Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

Efficient MDP Algorithms in POMDPs.jl

Efektivní MDP algoritmy v knihovně POMDPs.jl

BACHELOR THESIS

Author:	Tomáš Omasta
Study Program:	Open Informatics
Specialization:	Artificial Intelligence and Computer Science
Supervisor:	Ing. Jan Mrkos
Year:	2021

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Omasta Tomáš**	Personal ID number:	**483740**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Efficient MDP Algorithms in POMDPs.jl

Bachelor’s thesis title in Czech:

Efektivní MDP algoritmy v knihovně POMDPs.jl

Guidelines:

The goal of the project is to add and improve the MDP methods in the POMDPs.jl package. To this end, student is expected to fulfill the following objectives:
1) Survey and analyze MDP methods implemented in the POMDPs.jl package. Survey literature for MDP solution methods. Focus on methods for finite horizon MDPs. Select method or methods for implementation.
2) Implement selected methods and associated interfaces in the POMDPs.jl package. Where applicable, maintain interoperability with the rest of the package.
3) Evaluate the performance of the implemented method(s) on existing MDP instances. Where applicable, design and implement new test instances (e.g. finite horizon MDP instances). Where applicable, benchmark the implemented method(s) against comparable existing solvers in the POMDPs.jl package.

Bibliography / sources:

[1] Russell, Stuart J. and Norvig, Peter - Artificial Intelligence: A Modern Approach (2nd Edition) – 2002
[2] Mausam, Kolobov, Andrey - Planning with Markov Decision Processes: An AI Perspective - 2012
[3] Bellman, Richard, - A Markovian Decision Process – 1957
[4] Maxim Egorov et al. - POMDPs.jl: A Framework for Sequential Decision Making under Uncertainty – 2017

Name and workplace of bachelor’s thesis supervisor:

Ing. Jan Mrkos, Artificial Intelligence Center, FEE

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

Date of bachelor’s thesis assignment: **08.01.2021** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **30.09.2022**

Ing. Jan Mrkos	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

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

.

Date of assignment receipt	Student’s signature

Declaration

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

Prague, 21.05.2021 Tomáš Omasta

Acknowledgements

I want to thank my Supervisor, Ing. Jan Mrkos, for all the little things he has done to make this possible, particularly for his patience. My thanks belong to my family and girlfriend for their support and to FEE CTU for all joys and sorrows it enriched me with during my studies. Finally, I thank Zachary Sunberg for valuable insight into POMDPs.jl framework and for his reviewing work.

Tomáš Omasta

Title:
Efficient MDP Algorithms in POMDPs.jl

Author: Tomáš Omasta

Field of study: Open Informatics
Subfield: Artificial Intelligence and Computer Science

Supervisor: Ing. Jan Mrkos
Artificial Intelligence Center

Abstract:
Markov Decision Processes are one of the most well-known methods used for modeling and solving stochastic planning problems. Until recently, the problems solved in minutes consisted of tens of states at most. With the development of computing resources and specialized algorithms, it is easier to solve problems greater than ever. Moreover, such problems often include time-span limiting the number of actions we can execute.
In this thesis, we survey and implement methods for solving MDPs and POMDPs within the Julia POMDPs.jl framework. The algorithms and interfaces we have implemented integrate with and are interoperable with the rest of the tools provided by the framework. In the end, we evaluate the performance of our specialized methods against general benchmarks on both new and literature test-cases to showcase the performance improvements.

Key words: MDP, POMDP, Finite Horizon, Point-Based, Value Iteration, POMDPs.jl

Název práce:
Efektivní MDP algoritmy v knihovně POMDPs.jl

Autor: Tomáš Omasta

Abstrakt:
Markovovy rozhodovací procesy jsou jednou z nejznámějších metod používaných pro modelování a řešení stochastických problémů plánování. Donedávna se problémy složené z maximálně desítek stavů řešily v řádu minut. S rozvojem výpočetních prostředků a specializovaných algoritmů je snadnější řešit problémy větší než kdy dříve. Takové problémy navíc často obsahují časové rozpětí omezující počet akcí, které můžeme provést.
V této práci zkoumáme a implementujeme metody pro řešení MDP a POMDP v rámci frameworku POMDPs.jl implementovaného v jazyce Julia. Algoritmy a rozhraní, které jsme implementovali, jsou integrovatelné a interoperabilní s ostatními nástroji, které framework poskytuje. Nakonec vyhodnocujeme výkonnost našich specializovaných metod oproti obecným benchmarkům na nových a z literatury převzatých testovacích případech, abychom ukázali zlepšení výkonnosti.
Klíčová slova: MDP, POMDP, Finite Horizon, Point-Based, Value Iteration, POMDPs.jl

Contents

List of Algorithms

List of definitions

List of Figures

List of Tables

Chapter 1

Introduction

Markov Decision Processes are one of the most well-known methods used for modeling and solving stochastic planning problems. Until recently, the problems solved in minutes consisted of tens of states at most. With the development of computing resources and specialized algorithms, it is easier to solve problems greater than ever. Moreover, such problems often include time-span limiting the number of actions we can execute.

In this thesis, we aim to survey and extend methods specialized on Finite Horizon Markov Decision Processes with both fully (MDP) and partially (POMDP) observable environments. Our task includes the addition of any other methods or interfaces missing to implement such methods and preserving interoperability inside the target framework. In the end, we evaluate the performance of the implemented methods on benchmarks and new test instances designed for their purpose.

For our thesis, we choose the Julia programming language. Julia is a young programming language that starts to emerge into the scientific community's consciousness. While offering a high-level approach comparable to the one of Python, it also offers a similar speed to languages like C or C++ at the same time. Moreover, Julia contains a framework used for working with Markov Decision Processes specifically. The framework is called $POMDPs.jl$.

The $POMDPs.jl$ framework is an open-source framework for defining, solving, and simulating both MDP and POMDP problems. It builds on several interoperating interfaces to clearly define the structure of all code associated with the framework. Thanks to this, it is easy to change the script's functionality by only a few changes, implement new methods, or improve existing ones.

As a part of our contribution to the $POMDPs.jl$ framework, we introduce the interface for defining Finite Horizon (PO)MDPs *FiniteHorizonPOMDPs.jl*. With this interface, the user can either define his Finite Horizon problem or use the *fixhorizon* utility to transform an existing Infinite Horizon (PO)MDP into a Finite Horizon one.

To present the idea of finite horizon solver, we implement the Finite Horizon Value Iteration in *FiniteHorizonValueIteration.jl* algorithm, which solves problems defined as Finite Horizon MDPs. Moreover, we implement the *FiniteHorizonValueIteration.jl* algorithm in an easy-to-grasp way, demonstrating its structure and benefits.

Furthermore, we implement the state-of-the-art Finite Horizon Point-Based Value Iteration algorithm in $FiVI.jl$. The Finite Horizon Point-Based Value Iteration presents a great heuristic solution for solving finite horizon POMDP problems. Furthermore, it is an excellent starting point for any future contributions, as the authors of the algorithm propose improving heuristics.

To benchmark the $FIVI$, we also implement the *PointBasedValueIteration.jl* algorithm on which foundation build many heuristics already implemented in *POMDP.jl* framework.

Besides mentioned solvers and interface, we contribute with additional POMDP models, policies, and other minor changes necessary to provide the Finite Horizon POMDP support.

Finally, we benchmark all implemented solvers. We benchmark the MDP solver on custom MDP problems and the POMDP solvers on well-known POMDP problems from the literature.

In our experiments, we show that we correctly implemented the solvers. The Point-Based Value Iteration solver asymptotically reaches the same result as the state-of-the-art SARSOP algorithm. On top of that, the finite horizon solvers show up to a tenfold improvement, increasing with the rising problem size. Concluding that the specialized finite horizon algorithms are essential in environments with a limited time span.

Part I

Background

Chapter 2

Fully Observable Markov Decision Processes

The Markov decision processes (MDPs) are a part of the optimization problem solvers. The term MDP was probably first used in (Bellman 1957) and came from the name of Russian mathematician Andrey Markov who researched the stochastic processes, which create the foundation of MDPs. We describe the MDP problems with the environment. The environment consists of fully observable states and actions (with a corresponding stochastic transition model) for which the agent either pays a cost or receives a reward. The agent is an entity that lives inside of an MDP environment, and its task is to maximize the reward obtained for its actions. With the agent entity, we can intuitively demonstrate the MDPs solution. The environment does not change over time. According to the problem's description, the problem can have either an infinite horizon or finite horizon, limiting the number of steps the agent can take.

This chapter will first draw a motivation on why we use MDPs in modeling environments in planning problems. Then define the background of MDPs, possible approaches to solving MDPs, and in the end, we are going to discuss the advantages of using time-dependent MDPs.

The definitions, algorithms, and the construction of the following sections draw from (M. Kolobov; A. Kolobov 2012).

2.1 MDP and its Features

The Fully Observable Markov Decision Processes (MDPs) are tuples *(S, A, T, R)* containing a set of environment states S, actions A that the agent can execute in corresponding states, stochastic transition function T modeling the relationships between origin states s, actions a and the distributions of the destination states s'. Finally, the MDPs also contain the reward function R that assigns reward (or penalty) to tuples *(s, a, s')* - for transitions from s to s' when executing the action a More formally:

Definition 2.1 (Infinite Horizon Fully Observable Markov Decision Process (MDP)). A fully observable MDP is a tuple (S, A, T, R), where:

- S is the finite set of all possible states of the environment, also called the state space;

- A is the finite set of all actions an agent can take;
- $T : S \times A \times S \to [0, 1]$ is a transition function mapping the probability $T(s, a, s')$ of reaching the state s' if action a is executed when the agent is in state s;
- $R : S \times A \times S \to R$ is a reward function that gives a finite numeric reward value $R(s, a, s')$ obtained when the system goes from state s to state s' as a result of executing action a.

However, the MDP agent does not have free will. The agent moves according to orders. The agent's orders come from a policy. The policy is a function mapping every state to some action. Based on the agent's state, the agent executes an action corresponding to a policy's mapping of the agent's current state. We evaluate the policy with a reward, which the agent receives for following the policy's mappings. In our context, the best quality is the maximal one.

Naturally, we want the agent to move in such a way that it maximizes its reward. Such a policy that is better than any other policy is called optimal policy. To select the optimal policy, we have to evaluate all possible policies.

In this work, we do not consider separate time steps and different possible histories that could lead to a given state s in a time t. The history is a sequence $h = (s_0, a_0, s_1, a_1, \ldots, s_n)$ of consequent states and actions that ends up in the current state s_n by following a policy π. By considering policies ending up in a state s with various histories as similar ones, we vastly reduce the number of possible policies to $|A|^{|S|}$. These relaxed policies are called Markovian. In the following text, we are using the Markovian Policies.

Definition 2.2 (Markovian Policy). A probabilistic (deterministic) history-dependent policy $\pi : H \times A \to [0, 1]\,(\pi : H \to A)$ is *Markovian* if for any two histories $h_{s,t}$ and $h'_{s,t}$, both of which end in the same state s at the same time step t, and for any action a, $\pi(h_{s,t}, a) = \pi(h'_{s,t}, a)$ $(\pi(h_{s,t}) = a$ if and only if $\pi(h_{s,t}) = a)$.

We evaluate a policy with a sum of rewards obtained when executing policy from the initial state until termination. Other than a whole policy, we can also evaluate a given state s. We evaluate a given state s using a value function. The value function $V^\pi(s)$ returns the expected reward the agent obtains when executing the actions given by a policy π from a state s that end up in a terminal state. The terminal state, or goal state, stands for a sink state that the agent can not leave. The value function corresponds to $V : S \to [-\infty, \infty]$. In the following text, we will refer to the value function as $V(s, t)$ or $V(s)$.

Definition 2.3 (The Value Function of a Policy). Let $h_{s,t}$ be a sequence that terminates at state s and time t. Let $R_{t'}^{\pi_{h_{s,t}}}$ be random variables for the amount of reward obtained in an MDP as a result of executing policy π starting in state s for all time steps t' s.t. $t \leqslant t' \leqslant t_{max}$ if the MDP ended up in state s at time t via sequence $h_{s,t}$.The value function $V^\pi : H \to [-\infty, \infty]$ *of a sequence-dependent policy* π is a utility function u of the reward sequence $R_t^{\pi_{h_{s,t}}}, R_{t+1}^{\pi_{h_{s,t}}}, \ldots$ that one can accumulate by executing π at time steps $t, t+1, \ldots$ after sequence $h_{s,t}$. Mathematically, $V^\pi(h_{s,t}) = u(R_t^{\pi_{h_{s,t}}}, R_{t+1}^{\pi_{h_{s,t}}}, \ldots)$.

Among the policies evaluated with the value function V^π from an initial state, we will be able to find an optimal MDP solution (or optimal policy) denoted as π^*. The optimal policy's value V^* is called the optimal value function. Such optimal policy then satisfies $V^*(h) \geqslant V^\pi(h)$ for all sequences $h \in H$.

One possible approach to evaluating the value function is using the expected sum of rewards from executing actions given by the policy, called the Expected Linear Additive Utility.

Definition 2.4 (**Expected Linear Additive Utility**). An *expected linear additive utility* function is a function $u(R_t, R_{t+1}, \ldots) = E[\sum_{t'=t}^{|D|} \gamma^{t'-t} R_{t'}] = E[\sum_{t'=0}^{|D|-t} \gamma^{t'} R_{t'+t}]$ that computes the utility of a reward sequence as the expected sum of (possibly discounted) rewards in this sequence, where $\gamma \geqslant 0$ is the discount factor.

This approach eliminates the possibility of multiple different solution values resulting from multiple stochastic runs of the same policy. It employs the expected value and does not need to perform vector equality comparison as its results are scalars.

