would continue converging while the ϵ-greedy would slow down more.

In the upper bound there are not very large differences between the individual bandit algorithms, but that is caused by the presolve procedure which gets the initial approximation really close to the real value. However, the bottom graphs in (Figure 6.9) suggest that the situation is different, because it seems like the ϵ-greedy is the worst, while Exp3 and Successive elimination are closest to the optimal value.

To obtain the best performance, the suggestion would be to use the Exp3 algorithm and fine-tune its γ parameter more thoroughly, or use a combination of different bandits for the upper resp. lower bound.

6.4 Summary

In this chapter, we compared the bandit algorithms on different instances of games *Tag* and *Chase*. Generally, it can be said, that neither Best of N, Successive elimination nor their observable variants are usable in the context of games as they discard some actions during the learning without possible correction and thus can prevent themselves from finding the optima.

From the other bandit algorithms the best option is the observable UCB multi-armed bandit in combination with the SQRT(t) averaging step function as they almost always converge quickly and without fluctuations in value. The Exp3 bandit is also very good but sometimes struggles with speed and frequent oscillation around the optimal value with the SQRT(t) step. The observable ϵ-greedy usually performs well but cannot compete with the two before. Unsurprisingly, the non-observable variants of ϵ-greedy and UCB perform poorly as they seek to find a single best action. These two algorithms converge slowly and often to some suboptimal value.

It turns out, that the average play employed in the observable bandits is a powerful factor in quality of the solution. Also, the speed increasing SQRT(t) step is convenient and works really well for the average play UCB. The other algorithms are then caused to oscillate because of it. Perhaps, some function which decreases more rapidly than $\frac{1}{\sqrt{t}}$ but slower than $\frac{1}{t}$ could be proposed and lead to better results without fluctuations.

Then, we focused on the OS-POSG model and the proposed B-HSVI algorithm, which uses multi-armed bandits as well. This time, however, only the non-observable variants as they fit the OS-POSG model more. The comparisons of the bandit algorithms were conducted on a game of *Pursuit-Evasion* of size 3×3.

The experimental evaluation showed surprising results as the UCB algorithm whose observable complement mostly dominated in the context of observable SGs, now does not perform well except for a very specific choices of parameters. On the other hand, the ϵ-greedy algorithm outperformed its expectations as it did not converge as good in stochastic games. The Exp3 algorithm was consistently good and fulfilled its promises gained by performing well before, except for the fluctuations caused mainly by the alternative step function $\mathrm{SQRT}(t)$.

To conclude this chapter, we evaluated the bandit algorithms on two large domains of problems and compared their performance on specific instances. Some of them exceeded their anticipated results, some did not fulfil them. If a single bandit algorithms needed to be selected for each domain, it would be the *Observable UCB* bandit for stochastic games as it is able to converge to real values even for states with randomized strategies and the *Exp3* for One sided partially observable stochastic games as it performed consistently good for both upper and lower bound and showed more potential even after the fixed discrete horizon in the number of bandit updates.

Chapter 7
Conclusion

To conclude this thesis, we provide an overview of the discussed topics, the summary of the achieved results and state possible further improvements of the proposed methods and approaches.

7.1 Thesis overview and contributions

In (Chapter 2), we presented the used formal models for sequential reasoning, with a single or multiple independent agents in different types of environment. The simplest models for single-agent settings, MDPs and the generalizing POMDPs, were discussed together with their most important properties, which are essential for the more complicated formalisms. Also, the uncertainty about the state of the environment introduced by POMDPs and the means to eliminate the following problem of the infinite search space were stated as well.

The same was done for a multiagent setting, i.e. an environment where multiple independent units interact and influence each other's decisions. This setting is thoroughly described by the game theory, so all the relevant and important notions were explained, with focus on two-player zero-sum games. Special attention was given to stochastic games, the partially observable stochastic games and the one-sided subset, which was the main topic of this thesis.

The (Chapter 3) contains description of the existing methods which are guaranteed to find exact or approximate solutions of problems formalized as one of the models mentioned in (Chapter 2). The properties of these algorithms were investigated and possible disadvantages, which motivate the creation of this thesis, were stated. The main problem of the presented algorithms is the usage of linear programming framework, which especially for the non-observable games grows in size rapidly making very large instances intractable.

The next chapter (Chapter 4) introduces the multi-armed bandit algorithm framework as a standard tool of reinforcement learning and the main mean of replacement of the linear programs in the standard methods. The basic principles are presented and the standard definitions of stochastic and adversarial bandits are listed. Specifically, the bandits discussed are *Best of N*, *ϵ-greedy*, *Successive elimination* and the *UCB* bandits from the stochastic class and the *Exp3* as an adversarial algorithm.

In (Chapter 5) we modified the methods and algorithms discussed in the previous chapters to fit them more into the game theoretical context and to remove the need for linear programming in the existing solving algorithms. First, we adapted the standard bandit algorithms to contain more stochastic elements, for example changing the order of arms to prevent selecting the same arm all the time by order-dependent operations. Second, we proposed so-called observable variants of the stochastic bandits which accept the actions of the opponent as observations to form average play strategies of the adversary and leverage this statistics to make better decisions. Then, we introduced the *bandit iteration* algorithm which is derived from the value iteration method but replaces solving the stage game by a linear program with gradually selecting actions by bandit algorithms and learn the optimal strategies. Last, we incorporated the bandit algorithms into the HSVI algorithm for OS-POSGs to create *B-HSVI* algorithm which again replaces stage games with iterative updates of the bandit algorithms and learning Nash equilibria.

In the last (Chapter 6), we compared convergence of the altered multi-armed bandit algorithms from (Chapter 5) both in the *bandit iteration* and *B-HSVI* frameworks on specific instances of stochastic games and one-sided partially observable stochastic games. As stated in the introduction (Chapter 1), the experiments focused more on the stochastic games as the area of bandit algorithms was not fully investigated there, and the partially observable superset is just a harder problem. Thus, it is sensible to first thoroughly explore the perfect information games before moving to the imperfect information.

In the context of stochastic games, we compare the convergence of the average-play variants to the non-observable bandits. From the conducted experiments it results, that the observable algorithms are superior to the standard ones as they often converge faster and closer to the optimal values. Two functions serving as averaging rates were compared, the first being the regular arithmetic average $\mathrm{LIN}(t) = \frac{1}{t}$, where $t \in \{1, 2, \dots, T\}$ is the time step of the optimization, and the second being a faster learning factor $\mathrm{SQRT}(t) = \frac{1}{\sqrt{t}}$. The best bandits according to the experimental evaluation were the observable UCB together with the $\mathrm{SQRT}(t)$ learning step and the Exp3 algorithm. These two managed to converge to the optimal value even in states with randomized optimal strategies. On the other hand, the Best of N, Successive elimination and unobservable UCB perform very poorly in those states.

