Investigating the Practicability of Finite State Controllers for Large DecPOMDPs

Master Thesis

submitted by
Josha Landsmann

Matriculation number:
540640

Study programme:
M.Sc. Informatik

First reviewer:
JProf. Dr. Tanya Braun

Second reviewer:
Prof. Dr. Jan Vahrenhold

Münster, October 9, 2024

Abstract

In medical environments, hundreds of thousands of autonomous nanobots need to be coordinated, while their actions and observations are imprecise. This raises the requirement of scalability for planning problems, as conventional planning problems often have less than ten autonomous actors. Decentralized partially-observable Markov decision process (DecPOMDP) is a framework for modelling such sequential multi-agent planning problems, where each agent has no full knowledge about the state of the environment. A finite state controller (FSC) is a graph that is used to represent an agent's policy. Solving a DecPOMDP refers to finding a policy that maximizes the expected reward and is already NEXP-complete for DecPOMDPs with two agents. Exploring and therefore extending FSCs is also exponentially depend on the number of agents. Especially, when used for large DecPOMDPs with many agents, the handling of FSCs becomes intractable. However, in this thesis we investigate the practicability of FSCs for large DecPOMDPs. In lifted DecPOMDPs, the agent set is dvidided into groups of indistinguishable agents having the same capabilities. These groups allow to avoid redundant computations and improving the scalability. We present a novel approach on FSCs using lifted DecPOMDPs based on a heursitc policy iteration algorithm. We modify the algorithm to exploit the symmetries of lifted DecPOMDPs and improve the efficiency and therefore the scalability of FSCs. We implement the base algorithm and our modified version in a novel software framework for solving DecPOMDPs. Exploting the structures of isomorphic DecPOMDPs allows us to handle up to 20 agents, whereas in the ground case we handle only up to ten agents. Making furhter symmetry assumptions, we are able to handle more than 200,000 agents. This leads to the conclusion that FSCs are feasible for large DecPOMDPs, but require decent assumptions about symmetries within the agent set to significantly improve the scalability.

Acknowledgements

Calculations (or parts of them) for this publication were performed on the HPC cluster PALMA II of the University of Münster, subsidized by the DFG (INST 211/667-1).

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Settings, where multiple autonomous actors need to be coordinated while their local observations are incomplete and the communication between them is strongly limited, can be found in various application areas, e.g. multi-robot systems, networking and medical environments [13, 41]. The goal of planning problems is to enable the autonomous actors to decide how to act in different situations. In medical environments, these settings sometimes consist of more than 100,000 nano-agents [14]. This introduces a new requirement for effective scalability in planning problems given that conventional planning problems typically involve fewer than ten autonomous actors. Due to the intractability of planning problems, we investigate how to handle planning problems of this scale.

A framework to formalize planning problems is the decentralized partially-observable Markov decision process (DecPOMDP). DecPOMDPs enable modelling agents that can perform actions and make observations in each round. While the performed action is chosen by each agent, the sensed observation is solely affected by the current state. Furthermore, the system is always in a certain state, which in the case of partial observability is not known exactly. Since the actual state is driven by the performed actions and affects the observations being made, an approximate belief state can be derived and updated iteratively. The agents are cooperative and have a common goal: maximizing the joint reward, which is granted based on the current state of the system and the performed actions in that state [41]. Since the agents act autonomous, they need to decide when to perform which action. Due to uncertainty of the actual current state, a sequence of actions would be not feasible. Hence, each agent gets a policy assigned that defines which action to choose based on the history of actions and observations of that agent. One way of representing such policies are finite state controllers (FSCs). FSCs are graphs where each node represents some history of the agent. For each node a action selection is defined and based on the chosen action and the sensed observation the agent transitions to a successor node. In many cases stochastic FSCs are used where the action selection and successor nodes are given through a probability distribution.

Solving DecPOMDPs means to create a policy that maximizes the expected reward based on the initial belief state. The expected reward is calculated by the rewards expected in the first τ steps where τ defines the planning horizon. Solving DecPOMDPs with a finite planning horizon is NEXP-complete even for two agents [11]. For infinite planning horizons, DecPOMDPs be-

come undecidable [34, 41]. Thus, finding an optimal policy that maximizes the earned rewards is intractable even for small DecPOMDPs. Therfore, often approximate ϵ-solutions are produced or heursitics are used. The algorithm we implement in our work is based on the heuristic policy iteration (HPI) algorithm presented by Bernstein *et al.* [9]. They use stochastic FSCs to represent the infinite horizon policies of the agents and utilize so called exhaustive backups to generate one step policies for exploration. Latter procedure produces controller nodes in an exponential manner in every iteration. To evaluate the policies values, we need to solve the value function that is defined over every state and every combination of nodes of agents. Hence, its representation is already polynomial depend on the number of nodes and exponentially depend on the number of agents. Altough there is some effort to remove nodes that do not contribute to a high quality policy, the number of nodes growths exponentially with each iteration. Thus, it becomes harder in every round to solve the value function and evaluate the policy.

Much effort went into optimizing the achieved value for only small problems [16, 30, 57]. A recent approach on a compact representation of large DecPOMDPs are lifted DecPOMDPs [13]. This approach is based on the observation that in many large problems there are agents sharing the same action and observation capabilities. Thus, the agent set is divided into partitions based on their capabilities. Witihn these partitions, it may not matter which agent executes which action. In that case, we consider histograms of actions instead of permutations [14]. This assumption enables us to reduce the computational complexity of planning problems, as we reduce the combinationatorial space to consider. Another approach relies on the indistinguishability of agents within each partition. This leads to isomorphic agents acting indistinguishable for an identical history [14]. In the context of lifted DecPOMDPs, there has been some effort for finite-horizon planning problems [15] but none for infinite-horizon planning with FSCs, as war as we know.

We present an extendable framework for solving DecPOMDPs and implement the HPI algorithm as a showcase. Additionally, we investigate the practicability of FSCs for DecPOMDPs with numerous agents. Thus, we present a novel approach on FSCs using isomorphic DecPOMDPs and are able to perform a policy iteration on an isomorphic DecPOMDP with 20 agents, whereas traditional DecPOMDPs were just barely able to handle ten agents. We achieve this by exploiting isomorphic symmetries within the agent set that allows us reducing the number of local controllers and share a single controller across a partition which reduces the complexity of policy exploration. Furthermore, instead of evaluating the policy for the whole DecPOMDP in each iteration of the algorithm, we first perform an iterative policy ranking on a much smaller DecPOMDP before evaluating the policy for all agents. Finally, we show when additionally assuming representative observations that the number of agents can be increased up to 200,000.

The remainder of this thesis is structured as follows. In Section 1.1 we present related work on DecPOMDPs. Chapter 2 contains a formal introduction to (lifted) DecPOMDPs, FSCs and other fundamental principles. In Chapter 3, we present the HPI algorithm for DecPOMDPs. We introduce our implementation of the HPI algorithms in Chapter 4. Chapter 5 is about the extension of the algorithm and the implementation to support lifted DecPOMDPs. Then, we showcase our experimental results and compare them with the original algorithm in Chapter 6 and discuss them in Chapter 7. Finally, in Chapter 8, we conclude our work and provide an outlook on how the work can be continued.

1.1 Related Work

The DecPOMDP framework was introduced by Bernstein *et al.* [11]. It extends the partially-observable Markov decision process (POMDP) [6]. Latter framework has only a single agent that cannot observe the whole state and is itself an extension of the Markov decision process (MDP) [8], where the sole agent can observe the whole state. Apart from that, there exists the partially-observable stochastic game (POSG), where multiple agents cannot observe the whole space and can have conflicting goals, which can be seen as a generalization of the DecPOMDP [41]. Decentralized decision problems exist in various forms for more than 50 years [54]. To this date, DecPOMDPs are objective to extensive research in diverse contexts and variations [16, 33, 46].

In finite horizon DecPOMDPs, often planning trees are used for policy representation [15, 41]. In contrast FSCs are widely used to represent policies in infinite horizon DecPOMDPs [41, 49, 57]. Often, optimizing the achieved value is more important than the ability of handling DecPOMDPs with many agents [16, 30, 57]. Hence, most approaches remain intractable for large amounts of agents. A promising attemp in the context of infinite horizon DecPOMDPs, is the application of the concept of *Expectation Maximization* on DecPOMDPs which allows to plan problems with up to 2,500 agents [32, 56]. Other approaches on handling large DecPOMDPs focus on factorizing e.g. the state space but could only handle up to a few hundreds of agents [42, 43, 47].

We investigate lifted DecPOMDPs which is an approach that successfully reduces the representational size of DecPOMDPs [13, 14]. The term lifting originates from the context of probabilistic inference, where it is used to reduce the number of variables to consider [53]. Given the recency of this approach, it would appear that there has been a little research in the context of DecPOMDPs conducted on it, as far as we know [13, 14, 15].

2 Preliminaries

Before we dive deeper into the HPI algorithm, we explain the concept of DecPOMDPs and how these are structured. Afterwards, we introduce the concept of policies and how to examine the value of a DecPOMDP. Based on this concept, we give an example on how policies can be represented, by explaining FSCs and how they are used as policies. Last, we formally introduce lifted DecPOMDPs to be able to discuss possible algorithms later. The majority of definitions in this chapter is based on the definitions by Oliehoek and Amato [41].

2.1 DecPOMDP

The decentralized partially-observable Markov decision process (DecPOMDP) can be seen as a framework to model multi-agent decision making problems with uncertainty, in which all agents share a common state. Each agent has its own actions to perform and can only sense parts of their environment, which leads to an uncertainty about the concrete state they are currently in. While the performed actions are chosen by the agents, the sensed observations are solely affected by the current state. Additionally, a shared reward is earned, which is based on the joint action selection and the current state. In Section 6.1, we present a number of concrete examples of how a DecPOMDP may be structured.

Definition 2.1.1 (DecPOMDP)**.** Formally, a DecPOMDP can be defined as a tuple $\langle I, S, \vec{A}, T, R, \vec{\Omega}, O, \beta \rangle$, where

- I is a finite set of N agents.
- S is a finite set of states.
- $\vec{A} := \times_{i \in I} A_i$ is a set of joint action vectors, where A_i is a finite set of actions for agent $i \in I$.
- $T : S \times \vec{A} \to \Delta S = P(S'|S, \vec{A})$ is the transition function which defines the distribution of states to transition to based on a given state and the given joint action vector.
- $R : S \times \vec{A} \to \mathbb{R}$ is the joint reward function, which defines the reward earned, for performing the given joint actions in the given state.

- $\vec{\Omega} := \times_{i \in I} \Omega_i$ is a set of joint observation vectors, where Ω_i is a finite set of observations for agent $i \in I$.
- $O : \vec{A} \times S \to \Delta\vec{\Omega} = P(\vec{\Omega}|\vec{A}, S)$ is the observation function which defines the distribution of joint observation vectors sensed based on the given joint actions and ending in the given state.
- $\beta \in [0; 1]$ denotes the discount factor.

To express probability distributions, as denoted in the transition function, we use the Δ operator, which refers to the set of convex combinations. We define a convex combination and the Δ operator in the following.

Definition 2.1.2 (Convex combination)**.** We define $d \in \Delta X$ as a convex combination over a finite set X, where $d(x)$ represents the coefficient of a element $x \in X$, with $\forall x \in X : 0 \leq d(x) \leq 1$ and $\sum_{x \in X} d(x) = 1$. ΔX denotes the set of all possible convex combinations over X.

A POMDP is the special case of a DecPOMDP with a single agent ($N = 1$). In contrast to the more general POSG, the agents get rewarded jointly, which engages them to cooperate and not to rival.

Complexity The complexity of the representation of a DecPOMDP is exponentially within the number of agents N. This dependence can be seen in the sizes $\mathbb{T}, \mathbb{R}, \mathbb{O}$ of T, R and O respectively [14]. These can be derived from the definition of the functions, as they need to save a single number for each state and joint action (observation) vector. In the case of the transition function this is expanded, since it has to save a value for each triple of starting state, selected actions, and successor state.

$$\mathbb{T} \in \mathcal{O}(s^2 a^N) \qquad \mathbb{R} \in \mathcal{O}(s a^N) \qquad \mathbb{O} \in \mathcal{O}(s o^N) \tag{2.1}$$

where $s = |S|$, $a = \max_{i \in I} |A_i|$, and $o = \max_{i \in I} |\Omega_i|$. Because the representation itself is already exponential within the number of agents and optimally solving a DecPOMDP requires to consider every transition-, reward- and observation entry at least once, the computational complexity cannot be less than exponential. Oliehoek and Amato [41] showed optimally solving a finite horizon DecPOMDP is NEXP-complete, whereas for the infinite horizon case it is undecidable, until we are searching for an ϵ-optimal or heursitic solution.

2.2 Policy

A policy is a function which describes when to choose which action for an agent. In the single agent case, a policy can be an injective function from a (belief) state to an action (distribution). This statement does not hold in the

decentralized case, because each agent does only know about its own action choice and observation history, but the transition depends on the current state, the joint actions and observations. This is the reason why, we need a policy which decides based on the observation history of the agent.

Definition 2.2.1 (History)**.** Precisely, a local history $\Theta_{i,1:t} = (a_{i,1}, o_{i,1}, \ldots, a_{i,t}, o_{i,t})$ describes a sequence of local actions and observations for agent i, with $\forall_{k\in[1;t]} : o_{i,k} \in \Omega_i$ and $a_{i,k} \in A_i$. Whereas a joint history $\vec{\Theta}_{1:t} = (\vec{a}_1, \vec{o}_1, \ldots, \vec{a}_t, \vec{o}_t)$ describes a sequence of joint actions and observations with $\forall_{k\in[1;t]} : \vec{o}_k \in \vec{\Omega}$ and $\vec{a}_k \in \vec{A}$.

Based on the definition of a history, we define policies. We differentiate between local and joint policies. The former term describes the policy for a single agent while the latter term describes the combined policy for a set of agents.

Definition 2.2.2 (Local Policy)**.** Formally, a local policy can be defined as $\pi_i : o_{i,1:t} \to \Delta A_i$, where $P(a_i|\Theta_{i,1:t})$ represents the probability of action $a_i \in A_i$ being selected, when the history $\Theta_{i,1:t}$ is given.

Definition 2.2.3 (Joint Policy)**.** Given that the agent's action choice depends solely on the current history and not on the other agents, the joint policy is defined as $\pi : \vec{\Theta}_{1:t} \to \Delta\vec{A}$ where $P(\vec{a}|\vec{\Theta}_{1:t}) := \prod_{i\in I} P(a_i|\Theta_{i,1:t})$ defines the probability of the joint action vector $\vec{a}$ being selected for the given history $\vec{\Theta}_{1:t}$.

Given this definition, we formulate a recursive value function based on the policy with a finite horizon τ. The horizon describes how many first steps should be considered. Hence, the number of histories to consider becomes finite. The value function can be divided into two parts. On the one hand, there is the immediate reward which is granted for the given state and the action selection based on the given history. On the other hand, there is the sum over all possible pairs of successor states and successor histories weighted by their probability to be reached. The value function can be defined as follows [41]:

$$V(s, \vec{\Theta}_{0:t}) := \sum_{\vec{a}\in\vec{A}} P(\vec{a}|\vec{\Theta}_{0:t}) R(s, \vec{a}) + \beta \sum_{\vec{a}_{t+1}\in\vec{A}, \vec{o}_{t+1}\in\vec{\Omega}, s'\in S} P(\vec{a}_{t+1}|\vec{\Theta}_{0:t}) P(\vec{o}_{t+1}|\vec{a}_{t+1}, s') P(s'|s, \vec{a}_{t+1}) V(s', \vec{\Theta}_{0:t+1}) \quad (2.2)$$

with $\vec{\Theta}_{0:t+1} := \vec{\Theta}_{0:t} \circ (\vec{a}_{t+1}, \vec{o}_{t+1})$. $V(s, \vec{\Theta}_{0:\tau-1})$ denotes the recursion anchor. It consist only of the immediate reward, which is defined as $\sum_{\vec{a}\in\vec{A}} P(\vec{a}|\vec{\Theta}_{0:\tau-1}) R(s, \vec{a})$. The goal of solving a DecPOMDP is to find a policy

π^*, that maximizes the value for the DecPOMDP $V(b_o, \vec{\Theta}_{0:0})$. The value for a belief state is the sum of values of all states weighted by their probability and is defined as follows:

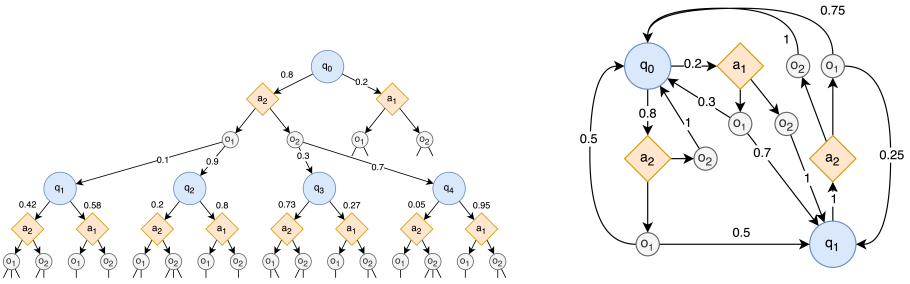
$$V(b, \vec{\Theta}_{j:k}) := \sum_{s \in S} b(s) V(s, \vec{\Theta}_{j:k}) \tag{2.3}$$

This explicit form of history representation is widely used [16, 33, 52], but has its limitations as we can only consider histories of a certain length. If the horizon would not limitate the length of histories, these become infinitely long and intractable. To be able to handle long running problems with many steps and thus handle an infinite horizon another representation is needed. To mitigate this problem, we define FSCs in the following section. Instead of an infinitely large sequence of actions and observations, we use internal states which are represented by nodes of a graph. The transition from one node to another are defined by the chosen action and the sensed observation. Therefore, the current node of such a controller can be seen as an implicit representation of the current history.

2.3 Finite State Controller

After explaining what a policy is used for, we show how we represent the policy. In the context of finite horizons DecPOMDP, trees are a common representation of the agent's policies [27, 28, 41, 50]. Trees can represent various outcomes of actions as well as invoke the history of the agent. By using stochastic edges, they can represent some uncertainty as well. A downside of this representation is that, due to the explicit representation of the history, the trees become infinitely large for problems with infinite horizon.

To mitigate these limitations, we utilize graphs to represent the policies of the agents. The usage of FSCs is proposed by various approaches in the context of infinite-horizon DecPOMDPs [36, 46, 49, 51, 57]. A graph can represent linear histories like a tree, but also share the same node for similar or reoccuring situations, where the agent needs to act the same. As the name suggests are FSCs finite graphs that can be cyclic, this way we can represent the infinite amount of histories with a finite set of controller nodes. In fact we use stochastic FSCs, where the edges form probability distributions for the action selection and the transition of the node. Figure 2.1 visualizes both approaches. Analogous to the policy, we distinguish between local and joint FSCs, which represent the policy for a single agent or the whole set of agents, respectively.



(a) Policy Tree **(b)** FSC

Figure 2.1: A visualization of an arbitrary policy tree (Fig. 2.1a) and an arbitrary FSC (Fig. 2.1b). Blue circles represent the nodes, whereas the orange diamonds visualize the action selection and the grey circle the sensed observations. Each edge is labelled with the probability for the agent to follow that edge.

2.3.1 Local Finite State Controller

A local FSC represents a single agent's policy and is specific to that agent.

Definition 2.3.1 (Local finite state controller)**.** We define a local controller for agent $i \in I$ as $C_i := \langle Q_i, \Omega_i, A_i, \psi_i, \eta_i \rangle$, where

- Q_i is a finite set of M_i nodes.
- Ω_i is a finite set of inputs, here equal to agent i's observations.
- A_i is a finite set of outputs, here equal to agent i's actions.
- $\psi_i : Q_i \to \Delta A_i = P(A_i|Q_i)$ is the action selection function, defining a distribution of actions to select, depending on the current node.
- $\eta_i : Q_i \times A_i \times \Omega_i \to \Delta Q_i = P(Q_i'|Q_i, A_i, \Omega_i)$ is the transition function, defining a distribution of succeeding nodes, depending on the current state, the selected action, and the observation made.

Because we update the current node based on the selected action and the sensed observation, we can represent the current history of the agent with a single node of the controller. Thus, we can redefine our policy to use controller nodes instead of histories by defining $\pi_i := \psi_i$.

2.3.2 Joint Finite State Controller

As we now can represent a local policy with a single FSC, we want to extend the joint policy, to be represented by FSCs as well. Thus, given a local controller for every agent, we can also define a joint controller as a product of the local controllers.