It turns out that this approach is even better as it guarantees a fundamental property of MDPs called **The Optimality Principle**, according to which *among the policies that we evaluated by the expected linear additive utility, there exists a policy that is optimal at every time step.*

2.2 MDPs Techniques

This part will briefly introduce a few fundamental algorithms for MDPs solving: Brute-Force algorithm, Policy Iteration, Value Iteration, and in the end, we present a possible solution for its disadvantages: Prioritizations.
After reading this section, the reader should know the options for solving MDPs and be ready for the following section to discuss the finite horizon's advantages.

2.2.1 Brute-Force Algorithm

As its name suggests, the Brute-Force is the most naive algorithm. The reader can encounter its modifications in all problem-solving classes of computer science. During the execution of the solver, the method evaluates all possible output combinations and chooses the best one. As a result of significant performance requirements, the algorithm is rarely ever used. In MDPs, the number of policies evaluated is $|A|^{|S|}$.

However, as the number of states or actions rises, such evaluation becomes enormous. Another problem is MDP action outcomes not being deterministic, and MDP structure often being cyclic. Both of these problems result in a steep complexity rise in terms of computational resources.

That is why another approach comes in.

2.2.2 An Iterative Approach to Policy Evaluation

The evaluation of cyclic environments requires a suitable equation that captures all transitions and rewards. That means that every state's value function should correspond to the sum of rewards for moving towards successor states. We call such formula as the Bellman equation, and we write the equation as follows:

$$V^{\pi}(s) = \sum_{s' \in S} T(s, \pi(s), s')[R(s, \pi(s), s') + V^{\pi}(s')] \tag{2.1}$$

For all states, the Bellman equation forms a system of linear equations. This system of linear equations reduces the complexity of the previous approach. It *can* be solved using Gaussian Elimination in $O(|S|^3)$. It is still inefficient, but we use the idea to find a solution with an iterative approach.

The iterative approach starts with a rough estimation and continuously works its way up to the asymptotically same solution as the linear system. The algorithm stops when the maximum gap between two consequent value functions of all states is lesser than a given value, called residual. This solution is optimal (M. Kolobov; A. Kolobov 2012). As it stores the previous iteration, it is a part of dynamic programming. Every iteration of this algorithm runs in $O(|S|^2)$.

Algorithm 1: Iterative Policy Evaluation

```
1  //Assumption: $\pi$ is proper (ends up in a goal state)
2  initialize $V_0^{\pi}$ arbitrarily for each state
3  $n \leftarrow 0$
4  repeat
5      $n \leftarrow n + 1$
6      foreach $s \in S$ do
7          compute $V_n^{\pi}(s) \leftarrow \sum_{s' \in S} T(s, \pi(s), s')[R(s, \pi(s), s') + V_{n-1}^{\pi}(s')]$
8          compute $residual_n(s) \leftarrow |V_n^{\pi}(s) - V_{n-1}^{\pi}(s)|$
9      end
10 until $max_{s \in S} residual_n(s) \leq \epsilon$;
11 return $V_n^{\pi}$
```

We know how to evaluate the environment's state given some policy, but we have not yet described how to choose the best policy for a given evaluation.

2.2.3 Policy Iteration

Given that the iterative policy evaluation is optimal after an undefined number of iterations for a specific policy, we can further improve it by iterating policy evaluation and policy improvement. For every iteration, we have *almost* optimal policy evaluation for the previous policy. We improve the policy by iterating over the state space, where for each state s we select the new policy for a given state as the action that maximizes the reward. The algorithm stops when the policy is the same in two consequent iterations. This algorithm is called Policy Iteration (Algorithm 2).

However, the policy iteration requires its initial policy to end in the goal state. If the policy ends up in the goal states, it converges significantly fast. On the other hand, if we do not meet the initial condition of policy ending in the terminal state, the policy evaluation step will diverge. Nevertheless, another algorithm can solve this drawback.

Algorithm 2: Policy Iteration

```
1  initialize $\pi_0$ to be an arbitrary policy ending in the goal state
2  $n \leftarrow 0$
3  repeat
4      $n \leftarrow n + 1$
5      Policy Evaluation: compute $V^{\pi_{n-1}}$
6      Policy Improvement:
7      foreach state $s \in S$ do
8          $\pi_n(s) \leftarrow \pi_{n-1}(s)$
9          $\forall a \in A$ compute $Q^{(V^{\pi_{n-1}})}(s, a)$
10         $V_n(s) \leftarrow max_{a \in A} Q^{(V^{\pi_{n-1}})}(s, a)$
11         if $Q^{(V^{\pi_{n-1}})}(s, \pi_{n-1}(s)) > V_n(s)$ then
12             $\pi_n(s) \leftarrow argmax_{a \in A} Q^{(V^{\pi_{n-1}})}(s, a)$
13         end
14     end
15 until $\pi_n == \pi_{n-1}$;
16 return $\pi_n$
```

2.2.4 Value Iteration

Value iteration focuses on improving the value function of all states instead of evaluating the policy. The algorithm creates a policy at its very end to maximize the reward in a given state. Thanks to this property, the initial policies are not used in the algorithm and can not lead to divergence.

The algorithm draws from Bellman equations, which mathematically express the optimal solution of an MDP.

$$\begin{aligned} V^*(s) &= max_{a \in A} Q^*(s, a) \\ Q^*(s, a) &= \sum_{s' \in S} T(s, a, s')[R(s, a, s') + V^*(s')] \end{aligned} \tag{2.2}$$

The equation's interpretation is maximizing the optimal value function of a given state. The result can be either zero if the state is among the goal states or a value that maximizes the result obtained after executing a given action and then following the optimal policy.

To approximate $V^*(s)$, the algorithm uses the so-called Bellman backup to improve the current $V_n(s)$ with $V_{n-1}(s')$.

$$V_n(s) \leftarrow max_{a \in A} \sum_{s' \in S} T(s, a, s')[R(s, a, s') + V_{n-1}(s')] \tag{2.3}$$

Value iteration is iteratively improving its value function estimation and is guaranteed to converge to the optimal solution (M. Kolobov; A. Kolobov 2012). Its stopping criterion is either the maximum residual or the number of iterations.

Algorithm 3: Value Iteration

```
1  initialize $V_0$ arbitrarily for each state
2  $n \leftarrow 0$
3  repeat
4      $n \leftarrow n + 1$
5      foreach $s \in S$ do
6          compute $V_n(s)$ using Bellman backup at $s$
7          compute $residual_n(s) = |V_n(s) - V_{n-1}(s)|$
8      end
9  until $max_{s \in S}\, residual_n\ (s) < \epsilon$;
10 return greedy policy: $\pi^{V_n}(s) = argmax_{a \in A} \sum_{s' \in S} T(s,a,s')[R(s,a,s') + V_n(s')]$
```

2.2.5 Prioritization

One of the most significant drawbacks of Value Iteration is that it requires full sweeps of the state space. This drawback results in many unnecessary value function evaluations because some of the state's values remain the same as the value function updates are heading from the goals state at first. It can take lots of iterations to evaluate all successor state's value functions. Furthermore, due to cyclic moves, the speed of convergence among states can differ.

Both of these problems often result in flawed performing algorithm execution.

Logical improvement is to, instead of iterating the whole state space, iterate over each state separately. Iterating over each separate state has two consequences. First, we have to develop or choose an algorithm that prioritizes states and, second, does not starve some of them (meaning every state gets updated accordingly).

Furthermore, iterating over the separate states means that we can no longer use the number of iterations as a termination condition because we do not know which states' value functions and how often they were updated. The only terminal condition remains maximum residual.

(M. Kolobov; A. Kolobov 2012) formalizes the intuition in Algorithm 4:

Algorithm 4: Prioritized Value Iteration

```
1  initialize $V$
2  initialize priority queue $q$
3  repeat
4      select state $s' = q.pop()$
5      compute V(s') using a Bellman backup at s'
6      foreach predecessor s of s', i.e. $\{s | \exists a[T(s,a,s') > 0]\}$ do
7          compute $priority(s)$
8          $q.push(s, priority(s))$
9      end
10 until $termination$;
11 return greedy policy $\pi^V$
```

We will introduce a few priority metrics whose features generally differ and may diverge under specific conditions. For details, head over to (M. Kolobov; A. Kolobov 2012).

Prioritized Sweeping

Prioritized sweeping estimates the expected change in the value of a state if a backup were to be performed on it now.

$$priority_{PS}(s) \leftarrow max\Big\{priority_{PS}(s), max_{a\in A}\{T(s,a,s')Res^V(s')\}\Big\} \tag{2.4}$$

Improved Prioritized Sweeping

Improved Prioritized Sweeping employs the idea that states with fast converging value functions should be the priority in terms of evaluating order. The consequence is that the algorithm first iterates the states near the goal and moves to other states only after the change between state iterations becomes small.

$$priority_{IPS}(s) \leftarrow \frac{Res^V(s)}{V(s)} \tag{2.5}$$

Focused Dynamic Programming

Focused Dynamic Programming is a particular case of prioritization techniques used when the start state s_0 is known. In such cases, the start case's knowledge can be employed and added as a penalty factor.

$$priority_{FDP} \leftarrow h(s) + V(s) \tag{2.6}$$

where:

h(s) *is lower bound on the expected reward for reaching s from* s_0, and

V(s) is regular value function for given state s.

In the previous sections, we have very briefly introduced the notion of MDPs and their solving methods. First, we showed the Brute-Force algorithm, which naively evaluated all possible policies. Then we discussed the value and policy iterations that improved the performance but still did lots of useless evaluations. And finally, we presented a solution that intelligently iterates through prioritized states.

2.3 Finite-Horizon MDPs

With all the necessary background layed out, let us present the Finite-Horizon MDPs.

Definition 2.5. Finite-Horizon MDP A finite-horizon MDP is an MDP as described in Definition 2.1 with a finite number of time steps.

The Infinite-Horizon MDPs discussed so far are a slightly different class of problems from Finite-Horizon MDPs. However, the solution of Finite-Horizon is somewhat similar to the prioritized value iteration of Infinite-Horizon MDPs. Moreover, we can transform each Infinite-Horizon MDP Finite-Horizon one.

The ability to transform the infinite horizon problem into finite horizon one is critical because the Finite-Horizon MDP has an acyclic state space, and the acyclic MDPs can be solved optimally using only one backup if used optimal backup order. (M. Kolobov; A. Kolobov 2012)

On the other hand, the transformation to Finite-Horizon MDP increases the state space size by copying a whole former MDP into a single stage of Finite Horizon MDP. This way, the memory consumption linearly increases with the number of stages.

Furthermore, the transformation to Finite Horizon MDP does not necessarily mean that the result will be the same or even optimal in an infinite horizon environment. With a finite horizon, the finite horizon agents focus on getting rewards obtainable in a given time span. The Infinite Horizon MDPs do not have such a constraint and can focus on getting a bigger reward later on.

As the finite horizon number is known, we can evaluate all states' value functions from maximum horizon time and work our way down to starting time. This way, we evaluate each state's successor's value functions, employing optimal backup order and finding the optimal policy on the first pass.

In conclusion, in cases where the infinite horizon methods are not sufficient, the MDPs can be transformed into Finite-Horizon ones. However, while solving the MDP in one pass, this transformation blows up the state space.

Chapter 3

Partially Observable Markov Decision Processes

Up until now, we have dealt with fully observable MDPs. In MDPs, the agent has the perfect knowledge about its state. In practice, however, fully observable environments are rare. Real-world agents often have only limited knowledge about their surroundings, making their state uncertain. These environment models are called partially observable MDPs (or POMDPs). With imperfect state information, the algorithms for MDPs are no longer valid, and we have to define new methods that deal with probability distributions instead of deterministic states.

In the following chapter, we will mainly draw from (Russell; Norvig 2010), but also from (Shani; Pineau; Kaplow 2013), (Pineau; Gordon; Thrun 2003) and (Walraven; Spaan 2019). We note that we cite the most important takeaways.

3.1 POMDP Models

The POMDPs (Definition 3.1) draw from MDPs, and thus, some of the model methods remain the same. POMDPs, similar to MDPs, define the transition model T, states S, actions A and reward model R. Besides that, the POMDPs also define a new model, which describes the possibility of receiving observations of the agent's surroundings, called the observation function O and set of possible observations Ω. Unlike MDPs, the POMDPs define the starting distribution b_0 as the distribution over states instead of a deterministic state.

The initial state distribution b_0 as well as all other possible state space distributions are called belief states. The belief states are $|S|$ long vectors describing the probability distribution over all states in POMDPs. We denote the probability of each state s with $b(s)$. Moreover, the space of belief states is continuous.

To record the effect of executing an action a and receiving an observation o, we have to execute a so-called belief update. For a given action a and observation o, the belief update is calculated as the conditional distribution. It stores only the last belief, but it is calculated from a full history of actions executed and observations received so far starting from the initial belief state. Every time an agent executes the action and obtains an observation,

Definition 3.1 (**Partially Observable Markov Decision Process (POMDP)**). POMDPs (Shani; Pineau; Kaplow 2013) are formally defined as a tuple $\{S, A, T, R, \Omega, O, b_0\}$, where:

- *S, A, T, R* are the same as for fully observable MDPs, often called the *underlying* MDP of the POMDP.

- Ω is a set of possible observations. For example, in the robot navigation problem(Section 6.3 or 6.4), Ω may consist of all possible wall occurrences in an POMDP.

- O is an observation function, where $O(a, s', o) = P(o|s', a)$ is the probability of observing observation o given that the agent has executed action a, reaching state s'. O can model robotic sensor noise, or the stochastic appearance of symptoms given a disease.