Then, the same comparison was conducted on the domain of one-sided partially stochastic games within the *B-HSVI* algorithm, but only for the standard bandits. From these comparisons, the best performing algorithms were the Exp3 and ϵ-greedy with high exploration $\epsilon = 0.3$. On the other hand, the UCB and Successive elimination almost never converged close to the optimal value.

7.2 Future ideas

This concludes the contributions of the bachelor thesis and only a few suggestions for a next work are listed as there is a space for improvement in the proposed algorithms.

As far as the algorithm for stochastic games, neither of the used averaging steps is perfect. The $\mathrm{LIN}(t)$ step causes slow convergence for all tested bandits, but in contrast the faster $\mathrm{SQRT}(t)$ showed large fluctuations in value at the later stages of the experimental evaluation. Some monotonously decreasing function, which however decreases faster than $\frac{1}{t}$, but slower than $\frac{1}{\sqrt{t}}$, could bring dramatic improvements.

For the OS-POSG domain, we designed the *B-HSVI* algorithm to use bandits of the same type for the upper and lower bound. However, the perfect information of the player 2 could be leveraged and lead to a better result. The bandits in the upper and lower bound belonging to the fully-informed player could possibly employ the average play approach similarly as the observable bandits for stochastic games, which showed much better properties.

The next large step is to use the bandit algorithms for another subclass of partially observable stochastic games, which is PO-POSGs, i.e. partially observable stochastic games with public observations [19], where both agents do not have perfect information and the search space is thus much larger. The HSVI algorithm for this domain is based on the similar principle of the refinement of the lower and upper bounds, even though, the situation is more complex, than in OS-POSGs.

Bibliography

[1] D. J. White, "Real applications of markov decision processes," *Interfaces*, vol. 15, no. 6, pp. 73–83, 1985.

[2] A. R. Cassandra, "A survey of pomdp applications."

[3] G. W. Greenwood and R. Tymerski, "A game-theoretical approach for designing market trading strategies," in *2008 IEEE Symposium On Computational Intelligence and Games*, pp. 316–322, 2008.

[4] J. S. Brown, "Why darwin would have loved evolutionary game theory," *Proceedings of the Royal Society B: Biological Sciences*, vol. 283, p. 20160847, Sept. 2016.

[5] T. Hazra and K. Anjaria, "Applications of game theory in deep learning: a survey," *Multimedia Tools and Applications*, vol. 81, pp. 8963–8994, Feb. 2022.

[6] A. Jain, K. Tripathi, A. Jatain, and M. Chaudhary, "A game theory based attacker defender model for ids in cloud security," 2022.

[7] Z. Liu, N. C. Luong, W. Wang, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "A survey on blockchain: A game theoretical perspective," *IEEE Access*, vol. 7, pp. 47615–47643, 2019.

[8] T. Smith and R. Simmons, "Heuristic search value iteration for pomdps," in *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI '04, (Arlington, Virginia, USA), p. 520–527, AUAI Press, 2004.

[9] K. Horák, B. Bošanský, and M. Pěchouček, "Heuristic search value iteration for one-sided partially observable stochastic games," 2017.

[10] A. Slivkins, "Introduction to multi-armed bandits," 2019.

[11] S. Russell and P. Norvig, *Artificial intelligence: A modern approach.* Pearson, 3 ed., 2009.

[12] Y. Shoham and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations.* Cambridge University Press, 2008.

[13] E. J. Sondik, "The optimal control of partially observable markov processes over the infinite horizon: Discounted costs," *Operations Research*, vol. 26, no. 2, pp. 282–304, 1978.

[14] J. Nash, John F., “Equilibrium points in n -person games,” Jan 1950.

[15] J. v. Neumann, “Zur theorie der gesellschaftsspiele,” Dec 1928.

[16] P. B. Larson, “Zermelo 1913,” 2010.

[17] K. Horák, *Scalable Algorithms for Solving Stochastic Games with Limited Partial Observability.* PhD thesis, Czech Technical University in Prague. Computing and Information Centre., jan 2020.

[18] W. A. J., O. F. A., and R. D. M., “Structure in the value function of two-player zero-sum games of incomplete information,” *Frontiers in Artificial Intelligence and Applications*, vol. 285, no. ECAI 2016, p. 1628–1629, 2016.

[19] K. Horák and B. Bošanský, “Solving partially observable stochastic games with public observations,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 2029–2036, Jul. 2019.

[20] L. S. Shapley, “Stochastic games,” Oct 1953.

[21] P. I. Cowling, E. J. Powley, and D. Whitehouse, “Information set monte carlo tree search,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 120–143, 2012.

[22] K. Richard, “Combining online learning and equilibrium computation in security games,” Master’s thesis, Czech Technical University in Prague. Computing and Information Centre., 2015.

[23] Metacentrum, “Resources.” `https://metavo.metacentrum.cz/pbsmon2/resource/halmir.metacentrum.cz/`, Last accessed on 06-05-2022.

[24] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, “Julia: A fresh approach to numerical computing,” *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.

[25] J. Forrest, S. Vigerske, T. Ralphs, L. Hafer, J. Forrest, jpfasano, H. G. Santos, M. Saltzman, Jan-Willem, B. Kristjansson, h-i gassmann, A. King, pobonomo, S. Brito, and to st, “coin-or/clp: Release releases/1.17.7,” Jan. 2022.

[26] T. H. Chung, G. A. Hollinger, and V. Isler, “Search and pursuit-evasion in mobile robotics,” *Autonomous Robots*, vol. 31, pp. 299–316, July 2011.

[27] J. Brož and O. Kubíček, “https://gitlab.fel.cvut.cz/brozjak2/HSVIforOneSidedPOSGs.jl.”

Appendix A

Time analysis in SG experiments

The execution time of the bandit iteration algorithm, as it was designed and mentioned in (6.2.2), strictly depends on the number of states of the stochastic game. In every round, for every state, two bandits, one for each player, have to select an action, receive the computed stage game value based on those actions and update their inner states.

