Università di Genova | **DIBRIS** DIPARTIMENTO DI INFORMATICA, BIOINGEGNERIA, ROBOTICA E INGEGNERIA DEI SISTEMI

Navigating by Scent: A Model-Based Approach to Olfactory Navigation using Partially Observable Markov Decision Processes

Arnaud Ruymaekers

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
https://www.dibris.unige.it/

Università di Genova

MSc Computer Science
Data Science and Engineering Curriculum

Navigating by Scent: A Model-Based Approach to Olfactory Navigation using Partially Observable Markov Decision Processes

Arnaud Ruymaekers

Supervisor: Alessandro Verri Reviewer: Lorenzo Rosasco

External Supervisor: Agnese Seminara

March, 2024

Abstract

This thesis aims to investigate solutions to the olfactory navigation problem utilizing Model-based Reinforcement Learning. In this prospect, the results of the paper by Rigolli N. et al. [9] are used as a benchmark. In ref [9], the authors use Point-Based Value Iteration to solve the olfactory navigation problem formulated as a Partially-Observable Markov Decision Process. We validate our results by comparing them to those observed in [9]. However, we found considerable variation in the performance when doing more extensive testing. Yet, the qualitative behavior from [9] is widely conserved despite considerable variations in performance. This more thorough testing is performed thanks to a new implementation of Point-Based Value Iteration in Python, optimized to use multithreading and add support to the GPU. More comprehensive testing has been performed with a test where a larger starting area is being tested rather than 3 points. We also investigate the possible causes of the significant variations in performance and propose areas of potential improvement.

Table of Contents

Chapter 1

Introduction

How do animals use their sense of smell to find food? In nature, odor cues do not travel in the air or water predictably; it is a turbulent environment. So, how can animals find the source of an odor so easily in practice? How does their brain work? The field of olfactory navigation attempts to answer this question.

Do animals have to learn to use their sense of scent, or is it something they inherently know? Can animals plan how to navigate by olfaction, or do they react directly to what they smell and change their strategy accordingly? Two different paradigms for reinforcement learning are known depending on whether prior information on the odor cue is available.

The study of olfactory navigation consists of studying how an agent or a group of agents attempt to find the source of an odor using only their sense of smell. At macroscopic scales, the odor travels in a turbulent fashion; it is hard to predict precisely. Vergassola M. et al. [16] developed an algorithm, "Infotaxis", where an agent moves towards the most significant information gain. More recently, however, model-based reinforcement learning was used to attempt to solve this problem more efficiently.

In this thesis, we focus on single-agent navigation. In particular, following the work of Rigolli N. et al. [9], we study an agent that can follow scent trails on the ground or sniff the air while standing still (Figure 1.1). We ask what the optimal alternation is between these two sensory modalities. Olfactory navigation can be solved using reinforcement learning. We will assume that the agent knows the probability of detecting the odor, and thus, we focus on model-based reinforcement learning. In this case, we can imagine the animal can use the direction of the wind to help guide it in this task.

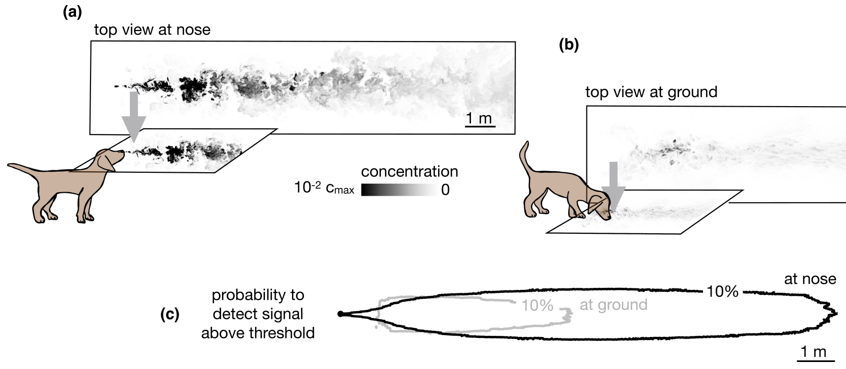


Figure 1.1: An illustration of an agent (here, a dog) able to smell odor cues in the air (a) and on the ground (b). The probability contour of a probability higher than 10% to smell anything is also shown (c). Figure from Rigolli N. et al. [9].

In the article by Rigolli N. et al. [9], the authors could reproduce a well-known biological behavior: casting and surging. Casting consists of scanning cross-wind to find some odor cues; then, during surging, the agent walks upwind.

A simple olfactory navigation problem can be formulated as a discrete Markov Decision Process (MDP). In a nutshell, an agent attempts to find the source of an odor by taking steps in its environment. As an MDP, the agent would have states representing its relative position to the source of the odor. More precisely, with the problem of olfactory navigation, the agent does not know where it is situated relative to the source. This uncertainty means we consider a more general case of Partially-Observable Markov Decision Process (POMDP), where the agent is unaware of its state. In olfactory navigation, the agent can, however, smell to gather information about the position of the odor source; in terms of POMDPs, this translates to the agent making an observation.

In the paper by Rigolli N. et al. [9], the authors implemented the Point-Based Value Iteration (PBVI) algorithm to find a solution to the POMDP formulation of the olfactory navigation problem. The PBVI algorithm is a modified version of the Value Iteration algorithm for MDPs [7]. In this thesis, we implemented a new version of PBVI in Python. This new version was built as a framework that can be used on various POMDP problems. Different flavors of the PBVI algorithm, such as Perseus [15], Heuristic-Search Value Iteration (HSVI) [14], and Forward-Search Value Iteration (FSVI) [12], are implemented within the framework. The implementation of this thesis also supports multithreading and GPU operations, considerably speeding up the solving process and simulations.

We confront the results of the implementation of this thesis with the ones of the Rigolli N. et al. [9] paper. We successfully reproduced the article's results, confirming our implementation is correct.

Thanks to the considerable speedup of this thesis's implementation, we were able to run more extensive experiments. These experiments highlight that the performance measure used in [9] only represents part of the picture of how an agent could perform under real circumstances. Additionally, we observe that the performance varies significantly between algorithm runs. The implementation is currently being used to comprehensively explore the efficiency of this model-based search strategy as a function of the environment and of the prior information.

First, we will describe the state-of-the-art concepts for optimal decision-making (Chapter 2). In this chapter, the notions of MDP and POMDP will be described, and how they can be solved. For MDPs, we will describe what a policy and value function are. We will then see the algorithms of value iteration and policy iteration and how they come to a stop once convergence is reached. Then, we will redefine policies and value functions in the context of POMDPs along with the concept of "belief". Following this, we will see how the value iteration algorithm is adapted to partial observability with a point-based version. To limit the size of the value function, we will describe how a pruning operation can be performed. Lastly, a small experiment will illustrate the transition from an MDP to a POMDP solution.

Then, the Python framework implementation will be explained (Chapter 3). Here, the various components that comprise the code base will be listed. A concept of "reachability" will be defined; this concept is used to speed up the various operations of the PBVI algorithm. Following this, the different ways the operations have been optimized in the code will be detailed; this includes the various precomputations performed, the vectorization of core functions, and how simulations are parallelized. An analysis of the computational complexity of the optimized function will be performed. To conclude this chapter, the code will be applied to a well-known POMDP problem to validate the correctness of the implementation.

Finally, the olfactory navigation will be defined as a POMDP within this paper's framework (Chapter 4). We will reproduce the results of the Rigolli N. et al. [9] paper. Then, further experimentation will be performed to explore the limits of the Point-Based Value Iteration algorithm for the olfactory navigation problem. Furthermore, an analysis of the runtimes of this thesis's implementation will be done.

This thesis was performed as an external thesis within the Physics informed Machine Learning for biological Behavior (PiMLB) unit of the Machine Learning Genoa (Malga) Center. The PimLB unit works on studying biological behaviors by combining the fields of physics and machine learning.

Chapter 2

Optimal Decision Making

This section will describe the theory used throughout this paper. In particular, the concepts of Markov Decision Processes and Partially Observable Markov Decision Processes will be defined, along with how optimal solutions can be found for them.

First, the theory of the fully observable version of Markov Decision Processes will be defined, and how they can be solved using Value Iteration and Value Functions (Section 2.1). From this, the theory of the partially observable version will be derived, along with the concept of beliefs and the algorithm of Point-Based Value Iteration and its different flavors (Section 2.2). Finally, we will illustrate the transition from the Markov Decision Problem in a fully observable setting to a partially observable setting and the implications of this transition (Section 2.3).

2.1 Markov Decision Processes

Markov Decision Processes (MDP) represent a decision-making process in which an agent aims to maximize its cumulative reward over time. Solving an MDP equates to finding an optimal strategy the agent can use to reach this goal.

First, we will formally define an MDP and its components (Section 2.1.1). Then, we will define how the agent's strategy is represented in the context of an MDP (Section 2.1.2). Finally, the two methods by which this strategy can be learned will be described (Section 2.1.3).

2.1.1 Formal Definition of an MDP

A Markov Decision Process (MDP) is a way to model a process in which an agent tries to maximize its rewards by taking actions in its environment. An agent can take *actions* that will affect its *state* in the environment and receive *rewards* (Figure 2.1). An MDP is a Markovian process because the next state is a function of the previous state and the action; all states and actions before are unrelated.

The agent can receive rewards in different circumstances. For example, we can imagine the agent receiving a reward when arriving in a particular state. The agent could also receive rewards based on what actions it takes or with a combination of both. Reward values can also be negative, in which case they act as penalties. The notion of *expected rewards* unifies how a reward can be received in different problems. The expected rewards are the rewards the agent can expect to receive by taking a certain action from a certain state.

The environment can have some uncertainty built into it; in such cases, when the agent takes some action from a state, it could arrive in a set of new states based on some probabilities. These probabilities are the so-called transition probabilities of the MDP.