- b_0 is an initial state distribution.

the belief is updated. We formulate the update of each state as:

$$b_o^a(s') = \frac{O(a, s', o)}{O(b, a, o)} \sum_{s \in S} T(s, a, s') b(s), \tag{3.1}$$

where $O(b, a, o)$ corresponds to the probability of obtaining observation o after executing action a in belief b. This probability is calculated as follows:

$$O(b, a, o) = \sum_{s' \in S} O(a, s', o) \sum_{s \in S} T(s, a, s') b(s). \tag{3.2}$$

This approach to calculating the normalizing constant is computationally more demanding than other approaches. However, it allows skipping the computation of the inner sum in cases where the probability is zero.

3.2 Value Functions for POMDPs

In MDPs, the value function computes the value for each discrete state. With POMDPs having a continuous space of belief vectors, we need to alter the model method. Instead of computing a value for a single state, we compute utility vectors of length $|S|$, called **α-vectors**, for belief states. The value of belief state then becomes $\sum_{s \in S} b(s)\alpha_\pi(s)$, where $\alpha_\pi(s)$ is the utility of executing a policy π in state s. We can interpret this linear combination as a hyperplane over the belief space (Pineau; Gordon; Thrun 2003). The optimal policy in a given belief then becomes an action a, whose α-vector maximizes the dot product with a given belief:

$$V(b) = V^{\pi^*}(b) = \max_\pi b \cdot \alpha_\pi \tag{3.3}$$

More importantly, the set of α-vectors creates a piecewise linear, and convex function (Russell; Norvig 2010), Figure 3.1.

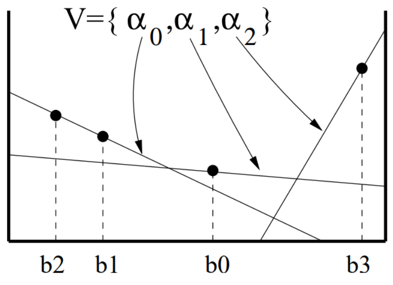

Figure 3.1: POMDP value function representation using PBVI (Pineau; Gordon; Thrun 2003)

In general, the value function can be iteratively updated with a value iteration algorithm. The value function update can be formulated using the Bellman update as follows:

$$V(b) = \max_{a \in A}[R(b,a) + \gamma \sum_{b' \in B} T(b,a,b')V(b')]. \tag{3.4}$$

In the formula above, we use $R(b,a) = \sum_{s \in S} b(s)R(s,a)$ and $T(b,a,b') = \sum_{o \in \Omega} P(o|b,a)\mathbb{1}(b' = b_o^a)$ for simplicity. The first formula stands for rewards obtained for executing an action a from a belief b calculated as a dot product of belief vector and vector of rewards for executing action a in each state. The second formula is the probability of transitioning from a belief state b to a new belief state b' given an action a. It computes the sum of the probabilities of observations o resulting in belief b'.

The Eq. 3.4 can be also written in terms of vector operations and operations on sets of vectors (Shani; Pineau; Kaplow 2013), as (note that r_a denotes the vector of rewards obtained after executing action a in each state s):

$$V' = \bigcup_{a \in A} V^a \tag{3.5}$$

$$V^a = \bigoplus_{o \in \Omega} V^{a,o} \tag{3.6}$$

$$V^{a,o} = \left\{ \frac{1}{|\Omega|} r_a + \alpha^{a,o} : \alpha \in V \right\} \tag{3.7}$$

$$\alpha^{a,o}(s) = \sum_{s' \in S} O(a,s',o)T(s,a,s')\alpha(s'), \tag{3.8}$$

In each iteration of the exact value iteration algorithm, the value function is updated across the entire belief space. For each possible action, observation and α-vector of old value function, the Eq. 3.8 computes new α-vector, costing $O(|V| \times |A| \times |\Omega| \times |S^2|)$ operations. The alpha vector is then summed with the reward corresponding to a given action in Eq. 3.7, cross-summed across the observation space in 3.6 and unioned across the action space in 3.5, adding another $O(|A| \times |S| \times |V|^{|\Omega|})$ to the resulting complexity of a single iteration which is $O(|V| \times |A| \times |\Omega| \times |S^2| + |A| \times |S| \times |V|^{|\Omega|})$.

As the POMDPs' belief space is continuous, the algorithm's performance rapidly decreases with the growing size of the problems.

3.3 Point-Based Value Iteration

Most of the POMDP problem simulations are unlikely to reach most of the points in the belief space. To reduce the complexity of solving POMDPs, approximation algorithms are at hand. In the past, researchers introduced several approximation algorithms ((Lovejoy 1991) suggests creating a grid-based belief set, (Milos Hauskrecht 2000) suggests creating a set of reachable beliefs). These algorithms, however, rely on naive approximations and underperform as a result. For example, creating grid-based belief sets turned out to be inaccurate in sparse grids or computationally expensive in the case of dense grids.

Adopting yet another strategy can overcome the problems described above. Given the most probable belief states, we can focus the solver on finding only their corresponding α-vectors and thus reduce the computation overhead. This algorithm is called Point-Based Value Iteration (PBVI).

The PBVI (Pineau; Gordon; Thrun 2003) is an anytime algorithm. The Algorithm 5 starts from an initial belief b_0 and iteratively improves its value function and expands its belief space. The belief space $B = \{b_0, b_1, \dots, b_m\}$ stores the the most probable belief vectors. The value function updates each belief vector, but unlike other approximation algorithms, such as grid-based VI, its value function spans over the whole belief space instead of only one belief vector. The algorithm ends when the two consecutive value function sets equal to each other or after a predefined number of iterations.

Algorithm 5: Generic PBVI

```
1 $B \leftarrow b_0$
2 while Stopping criterion not reached do
3     $Improve(V, B)$
4     $B \leftarrow Expand(B)$
5 end
```

Thanks to being an anytime algorithm, the solver can be stopped whenever needed, effectively exchanging computation time and solution quality. It is also possible to choose how big the maximum gap between iterations of value updates will be.

3.3.1 Improve Function

The value function improvement, or backup, is an adjustment of the value update from Eqs. 3.5 - 3.8 It maintains only one α-vector per belief vector, It also chooses only the best α, pruning the dominated vectors twice, at each *argmax* expression, and reducing the complexity of algorithm. The backup can be compactly written as:

$$backup(V, b) = \underset{\alpha_a^b : a \in A, \alpha \in V}{\operatorname{argmax}} \; b \cdot \alpha_a^b \tag{3.9}$$

$$\alpha_a^b = r_a + \gamma \sum_{o \in \Omega} \underset{\alpha^{a,o} : \alpha \in V}{\operatorname{argmax}} \, b \cdot \alpha^{a,o}, \tag{3.10}$$

where r_a represents vectors of rewards for executing a given action a in each state, and

$\alpha^{a,o}$ has the same meaning as in the Equation 3.8.

Algorithm 6: PBVI Improve

```
1 repeat
2     foreach $b \in B$ do
3         $\alpha \leftarrow backup(b, V)$ /* execute a backup operation on all points in
              B in arbitrary order                                        */
4         $V \leftarrow V \bigcup \{\alpha\}$
5     end
6 until $V$ has converged;
  /* repeat the above until V stops improving for all points in B   */
```

The improve function starts with the α-vector update same as in the exact value update Eq. 3.8 with the complexity of $O(|S|^2 \times |A| \times |V| \times |\Omega|)$. The summation and dot product in Eq. 3.9 then takes further $O(|\Omega| \times |S|)$ and $O(|S)|)$ for adding a reward vector. With another $O(|S|)$ operations for the dot product in 3.9 the complexity of full point-based backup requires $O(|S|^2 \times |A| \times |V| \times |\Omega| + |A| \times |S| \times |\Omega|)$.

The PBVI algorithm, unlikely the original value update, does not need to cross-sum the α-vectors. Furthermore, the $\alpha^{a,o}$-vectors can be cached, because they are independent of the current belief b. Thus, computing the backup of whole $|B|$, with $\alpha^{a,o}$ cached requires only $O(|A| \times |\Omega| \times |V| \times |S|^2 + |B| \times |A| \times |S| \times |\Omega|)$, instead of $O(|V| \times |A| \times |\Omega| \times |S^2| + |A| \times |S| \times |V|^{|\Omega|})$ in case of exact backup (Eqs. 3.5 - 3.8).

The backup runs until the convergence of α-vector pairs or until a predefined number of iterations.

3.3.2 Expand Function

After the value function improvement, the algorithm executes the belief space expansion. At this point, the goal is to reduce the error bound as much as possible. The error-bound reduction is performed by greedily expanding the belief set with a new furthest belief accessible from each stored belief. The distance function is defined as follows, with L being the chosen distance metric:

$$|b' - B|_L = \min_{b \in B} |b - b'|_L, \tag{3.11}$$

and the expanded belief results from :

$$b' = \max_{a,o} |b^{a,o} - B|_L \tag{3.12}$$

The choice of the distance metric is not crucial as the results appear to be identical. The authors of the algorithm recommend adding one new belief per 1 old (Pineau; Gordon;

Thrun 2003).

Algorithm 7: PBVI Expand

```
1 $B_{new} \leftarrow B$ foreach $b \in B$ do
2     $Successors(b) \leftarrow \{b^a_o | O(b, a, o) > 0\}$
3     $B_{new} \leftarrow B_{new} \bigcup \text{argmax}_{b' \in Successors(b)} \|B, b'\|_L$ /* add the furthest
          successor of b                                                    */
4 end
```

3.4 Overview of Alternative Point-Based Algorithms

The generic PBVI, while being simple and somewhat clever with its belief set expansion, becomes cumbersome when solving large POMDP instances. However, we often have to expand to large belief sets. Furthermore, we want to keep the possibility to compute a compact value function. To cope with these circumstances, the researchers developed various alternatives and heuristics.

3.4.1 Perseus

The idea of Perseus (Spaan; Vlassis 2005) comes from the Achilles heel of PBVI. The PBVI is cleverly expanding its belief space with the most probable belief states. However, such an approach is significantly demanding. On the other hand, the Perseus algorithm shows that even running random trials and exploring a large number of belief vectors may be more efficient if used with clever value updates.

3.4.2 HSVI

The Perseus algorithm randomization is well suited for small and mid-sized problems but fails in larger ones. The intractability is caused by storing an enormous number of belief vectors. The HSVI (Smith; Simmons 2012) algorithm solves this issue by employing a straightforward yet effective heuristic that helps cut down the gap between the upper and lower bounds on the optimal value function. The HSVI stores the belief vector visits and backs them up in the reversed order.

3.4.3 FSVI

The FSVI (Shani; Brafman; Shimony 2007) uses another heuristic creating new trajectories by using the best action in the underlying MDP. Such heuristics focus on searching high reward policies but tend to fail when obtaining additional information is crucial for getting the optimal result.

3.4.4 SARSOP

SARSOP (Kurniawati; Hsu; Lee 2008), unlike the others, focuses on calculating an optimally reachable belief space. It focuses on exploring areas that an agent will visit under the optimal policy and ignores areas which the agent will not visit under the optimal policy. The SARSOP obtains the area reachable under the optimal policy heuristically.

Chapter 4

Finite Horizon POMDPs

Unlike the Finite Horizon MDPs, the Finite Horizon POMDPs do not offer a one-pass optimal solution. Furthermore, Finite Horizon POMDPs cause additional memory and performance requirements. Causing additional memory and performance requirements does not mean that it is necessarily a flawed concept. In the following sections, we will explain why the Finite Horizon POMDPs are actually beneficial, why we need to use appropriate methods or develop new ones designed specifically for finite horizon problems and describe the state-of-the-art algorithm for solving Finite Horizon POMDPs. In the end, we will introduce possible heuristics that the algorithm can employ.

In this chapter, we will mainly draw from (Walraven; Spaan 2019).

4.1 Finite Horizon Problems

The real-world problems span from non-stop operational agents to the ones that are time-limited. Say, an elevator focusing on short-term rewards from personnel transport in the former case and the electric vehicle charging provider focusing on day-to-day forecasts in the latter case.

The Finite Horizon POMDPs are defined in line with Definition 3.1, but their time-span is limited by the horizon h or t_{max}. We call each specific time moment a *stage*. Distinguishing between stages opens up the possibility of different state spaces in various stages. Note that the transition is defined as a mapping from state s in stage t and action a to state s' in stage $t+1$. The observation function Ω, observation set O, and reward function R are typically extended with the stage information as well.

In this chapter, we consider the discount $\gamma = 1$, as we focus on problems, which do not include the discount factor. In such cases, we want to obtain the policy that maximizes the expected sum of rewards:

$$E\left[\sum_{t=1}^{h} r_t\right], \tag{4.1}$$

where $t = 1$ is the first time step and h stands for the horizon number, meaning there are h steps total. In finite horizon problems, we store different value functions $V^{\pi}(t, b)$ and policies $\pi(t, b)$ for each time step, mapping both time and state to the optimal action and its value. The value function computes the expected sum of rewards the agent obtains when following the policy from belief b and stage t to stage h. We define the formula as follows:

$$V^{\pi^*}(t, b) = \begin{cases} \max_{a \in A} \left[\sum_{s \in S} R(s, a) b(s) + \sum_{o \in O} O(b, a, o) V^{\pi^*}(t + 1, b_o^a) \right] & t \leq h \\ 0 & t > h \end{cases} \tag{4.2}$$

where b_o^a is the belief updated with observation o obtained after executing action a defined in Eq. 3.1.

The optimal policy π^* corresponding to the optimal value function is defined as:

$$\pi^*(t, b) = \mathrm{argmax}_{a \in A} \left[\sum_{s \in S} R(s, a) b(s) + \sum_{o \in O} O(b, a, o) V^{\pi^*}(t + 1, b_o^a) \right]. \tag{4.3}$$

For time steps from *1* to h the policy returns optimal action corresponding to optimal value-function in time t and belief b.