Definition 2.3.2 (Joint finite state controller)**.** We define a joint controller as $C := \langle \vec{Q}, \vec{\Omega}, \vec{A}, \psi, \eta \rangle$, where

- $\vec{Q} := \times_{i \in I} Q_i$ is a finite set of joint nodes, where Q_i is the set of nodes from C_i.
- $\vec{\Omega} := \times_{i \in I} \Omega_i$ is a finite set of joint inputs, where Ω_i is the set of inputs from C_i.
- $\vec{A} := \times_{i \in I} A_i$ is a finite set of joint outputs, where A_i is the set of outputs from C_i.
- $\psi : \vec{Q} \to \Delta\vec{A} = P(\vec{A}|\vec{Q})$ is the joint action selection function.
- $\eta : \vec{Q} \times \vec{A} \times \vec{\Omega} \to \Delta\vec{Q} = P(\vec{Q}\,'|\vec{Q}, \vec{A}, \vec{\Omega})$ is the joint transition function.

Analogous to the local policy, we can redefine the joint policy as $\pi := \psi$. Since the action selection function, as well as the node transition function of the local controller are independent from each other, we can define the joint action selection function as:

$$P(\vec{a}|\vec{q}) := \prod_{i \in I} P(a_i|q_i) \tag{2.4}$$

and the joint node transition function as:

$$P(\vec{q}\,'|\vec{q}, \vec{a}, \vec{o}) := \prod_{i \in I} P(q_i'|q_i, a_i, o_i) \tag{2.5}$$

2.3.3 Value Function for a Joint Finite State Controller

Now, we can define a value function for a given joint FSC. In contrast to the value function for a finite-horizon policy (cf. Equation (2.2)), this value function is defined over an infinite horizon. Hence, there exists no recursion anchor and this value function forms a linear equation system with an equation and an unknown for each pair of $s \in S$ and $\vec{q} \in \vec{Q}$. Apart from that, both value functions are analogously structured: First comes the immediate reward and after that the summation over all possible successors weighted by their probability. The following equation defines the value function for a joint FSC:

$$\begin{aligned} V(s, \vec{q}) := & \sum_{\vec{a} \in \vec{A}} P(\vec{a}|\vec{q}) R(s, \vec{a}) + \\ & \beta \sum_{\vec{a} \in \vec{A}, \vec{o} \in \vec{\Omega}, s' \in S, \vec{q}\,' \in \vec{Q}} P(\vec{a}|\vec{q}) P(\vec{o}|\vec{a}, s') P(s'|s, \vec{a}) P(\vec{q}\,'|\vec{q}, \vec{a}, \vec{o}) V(s', \vec{q}\,') \end{aligned} \tag{2.6}$$

We can define the value for a belief state and a node vector analogously to Equation (2.3):

$$V(b, \vec{q}) := \sum_{s \in S} b(s) V(s, \vec{q}) \tag{2.7}$$

Assuming that the system of linear equations was solved, the value of a controller C can be defined as the maximum value for any joint node vector to start at:

$$V(b_o) := \max_{\vec{q} \in \vec{Q}} V(b_0, \vec{q}) \tag{2.8}$$

As we search for an optimal policy to maximize the expected reward for the initial belief state b_o, we may search for an optimal joint controller C^* to maximize the expected reward. Therefore, we define $C^* := \arg\max_C V(b_0)$.

Complexity Since we have a local controller for each agent, the size of the joint controller is polynomial within the number of agents. Because the value function is defined for every state and every joint node vector, and the number of joint node vectors is exponential within the number of agents, the size of the value function for a joint controller is also exponential within the number of agents. This dependence can be seen in the sizes $\mathbb{C}, \mathbb{V}$ of the joint controller C and its value function V. Because we only discuss infinite horizon problems, we need to evaluate the value function by solving a system of linear equations. Solving a linear equation system has a computational complexity of $\mathcal{O}(n^3)$ in general, assuming a matrix of the size $n \times n$ [19]. In addition, we need to consider the computational complexity of computing all $\mathcal{O}(n^2)$ coefficients for the matrix and the vector. The number of floating point operations required to calculate a single coefficient for the equation system is denoted as $\mathbb{E}$ and is deduced from the value function.

$$\mathbb{C} \in \mathcal{O}(NM^2ao) \qquad \mathbb{V} \in \mathcal{O}(sM^N) \qquad \mathbb{E} \in \mathcal{O}(a^N o^N s M^N N) \tag{2.9}$$

with $s = |S|$, $M = \max_{i \in I} M_i$, $a = \max_{i \in I} |A_i|$ and $o = \max_{i \in I} |\Omega_i|$.

To be more precise, the size of the joint controller comes from N local FSCs, where each of which has a transition function of the size $\mathcal{O}(M^2ao)$. As the calculated value function defines a value for each pair of a state and a vector of nodes, the size of the value function is as large as their are such pairs. The number of floating point operations required for a single coefficient to be calculated can be derived from the definition of the value function. It reveals that we need to iterate over all action-, observation-, node combinations and states, and perform a constant number of multiplications and summations per iteration. Equations (2.4) and (2.5) show that the action selection- and node transition probability consists of N multiplications. Because we need to

calculate these probabilities in each iteration, we have $\mathcal{O}(N) + \mathcal{O}(1)$ floating point operations per iteration. Together, this leads to the complexity $\mathbb{E}$ given in Equation (2.9).

2.4 Dominance

Next, we introduce the concept of dominance [9]. Since we maximize the expected reward for the initial belief state, we need to find those joint node vectors that have the highest value at the initial belief state. The value of a single node (for a given state) is given through the value of all node vectors with that node (for the same state) and therefore cannot be calculated on its own. We call a node that has higher value than all other nodes domintating and formally introduce this term in the following.

Definition 2.4.1 (Dominance)**.** A node $q_i \in Q_i$ dominates $q_i' \in Q_i \setminus \{q_i\}$, if for all belief states $b \in \Delta S$ and for each combination of nodes of the other agents $\vec{q_{-i}} \in \times_{j \in I \setminus \{i\}} Q_j$ the following inequation stands:

$$V(b, q_i', \vec{q_{-i}}) \leq V(b, q_i, \vec{q_{-i}}) \tag{2.10}$$

We can extend the definition for joint node vectors. A joint node vector $\vec{q} \in \vec{Q}$ dominates $\vec{q}' \in \vec{Q} \setminus \{\vec{q}\}$, if for all belief states $b \in \Delta S$ the following inequation stands:

$$V(b, \vec{q}') \leq V(b, \vec{q}) \tag{2.11}$$

Figure 2.2 visualizes the value functions of four arbritrary node vectors for an examplary DecPOMDP with two states. It shows that depending on the node vector that is queried for the value function returns higher or lower values with respect to the belief state. The diagram shows that joint node vectors can lead to a high value for a decent belief state, but can have a lower value for other belief states. Therefore, in some situations a node is not dominated by a single other node but by a convex combination of other nodes. So we refine our definition of dominance for this convex case.

Definition 2.4.2 (Convex Dominance)**.** A node $q_i \in Q_i$ is dominated by a convex combination of nodes $\delta_i \in \Delta(Q_i \setminus \{q_i\})$, if for all belief states $b \in \Delta S$ and for each combination of nodes of the other agents $\vec{q_{-i}} \in \times_{j \in I \setminus \{i\}} Q_j$ the following inequation holds place:

$$V(b, q_i, \vec{q_{-i}}) \leq \sum_{q_i' \in Q_i} \delta_i(q_i') V(b, q_i', \vec{q_{-i}}) \tag{2.12}$$

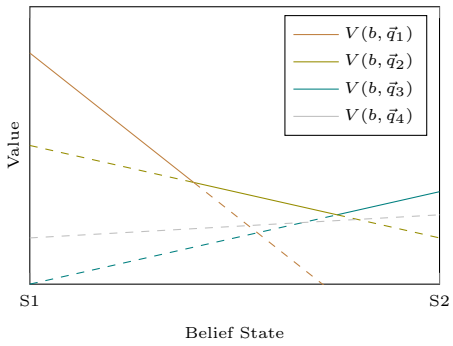


Figure 2.2: Dominance of value functions for an arbritrary DecPOMDP. The horizintal axis describes the belief state, which consists of two states. The vertical axis represents the value of the value functions for the according belief state. As these are arbritrary value functions, we omit values for the vertical axis. Dashed lines refer to dominated parts of the value function, whereas solid lines refer to dominating parts.

2.5 Lifted DecPOMDP

Because of the enhanced complexity of DecPOMDPs with respect to the number of agents, we need an advanced framework for modelling larger decentralized decision problems. One approach is to use lifting, where we utilize symmetries in the agent set, to divide it into $K \ll N$ partitions and reduce the complexity. Every agent in a partition has the same actions and observations. We discuss two types of such symmetries: counting and isomorphic symmetry. Those symmetries lead to two frameworks: counting and isomorphic DecPOMDP, which we explain in the remainder of this section. Ordinary DecPOMDPs are also called ground DecPOMDPs for better differentiation.

Braun *et al.* [13, 14] introduced counting as well as the more constrained isomorphic DecPOMDPs. Both of them have their origin in lifting probabilistic graphical models, where regularities in the structure of such models are utilized. Counting DecPOMDPs use counting random variables (CRVs) to represent how many agents perform which action, while isomorphic DecPOMDPs utilize parameterized random variables (PRVs) to reduce the agents of a partition to a single representative. Since we focus on the solving of such lifted DecPOMDPs, we omit further technical details on the background of lifting, which were explained in previous papers [13, 14, 53]. Most of the follwoing definitions are based on the definitions by Braun *et al.* [14]. We adjust them to syntactically fit the preceeding definitions.

2.5.1 Counting DecPOMDP

Counting DecPOMDPs are based on the assumption that for each partition, it does not matter which agent performs which action, but only how many agents perform an action. That assumption given, we no longer need to consider all

combination of actions for each agent, but only all possible histograms of action choices for each partition and the combination of those.

Definition 2.5.1 (Counting DecPOMDP)**.** A counting DecPOMDP can formally be described as a tuple $\langle I_C, S_C, \vec{A_C}, T_C, R_C, \vec{\Omega_C}, O_C \rangle$, where

- I_C is a partitioning $\{I_k\}_{k=1}^K$ of N agents, with $N_k = |I_k| > 0$ and $N = \sum_{k=1}^K N_k$.
- S_C describes a finite set of states.
- $\vec{A_C} := \times_{k=1}^K \#(A_k, N_k)$ is a set of joint action vectors, with A_k denoting the actions of agents in partition k.
- $T_C : S_C \times \vec{A_C} \to \Delta S_C = P(S'_C | S_C, \vec{A_C})$ describes the transition function as before.
- $R_C : S_C \times \vec{A_C} \to \mathbb{R}$ describes the reward function as before.
- $\vec{\Omega_C} := \times_{k=1}^K \#(\Omega_k, N_k)$ is a set of joint observation vectors, with Ω_k denoting the observations of agents in partition k.
- $O_C : \vec{A_C} \times S_C \to \Delta \vec{\Omega_C} = P(\vec{\Omega_C} | \vec{A_C}, S'_C)$ describes the observation function as before.

Definition 2.5.2 (Histogram)**.** $h \in \#(X, n)$ denotes a histogram over the elements of a set X, where $h(x)$ defines the number of items in a bucket $x \in X$ and $n = \sum_{x \in X} h(x)$ marks the total number of elements in the histogram. $\#(X, n)$ describes the set of histograms with n elements over X.

Complexity In contrast to regular DecPOMDPs which grow exponentially in size with the number of agents N, the size of a counting DecPOMDP is exponentially within the number of partitions K. This improvement is due to the reduced number of joint action and observation vectors, which summarize the actions and observations as histograms for each partition and can be observed in the sizes $\mathbb{T}_C, \mathbb{R}_C, \mathbb{O}_C$ of T_C, R_C and O_C respectively:

$$\mathbb{T}_C \in \mathcal{O}(s^2 a_C^K) \qquad \mathbb{R}_C \in \mathcal{O}(s a_C^K) \qquad \mathbb{O}_C \in \mathcal{O}(s o_C^K) \tag{2.13}$$

where $s = |S_C|$, $a_C = \max_{k=1}^K |\#(A_k, N_k)|$, and $o_C = \max_{k=1}^K |\#(\Omega_k, N_k)|$. The number of action histograms of a partition is given by the following equation [14]:

$$|\#(A_k, N_k)| := \binom{N_k + |A_k| - 1}{|A_k| - 1} \leq N_k^{|A_k|} \tag{2.14}$$

The number of observation histograms can be defined analogously. Thus, the complexity of representing counting DecPOMDPs is much smaller than

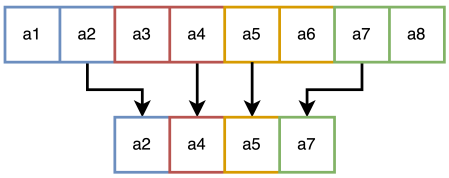


Figure 2.3: Exemplary grounding of an arbitrary vector consisting of four partitions with two elements each. The upper list of elements depicts the vector the grounding is applied on, whereas the lower list of elements depicts the resulting vector. The grounding itself defines which element of each partition is selected for the resulting vector. The colors emphasize the four different partitions.

for ground DecPOMDPs. In Section 5.1, we discuss how the framework of counting DecPOMDPs affects the size of the joint controller and the costs to evaluate and save the value function.

2.5.2 Isomorphic DecPOMDP

Isomorphic DecPOMDPs presume symmetries not only in the agent set, but also in the transition-, reward- and observation function. These symmetries allow a decomposition into factored functions that take representative action (observation) vectors [14]. These functions are described in the following:

$$T_I(S_I, \vec{A}_I) = \prod_{d \in D} \bar{T}_I(S_I, \vec{A}_I^d) \qquad R_I(S_I, \vec{A}_I) = \sum_{d \in D} \bar{R}_I(S_I, \vec{A}_I^d)$$
$$O_I(\vec{A}_I, S'_I) = \prod_{d \in D} \bar{O}_I(\vec{A}_I^d, S'_I) \tag{2.15}$$

where $\bar{P}(S'|\vec{A}, S)$ and $\bar{P}(\vec{\Omega}|\vec{A}, S')$ parameterize the functions $\bar{T}_I$ and $\bar{O}_I$.

For decomposition groundings are used. A grounding describes the selection of one representative per partition. In most cases the grounding is applied to an vector that is defined over all agents. The resulting vector from the application of the grounding contains a single element per partition, which are selected by the grounding. Such a grounding and the resulting vector are depicted in Figure 2.3. In the following, we formally define groundings.

Definition 2.5.3 (Grounding)**.** $D = \times_{k=1}^{K}[1..N_k]$ describes the set of groundings, whereas a grounding $d \in D$ is a distinct selection of one agent per partition. For a given vector $\vec{A}$ over all agents from all partitions, $\vec{A}^{\,d}$ describes a vector with a single element per partition, which selects the elements based on the grounding d. $L = \prod_{k=1}^{K} N_k$ denotes the grounding constant [15], which is equal to the number of existing groundings.

Braun *et al.* [14] showed, the decomposed transition-, reward- and observation function enables us to only consider peak-shaped action histograms when ranking policies. Ranking policies refers to comparing and finding high quality policies, without calculating the actual value of the DecPOMDP. Thus, during policy ranking it is sufficient to consider a representative DecPOMDP with $\forall_{k\in[1;K]} N_k = 1$ as the number of agents does not influence the decisions made but only the reward earned [14]. This enables us to search for a good FSC on a much smaller DecPOMDP and afterwards evaluating the value of the final FSC for the larger DecPOMDP. Unfortunately, during policy evaluation, we need to consider every observation and action combination, which we explain in detail in Section 5.2. We now define an isomorphic DecPOMDP formally.

Definition 2.5.4 (Isomorphic DecPOMDP)**.** An isomorphic DecPOMDP can be described as a tuple $\langle I_I, S_I, \vec{A}_I, T_I, R_I, \vec{\Omega}_I, O_I\rangle$, where

- I_I is a partitioning $\{I_k\}_{k=1}^K$ of N agents, with $N_k = |I_k| > 0$ and $N = \sum_{k=1}^K N_k$.
- S_I describes a finite set of states.
- $\vec{A}_I := \times_{k=1}^K \#(A_k, N_k)$ is a set of joint action vectors, with A_k denoting the actions of agents in group k.
- $T_I : S_I \times \vec{A}_I \to \Delta S_I = P(S'_I|S_I, \vec{A}_I)$ describes the transition function, which decomposes into $\bar{T}_I$.
- $R_I : S_I \times \vec{A}_I \to \mathbb{R}$ describes the reward function, which decomposes into $\bar{R}_I$.
- $\vec{\Omega}_I := \times_{k=1}^K \#(\Omega_k, N_k)$ is a set of joint observation vectors, with Ω_k denoting the observations of agents in group k.
- $O_I : \vec{A}_I \times S_I \to \Delta\vec{\Omega}_I = P(\vec{\Omega}_I|\vec{A}_I, S'_I)$ describes the observation function, which decomposes into $\bar{O}_I$.

Complexity As the decomposition allows us to only store the decomposed functions instead of the full joint functions, the size of an isomorphic DecPOMDP itself is only exponentially within the number of partitions K, which is assumed to be much smaller than the number of agents. The sizes $\bar{\mathbb{T}}_I, \bar{\mathbb{R}}_I, \bar{\mathbb{O}}_I$ of $\bar{T}_I$, $\bar{R}_I$ and $\bar{O}_I$ respectively, reflect this decomposition:

$$\bar{\mathbb{T}}_I \in \mathcal{O}(s^2 a_I^K) \qquad \bar{\mathbb{R}}_I \in \mathcal{O}(s a_I^K) \qquad \bar{\mathbb{O}}_I \in \mathcal{O}(s o_I^K) \tag{2.16}$$

where $s = |S_I|$, $a_I = \max_{k=1}^K |A_k|$, and $o_I = \max_{k=1}^K |\Omega_k|$. The independence of the number of agents allows to compactly represent DecPOMDPs with thousands of agents, as long as the number of partition remains small. In Section 5.2, we discuss to what extent isomorphic DecPOMDPs influence the costs to evaluate and save the value function as well as the size of the joint controller.

3 Policy Iteration for DecPOMDPs

Now, after we have a common understanding of the basic concepts of DecPOMDPs and FSCs, we present the algorithm we use for investigating the scalability of FSCs. There are different approaches on FSCs in the context of DecPOMDPs [9, 32, 33], but we intentionally focus on a straightforward algorithm to create a fundamental understanding of FSCs in combination with lifted DecPOMDPs. That is why we choose the HPI algorithm by Bernstein *et al.* [9]. This algorithm applies to ground DecPOMDPs, whereas our extension to support lifted DecPOMDPs is presented in Chapter 5. Next to the HPI algorithm, the authors present a set of policy iteration algorithms using FSCs to represent policies. This set consists of four algorithms, from which we decide to use the HPI, because their results show that the heuristic variant is more efficient, since it produces better results with less memory consumption as its competitors. Another algorithm is an extension of the HPI which involves the solving of non-linear programs to improve the structure of a controller without expanding it. This variant does indeed improve the resulting values, but increases the computational complexity of each iteration. Thus, we focus on the HPI to be able to perform as many iterations as possible.

The idea of the heuristic algorithm is to use a small set of belief states, so called belief points, to direct the pruning and to prevent overfitting of the initial belief state b_0. This way, we do not need to consider the whole state space, which can lead to significant performance improvements [50, 52]. Algorithm 3.1 gives a broad overview on the heuristic algorithm. First, we calculate belief points for each agent to direct the pruning. After that step, we start to evaluate the arbitrary controller by solving the system of linear equations of the value function. Then, we perform an exhaustive backup to explore the policy space. In the next step, we find nodes, which dominate at some belief point. Afterwards, we retain all nodes that contribute to the highest value for some belief point and prune the remaining by finding convex combination of dominating nodes for those. Those steps are repeated until the joint controller does not change during an iteration. We explain the different steps of the heuristic variant in more detail in the following Sections 3.1 to 3.5.

Algorithm 3.1 Heuristic Policy Iteration

Require:
DecPOMDP with joint controller C
Desired number of belief points $k \in \mathbb{N}$
Initial belief state $b_0 \in \Delta S$
Local policies $\forall_{i \in I} : \pi_i$

1: Generate k belief points, based on the policies π_i, starting from b_0.
2: Evaluate the value function V by solving the linear equation system.
3: Perform an exhaustive backup for exploration.
4: Find dominating nodes at belief points and retain them and their successors.
5: Prune dominating nodes with combination of nodes.
6: Evaluate the value function V by solving the linear equation system.
7: Terminate, if controller did not change, or else go to Step 3.