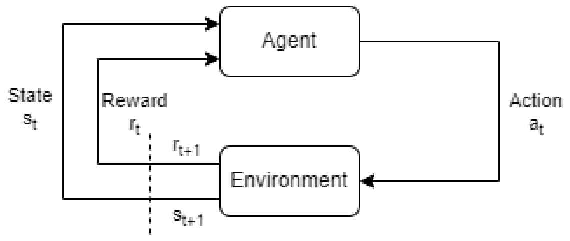


Figure 2.1: MDP Cycle

Formally, we can define a MDP as a tuple $< S, A, T, R >$, where we have:

- S: The set of states defining the problem.
- A: The set of actions the agent can take.
- T: The transition function. It can come in the form of a matrix of size $|S|$ by $|A|$ by $|S|$ representing the probabilities for the agent to arrive in any state s' after taking any action a from any state s. It is as conditional probability: $T(s, a, s') = P(s'|s, a)$. The probabilities of arriving in any state s' must sum to one for each state-action pair: $\sum_{s' \in S} P(s'|s, a) = 1, \forall s \in S, \forall a \in A$.
- R: The reward function of the problem. As stated previously, it encompasses what reward is expected to be received on average when taking actions a from state s. It can be any value but should be adequately selected

to represent the properties of the problem; for example, if taking action a_1 leads to victory and action a_2 leads to an immediate loss, the rewards should be completely different, say $(10, -10)$ rather than $(10, 9)$. These expected rewards can be computed from immediate rewards. Say the immediate rewards are rewards associated with each triple $< s, a, s' >$ and represented by the function $R^i(s, a, s')$; we can compute the expected rewards using the transition function T:

$$R(s, a) = \sum_{s'} R^i(s, a, s')T(s, a, s'), \forall s \in S, \forall a \in A \tag{2.1}$$

In MDP problems, the agent is all-knowing; it always knows its current state. The model can have uncertainty; for example, doing an action can lead to multiple states with some probabilities, or the reward received can be probabilistic. However, since the agent is all-knowing, it also knows the probabilities of the transitions and the rewards.

Solving MDPs is known as Model-Based Reinforcement Learning because the agent can access a model of the environment. The model is encapsulated in the states, actions, and transition and reward functions. The goal of finding the optimal strategy to maximize the expected rewards is known as planning. In contrast to Model-Based, in Model-Free Reinforcement Learning, the agent must learn the environment on top of the actions to take; the agent has no prior knowledge of its environment and must, therefore, learn by experience only. While in Model-Based RL, the agent is trained ahead to get a policy (planning), in Model-Free RL, the agent reacts to its environment and updates its policy during the simulation.

2.1.2 Value Function and Policies

An essential part of the Markov Decision Process is for the agent to be able to react to its environment. It must be able to choose an appropriate action when in a given state. The agent chooses an action following a policy π. A policy is a set of instructions telling the agent to make a certain action when in a given state. A policy is a function mapping states to actions:

$$\pi : S \rightarrow A \tag{2.2}$$

A policy can be evaluated by taking the sum of the rewards over time. However, a policy leading to a reward at a given time is more valuable than one leading to the same reward but ten timesteps later. Therefore, a better measure of the quality of a policy is a discounted sum of the rewards received by the agent. The discount factor is a parameter $\gamma \in [0,1]$. If $\gamma = 0$, only the immediate rewards matter. An optimal policy π^* is a policy that maximizes the expected sum of discounted rewards over an infinitely long time series:

$$E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t)) | \pi, s_0\right] \tag{2.3}$$

Another concept we can define is a value function that maps states to values:

$$V : S \to \Re \tag{2.4}$$

A value function can be used to evaluate a policy π at a state s using the estimated discounted reward (Equation 2.3) with $s_0 = s$.

$$V(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t)) | \pi, s_0 = s\right] \tag{2.5}$$

An alternative way to formulate the value function is through a concept of a quality function Q. The quality function, or Q function, maps states and actions to values. It is mainly used for the Model-Free technique of Q-learning. In Q-learning, a Q-table is used to compile the values of the state-action pairs. With this definition of a Q function, we can decompose the value function (Equation 2.5).

$$Q : S \to \Re \tag{2.6}$$
$$V(s) = \max_a Q(s, a) \tag{2.7}$$

From the Q function, we can build a Q-table that compiles the values of each state-action pair. For a 2-state, 2-action problem, this table looks as follows:

	s_0	s_1
a_0	$Q(s_0, a_0)$	$Q(s_1, a_0)$
a_1	$Q(s_0, a_1)$	$Q(s_1, a_1)$

Table 2.1: Q-table of a 2-state, 2-action problem

Although Q-learning will not be further developed in this thesis, the concept of Q-table and assigning a value to a state-action pair will be convenient when defining value functions for the partially observable case.

2.1.3 Value Iteration and Policy Iteration

From the concepts of policies and value functions, we now explore how a solution to an MDP is found. The techniques of value iteration and policy iteration will be described. With these techniques, an optimal policy can be computed.

First, we can reformulate the value function equation (Equation 2.5), which uses an estimation of the discounted reward for a given state to be a recursive version of itself instead. We can use the value function V to compute a new value function V'. A sum of V, weighted by the transition probabilities at each possible next state, by the discount factor is added to the immediate reward to make the value V'.

$$V'(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s')V(s') \tag{2.8}$$

Value iteration aims to define the value function as a recursive version of itself. This recursive formulation of the value function is called the Bellman update equation. As the name suggests, the value function is updated iteratively to aim towards optimality. Also, instead of relying on a policy to choose the action, the value is computed for each action, and the highest value is retained. The next value function equation (Equation 2.8) is therefore reformulated as follows:

$$JV(s) = \max_{a \in A} \left(R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')V(s') \right) \tag{2.9}$$

The J is a Bellman operator, signifying the operation is iterated until convergence. In other words, we apply a Bellman update until reaching a satisfactory point, in which case we consider the value function to be optimal, denoted by V^*:

$$V^*(s) = \max_{a \in A} \left(R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')V^*(s') \right) \tag{2.10}$$

The value iteration process, therefore, iteratively updates the value function. The value function is updated using the Bellman update equation (Equation 2.9) for each state $s \in S$. The value function is typically initialized as the expected reward for each state. The iteration runs until some convergence criteria are reached.

An alternative technique to solve an MDP is policy iteration. Instead of iterating the value for each state, we consider the whole policy. We take an initial policy, either at random or a greedy policy (maximizing expected rewards), evaluate this policy by computing its quality value Q_π using the estimated discounted reward (Equation 2.3), and update the policy based on the evaluation:

$$\pi'(s) \leftarrow \underset{a \in A}{\operatorname{argmax}}\, Q^\pi(s, a) \tag{2.11}$$

Both techniques have their advantages and disadvantages. Value iteration tends to converge slower than policy iteration. However, policy iteration can become unfeasible to compute on large state spaces because each new policy needs to be evaluated in its entirety.

2.1.3.1 Convergence

The value iteration process leads to an increasingly refined solution throughout the iterations. A parameter ϵ can be defined as controlling what rate of change we consider acceptable for the solution. The maximum acceptable change uses the discount factor γ in combination with ϵ. With γ, the closer it gets to 1, the lower ϵ should be chosen for an identical acceptable change.

$$max\_rate = \epsilon \left(\frac{\gamma}{1-\gamma} \right) \tag{2.12}$$

We consider the change between values functions (V_t and V_{t+1}) at two different iterations to be the maximum change of their components.

$$change = \max_{s \in S} |V_{t+1}(s) - V_t(s)| \tag{2.13}$$

The convergence check is then defined as follows:

$$change < max\_rate \tag{2.14}$$

2.2 Partially-Observable Markov Decision Processes

Partially Observable Markov Decision Processes (POMDPs) are a generalization of MDPs where the agent is not all-knowing anymore. The agent does not have full knowledge of its state in the environment; it is hidden from him. To mediate this uncertainty about its state, the agent can make observations about its environment (Figure 2.2). In some cases, the agent can observe its current state directly, with some uncertainty. Alternatively, it can observe something completely arbitrary that gives some information about its state. Based on the agent's observations, it has a belief about its current state in the environment.

Solving POMDPs is still considered Model-Based reinforcement learning as the agent is still given a model of its environment. Although there will be uncertainty in which state the agent is in, the agent is still aware of the transition probabilities between states. The probability distributions for the observations are also given. These probability distributions make up the agent's model of the environment.

For example, we can imagine a problem where an agent has to navigate in a dark room about which it has some prior knowledge. In this problem, the agent will not know its precise position in space. However, it could be given the ability to sense (i.e., observe) whether a wall is directly adjacent to it. The goal could be to find the exit, which it must be able to find relying on its observations and prior knowledge of the room.

Note: In the particular case where the agent can observe its current state with no randomness, it holds that we are in the case of an MDP problem.

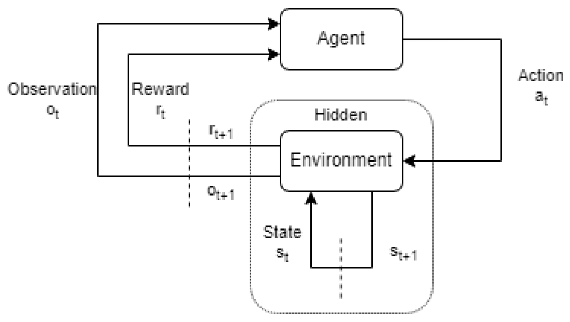


Figure 2.2: POMDP cycle

2.2.1 Formal Definition of a POMDP

Formally, a POMDP is defined by the tuple $< S, A, T, R, \Omega, O, b_0 >$ where the S, A, T, R components are defined as for the MDP (Section 2.1.1) and the others are as follows:

- Ω: The set of possible observations the agent can make about the environment.

- O: The observation function of the problem. It represents the probabilities for the agent to observe an observation o[1] given it has performed action a and reached state s', formally: $O(a, s', o) = P(o|a, s')$. Same as for the transition function, but for each state-action pair, the sum observation probabilities must equal 1: $\sum_{o \in \Omega} O(a, s', o) = 1, \forall a \in A, \forall s' \in S$.
- b_0: The initial belief of the agent. Typically, it is a uniform probability distribution over all the possible states the agent can start in. It serves as a way to give prior knowledge to the agent about where it can start in its environment.

Rewards could also be defined as received by the agent when it makes some observation. The immediate reward function then becomes a function taking four arguments: $R^i(s, a, s', o)$. This fourth parameter indicates that the expected reward function must also use the observation function to be calculated from the immediate reward function R^i. Hence, we modify how the expected rewards are computed (Equation 2.1), which gives:

$$R(s, a) = \sum_{s' \in S, o \in \Omega} R^i(s, a, s', o) T(s, a, s') O(a, s', o), \forall s \in S, \forall a \in A \tag{2.15}$$

2.2.2 Beliefs

To account for the agent's uncertainty about its current state in a partially observable setting, we introduce the concept of beliefs. Beliefs are probability distributions over the states, representing the agent's belief of its state in the environment. The probabilities of being in each state in a given belief are represented as $b(s) = P(o|h)$, where h is the sequence of actions taken and observations received by the agent thus far. It is a probability distribution, so it holds that $\sum_{s \in S} b(s) = 1$.