4.2 Limitations of POMDP Algorithms in Finite Horizon Settings

The existing POMDP solvers work with discounting reward, but it turns out that they can not easily generalize to the finite horizon setting without discounting. This section will discuss the approaches that could be leveraged in Finite Horizon POMDP solving and why the existing solvers can not apply the discount factor $\gamma = 1$.

In the first part, we discuss possible high-level approaches from both MDP and POMDP solvers and why we can not use them. In the second part, we will review a few Infinite Horizon POMDP solvers and discuss why they are not the correct choice for solving Finite Horizon problems.

4.2.1 Solution Strategies for Finite Horizon Problems

The first approach solves the POMDP as if it is a fully observable MDP. This model change is possible thanks to the finite number of reachable beliefs of POMDP. Thanks to that, we can treat each belief as an MDP state and solve it with MDP solvers. This approach, however, creates up to $(|A||O|)^h$ beliefs and often becomes intractable.

The second approach assumes solving the Finite Horizon POMDP as if it is an infinite horizon one. In this approach, we can safely assume discounting. However, because of wrongly assuming the infinite horizon, the policy does not have the information that its execution is terminated after a finite number of steps, meaning that it could focus on obtaining the unreachable reward obtainable only after the maximum number of steps. The second disadvantage is that the policy focuses on early reward because of the discounting, even if there is a larger reward later on. The agent, however, does not obtain the larger reward because its value is negatively decreased by discounting and, as such, becomes lower than the early small reward.

The third approach supposes that the Finite Horizon POMDP states are reachable in all stages with a terminal state after the horizon stage, resulting in $|S| \times h + 1$ states. This approach is inefficient as it increases the number of beliefs and the other solution representation as α-vectors, resulting in an undesired increase in both performance and memory requirements.

4.2.2 Discarding the Discount Factor in Infinite Horizon Algorithms

In the text above, we discussed possible approaches to solving Finite Horizon POMDPs without discounting and why they are often intractable or have other undesirable effects. In the following text, we will describe why the solvers introduced in the section on Infinite Horizon POMDPs can not be used in the Finite Horizon setting with discount factor $\gamma = 1$. Besides, we discuss their optimality and whether they compute bounds on their results.

The optimality of the result refers to the policy starting from a known initial belief. The results may be suboptimal if starting from the other than the initial belief. We will start from the basic solvers and work our way to the most advanced heuristics presented in this work.

Exact Value Iteration can be used with the discount factor $\gamma = 1$. It computes the optimal solution for any initial belief. However, it is intractable for problems with large state spaces.

Point-Based Value Iteration does not suffer from computing the value function for the whole belief space. It incrementally expands its belief space, and it can evaluate its methods without using the discounting factor. Unfortunately, the worst-case error bound assumes a discount $\gamma < 1$, which opposes our requirements. The number of beliefs in PBVI can still become large, and the improved algorithms based on PBVI show better benchmark results.

Perseus algorithm also requires the lower bound to be initialized with $\gamma < 1$. It does not keep track of the upper bound. It is randomized and, in particular, does not provide any guarantee on performance and optimality.

Both SARSOP and HSVI incrementally expand their set of belief points, and their backup and upper bound update are well-defined for $\gamma = 1$. They produce the upper bound on the optimal value function and thus guarantee the optimality in the limit. However, their lower bound initialization, as in the preceding algorithm, requires a discount $\gamma < 1$. Thus, to employ both algorithms in the finite horizon environment, we need to adapt them.

All algorithms presented in the previous section as infinite horizon solvers lack at least one of the three declared properties - the possibility to run with discount factor $\gamma = 1$, the optimality of the result, or the missing computation of their result's bounds. The algorithm we will present in the next section meets all the required properties. It also draws on the well-performing properties of the Finite Horizon POMDP solvers. Furthermore, the algorithm converges to optimality, computes both bounds, and does not require problems to contain discounting.

4.3 FiVI: Finite Horizon Point-Based Value Iteration

This section describes FiVI (Walraven; Spaan 2019), a point-based value iteration algorithm for solving finite horizon POMDPs. *The algorithm unifies techniques and concepts from existing state-of-the-art point-based value iteration algorithms, and it provides attractive convergence characteristics and optimality guarantees.* (Walraven; Spaan 2019)

On the following pages, we will consecutively describe the functions essential for the correct execution of FiVI - the value function, backup, and the bound update. With the knowledge of value function update, backup, and the bounds update, we describe the high-level FiVI overview. After that, we cover the belief space expansion, convergence of the algorithm, and possible heuristics.

4.3.1 Time-Dependent Value Functions and Backups

The standard way of computing value functions in point-based value iteration algorithms is to run a value function update similar to the one defined by Eqs. 3.9, 3.10 and 3.8. In FiVI, we have to extend the value update with the non-stationary staged environment elements. The algorithm utilizes staged value functions V_t ranging from stage 1 to the horizon stage. Each value function V_t is represented by a set of α-vectors Γ_t.

The vectors constituting to a value function are defined by a time-dependent backup function:

$$\Gamma_t = \bigcup_{(b,\bar{v}) \in B_t} \mathit{backup}(b,t) \tag{4.4}$$

The FiVI needs to keep track of beliefs' upper bounds $\bar{v}$ to converge. The $\mathit{backup}(b,t)$ operator exploits the knowledge of next step value function Γ_{t+1} to compute α-vectors of Γ_t. The FiVI defines the time-dependent $\mathit{backup}(b,t)$ operator as following:

$$\mathit{backup}(b,t) = \operatorname*{argmax}_{\{z_{b,a,t}\}_{a \in A}} b \cdot z_{b,a,t}, \tag{4.5}$$

where

$$z_{b,a,t} = \begin{cases} r_a + \sum_{o \in \Omega} \operatorname*{argmax}_{\{z_{a,o}^{k,t+1}\}_k} b \cdot z_{a,o}^{k,t+1} & t < h \\ r_a & t = h \end{cases}, \tag{4.6}$$

and $z_{a,o}^{k,t}$ denotes the back-projection of vector $\alpha^{k,t} \in \Gamma_t$:

$$z_{a,o}^{k,t}(s) = \sum_{s' \in S} O(a,s',o) T(s,a,s') \alpha^{k,t}(s') \; \forall s. \tag{4.7}$$

The vector r_a represents the immediate reward vector for executing given action a from all states. We also assume that the backup has access to all vector sets and the reward vectors r_a.

4.3.2 Time-Dependent Value Upper Bounds and Bound Updates

Computing the upper bound of belief, typical for Point-based Value Iteration algorithms, enables the solution quality assessment. The FiVI algorithm stores the upper bound value as $\bar{v}$ in the belief-bound pairs $(b, \bar{v})$ kept in belief sets B_t. Generally, we compute the upper bound values $\bar{v}$ by employing linear programming interpolation. However, this approach is computationally expensive, and the researchers introduced the sawtooth approximation in (M. Hauskrecht 2000).

Algorithm 8: Sawtooth approximation (UB)

```
  Input  : belief $b'$, set $B$ containing belief-bound pairs
  Output: upper bound corresponding to belief $b'$
1 for $(b, \bar{v}) \in B \setminus \{(e_s, \cdot) \mid s \in S$ do
2     $f(b) \leftarrow \bar{v} - \sum_{s \in S} b(s)B(e_s)$
3     $c(b) \leftarrow \min_{s \in S} b'(s) \;/\; b(s)$
4 end
5 $b^* \leftarrow \text{argmin}_{\{b|(b,\bar{v} \in B \setminus \{e_s \mid s \in S\}\}} c(b)f(b)$
6 return $c(b^*)f(b^*) + \sum_{s \in S} b'(s)B(e_s)$
```

The sawtooth algorithm simplifies the upper bound update by imposing the weights c_b only to so-called corner beliefs. The corner belief marked $e(s)$ stands for the belief that has a deterministic distribution over a given state s ($b(s) = 1.$). The Algorithm 8 takes a belief b' and a belief set B and returns the upper bound interpolation for b'. The $B(e_s)$ notation denotes the upper bound of belief $e(s)$ in the belief set B.

In the finite horizon settings we update the upper bound value of the belief b in time $t < h$ as follows:

$$\bar{v} = \max_{a \in A} \sum_{s \in S} R(s,a)b(s) + \sum_{o \in I} O(b,a,o)UB(b_o^a, B_{t+1}), \tag{4.8}$$

where we interpolate based on the belief set B_{t+1} from the next stage. In the horizon stage we compute only $\max_{a \in A} \sum_{s \in S} R(s,a)b(s)$ as the upper bound value of the sink state is zero.

4.3.3 Algorithm Description of FiVI

With the two most essential algorithms described, we can overview the complete algorithm. The FiVI algorithm is an iterative algorithm that takes the POMDP, required precision, and time limit as the inputs and outputs the vector of α-vector sets Γ_t and the upper bound of the initial belief b_0.

At the very beginning, the algorithm initiates its model representations of belief sets B, α-vectors Γ, the reward vector of each action r_a, the variables storing the number of iterations δ and time elapsed τ' (lines 1 - 5).

Following lines 6 - 34, execute the body of the algorithm. The do-while loop consists of 3 main parts - belief expansion, belief backup, and belief upper bound update.

Algorithm 9: Finite Horizon Point-Based Value Iteration (FiVI)

```
Input  : POMDP M, precision $\rho$, time limit in seconds$\tau$
Output: sets $\Gamma_t$ for each time step $t$, upper bound $v_u$
1  $\Gamma_t \leftarrow \emptyset \; \forall t$
2  $B_t \leftarrow \emptyset \; \forall t$
3  $r_a \leftarrow (R(s_1, a), R(s_2, a), \ldots, R(s_{|S|}, a)) \; \forall a$
4  add corner beliefs to $B_t$ with upper bound $\infty$ ($\forall t$)
5  $\tau' \leftarrow 0, \delta \leftarrow 0$
6  do
7      $\delta \leftarrow \delta + 1$
8      expand(M, $\{\Gamma_1, \ldots, \Gamma_h\}, \{B_1, \ldots, B_h\}$, r)
9      for $t$ = $h$, $h$ - $1$, ..., $1$ do
10         $\Gamma_t \leftarrow \emptyset$
11         for $(b, \bar{v}) \in B_t$ do
12             $\alpha \leftarrow backup(b, t)$
13             $\Gamma_t \leftarrow \Gamma_t \cup \{\alpha\}$
14         end
15         for $(b, \bar{v}) \in B_t$ do
16             $\bar{v} \leftarrow -\infty$
17             for $a \in A$ do
18                 $v \leftarrow r_a \cdot b$
19                 if $t < h$ then
20                     for $o \in O$ do
21                         if $O(b, a, o) > 0$ then
22                             $v \leftarrow v + O(b, a, o) \cdot$ UB$(b^a_o, B_{t+1})$
23                         end
24                     end
25                 end
26                 $\bar{v} \leftarrow \max(\bar{v}, v)$
27             end
28         end
29     end
30     $v_l \leftarrow \max_{\alpha \in \Gamma_1} \alpha \cdot b_0$
31     $v_u \leftarrow$ upper bound $\bar{v}$ associated with $(b_0, \bar{v}) \in B_1$
32     $g_a \leftarrow 10^{\lceil \log_{10}(\max(|v_l|, |v_u|)) \rceil - \rho}$
33     $\tau' \leftarrow$ elapsed time after the start of the algorithm
34 while $\tau' < \tau \wedge v_u - v_l > g_a$;
35 return $(\{\Gamma_1, \ldots, \Gamma_h\}, v_u)$
```

The belief expansion on line 8 finds new beliefs B_t to shrink the gap between the upper and the lower bound.

The lines 9 - 29 are iterating over all non-terminal stages backwards, computing belief backup $\Gamma_t \leftarrow \bigcup_{(b,\bar{v})\in B_t} backup(b,t)$ on lines 11 - 14 and belief upper bound value on lines 15 - 27 of stage t with the model representations of stage $t+1$.

An iteration ends with a check of termination conditions - the maximum gap g_a and minimal time elapsed τ. The difference between the upper bound v_u of initial belief b_0 and the lower bound v_u denotes the current gap. The algorithm stops if the gap is at most one unit at the ρ-th significant digit or if a time limit τ has been exceeded.

The FiVI algorithm returns the vector of α-vector sets $\{\Gamma_1, \ldots, \Gamma_h\}$ denoting the lower bound and the upper bound v_u of the initial belief b_0. The gap implicitly defines a guarantee on the quality of computed solution (Walraven; Spaan 2019).

4.3.4 Belief Points and Convergence of the Algorithm

The optimality and speed of convergence of FiVI and Point-Based algorithms heavily depends on the quality of expanded belief backup executions. The algorithm can take one of two directions. Either randomly select new beliefs or steer the new beliefs into space, where the optimal policy would go as well. However, we do not know the optimal policy beforehand.

The FiVI algorithm uses a heuristic procedure similar to HSVI (Smith; Simmons 2012), or SARSOP (Kurniawati; Hsu; Lee 2008) to explore the promising areas.

The algorithm chooses the action and observation that lowers the gap as much as possible, effectively reducing the overall gap. We will motivate this by the term $U(t,b,a) - V(t,b,a)$ which shows the difference between the potential upper bound $U(t,b,a)$ of the belief b in the stage t after executing action a and its expected value function $V(t,b,a)$. The value function V is, by definition, maximizing its value and the upper bound U minimizing, reducing the resulting gap, and improving the quality of the result.