In our case, the number of rounds is fixed for all runs of the algorithm to $T = 10^5$, the number of states, however, differs for every instance. This number is equal to the second power of either the number of empty cells of the maze in the case of *Tag* or the size of the set of all vertices in the case of *Catch*. To this count, the absorbing terminal state, which completes the SG model, must be added before computing the second power. The total number of states is then computed as $|S| = (K+1)^2$, where K is the count based on the type of game mentioned above. Thus, the performance degrades fast with larger number of states.

instance	no. of states	min	median	max
tag_2_01	16	25.1 s	27.1 s	39.1 s
Chase3	16	34.5 s	39.7 s	66.0 s
Chase4	25	2.3 min	2.4 min	4.1 min
Chase5	36	6.0 min	6.6 min	10.4 min
tag_3_04	49	17.3 min	18.5 min	19.9 min
tag_3_01	64	31.7 min	33.6 min	66.6 min
tag_3_03	64	32.8 min	33.7 min	35.9 min
tag_3_06	64	33.2 min	34.2 min	55.0 min
tag_3_02	64	48.2 min	50.6 min	53.1 min
tag_3_07	64	49.4 min	50.6 min	52.8 min
tag_3_05	64	49.6 min	51.5 min	57.5 min

Table A.1: Statistics of execution times for individual instances of both **Chase** and **Tag** processed over 10 independent runs. Each run was solved by bandit iteration for 10^5 iterations. The rows in the table are sorted in an ascending order by the median time.

(Table A.1) shows statistics about running times of the bandit iteration algorithm on individual instances of both games. The experiments were conducted according to (Section 6.2.2), so they ran for 10^5 iterations each, every different setting 10 times with random seeds.

Note that, the instances were solved by all bandits and some bandits are more computationally demanding than the others, which causes the significant spread between *min* and *max* times. The few cases, where the maximal execution time is much longer than the median, were very limited in number and always occurred within trials of a single bandit algorithm. This could be caused by damaged pre-compilation of the code, which would force Julia to compile during the first runs of the cluster job. Another possible origin of these outliers is temporary overload of the cluster.

The data confirm the claims above, that the execution time depends strongly on the number of states in the model. Those instances with the same number of states then differ in number of actions per state. The higher the number of actions in a state, the more computationally demanding the corresponding bandit is and thus the whole algorithm is slowed down. This phenomenon can be showcased on two groups of *Tag*, the first group being `tag_3_01`, `tag_3_03`, `tag_3_06` and the second `tag_3_02`, `tag_3_07`, `tag_3_05`. While the instances in the first group average approximately 5.4 actions per state (both tagger and evader actions are taken together), the second group averages around 6.5. actions per state. It can be seen that the difference of one action per state adds over the course of 10^5 iteration roughly 20 minutes.

Appendix B

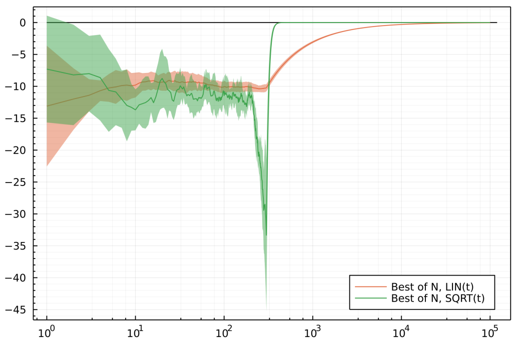
Performance of individual multi-armed bandit algorithms in stochastic games

B.1 Best of N

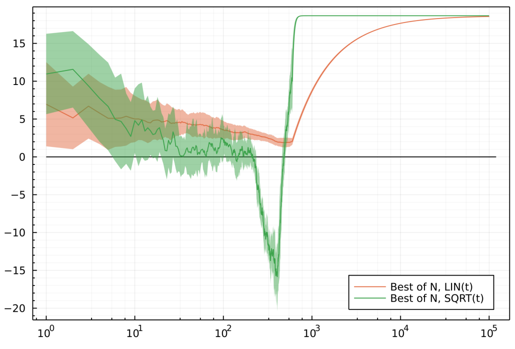
Firstly, we investigate the most basic multi-armed bandit algorithm and that is Best of N and its observable variant.

B.1.1 Mixed or pure strategies

Here, two states are sufficient to analyse the performance of this bandit algorithm and showcase its gravest weakness.



(a) : `tag_3_01` in state $s = (1, 1)$



(b) : `tag_3_02` in state $s = (4, 4)$

Figure B.1: The two graphs depict performance of the Best of N bandit, for $N = 100$, in starting states of two **Tag** instances, namely `tag_3_01` and `tag_3_02`. The left figure belongs to a state where the optimal strategy is pure, conversely the right figure to a starting state with mixed optimal strategy. The graphs represent the development of deviation of the returned value from the true optimal value in each iteration $t \in [0, 10^5]$. A comparison of the two step functions is displayed and in detail described in B.1.

Graphs (Figure B.1) depict, that after the first random search over all actions, until each was tried N times, the bandit found the single best pure strategy and played it to the end, thus receiving the same reward forever.

The first state is from the instance `tag_3_01`, where both the tagger and the evader are located on position 1 so a joint state $s = (1, 1)$. The map of this game (Figure 6.2b) suggests, that the optimal strategies of both players are pure. The evader can only go down and thus the tagger can always hit him with the beam by shooting vertically. The graph (Figure B.1a) confirms that the Best of N bandit can and will find such a pure best actions for sufficiently large N and converge to the optimal value.

On the contrary, the second state $(4, 4)$ from `tag_3_02` (Figure 6.2c) has mixed optimal strategies. Neither player can commit to play a single action, because the adversary would exploit this information and obtain better outcome than in the case of a randomized strategy. For the evader, all directions are equally good and the tagger has to decide the direction of the beam as he is always able to hit the evader. But clearly he has to randomize, because he cannot know the direction of the evader. This can be verified by extracting the strategies from the value iteration algorithm, where the tagger chooses the beam direction with uniform probability. The chart in (Figure B.1) confirms the intuitive idea, that the Best of N bandit is not able to find the optimal value and converges to a different value. The deviation improves over the course of the random search, but once the single pure action is selected, the value recedes from the optimum.

B.1.2 Steps

From the graphs in previous subsection, we can also compare the two types of steps, LIN(t) and SQRT(t). The increased weights for the newer presumably more accurate values have no real benefit here. Although, the SQRT(t) converges to the resulting value much faster during the initial random phase, both of these step methods eventually converge to the same value. On the other hand, during the random search at the beginning, the SQRT(t) step deviates much farther.