3.1 Belief Point Generation

The algorithm uses sets of belief points to direct the pruning. Algorithm 3.2 depicts how the belief points are generated. Briefly summarized, the algorithm needs an arbitrary policy for each agent π_i and the desired number of belief points per agent k. Given these inputs, the algorithm starts with the belief point generation, which is identical for each agent i: It generates a new belief point based on the previous belief point, starting at the initial belief state b_0. In each iteration, we select an arbitrary action and observation for the current agent. For all other agents, we iterate over all their actions and observations. To do so, we assume the other agents policies are fixed, and then iterate over all possible joint action and observation vectors with the given action and observation fixed for the current agent and calculate the transition probability to each state $s' \in S$. The action for the current agent is considered to be set, while the action selection probability is only calculated for the other agents, based on their local policy π_i, The probability of s' in the following belief state can be calculated as described in the following equation:

$$b'(s') = \sum_{\vec{a} \in \vec{A}', \vec{o} \in \vec{\Omega}'} \frac{\sum_{s \in S} b(s) P(\vec{a_{-i}}|s) P(s'|s, \vec{a}) P(\vec{o}|\vec{a}, s')}{\sum_{s, s'' \in S} b(s) P(\vec{a_{-i}}|s) P(s''|s, \vec{a}) P(\vec{o}|\vec{a}, s'')} \tag{3.1}$$

where $\vec{A}', \vec{\Omega}'$ describe the joint action (observation) vectors. These vectors are build by the selected action (observation) for agent i and all actions (observations) of the other agents. This process is repeated with each generated belief point until the number of desired belief points is reached. To ensure diversity, we only add new belief points that differ by a certain amount from all the belief points already generated.

Algorithm 3.2 Belief point generation

Require:
Desired number of belief points $k \in \mathbb{N}$
Initial belief state $b_0 \in \Delta S$
Local policies $\forall_{i \in I} : \pi_i$

1: $B \leftarrow \emptyset$
2: **while** $|B| < N$ **do**
3: $i \leftarrow |B|$
4: $B_i \leftarrow \{b_0\}$
5: $B'_i \leftarrow \{b_0\}$
6: **while** $0 < |B'_i|$ and $|B_i| < k$ **do**
7: Choose $b \in B'_i$
8: Fix other agents policies π_{-i}
9: Choose random $a \in A_i$ and $o \in \Omega_i$
10: Calculate following belief state b'
11: Add belief point, if new, to B_i and B'_i
12: If $|B_i| < k$, generate new policies and goto Step 5
13: Add B_i to B
14: **return** B

3.2 Value Evaluation

In order to assess the quality of the joint controller, we need to evaluate it. As we describe in Section 2.3 the value of a controller can be defined by Equation (2.8). Thus, we must know $V(b_0, \vec{q})$ for all joint node vectors $\vec{q} \in \vec{Q}$. These values can be calculated by solving the system of linear equations described in Equation (2.6). Since the values can refer to the value of each state and each other node, we need to calculate the value for all of them. This equation system consists of one equation and one unknown per $(s, \vec{q}) \in (S, \vec{Q})$. Each coefficient represents the probability to transition from $(s, \vec{q})$ to $(s', \vec{q}')$. Because of potential self loops, a sequential solving of those functions is rarely possible. After solving the system of linear equations, we know the value for each pair of state and joint node vector and calculate the value of the controller by finding the joint node vector $\vec{q} \in \vec{Q}$ that maximizes $V(b_0, \vec{q})$. The value of the controller for the initial belief state serves as an indicator for the quality of the controller.

3.3 Exhaustive Backup

Since we start with an arbitrary controller, we need to explore new policies. To do so, we create new nodes and add them to the agents' controllers. This procedure is called *exhaustive backup*. For each agent $i \in I$, we add a new

deterministic node q_i to the controller, for each action $a_i \in A_i$ and each injective function $f : \Omega_i \to Q_{\text{orig},i}$, with $P(a_i|q_i) = 1$, $\forall_{o_i \in \Omega_i} P(f(o_i)|q_i, a_i, o_i) = 1$ and $Q_{\text{orig},i}$ denoting the nodes of agent i before starting the *exhaustive backup*. To achieve a more comprehensive grasp of *exhaustive backup* the process is depicted in Figure 3.1.

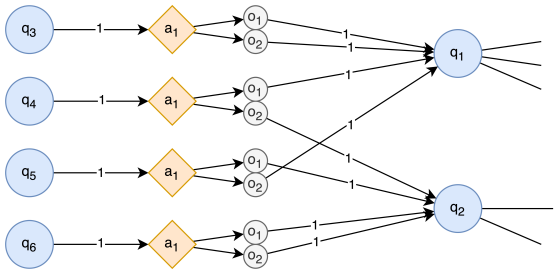


Figure 3.1: *Exhaustive backup* for an exemplary controller, with $Q_{\text{orig},i} = \{q_1, q_2\}$ and $\Omega_I = \{o_1, o_2\}$. In favor of readability, we only depict the *exhaustive backup* for a single action.

This procedure results in $|A_i||Q_{\text{orig},i}|^{|\Omega_i|}$ new nodes that we add to each agent's local controller. For each new joint node vector, we need to calculate the value for each state to find the dominating joint nodes in the next step. Since all new nodes link into the already existing nodes, we can calculate their values based on the results of the linear equation system. Thus, for a newly added joint node vector $\vec{q}$ we can calculate its value for each state $s \in S$ as follows:

$$V(s, \vec{q}) := \sum_{\vec{a} \in \vec{A}} P(\vec{a}|\vec{q}) R(s, \vec{a}) + \beta \sum_{\vec{a} \in \vec{A}, \vec{o} \in \vec{O}, s' \in S, \vec{q}\,' \in \vec{Q}} P(\vec{a}|\vec{q}) P(\vec{o}|\vec{a}, s') P(s'|s, \vec{a}) P(\vec{q}\,'|\vec{q}, \vec{a}, \vec{o}) V_{\text{orig}}(s', \vec{q}\,') \quad (3.2)$$

where V_{orig} denotes the value function, which is calculated during the the latest value evaluation.

3.4 Finding Dominating Nodes

After the *exhaustive backup*, we need to reduce the controller size. Otherwise, the number of joint node vectors become intractable. Therefore, we distinguish between valuable nodes that contribute high values at some belief point and less valuable nodes that do not contribute good values at any belief point. In contrast to Definitions 2.4.1 and 2.4.2, we now consider dominance over a

Algorithm 3.3 Linear program to find a dominating convex combination of nodes for a given node $q_i \in Q_i^*$

Variables: $\epsilon, \forall q_i' \in Q_i : x(q_i')$
Objective: maximize ϵ
Improvement constraints:
$\forall b \in B_i, \vec{q_{-i}} \in \times_{j \in I \setminus i} Q_j^* :$
$\sum_{s \in S} b(s) \left[\sum_{q_i' \in Q_i} x(q_i') V(s, q_i', \vec{q_{-i}}) - V(s, q_i, \vec{q_{-i}}) \right] \geq \epsilon$
Probability constraints:
$\sum_{q_i' \in Q_i} x(q_i') = 1 \qquad \forall q_i' \in Q_i : 0 \leq x(q_i') \leq 1$

specific (belief) state, instead of over the whole state space. Thus, we find the valuable nodes that dominate every other node of the same controller at one or more belief points. To do so, we iterate over all belief points B_i for each agent i and take the joint node vector with the highest value at that belief point:

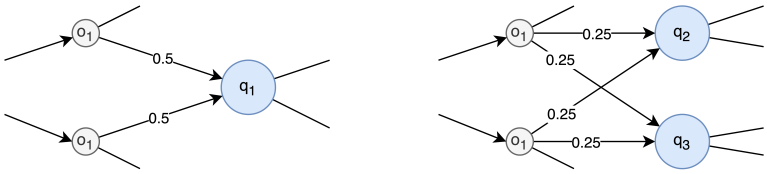
$$\vec{Q}^* = \{\arg\max_{\vec{q} \in \vec{Q}} V(b, \vec{q}) | \forall b \in B_i\} \tag{3.3}$$

We denote Q_i^* as the nodes of C_i that are part of some vector in $\vec{Q}^*$. Bernstein *et al.* [9] refered to these nodes as *initial nodes*, because they provide a good value for starting at these nodes for some belief point. We mark all nodes in Q_i^* as *initial nodes*. Then, we simultaneously prune all nodes from agent i that do not contribute to any belief point.

3.5 Combinatorial Pruning

Now, we check if the remaining nodes can be pruned. Given that the nodes may be dominanted only at certain points in the state space, we seek for a convex combination of nodes that dominates the node at all belief points (cf. Definition 2.4.2) for all initial nodes of the other agent's controllers. If such a convex combination exists, we replace incoming edges of the dominated node with weighted edges to the nodes the convex combination consists of.

Algorithm 3.3 describes the linear program that tries to find convex combination of nodes. For each node $q_i \in Q_i^*$, the program tries to find a convex combination of nodes of the same controller that achieves a higher value at belief points when combined with every node of every other agent. If the linear program finds a solution, $x(q_i')$ represents the coefficients of the convex combination of nodes, which then can be used to prune the node. To prune a node, we remove all outgoing connections of that node, replace all incoming connections with the convex combination, and finally delete the node from the controller. Figure 3.2 depicts a small excerpt from a FSC, to visualize the process of pruning and replacing a node with a convex combination of nodes.



(a) Before pruning **(b)** After pruning

Figure 3.2: Pruning node q_1 and replacing it with the convex combination c with $c(q_2) = c(q_3) = 0.5$. Fig. 3.2a shows the sector of the FSC before q_1 was pruned, while Fig. 3.2b shows the same sector after the pruning.

4 Implementation

Before we investigate lifted DecPOMDPs, we implement the HPI algorithm. Our goal is to build an application that not only supports FSCs and this specific algorithm, but can be extended to be used with various algorithms for solving DecPOMDPs. This objective is reflected in the structure of our application, which helps us with our future work to experiment with other algorithms, techniques, and their practicability for large DecPOMDPs. For this work, we focus on the HPI algorithm. Due to partially vague information given by Bernstein *et al.* [9], some parts of the implementation may differ from the original one. Nevertheless, we sticked to the original algorithm as close as possible for most of the time.

In this chapter, we first describe our architecture and other general aspects of the implementation before we give some details on which libraries we use and why. After these basics, we describe how we implement the different steps of the algorithm. To conclude this chapter, we discuss the limitations of our implementation. The implementation is publicly available on *GitHub*[1].

4.1 Architecture

Now, we present various aspects of the architecture of our implementation. First, we describe our domain and how it is structured. The domain is the core of the application and provides problem specfic classes as the representaton for a DecPOMDP or FSC. Next, we show how we instantiate DecPOMDPs from a definition file. Then, we give some details on input and output of our application which is based on a command line interface and finally we give a broad overview on our testing and documentation approach.

Ground Domain Therefore, we first define our domain for ground DecPOMDPs without any policy representation. Figure 4.1 depicts this ground domain. It can be seen that the used terms are similar to those employed in the original source. We improve the readability and ease the understanding of the implementation by using a single term for the same concept in the algorithm and the implementation. Additionally, we improve the readability, by using wrapper classes for strings, to represent simple objects like a state, an action or an observation. These classes improve the developer experience

[1] `https://github.com/jlandsmann/dec-pomdp-solver`

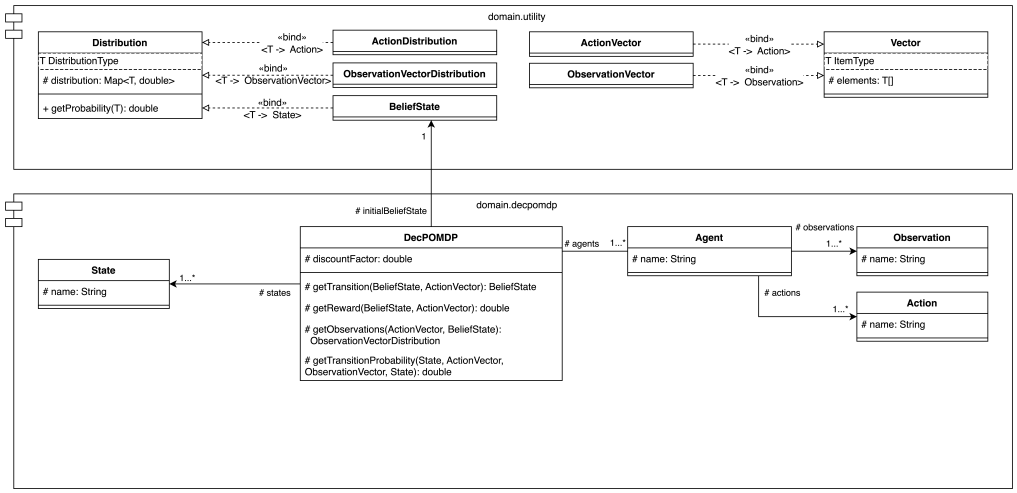


Figure 4.1: Class diagram of the ground domain for the DecPOMDP solver application. For readability only the most important functions and properties are depicted.

as well, because one cannot accidentally pass a state string, where the application expects an action or an observation string. Furthermore, we use interfaces to split the concrete implementation from the design which are not depicted explicitly in the given class diagrams for readability. We use nested `HashMap`s to represent the transition-, observation- and reward function. These provide fast and reliable operations, which is important, as we query those thousands of times during the heuristic algorithm. The `Vector` class wraps a normal list, but helps us to provide some special operations on vectors, which are also not depicted in the class diagram to improve readability. Furthermore, the wording helps the reader of the code to understand that such a list contains elements for each agent. Similarly, the `Distribution` class represents a probability distribution over some elements. One could use a simple map but such could not ensure constraints as the sum of probabilities must add up to one. Since the properties of probability distributions and convex combinations are quite similar, we use that class for both appliances, to prevent two classes with same properties and behavior, but different names.

Finite State Controller Next, we present the domain for FSCs, which consists of an agent and a DecPOMDP to represent their policy. Figure 4.2 shows the class diagram for the FSC domain. As we can see, the DecPOMDP stores values for a pair of state and node vector. This property comes in handy, especially when using infinite-horizon algorithms where we need to solve a system of linear equations to calculate the values. In finite-horizon settings, the storing of precalculated values can also improve the performance as one would not need to calculate values on demand. As we do not perform graph algorithms

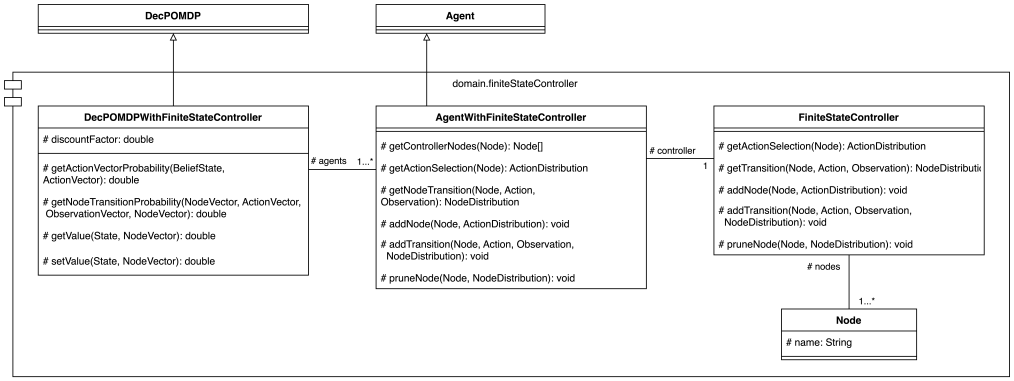


Figure 4.2: Class diagram of the FSC domain for the DecPOMDP solver application, including the controller class itself, an agent using it and a DecPOMDP supporting such an agent.

or similar complex operations, we do not need to use highly optimized graph libraries. Since we are only interested in stochastic transitions from node to node and adding new nodes, a simple functional representation turns out to be adequate. Therefore, we use the efficient `HashMap`s again, for the representation of the transition- and action selection-function, as well as for the value function.

Solving Additionally, we define the `IDecPOMDPSolver` interface for solver classes, which optionally accepts a configuration object, as well as the DecPOMDP. Those are assumed to return the expected value for the initial belief state. One implementation of this solver class is the `HeuristicPolicyIterationSolver`, which we describe in Section 4.3.

Loading DecPOMDPs Oliehoek *et al.* [40] provide a compilation of DecPOMDP models. They describe these with the custom `.dpomdp` text-based file format. This format works with sections, in each of which some aspects of the DecPOMDP are defined. To provide users with a simple way to work with various DecPOMDPs, we offer a parser for this file format that performs syntax and consistency checks. Figure 4.3 shows the class diagram of the parsing classes. We split the responsibilities for better testability and readability [35]. `DPOMDPFileParser` locates the file, reads it from the disk and divides it in its sections. `DPOMDPSectionParser` receives the section and delegates the parsing to one of the section parsers. Each of those parsers takes care of a single section. We use the builder design pattern [17] to be able to create DecPOMDPs in stages as the sections are parsed in serial. `DPOMDPSectionParser` finishes up the parsing process by gathering all data from the section parsers, and building the DecPOMDP and its agents.

Command Line Interface In order to interact with our application, we provide a command line interface, which is build on a set of commands to configure and execute the various algorithms. One example is the `HeuristicPolicyIterationAlgorithmCommand`, which consists of multiple subcommands as initializing, loading a DecPOMDP, configuring the algorithm and starting the solving.

Testing Ensuring quality and correctness is an important aspect of software development. One could only test manually, but as we work on the software and change it many times, one would need to retest the whole software over and over. Thus, in addition to manual testing, we use automatic and programmatic testing for our application. Unit-Testing is a popular technique [26], where single components of a software are tested independently. In this context, the term component refers to a single function or class. The goal of Unit-Testing is to test units as small and meaningful as possible [35]. We use this technique for most of our components, which helps us to build a robust and reliable software.

Documentation Since we build an extendable application and provide it to others in the future, we prepare a user documentation as well as a developer documentation. Users interact with the application through a command line interface. Therefore, we provide some help commands that explain how to use commands and how to configure the behavior of the application. For the developer documentation we use Java's Javadoc, which is used to generate a navigable documentation from comments in the source code. Although, comments can be misleading and can become outdated [35], documentation of interfaces is important, as such interfaces often have implications and constraints.

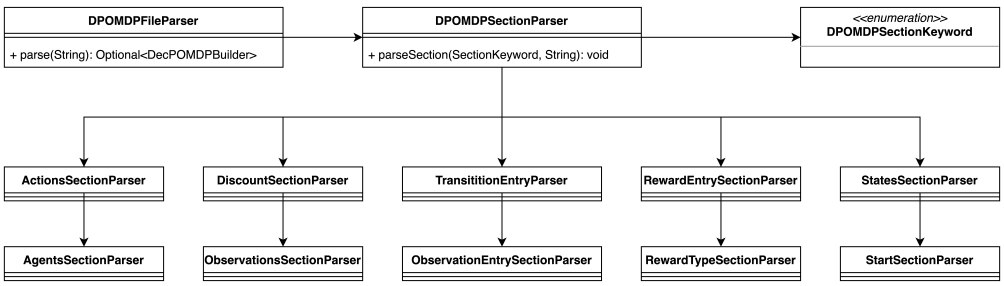


Figure 4.3: Class diagram of the parser for the `.dpomdp` file format.

4.2 Libraries

In software development their is always the strive to find a good balance between using third-party libraries and implementing something by oneself. Third-party libraries have the advantage of solving complex tasks and acting as black boxes for the user. Another big advantage can be that these libraries are often highly optimized and therefore much faster and more efficient than own implementations. However, a high quality or the correctness is not necessarily ensured. In addition, aspects as maintenance and future-proofness are strongly dependent on the publisher of the library. But for some tasks, an individual implementation requires to much effort or a large amount of maintenance work, which paves the way for the use of third-party libraries.

In our case, we identified three tasks, for which we use third-party libraries, to ensure software quality and reduce development expenses. These three tasks are describes in the remainder of this section

4.2.1 Framework

We use a framework for our implementation which helps us with parsing commands and provides dependency injection as well as other development infrastructure. Since we use Java as programming language, a very popular choice is Spring Boot [24, 25, 26]. It is primarily used as a web framework but also provides utilities to create command line interfaces. These capabilities enable us to create a simple program with a prompt first and allow us to add programmatic interfaces later e.g. a REST-API. Since solving DecPOMDPs is very resource-intensive, such an extension could ease the process of solving DecPOMDPs on a server or a cluster.