The expected value function for belief b in stage t after executing the action a is:

$$V(t,b,a) = \sum_{s\in S} R(s,a)b(s) + \sum_{o\in O} O(b,a,o)V(t+1, b_o^a), \tag{4.9}$$

and the potential upper bound:

$$U(t,b,a) = \sum_{s\in S} R(s,a)b(s) + \sum_{o\in O} O(b,a,o)UB(b_o^a, B_{t+1}). \tag{4.10}$$

After choosing the best action a, the algorithm chooses the observation leading to a belief with a maximum gap in stage $t+1$:

$$o = \text{argmax}_{\{o\in O \ \mid \ O(b,a,o)>0\}}\{UB(b_o^a, B_{t+1}) - \max_{\alpha\in\Gamma_{t+1}} \alpha \cdot b_o^a\}. \tag{4.11}$$

All expanded beliefs with stage $t > 1$ contribute to the final bounds and the gap associated with the initial belief b_0.

The expand algorithm (Algorithm 10) expands at most one belief for each stage t. It adds the resulting belief to the next stage belief set B_{t+1} as the new belief comes from the stage t belief b. The procedure follows the rules outlined in the lines above, starting from the initial belief. We do not have to consider the stage $t = h$ as we are not interested in the terminal beliefs.

Algorithm 10: Belief expansion (expand)

```
  Input: M, $\{\Gamma_1, \dots, \Gamma_h\}, \{B_1, \dots, B_h\}$, r
1 $b \leftarrow b_0$
2 for $t$ = 1, ..., $h$ - 1 do
3     $a \leftarrow \text{argmax}_{a \in A}\{r_a \cdot b + \sum_{\{o \in O|\ O(b,a,o)>0\}} O(b,a,o)\cdot \text{ UB}(b^a_o, B_{t+1})\}$
4     $o \leftarrow \text{argmax}_{\{o \in O\ |\ O(b,a,o)>0\}}\{\text{ UB}(b^a_o, B_{t+1}) - \max_{\alpha \in \Gamma_{t+1}} \alpha \cdot b^a_o\}$
5     $B_{t+1} \leftarrow B_{t+1} \cup \{(b^a_o, \infty)\}$
6     $b \leftarrow b^a_o$
7 end
```

The number of iterations performed by FiVI is at most $(|A||O|)^h$ as is the number of all beliefs. However, this number of iterations is unlikely as the algorithm leads its exploration to the promising areas of belief space.

4.3.5 Backup and Update Heuristics

In every iteration, the FiVI algorithm recomputes the value function and upper bound update for each belief. Both approaches are pretty clean and straightforward. However, they can quickly become inefficient. In value function update, the algorithm executes backup on all beliefs. We note that the number of beliefs is increasing with every iteration. In the upper bound update, the algorithm interpolates the upper bound of every belief using all other beliefs, but only a part of all beliefs affect the resulting value.

The authors of the FiVI algorithm describe possible solutions to both problems.

To reduce the complexity of value function backup, the authors propose to take inspiration from the Perseus (Spaan; Vlassis 2005) algorithm. In each iteration, Perseus randomly selects only a part of beliefs and updates them. This approach ensures each new Γ_t set to be at least as good as the previous one. Furthermore, one backup may improve the value of multiple beliefs, thus removing the requirement to execute backups for all beliefs.

To reduce the complexity of error-bound updates, the authors propose to store the dependencies of all beliefs' error-bound updates. Thus, the single error bounds are not computed from all other beliefs but only from a part, allowing more efficient upper bound updates. This approach is called Dependency-Based Bound Updates (DBBU).

Finally, both improvements can be combined, as they increase the efficiency of different parts of the algorithm. For a more throughout description, we refer reader to (Walraven; Spaan 2019).

Part II

Implementation

Chapter 5

Implementation

This chapter focuses on the practical part of the thesis. In the beginning, we present the *POMDPs.jl* framework we extended with this work and the assigned tasks. Then, we follow up with the analysis and description of the implemented methods. The following chapter presents the benchmark problems, and in the end, we present the validations and benchmarks.

This thesis aims to survey and extend the methods implemented in the *POMDPs.jl* library, emphasizing the finite horizon ones. To add any other methods or interfaces missing to implement such methods and maintain interoperability with the rest of the library. Furthermore, to evaluate the performance of the implemented methods on benchmarks and design new test instances, if applicable.

5.1 POMDPs.jl

POMDPs.jl (Egorov; Sunberg; Balaban; Wheeler; Gupta; Kochenderfer 2017) is an open-source framework for solving MDPs and POMDPs written in Julia. It builds on the interface of the same name, unlike other frameworks for solving MDPs, the *POMDPs.jl* supports solving POMDPs and is not dependent on a specific external problem definition format. For new users, the *POMDPs.jl* presents a fast, easy to learn, and prototype modular tool.

Its design supports the possibility to easily 1) define new problems, 2) create new solvers, or 3) run experiments. Through the unified interface, the goals of *POMDPs.jl* are to simplify adapting to the code written by others, offer simple benchmarking, and mainly build an open-source environment open to new contributions.

The main advantages of *POMDPs.jl* are thus:

1. Its **simplicity** thanks to providing *a minimal set of types and functions necessary to define problems, solvers, or experiments in a partially observable setting.*

2. Its **expressiveness** of interfaces providing *the flexibility to solve both fully or partially observable problems, continuous or discrete.* Furthermore, this paper extends interfaces by allowing for differing between infinite and finite horizon POMDPs. It

also supports using *explicitly defined distributions or generative models. The combination of POMDPs.jl interfaces and Julia Languages makes it easy to prototype, effectively removing the need for problem definitions of another format.*

3. Its interface **extensibility** by *allowing new algorithms to be implemented with minimal effort.*

4. Its **usability** by offering more than 20 solvers that are easy to run and interchange.

The interface offers unified methods for defining problems (such as *states* for representing all states of the problem), or methods for executing the selected algorithm (*solve* method has the same parameters for all solvers). It also unifies the structures for storing resulting policies (*AlphaVectorPolicy* for storing α-vectors and corresponding actions of POMDP policies) or structures for passing the parameters to the solver. Given that the interface defines these methods, the users can use them without changes, no matter the problem or the solver.

5.2 Assignment

At the assignment time, the *POMDPs.jl* framework does not include the Finite Horizon POMDP solvers, which are the main task of this work. The interface deemed essential for such solvers has been proposed but not implemented. Such an interface is needed as the solvers need to operate over single stages while the current interface supports only handling the whole problem representation.

Because of that, the objective of this thesis is to implement the interface for Finite Horizon POMDPs. On top of that, we are to implement the selected algorithms as proof of work. For that, we selected the Finite Horizon Value Iteration for MDPs and Point-Based Finite Horizon Value Iteration for POMDPs.

These algorithms are the foundations for many improvements already published, making them an excellent starting point for future additions to the library. These algorithms are introduced in (Shani; Pineau; Kaplow 2013) and (Walraven; Spaan 2019) and are recommended implementations to any reader interested in developing a new algorithm for the *POMDPs.jl* framework.

We were to evaluate the implemented algorithms against their infinite horizon counterparts. However, while implementing them, we found out that the author of the infinite horizon point-based value iteration *POMDPs.jl* algorithm, which we formerly wanted to use as a benchmark, implemented the algorithm in such a way that it only supports specific POMDPs with two states. Thus, we extended our work to fix and update the infinite horizon point-based value iteration.

5.3 Finite Horizon POMDPs

This chapter introduces the interface for defining Finite Horizon POMDPs *FiniteHorizon-POMDPs.jl*[1]. We implemented the interface in line with the initial design proposed by the

[1]https://github.com/JuliaPOMDP/FiniteHorizonPOMDPs.jl, commit bcda69dd41c18c3327d17fec0cf31f3077ce356e, version 0.3.1

POMDPs.jl co-author Zachary Sunberg in May 2019. Zachary Sunberg also wrote the minimal working example to preserve the interoperability and the idea of the library. The extensions and improvements are part of our work.

The goal of the Finite Horizon POMDPs interface was to create a *POMDPs.jl*-compatible interface for defining MDPs and POMDPs with finite horizons. Its key ideas were in particular, to:

- provide a way for value-iteration-based algorithms to start at the final stage and work backward,
- be compatible with generic *POMDPs.jl* solvers and simulators,
- provide a Finite-Horizon wrapper for Infinite-Horizon MDPs, and
- be compatible with other interface extensions like constrained POMDPs and mixed observability problems.

And its implementation was to contain the declaration of the following interface functions:

- *HorizonLength(::Type<:Union{MDP,POMDP}) = InfiniteHorizon()*
  - *FiniteHorizon*
  - *InfiniteHorizon*
- *horizon(m::Union{MDP,POMDP})::Int*
- *stage_states(m::Union{MDP,POMDP}, t::Int)*
- *stage_stateindex(m::Union{MDP,POMDP}, t::Int, s)*
- *stage_actions(m::Union{MDP,POMDP}, t::Int, [s])*

The whole idea of the interface lies on the concept of the *HorizonLength* type, which returns the *FiniteHorizon* or *InfiniteHorizon* type corresponding to either a (PO)MDP being a finite or infinite horizon. In the case where (PO)MDP is a finite horizon, we can obtain the horizon value by calling the *horizon* method and the current stage by calling *stage* method. Furthermore, the interface extends the methods defined for infinite horizon problems by declaring their finite horizon counterparts. Methods belonging to this group are *stage_states*, which returns states for given stage only, *stage_stateindex* which returns the index of a given state in a given stage; *stage_observations* which returns observations for a given stage and *stage_obsindex*, which returns the index of a given observation in a given stage. *FiniteHorizonPOMDPs.jl* also defines *ordered_stage_states* and *ordered_stage_observations* which return ordered states or observations of a given stage. At the time of writing this work, the *stage_actions* method is not implemented, as most of us known (PO)MDP problems contain actions that span all stages.

By using the combination of methods from *FiniteHorizonPOMDPs.jl* and other interfaces, the user can define his own finite horizon (PO)MDP problem, (PO)MDP solver or other (PO)MDP tool. At the same time, we recognize it can be difficult, or even lengthy to define one. For exactly these cases, the interface contains the utility, whose goal is to transform the user-defined infinite horizon (PO)MDP into a finite horizon one. The design concept of the proposal was formerly to define a simple function that would transform a received infinite horizon (PO)MDP into a finite horizon (PO)MDP wrapper:

- *fixhorizon(m::Union{MDP,POMDP}, T::Int) creates one of*
  - *FiniteHorizonMDP{S, A} <: MDP{Tuple{S,Int}, A}*
  - *FiniteHorizonPOMDP{S, A, O} <: POMDP{Tuple{S,Int}, A, O}*

Finally, this utility became a set of methods that can easily create a package of its own. However, it is necessary to keep the utility inside the *FiniteHorizonPOMDPs.jl* package to be easily found and ready to use when using the interface.

Other than *fixhorizon* method, the utility defines new output types to recognize transformed (PO)MDPs. These types are called *FixedHorizonMDPWrapper* and *FixedHorizon-POMDPWrapper*. If there is no need to differentiate between them, the third type, *FH-Wrapper*, is used. Thanks to this distinction, Julia Programming Language can dispatch the correct method on each call dynamically.

To distinguish between the states of the infinite horizon (PO)MDP and the finite horizon one, the *fixhorizon* utility introduces a new structure. It wraps the pairs of states or observations and stages into tuples *(state, stage)*, encoding the stage information without much performance loss.

The $fixhorizon$ utility defines various methods from multiple interfaces. Namely, the *POMDPs.jl* interface methods ensuring proper functionality no matter the finality of (PO)MDP: *actions transition*, *reward*, *isterminal*, *discount* etc. *states* and *observations* methods wrapping the Infinite Horizon methods in Cartesian product with stages. Furthermore, the $fixhorizon$ utility defines *FiniteHorizonPOMDPs.jl* interface methods described in the text above such as *stage_states* or *stage_observations*. Other than that, the $fixhorizon$ utility implements methods from *POMDPModelTools.jl*, which define distributions for (PO)MDPS or methods for ordering states, actions, and observations. The *fix_horizon* utility also implements the methods from *Random.jl*, offering the mean to compute probabilistic data about the distributions - *mean*, *mode*, *support*, or *pdf*.

To define the Finite Horizon (PO)MDP instance, the user has to define the methods declared in *POMDPs.jl* (Egorov; Sunberg; Balaban; Wheeler; Gupta; Kochenderfer 2017) and from other library's interfaces based on his solver's choice. At this moment, the user is open to choose whether he wants to continue to implement his own Finite Horizon (PO)MDP or if he leverages the utility to transform his Infinite Horizon (PO)MDP to Finite Horizon one. We implement both approaches as a part of this work at *JuliaPOMDP/FiniteHorizonPOMDPs.jl.*

5.4 Finite Horizon Value Iteration

The Finite Horizon Value Iteration[2] (FHVI) extends the original Value Iteration algorithm (defined in Section 2.2.4) with the knowledge of Finite Horizon MDPs (from Section 2.3). The algorithm offers a one-pass optimal solution. The FHVI iterates over the stages in reversed order from the last stage (excluding the terminal stage) to the first one and solves

[2] https://github.com/JuliaPOMDP/FiniteHorizonValueIteration.jl, commit 8b634e55e54c920b3845358bee4d462bb944b0f3, version 0.3.0

each stage with the Value Iteration algorithm. In the end, FHVI combines the resulting policies of all stages into the resulting policy of a whole solver.

The algorithm is written in line with the *POMDPs.jl* solvers' architecture, consisting of three essential pillars - the structure for storing the algorithm parameters *FiniteHorizonSolver*, the policy structure *FiiteHorizoVakuePolicy* and the method executing the algorithm, *solve*.