At any instant in the simulation process, the agent has a belief b about its state. For the agent to update this belief b, after it has taken action a and received an observation o, it can use the transition function T and the observation function O. This will result in a belief $b^{a,o}$ representing its new belief about its state.

$$b^{a,o}(s') = \frac{\sum_{s \in S} O(a, s', o) T(s, a, s') b(s)}{P(o|b, a)} \tag{2.16}$$

[1]Individual observations are often denoted by z in literature, but in this paper, it will be denoted as o to make it clear we are speaking about observations.

The denominator of the belief update (Equation 2.16) represents a normalization term. It ensures the belief vector $b^{a,o}$ still sums to 1, i.e., $\sum_{s \in S} b^{a,o}(s) = 1$. It represents the agent's probability of observing o while performing action a in belief b. This observation probability can be computed as follows:

$$P(o|b,a) = \sum_{s \in S} b(s) \sum_{s' \in S} O(a,s',o) T(s,a,s') \tag{2.17}$$

Note: The observation probability in a certain belief (Equation 2.17) can be computed once and cached for every s' in the belief update (Equation 2.16).

In this new setting, we can also reformulate the expected rewards for belief points instead of states. Therefore, the expected rewards (Equation 2.15) can be used as a weighted sum over each state $s \in S$ of that state's expected reward by the agent's belief to be in said state.

$$R(b,a) = \sum_{s \in S} b(s) R(s,a) \tag{2.18}$$

2.2.3 Value Function and Policy

Considering partial observability, the agent has to use its belief about the environment to make decisions. Therefore, in the context of a POMDP, a policy maps beliefs to actions instead:

$$\pi : B \to A \tag{2.19}$$

, where B is the set of all possible belief points. As stated above, beliefs are points in belief space represented by a simplex of dimensionality $|S|$. Following the different mapping of the policy, the estimated discounted rewards (Equation 2.3) needs to be rewritten in terms of belief points instead of states:

$$E\left[\sum_{t=0}^{\infty} \gamma^t R(b_t, \pi(b_t)) | \pi, b_0\right] \tag{2.20}$$

, with the reward function based on beliefs (Equation 2.18).

It is useful to consider the Q values to define value functions in terms of belief points instead of state points. Belief points are points in the continuous space between the points of certainty (the actual states). We can imagine connecting, for each action, the Q values (Equation 2.6) at each state by a plane. These planes would, therefore, be in belief space. These planes are called alpha planes (or alpha vectors). Each row is one alpha plane from the Q-table (Table 2.1).

With some arbitrary values, for a 2-state, 2-action problem, we can plot the planes associated with each action.

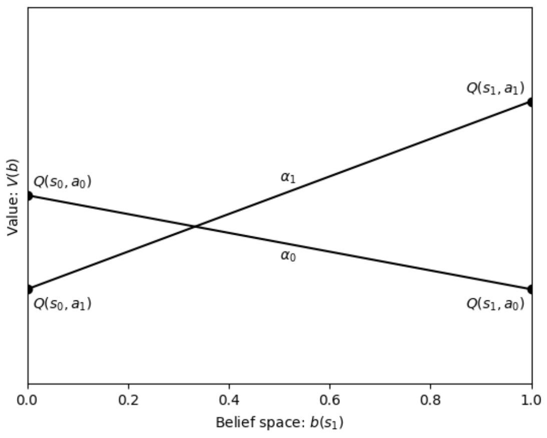


Figure 2.3: Example value function in belief space of a 2-state, 2-action problem

Here the alpha plane α_0, associated with action a_0. It is written as: $\alpha_0 = (Q(s_0, a_0), Q(s_1, a_0))$. These values are the quality values of the row of the action a_0 in the Q-table (Table 2.1).

In this plot, the x-axis is the belief space. In particular, the x-axis represents the amount of certainty to be in state s_1. Therefore, an agent's confidence to be in state s_0 is $b(s_0) = 1 - b(s_1)$. In the 2-state example problem, the belief $b(s_1)$ of 1.0 expresses the agent's certainty of being in state s_1. The belief point for this would look as follows: $b = \{0.0, 1.0\}$. Moreover, a belief of $b = \{0.3, 0.7\}$ represents the agent being 70% sure to be in state s_1 and, therefore, not ruling out the possibility of being in state s_0.

We can redefine the value function for POMDPs as a set of alpha-planes: $V : \alpha_i \forall i \in n$. To compute the value at belief point b, we can reformulate the value function equation (Equation 2.7) to maximize over the alpha planes instead of the actions:

$$\begin{aligned} V(b) &= \max_{\alpha \in V} \sum_{s \in S} b(s)\alpha(s) \\ &= \max_{\alpha \in V} b \cdot \alpha \end{aligned} \tag{2.21}$$

Graphically, it looks as follows:

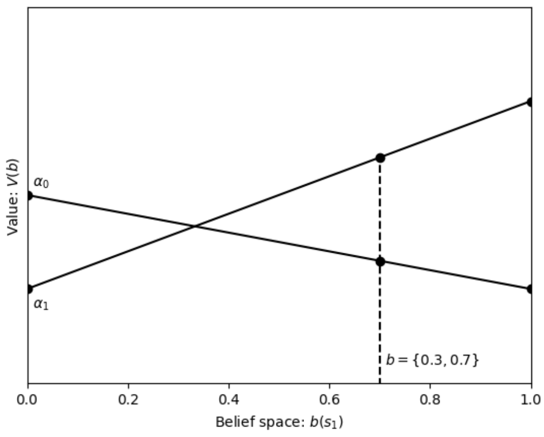


Figure 2.4: Example value function - belief value computation

Like in full observability, a policy can be defined by taking the action associated with the highest value over the belief space. However, it might not be ideal to store the policy as it is now in a continuous space and not anymore in individual states. So it is best to find the best action to play at a given belief based on the value function using:

$$\begin{aligned} \pi(b) &= \underset{\alpha \in V}{\text{argmax}} \sum_{s \in S} b(s)\alpha(s) \\ &= \underset{\alpha \in V}{\text{argmax}}\, b \cdot \alpha \end{aligned} \tag{2.22}$$

Note: each α plane is associated with a single action, so finding the best α plane equates to finding the best action.

We can visualize the policy for a given value function as follows:

With the policy of this example value function, we see that when the agent is a belief to be in state s_1 in $[0.0, 0.333]$, the agent should take action a_0 and a_1 when having the belief to be in state s_1 in $[0.666, 1.0]$.

So far, we have considered a value function comprising two alpha planes. However, it may come up that some third action is preferred when the agent is in a given state of uncertainty. For this, additional alpha vectors must be added to the value function. The iterative addition of alpha planes leads to a convex and piecewise continuous surface. The following section will explore the iterative addition of alpha planes to the value function.

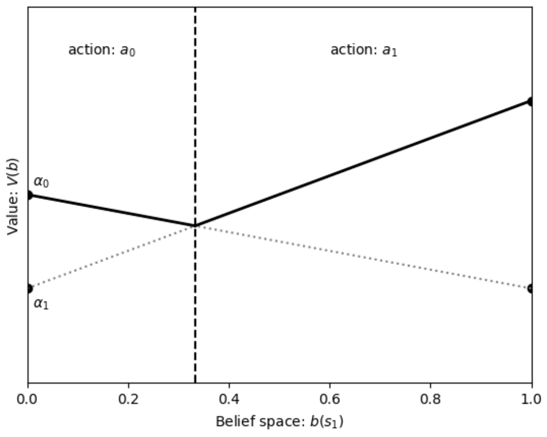


Figure 2.5: Example value function - action policy

We can also state that the value function of a POMDP will always be lower than the value function of the equivalent MDP model (so, where the observations are suppressed); this is because some value is lost with the uncertainty of observation. This statement will be proved by a simple experiment in Section 2.3. It also implies that the MDP value function of POMDP problems can be used as an upper bound and, therefore, be a metric of how close the POMDP value function is to optimality.

2.2.4 Point-Based Value Iteration

Adapting the solving methods for MDPs to POMDPs requires considering the value function maps a continuous space of beliefs to values. As a direct consequence, the policy evaluation step of policy iteration does not make sense due to the complexity of the evaluation over a continuous space. However, value iteration can more easily be adapted to work with belief points instead of state points. Considering individual belief points for the value iteration process gives rise to *Point-Based* Value Iteration (PBVI).

While the process of value iteration for MDPs leads to an exact solution, PBVI is an approximation as it improves the value function at a given set of belief points. If we wanted to solve POMDPs exactly, we would need to consider the entire space of beliefs, which would be infeasible.

We can, thus, modify the Bellman update equation (Equation 2.8) to work with beliefs instead of states:

$$V'(b) = \max_{a \in A} \left(R(b,a) + \gamma \sum_{o \in \Omega} P(o|b,a) V(b^{a,o}) \right) \tag{2.23}$$

Updating the value function is known in the literature as either the "update" or the "backup" function; this paper will refer to it as the backup function. Its role is to update the value function based on belief points.

Then, another question comes up: How do we choose which belief points to update the value function with? Generating new belief points is the task of the "explore" or "expand" function. This paper will refer to this function as the expand function.

Initially defined by Pineau et al. [7], Point-Based Value Iteration is a method to approximate the value function with a set of belief points. First, we generate new belief points based on previous ones (guided by some heuristic) (the *expand* step). Then, using these new belief points, we update the value function (the *backup* step). These two steps are run iteratively until a convergence criterion is reached. We start the process with a set of beliefs consisting only of the initial belief b_0. The value function is initialized using the expected rewards (Equation 2.15). These steps are summarized in the following procedure:

Algorithm 2.1 PBVI framework

```
$V \leftarrow R$
$B \leftarrow \{b_0\}$
while not Converged do
    $B \leftarrow Expand(B, V)$
    $V \leftarrow Backup(B, V)$
end while
```

Returning to the example value function for a 2-state, 2-action problem (Figure 2.3). We can say that in this problem, the agent can make one of two observations. From this example, we can illustrate the PBVI process as defined in Algorithm 2.1:

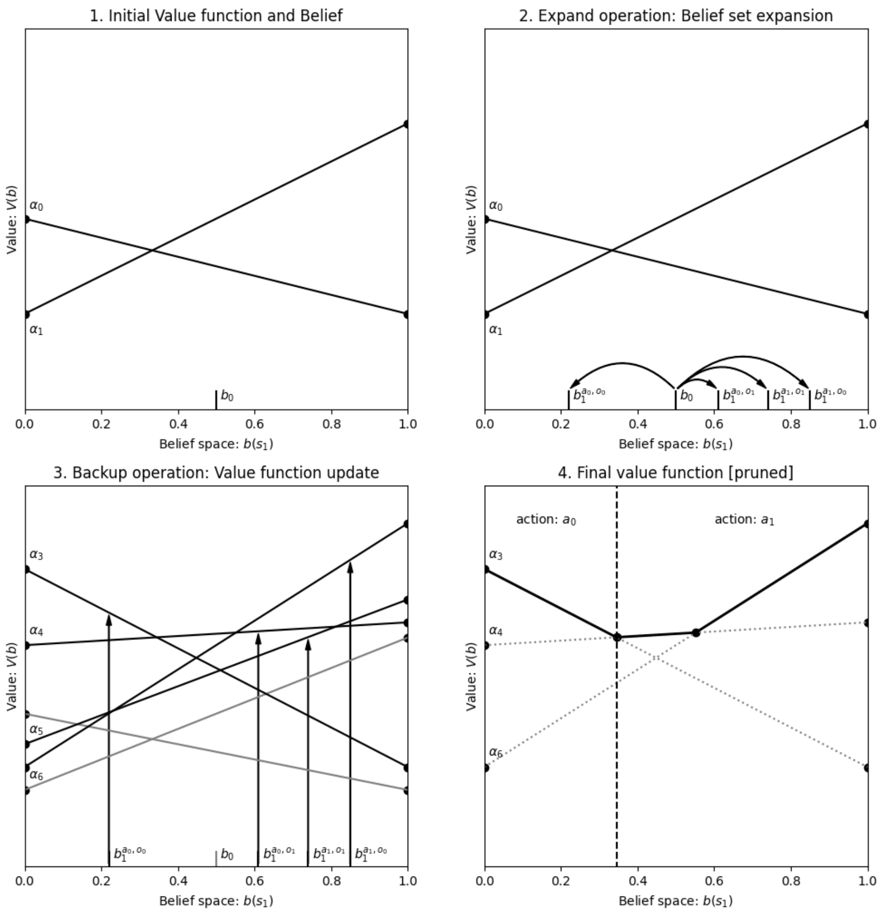


Figure 2.6: PBVI Steps

In this example (Figure 2.6), from the initial belief b_0, at most, four new beliefs can be generated. However, less could be chosen during each iteration depending on the belief exploration strategy. We will define various exploration strategies in the PBVI Expand Function section (Section 2.2.6). Then, the value function is updated at each belief point, and a new alpha plane is generated for each point. The update function will be described in the PBVI Backup Function section (Section 2.2.5). Finally, to avoid filling the value function with alpha planes that are not useful, we apply a pruning step. This pruning step will be further described in Section 2.2.7.

2.2.5 PBVI Backup Function

In this function, we shall expand on what is involved in the computation of the update step of the value function. We can expand on the Bellman update equation (Equation 2.23) by making use of components of the POMDP model and previously defined concepts:

$$
\begin{aligned}
V'(b) &= \max_{a \in A} \left(R(b,a) + \gamma \sum_{o \in \Omega} P(o|b,a) V(b^{a,o}) \right) && (2.24) \\
&= \max_{a \in A} \left(\left(\sum_{s \in S} b(s) R(s,a) \right) + \gamma \sum_{o \in \Omega} P(o|b,a) V(b^{a,o}) \right) && (2.25) \\
&= \max_{a \in A} \left(\left(\sum_{s \in S} b(s) R(s,a) \right) + \gamma \sum_{o \in \Omega} P(o|b,a) \max_{\alpha \in V} \sum_{s' \in S} b^{a,o}(s') \alpha(s') \right) && (2.26) \\
&= \max_{a \in A} \Bigg(\left(\sum_{s \in S} b(s) R(s,a) \right) \\
&\qquad + \gamma \sum_{o \in \Omega} P(o|b,a) \max_{\alpha \in V} \sum_{s' \in S} \sum_{s \in S} \frac{O(a,s',o) T(s,a,s') b(s)}{P(o|b,a)} \alpha(s') \Bigg) && (2.27) \\
&= \max_{a \in A} \Bigg(\left(\sum_{s \in S} b(s) R(s,a) \right) \\
&\qquad + \gamma \sum_{o \in \Omega} \max_{\alpha \in V} \sum_{s' \in S} \sum_{s \in S} O(a,s',o) T(s,a,s') b(s) \alpha(s') \Bigg) && (2.28)
\end{aligned}
$$

Starting from the Bellman update Equation 2.23, we can first replace the reward on the belief with the expected reward weighted by the belief at each state to use the expected rewards $R(s,a)$ (Equation 2.25). Then, we can use the definition of the value function in POMDPs (Equation 2.21), where it is defined in terms of alpha planes to compute and decompose the $V(b^{a,o})$ statement (Equation 2.26). Following this, we can use the belief update (Equation 2.16) to decompose $b^{a,o}$ (Equation 2.27). As we can see, this brings up a normalization term $P(o|b,a)$, and since this same term is used to multiply the value of the updated belief, it cancels out, leading to the final equation (Equation 2.28).

From this developed value function update equation, we can identify three steps:

1. Generation of a set of alpha planes for each previous alpha vector α_i in V,

action a, and observation o, which we can call this set of alpha planes as $\Gamma_i^{a,o}$:

$$\Gamma_i^{a,o} \leftarrow \alpha_i^{a,o}(s) = \gamma \sum_{s' \in S} O(a, s', o) T(s, a, s') \alpha(s') \tag{2.29}$$

2. Then, from this set of alpha planes $\Gamma_i^{a,o}$, for each belief b, we can choose the best alpha plane per observation o and sum these alpha planes together. We can sum each of these new alpha planes by the reward vector for the corresponding action. This results in a new set of alpha planes Γ_b^a (formulated as vector operations):

$$\Gamma_b^a \leftarrow \alpha_b^a = \{R(s,a), \forall s \in S\} + \sum_{o \in \Omega} \operatorname*{argmax}_{\alpha_i^{a,o} \in \Gamma_i^{a,o}} \alpha_i^{a,o} \cdot b \tag{2.30}$$

3. Then, reducing the set again, we can select the final alpha plane per belief b by taking the dot product of each alpha plane in Γ_b^a:

$$\alpha_b = \operatorname*{argmax}_{\alpha_b^a \in \Gamma_b^a} \alpha_b^a \cdot b \tag{2.31}$$

As we can see, the first step uses the previous value function to generate a new set of alpha vectors for each action-observation pair. It is the heavy operation of the backup operation. Given a value function V, the resulting set $\Gamma^{a,o}$ would be of size $|V| \times |A| \times |\Omega|$. The time required to compute this set would be $|S|^2 \times |V| \times |A| \times |\Omega|$.

Depending on the flavor of PBVI implementation, the value function update can be done on all the beliefs generated since the initialization, in which case we can call it a *full backup*. Otherwise, if only the newly generated belief points are used to update the value function, it shall be called *new points backup*.

To make sure not to lose information when running the value function update, especially with *new points backup*, we need to union the new alpha planes with the old ones to create the value function:

$$V' = V \cup \{\alpha_b, \forall b \in B\} \tag{2.32}$$

2.2.6 PBVI Expand Function

The goal of the belief set expansion is to explore the belief space by adding belief points to the belief set that will, in turn, be used by the backup function. From the initial belief point b_0, we can identify a belief expansion tree in which each belief generates a new belief for each possible action and observation combination.

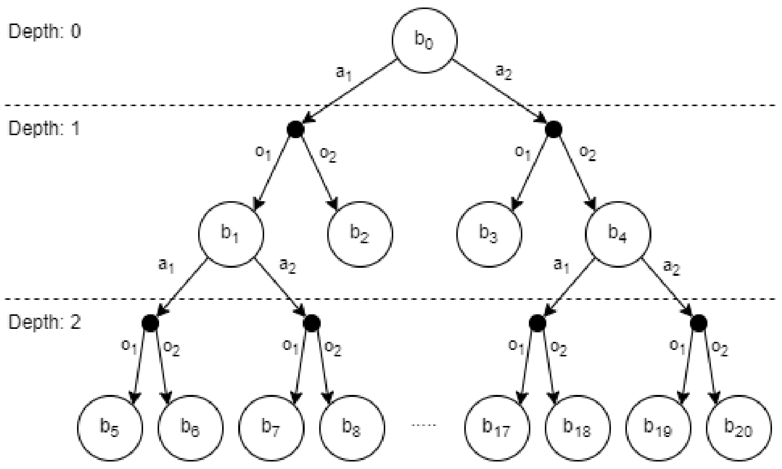


Figure 2.7: Belief set expansion tree

Figure 2.7 shows the belief expansion tree for a POMDP model with two actions and two observations. We can see that the number of belief points at each depth grows exponentially. The goal of the expand function is to limit the growth of the belief set by expanding to belief points worth exploring. Typically, the belief set's size doubles or increases by a maximum of N new belief points.

This section will describe various belief exploration techniques and explain their advantages and drawbacks.

2.2.6.1 Basic

In the original definition of PBVI [7], some basic ways to expand the belief set have been defined:

- Random belief selection (RA):

  This technique involves picking completely random points in belief space. It is a straightforward way of generating beliefs. The flaw of this technique is that the belief points it generates will most likely never be encountered in actual simulations.
- Stochastic Simulation with Random Action (SSRA):

  In this technique, a random action a is chosen for each belief already in the set. Following this, a random observation o is selected. We then add the updated beliefs $b^{a,o}$ to the belief set. Here, beliefs added are indeed reachable from other beliefs. However, some actions that, given a current belief, would not make sense to pick can be chosen.

- Stochastic Simulation with Greedy Action (SSGA):

  Finally, we can use a concept that is already well-known in the field of reinforcement learning, namely Epsilon Greedy action selection. Instead of selecting actions randomly for each previous belief point, we select an action following the policy learned so far (the greedy action) with a chance of epsilon; otherwise, we choose a random action. This technique has the potential to scale better to models with higher state counts depending on how it is configured.

The techniques explored so far are very basic, easy to implement, and cheap computationally, but they have some apparent flaws. The belief set expansion is guided mainly by randomness. So, the following techniques attempt to reduce this randomness by guiding the search using some heuristic.

2.2.6.2 Stochastic Simulation of Exploratory Action (SSEA)

As the first technique using a heuristic to generate beliefs, we consider Stochastic Simulation of Exploratory Action (SSEA). SSEA, described by Pineau J. et al. [7], chooses belief points to explore the belief space as much as possible.