4.2.2 Solving Systems of Linear Equations

Another tasks is solving a system of linear equations, which is recurring due to its importance to the algorithm. Such tasks are not hard to solve on its own but very hard to do so efficiently. As we solve systems of equations with thousands of equations and unknowns, we need an efficient and reliable library to do the job.

We test two different libraries that performed well in the Java Matrix benchmark. [2]: *EJML* [1] and *ojAlgo* [48]. Both of them are pure Java libraries and do not have external dependencies. *EJML* provides as its competitor different matrix decomposition techniques, that we able to solve equation system with up to 14,000 variables and equations. However, *ojAlgo* performed best due to its multi-core support, which reduced the required time by factor seven compared to single-core performance. Fruthermore, *ojAlgo* especially convinces for large equation systems, for which it needs less than half of the time as it competitor. Details on our tests and the results can be found in Appendix A.

Among its stability and performance, *ojAlgo* convinces with its broad application areas. Furthermore, it supports the usage of commercial solvers as *CPLEX* [48], without the need of adjusting the applications interfaces or additional mappings. Hence, we use *ojAlgo* for solving linear equation systems.

4.2.3 Solving Linear Programs

The third task we use a library for is solving linear programs. In order to find dominating nodes, we need to solve linear programs which consist of many constraints as these are proportional to the number of node combinations. Therefore, we need to find a good and efficient way to solve those linear programs. Since there are several libraries that can solve linear programs, we again look for those that can handle large amounts of constraints efficiently and have few dependencies to prevent them from bloating our application.

ojAlgo provides not only a solver for linear equation systems, but also for linear programs. Thus, we test the linear program solver of *ojAlgo* and compare it to the solver from Google's *OR-Tools* [20] and the solver of the *Apache Math Commons* library [5]. Details on our tests and the results can be found in Appendix B. We use *ojAlgo*, because it performs best in most cases, provides an easy-to-use programming interface and has no dependencies. In addition to that it supports plugin to use native solvers, like *OR-Tools* [48].

4.3 Algorithm

Next, we give some details on the implementation of the HPI algorithm, which we present in Chapter 3. We use the `IDecPOMDPSolver` interface and create an implementing subclass `HeuristicPolicyIterationSolver`. This class accepts a configuration and a DecPOMDP and returns the value by performing the algorithm on the given DecPOMDP. Every step of the algorithm is split into separate classes, as it is proposed by the single responsibility principle [35]. In the remainder of this section, we discuss the implementation of the algorithm. We primarily focus on certain aspects of the algorithm that were challenging when we put theory into practice.

4.3.1 Generating Belief Points

Bernstein *et al.* [9] mentioned how to generate belief points iteratively and warned about belief points being to similar giving poor results. Unfortunately, they gave no details what similarity means in this context and how close means too close. Therefore, we use the Chebyshev distance [7] as a metric for similarity. The Chebyshev distance defines the distance of two points as the maximum distance of two elements in one of their dimensions.

Definition 4.3.1 (Chebyshev distance)**.** Formally, we define a distance $D(b, b')$ of two belief points $b, b' \in \Delta S$, as $\max_{s \in S} |b(s) - b'(s)|$. We define two belief points $b, b' \in \Delta S$ as too close, if $D(b, b') < \epsilon$, with ϵ as the threshold for this condition.

We generate belief points iteratively per agent starting with the initial belief state as first belief point. Afterwards, we select a random action and observation and calculate the next belief state. To achieve a diverse set of belief points, we check if the point is too close to an already added point before adding this belief point to the set and adding the new point to the queue of nodes. Then, we repeat this process until the desired number of belief points is acquired. If the desired number of belief points is not reached when the queue for belief points to explore runs empty, we generate new random policies for the other agents and start again with the initial belief state. We limit the number of policy generations, as some problems (e.g. the *DecTiger* problem (cf. Section 6.1.1)) only have a few states and random state transitions, which makes it difficult to achieve a high diversity of belief points.

The original authors mentioned in their paper, the initialization of the algorithm is very important, as it has an huge impact on the pruning process, which impacts the number of performable iterations directly [9]. Hence, we use seeding to initialize all of our randomization for a better comparison and to reproduce results locally. This technique helps debugging, as randomization can raise bugs and errors which only occur for specific cases.

4.3.2 Evaluating the Value of the Controller

To evaluate the value of the controller, we solve the system of linear equations, which is defined through the value function of the DecPOMDP. As we describe in Section 4.2.2, we use a library for matrix multiplications. Thus, we transform the value function into an equation system of the form $A \cdot x = b$. As we use FSCs, our value function is not only defined over S but over $S \times \vec{Q}$, which leads to $A \in R^{|S \times \vec{Q}| \times |S \times \vec{Q}|}$ and $x, b \in R^{|S \times \vec{Q}|}$. For each coefficient of the matrix, we need to consider all action-, observation- and succeeding node combinations, as well as all possible followup states. Java provides a convenient API [44] to handle parallel tasks. The Stream API supports not only parallel execution, but also offers functions to merge the results. Fortunately, the calculation of each coefficient is independent of any other coefficient, which allows intensive parallelization. Furthermore, we optimized the number of multiplications, by extracting multiplications out of sums where possible. All these optimizations do not affect the worst time complexity, but can impact the practical performance as we can skip whole branches, as soon as one factor equals zero.

Another optimization we make is to consider only those action vectors that can be selected from the current node vector and only those subsequent node vectors that are directly reachable from the current node vector. Given the current node, we receive the selectable actions directly from the action selection

function. We could calculate the successor nodes analogously but we would need to iterate over all following actions and every observation to gather all possible nodes. In this case, we would iterate over all the actions and observations again and without achieving any improvements. Therefore, we precalculate the successor nodes, which we achieve by counting the references from one node to another, when adding the transition to the controller. A transition can be stored in two ways in the controller: During the initialization of the controller or later by calling the function `addTransition`. In the first case, we iterate over all nodes, actions and observations to count the references. Since the initial controller sizes are quite small in our case, this iterating does not have a large impact. In the former case, we receive the successor node distribution for a given node, an action and an observation, then remove the previous distribution's nodes, if existent, and finally add the nodes of the new distribution. We need to count the references to each succeeding node, as there could be multiple edges pointing on a single node, but only some of these edges are removed.

Internally we use Java's `HashMap` class to represent the value-function. `HashMap`s have an initial capacity, which describes the number of buckets used. When this capacity is exceeded, the number of buckets is doubled and all elements of the `HashMap` have to be re-hashed [44]. This process is quite expensive if the mapping consists of thousands of entries. During controller evaluation we generate many new `HashMap`s and add thousands of entries to them. So, we manipulate the initial capacity and set it to 200,000 (default is 16). Additionally, we increase the load factor to 1, which decreases the iteration performance, but increases the threshold for re-hashing all elements in the mapping. As a result, the performance of storing the results of the controller evaluation improves to a reasonable extent.

4.3.3 Exhaustive Backup

During the *exhaustive backup*, all local controllers are extended with a decent amount of new nodes. As each controller can be updated independently, this task can be done in parallel. The most computationally demanding aspect of the *exhaustive backup* is the augmentation of the value function with new node vectors. As we explain in Section 3.3, we can calculate the values for new vectors without solving a system of linear equations. The large number of new node vectors that are generated in each iteration leads to the computational complexity. We reduce the complexity by restricting the calculation of values to states that are part of some belief point of some agent, because in the next steps we only consider the values for the belief points that are calculated exclusively from these states. Another reduction can be made: Analogous to the optimizations made when calculating the coefficients, we only consider those action- and successor node vectors that are directly reachable from the current vector. Especially, the limited action vectors improve the performance,

because each newly added node has per definition only a single action to be chosen. This step can be parallelized, as the computations do not depend on each other. Since we add hundreds of thousands of values to the value function, this process benefits from higher thresholds for re-hashing as well.

4.3.4 Finding Dominating Nodes

After exploring the policy space and expanding the local controller, we find dominating nodes. For each local controller, we find those nodes that dominate every other node at some belief point. Therefore, we iterate over all node vectors and take those vectors that maximize the value for some belief point. For each vector, we extract the node that belongs to the currently considered controller. We mark these extracted nodes as *initial nodes* and use these for the combinatorial pruning.

Pruning nodes that do not contribute to any belief point is less straightforward. We cannot remove a node if it has incoming edges, because we would need to redirect the edge to another node. Otherwise, it could happen that for a given node, action and observation, no successor node is defined. But we can identify the initial node's connected components in the state controller. In this context we denote a connected component of a node, as the subgraph with all nodes that are reachable from the starting node through a directed edge. Given this definition, we can prune all other nodes simultaneously, since these nodes are not reachable from any initial node and hence potential incoming edges must come from nodes that are also to be pruned. Otherwise, the node would be reachable from any initial node and therefore would not be pruned.

4.3.5 Combinatorial Pruning

As we describe in Section 3.5, we check if any of the remaining nodes is dominated by (a convex combination of) other nodes. To find this convex combination, we solve a linear program for each initial node for each controller. As long as we ensure concurrent modifications to be executed correctly, we can prune each node in parallel. Replacing a node is achieved by (i) removing any outgoing connections from that node and (ii) replacing incoming edges with weighted edges based on the computed convex combination. This process potentially leads to nodes that are not reachable anymore, which we need to consider further on, since those nodes are potential starting nodes for some belief point in one of the next iterations.

4.3.6 Termination Criterion

The original algorithm terminates if "controller sizes and parameters do not change" [9]. We determine the current controller state by using the value of the initial belief state and comparing it to the value at the beginning of

the iteration. For better control on testing our implementation, we added another termination criterion: A maximum number of iterations, which can be configured when initializing the algorithm.

4.4 Limitations

Given the known computational costs associated with DecPOMDPs and their evaluation, our implementation is necessarily constrained. In this section we examine the difficulties encountered by our implementation.

Handling Large DecPOMDPs Solving DecPOMDPs is quite hard, as it scales exponentially with the number of agents. This is not only a problem of our implementation, but a problem of solving DecPOMDPs in general [11]. Due to this limitation, we limit our implementation to only support models that can be solved within the boundaries of the memory of the computer it is running on. One could outsource memory-intensive parts of the model, e.g. a complex transition function, into a file or database, but at most algorithms the representational costs of a DecPOMDP are not the limiting factor, as we discuss in Chapter 6.

Handling Large FSCs Since we add a certain amount of controller nodes to each controller in every iteration of the algorithm, the number of joint controller node vectors grows exponentially with the number of agents and the number of iterations. Directed pruning tries to mitigate this circumstance, but the pruning only allows a few more iterations to be performed. Since the number of joint node vectors is crucial for various steps of the algorithm, further investigations need to be made on, how to reduce the number of considered nodes without losing the ability to explore the policy space.

5 Extend for Lifted DecPOMDPs

The HPI algorithm does not scale well with the number of agents and their controller sizes. Calculating the value of the various node combinations is crucial to assess the quality of the current controller and to direct the pruning. Unfortunately, solving the system of linear equations for the value function as well as calculating the value of new node combinations after the *exhaustive backup* are both computationally expensive and time consuming (cf. Section 6.2). Hence, we investigate how lifted DecPOMDPs can be used to improve the performance of the algorithm while taking advantage of their symmetries. Afterwards, we extend the implementation to validate the quality of our extensions. In the remainder of this chapter, we discuss three different approaches and their improvements regarding the complexity of evaluating the value function as well as the size of the joint controller.

5.1 Using Counting DecPOMDPs

Counting DecPOMDP is the less constraining framework for lifted DecPOMDPs. It indeed improves the size of the transition-, reward- and observation function, but as we explain in this section this is not the case for the joint state controller. Although the agents in each partition are indistinguishable, in terms of which agent executes which action, each of them still has their own local policy [13], which results in N different local controller. Thus, the size of the joint state controller remains exponential within N.

Definition 5.1.1 (Counting joint controller)**.** The joint controller for a counting DecPOMDP can be defined as $C_C := \langle \vec{Q}_C, \vec{\Omega}_C, \vec{A}_C, \psi_C, \eta_C \rangle$, where

- $\vec{Q}_C := \times_i^N Q_i$ is a finite set of joint nodes, where Q_i is the set of nodes from C_i.
- $\vec{\Omega}_C := \times_k^K (\times_i^{N_k} \Omega_k)$ is a finite set of joint inputs, where Ω_k is the set of inputs for each agent of partition k.
- $\vec{A}_C := \times_k^K (\times_i^{N_k} A_k)$ is a finite set of joint outputs, where A_k is the set of outputs for each agent of partition k.
- $\psi_C : \vec{Q}_C \to \Delta\vec{A}_C = P(\vec{A}_C | \vec{Q}_C)$ is the joint action selection function.
- $\eta_C : \vec{Q}_C \times \vec{A}_C \times \vec{\Omega}_C \to \Delta\vec{Q}_C = P(\vec{Q}'_C | \vec{Q}_C, \vec{A}_C, \vec{\Omega}_C)$ is the joint transition function.

As described in Definition 2.3.2, we can decompose ψ_C and η_C into multiplications over all agents local controller functions ψ_i and η_i, that lead to a polynomial representational complexity (cf. Equation (5.2)). Unfortunately, this optimization provides no improvements compared to the ground joint controller.

Next, we show that the value function of a counting DecPOMDP with a FSC does not improve either. When calculating the transition- and observation probabilities, we can reduce the vectors to histograms, which simplifies the calculation slightly. But, because each agent has its own controller, the controller transitions have to be calculated individually per agent. This circumstance requires to iterate over all joint node-, action- and observation-combinations for each agent, which can be seen in Equation (5.1). This value function is analogously defined to Equation (2.9), but instead of vectors we query the transition- and observation probability, as well as the immediate reward by using histograms.

$$\begin{aligned} V_C(s,\vec{q}) := & \sum_{\vec{a}\in\vec{A}_C} P(\vec{a}|\vec{q})R_C(s,\#\vec{a})+ \\ \beta & \sum_{\vec{a}\in\vec{A}_C,\vec{o}\in\vec{\Omega}_C,s'\in S,\vec{q}\,'\in\vec{Q}_C} P(\vec{a}|\vec{q})P(\#\vec{o}|\#\vec{a},s')P(s'|s,\#\vec{a})P(\vec{q}\,'|\vec{q},\vec{a},\vec{o})V_C(s',\vec{q}\,') \end{aligned} \tag{5.1}$$

$\#\vec{x}$ defines the histogram over the elements of a vector $\vec{x}$. The required number of floating point operations $\mathbb{E}_C$, to calculate a single coefficient for the system of linear equations, does not change, because we still need to consider every successor state, joint action-, observation- and successor node vector. Since we still have N independent local controller, the calculation of the joint action selection and the joint node transition probability, consists of $\mathcal{O}(N)$ operations. Besides the representational complexity of the joint controller $\mathbb{C}_C$, Equation (5.2) shows the size of the value function $\mathbb{V}_C$ and the number of required floating point operations $\mathbb{E}_C$ to calculate a single coefficient.

$$\begin{gathered} \mathbb{C}_C \in \mathcal{O}(NM^2ao) \qquad \mathbb{V}_C \in \mathcal{O}(sM^N) \\ \mathbb{E}_C \in \mathcal{O}(a^N o^N s M^N N) \end{gathered} \tag{5.2}$$

with $M = \max_{i\in[0;N]} |M_i|$, $a = \max_{k\in[0;K]} |A_k|$ and $o = \max_{k\in[0;K]} |\Omega_k|$.

Calculating all coefficients leads to a complexity of $\mathcal{O}(\mathbb{V}_C^2\mathbb{E}_C)$ and solving the resulting system of linear equations has a complexity of $\mathcal{O}(\mathbb{V}_C^3)$. Hence, evaluating the value function is still exponentially within the number of agents. In the context of FSCs, we can see a counting DecPOMDP does not improve the controller size nor the size of the value function. Therefore, we do not implement the algorithm for counting DecPOMDPs, as the advantages of counting DecPOMDPs do not improve the critical parts of the algorithm.

5.2 Using Isomorphic DecPOMDPs

The more restraining framework of isomorphic DecPOMDPs has two advantages: In contrast to counting DecPOMDPs, isomorphic DecPOMDPs share a policy for the whole partition. Furthermore, we do not need to consider the partition sizes during policy ranking, as Braun *et al.* [14] showed, because the number of agents have no influence on the action selection or the history. Consequently, we generate the joint controller by solving a representative DecPOMDP with K agents. Then, we take the controller and evaluate the value function for the isomorphic DecPOMDP. Thus, during policy ranking the size and evaluation costs are the same as in Equation (2.9) with $N = K$.

Because we are interested in the specific value of a controller and not only its ranking, we need to consider the complexity for policy evaluation as well. Although each partition shares a controller and the same policy and acts the same for the same history, they can still make different observations and thus have different histories that result in different nodes. Due to the indistinguishability of agents in a partition, it does not matter which agent is at which node, but only how many. This indistinguishability enables us to use combinations of histograms for nodes. Because, per definition, each agent selects the same action for the same history, all agents in the same bucket of the histogram select the same action. However, we need to consider each selectable action for every bucket and the combination of those. More complex is the handling of observations, as each agent can sense different observations. Due to the indistinguishability, we need to consider how many agents make which observation, for each node any agent is currently at. Given these declarations, we now define the isomorphic joint controller.

Definition 5.2.1 (Isomorphic joint controller)**.** The joint controller for an isomorphic DecPOMDP is defined as $C_I := \langle \vec{Q}_I, \vec{\Omega}_I, \vec{A}_I, \psi_I, \eta_I \rangle$, where

- $\vec{Q}_I := \times_{k=1}^{K} \#(Q_k, N_k)$ is a finite set of joint node vectors, where Q_k is the set of nodes from the local controller C_k of partition k.
- $\vec{\Omega}_I := \times_{k=1}^{K}(\times_{m=1}^{|Q_k|} \#(\Omega_k, n_m))$ is a finite set of joint input vectors, where Ω_k is the set of inputs for each agent of partition k and n_m denotes how many agents are currently in node q_m with $\forall_{k\in[1;K]}(\sum_{m=1}^{|Q_k|} n_m) = N_k$.
- $\vec{A}_I := \times_{k=1}^{K}(\times_{m=1}^{|Q_k|} A_k)$ is a finite set of joint output vectors, where A_k is the set of outputs for each agent of partition k.
- $\psi_I : \vec{Q}_I \to \Delta\vec{A}_I = P(\vec{A}_I|\vec{Q}_I)$ is the joint action selection function.
- $\eta_I : \vec{Q}_I \times \vec{A}_I \times \vec{\Omega}_I \to \Delta\vec{Q}_I = P(\vec{Q}'_I|\vec{Q}_I, \vec{A}_I, \vec{\Omega}_I)$ is the joint transition function.