As the algorithm is one-pass, the *FiniteHorizonSolver* structure contains only the boolean *verbose* parameter, denoting whether to print debugging output or not.

```
struct FiniteHorizonSolver <: Solver
    verbose::Bool
end
```

The algorithm policy *FiniteHorizonValuePolicy* contains the Q matrix consisting of state and action index mapping to value, *util* array mapping given state index to the best reward obtained after executing all actions executable from given state, *policy* array mapping given state index to the action index corresponding to the utility value of given state. The *include_Q* boolean flag for storing the Q matrix and m for storing the MDP problem definition.

```
mutable struct FiniteHorizonValuePolicy{Q<:Array, U<:Array,
                P<:Array, A<:Array, M<:MDP} <: Policy
    qmat::Q
    util::U
    policy::P
    action_map::A
    include_Q::Bool
    m::M
end
```

The solver also defines *value* and *action* methods to obtain the value or the action of the given state.

The *solve* method checks whether the MDP is a Finite Horizon one, initializes the policy and utility vector, and then reverse iterates over stages (excluding the terminal stage), executing the Value Iteration and updating the policy in each iteration.

```
function solve(solver::FiniteHorizonSolver, m::MDP)
    # check finality of MDP
    # initialize policy and utility

    # iterate reversely over stages
        # execute Value Iteration over the given stage
        # update policy with the result of Value Iteration

    # return policy
end
```

5.5 Point-Based Value Iteration

The Point-Based Value Iteration[3] (PBVI) algorithm is the second version of the package. However, the former algorithm worked only for two states and completely omitted the expansion phase. The former algorithm initiated the initial belief space with the uniformly distributed beliefs. We fully reimplemented the former algorithm but inspired ourselves with the former version.

The algorithm accepts problems defined in *POMDPs.jl* interface.

We hand over the PBVI's setting by the *PBVISolver* structure, which accepts the following parameters: *num_iteration* denoting the maximal number of iterations the solver is to run, ϵ denoting the maximal gap between the α-vectors from consecutive iterations of step improvement and *verbose* storing the boolean flag for printing the debug output.

```
struct PBVISolver <: Solver
    max_iterations::Int64
    ϵ::Float64
    verbose::Bool
end
```

The PBVI uses the *AlphaVectorPolicy* defined in *POMDPPolicies.jl*. The *AlphaVectorPolicy* contains the POMDP *pomdp* to which the policy belongs, *n_states* storing the number of states, a list of α-vectors *alphas*, and a mapping from α-vectors to actions *action_map*.

```
struct AlphaVectorPolicy{P<:POMDP, A} <: Policy
    pomdp::P
    n_states::Int
    alphas::Vector{Vector{Float64}}
    action_map::Vector{A}
end
```

The *solve* method's parameters are in line with the POMDPs interface. That is, the method *solve* accepts two parameters, solver's settings *solver* and POMDP *pomdp* to be solved. The algorithm is based on section 3.4. It first initializes all necessary representations, then it iterates over the α-vectors and improves them, and finally expands the belief space. If the expansion algorithm does not find any new belief worth visiting, it terminates the

[3]https://github.com/JuliaPOMDP/PointBasedValueIteration.jl, commit a8f93740be6947e46f98fab2f23cf03a89c91bd6, version 0.2.1

algorithm.

```
function solve(solver::PBVISolver, pomdp::POMDP)
    # initialize all POMDP representations

    # iterate up to max_iterations times
        # improve α-vectors
        # expand belief space
        # check whether the belief space expanded, if not terminate early

    # return AlphaVectorPolicy
end
```

5.6 Finite Horizon Point-Based Value Iteration

Finite Horizon Point-Based Value Iteration[4] (FiVI) is the state-of-the-art algorithm that leverages the point-based approach and employs the heuristics of the best performing infinite horizon POMDP problem solvers in Finite Horizon environments.

The solver accepts its parameters in the form of *FiVISolver* structure. *FiVISolver* supports two arguments. *precision* denoting the maximum tolerance in the gap between upper and lower bound and *time-limit* during which the solver is to be executed.

```
struct FiVISolver <: Solver
    precision::Float64
    time_limit::Int64
end
```

The solver outputs the resulting policy and the upper bound of the initial belief. The policy is stored in the *StagedPolicy* which we contributed with to the *POMDPPolicies.jl*[5]. The *StagedPolicy* stores the array of the resulting policies for each stage *staged_policies* and the problem definition *pomdp*.

```
struct StagedPolicy{M<:FiniteHorizonPOMDPs,
                        FHWrapper, P<:Policy}<:Policy
    pomdp::M
    staged_policies::Array{P, 1}
end
```

The *solve* method definition is based on the interface. It accepts the algorithm parameters and the problem definition. The algorithm checks whether the problem is a finite horizon POMDP, initiates all variables needed to run the algorithm, and sets the upper bound of the beliefs in the terminal stage to zero. The algorithm then iterates in the outer loop until one of the terminal conditions evaluates to true. The inner loop iterates over the stages in the reversed order. It sequentially empties the vector of *alpha*-vectors, backups

[4]https://github.com/Omastto1/FiVI.jl, commit f7caaa2fbdff9a2364fc64b4158396714459666b
[5]https://github.com/Omastto1/POMDPPolicies.jl, commit f15f49f6d9d546fd6ee0020c61cc7753503532b0

the α-vectors of beliefs, and updates their upper bound. If the terminal conditions evaluate to false, the algorithm expands its belief space.

Unlike in the (Walraven; Spaan 2019), our algorithm expands its belief space as the last thing it does in the outer loop. By using this approach, the algorithm successfully executes. If the algorithm executed the belief space expansion at the beginning of the loop, it would calculate with the infinity initiated upper bounds of beliefs and end up with errors. The algorithm's author also uses this approach in the implementation of the algorithm here[6].

```
function solve(solver::FiVISolver, pomdp::POMDP)
    # check finality of pomdp, if pomdp is infinite throw error

    # initiate empty vectors for α-vectors, belief space, and belief set
    # initialize elapsed time to zero

    # set upper bound of belief in terminal stage to zero

    # iterate while all terminal conditions evaluate to true
        # iterate stages in reversed order
            # empty the vector for α-vectors in t-th stage

            # backup all belief in t-th stage with the α-vectors
            #       from t+1-th stage
            # update upper bound of all belief vectors with the belief
            #       space from t+1-th stage

        # check terminal conditions (time elapsed > time_limit
        #           or gap lower then maximum allowed)

        # expand belief space and update belief set

    # initialize StagedPolicy
    # return policy and upper bound of initial belief
end
```

5.7 Optimizing and Profiling

To optimize the solvers and find possible bottlenecks, we used the package *Profiler.jl*.

It turns out that even if the algorithms are optimized, the most significant bottlenecks come from incorrectly implemented user-defined functions. As the solvers heavily depend on the interface methods, every detail on the user's side is relevant. Such functions allocate small memory blocks each time the solver calls them, resulting in massive overhead in memory consumption. Thus, we obtain the most significant performance gains when the methods return tuples or generators instead of arrays.

Based on the profiling of methods, we have implemented speed-ups and improvements to the implementations of solvers. The most beneficial changes are connected to memory

[6]https://github.com/AlgTUDelft/ConstrainedPlanningToolbox/blob/master/src/main/java/algorithms/pomdp/cgcp/FiniteVI.java

handling as well. Consider pushing the new values to the array. To reduce the memory overhead, we preallocate the array before the loop. Other memory consumption improvements come from caching results instead of computing them again each time it is necessary.

The next improvement we made was to vectorize the code. However, this was often not an improvement, as Julia uses intelligent optimization for loops. The Julian vectorization using the dot operator makes the code more readable. However, it is the programmer's task to verify whether the vectorization improves the algorithm's performance.

Part III

Experiments

Chapter 6

Domains

In this chapter, we describe benchmark problems for both MDP and POMDP solvers, and in the end, we benchmark and validate solvers implemented as part of this work.

For MDPs, we have come up with custom, straightforward, dynamically sized problems sufficient for our validations of MDP solvers. Two problems that we defined are called 1D Grid and Pyramid. The 1D Grid imitates a simplified version of MDP 4 x 3 Grid published in (Russell; Norvig 2010). The Pyramid's design resembles an exploration problem with a limited horizon, wherein each stage t the agent can visit up to t states. The Pyramid simulates a specific finite horizon problem, where each stage has different states. The 1D Grid and Pyramid MDP problems are defined here[1].

Unlike the MDPs, the scientists are still actively researching the POMDPs. Thus, in our benchmarks, we use well-known POMDP benchmark problems Mini Hallway and Hallway introduced in (Littman; Cassandra; Kaelbling 1995). Furthermore, one of the authors published implementations of these and other problems at (Cassandra 1999). However, the author uses a format, which the *POMDPs.jl* framework does not support yet.

As there is no literature on MDP problems, and the POMDPs problems' implementation format does not match any format supported by *POMDPs.jl*, we implemented the benchmark problems mentioned above to the framework.

In the following text, we briefly describe each benchmark domain's properties (Table 6.1).

Domain	$\|S\|$	$\|A\|$	$\|\Omega\|$	Observability	Transitions	Observations
1D Grid	custom	2	-	Fully Observable	Stochastic	-
Pyramid	custom	2	-	Fully Observable	Stochastic	-
Mini Hallway	13	3	9	Partially Observable	Deterministic	Deterministic
Hallway	60	5	21	Partially Observable	Stochastic	Stochastic

Table 6.1: The domains used in our experiments

[1]https://github.com/JuliaPOMDP/FiniteHorizonValueIteration.jl/tree/master/test/instances

6.1 1D Grid

1D Grid is a custom MDP problem (Figure 6.1) with a user-defined number of states with goal states on both ends. The agent can execute two actions (*left* and *right*) with a 30% chance of executing the wrong one. For each action, the agent pays one point, and for reaching the terminal state, the agent receives ten points.

The 1D Grid imitates a simplified version of MDP 4 x 3 Grid published in (Russell; Norvig 2010). We design the 1D Grid problem as the infinite horizon problem. In experiments, we transform the problem into the finite horizon one and evaluate the performance of finite horizon VI and infinite horizon VI, solvers.

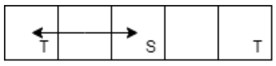

Figure 6.1: 1D Grid environment

6.2 Pyramid

Pyramid is a pyramid-like MDP problem (Figure 6.2) explicitly designed for Finite Horizon MDP solvers. It resembles an exploration problem, where the MDP's state space is increasing by one state in each stage.

Starting in the first stage with one state at the Pyramid's top, the pyramid problem grows "downward", increasing its number of states in each stage (the first stage has *1* state, ..., the nth stage has n states). The Pyramid MDP has two possible actions, moving $down-left$ or *down – right*. By default the terminal states are contained in the last (*horizon + 1*) stage. Optionally, the user can define additional goal states. For each action, the agent pays one point. For reaching a goal state, the agent receives ten points. For reaching a terminal state, the agent receives zero points.

In experiments, we evaluate the difference between specialized and generalized VI algorithms. In the Pyramid problem, the Finite Horizon Value Iteration evaluates only n states in the *nth* stage, but the Infinite Horizon Value iteration evaluates all states in each iteration.

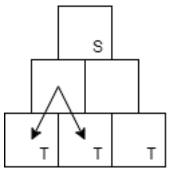

Figure 6.2: Pyramid environment

6.3 Mini Hallway

Mini Hallway(Littman; Cassandra; Kaelbling 1995) is a small navigation POMDP problem consisting of 13 states (Figure 6.3). The environment consists of 3 rooms where the agent can face four directions and one terminal room denoted by a star where the agent's orientation does not matter. The agent can receive nine observations (relative locations of surrounding walls in each position) and execute three actions ($forward$, $rotateleft$, $rotateright$).

The Mini Hallway problem models a hallway with a robot whose task is to enter the room marked with the star. For reaching the room denoted by a start, the agent receives one point. The actions cost zero points. The transitions and observations are deterministic, and the problem defines the initial state belief as a uniform distribution over all twelve non-terminal states.

The Mini Hallway problem has sufficient size in finite horizon experiments to show the performance difference between the FiVI and PBVI algorithm. In more significantly sized finite horizon problem benchmarks, the PBVI's performance rapidly decreases, making the algorithm execution so long that the benchmark does not make sense.

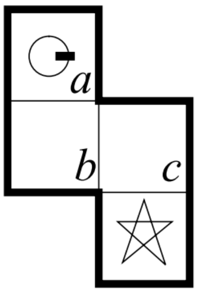

Figure 6.3: Mini Hallway environment

6.4 Hallway

Hallway(Littman; Cassandra; Kaelbling 1995) is a middle-sized navigation POMDP problem (Figure 6.4) consisting of 60 states. The environment consists of 14 rooms where an agent can face four directions and one room marked with a star with four goal states. Other than that, the model contains 21 observations (relative locations to surrounding walls, to the star denoting the goal states, and to three numbered landmarks in each position) and five actions ($stayinplace$, $moveforward$, $turnright$, $turnleft$, $turnaround$).

Both transitions and observations are extremely noisy. For reaching goal states, the agent receives one point. The agent does not pay for executing actions. The problem defines the initial state belief as a uniform distribution over all 56 non-terminal states.

The Hallway problem represents a middle-sized problem sufficient for benchmarking the Point-based Value Iteration algorithm against the SARSOP in infinite horizon environments. However, the Hallway problem is unbearably large for benchmarking the Point-Based Value Iteration in finite horizon environments.

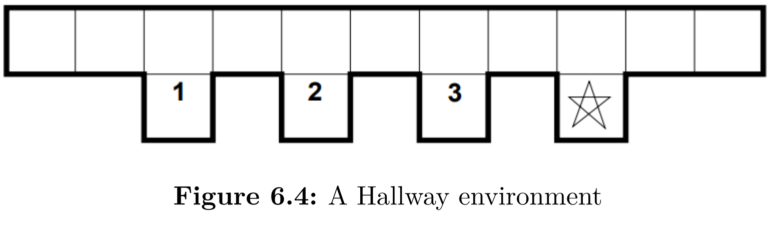

Figure 6.4: A Hallway environment

In POMDPs benchmark problem definitions in $POMDPModels.jl$, we have changed the transition from terminal states. In (Littman; Cassandra; Kaelbling 1995) the authors define the transitions from terminal states with the initial belief distribution. In our implementation, we define the transitions from terminal states as deterministic transitions to the same terminal states.