B.1.3 Observable variant

As displayed in the example state plotted in (Figure B.2), the observable variant behaves almost exactly the same as the standard bandit algorithm. Here, the average play of the adversary matters only once and that is in the selection of the best action after each was tried N times. But, at that moment the average play is a uniform strategy, so it does not have any effect on ruling out less promising options. Other choices are either completely random or conversely fixed to a single action. Even though, from the graph it looks that the observable bandit converged exactly to the optimum, the intervals of standard deviation almost completely overlap and thus it cannot be confidently said that one reached the optimum while the other did not.

B.2 ϵ-greedy

The second bandit algorithm for analysis is ϵ-greedy. Be reminded, that the search parameter was posed to $\epsilon = 0.1$.

B.2.1 Convergence with average step LIN(t)

In this part we consider only the averaging step LIN(t), the other one will be discussed separately.

The ϵ-greedy does not have a problem in getting close to the correct values in easy states with pure optimal strategies. Although, it did not converge exactly to the value iteration

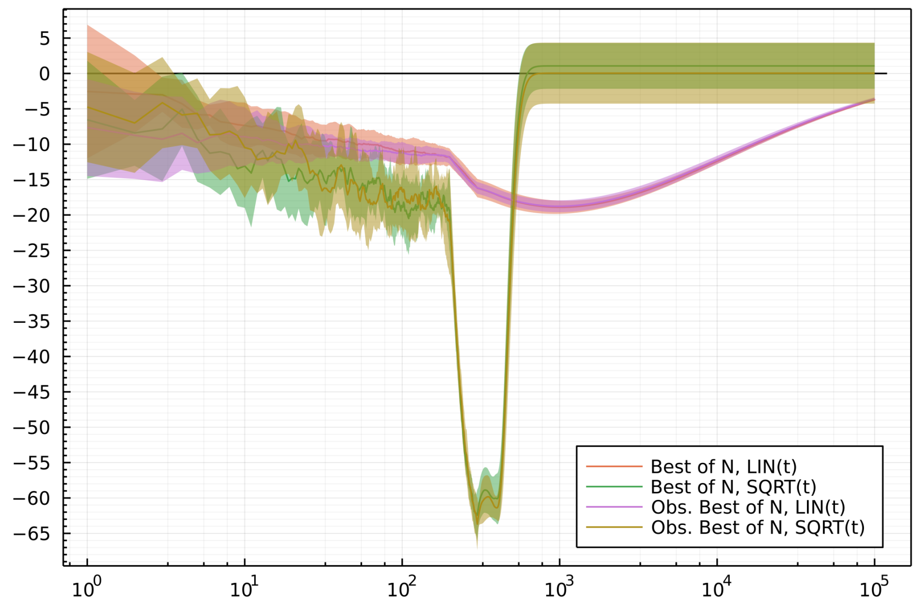


Figure B.2: The figure shows comparison deviations from ground truth of the observable and non-observable variant of the Best of N bandit on the instance `tag_3_01` in a joint initial state $s = (1, 3)$. It includes both types of step functions, LIN(t) and SQRT(t). The x axis represents iterations of the bandit iteration algorithm.

results for states with many actions and randomized decisions (thus more difficult to learn the strategies precisely), based on the demonstrated trend of the deviation going towards zero, it would eventually converge given many more iterations. The figure (Figure B.3) displays one easy and one hard state and the convergence in those respective states. It displays states $s_1 = (4, 4)$ of the instance `tag_3_02` (Figure 6.2c) and $s_2 = (4, 3)$ in the `Chase5` instance (Figure 6.1c). Moreover, from both of these figures, the observable variant outperforms the standard one and gets closer and faster to the optimum. In contrast to the Best of N bandit (Section B.1), ϵ-greedy selects best action 9 times out of 10 trials on average and thus the average play of the opponent influences the decisions in the observable variant.

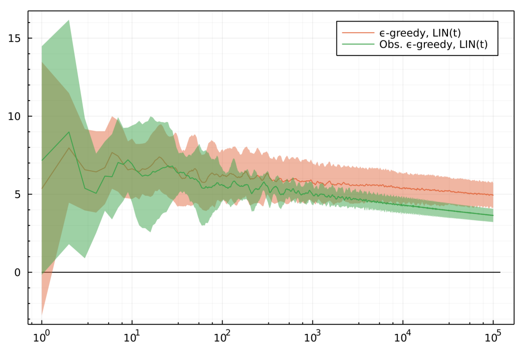
B.2.2 Accumulation step SQRT(t)

Generally, the ϵ-greedy bandit struggles with the intensive exploration at the end, causing bigger fluctuations in the accumulated value. This gets highlighted by the use of the step function SQRT(t). The algorithm definitely gets closer to the optimal value than the LIN(t) but then, due to the quite frequent random decisions, the solution rather oscillates around the optimum. However, in easy states, this averaging step is effective, especially with the employed average play.

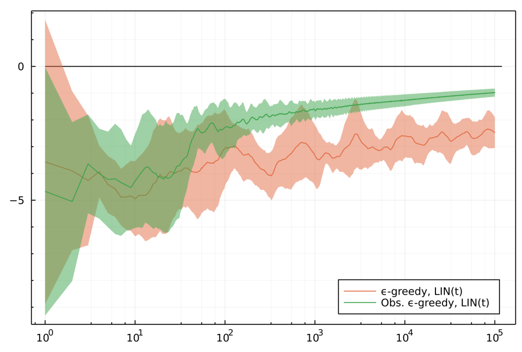
The figures (Figure B.4) depict these two cases on the observable bandit.

This makes the accumulation step SQRT(t) not very convenient for ϵ-greedy, because the one random explorative action selection shifts the value more than is desirable. This flaw could be improved by employing some adaptive way of setting ϵ for each iteration and cool down exploration in later rounds.