Since every partition has its own controller, which is shared across all agents in the partition, the size $\mathbb{C}_I$ of the joint controller C_I is, as described in Equation (5.6), polynomial within K. The action selection- and transition function can be calculated based on the local functions, as the decisions of each agent are independent of each other. This can be seen in the following equation:

$$P(\vec{a}|\vec{q}) = \prod_{i=1}^{N} P(a_i^d|q_i^d) \qquad P(\vec{q}'|\vec{q},\vec{a},\vec{o}) = \prod_{i=1}^{N} P(q_i'^d|q_i^d,a_i^d,o_i^d) \tag{5.3}$$

Altough, we share a controller across a partition, the calculation of the joint action selection and the joint node transition consists of N multiplications each. The value function of an isomorphic DecPOMDP can be defined as:

$$\begin{aligned} V_I(s,\vec{q}) := & \sum_{\vec{a}\in\vec{A}_I} P(\vec{a}|\vec{q})R_I(s,\vec{a}) + \\ \beta & \sum_{\vec{a}\in\vec{A}_I,\vec{o}\in\vec{\Omega}_I,s'\in S,\vec{q}\,'\in\vec{Q}_I} P(\vec{a}|\vec{q})P(\vec{o}|\vec{a},s')P(s'|s,\vec{a})P(\vec{q}'|\vec{q},\vec{a},\vec{o})V_I(s',\vec{q}') \end{aligned} \tag{5.4}$$

Although we can decompose the transition-, reward- and observation function, this does not apply for the action selection- and node transition-function. Furthermore, symmetries within the transition function do not necessarily lead to symmetries within the value function that would allow a decomposition [31]. Nevertheless, isomorphic DecPOMDPs improve the size $\mathbb{V}_I$ of the value function V_I to be exponentially within the number of partitions instead of the number of agents, as the number of joint node vectors is at most $\prod_{k=1}^{K}(N_k^{|M_k|})$. This formula is derived from Equation (2.14). In the worst case, there is a single agent per node. This would render the optimizations obsolete, because the number of action and observation combinations to consider both still would be exponential within the number of agents. Furthermore, in order to calculate the node transition probability from one histogram of nodes to another, we need to consider the transition in every possible node vector that corresponds to the same histogram. This limitation exists, because we can only calculate the probabilities based on the individual transition probabilities of the local controller.

$$\prod_{k=1}^{K} |\#(Q_k,N_k)| * \prod_{k=1}^{K} (M_k^{N_k}/|\#(Q_k,N_k)|) = \prod_{k=1}^{K} M_k^{N_k} \in \mathcal{O}(M^N) \tag{5.5}$$

As there are $|\#(Q_k,N_k)|$ histograms and $Q_k^{|N_k|}$ permutations for each partition, on average we need to consider $\mathcal{O}(Q_k^{|N_k|}/|\#(Q_k,N_k)|)$ permutations per partition per histogram. As we iterate over all possible histograms, it turns out

Algorithm 5.1 Isomorphic Heuristic Policy Iteration

Require:
Isomorphic DecPOMDP
Desired number of belief points $k \in \mathbb{N}$
Initial belief state $b_0 \in \Delta S$
Local policies $\forall_{k \in K} : \pi_k$

1: Create representative DecPOMDP from isomorphic DecPOMDP.
2: Call heuristic policy iteration with representative DecPOMDP.
3: Extract local controllers to isomorphic DecPOMDP.
4: Evaluate the value function V_I by solving the linear equation system.
5: Return maximum expected reward for initial belief state.

the same as iterating over all observation permutations in matters of computational complexity. If we combine all these aspects and the fact that we need to consider all action-, observation and successor node vectors, the required number of floating point operations $\mathbb{E}_I$ to calculate a single coefficient still is in the worst case exponential within the number of agents. The following equaion denotes $\mathbb{C}_I$, $\mathbb{V}_I$ the sizes of C_I and V_I as well as the complexity of calculating a single coefficient $\mathbb{E}_I$.

$$\begin{gathered} \mathbb{C}_I \in \mathcal{O}(KM^2ao) \qquad \mathbb{V}_I \in \mathcal{O}(sM_{\#}^K) \\ \mathbb{E}_I \in \mathcal{O}(a^N o^N s M^N L N) \end{gathered} \tag{5.6}$$

with $M = \max_{k \in [0;K]} M_k$, $M_{\#} = \max_{k \in [0;K]} |\#(Q_k, N_k)|$, $a = \max_{k \in [0;K]} A_k$ and $o = \max_{k \in [0;K]} \Omega_k$. To calculate the joint reward, the transition- and observation probabilities, we need to consider each of L groundings [13]. As before, in the worst case we must perform $\mathcal{O}(N)$ multiplications per node vector to calculate the joint node transition- and action selection probabilities. Nonetheless, in the context of FSCs an isomorphic DecPOMDP does indeed reduce the size of the joint controller, and furthermore can reduce the size of the value function as well as the complexity of solving it. Unfortunately, latter still remains exponentially within the number of agents.

5.2.1 Algorithmic Adjustments

Even though the evaluation of the joint controller is still exponential within the number of agents, we can perform policy ranking with a representative DecPOMDP with only K agents. Thus, we show, how isomorphic DecPOMDPs can be used to improve the HPI algorithm. Since only the final controller has to be evaluated for the isomorphic DecPOMDP, we can run the original HPI algorithm within K agents.

Algorithm 5.1 shows our extended algorithm. We take the same parameters as the ground version of the HPI algorithm, but require the DecPOMDP

to be isomorphic. Then, we create a representative DecPOMDP from the isomorphic one by copying the DecPOMDP and setting each partition size $\forall_{k\in[1;K]} : N_k = 1$. After the transformation, we let the ground algorithm solve the representative DecPOMDP. As soon as the controller does not change anymore, we adopt it as the controller for the isomorphic DecPOMDP. Now, we can evaluate the value of the controller for the original isomorphic DecPOMDP. To achieve this, we need to solve the system of linear equations of the value function (cf. Equation (5.4)) for all combinations of states and joint node vectors. Since the observations of the agents may differ, we need to take all possible histograms for each partition into account and all combinations of those as joint node vectors. After the value function is solved, we can find the best joint node vector to start at and return the expected value.

5.2.2 Implementation

The implementation of this algorithm requires some adjustments to the original implementation: First of all, we adjust our parser to allow parsing lifted DecPOMDPs as well as classes that represent those lifted and especially isomorphic DecPOMDPs. Then we implement a solver (cf. Section 4.1) that performs the adjusted algorithm itself, and finally we adjust the value function evaluation.

Parsing First, we extend the `.dpomdp` file format with an additional section: the partition size section. It is structured analogously to the action and observation section, where each row refers to one agent. Every row consists of exactly one positive integer. Each integer defines how many agents of this type are existent in the model. The section is mandatory and has to follow directly after the observation section. As we do not adjust the number of agents, the action and observation definitions can be the same as for a ground DecPOMDP. Furthermore, the transition-, reward- and observation function definitions do not need to be updated as they are expected to be isomorphic functions that accept one argument for every partition. As lifted DecPOMDPs, especially isomorphic DecPOMDPs, are more general than ground DecPOMDPs, we create the `LiftedDecPOMDP` and `IsomorphicDecPOMDP` base classes and the `IsomorphicDecPOMDPwithStateController` for the use of isomorphic DecPOMDPs with FSCs as policy representation.

Solver Next, we write a solver class that accepts an isomorphic DecPOMDP, transforms it into a representative DecPOMDPs and calls the original solver with the transformed DecPOMDP. When the original solver finishes solving, the solver extracts the resulting local controllers and transfers these back to the original isomorphic DecPOMDP. Since the original algorithm executes until it runs out of memory or time or the controller stagnates, we define a maximum number of iterations that the original algorithm is allowed to complete.

Thereafter, the solver starts evaluating the value function of the isomorphic DecPOMDPs. Eventually, it returns the calculated value for the initial belief state. To enable users to use this new solver, we add a corresponding command line interface.

Value Function Evaluation Lastly, we adjust the value function evaluation. As we previously state, we only consider node histograms instead of permutations. Therefore, we need to find an efficient way to calculate the node transition probability from one histogram to another. One way to do so is to iterate over all vectors that result in the target histogram as explained before. This process again can be computed in parallel and the resulting coefficients can be summed up. At this point the optimizations regarding the action vectors and joint followup node vectors come in handy, since they can reduce the runtime for situations where there are many agents in the same node.

5.3 Using Representative Observations

Since solving counting and isomorphic DecPOMDPs is in the context of FSCs still exponential within the number of agents, we need to find another technique to be able to reduce the size of the value function and the complexity of evaluating it. One approach to do so is to assume inter-partition independence. As we already assume intra-partition independence, the model would fall apart into K POMDPs [14, 15]. This way it would be quite simple to solve the model, but the model would be far less expressive as it could no longer model interactions and cooperations between agents, which is one of the ground pillars of DecPOMDPs.

Thus, we make another assumption about the isomorphism of the agents in every partition. Until now, we state that it does not matter which agent of a partition executes which action and that for the same history each agent of the partition performs the same action choice. However, as the observations sensed by each agent are independent of each other, we must consider every combination of observations when evaluating the value function. This results in the value function's size being exponential in terms of the number of agents involved. If we assume each agent of a partition to sense the same observation, we would no longer need to consider all combinations of observations for N agents but for K partitions.

As Braun *et al.* [14] showed, is it sufficient to calculate the best policy using only a single representative per partition. Furthermore, since all agents of a partition make the same observations and action choices, and because they are indistinguishable and isomorph, they begin at the same node and transfer each round to the same node. When we assume representative observations as well as same action selection and transition, we can reduce the value function drastically. In this case, every agent of a partition is always in the same node,

selects the same action, makes the same observation and transitions into the same node. Thus, we only need to consider peak-shaped node-histograms for each partition, where each agent of a partition refers to the same element. From this finding, we can deduce that every grounding results in the same element selection, which we can exploit to simplify the calculation of the value function parameters. Given the definition of the decomposed transition-, reward- and observation function (cf. Equation (2.15)) and the fact that all grounding result in same selection, the following equations hold with $d \in D$ being an arbitrary grounding:

$$\begin{aligned} R_I(s, \vec{a}) = L \cdot \bar{R}_I(s, \vec{a}^{\,d}) \qquad P(\vec{o}|\vec{a}, s') = \bar{P}(\vec{o}^{\,d}|\vec{a}^{\,d}, s')^L \\ P(s'|s, \vec{a}) = \bar{P}(s'|s, \vec{a}^{\,d})^L \end{aligned} \tag{5.7}$$

In addition, we can simplify the calculation of the action selection and the node transition function, in comparison to isomorphic DecPOMDPs (cf. Equation (5.3)) as defined in the following equation:

$$P(\vec{a}|\vec{q}) = \prod_{k=1}^{K} P(a_k^d|q_k^d)^{N_k} \qquad P(\vec{q}\,'|\vec{q}, \vec{a}, \vec{o}) = \prod_{k=1}^{K} P(q_k'^d|q_k^d, a_k^d, o_k^d)^{N_k} \tag{5.8}$$

With these simplifications given, we define the value function for an isomorphic DecPOMDP with representative observations:

$$\begin{aligned} V_R(s, \vec{q}) := & \sum_{\vec{a} \in \vec{A}_R} L \cdot P(\vec{a}|\vec{q}) \bar{R}_I(s, \vec{a}) + \\ \beta & \sum_{\vec{a} \in \vec{A}_R, \vec{o} \in \vec{\Omega}_R, s' \in S, \vec{q}\,' \in \vec{Q}_R} P(\vec{a}|\vec{q}) \bar{P}(\vec{o}|\vec{a}, s')^L \bar{P}(s'|s, \vec{a})^L P(\vec{q}\,'|\vec{q}, \vec{a}, \vec{o}) V_R(s', \vec{q}\,') \end{aligned} \tag{5.9}$$

where $\vec{A}_R = \times_{k=1}^{K} A_k$, $\vec{\Omega}_R = \times_{k=1}^{K} \Omega_k$ and $\vec{Q}_R = \times_{k=1}^{K} Q_k$. This equation leads to the following complexities for size and evaluation:

$$\begin{aligned} \mathbb{C}_R \in \mathcal{O}(KM^2 a_I o_I) \qquad \mathbb{V}_R \in \mathcal{O}(sM^K) \\ \mathbb{E}_R \in \mathcal{O}(a_I^K o_I^K sM^K \log(L) K \log(N_{k,\max})) \end{aligned} \tag{5.10}$$

with $N_{k,\max} = \max_{k \in [1..K]} N_k$. The terms $\log(L)$ and $\log(N_{k,\max})$ follow from the complexity of exponentiation [14]. $\mathbb{E}_R$ shows, we can calculate coefficients without being exponentially dependent on the number of agents. Thus, the same applies for the complexity of value function evaluation. As we are now only logarithmically dependent on the number of agents, we expect a massive improvement during policy evaluation.

5.3.1 Algorithmic Adjustments

For solving isomorphic DecPOMDPs with representative observations, we use Algorithm 5.1 further on. To profit from the advantages of representative observations, it is sufficient to use the optimized value function for policy evaluation instead, because the controller ranking happens on a representative DecPOMDP. The way we transform an isomorphic DecPOMDP with representative observations into a representative one and how we transfer the controller back to the original model, remains the same as for isomorphic DecPOMDPs with individual observations.

5.3.2 Implementation

Since we use large parts of the algorithm for isomorphic DecPOMDPs, we need to adjust our implementation just slightly. Because we modify the value function we need to introduce the `RepresentativeObservationsDecPOMDP` and its fellow FSCs extension. These classes have a modified value function that exploits the made assumptions, as we described it in Equation (5.9).

Parsing Since we only make stricter assumptions about the behavior of the agent, we do not adjust the definition of the underlying model. Hence, we do not modify the file format for defining isomorphic DecPOMDPs and use the previously introduced modifications to parsing further on.

Solver Although we basically use the same algorithm, some parts of the algorithm differ. Therefore, we introduce the `RepresentativeObservations-HeuristicPolicyIterationSolver`, which implements Algorithm 5.1 for representative observations. Again, to enable the user to interact with this solver, we introduce a third set of commands for initializing the algorithm, loading a DecPOMDP, configuring initial policies and solving. All commands are structured similarly to ease the understanding and to improve the usability. We do not enforce these similarities, if future extensions require other steps or commands.

Value Function Evaluation As we only consider peak-shaped node histograms per partition, we no longer need to calculate histogram to histogram node transitions, where we iterate over all possible node vectors. Hence, we go back to the normal coefficient evaluation. The various probabilities are calculated by the DecPOMDP class itself and therefore require no modifications to be made on the side of the equation system solver.

6 Experimental Results

In this chapter, we present the problem instances that we use for testing purposes. Subsequently, we discuss the results of our implementation of the HPI algorithm comparing them with the original implementation. Furthermore, we conduct tests on lifted DecPOMDPs. To validate our results for lifted DecPOMDPs, we compare them with the results of our implementation of the non-optimized HPI algorithm.

Due to the complexity of the investigated problems, we test the algorithm on more advanced hardware. This hardware enables us to perform multiple iterations on larger problem instances. Therefore, we perform our tests on the PALMA II cluster, a high performance cluster, which is operated and maintained by the University of Münster. For all of the tests, we use a single node with 36 CPU cores and 92GB of memory. Due to multi-threading, we are able to run 72 threads in parallel. This opportunity is particularly useful when calculating the value function and solving the system of linear equations. Each solving has a time limit of 24 hours.

6.1 Test Domains

In order to test our implementation and to be able to validate our implementation's results for reasonability against the original algorithm, we use the same problems for testing as Bernstein *et al.* [9]. Additionally, we introduce an instance of a *Medical Nanoscale System*, which was outlined by Braun *et al.* [13].

6.1.1 DecTiger

The *DecTiger* problem, also called the *Two Agent Tiger* problem, was introduced by Nair *et al.* [39]. We use the version from the MADP Toolbox [40]. It consists of two states and two identical agents with three actions and two observations. In the setting of this problem there are two doors. Behind one of which is a tiger, whereas behind the other door is a treasure. Goal of both agents is to jointly open the door with the treasure behind it to earn a reward and to avoid the large penalty of -100 points, which is given if they open the door with the tiger behind it. If they both choose to open the same door, the penalty is reduced to -50 points. Listening costs 1 point for each agent and opening the correct door earns them 10 points each. These costs only apply if

none of them opens the door with the tiger at the same time. The agents have three options to act: open the left door, open the right one, and listen. When they listen, they observe a noise behind one of the doors. With a probability of 85%, they hear the noise behind the correct door. If one of them opens a door, the observations are uniformly distributed. In this case the following state is also uniformly distributed, otherwise it remains unchanged. In the beginning, the tiger is behind one of the doors with equal probability. To fit the description of Bernstein *et al.* [9], we rearrange the order of actions for both agents.

6.1.2 Meeting on a Grid

The *Meeting on a Grid* problem, also called the *GridSmall* problem, was introduced by Bernstein *et al.* [10]. Again, we use the version from the MADP Toolbox [40], which is based on the version from Amato *et al.* [3]. It consists of a 2×2 grid without any obstacles but with two agents on it. This grid results in 16 different positions for the agents and therefore in 16 different states. The goal of both agents is to share a tile to earn a reward of 1 point. In all other states, the reward is 0 points. Both agents can move in all four directions or stay in their current tile. As in the *DecTiger* problem, transitions are noisy. The agents only move with a probability of 60% in the intended direction. Otherwise, they either stay in the same square or move in another direction with a 10% probability each. Moving against a wall results in staying in the same square. Both agents do not interfere or sense each other, but they can observe if there is either a wall on the left or the right. In the beginning, both agents are in the upper left corner of the grid.

6.1.3 Box Pushing

The third problem, the *Box Pushing* problem, was introduced by Seuken *et al.* [50]. It consists of 100 states and two agents with four actions and five observations each. It represents a 4×3 grid with one large and two small boxes in the center row, which should be pushed to the top row. The small boxes can be moved by a single agent, but the large box has to be pushed by both agents at the same time in the same direction. Both agents start at the corners of the bottom row, facing each other, their actions are move, turn left, turn right, and staying in place. The movement succeeds with a probability of 90%, but in contrast to the *Meeting on a Grid* problem, the two agents are not able to share a tile. The agents sense the tile in front of them, and can observe either an empty space, a wall, the other agent, a small box or a large one. As soon as a box is pushed to the top row, the environment resets. If both agents push the large box to the top row, a reward of 100 points is earned, for a small box the reward is 10 points. A penalty of -5 points is given, if an agent cannot move and each time step costs 0.1 points per agent.

6.1.4 Medical Nanoscale System

At last, we introduce a problem that exhibits isomorphic symmetries: the *Medical Nanoscale System* proposed by Braun *et al.* [13]. It consists of two types of agents: sensors and bots. Sensors can sense if a sensor-specific biochemical marker is present or not and can output a sensor-specific message. Bots can observe if a specific (composed) message is present or not and can output medicine. Composed messages are assumed to be assembled automatically. The aim of this system is to ensure that the bots only release medicine when certain markers are present. The number of states depends on the number of markers and messages to consider. In our case, we focus on a system with a single type of sensor and a single type of bot, which leads to one marker and a single message to be present and $2^2 = 4$ states in total. Each agent has two actions and observations: Sensors output a message or do nothing, while bots output medicine or do nothing instead. Latter can make two observations: message present or not present, whereas sensors can sense if their specific marker is present or not. If no medicine is present, the presence of each marker remains unchanged with a probability of 90%. If the corresponding medicine is present, the probability for absorbing the present marker is at 99%, while the probability for a marker being present afterwards is at 1%. Messages cannot occur randomly, but only if a tile is outputted by an agent. If an agent drops a tile, the probability of success is at 90%. If a message or tile is already present, it decays with a probability of 10% and if an agent drops a tile additionally, there is a message present afterwards with a probability of 99%. False positive observations occur with a probability of 1% and false negative observation occur with a probability of 20%. Outputting a tile (medicine) is rewarded by 20 points if a marker (message) is present and by -10 points otherwise. Not outputting a tile (medicine) is not rewarded or fined, if no marker (message) is present, and fined with a penalty of 5 points otherwise. In real-world situations, the number of agents per partition can rise up to thousands of agents [14]. Since this number is crucial for value evaluation, we vary this number based on the type of model we use.

6.2 Heuristic Policy Iteration

As we work with the same testing domains as Bernstein *et al.* [9], we compare our implementation's results with those of the original algorithm. Unfortunately, they give no details on the controller sizes during the execution of their HPI algorithm, but only on the value achieved after each iteration. Thus, we can only compare the number of iterations possible and the value achieved for each problem.

Bernstein *et al.* [9] used initial policies for the belief point generation. In the case of the *DecTiger* problem, each agent chooses in every state to listen with a probability of 80%, and to open the left or right door with 10% each.

For the other two problems they use uniformly distributed action selections for all states. At the beginning of the algorithm, they start with a single node controller for each agent, which selects the first defined action of the agent and then transitions into the sole node for all observations. Bernstein *et al.* [9] stated that for the *DecTiger* problem the first action is to open the left door, while for the *Meeting on a Grid* problem that action is to move up and for the *BoxPushing* problem to turn left. The original models use a discount factor of $\beta = 1$ [40], which is not applicable for infinite horizon solving, because the sum for the value function would be infinite as well. Thus, we use a discount factor of $\beta = 0.9$ for all problems, as Bernstein *et al.* [9] did.

6.2.1 Diversity

Before we have a look at our results, we discuss the diversity of belief points. Since Bernstein *et al.* [9] mentioned the importance of diversity within the set of belief points, we briefly present what we observe while varying the degree of diversity. We need to find a good balance between over-fitting the initial belief state and considering uncommon parts of the state space. Because the number of nodes limits the algorithm to a few iterations and belief points influence finding convex combinations to prune nodes, a good initialization of belief points can increase the number of iterations drastically. Therefore, it can increase the value of the controller. As the threshold for minimum allowed distance between any pair of points in the belief state space increases, the maximum number of points that do fit in this space, maintaining the given minium distance, decreases. Due to the various different structures of the state spaces of the problems, the impact of threshold differs on every problem instance. However, we stick with $\epsilon = 2 \times 10^{-8}$ because we achieve good results with this setting across various domains.