Chapter 7

Validations and Benchmarks

In the validations and benchmarks section, we are using algorithms implemented in the *POMDPs.jl* framework. All experiments were executed on a 2.6Ghz i7 CPU with four cores and 16GB RAM. The algorithms were not run in parallel, nor were they multi-threaded. Results for MDP solvers were evaluated using *BenchmarkTools.jl* package. The benchmarking stopped either after evaluating 10 000 algorithm executions or after 100 seconds. The POMDP results were evaluated for each algorithm iteration separately. For each policy resulting from each iteration of each algorithm, we simulated the policy's behavior by running *POMDPSimulators.jl* simulations 10 000 times.

In the following text, we evaluate the performance of all algorithms we implemented as part of this thesis - Finite Horizon Value Iteration, Point-Based Value Iteration, and Finite Horizon Point-Based Iteration.

7.1 Finite Horizon Value Iteration

To benchmark the Finite Horizon Value Iteration, we evaluated the algorithm against the Infinite Horizon Value Iteration. In our benchmarks, we used the MDP problems defined in the previous section - 1D Grid and Pyramid and transformed them into a Finite Horizon POMDP. We evaluate the results against our theoretical memory requirement assumptions.

In our benchmarks, we evaluate the improvement connected with using specialized Finite Horizon Value Iteration. While both solvers guarantee their convergence, the Finite Horizon Value Iteration offers a one-pass solution, but the Infinite Horizon Value Iteration converges after an unknown number of iterations.

Furthermore, the Infinite Horizon Value Iteration updates the value function of all states in each iteration. On the other hand, the Finite Horizon Value Iteration only updates the value of states that corresponds to a given stage and ends after *horizon* iterations.

7.1.1 1D Grid

In the 1D Grid problem, we used problems with 5, 11, 25, 51, and 101 states in each stage. The terminal states were at the edges and the starting point in the middle. In this problem, it takes the agent $\frac{n-1}{2}$ actions to get to one of the terminal states. It takes the same amount of iterations to update the value functions of all states correctly. Thus, for solving the finite horizon 1D Grid problem, the Finite Horizon Value Iteration algorithm requires at least $\frac{\frac{n-1}{2}^2}{2}$ less memory. For every iteration out $\frac{n-1}{2}$ iterations that the Finite Horizon Value Iteration executes, the Finite Horizon Value Iteration needs at least $\frac{n-1}{2}$ lesser states (every iteration evaluates only states corresponding to a given stage instead of all states). Furthermore, the Finite Horizon Value Iteration initiates state values for a new stage, which corresponds to halving in the memory requirements formula.

The benchmarking results are validating our assumptions. Consider 1D Grid problem with 51 states. According to our assumption, it would take $\frac{25*25}{2}$ less memory. Thus, we assume the memory requirement of Finite Horizon Value Iteration to be 312.5 times lesser than the memory required for the Infinite Horizon Value Iteration used on the finite horizon 1D Grid problem. 312.5 times reduction equals to reduction to 0.32 %. The experiment in Table 7.2 validates our prediction.

Similar trends can be seen in Table 7.1. The dependence between the number of states of one stage in Finite Horizon 1D Grid Problem and the time needed to solve the problem is in Figure 7.1.

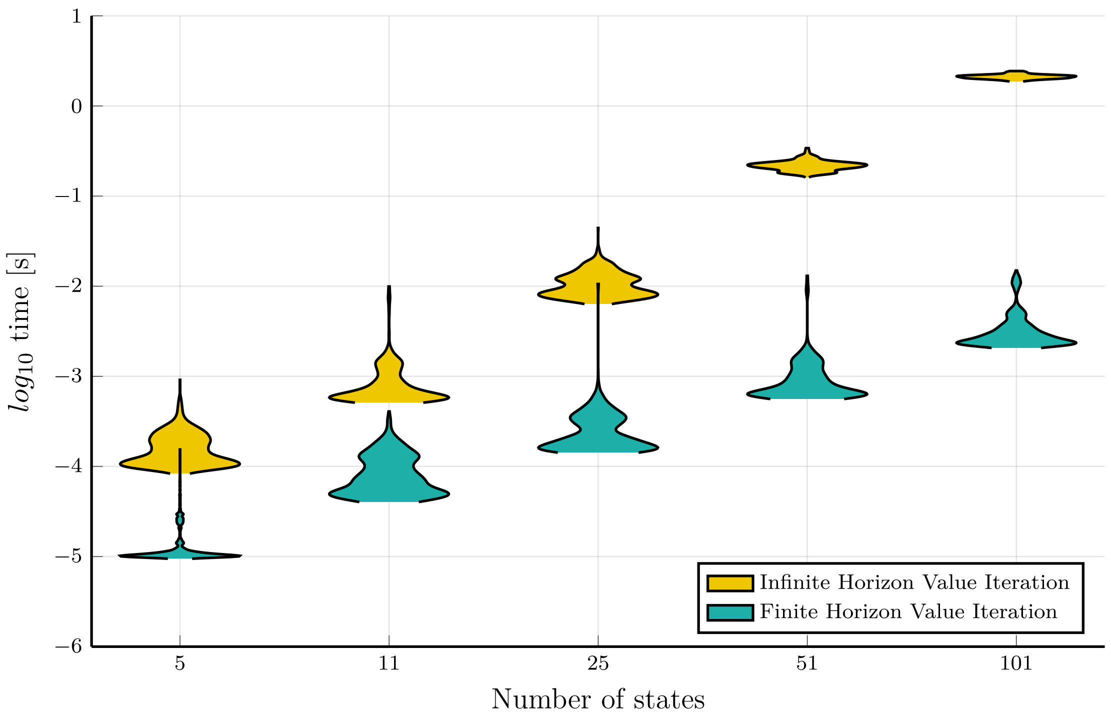

Figure 7.1: Comparison of Finite and Infinite Horizon Value Iteration solvers on various sized staged 1D Grid problem

Number of states	5	11	25	51	101
IH VI [ms]	0.165	0.933	10.850	218.452	2115.453
FH VI [ms]	0.014	0.089	0.289	1.045	3.466
Improvement [×]	11.8	10.5	37.6	209.0	610.3

Table 7.1: Mean solving time comparison of Finite Horizon and Infinite Horizon Value iteration of various sized staged 1D Grid problem

Number of states	5	11	25	51	101
IH VI [MB]	0.044	1.118	17.904	313.767	4336.078
FH VI [MB]	0.009	0.051	0.225	0.987	3.768
Reduced to [%]	20.34	4.54	1.25	0.31	0.08

Table 7.2: Memory consumption comparison of Finite Horizon and Infinite Horizon Value iteration of various sized staged 1D Grid problem

7.1.2 Pyramid

In the Pyramid problem, we define the number of horizons instead of the number of states. In the pyramid problem, the number of states in the last stage is equal to the horizon of this problem. The Pyramid MDP problem requires less memory than the 1D Grid problem, and thus we extend our validations to 5, 11, 25, 51, 101, 251, 501 horizon Pyramid MDP.

Unlike the 1D Grid MDP, the agent in Pyramid MDP needs to execute *number of horizons* actions to get into the terminal state. The memory requirements in Table 7.4 show memory reduction increasing with the horizon. However, for the Pyramid problem, the memory reduction is lower than in the 1D Grid problem. The memory reduction is lower because the Finite Horizon Value Iteration deals with the states corresponding to given stage t and the following stage $t + 1$, which correspond to a more significant part of the whole state space in the case of the 1D Grid problem.

As expected, the specialized Finite Horizon Value Iteration outperforms the general Infinite Horizon Value Iteration in terms of performance as well. The time-wise performances are shown in Table 7.3. The distribution of time-wise performances for each size of the problem is in Figure 7.2.

Horizon	5	11	25	51	101	251	501
IH VI [ms]	0.058	0.119	0.518	3.651	27.456	401.497	3289.083
FH VI [ms]	0.021	0.031	0.099	0.236	0.668	2.605	9.605
Improvement [×]	2.8	3.8	5.2	15.5	41.0	154.1	342.4

Table 7.3: Mean solving time comparison of Finite Horizon and Infinite Horizon Value iteration of various sized staged Pyramid problem

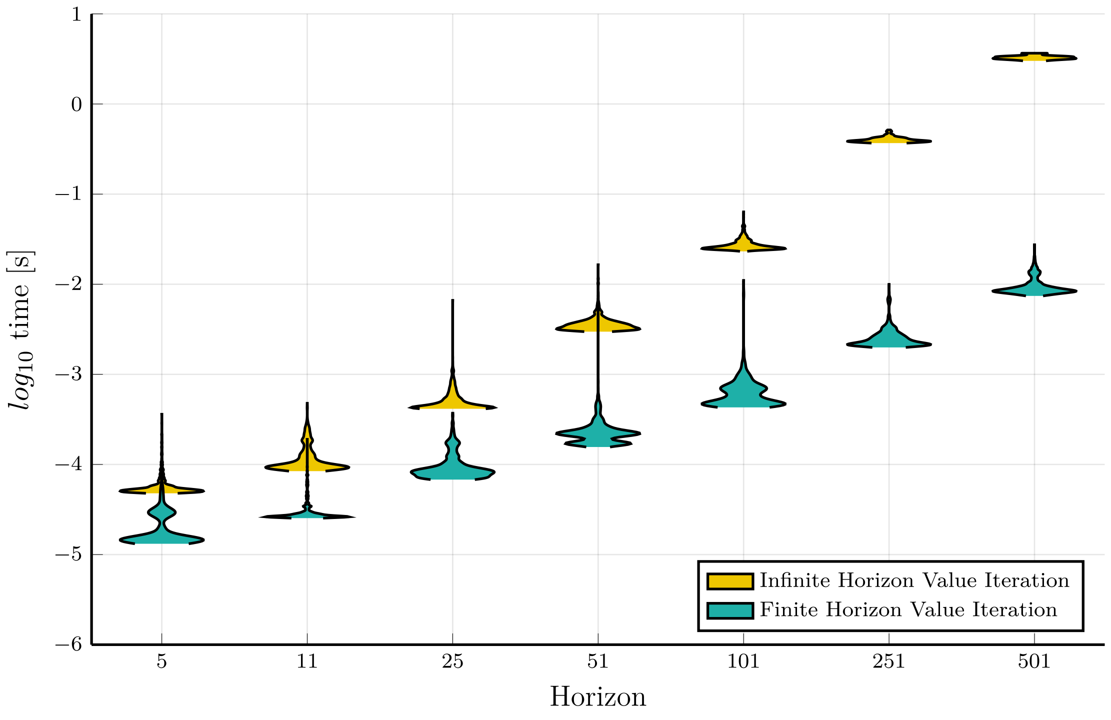

Figure 7.2: Comparison of Finite and Infinite Horizon Value Iteration solvers on staged Pyramid problem with different horizons

Horizon	5	11	25	51	101	251	501
IH VI [MB]	0.011	0.028	0.096	0.376	1.481	6.923	23.574
FH VI [MB]	0.007	0.015	0.041	0.119	0.376	1.970	7.304
Reduced to [%]	59.0	52.6	42.7	31.8	25.4	28.5	31.0

Table 7.4: Memory consumption comparison of Finite Horizon and Infinite Horizon Value iteration of various sized staged Pyramid problem

7.2 Point-Based Value Iteration

Formerly, we wanted to benchmark the FiVI against the state-of-the-art algorithm SARSOP (Kurniawati; Hsu; Lee 2008). However, the SARSOP implementation in *POMDPs.jl* runs an external solver that requires a specific POMDP format as input. The *SARSOP.jl* parser does not support the *FiniteHorizonPOMDPs.jl* interface methods declared as part of this thesis, and as such, could not be used.

Another solver that we considered to be a suitable reference did not contain a correctly implemented algorithm. The solver is called *PointBasedValueIteration.jl*, and we decided to implement it on top of the original assignment.

This section serves as the validation of the correct algorithm implementation. We evaluate the Point-Based Value Iteration against the SARSOP solver. The SARSOP is based on the PBVI, and as such, the PBVI should not outperform the SARSOP. However, the PBVI can obtain an asymptotically similar reward after an extensive solution execution.

In Figure 7.3, we can see that the PBVI starts with a decent 0.3 mean reward, then dips, and in the end reaches a similar reward as the SARSOP. In the beginning, the PBVI's explored space contains only a tiny part of the belief space. The starting reward can be awarded to α-vector correctly executing an action corresponding to maximizing the value function of the belief unknown to the solver. PBVI did not yet calculate the value function of those specific beliefs, thus making them inaccurate. In the middle dip, the algorithm starts to explore the first most promising beliefs. The solver knows that the beliefs are promising but yet does not know which way to go. The solver finally explores the optimal path in the last iteration and gradually updates all other beliefs with new actions, obtaining the same reward as SARSOP.

The SARSOP leverages heuristics to explore the state-space reachable under the optimal policy efficiently. The PBVI does not. The use of heuristics results in PBVI being significantly slower than SARSOP. The advantage of heuristics highlights more than hundreds of times more efficient memory consumption of SARSOP.