In the easiest states, the both step functions produce equivalent results. For example, in

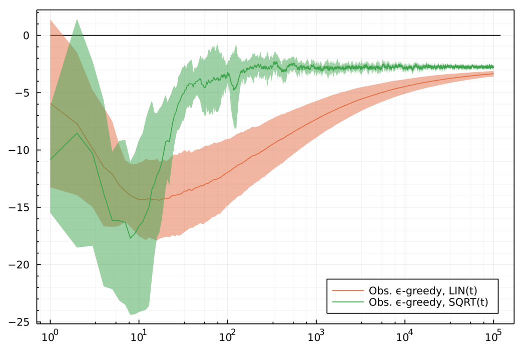


(a) : `tag_3_02` in state $s = (4, 4)$

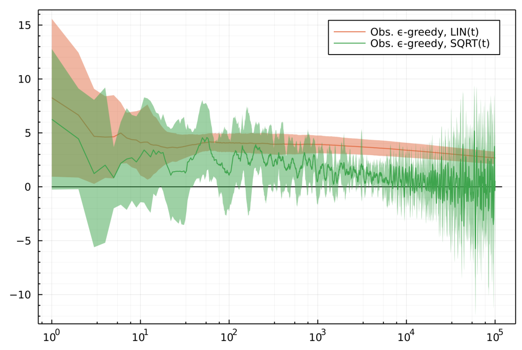


(b) : `Chase5` in state $s = (4, 3)$

Figure B.3: These graphs display convergence of the ϵ-greedy bandit algorithm to the values of states with mixed optimal strategies, when the $\mathrm{LIN}(t)$ step function is used to accumulate rewards. The y axis represents deviation from the value iteration optimal value shown as a black graph of a constant function in each iteration on x axis.



(a) : $\mathrm{SQRT}(t)$ outperforms $\mathrm{LIN}(t)$ on `tag_3_01` in $s = (1, 4)$



(b) : $\mathrm{SQRT}(t)$ oscillates in `tag_3_02` in $s = (1, 4)$

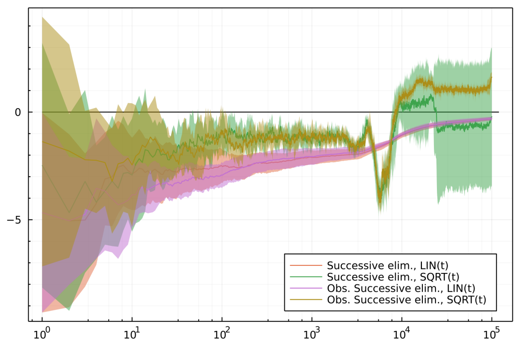
Figure B.4: This chart compares the two step functions $\mathrm{LIN}(t)$ and $\mathrm{SQRT}(t)$ and how they influence performance of the ϵ-greedy bandit in states with mixed optimal strategies. The performance is measured as a dependence of difference between the optimal value and the value from bandit iteration on iterations t.

states where only a single action is available for one of the players.

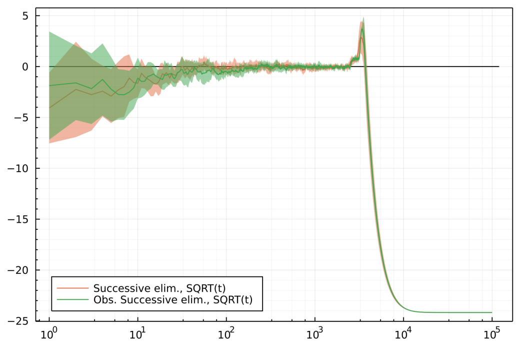
B.3 Successive elimination

The Successive elimination multi-armed bandit is the first examined algorithm, which adapts its behaviour based on the received rewards. It should perform better in learning randomized strategies, but due to the deactivation of actions when the confidence intervals do not overlap, it can still degrade to pure strategy even though it was not optimal. This also depends on the setting of the exploration parameter α.

In the figures (Figure B.5), the extreme occurrences of the phenomenon described before are showcased. On the left figure (Figure B.5a), we see an example, when the function $\mathrm{LIN}(t)$ slowly converges to the optimal value. The $\mathrm{SQRT}(t)$, however, gets close very quickly but then some arm which was important in the mixed strategy gets deactivated and the value shifts



(a) : Behaviour in joint state $s = (4, 3)$



(b) : Behaviour in joint state $s = (2, 1)$

Figure B.5: The figures depict graphs of convergence of returned values of the Successive elimination multi-armed bandit when used with the bandit iteration on `Chase5`. The learned values correspond to values read on the y axis and the iterations of the framework on the x axis.

somewhere else. The rapid changes in the graph of the obtained values exactly correspond to deactivation of some arm by the bandit. And while the LIN(t) function does not sway very much, the SQRT(t) step causes high fluctuations.

In the right figure (Figure B.5b), the algorithm converged almost exactly to the optimal value, but after the important action is removed from the decisions, the value drifts afar.

An example of a situation when the deactivation rule benefits the learning. In this state

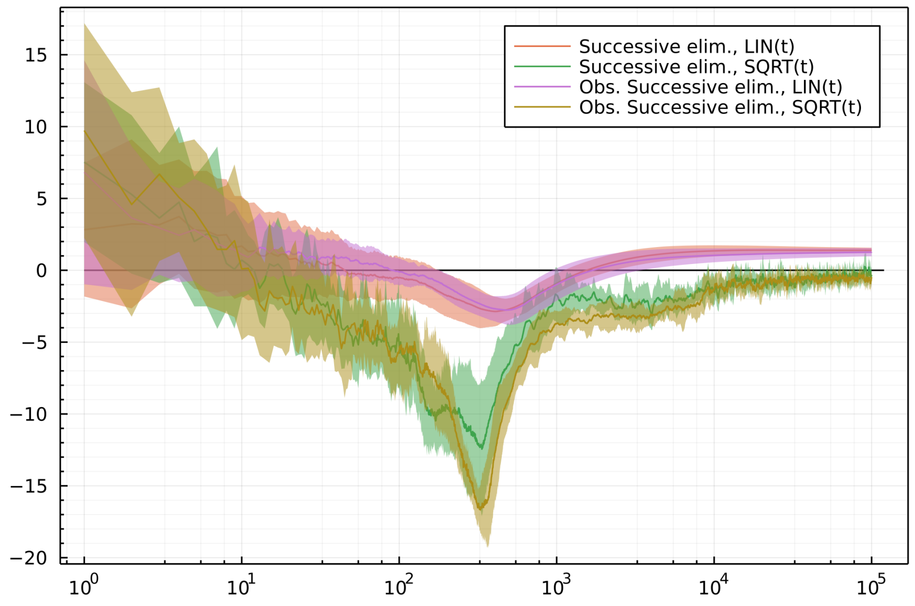


Figure B.6: The graph displays dependence of the deviation of the learned values from the true values on iteration step t on an instance `tag_3_02` in a state $s = (2, 6)$, where a mixed strategy is optimal. The two step functions are included in the comparison.

of `tag_3_02` (Figure 6.2c), the mixed strategy initially contained actions which was not in the optimal strategy and thus the graph of the value iteration dips below the optimal value. After the wrong action is not used any more, the value approaches the optimal zero deviation.