6.2.2 DecTiger

Now, we execute our algorithm on the *DecTiger* problem and present our results in the following. Similar to the original authors, we use ten belief points per agent, a discount factor of $\beta = 0.9$ and the initial polices mentioned above.

Table 6.1 depicts our implementations results. As one can see, we are able to complete 15 iterations within 24 hours. The controller sizes starting at one, increase every iteration up to 34 and 32 nodes. At the last iteration we reach a value of 19.9998 points, which relates to both agents choosing to listen again and again which leads to costs of 2 points per round. With a discount factor of $\beta = 0.9$, the rewards sum up to -20 points as the expected reward for the initial state. The same value was achieved by Bernstein *et al.* [9] when using the HPI algorithm. Further inspections of the controller show that both agents choose to listen in all situations. Figure 6.1a depicts our results in relation to

DecTiger					
It.	**Start**	**exh. Backup**	**Pruning**	**comb. Pr.**	**Value**
0	(1, 1)	-	-	(1, 1)	-150.0000
1	(1, 1)	(4, 4)	(3, 3)	(3, 3)	-137.0000
2	(3, 3)	(30, 30)	(6, 5)	(5, 5)	-117.8525
3	(5, 5)	(80, 80)	(7, 8)	(7, 7)	-98.8986
4	(7, 7)	(154, 154)	(9, 9)	(9, 9)	-91.0087
5	(9, 9)	(252, 252)	(14, 14)	(14, 14)	-83.9078
6	(14, 14)	(602, 602)	(16, 16)	(16, 15)	-68.4525
7	(16, 15)	(784, 690)	(19, 17)	(19, 17)	-62.6728
8	(19, 17)	(1,102, 884)	(24, 22)	(24, 22)	-58.1344
9	(24, 22)	(1,752, 1,474)	(28, 25)	(27, 24)	-46.5644
10	(27, 24)	(2,214, 1,752)	(28, 26)	(28, 26)	-42.7463
11	(28, 26)	(2,380, 2,054)	(33, 29)	(33, 29)	-40.4716
12	(33, 29)	(3,300, 2,552)	(38, 32)	(38, 31)	-33.2794
13	(38, 31)	(4,370, 2,914)	(38, 33)	(38, 33)	-28.4219
14	(38, 33)	(4,370, 3,300)	(36, 34)	(36, 34)	-27.5797
15	(36, 34)	(3,924, 3,502)	(38, 34)	(34, 32)	-19.9998

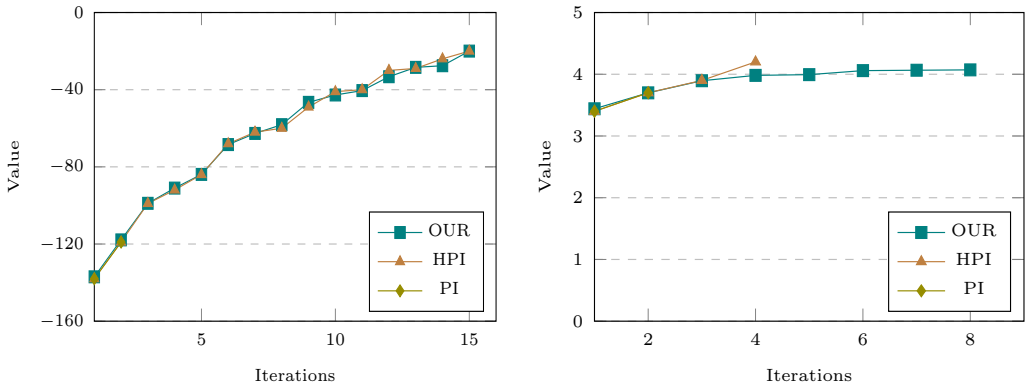
Table 6.1: Experimental results of *DecTiger* problem. The first column shows the number of iteration, the second shows the controller sizes at the beginning of the iteration, and the third column shows the controller sizes after the exhaustive backup. Column four shows the controller sizes after retaining all dominating nodes and their successors, and column five shows the the same size after the combinatorial pruning step. The last column shows the value at the end of the iteration.

the author's results. As one perceives, the results are close to each other for all iterations. We also add the author's results of the optimal policy iteration to underline the advantages of the directed pruning. One can see the impact of heuristic pruning as the optimal algorithm is only able to complete three iterations.

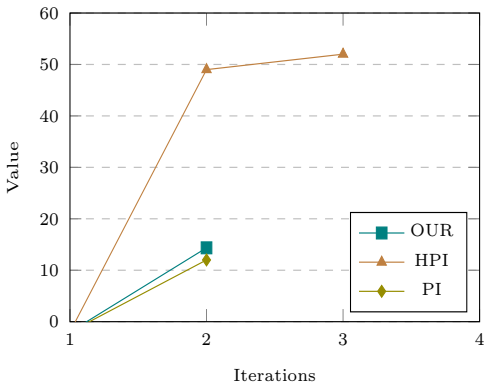
6.2.3 Meeting on a Grid

The second domain we test our implementation on is the *Meeting on a Grid* problem. As proposed by Bernstein *et al.* [9], we use ten belief points and uniform action selection as the initial policy for each agent, as well as a discount factor of $\beta = 0.9$.

Table 6.2 shows our experimental results on the *Meeting on a Grid* problem. We are able to complete eight iterations before the number of possible node combinations become too large. We achieve a value of 4.0711 points. Due to the larger state space which consist of 16 states in this case ($|S| = 2$ for the *DecTiger* problem), the number of iterations decreases significantly to



(a) DecTiger **(b)** Meeting on a Grid



(c) BoxPushing

Figure 6.1: Comparison of our implementation (OUR) and the originals results for the heuristic (HPI) and optimal (PI) policy iteration algorithm. The horizontal axis denotes the number of iterations, whereas the vertical axis refers to the value achieved at the end of each iteration.

four. Figure 6.1b shows our results in direct comparison with the authors results, which achieve a slightly higher value of approximate 4.2 points after four iterations. We attribute these differences to the heuristic initialization of the algorithm, which affects the belief points, which affect the pruning and value optimization. Additionally, Table 6.2 shows the importance of pruning, as the controller sizes after the exhaustive backup increase massively, when starting at only 24 nodes.

6.2.4 BoxPushing

The last domain the original algorithm is tested on is the *BoxPushing* problem. As the authors of the original algorithm propose, we use twenty belief points, an uniform action selection as the initial policy and a discount factor of $\beta = 0.9$.

Meeting on a Grid					
It.	**Start**	**exh. Backup**	**Pruning**	**comb. Pr.**	**Value**
0	(1; 1)	-	-	(1; 1)	2.8008
1	(1; 1)	(6; 6)	(4; 2)	(4; 2)	3.4407
2	(4; 2)	(84; 22)	(8; 5)	(7; 5)	3.6989
3	(7; 5)	(252; 130)	(10; 9)	(10; 8)	3.8951
4	(10; 8)	(510; 328)	(16; 11)	(14; 10)	3.9820
5	(14; 10)	(994; 510)	(18; 14)	(18; 12)	3.9938
6	(18; 12)	(1,638; 732)	(22; 16)	(21; 14)	4.0589
7	(21; 14)	(2,226; 994)	(27; 19)	(24; 16)	4.0652
8	(24; 16)	(2,904; 1,296)	(25; 20)	(22; 18)	4.0711

Table 6.2: Experimental results of *Meeting on a Grid* problem. The first column shows the number of iteration. As before, the columns two to five, describe the number of nodes after the corresponding step. The last column shows the value at the end of the iteration.

Table 6.3 shows the results of our experiments on the third problem. Because of the large state space and the amounts of actions and observations per agent, we only are able to complete two iterations of the algorithm and achieve a value of 14.3562 points. In the next iteration, which does not complete, the controllers consist of 31,110 and 4,100 nodes. Since we would need to calculate the value for all states of the belief points and each new combination of nodes, we would need to compute the value for around ten billion entries of the value function and iterate over all of these to find the dominating ones. Unfortunately, we are not able to verify the results presented by the authors of the original heuristic algorithm, as Figure 6.1c illustrates. They were able to perform three iterations and achieved a value of approximate 52 points. Again, we assume that such differences come from a deviating initialization, which allows more nodes to be pruned. Nevertheless, the results for the other domains validate the quality of the algorithm and confirm that using FSCs does work for solving DecPOMDPs.

Box Pushing					
It.	**Start**	**exh. Backup**	**Pruning**	**comb. Pr.**	**Value**
0	(1; 1)	-	-	(1; 1)	-1.9999
1	(1; 1)	(5; 5)	(3; 3)	(2; 2)	-2.0000
2	(2; 2)	(130; 130)	(7; 7)	(6; 4)	14.3562

Table 6.3: Experimental results of *BoxPushing* problem. The first column shows the number of iteration. As before, the columns two to five, describe the number of nodes after the corresponding step. The last column shows the value at the end of the iteration.

6.2.5 Medical Nanoscale System

Finally, we present our results on the *Medical Nanoscale System* problem. We execute the algorithm with partition sizes of one to show what the HPI algorithm is capable of. To emphasize the improvements on performance made by isomorphic DecPOMDPs, we execute the original algorithm on a ground DecPOMDP with multiple agents. We calculate that ground DecPOMDP from the isomorphic definition by multiplying the transition- and observation probabilities over all groundings for all possible action combinations. Same applies to rewards, which are summed up. For each agent in each partition, we create a ground agent with an own local controller. We underline that this ground version is not equivalent to the isomorphic one, because we do not ensure same action selection for same histories within partitions. Therefore, the values are not comparable, instead we depict the scalability of ground DecPOMDPs.

As the original algorithm's execution time scales rapidly with the number of agents, we start with two agents in each partition, which leads to a ground DecPOMDP with four agents in total. This way, we can check the scalability of ground FSCs and we get a basis for comparison for lifted DecPOMDPs as well. Since the number of iterations possible decreases rapidly when increasing the partition sizes, we stop after a partition size of five agents.

Table 6.4 shows the results of running the HPI algorithm on the *Medical Nanoscale System* with two agent types. We are able to perform 20 iterations and achieve a value of 115.6185 points. One cannotice the much higher number of iterations in comparison to the other problems, which can be reasoned by the small state space. Unfortunately, the state space growths with the number of different sensors and bots. As we focus on a system with one kind of each, the state space remains quite small. Being able to perform this many iterations helps us for the later experiments, where we perform policy ranking with a representative DecPOMDP of the *Medical Nanoscale System* problem.

Next, we max out the limits of the algorithm by increasing the partition sizes progressively. Figure 6.2 shows the joint controller sizes at the end of every iteration for the DecPOMDPs with the given partition size. The figure depicts the limited scalability of the joint FSC, as the DecPOMDP with a partition size of two agents per partition has a larger joint FSCs than the DecPOMDP with two agents in total after 20 iterations. It is even more noticeable with the larger DecPOMDPs with eight or ten agents. In the case of $N_k = 4$, every agent except one has only two nodes and the eighth agent has a single node in their local controller, but the number of joint node vectors is already above 100. We stop to increase the partition sizes after $N_k = 5$, as the algorithm is not able to complete the second iteration. Again, these results underline the limited scalability of DecPOMDPs with regard to the number of agents, and emphasize why lifting agents and exploiting symmetries is so important for solving large DecPOMDPs.

Grounded Medical Nanoscale System ($N_k = 1$)					
It.	**Start**	**exh. Backup**	**Pruning**	**comb. Pr.**	**Value**
0	(1;1)	-	-	(1;1)	-38.1579
1	(1; 1)	(3; 3)	(2; 2)	(2; 2)	-22.7829
2	(2; 2)	(10; 10)	(2; 5)	(2; 5)	-4.7820
3	(2; 5)	(10; 55)	(3; 8)	(3; 8)	12.5837
4	(3; 8)	(21; 136)	(5; 10)	(5; 10)	27.0469
5	(5; 10)	(55; 210)	(9; 10)	(8; 10)	40.0804
6	(8; 10)	(136; 210)	(11; 10)	(10; 10)	50.5603
7	(10; 10)	(210; 210)	(13; 11)	(13; 11)	58.6376
8	(13; 11)	(351; 253)	(15; 14)	(15; 14)	66.1698
9	(15; 14)	(465; 406)	(18; 17)	(18; 17)	73.6488
10	(18; 17)	(666; 595)	(21; 16)	(21; 16)	79.4078
11	(21; 16)	(903; 528)	(23; 17)	(23; 17)	84.8559
12	(23; 17)	(1,081; 595)	(26; 20)	(26; 20)	90.4471
13	(26; 20)	(1,378; 820)	(28; 23)	(28; 23)	94.6603
14	(28; 23)	(1,596; 1,081)	(30; 22)	(30; 22)	98.4639
15	(30; 22)	(1,830; 990)	(32; 23)	(32; 23)	102.6934
16	(32; 23)	(2,080; 1,081)	(35; 26)	(35; 26)	105.7797
17	(35; 26)	(2,485; 1,378)	(37; 31)	(37; 31)	108.3850
18	(37; 31)	(2,775; 1,953)	(39; 28)	(39; 28)	111.6223
19	(39; 28)	(3,081; 1,596)	(41; 29)	(40; 29)	113.8836
20	(40; 29)	(3,240; 1,711)	(39; 33)	(39; 31)	115.6185

Table 6.4: Experimental results of grounded *Medical Nanoscale System* problem with a single agent per partition. The first column shows the number of iteration. As before, the columns two to five, describe the number of nodes after the corresponding step. The last column shows the value at the end of the iteration.

6.3 Isomorphic DecPOMDPs

We explain in Section 5.1 that we skip the experimental results for counting DecPOMDPs, because they cannot improve the critical sections of the algorithm significantly. Thus, we concentrate our experimental results on isomorphic DecPOMDPs. Since there are no known isomorphic symmetries within our first three domains, we focus on the last domain; the *Medical Nanoscale System*. On the one hand, we show isomorphic DecPOMDPs can indeed improve the performance on solving large DecPOMDPs, by comparing the results with the ground DecPOMDP's ones. On the other hand we show this approach has its limitations too, as it is still exponential within the number of agents, because we need to consider every observation combination.

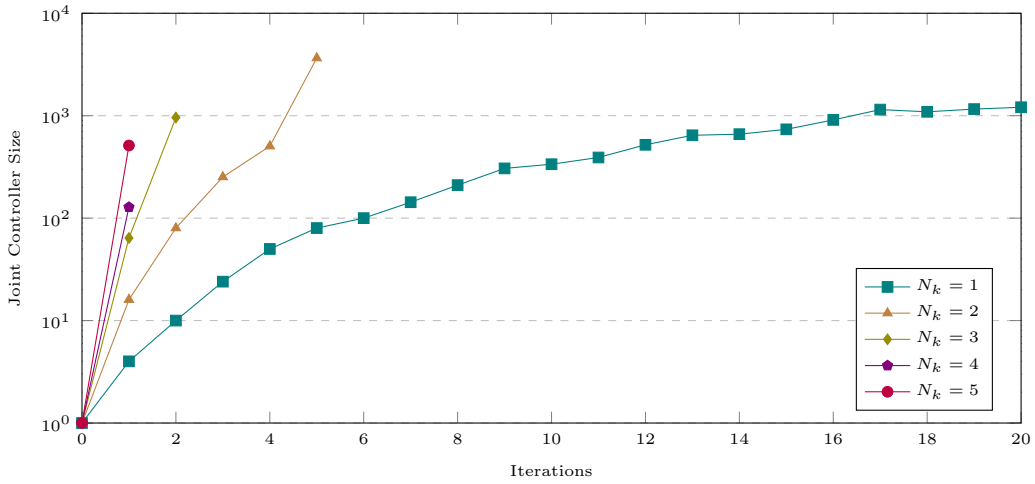


Figure 6.2: Logarithmic visualization of joint controller sizes of the ground DecPOMDP representing the *Medical Nanoscale System* problem in relation to the iteration by partition size.

We use our modified algorithm (cf. Algorithm 5.1) for executing a policy ranking on a representative DecPOMDP and a policy evaluation on the original isomorphic DecPOMDP afterwards. We limit the number of iterations performed on the representative DecPOMDP to ensure that these finish with enough remaining time to evaluate the policy iteration. Our previous results (cf. Table 6.4) show that our algorithm can perform 20 iterations on the *Medical Nanoscale System* with one agent per partition. Since the policy evaluation is computationally intensive in relation to the partition sizes, we decrease the number of performed iterations on the ground DecPOMDP as we increase the partition sizes. Due to fewer iterations, the controllers remain smaller, which leads to less joint node vectors that need to be considered during policy evaluation. Thus, we can ensure the completion of the policy evaluation within its time constraints.

We omit the value during the iterations which we listed previously as we only perform a policy ranking. Thus, the calculated value during the iteration does not reflect the real value for the isomorphic DecPOMDP. Instead, we describe how many iterations we can complete before evaluating the policy without exceeding the time limit of 24 hours and the evaluated value of the isomorphic joint controller. Again, we underline that the value of these results cannot be compared to the ones from the ground DecPOMDP, in the matter of the achieved value, since we make stronger assumptions on the symmetries as in the ground case. Table 6.5 shows our results on the *Medical Nanoscale System* with various partition sizes. It visualizes that the number of possible iterations decreases rapidly as partition sizes increase. The isomorphic DecPOMDP is capable of ten agents per partition with the loss of performable iterations.

With growing partition sizes the effective horizon of the expected reward tends to zero. Thus, only the immediate reward, the reward for the first selected actions based on the initial belief state, is considered, because we make restrictive assumptions about the selectable action vectors. As one can see in Equation (5.4), the transition probabilities are exponential within the grounding constant, which grows with the partition sizes. This dependence zeros the probabilities for transition from one pair of state and node vector to another, which eradicates the second term of the value function, which is equivalent to a value function with horizon zero.

6.4 Representative Observations

Finally, we present our results for solving isomorphic DecPOMDPs with representative observations. Again, we solve the *Medical Nanoscale System* problem for different partition sizes. In contrast to our previous experiments, we make further assumptions about the agent's behavior to be able to reduce the complexity further on. As before, we utilize Algorithm 5.1, but with the optimizations described in Section 5.3. We execute the tests the same way as for isomorphic DecPOMDPs with individual observations: We perform as many iterations on the representative DecPOMDP as possible and then we evaluate the generated joint policy for all agents. If the policy evaluation does not complete, we reduce the iterations and repeat the test.

The results of our tests are depicted in in Table 6.6. We repeated the execution with various partition sizes, testing the boundaries of this approach. The results are outstanding, as we are able to complete 20 iterations with 200,000 agents within the time limit of 24 hours. We expect for even larger models that the number of possible iterations does not decrease significantly, due to the logarithmic dependence on the number of iterations. This result stresses the far better scalability of this approach. As we observe in the context of isomorphic DecPOMDPs without representative observations, the effective horizon of these problems becomes zero, since the probabilities scale exponentially with the grounding constant.

Isomorphic Medical Nanoscale System					
N_k	**It.**	**Value**	N_k	**It.**	**Value**
1	20	1.16E+02	6	2	3.60E+02
2	10	5.32E+01	7	2	4.90E+02
3	4	9.02E+01	8	1	6.40E+02
4	3	1.60E+02	9	1	8.10E+02
5	3	2.50E+02	10	1	1.00E+03

Table 6.5: Experimental results of isomorphic *Medical Nanoscale System* problem. The first column shows the partition sizes, the second coulumn how many iterations were possible on the representative DecPOMDP and the third column contains the value for the DecPOMDP. The remaining columns continue the first three ones.

Iso. Medical Nanoscale System with repres. observations					
N_k	**It.**	**Value**	N_k	**It.**	**Value**
1	20	1.16E+02	40	20	1.60E+04
2	20	6.84E+01	45	20	2.03E+04
3	20	1.02E+02	50	20	2.50E+04
4	20	1.63E+02	60	20	3.60E+04
5	20	2.49E+02	70	20	4.90E+04
6	20	3.60E+02	80	20	6.40E+04
7	20	4.90E+02	90	20	8.10E+04
8	20	6.40E+02	100	20	1.00E+05
9	20	8.10E+02	500	20	2.50E+06
10	20	1.00E+03	1,000	20	1.00E+07
20	20	4.00E+03	5,000	20	2.50E+08
25	20	6.25E+03	10,000	20	1.00E+09
30	20	9.00E+03	50,000	20	2.50E+10
35	20	1.23E+04	100,000	20	1.00E+11

Table 6.6: Experimental results of isomorphic *Medical Nanoscale System* problem with representative observations. The first column shows the partition sizes. Column two shows how many iterations were possible on the representative DecPOMDP. The third column contains the value for the isomorphic DecPOMDP with representative observa. The remaining columns continue the first three ones.