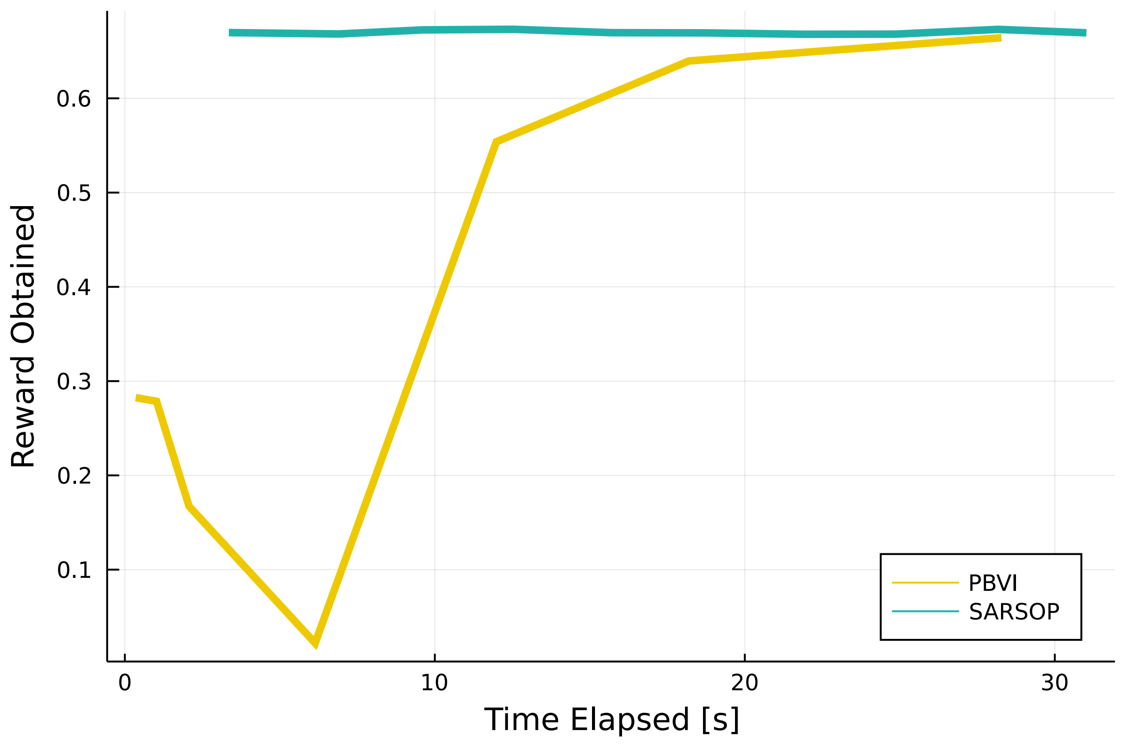

Figure 7.3: Comparison of the expected reward depending on the length of problem solving between Point Based Value Iteration and SARSOP on the Hallway problem.

7.3 Finite Horizon Point-Based Value Iteration

The FiVI algorithm presents a state-of-the-art algorithm for Finite Horizon POMDP solvers for *POMDPs.jl*. Moreover, the FiVI is a great starting point for further contribution, as the authors propose further heuristics in (Walraven; Spaan 2019).

This section shows that we correctly implemented the *FiVI.jl* solver. As we failed to benchmark the FiVI against the more advanced SARSOP, we benchmark the FiVI algorithm against the PBVI algorithm. In the benchmark scenario, the FiVI easily dominates PBVI, the base solver for Infinite Horizon POMDPs, in terms of time performance.

We have benchmarked the FiVI and PBVI on the staged Mini Hallway problem. Mini Hallway offers a moderate problem for FiVI but is a significant challenge in size for PBVI. The choice of Finite Horizon POMDPs with fewer states, as TigerPOMDP or BabyPOMDP from *POMDPPolicies.jl*, does not represent a suitable sized benchmark. On the other hand, the finite horizon Hallway problem would take the PBVI hundreds or thousands of seconds to solve.

We benchmark the Mini Hallway problem with multiple choices for the horizon value. We show the results in Table 7.5. With different horizon choices, we simulate multiple various-sized problems. The horizon values are 3, 6, 9. With the uniformly distributed initial belief, horizon 3 offers only a slight possibility for reaching terminal states. Horizon 6 is a middle-sized problem in which the agent has more steps to execute and improves its expected reward. Horizon 9 simulates a long-range problem, challenging for the PBVI solver.

In the first two benchmarks, we can see that the PBVI almost reaches a similar expected reward as the FiVI. However, the time needed to obtain such a policy is far greater. In the last benchmark, the PBVI does not solve the problem in the time limit and stops with the expected reward of 0.31. The FiVI, on the other hand, converges and obtains an expected reward of 0.58.

The most significant problem for leveraging the PBVI in Finite Horizons is that it starts with only an initial belief, and it has to expand its belief space iteratively. FiVI, on the other hand, starts with the initial belief and another $|S|$ beliefs in each stage. This way, the FiVI algorithm contains significantly more beliefs in its first stage, allowing it to solve problems significantly faster.

	Expected reward			Time[s]		
horizon	3	6	9	3	6	9
PBVI	0.23	0.31	0.31	18.59	90.45	279.69
FiVI	0.24	0.33	0.58	2.36	30.52	37.59

Table 7.5: Comparison of expected rewards and time needed to solve the problem with Finite Horizon or Infinite Horizon Value Iteration on Mini Hallway problem.

Chapter 8

Conclusion

In this work, we introduced the interface for Finite Horizon (PO)MDPs to the *POMDPs.jl* framework, surveyed and analyzed existing methods, and extended the framework with the new solvers, mainly specialized in Finite Horizon problems.

POMDPs.jl is an open-source framework for using POMDPs implemented in Julia. *POMDPs.jl* combines interoperable interfaces that users can use to implement their own (PO)MDP problem or solver. Users can also use any of the already existing tools the framework offers, for example, simulating.

At the beginning of the thesis, we introduced all the necessary background for MDPs and POMDPs, both Infinite Horizon and Finite Horizon, and the motivation on why it is beneficial to use Finite Horizon (PO)MDPs in a time-constrained environment.

On top of that knowledge, we implemented a missing interface for Finite Horizon (PO)MDPs called *FiniteHorizonPOMDPs.jl* according to its design presented in the package template. With the interface set up, we implemented the *fixhorizon* utility. Thanks to the *fixhorizon* utility, anyone who has already implemented their own (PO)MDP in *POMDPs.jl* framework can transform their Infinite Horizon (PO)MDP into a Finite Horizon one with only a few lines of code.

To solve the finite horizon problems, we implemented the modified version of the Value Iteration algorithm, the *FiniteHorizonValueIteration.jl.* It specializes in solving the Finite Horizon MDPs. The *FiniteHorizonValueIteration.jl* solver builds on top of a custom implementation of Infinite Horizon Value Iteration. It introduces the Finite Horizon solvers to an uninformed user in an easy-to-grasp way, demonstrating its structure and benefits.

During the survey of POMDP methods, we found out that we can not use the majority of already implemented solvers in *POMDPs.jl.* As a solution to a missing benchmarking method for the Finite Horizon POMDPs, we implemented an additional solver, the $PointBasedValueIteration.jl$. Apart from being a benchmark for the other algorithm that we implemented, it is an essential algorithm that is a foundation for many other solvers.

With the interface and benchmark solver contributions completed, we implemented the state-of-the-art Finite Horizon Point-Based Value Iteration algorithm in package $FiVI.jl$. The FiVI solver presents a great heuristic solution for solving finite horizon problems. Furthermore, it is an excellent starting point for any future contributions, as the algorithm's authors propose improving heuristics. And finally, the FiVI demonstrates the necessity for specialized Finite Horizon solvers, as it dominates the performance of general solvers in finite horizon problems. Other than that, the use of finite horizon solvers often results in different policies focusing on rewards reachable in a given time span. Thus, the finite horizon solvers often improve the resulting reward.

Other than mentioned solvers and interface, we contributed with additional POMDP models, policies, and other minor changes necessary to provide the Finite Horizon POMDP support. The benchmark problems used in our validations are implemented in *POMDP-Models.jl*, the *StagedPolicy* used in $FiVI.jl$ is implemented in $POMDPPolicies.jl$.

Finally, we provided benchmarks and experiments to validate our methods' correctness and their advantages.

The finite horizon methods clearly show that their infinite horizon counterparts do not perform well on specialized finite horizon problems. The benchmarks show up to a hundredfold improvement in performance and memory efficiency from finite horizon solvers on benchmarked finite horizon problems. Furthermore, finite horizons tend to perform even better with more extensive problems. Thus, making the specialized finite horizon solvers essential in environments with a finite horizon.

The PBVI benchmark against the SARSOP algorithm shows that even that the PBVI is a generic solver without any clever heuristic, it still achieves similar results in the long run.

As future work, we deem it essential to improve the support for Finite Horizon POMDPs. The *POMDPs.jl* main framework packages support Finite Horizon POMDPs, but there are still a few minor packages that do not. For example *POMDPSimulators.jl.* Other than that, Finite Horizon (PO)MDPs may well deserve a discrete belief designed for finite horizon specifically, as it could simplify some implementations.

In terms of algorithm improvements, in particular the performance of $FiVI.jl$, it can be improved with caching of α-vectors, or heuristics introduced in (Walraven; Spaan 2019)

We believe that our framework addition will be of great use to everyone searching for a way to solve their Finite-Horizon (PO)MDP problem with an already written package that is also efficient. The *POMDPs.jl* is an excellently designed framework for POMDPs oriented tasks, and with the advent of Julia, it will get even more attention.

In conclusion, with the new interface, solvers, and other minor methods, we contribute to an extensive framework *POMDPs.jl* and further improve its library environments. We introduce a whole new branch of finite horizon problems and tools not supported by *POMDPs.jl* at the time of its completion. Finally, we test the correctness and efficiency of the newly implemented solvers.

Bibliography

1. BELLMAN, Richard. A Markovian Decision Process. *Journal of Mathematics and Mechanics*. 1957, roč. 6, č. 5, pp. 679–684. ISSN 00959057, ISSN 19435274. Available also from: `http://www.jstor.org/stable/24900506`.
2. KOLOBOV, Mausam; KOLOBOV, Andrey. Planning with markov decision processes: An AI perspective. *Synthesis Lectures on Artificial Intelligence and Machine Learning*. 2012, roč. 17, pp. 1–203. ISBN 9781608458868. ISSN 19394608. Available from DOI: `10.2200/S00426ED1V01Y201206AIM017`.
3. RUSSELL, Stuart; NORVIG, Peter. Artificial Intelligence: A Modern Approach. In: 3. vyd. Prentice Hall, 2010, pp. 42–44.
4. SHANI, Guy; PINEAU, Joelle; KAPLOW, Robert. A survey of point-based POMDP solvers. *Autonomous Agents and Multi-Agent Systems*. 2013, roč. 27, č. 1, pp. 1–51. ISSN 1573-7454. Available from DOI: `10.1007/s10458-012-9200-2`.
5. PINEAU, Joelle; GORDON, Geoffrey; THRUN, Sebastian. Point-based value iteration: An anytime algorithm for POMDPs. In: 2003, pp. 1025–1032.
6. WALRAVEN, Erwin; SPAAN, Matthijs T. J. Point-Based Value Iteration for Finite-Horizon POMDPs. *J. Artif. Int. Res.* 2019, roč. 65, č. 1, pp. 307–341. ISSN 1076-9757. Available from DOI: `10.1613/jair.1.11324`.
7. LOVEJOY, William S. Computationally Feasible Bounds for Partially Observed Markov Decision Processes. *Operations Research*. 1991, roč. 39, č. 1, pp. 162–175. ISSN 0030364X, ISSN 15265463. Available also from: `http://www.jstor.org/stable/171496`.
8. HAUSKRECHT, Milos. Value-Function Approximations for Partially Observable Markov Decision Processes. *J. Artif. Intell. Res.* 2000, roč. 13, pp. 33–94. Available also from: `http://dblp.uni-trier.de/db/journals/jair/jair13.html#Hauskrecht00`.
9. SPAAN, Matthijs T. J.; VLASSIS, Nikos. Perseus: Randomized Point-Based Value Iteration for POMDPs. *J. Artif. Int. Res.* 2005, roč. 24, č. 1, pp. 195–220. ISSN 1076-9757.
10. SMITH, Trey; SIMMONS, Reid G. Heuristic Search Value Iteration for POMDPs. *CoRR*. 2012, roč. abs/1207.4166. Available from arXiv: `1207.4166`.
11. SHANI, Guy; BRAFMAN, Ronen; SHIMONY, Solomon. Forward search value iteration for POMDPs. In: 2007, pp. 2619–2624.
12. KURNIAWATI, Hanna; HSU, David; LEE, Wee Sun. Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In: *Robotics: Science and systems*. 2008, sv. 2008.

13. HAUSKRECHT, M. Value-Function Approximations for Partially Observable Markov Decision Processes. *Journal of Artificial Intelligence Research*. 2000, roč. 13, pp. 33–94. ISSN 1076-9757. Available from DOI: `10.1613/jair.678`.

14. EGOROV, Maxim; SUNBERG, Zachary N.; BALABAN, Edward; WHEELER, Tim A.; GUPTA, Jayesh K.; KOCHENDERFER, Mykel J. POMDPs.jl: A Framework for Sequential Decision Making under Uncertainty. *Journal of Machine Learning Research*. 2017, roč. 18, č. 26, pp. 1–5. Available also from: `http://jmlr.org/papers/v18/16-300.html`.

15. LITTMAN, Michael L.; CASSANDRA, Anthony R.; KAELBLING, Leslie Pack. Learning Policies for Partially Observable Environments: Scaling Up. In: *Proceedings of the Twelfth International Conference on International Conference on Machine Learning*. Tahoe City, California, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 362–370. ICML'95. ISBN 1558603778.

16. CASSANDRA, Anthony R. *Tony's POMDP Page* [`https://cs.brown.edu/research/ai/pomdp/index.html`]. 1999. [Online; accessed 7-May-2021].

Appendices

A Attached files

We are attaching a folder with the source codes of all the packages we have implemented or contributed to. The folder also contains a directory with the experiments and corresponding data.