To do this, we first have to generate all successor beliefs for each belief already present in the set. This means that for each action a and observation o, we create an updated belief $b^{a,o}$ (Equation 2.16). Then, we select the belief maximizing the distance from any other belief point already present in the set:

$$b_{new} = \underset{b^{a,o} \forall a \in A, \forall o \in \Omega}{\text{argmax}} \; \min_{b \in B} \sqrt{\sum_{s \in S} (b(s) - b^{a,o}(s))^2} \tag{2.33}$$

This operation is costly, and as the belief set B grows, the cost of the operation grows rapidly. To select N new belief points, it would require $N \times |B| \times |A| \times |\Omega|$ checks and a computational time of $N \times |B| \times |A| \times |\Omega| \times |S|$. However, the trade-off for the higher computational cost is the guarantee that the belief points added to the belief set offer increasingly better coverage of the belief space.

2.2.6.3 Greedy Error Reduction (GER)

The Greedy Error Reduction (GER) technique was introduced in a follow-up paper by Pineau J. et al. [8]. GER consists of choosing belief points that will most improve the value function. The error between two belief points b, and b' is computed to be the difference between the belief points times the difference between the alpha plane associated with the belief point b and either the max or the min discounted reward, and such at each state and summed together. Formally, this gives:

$$\epsilon(b') = \sum_{s \in S} \begin{cases} (\frac{R_{max}}{1-\gamma} - \alpha(s))(b'(s) - b(s)), \text{ if } b'(s) \geq b(s) \\ (\frac{R_{min}}{1-\gamma} - \alpha(s))(b'(s) - b(s)), \text{ if } b'(s) < b(s) \end{cases} \tag{2.34}$$

, where α is the plane in the value function V such that $\alpha = \operatorname{argmax}_{\alpha \in V} \alpha \cdot b$.

To pick new belief points, we first generate all the candidate belief points as the successors from belief points in the belief set B. Then, we choose an initial belief b from B and an action that maximizes:

$$\tilde{b}, \tilde{a} = \operatorname*{argmax}_{b \in B, a \in A} \sum_{o \in \Omega} \left(\epsilon(b^{a,o}) \sum_{s \in S} \sum_{s' \in S} T(s,a,s') O(a,s',o) b(s) \right) \tag{2.35}$$

Finally, given the found belief b and action a, we select the observation o that maximizes the error bound reduction:

$$\tilde{o} = \operatorname*{argmax}_{o \in \Omega} \left(\epsilon(\tilde{b}^{\tilde{a},o}) \sum_{s \in S} \sum_{s' \in S} T(s,\tilde{a},s') O(\tilde{a},s',o) b(s) \right) \tag{2.36}$$

The belief $\tilde{b}^{\tilde{a},\tilde{o}}$ is then added to the belief set. This process is done iteratively until a number n of new belief points are added to the belief set.

This technique has some good potential but works well only on models with small state spaces due to its computational intensity. For this reason, other techniques using simpler heuristics perform better on larger state spaces.

2.2.6.4 Perseus

Spaan M. and Vlassis N. [15] developed the Perseus algorithm to tackle larger problems more efficiently. The exploration strategy consists of generating sequences of beliefs in a random fashion. The difference with the Stochastic Simulation with Random Action is that the states are not used with this technique. Instead, the observation is chosen according to the probability $P(o|b,a)$. The action a, however, is chosen at random. The sequence of beliefs starts every time with the starting belief b_0. A parameter n controls the amount of beliefs generated.

Algorithm 2.2 Perseus - Random Explore function

```
1: function RANDOM_EXPLORE( $n$ )
2:     $B \leftarrow \emptyset$
3:     $b \leftarrow b_0$
4:     repeat
5:         Sample $a$ from $A$
6:         Sample $o$ from $P(o|b,a)$
7:         Sample $o$ from $O(a, s', *)$
8:         $b \leftarrow b^{a,o}$
9:         $B \leftarrow B \cup \{b\}$
10:    until $|B| = n$
11: end function
```

The update function runs only on the latest belief set generated.

Compared to previously defined ones, the advantage of this technique is that the beliefs are explored in a depth-first search fashion. However, the beliefs generated follow a sequence of actions that do not guarantee optimal performance, as in some cases, taking a particular action would never happen in practice. Some heuristics could be introduced to choose the action to be taken during the belief generation more wisely to generate belief points more likely to be encountered in practice.

2.2.6.5 Heuristic Search Value Iteration (HSVI)

Heuristic Search Value Iteration (Smith, T. and Simmons, R. [14]) attempts to minimize the distance between a lower and an upper bound at given belief points. The upper bound is the solution of the fully observable version of the problem, while the lower bound starts as the expected rewards for each action. Throughout the process, the lower bound is refined using the backup function (Section 2.2.5), and the upper bound is refined using a method called the "Sawtooth" method (Hauskrecht, M. [4]).

The Sawtooth method consists of computing the value at a given belief point and adding this to a belief-value mapping dictionary V_{map}. The value of a given belief is computed by taking the minimum weighted sum of the values at each belief point in this dictionary. In other words, the computed value at the given belief point is saved to aid in finding the value at new belief points later on. The value at a given belief point can then be queried as follows:

Algorithm 2.3 Sawtooth algorithm

```
1: function SAWTOOTH_GET_VALUE( $b$ )
2:     if $b \in V_{map}$ then return $V_{map}(b)$
3:     end if
4:     $min\_val \leftarrow \sum_{s \in S} b(s) \cdot V^*_{MDP}$
5:     for $\langle b_i, v_i \rangle \in V_{map}$ do
6:         $val \leftarrow \left(v_i - \sum_{s \in S} b(s) \cdot V^*_{MDP}\right) \left(\min_{s \in S: b_i(s) > 0} \frac{b(s)}{b_i(s)}\right)$
7:         if $val < min\_val$ then
8:             $min\_val \leftarrow val$
9:         end if
10:    end for
11:    $V_{map}(b) \leftarrow min\_val$
       return $min\_val$
12: end function
```

The action is chosen to maximize the value of the upper bound. Then, we choose the observation with which the gap between the upper and lower bounds is the largest, meaning there is the most potential to improve the bound. The bound for a given belief b at iteration t can be computed with the excess function defined as follows:

$$excess(b, t) = \left(\overline{V}(b) - \underline{V}(b)\right) - \frac{\epsilon}{\gamma^t} \tag{2.37}$$

, with $\overline{V}$ being the upper bound of the value function and $\underline{V}$ being the lower bound. The excess is also computed with a parameter ϵ representing a number under which we consider the excess small enough. This ϵ number is also scaled by the discount factor γ to the t (the iteration number).

It was first developed to have the value function update steps interweaved with the belief exploration steps, but it can be modified to isolate the two sets of steps. Therefore, the exploration step would consist of choosing an action, adding a belief-value pair to the upper bound value function, and choosing an observation until a number n of beliefs has been generated.

With this technique, a simple stopping condition can be defined. When the maximum gap between the upper and lower bounds is lower than a parameter ϵ, we can consider that the algorithm has converged.

Algorithm 2.4 HSVI - HSVI Explore function

```
1: function BoundUncertaintyExplore( $b, t$ )
2:     if $excess(b, t) \leq 0$ or $t < n$ then
3:         $a^* \leftarrow \text{argmax}_{a \in A} Q_{\overline{V}(b,a)}$
4:         $o^* \leftarrow \text{argmax}_{o \in \Omega} P(o|b, a^*) excess(b^{a^*,o}, t)$
5:         $BoundUncertaintyExplore(b^{a^*,o^*}, t + 1)$
6:         $\overline{V}(b) \leftarrow J\overline{V}(b)$
7:     end if
8: end function
```

2.2.6.6 Forward Search Value Iteration (FSVI)

Forward Search Value Iteration (FSVI) (Shani, G. et al. [12]), like HSVI, uses the solution of the equivalent MDP problem. In this case, however, the solution is used to generate sequences of actions following this policy. Starting from a random state s_0, we choose actions recursively, leading to a new state and an observation until an end state is reached. With this sequence of actions and observations, we can update iteratively the belief points, starting from belief b_0, forming a set of new belief points. The recursion stops when either an end state is reached or the recursion depth reaches n, in which case enough belief points have been generated. From this, we can formalize a pseudocode for this recursive function:

Algorithm 2.5 FSVI - MDP Explore function

```
1: function MDP_Explore( $b, s, t$ )
2:     if $s \in EndStates$ or $t < n$ then
3:         return $\{b\}$
4:     else
5:         $a \leftarrow \pi_{mdp}(s)$
6:         Sample $s'$ from $T(s, a, *)$
7:         Sample $o$ from $O(a, s', *)$
8:         return $\{b\} \cup$ MDP_Explore($b^{a,o}, s', t + 1$)
9:     end if
10: end function
```

The FSVI expand function calls the MDP_Explore starting at a random state s_0 sampled from the starting probabilities and the belief b_0. The function then returns a set of N belief points at most.

With the FSVI strategy, the backup function can be applied exclusively to the newly generated points. All the generated belief sets are sequences starting from the belief point b_0, meaning they can be considered independently.

2.2.7 Value Function Pruning

We must prune unused alpha planes to limit the value function's growth rate. Pruning is an essential step as the size of the value function directly impacts how much time will be spent in the backup function. This operation involves seeing what vectors do not have a top surface in belief space. Pruning is a costly step. A trade-off must, therefore, be found between how much time to spend pruning and how much time it will save in the backup function.

We define different pruning levels to limit the time spent on the pruning operation. The levels increase in how thorough the usefulness check is, but also how costly the check is.

The first level of pruning is implicitly defined in the union operation (Equation 2.32). If an alpha plane is already present in the value function, we can omit it from the value function.

Then, a check for absolute dominance can be performed. An alpha plane α_1 is considered absolutely dominated by another α_2 if the value of α_1 is lower or equal to the value of α_2 for all states:

$$\alpha_1(s) \leq \alpha_2(s), \forall s \in S \tag{2.38}$$

In the worst case, this check must run between all pairs of alpha planes in the value function, close to $|V|^2$ checks.

Next, we can use the set of all belief points explored so far to check for dominance. We retain only the alpha vectors that lead to the highest value for at least one belief point. This procedure can be summarized as follows:

Algorithm 2.6 Dominance over beliefs

```
1: $V_{pruned} = \{\}$
2: for $b \in$B do
3:     $\alpha_b^* = \text{argmax}_{\alpha \in V} \sum_{s \in S} b(s)\alpha(s)$
4:     $V_{pruned} \leftarrow V_{pruned} \cup \alpha_b^*$
5: end for
```

This process is about as computationally expensive as the absolute dominance check. It would require $|V||B|$ operations, and the size of the belief set B grows around the same rate as V.

However, this procedure has the drawback that it is too aggressive and could be remove alpha vectors that would be useful. It is because we only consider the belief points explored so far, while the value function is defined in the continuous belief space. This means that removed alpha vectors could be dominant for belief points not yet encountered with the expand function.