7 Discussion

Our results point out, using FSCs for large DecPOMDPs is quite challenging as the joint controller does scale exponentially within the number of agents. Because the number of nodes grows in every round in an exponential manner, the algorithm does exceed its limitations within a few agents. In the case of the *Medical Nanoscale System*, we are able to handle up to ten agents, but the number of completable iterations reduces to one. Because the algorithm is an iterative one, it does not provide good results after a single iteration.

Counting DecPOMDPs can reduce the representational size of the DecPOMDP, since the transition-, reward- and observation function can be reduced in size. However, the size of the joint controller cannot be reduced, because each agent of a partition in a counting DecPOMDP still has its own local policy. This dependence leads to an individual local controller for each agent. Thus, the number of joint node vectors is still exponentially within the number of agents. Since the computationally intensive parts like calculating and solving the linear equation system heavily depend on the number of joint node vectors, counting DecPOMDPs cannot mitigate the problem of bad scalability.

With the use of isomorphic DecPOMDPs, we are able to extend the limitations although the problem remains exponential within the number of agents. Partitions in isomorphic DecPOMDPs can share a single controller across all agents, which can reduce the number of joint node vectors. Furthermore, isomorphic DecPOMDPs allow to perform policy ranking on a representative DecPOMDP, which is not depending on the number of agents, but only on the number of partitions. When the policy ranking finishes or stops, the current controller is transferred to the isomorphic DecPOMDP and the policy of the isomorphic DecPOMDP is evaluated. These performance improvements manifest themselves in the extended limitations, as we are able to perform ten iterations for four agents and three iterations for ten agents, which is a moderate improvement compared to ground DecPOMDPs. Unfortunately, this approach has its limitations too, as the policy evaluation is still not tractable for large amounts of agents. Figure 7.1 depicts the resulting improvements. It can be seen that isomorphic DecPOMDPs improve the possible iterations slightly, but not significantly.

Here, the role of representative observations becomes evident. Constraining isomorphic DecPOMDPs further helps to reduce the complexity of solving the value function as we presume each agent to be fully indistinguishable and to make identical observations. This assumption leads to some further sim-

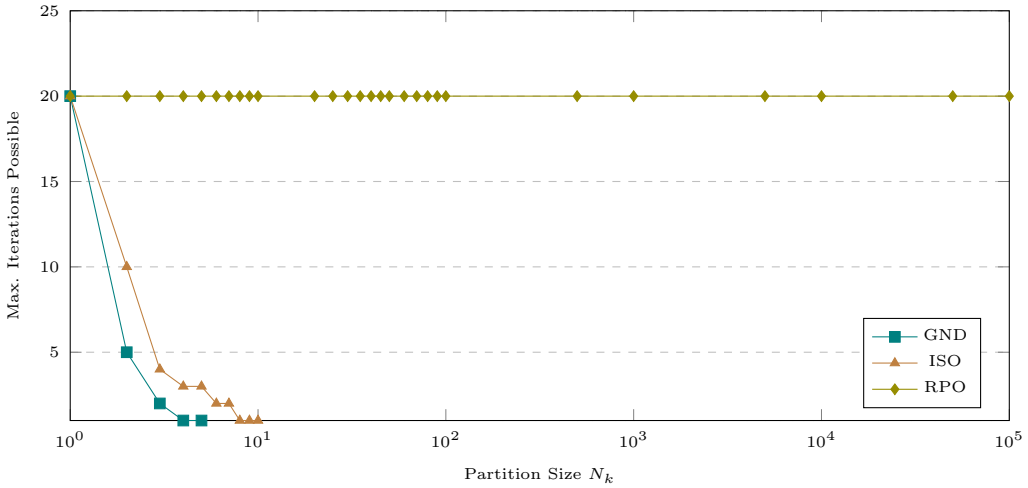


Figure 7.1: Comparison of solving large DecPOMDPs as ground DecPOMDP (GND), as isomorphic DecPOMDPs (ISO) and as isomorphic DecPOMDP with representative observations (RPO) with the (modified) HPI algorithm. The horizontal axis shows the partition sizes logarithmically, whereas the vertical axis shows the number of iterations the approach was able to complete.

plifications becoming applicable. Thus, the policy evaluation does not scale exponentially, but logarithmical with the number of agents. Because of that weak coupling, we are able to complete 20 iterations on ten agents, instead of three or one as before. This weak computational dependence on the number of agents allows us to perform the policy evaluation after 20 iterations on 200,000 agents. Figure 7.1 also visualizes these improvements. The number of iterations being possible emphasizes the low coupling to the number of agents.

In this thesis, we work with a basic algorithm that is based on FSCs. The HPI algorithm only allows to solve small DecPOMDPs with up to ten agents. A significant amount of research has been dedicated to optimising algorithms for solving infinite-horizon DecPOMDPs with the objective of achieving near-optimal values often at the cost of scalability. Thus, many previous algorithms and approaches were also only capable of handling a small number of agents [16, 30, 57]. The scalability of FSCs in our approach is coupled to (strong) constraints, namely isomorphic symmetries within the agent set and the assumption of representative observations. Otherwise, the biggest problem of numerous joint node vectors that need to be considered remains open. One drawback, which comes with large isomorphic DecPOMDPs with and without representative observations, is the reduction of the effective horizon of the expected reward towards zero. Fortunately, the effective horizon does only apply to the expected reward and has no influence on the local policies, since those are computed beforehand.

8 Conclusion and Outlook

To conclude, this thesis presents the scalability of FSCs in the context of DecPOMDPs on the basis of the HPI algorithm by Bernstein *et al.* [9]. We show that the scalability is strongly limited in the ground case. Additionally, we extend the algorithm to support isomorphic DecPOMDPs and take advantage of the given symmetries. Isomorphic DecPOMDPs include indistinguishability of agents within a partition, which leads to the circumstance that agents in a partition act the same for the same history. From this independence, we can simplify the joint controller by defining a single local controller for a whole partition instead of one local controller for each agent of every partition as in the ground case. This adjustment improves the scalability of the FSC a little, especially since the policy ranking can be performed on a representative ground DecPOMDP and the policy is evaluated for the whole DecPOMDP afterwards. We further show that the additional assumption of representative observations improves the scalability much more, because of the improvements that can be applied due to the isomorphism and the fact that each agent acts exactly the same in all matters. Latter allows the calculation of the value function to be massively simplified. This simplification leads to logarithmical dependence on the grounding constant and allows to perform policy ranking with FSCs for DecPOMDPs with hundreds of thousands of agents. In addition to our results on the use of FSCs for large DecPOMDPs, we present a software framework for solving DecPOMDPs that is to be extended.

Based on our promising results, there are broad areas of research to investigate further. In the the remainder of this chapter we focus on four of those, and explain how these approaches could help to solve large DecPOMDPs with stochastic FSCs.

Correlation Bernstein *et al.* [9] showed that independence can be limiting. To mitigate this limitation, the concept of correlation is applied [4, 9, 55]. Thus, a correlation device is used to define a correlated joint controller. This correlation device is a simple controller consisting of nodes and a node transition function, which only depends on the current correlation node and helps the agent's controller to synchronize. Synchronizing can help to reduce the controller in size and could lead to smaller controllers and therefore to slower controller growth. As the number of joint controller nodes which now have to include a correlation node could decrease, using a correlation device could lead to potential performance and capability improvements.

Exploiting structures in the state space Not only structures within the agent set can be exploited to optimize the performance of solving DecPOMDPs. Factored state spaces and factored value functions are objective to various research [12, 22, 42, 43, 45, 47]. Oliehoek, Whiteson, and Spaan [42] stated that they achieved high quality solutions for solving DecPOMDPs with hundreds of agents. However, instead of policy iteration with FSCs, they used a so called "forward-sweep policy computation method" [42]. But, we see potential in factored state space, since factorizing the state space could help generating belief points and pruning parts of the FSC. Obviously, this is especially important for DecPOMDPs with a large state space.

Expectation Maximization Kumar and Zilberstein [32] presented the solving technique of *Expectation Maximization* that is objective to recent research [33, 45, 56]. This approach uses graphical dynamic Bayesian networks (DBN) which allow probabilictic inference techniques to be applied for solving [32]. Kumar and Zilberstein [32] presented that this algorithm outperforms approaches as the HPI or optimizing fixed-size FSCs from Bernstein *et al.* [9]. Further extensions, e.g. by using *Monte-Carlo Expectation Maximization* as propsed by Wu, Zilberstein, and Jennings [56], allow to solve large ground DecPOMDPs with up to 2,500 agents. Due to this scalability and the allowance for inference, future work should investigate if the proposed improvements can also be applied on DBNs to combine the advantages of both approaches. This raises potential for even better scalability with perhaps less constraints.

Parallelization on GPU Instead of further optimizing the algorithm itself, a contrasting approach is to optimize the implementation and its performance to equalize computational complexity. As we emphasized during our thesis, most parts of the algorithm are heavily data-parallel, which means the same operations are executed on different data. Graphical processing units (GPUs) are often optimized to perform data-parallel tasks, e.g. calculating graphics. Those GPUs can also be addressed for non-graphical computations, which often leads to massive performance improvements [21, 29].

A Comparison of Libraries for Solving Linear Equation Systems

As solving linear equation system of various sizes is crucial for the execution of the proposed algorithm, we compare different libraries in order to be able to make a profound decision on which library we use for solving equation systems.

There are thousands of different libraries in various programming languages that handle with linear algebra and equation systems. We base our selection on the *Java Matrix Benchmark - Runtime: Xeon W-1250* [2] and select two of the best performing pure-Java libraries. On the one hand, we work with *EJML* [1], which is programmed and maintained by Abeles, who is the author of the benchmark. On the other hand, we select *ojAlgo* [48], which is a small and fast library for linear algebra and optimization problems. Both libraries work without any dependencies and are both solely written in Java.

For testing, we generate random matrices and vectors of increasing sizes and measure the time each library needed to solve the system of linear equations. Both libraries consist of different solvers, which use different algorithms. We focus on those equation systems, where the number of equations equals the number of variables, since this is always the case during policy evaluation.

Size of LES	***SimpleMatrix***	***DDRM QR***	***DDRM LU***
100	8 ms	8 ms	**6 ms**
500	33 ms	59 ms	44 ms
1,000	269 ms	373 ms	263 ms
2,000	2,195 ms	2,068 ms	2,184 ms
4,000	18,773 ms	14,515 ms	18,928 ms
6,000	65,640 ms	46,539 ms	66,336 ms
8,000	158,569 ms	108,588 ms	158,416 ms
10,000	307,284 ms	337,978 ms	312,102 ms
12,000	621,742 ms	367,877 ms	-
14,000	844,075 ms	567,758 ms	-

Table A.1: Timings of solving randomly generated systems of linear equations of various sizes with different solving methods provided by the *EJML* library. Emphasized cells showed the best results for the given size of the equation system, across all tested solving methods and libraries.

Size of LES	*QR*	*LU*	*SVD*	*SolverTask*
100	30 ms	25 ms	55 ms	28 ms
500	50 ms	**30 ms**	248 ms	31 ms
1,000	**148 ms**	167 ms	1,472 ms	152 ms
2,000	478 ms	**491 ms**	10,616 ms	501 ms
4,000	4,682 ms	**5,162 ms**	92,073 ms	5,230 ms
6,000	16,808 ms	17,317 ms	-	**17,312 ms**
8,000	40,478 ms	39,899 ms	-	**39,727 ms**
10,000	78,422 ms	77,363 ms	-	**77,215 ms**
12,000	163,519 ms	**130,932 ms**	-	132,028 ms
14,000	254,134 ms	**216,120 ms**	-	228,250 ms

Table A.2: Timings of solving randomly generated systems of linear equations of various sizes with different solving methods provided by the *ojAlgo* library. Emphasized cells showed the best results for the given size of the equation system, across all tested solving methods and libraries.

Table A.1 shows the results for the tests described above. We execute the test with three different solvers from *EJML*: *SimpleMatrix*, *QR* and *LU*. *SimpleMatrix* is a high-level interface, which selects automatically if it uses QR or LU decomposition based on the shape of the matrix. *QR* and *LU* are both more low level interfaces, which use QR (LU) decomposition for solving the equation system. As we see, the QR decomposition performs best for large equation systems. The overhead of the higher level interface scales with the size of the system, which leads to a fairly good performance for equation systems with up to 2,000 variables.

In Table A.2 we list the results of testing *ojAlgo*. Again, we execute the test with various solving methods of the library: *QR*, *LU*, *SVD* and *SolverTask*. As before, *QR*, *LU* and *SVD* are low level interfaces, which use QR (LU, SVD) decomposition for solving. *SolverTask* is a more high level interface, which determines based on the shape and structure of the matrix which decomposition method is used. In contrast to the higher level interface of *EJML*, *SolverTask* is able to compete at large sizes and performs best for equation system between 6,000 and 10,000 variables and equations. In the other cases *LU* performs best, with a few exceptions. *SVD* is not able to solve equation systems with more than 4,000 variables.

We highlight the best performing solver for each size across Table A.1 and Table A.2. We see that the *LU* solver of *ojAlgo* performs overall the best. Although it does not outperform at every size, it is always close to the fastest solver. Thus, we focus on using the *LU* solver from *ojAlgo*.

B Comparison of Libraries for Solving Linear Programs

Solving linear programs is another crucial task to perform the proposed algorithm. When it comes to solving linear programs, not only the speed does matter, but also the capability of finding an (optimal) solution. Especially, as finding solutions leads to pruning nodes, which reduces the complexity for the next steps of the algorithm. In this chapter, we present our results on comparing different libraries and their capabilities of solving linear programs.

Since there exists a broad palette of libraries for such task, we focus on three selected libraries. First, we choose *ojAlgo* [48] (*OJA*), because it performs pretty well at solving linear equation systems. We anticipate that it also has good linear program solving capabilities. *Commons Math* [5] (*ACM*) is the second library. We choose it for its popularity, as it is one of the most downloaded math libraries on the Java package manager Maven [38]. Third, we choose *OR-Tools* [20] (*OR*), because it works with native code, which could potentially lead to faster computations.

To test these libraries we use the the collection of linear programs from Netlib [18]. It consists of 97 feasible linear programs that are to be minimized. The linear programs are described in the `.mps` file format [37], which is not only supported by open source tools, as the ones we mentioned above, but also is supported by commercial solvers as *CPLEX* [23]. It is sometimes labelled as the de-facto industry standard [23].

To parse the `.mps` files, we use the parser from *OJA*. We parse every linear program, but had problems to parse `AGG`. Thus, we skipp this problem. *OJA* provides extensions to transform these problems from the internal format, into a format, which *ACM* and *OR* can handle and solve. Finally, we measure the time each library needs to compute a solution or to state that no solution exists. We summarize our results for all remaining 96 programs in the following Table B.1. The fastest library run is highlighted for every linear program. Table B.1 shows that all results are close to each other. *OJA* performs best in 60 of 96 programs, whereas *ACM* performs best in 59 cases and *OR* only performs best in 46 cases. This is probably due the parsing of *OJA*, which requires transforming and transferring the problem to solve it with *OR* or *ACM*. Nevertheless, because of the closeness of the results, we choose *OJA* as a library for solving linear programs.

Problem	**# Var.**	**# Const.**	***OJA***	***ACM***	***OR***
25FV47	1,571	822	**99 ms**	100 ms	139 ms
80BAU3B	9,799	2,263	183 ms	**173 ms**	195 ms
ADLITTLE	97	57	**1 ms**	2 ms	2 ms
AFIRO	32	28	**0 ms**	1 ms	1 ms
AGG2	302	517	**7 ms**	**7 ms**	8 ms
AGG3	302	517	**7 ms**	**7 ms**	8 ms
BANDM	472	306	**12 ms**	15 ms	**12 ms**
BEACONFD	262	174	**3 ms**	**3 ms**	**3 ms**
BLEND	83	75	**1 ms**	**1 ms**	**1 ms**
BNL1	1,175	644	**37 ms**	38 ms	38 ms
BNL2	3,489	2,325	**89 ms**	**89 ms**	90 ms
BOEING1	384	352	12 ms	12 ms	**11 ms**
BOEING2	143	167	3 ms	**2 ms**	**2 ms**
BORE3D	315	234	4 ms	**3 ms**	5 ms
BRANDY	249	221	**5 ms**	**5 ms**	**5 ms**
CAPRI	353	272	**6 ms**	**6 ms**	7 ms
CYCLE	2,857	1,904	82 ms	**68 ms**	88 ms
CZPROB	3,523	930	**26 ms**	**26 ms**	**26 ms**
D2Q06C	5,167	2,172	577 ms	**555 ms**	581 ms
D6CUBE	6,184	416	**454 ms**	455 ms	460 ms
DEGEN2	534	445	**15 ms**	16 ms	16 ms
DEGEN3	1,818	1,504	157 ms	**154 ms**	163 ms
DFL001	12,230	6,072	3,719 ms	**3,713 ms**	3,742 ms
E226	282	224	**4 ms**	5 ms	6 ms
ETAMACRO	688	401	**10 ms**	11 ms	11 ms
FFFFF800	854	525	8 ms	**7 ms**	8 ms
FINNIS	614	498	15 ms	10 ms	**9 ms**
FIT1D	1,026	25	**20 ms**	**20 ms**	22 ms
FIT1P	1,677	628	**25 ms**	**25 ms**	**25 ms**
FIT2D	10,500	26	**1,336 ms**	**1,336 ms**	1,350 ms
FIT2P	13,525	3,001	616 ms	**603 ms**	612 ms
FORPLAN	421	162	**4 ms**	5 ms	**4 ms**
GANGES	1,681	1,310	**32 ms**	33 ms	36 ms
GFRD-PNC	1,092	617	**10 ms**	11 ms	12 ms
GREENBEA	5,405	2,393	776 ms	774 ms	**768 ms**
GREENBEB	5,405	2,393	617 ms	**612 ms**	622 ms
GROW15	645	301	17 ms	**15 ms**	16 ms
GROW22	946	441	**36 ms**	**36 ms**	**36 ms**
GROW7	301	141	**5 ms**	**5 ms**	**5 ms**
ISRAEL	142	175	3 ms	**2 ms**	3 ms
KB2	41	44	1 ms	**0 ms**	**0 ms**

Problem	**# Var.**	**# Const.**	***OJA***	***ACM***	***OR***
LOTFI	308	154	**1 ms**	2 ms	2 ms
MAROS-R7	9,408	3,137	559 ms	**556 ms**	564 ms
MAROS	1,443	847	**38 ms**	**38 ms**	39 ms
MODSZK1	1,620	688	**17 ms**	18 ms	**17 ms**
NESM	2,923	663	**61 ms**	62 ms	69 ms
PEROLD	1,376	626	46 ms	**45 ms**	46 ms
PILOT-JA	1,988	941	**127 ms**	**127 ms**	131 ms
PILOT-WE	2,789	723	**56 ms**	**56 ms**	**56 ms**
PILOT	3,652	1,442	**739 ms**	750 ms	829 ms
PILOT4	1,000	411	**19 ms**	20 ms	20 ms
PILOT87	4,883	2,031	**1,941 ms**	1,950 ms	1,956 ms
PILOTNOV	2,172	976	**87 ms**	**87 ms**	88 ms
QAP12	8,856	3,193	13,370 ms	**13,079 ms**	13,095 ms
QAP15	22,275	6,331	139,813 ms	**138,758 ms**	140,307 ms
QAP8	1,632	913	190 ms	**186 ms**	187 ms
RECIPELP	180	92	**1 ms**	**1 ms**	**1 ms**
SC105	103	106	1 ms	1 ms	**0 ms**
SC205	203	206	**2 ms**	**2 ms**	**2 ms**
SC50A	48	51	1 ms	1 ms	**0 ms**
SC50B	48	51	1 ms	**0 ms**	**0 ms**
SCAGR25	500	472	**5 ms**	15 ms	6 ms
SCAGR7	140	130	**1 ms**	**1 ms**	**1 ms**
SCFXM1	457	331	**5 ms**	6 ms	**5 ms**
SCFXM2	914	661	14 ms	**13 ms**	14 ms
SCFXM3	1,371	991	**22 ms**	**22 ms**	**22 ms**
SCORPION	358	389	**2 ms**	3 ms	3 ms
SCRS8	1,169	491	21 ms	20 ms	**16 ms**
SCSD1	760	78	4 ms	**3 ms**	**3 ms**
SCSD6	1,350	148	7 ms	**6 ms**	**6 ms**
SCSD8	2,750	398	25 ms	**22 ms**	24 ms
SCTAP1	480	301	**2 ms**	3 ms	3 ms
SCTAP2	1,880	1,091	9 ms	**8 ms**	9 ms
SCTAP3	2,480	1,481	**13 ms**	14 ms	14 ms
SEBA	1,028	516	13 ms	**7 ms**	**7 ms**
SHARE1B	225	118	**3 ms**	**3 ms**	**3 ms**
SHARE2B	79	97	**1 ms**	**1 ms**	**1 ms**
SHELL	1,775	537	**14 ms**	**14 ms**	**14 ms**
SHIP04L	2,118	403	**7 ms**	**7 ms**	**7 ms**
SHIP04S	1,458	403	**5 ms**	**5 ms**	6 ms
SHIP08L	4,283	779	**14 ms**	16 ms	16 ms
SHIP08S	2,387	779	9 ms	**8 ms**	9 ms

Problem	**# Var.**	**# Const.**	***OJA***	***ACM***	***OR***
SHIP12L	5,427	1,152	18 ms	18 ms	**17 ms**
SHIP12S	2,763	1,152	12 ms	**11 ms**	12 ms
SIERRA	2,036	1,228	**17 ms**	23 ms	18 ms
STAIR	467	357	**10 ms**	11 ms	12 ms
STANDATA	1,075	360	**12 ms**	14 ms	13 ms
STANDGUB	1,184	362	15 ms	**13 ms**	14 ms
STANDMPS	1,075	468	**16 ms**	**16 ms**	**16 ms**
STOCFOR1	111	118	**1 ms**	**1 ms**	**1 ms**
STOCFOR2	2,031	2,158	**27 ms**	**27 ms**	29 ms
STOCFOR3	15,695	16,676	**553 ms**	572 ms	562 ms
TRUSS	8,806	1,001	231 ms	226 ms	**220 ms**
VTP-BASE	203	199	**4 ms**	**4 ms**	**4 ms**
WOOD1P	2,594	245	**36 ms**	**36 ms**	41 ms
WOODW	8,405	1,099	**214 ms**	225 ms	216 ms

Table B.1: Time needed for minimizing linear programs with various sizes. First column contains the name of each problem instance. The next two columns describe how many variables and constraints there exist. The remaining columns denote the time needed by the tested libraries. Emphasized cells showed the best results for the given linear program, across all libraries.