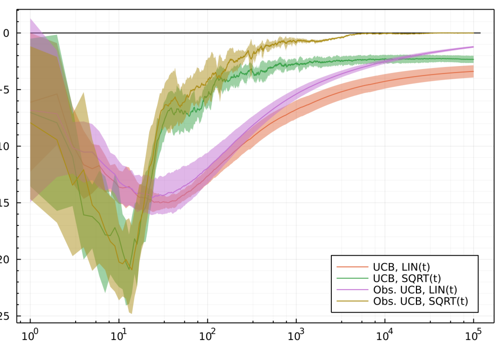
Moreover, the SQRT(t) step converges much closer than the LIN(t) arithmetic average. On the other hand, it again fluctuates much more aggressively.

The Successive elimination bandit improves over the ϵ-greedy in randomized decisions, but as well as Best of N, it can degrade to pure strategy by deactivating a wrong arm. This could be prevented by tuning the α parameter for a specific problem. Here, the SQRT(t) method can be beneficial, but can also lead to larger deviation when combined with the another problem mentioned first.

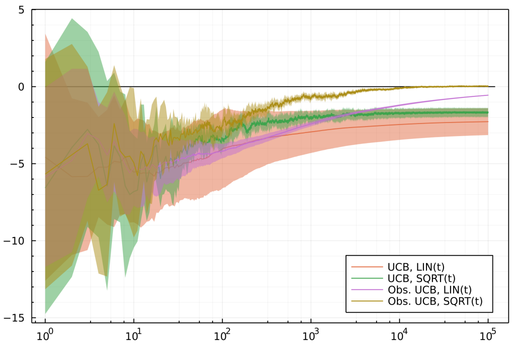
B.4 UCB

The Upper Confidence Bound bandit algorithm is an alternation of the Successive elimination bandit. In contrast, it does not deactivate presumably bad arms, but it always optimistically selects the best arm. Even though it performs well in learning MDPs, it is primarily designed to converge to the single best action, i.e. pure strategy. In these experiments, the search parameter is $\alpha = 20$. Now, we analyse it on stochastic games together with its observable variant.

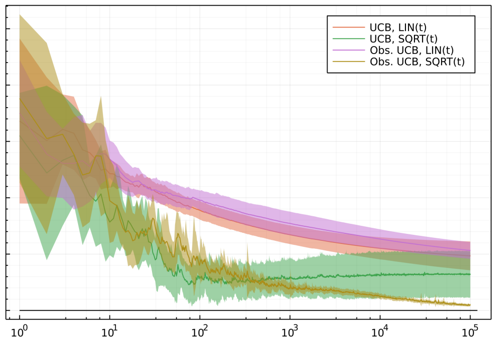
The first comparison is done on the `tag_3_01` and `tag_3_02` instances and is depicted in figure (Figure B.7). Here are shown two hard states from every chosen instance, which were



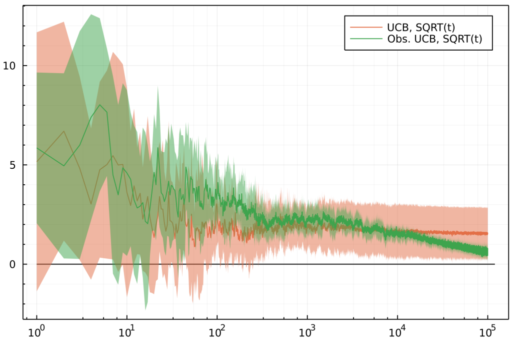
(a) : `tag_3_01` in a state $s = (2, 4)$



(b) : `tag_3_01` in a state $s = (3, 3)$



(c) : `tag_3_02` in a state $s = (2, 4)$



(d) : `tag_3_02` in a state $s = (4, 4)$

Figure B.7: The graphs display convergence of observable and standard UCB multi-armed bandits with both of the step functions, LIN(t) and SQRT(t), and the development of learned values change in time represented by discrete iteration steps. The comparison is done on two instances, `tag_3_01` and `tag_3_02`, in states where true optimal strategies are mixed.

either hard for previous bandits or they did not converge close to the optimum. From these graphs, it can be derived that the observable variant of UCB is significantly better than the non-observable UCB. In all these states, it either finds the optimum or continues pushing the deviation to 0 even after the standard algorithm finds its best pure strategy and converges to some suboptimal value.

Also, in the case of UCB, the SQRT(t) accumulation step boosts the speed of convergence for both of the algorithms without unnecessary fluctuations as it was with previous bandits. Even though the observable UCB with LIN(t) step appears to approach 0 as well, the SQRT(t) reaches this value much faster and most importantly stays there as opposed to other multi-armed bandits.

Here is another example of the same phenomenon as described above, but this time for a `Chase5` instance (Figure 6.1c). This particular state is interesting, because both bandits

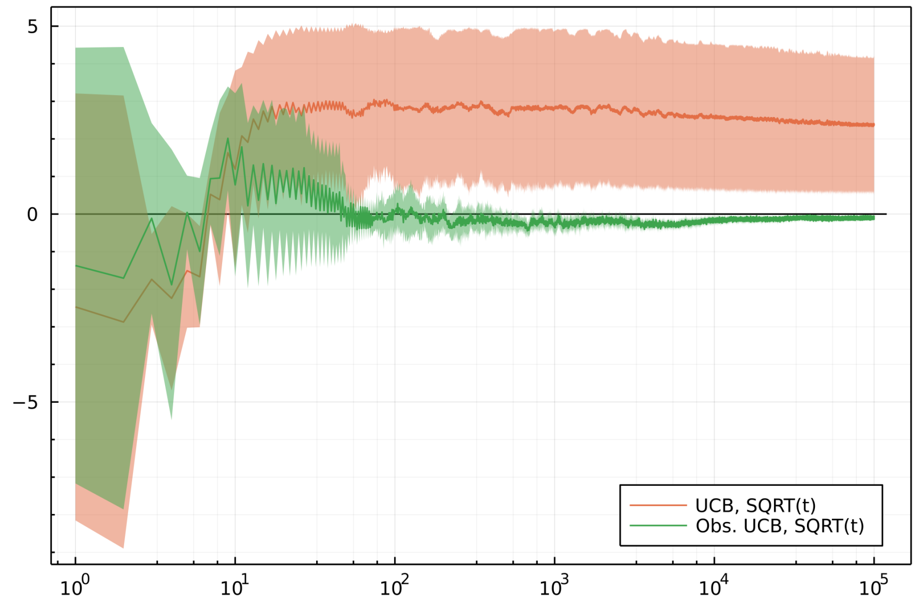


Figure B.8: This comparison chart depict performance of UCB in comparison to its observable counterpart on a **Chase** instance `Chase5` in a mixed-strategy state $s = (3, 3)$. The y axis shows deviation from the ground truth represented by the black constant function depending on the time stamp t on the x axis.

start in the same vertex of the graph in (Figure 6.1c), this node being the node 3. The graph visualization implies that both players would have to randomize their action selection, i.e. employ a mixed strategy. This is because if on of them committed to either using the loop and staying in the node 3 or use action 2 to reach vertex 5, his adversary would instantly leverage this. The runner would choose the opposite action, while the chaser would choose the same action. From the graph, UCB has clearly an issue with converging in a mixed strategy state, while the average play variant tries to respond to the frequent choices by the adversary.