This last procedure can, however, be used to filter the alpha vectors before adding them to the value function in the backup function. The test can then be performed with the same set of belief points used for the backup function. This greatly decreases the required checks while avoiding too high of an increase in the value function size.

Finally, the most expensive way to compute whether an alpha plane is significant in the value function is to solve a Linear Programming problem. So far, we have checked whether an alpha plane is dominated by comparing it to individual alpha planes. Now, there are situations where a combination of planes dominates a single plane. Take Figure 2.8; we can see that alpha plane α_2 is dominated, but not by a single vector; it is dominated by the combination of both α_1 and α_2.

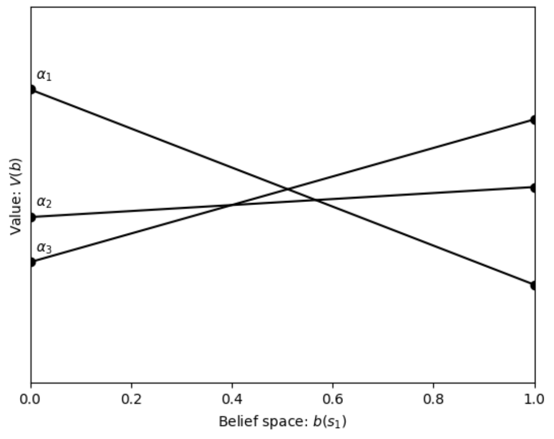


Figure 2.8: Value function with dominated alpha plane

This method to check for dominance with an LP is called the Witness Algorithm and was developed by Littman et al. [6]. This algorithm finds a belief point *witnessing* whether a particular plane is dominated. Formally, it gives:

$$\begin{array}{ll} \text{Maximize} & d \\ \text{Such That} & d \leq (\alpha_i - \alpha_{new}) \cdot b, \forall \alpha_i \in V \\ & \sum_{s \in S} b(s) = 1 \\ \text{And} & b(s) \geq 0, \forall s \in S \end{array}$$

Visually, on the example problem defined previously (Figure 2.3), checking for an alpha vector α_2, we have:

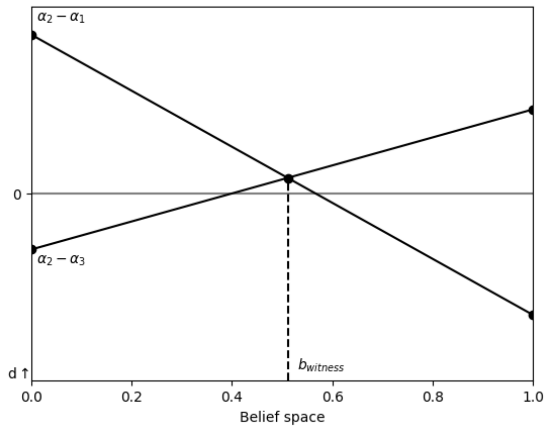


Figure 2.9: Example representation of the LP for the witness algorithm

If d is positive, this means that belief b, whose components are the output of the LP, is the witness to the fact that the α_{new} plane is not dominated. Otherwise, if $d \leq 0$, we can affirm that α_{new} is dominated and prune it out.

This last method is the most expensive as it involves solving an LP at every check we want to make. An improvement to this Witness Algorithm has been developed by Walraven, E. and Spaan, M. [17] that makes use of Benders decomposition (Benders, J. [2]) to select constraints more efficiently.

2.2.8 PBVI Convergence

Compared to the fully observable case, the convergence of Point-Based Value Iteration is more complex to calculate in the partially observable case, as the value function in the fully observable case is fully encapsulated at the state points. In contrast, in the partially observable case, the value function exists in a continuous space of belief points.

Typically, due to the computationally expensive nature of PBVI, the algorithm was run for a set amount of time before returning a solution or with a set number of iterations. Setting a fixed amount of steps is a valid approach as the PBVI algorithm is defined as an "anytime" algorithm (Pineau, J. et al. [7]); this means that a valid solution is guaranteed even when stopping at an arbitrary time. An even stronger statement is that the algorithm will output an increasingly good solution with the longer it runs.

In some literature (Russel, S. and Norvig, P. [10]: Section 17.4), a stopping criterion, however, is to compute the maximum change at the points of certitude, like in the fully observable case. If this maximum change is less than a threshold parameter, we can conclude that the process has converged.

In experiments, we observed that this measure varies greatly between iterations, particularly for larger models, and proves unreliable when assessing a solution's quality. An alternative option to assess the convergence is to run a set of simulations and compute the Average Discounted Rewards (ADR) at the end. The ADR gives a more accurate metric of how the algorithm will perform in practice, but computing this number is computationally expensive. Therefore, running this at set intervals of iterations could be a good solution.

2.3 Study of Observability Variations

Now that the theory of Partially Observable Markov Decision Processes has been defined, one could wonder what happens when the information provided by each action changes. For this, we consider a problem where the observations are the states themselves, meaning the agent observes in what state it currently is. We can modulate the amount of information an action gives by changing the observation probabilities of said actions.

For this, we define a 2-state POMDP problem, where a unit of reward is gathered from each state as a given probability. The POMDP has two actions, "stay" (in the current state) or "move" (to the other state), which have an 80% success rate. The agent can observe its current state with some probability.

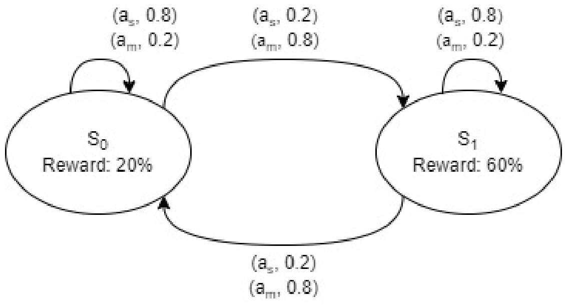


Figure 2.10: An example 2-state POMDP model's state diagram

The probability for the agent to correctly observe its current state is set between 50% (the agent gains no information from observations) and 100% (the agent is in a case of full observability). Running the Point-Based Value Iteration solver for this model until convergence, we get a value function. The actual implementation of the solver used to find a solution will be discussed in more detail in the Implementation chapter (Chapter 3). The discount factor and the epsilon parameters used in the process are 0.99 and 0.001, respectively.

We select a fixed set of beliefs to avoid randomness and run the backup function iteratively. This set is made of points ranging from $[0.0, 1.0]$ to $[1.0, 0.0]$ with increments of 0.01. The set is composed of 101 belief points. With this set, the expand function is unnecessary and not desired to avoid randomness.

The solution for the fully observable version of the problem, so taking away the concept of observations and beliefs, we can find a solution with the regular Value Iteration algorithm. We can plot it in belief space as follows:

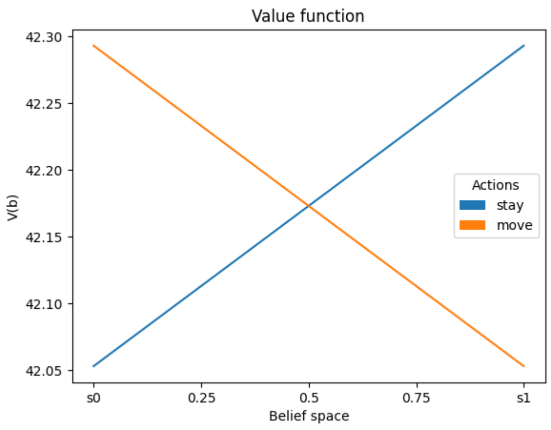


Figure 2.11: The solution of the MDP version of the problem

Our hypothesis for how the solutions will look is that the policy of the solutions will be the same. However, we expect the actual values will be lower due to some value lost with the observation uncertainty.

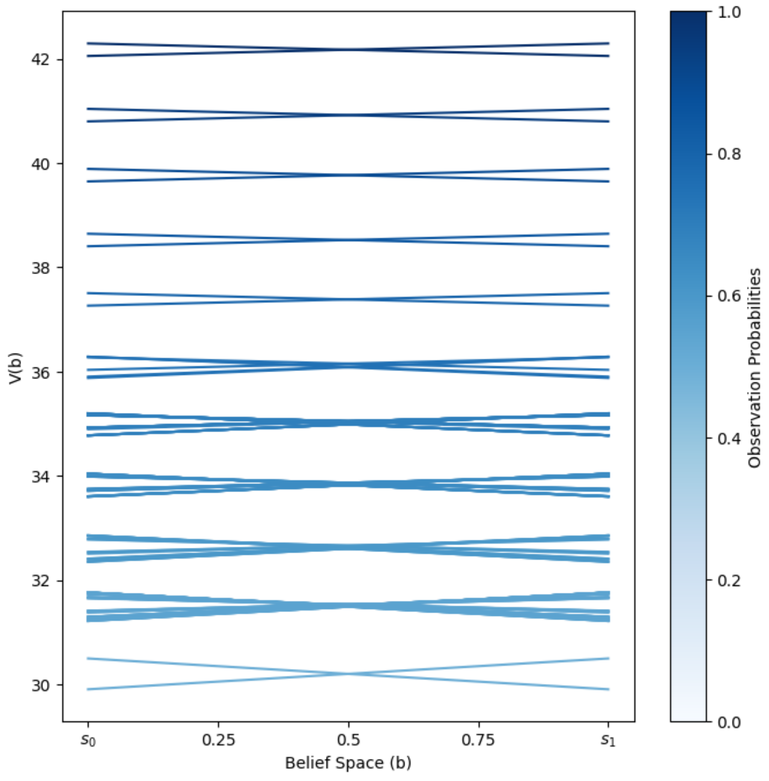


Figure 2.12: Solutions of models with observation probabilities between 0.5 and 1 with increments of 0.05

In a POMDP problem with observation probabilities of 1, when solved with the same parameters, we can see that the value function converges to the same point as the fully observable version of the problem. We can also note that the values at which the value function converges are linearly decreasing, with the observation probability tending towards 50%.

Another interesting fact is the number of alpha vectors the value function comprises. The lower the observation probability, the more alpha vectors make up the value function.

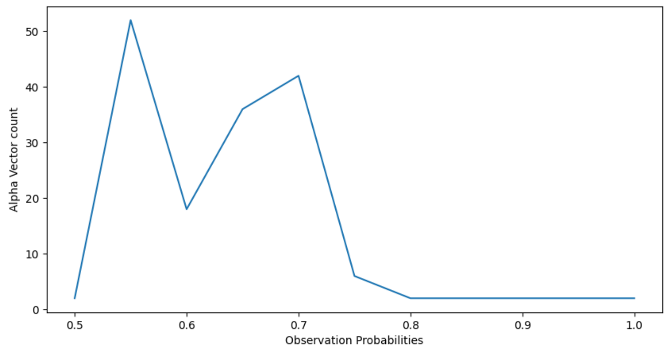


Figure 2.13: Count of alpha vectors making up the value function when solving with different observation probabilities

We can explain that with the fact that the agent moves more in the belief space when less information can be gathered from actions. Following this, more alpha planes are required to describe the value function in uncertainty.