Bibliography

[1] Peter Abeles. *Efficient Java Matrix Library*. Version 0.43.1. Sept. 23, 2023. URL: https://ejml.org (visited on 10/01/2024).

[2] Peter Abeles. *Java Matrix Benchmark - Runtime: Xeon W-1250*. Dec. 2021. URL: https://lessthanoptimal.github.io/Java-Matrix-Benchmark/runtime/2021_12_Xeon1250/ (visited on 10/01/2024).

[3] Christopher Amato, Daniel S Bernstein, and Shlomo Zilberstein. "Optimal Fixed-Size Controllers for Decentralized POMDPs". In: *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. AAMAS'06. Hakodate, Japan: Association for Computing Machinery, May 8, 2006. ISBN: 978-1-59593-303-4. URL: https://dl.acm.org/doi/proceedings/10.1145/1160633 (visited on 09/15/2024).

[4] Christopher Amato, Daniel S. Bernstein, and Shlomo Zilberstein. "Optimizing Memory-Bounded Controllers for Decentralized POMDPs". In: *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*. UAI'07. Vancouver, Canada: AUAI Press, 2007, pp. 1–8. ISBN: 0-9749039-3-0. URL: http://arxiv.org/abs/1206.5258 (visited on 09/03/2024).

[5] Apache. *Commons Math*. Version 3.6.1. Mar. 31, 2016. URL: https://commons.apache.org/proper/commons-math/ (visited on 08/20/2024).

[6] K.J Åström. "Optimal control of Markov processes with incomplete state information". In: *Journal of Mathematical Analysis and Applications* 10.1 (Feb. 1965), pp. 174–205. DOI: 10.1016/0022-247X(65)90154-X. (Visited on 05/21/2024).

[7] Aurelien Bellet, Amaury Habrard, and Marc Sebban. *Metric learning*. Synthesis Lectures on artificial intelligence and machine learning 30. Morgan & Claypool Publishers, 2015. 151 pp. ISBN: 978-1-62705-366-2.

[8] Richard Bellman. "A Markovian Decision Process". In: *Indiana University Mathematics Journal* 6.4 (1957), pp. 679–684. ISSN: 0022-2518. URL: https://www.iumj.indiana.edu/IUMJ/FULLTEXT/1957/6/56038.

[9] D. S. Bernstein *et al.* "Policy Iteration for Decentralized Control of Markov Decision Processes". In: *Journal of Artificial Intelligence Research* 34 (Mar. 1, 2009), pp. 89–132. ISSN: 1076-9757. DOI: 10.1613/jair.2667. (Visited on 04/09/2024).

[10] Daniel S Bernstein, Eric A Hansen, and Shlomo Zilberstein. "Bounded Policy Iteration for Decentralized POMDPs". In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence*. IJCAI'05. Edinburgh, Scotland: Morgan Kaufmann Publishers Inc., July 30, 2005, pp. 1287–1292. URL: https://dl.acm.org/doi/10.5555/1642293.1642498 (visited on 09/15/2024).

[11] Daniel S. Bernstein *et al.* "The Complexity of Decentralized Control of Markov Decision Processes". In: *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*. UAI'00. Stanford, United States: Morgan Kaufmann Publishers Inc., June 30, 2000, pp. 32–37. ISBN: 978-1-55860-709-5. URL: https://dl.acm.org/doi/10.5555/2073946.2073951 (visited on 10/07/2024).

[12] Xavier Boyen and Daphne Koller. "Exploiting the Architecture of Dynamic Systems". In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*. AAAI '99/IAAI '99. Orlando, Florida, USA: American Association for Artificial Intelligence, July 18, 1999, pp. 313–320. ISBN: 0-262-51106-1. URL: https://dl.acm.org/doi/abs/10.5555/315149.315313 (visited on 09/15/2024).

[13] Tanya Braun *et al.* "Lifting DecPOMDPs for Nanoscale Systems – A Work in Progress". In: *Tenth International Workshop on Statistical Relational AI*. StarAI'21. Oct. 18, 2021. DOI: 10.48550/arXiv.2110.09152. (Visited on 07/09/2024).

[14] Tanya Braun *et al.* "Lifting in Multi-agent Systems under Uncertainty". In: *Proceedings of the 38th Conference on Uncertainty in Artificial Intelligence*. UAI'22. Vol. 180. Proceedings of Machine Learning Research. Eindhoven, Netherlands: PMLR, Aug. 2022, pp. 233–243. URL: https://proceedings.mlr.press/v180/braun22a.html (visited on 09/15/2024).

[15] Constantin Castan. "Lifting Multi-agent A* - Implementing a New Optimisation Problem in Isomorphic DecPOMDPs: How Many Agents Do We Need?" Master thesis. Münster, Germany: University of Münster, Nov. 2022. 35 pp. URL: https://www.uni-muenster.de/Informatik.AGBraun/en/theses/ma220504.html (visited on 09/15/2024).

[16] Barış Eker and H. Levent Akın. "Solving decentralized POMDP problems using genetic algorithms". In: *Autonomous Agents and Multi-Agent Systems* 27.1 (July 2013), pp. 161–196. DOI: 10.1007/s10458-012-9204-y. (Visited on 09/07/2024).

[17] Erich Gamma *et al. Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Boston, United States: Addison-Wesley Longman Publishing Co., Jan. 2, 1995. 395 pp.

ISBN: 978-0-201-63361-0. URL: https://dl.acm.org/doi/10.5555/186897 (visited on 08/08/2024).

[18] David M. Gay. *netlib LP/DATA*. Aug. 22, 2013. URL: https://netlib.org/lp/data/ (visited on 08/28/2024).

[19] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Fourth edition. Johns Hopkins studies in the mathematical sciences. Baltimore, United States: The Johns Hopkins University Press, 2013. 756 pp. ISBN: 978-1-4214-0794-4.

[20] Google Inc. *OR-Tools*. Version 9.10.4067. May 2024. URL: https://developers.google.com/optimization (visited on 08/20/2024).

[21] Scott Grauer-Gray *et al.* "Accelerating financial applications on the GPU". In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. GPGPU-6. Houston, United States: Association for Computing Machinery, Mar. 16, 2013, pp. 127–136. ISBN: 978-1-4503-2017-7. DOI: 10.1145/2458523.2458536. (Visited on 09/24/2024).

[22] Carlos Guestrin *et al.* "Efficient Solution Algorithms for Factored MDPs". In: *Journal of Artificial Intelligence Research* 19 (2003), pp. 399–468. DOI: 10.1613/jair.1000. (Visited on 09/15/2024).

[23] IBM Corporation. *MPS file format: industry standard*. ILOG CPLEX Optimization Studio. Dec. 9, 2022. URL: https://www.ibm.com/docs/en/icos/22.1.1?topic=cplex-mps-file-format-industry-standard (visited on 08/28/2024).

[24] JetBrains. *Java Programming - The State of Developer Ecosystem 2021*. The State of Developer Ecosystem. July 2021. URL: https://www.jetbrains.com/lp/devecosystem-2021/java/ (visited on 08/05/2024).

[25] JetBrains. *Java Programming - The State of Developer Ecosystem 2022*. The State of Developer Ecosystem. July 2022. URL: https://www.jetbrains.com/lp/devecosystem-2022/java/ (visited on 08/05/2024).

[26] JetBrains. *Java Programming - The State of Developer Ecosystem 2023*. The State of Developer Ecosystem. July 2023. URL: https://www.jetbrains.com/lp/devecosystem-2023/java/ (visited on 08/05/2024).

[27] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. "Planning and acting in partially observable stochastic domains". In: *Artificial Intelligence* 101.1 (May 1998), pp. 99–134. DOI: 10.1016/S0004-3702(98)00023-X. (Visited on 04/10/2024).

[28] Byung Kon Kang and Kee-Eung Kim. "Exploiting symmetries for single- and multi-agent Partially Observable Stochastic Domains". In: *Artificial Intelligence* 182-183 (May 2012), pp. 32–57. DOI: 10.1016/j.artint.2012.01.003. (Visited on 09/09/2024).

[29] Andrew Kerr, Dan Campbell, and Mark Richards. "QR decomposition on GPUs". In: *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU '09. Washington D.C., United States: Association for Computing Machinery, Mar. 8, 2009, pp. 71–78. ISBN: 978-1-60558-517-8. DOI: 10.1145/1513895.1513904. (Visited on 09/24/2024).

[30] Youngwook Kim and Kee-Eung Kim. "Point-based bounded policy iteration for decentralized POMDPs". In: *Proceedings of the 11th Pacific Rim International Conference on Trends in Artificial Intelligence*. PRICAI'10. Daegu, Korea: Springer-Verlag, 2010, pp. 614–619. ISBN: 3-642-15245-7. URL: https://dl.acm.org/doi/10.5555/1884293.1884355 (visited on 09/09/2024).

[31] Daphne Koller and Ronald Parr. "Policy Iteration for Factored MDPs". In: *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*. UAI'00. Stanford, United States: Morgan Kaufmann Publishers Inc., June 30, 2000, pp. 326–334. ISBN: 978-1-55860-709-5. URL: https://dl.acm.org/doi/10.5555/2073946.2073985 (visited on 09/15/2024).

[32] Akshat Kumar and Shlomo Zilberstein. "Anytime planning for decentralized POMDPs using expectation maximization". In: *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*. UAI'10. Catalina Island, United States: AUAI Press, 2010, pp. 294–301. ISBN: 978-0-9749039-6-5. URL: https://dl.acm.org/doi/10.5555/3023549.3023584 (visited on 09/09/2024).

[33] Miao Liu *et al.* "Stick-breaking policy learning in Dec-POMDPs". In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI'15. Buenos Aires, Argentina: AAAI Press, July 25, 2015, pp. 2011–2017. ISBN: 978-1-57735-738-4. URL: https://dl.acm.org/doi/10.5555/2832415.2832528 (visited on 09/09/2024).

[34] Omid Madani, Steve Hanks, and Anne Condon. "On the Undecidability of Probabilistic Planning and Infinite-Horizon Partially Observable Markov Decision Problems". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI '99. Vol. 16. Orlando, United States: AAAI Press, July 18, 1999. ISBN: 978-0-262-51106-3. URL: https://aaai.org/papers/077-aaai99-077-on-the-undecidability-of-probabilistic-planning-and-infinite-horizon-partially-observable-markov-decision-problems/ (visited on 10/07/2024).

[35] Robert C. Martin, ed. *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, United States: Prentice Hall, 2009. 431 pp. ISBN: 978-0-13-235088-4.

[36] Nicolas Meuleau *et al.* "Learning Finite-State Controllers for Partially Observable Environments". In: *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*. UAI'99. Stockholm, Sweden: Morgan Kaufmann Publishers Inc., July 30, 1999. ISBN: 978-1-55860-614-2. URL: http://arxiv.org/abs/1301.6721 (visited on 04/10/2024).

[37] Bruce A. Murtagh. "Commericial Systems: Organization of Data". In: *Advanced Linear Programming: Computation and Practice*. McGraw-Hill International Book Company, 1981, pp. 163–176. ISBN: 978-0-07-044095-1.

[38] MvnRepository. *Math Libraries*. June 27, 2024. URL: https://mvnrepository.com/open-source/math-libraries (visited on 08/27/2024).

[39] Ranjit Nair *et al.* "Taming Decentralized POMDPs: Towards efficient policy computation for multiagent settings". In: *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*. IJCAI'03. Acapulco, Mexico: Morgan Kaufmann Publishers Inc., Aug. 9, 2003, pp. 705–711. URL: https://www.ijcai.org/Proceedings/03/Papers/103.pdf (visited on 09/15/2024).

[40] Frans Oliehoek *et al. The MADP Toolbox*. In collab. with Erwin Walraven *et al.* Version 0.4.1. Mar. 8, 2017. URL: https://www.fransoliehoek.net/fb/index.php?fuseaction=software.madp (visited on 05/22/2024).

[41] Frans A. Oliehoek and Christopher Amato. *A Concise Introduction to Decentralized POMDPs*. 1st ed. SpringerBriefs in Intelligent Systems. Springer Cham, June 3, 2016. ISBN: 978-3-319-28929-8. DOI: 10.1007/978-3-319-28929-8. URL: http://link.springer.com/10.1007/978-3-319-28929-8 (visited on 09/15/2024).

[42] Frans A. Oliehoek, Shimon Whiteson, and Matthijs T.J. Spaan. "Approximate Solutions for Factored Dec-POMDPs with Many Agents". In: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems*. AAMAS'13. Saint Paul, United States: International Foundation for Autonomous Agents and Multiagent Systems, May 6, 2013, pp. 563–570. ISBN: 978-1-4503-1993-5. URL: https://www.fransoliehoek.net/docs/Oliehoek13AAMAS.pdf (visited on 08/31/2024).

[43] Frans A. Oliehoek *et al.* "Exploiting locality of interaction in factored Dec-POMDPs". In: *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*. AAMAS'08. Vol. 1. Estoril, Portugal: International Foundation for Autonomous Agents and Multiagent Systems, May 12, 2008, pp. 517–524. ISBN: 978-0-9817381-0-9. URL: https://dl.acm.org/doi/abs/10.5555/1402383.1402457 (visited on 08/31/2024).

[44] Oracle. *Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification*. Sept. 14, 2021. URL: https://docs.oracle.com/en/java/javase/17/docs (visited on 08/07/2024).

[45] Joni Pajarinen and Jaakko Peltonen. "Efficient planning for factored infinite-horizon DEC-POMDPs". In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume One*. IJCAI'11. Barcelona, Catalonia, Spain: AAAI Press, 2011, pp. 325–331. ISBN: 978-1-57735-513-7. URL: https://dl.acm.org/doi/10.5555/2283396.2283451 (visited on 09/09/2024).

[46] Joni Pajarinen and Jaakko Peltonen. "Periodic Finite State Controllers for Efficient POMDP and DEC-POMDP Planning". In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. NIPS'11. Granada, Spain: Curran Associates Inc., Dec. 12, 2011, pp. 2636–2644. ISBN: 978-1-61839-599-3. URL: https://dl.acm.org/doi/10.5555/2986459.2986753 (visited on 10/07/2024).

[47] Joni Pajarinen *et al.* "Efficient planning in large POMDPs through policy graph based factorized approximations". In: *Proceedings of the 2010 European Conference on Machine Learning and Knowledge Discovery in Databases: Part III*. ECML PKDD'10. Barcelona, Spain: Springer-Verlag, Sept. 20, 2010, pp. 1–16. ISBN: 3-642-15938-9. URL: https://dl.acm.org/doi/10.5555/1889788.1889790 (visited on 09/09/2024).

[48] Anders Peterson. *oj! Algorithms*. Version 53.3.0. Feb. 4, 2024. URL: https://www.ojalgo.org/ (visited on 09/04/2024).

[49] Pascal Poupart and Craig Boutilier. "Bounded Finite State Controllers". In: *Proceedings of the 16th International Conference on Neural Information Processing Systems*. NIPS'03. Whistler, Canada: MIT Press, Dec. 9, 2003, pp. 823–830. URL: https://dl.acm.org/doi/abs/10.5555/2981345.2981448 (visited on 10/07/2024).

[50] Sven Seuken and Shlomo Zilberstein. "Improved Memory-Bounded Dynamic Programming for Decentralized POMDPs". In: *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*. UAI'07. Vancouver, Canada: AUAI Press, 2007, pp. 344–351. ISBN: 0-9749039-3-0. URL: http://arxiv.org/abs/1206.5295 (visited on 08/11/2024).

[51] Thiago D. Simão, Marnix Suilen, and Nils Jansen. "Safe Policy Improvement for POMDPs via Finite-State Controllers". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 37.12 (June 26, 2023), pp. 15109–15117. DOI: 10.1609/aaai.v37i12.26763. (Visited on 09/09/2024).

[52] Daniel Szer and François Charpillet. "Point-based dynamic programming for DEC-POMDPs". In: *Proceedings of the 21st National Conference on Artificial Intelligence*. AAAI'06. Vol. 2. Boston, United States: AAAI Press, July 16, 2006, pp. 1233–1238. ISBN: 978-1-57735-281-5. URL: https://dl.acm.org/doi/10.5555/1597348.1597384 (visited on 09/03/2024).

[53] N. Taghipour *et al.* "Lifted Variable Elimination: Decoupling the Operators from the Constraint Language". In: *Journal of Artificial Intelligence Research* 47 (July 8, 2013), pp. 393–439. ISSN: 1076-9757. DOI: 10.1613/jair.3793. (Visited on 07/01/2024).

[54] H.S. Witsenhausen. "Separation of estimation and control for discrete time systems". In: *Proceedings of the IEEE* 59.11 (1971), pp. 1557–1566. ISSN: 0018-9219. DOI: 10.1109/PROC.1971.8488. (Visited on 09/21/2024).

[55] Feng Wu and Xiaoping Chen. "Solving Large-Scale and Sparse-Reward DEC-POMDPs with Correlation-MDPs". In: *RoboCup 2007: Robot Soccer World Cup XI*. Vol. 5001. Berlin: Springer, 2008, pp. 208–219. ISBN: 978-3-540-68847-1. URL: http://link.springer.com/10.1007/978-3-540-68847-1_18 (visited on 09/07/2024).

[56] Feng Wu, Shlomo Zilberstein, and Nicholas R Jennings. "Monte-Carlo Expectation Maximization for Decentralized POMDPs". In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*. IJCAI '13. Beijing, China: AAAI Press, Aug. 3, 2013. ISBN: 978-1-57735-633-2.

[57] Yang You *et al. Solving infinite-horizon Dec-POMDPs using Finite State Controllers within JESP*. Sept. 17, 2021. arXiv: 2109.08755[cs]. URL: http://arxiv.org/abs/2109.08755 (visited on 04/09/2024).

Declaration of Academic Integrity

I hereby confirm that this thesis on *Investigating the Practicability of Finite State Controllers for Large DecPOMDPs* is solely my own work and that I have used no sources or aids other than the ones stated. All passages in my thesis for which other sources, including electronic media, have been used, be it direct quotes or content references, have been acknowledged as such and the sources cited.

Josha Landsmann, Münster, October 9, 2024

I agree to have my thesis checked in order to rule out potential similarities with other works and to have my thesis stored in a database for this purpose.

Josha Landsmann, Münster, October 9, 2024