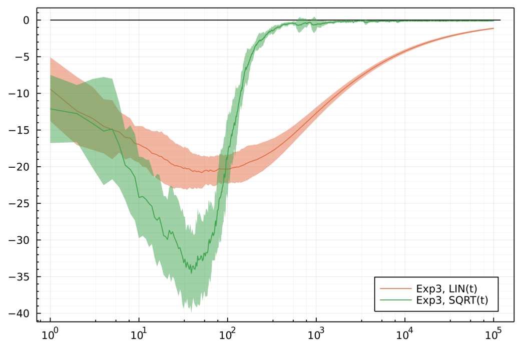
The observable UCB bandit converged to the optimal value in almost all states of all instances, or at least got very close. In contrast, the standard UCB sometimes struggles to reach the optimum and finds the best pure strategy. This showcases that the opponent's average play provides an actual advantage in bandit learning in stochastic games. It permits the bandit originally intended to search for a single best action, to rather find the best response to the average play of its adversary.

Moreover, in this case the SQRT(t) step function is far superior to the LIN(t) and significantly improves the speed of convergence as is documented in the figures in this section. Of all these combinations, the observable UCB bandit with the use of SQRT(t) dominated the other options.

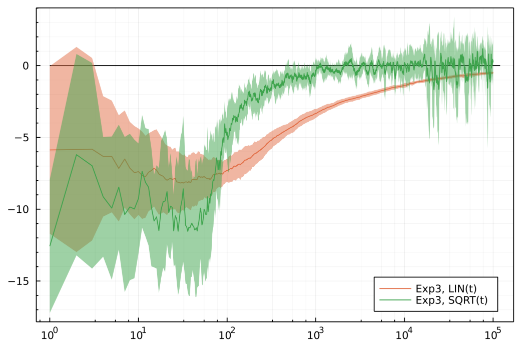
B.5 Exp3

The adversarial Exp3 bandit, in contrast to the others, was designed to learn a whole strategy rather than a single best action.

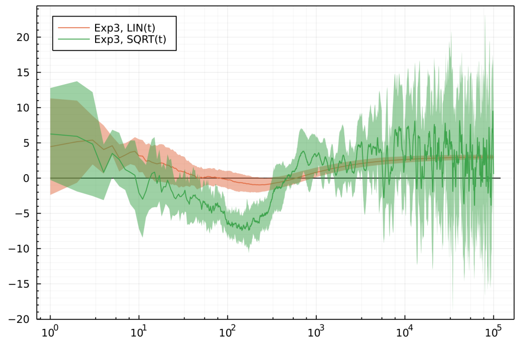
In figure (Figure B.9) are showcased 4 different examples of states with various difficulty. The first one (Figure B.9a), is the easiest with a pure strategy optimum and the Exp3



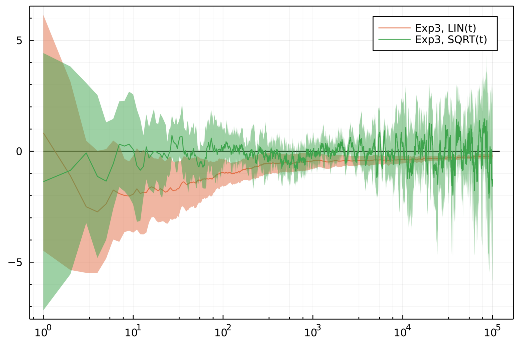
(a) : `tag_3_01` in a state $s = (2, 7)$



(b) : `tag_3_01` in a state $s = (3, 3)$



(c) : `tag_3_02` in a state $s = (1, 4)$



(d) : `tag_3_02` in a state $s = (3, 3)$

Figure B.9: These graphs display different convergence courses of the Exp3 algorithm in combination with either LIN(t) or SQRT(t) on two **Tag** instances in states where the optimal strategy is mixed. The graphs show dependence of deviation from the true value on the iterations t of the bandit iteration algorithm.

algorithm converges to this value. In this specific case, the SQRT(t) step again increases speed of convergence.

The rest of the examples are states, where mixed strategies has to be used to find the optima. Clearly, the LIN(t) step can find this value, even though slowly. The SQRT(t) step, on the other hand, reaches the value quicker but then oscillates around it with high amplitude.

This could be caused by computing the probability from the sums of so far received rewards in the exponential. Moreover, the bandit samples from these computed probabilities, so the explorative power is quite strong. Then, one such value which was then divided by the low probability of choosing the corresponding action can shift the sums in such a way that the learned probability of selection changes largely.

It can be said, that the SQRT(t) step function is not very appropriate for the Exp3 algorithm as the fluctuations influence the resulting values gravely. On the other hand, these values appear centred around the 0 deviation, so their mean could be taken as the optimal value.

Appendix C
Implementation details

In this appendix, the implementation details for both the bandit iteration and the B-HSVI algorithms are shortly described with the focus on how to execute the respective methods.

C.1 Bandit iteration

The implementation is located in a git repository on `https://gitlab.fel.cvut.cz/radajak5/HSVIforPOPOSG.git`. Even though, the repository is named after the subclass of partially observable stochastic games with public observations, this functionality is not ready yet. Instead, it offers solving methods for stochastic games for the first experimental evaluation, the value iteration (Section 3.3.1) as the reference method and bandit iteration (Section 5.2) as the tested method. Moreover, it provides means to load and generate the two SG instances, *Tag* and *Chase* 6.2.1.

To install this package, clone the repository, and then it can be imported to an environment with the standard `Pkg` julia commands.

Note, that for the sake of simplicity, only the functions essential for the game generators and the individual algorithms are presented.