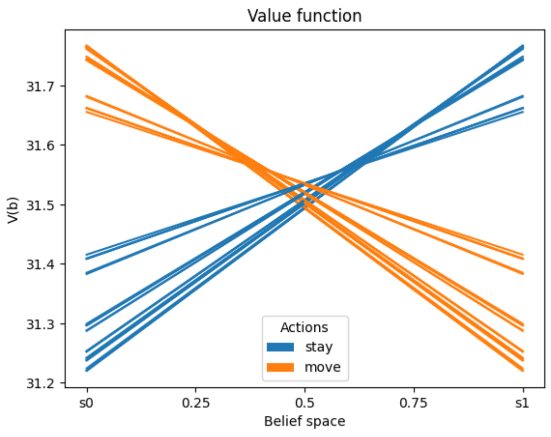


Figure 2.14: Solution of POMDP problem with observation probability of 0.55 and made up of 52 alpha vectors.

Chapter 3

Implementation

In this chapter, we will focus on how the solving process was implemented in practice (Section 3.1) and the techniques used to optimize the performance of the code (Section 3.2, Section 3.3, and Section 3.4). Then, we will analyze the theoretical complexity of the code in terms of time and memory (Section 3.5). The chapter will then conclude with an example problem of POMDP and its solving with our implementation (Section 3.6).

3.1 Framework

The code was built with the purpose of being used not only for this thesis but for a multitude of different projects. The code is centered around two modules, *mdp* and *pomdp*. In each module, the user can use and interact directly with a set of main classes: *Model*, *Solver*, and *Agent*. These classes then use and return secondary classes (*ValueFunction*, *SolverHistory*, *SolverHistory*, *Belief*, *BeliefSet*, etc.). These classes are not meant to be instantiated directly but can be used for more advanced purposes. The secondary or helper classes allow for functionalities such as plotting or saving.

The main classes will be described in detail in the following section, and then the functionality brought by the helper classes will be listed. Finally, the various dependencies required to run the code will be listed. Only the arguments and parameters of the functions and classes relevant to this thesis will be listed.

3.1.1 Model Class

The model class is built to encapsulate all the characteristics of an MDP and a POMDP, as defined in Section 2.1.1 and Section 2.2. Functions defined in this class are not part of the core functionalities but are helpful to avoid code repetition and as helper functions for other classes.

Parameter	**Purpose**
states	The number of states or a list of state IDs or labels.
actions	The number of actions or a list of action IDs or labels.
observations	The number of observations or a list of observation IDs or labels.
transitions	A table with the transition probabilities for each state, action, and next state. Or a function returning a probability from a state, action, and next state. (Optional if reachable_states parameter is provided)
reachable_states	A table with a mapping of state IDs or labels and action to reachable states. (optional)
rewards	A table with the rewards for each state, action, and next state. Or a function returning a reward from a state, action, and next state. (optional if end_states is provided)
observation_table	A table with the observation for each action, next state, and observation.
state_grid	A table of state IDs defining the 2d grid underlying a problem. (Optional)
start_probabilities	A list of probabilities of what state an agent can possibly start in. (By default: uniform over the state space)
end_states	A list of states in which a simulation ends. If rewards parameter is not provided, a reward of 1 is awarded for each state in end_states. (Optional)
end_actions	A list of actions which, when taken, end a simulation. (Optional)

Table 3.1: Model class - Parameters

Function	**Details**
transition	Arguments: • state • action Returns a random next state based on the transition probabilities of the model based on the *state* and what *action* is taken.
reward	Arguments: • state • action • next_state Returns a reward based on which *action* has been taken from what *state* and which is the *next_state*.
observe	Arguments: • action • next_state Returns a random observation based on the observation probabilities of the model based on what *action* has been taken and which *next_state* it has led to.
get_coords	Arguments: • state Returns the 2D coordinate of the *state* on the state grid of the model.
save	Arguments: • file_name • path Saves the model to a pickle file ('*file_name/path*') to be able to be reused without redefining entirely.
load_from_file	Arguments: • file Loads a model from a pickle *file*.

Table 3.2: Model class - Functions

Figure 3.1 is an example of a 2-state MDP model. It displays the transition between states and reward probabilities as a state diagram.

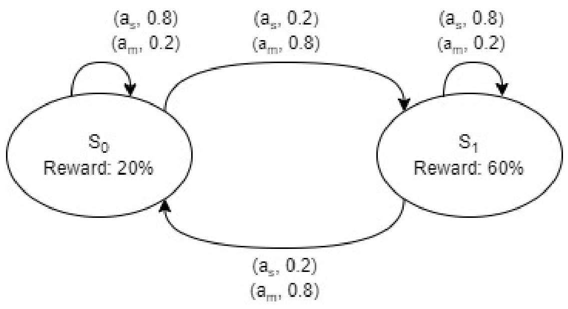


Figure 3.1: An example 2-state POMDP model's state diagram

The observations are the states, representing the agent observing its current position. The observation has an 80% success; therefore, the observation table contains 0.8 when the next state is equal to the observation and 0.2 otherwise.

In Python, it is done as follows:

```
model = Model(states=['s0','s1'],
              actions=['stay', 'move'],
              observations=['s0', 's1'],
              transitions=transition_table,
              rewards=reward_table,
              observation_table=observation_table)
```

3.1.2 Solver Class

The class contains the functionality necessary to find an optimal policy for a model defined in a Model object. The function takes in a Model object and Belief object, then iteratively generates BeliefSet objects and updates a ValueFunction object. Finally, it returns a ValueFunction object and a SolverHistory object.

Parameters	**Purpose**
gamma	The discount factor to be used during the solving process. It corresponds to the inverse of the learning rate. (default: 0.99)
eps	Used as a convergence threshold. (default: 0.001)
expand_function	For POMDP, the belief expansion strategy is to be used. Choice between: RA, SSRA, SSGA, SSEA, GER, HSVI, FSVI, Perseus.
expand_function_ params	Additional parameters to be passed to the expand function, as a dictionary. Example: For SSGA, an "epsilon" parameter has to be provided. For HSVI and FSVI an "mdp_policy" parameter has to be provided.

Table 3.3: Solver class - Parameters

Function	**Details**
expand	Arguments: • model • belief_set • max_generation • *other* Returns a new *belief_set* for the given *model* containing at most *max_generation* beliefs. The specific flavor of the expand strategy is defined using the "expand_function" class parameter. It follows the definition in the PBVI Expand Function section (Section 2.2.6).
backup	Arguments: • model • belief_set • value_function Returns an updated version of the *value_function* for the given *model* from the belief points contained in the *belief_set*. It works as defined in the PBVI Backup Function section (Section 2.2.5).
solve	Arguments: • model • expansions • full_backup • update_passes • max_belief_growth • initial_belief • initial_value_function • use_gpu The main function used to solve a MDP or POMDP *model* by running a number of iterations (*expansions*), and in each iteration, generate at most a number of beliefs (*max_belief_growth*). If the parameter *full_backup* is enabled, the new beliefs generated are added to a general pool of belief points. The backup function can be run multiple times on the same set of beliefs (*update_passes*). Finally, the belief and value function can be initialized manually (*initial_belief* and *initial_value_function*); otherwise, the belief is initialized as the b_0 and the value function as the expect reward matrix. It follows the procedure defined in the PBVI Framework (Algorithm 2.1).

Table 3.4: Solver class - Functions

We can use this class to solve a given model with Point-Based Value Iteration and Stochastic Simulation with Exploratory Action. The parameters could be set to 100 iterations (*expansions*) with 100 beliefs generated per iteration (*max_belief_growth*). We can also set the solver to run on the GPU with the *is_gpu* parameter. In Python, this gives:

```
pbvi_solver = PBVI_Solver(0.99, eps=1e-6, expand_function='ssea')

pbvi_solution, hist = pbvi_solver.solve(model=model,
                                        expansions=100,
                                        max_belief_growth=100,
                                        use_gpu=True)
print(hist.summary)
```

3.1.3 Agent Class

The agent class has been built for simulation purposes. The class allows an agent to be trained using a value iteration with the Solver class. The solution from the solver can be used for a simulation or a set of simulations.

Parameter	**Purpose**
model	The model under which the agent lives.
value_function	The value function that will be used to choose actions. (Optional)

Table 3.5: Agent class - Parameters

Function	**Details**
train	Arguments: • The arguments to pass to the *Solver.solve()* function. The function to run if no value function has been passed to the Agent in order to generate one.
simulate	Arguments: • simulator • max_steps • start_state The function to run a simulation with the agent up to *max_steps* steps with a given simulator and from a given *start_state*. It returns a sequence of states, actions, observations, and rewards.
run_n_simulations	Arguments: • simulator • n • max_steps • start_state The function to run sequentially *n* simulations.
run_n_simulations_parallel	Arguments: • n • max_steps • start_states Function to run a set of *n* simulations. A simulator cannot be set as this function has been purpose-built to be optimized for parallel running.

Table 3.6: Agent class - Functions

Therefore, with this class, we can use the model defined previously and the value function output by the solver to define an Agent object. This agent can then be used in a simulation to evaluate how the agent fairs with this value function in practice.

For example, in Python:

```python
a = Agent(model=model, value_function=pbvi_solution)
sim_hist = a.simulate(start_state=0)
```

3.1.4 Helper Classes

We define helper classes as classes whose objects are meant to be used within and returned by the main classes. They provide extra functionalities that are not expressly necessary to solve an MDP model but that provide the ability to make some visualizations and input/output.

- Belief class:

  This class is a wrapper to the 1D Numpy array with the belief probability for each state. The belief object can be plotted as a grid to visualize the probability of being in each state.