C.1.1 Instance generators

An already created game can be loaded into a special `SG` structure, which can then be passed to the two algorithms, by calling its constructor with the *path* to the file as the only parameter.

```
sg = SG("./path/to/game/file.txt")
```

To generate an instance of the *Tag* game, one needs to create a `map.txt` text file, which contains map of the maze similar to the games in (Figure 6.1) and (Figure 6.2) and to the following example.

```
#####
# # #
# # #
#   #
#####
```

The stochastic game with this map is then generated by the following command.

```
sg = generateSGtag("./path/to/map.txt")
```

To generate an instance of the *Chase* game, a custom random graph generator is used with the following function.

```
sg = generateSGrandom(states;
    outdegree = 1:1,
    multiplicity = 1:1,
    penalty = -1:-1,
    victory = 10:10,
    seed = -1
)
```

The parameters have the following effects

- `states` ... the number of vertices of the generated graph,
- `outdegree` ... optional range argument to specify the number of edges going out from each vertex (the actual number is uniformly sampled from this range),
- `multiplicity` ... optional range argument to specify the number of potential targets of a single action, i.e. actions with stochastic effects,
- `penalty` ... optional range from which the rewards for a step without catching the opponent are uniformly sampled,
- `victory` ... the same as penalty but the rewards are for catching the opponent,
- `seed` ... optional initial seed for the pseudo-random generator

C.1.2 Algorithms and parameters

To execute the reference value iteration algorithm on a loaded `sg` instance with a precision $\epsilon > 0$, the following command is used.

```
V = valueiteration(sg;
    gap::Real = 1e-6,
    discount::Real = -1,
    output::Type{<:Output} = Brief
)
```

The important parameters are `gap`, which is the maximal absolute value between the last two approximations of V^* as described in 3.3.1, `discount` which can change the default discount factor $\gamma = 0.95$ to another value. The `output` parameter can be set to `Detailed` to return a vector of all approximations of V instead of only the last approximation in the case of the `Brief` option.

To execute the tested bandit iteration algorithm on a loaded `sg` instance serves the following command.

```
V = bandititeration(sg::SG,
    bandittype::Type{<:Bandit};
    epsilon::Real = 0.0,
    expgamma::Real = 0.0,
    expeta::Real = 0.0,
    alpha::Real = 0.0,
    maxtrials::Integer = 0,
    iterations::Integer = 1000,
    seed::Integer = -1,
    discount::Real = -1,
    output::Type{<:Output} = Brief,
    step::Type{<:Step} = LinT
)
```

The `banditttype` is one of the tested bandit algorithms with its corresponding parameter, i.e. `(Observable)BestOfN` with `maxtrials`, `(Observable)EpsilonGreedy` with `epsilon`, `(Observable)SuccessiveElimination` and `(Observable)UCB` with `alpha` and `Exp3` with `expgamma` and `expeta`. The other optional parameters have the following meanings:

- `iterations` ... number of iterations of the run,
- `seed` ... initial value for the pseudo-random generator
- `discount` ... change the default discount factor $\gamma = 0.95$ to another value
- `output` ... the `Detailed` option returns a vector of all approximations, matrix of bandit algorithms for every state and the initial seed instead of the last approximation in the case of `Brief` option,
- `step` ... choose the accumulation step function, either `LinT` or `SqrtT`.

These versions of the algorithms were used to conduct the experiments as in (Section 6.2). The other possible call options with non-essential parameters can be clearly understood from the code, but these listed options are sufficient to run the algorihtms.

C.2 B-HSVI

This B-HSVI algorithm continues the work of [27] in the git repository `https://gitlab.fel.cvut.cz/brozjak2/HSVIforOneSidedPOSGs.jl.git`. The implemented bandit alternative is located in the *bandits* git branch. The core of the algorithm is their work, the contribution of this thesis are the bandit algorithms and the other modifications in the mentioned branch.

As the repository and possible parameters is very large, we describe only the parameters essential for our purposes, the rest can be viewed in the mentioned repository.

The instance used for evaluation is located in *"HSVIforOneSidedPOSGs.jl/games/pursuit-evasion/no-fail/peg03.posg"*, but the other instances can be used as well. After the package is installed according to the standard procedure, i.e. cloning the repository and adding the package to the desired environment, the only available function is `hsvi` in detail described below.

```
hsvi("./path/to/game/file.posg", epsilon;
    ub_value_method::String = "lp",
    stage_game_method::String = "bandit",
    alpha_vector_creation::String = "lp",
    evaluate_strategy::Bool = false,
    seed::Int64 = 42,
    bandit_type::Type{<:Bandit} = EpsilonGreedy,
    bandit_epsilon::Float32 = 0.1f0,
    bandit_maxtrials::Int64 = 10,
    bandit_alpha::Float32 = 10f0,
    bandit_gamma::Float32 = 0.1f0,
    alpha_vector_epsilon::Float32 = 0f0,
    alpha_vector_random_size::Int64 = 10,
    experiments_dir::String = "",
    change_discount::Float32 = 0f0,
    change_partition::Int64 = 0,
    change_belief::Vector{Float32} = Vector{Float32}([]),
    max_explores::Int64 = 50000
)
```

In the following list, only the parameters which can be *changed* to test the bandit algorithm are shown, the other parameters are necessary to remain set to the written values.

- `seed` ... the initial value for the pseudo-random generator,
- `bandit_type` ... one of the following bandits with its respective parameter: `BestOfN` with `bandit_maxtrials`, `EpsilonGreedy` with `bandit_epsilon`, `SuccessiveElimination` and `UCB` with `bandit_alpha` and `Exp3` with `bandit_gamma`,
- `alpha_vector_epsilon` ... probability of choosing α-vector uniformly from the set of best α-vectors of size `alpha_vector_random_size`, instead of selecting always the best one,
- `experiments_dir` ... directory for the output files of the course of search,
- `change_discount` ... different value as a discount factor γ than the default 0.95,
- `change_partition` ... different initial information set than the one specified in the game file,
- `change_belief` ... different to default initial belief over the states of the specified information set,
- `max_explores` ... maximal number of bandit updates of both value function bounds (the search is terminated after the closes return from the recursive call of EXPLORE procedure).

Note that the method to compute the value of the upper bound and creation of the new α-vector is still using the linear programming method. However, the stage game linear program was avoided by the use of the bandit algorithms.

This description of the execution of the B-HSVI algorithm is sufficient for our purposes. For more detail, visit the repository page.