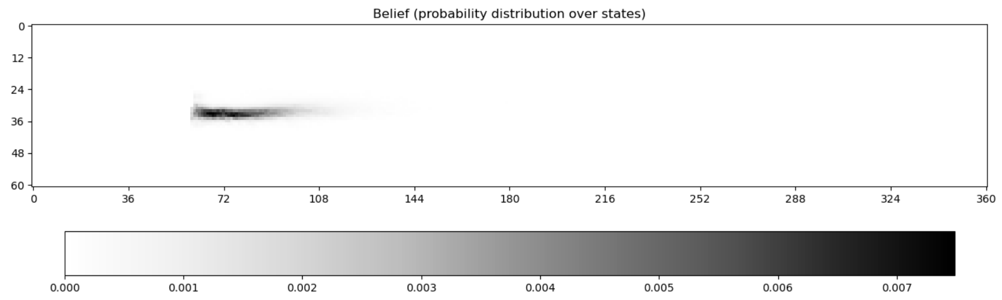
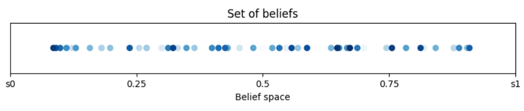


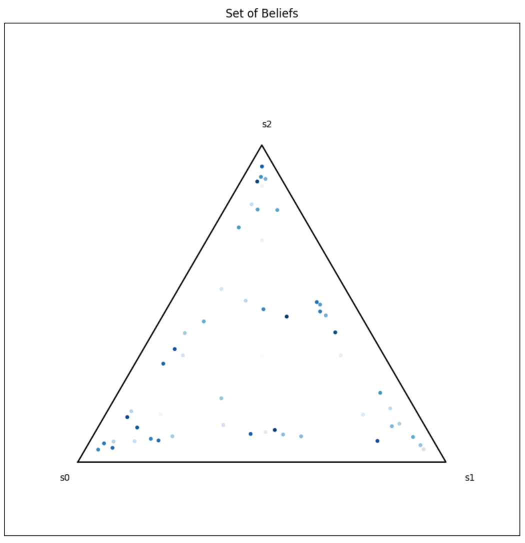
Figure 3.2: Example belief of state probability distribution

- BeliefSet class:

  This class allows the user to get a list of points or points as a matrix of shape $|B| \times |S|$ with $|B|$ being the number of belief points in the set. It is built to make it easy to interchangeably interact with either a sequence or a matrix of all beliefs. For models in 2D or 3D (meaning with 2 or 3 states), the belief set can be plotted in belief space (Figure 3.3).



(i) 2-state model



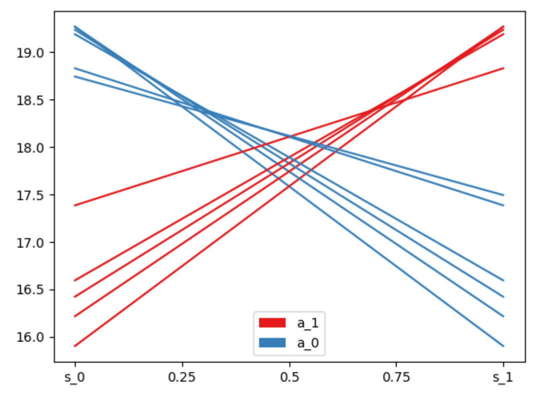
(ii) 3-state model

Figure 3.3: The 2- and 3-state model's belief sets after several iterations.
The darkness of the belief points corresponds to the time the belief point was added to the set.

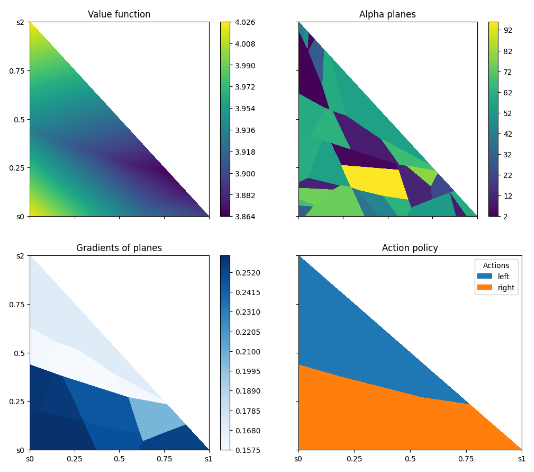
- ValueFunction class:

  The ValueFunction class is a central helper class because it will allow for a policy to be drawn from it that the agent can follow. Like in the BeliefSet class, it can be chosen to retrieve either a list of AlphaVector objects or a matrix of shape $|V| \times |S|$, given that $|V|$ alpha vectors are in the value function. The AlphaVector objects are wrappers over a 1D Numpy array or size $|S|$ along with an associated action.

  Then, this class also allows for plotting the value function. It is more accurate and useful for a 2- or 3-state model as we can visualize the continuity between certainty belief points. However, the class allows for the plotting of value functions associated with models with higher state counts; however, this requires some tradeoffs. The tradeoff is that at each state (certainty belief points), we can show the highest value and which action is to be taken. With this representation, we cannot grasp the policy for belief points between points of certainty.



(i) 2-state model



(ii) 3-state model

Figure 3.4: Value function representation in belief space with the alpha vectors and the policy for a 2- and 3-state model

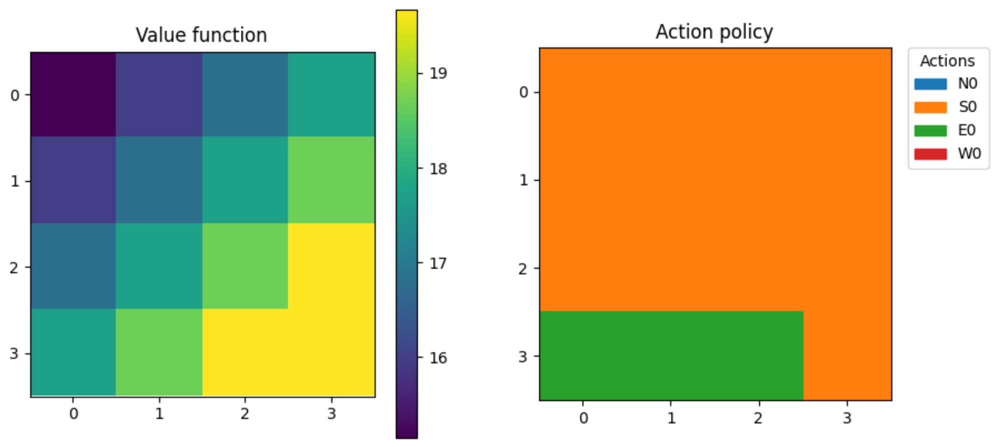


Figure 3.5: Value function representation at certainty points (states) with the maximum values and policies for a 16-state model as a 4×4 grid.

Finally, ValueFunction instances can be saved and loaded for future use and to reproduce simulations with the *save* and *load_from_file* functions. The *save* function saves the value function in a CSV file, with the first column being the actions and all the others being the values at each state.

- SolverHistory class:

  The SolverHistory class is used to track the progress of the solving method. The main items it records are the times to run individual parts of the solving process for each iteration, for example, the expand and backup functions. Then, the belief set and value function sizes are recorded after each iteration. Moreover, at a higher tracking level, the actual belief set and value functions can be recorded at the cost of memory.

One last functionality of the class is the ability to generate a video plotting the value function over the solving process to visualize the evolution over time. For 2- and 3-state models, the set of belief points is also visualized along the value function.

- SimulationHistory class:

  The SimulationHistory class tracks the progress of an agent along a simulation. It records all the states the agent passed through, along with the actions it made, the observations it made, and the rewards it received. The beliefs the agent goes through during the simulation process are also recorded. With this recorded information, the path taken by the agent can be plotted. Additionally, a video can be generated to visualize the current position of the agent over time, the action taken, the observation made, and the agent's belief. Finally, the simulation can be recorded in a CSV file with the columns being "states," "actions," "observations," and "rewards."

3.1.5 Dependencies and Details

The code is implemented in Python 11.3 and uses mainly the following libraries:

- *Numpy* v1.26 (`https://numpy.org/`)
- *Pandas* v2.1 (`https://pandas.pydata.org/`)
- *Matplotlib* v3.7 (`https://matplotlib.org/`)
- *Cupy* v12.2 (`https://cupy.dev/`) (Optional: enables the use of GPU to speed the solving and simulation processes)
- *Scipy* v1.11 (`https://scipy.org/`) (Optional: enables the use of the witness algorithm for alpha vector pruning)

The code can be found on the GitHub of the PiMLB lab. It is available at the following link: `https://github.com/PimLb/POMDP_PBVI_Exploration`

3.2 Reachability

We note that some POMDP problems have a very sparse transition function, meaning one single state and action pair can lead to few new states. From this observation, we can define a concept of reachability, representing the amount of states reachable from any state-action pair.

To use this concept, we can reformulate the transition matrix as two smaller matrices. The first matrix will be a mapping, stating the indices of the states reachable for each state and action, while the second matrix will be the probability of reaching these states. If we consider that the maximum amount of states reachable at any point is R, the shape of the two matrices will be $|S| \times |A| \times R$. Let us consider the case where, at most, one state is reachable from any state; we can say that the problem is deterministic. This means that the two matrices will be of shape $|S| \times |A| \times 1$.

3.3 Optimizations

The most computationally expensive operations of Point-Based Value Iteration (PBVI) are the backup function and the belief update in the expand function. For this, we will first see how it would be expressed as an algorithm and then how these algorithms can be reformulated to utilize vectorization.

3.3.1 Pre-computation

In order to trade memory for better runtimes, we can pre-compute operations that are run multiple times without change.

For instance, we consider the pre-computation of the transition probability matrix (T, having shape $|S| \times |A| \times |S|$) with the observation probability matrix (O, having shape $|A| \times |S| \times |\Omega|$). This would mean generating a matrix of shape $|S| \times |A| \times |S| \times |\Omega|$ from taking the sum-product along the dimensions of the actions and the next states, which we can call the *transitional-observation* probability matrix. The operation can be defined in Python with the “einsum” function of the Numpy library; with this, we can state how the matrices should be combined. In our case, we take the first matrix (the transition probabilities) with shape $s \times a \times n$ and the second matrix (the observation probabilities) with shape $a \times n \times o$, leading to a resulting matrix (the transitional-observation probabilities) with shape $s \times a \times n \times o$. (Note: the n shape is the same as s; we define them separately to represent that one dimension is the *states* dimension while the other is the *next states* dimension)

```
transitional_observation_m = np.einsum('san,ano->sano',
                                       transition_m,
                                       observation_m)
```

Moreover, we can now use the notion of reachability with the reachable transition observation probability matrix; we can redefine this as a sum-product leading to a matrix of shape $|S| \times |A| \times |R| \times |\Omega|$. Formulated in Python:

```
reach_transitional_observation_m = np.einsum('sar,aro->saro',
                                             reach_transition_m,
                                             observation_m)
```

3.3.2 Backup Function Vectorization

As we can remember from the PBVI backup function theory (Section 2.2.5), the backup function is made of 3 steps, which we can formulate in pseudocode form:

1. Generation of alpha planes (Equation 2.29):

Algorithm 3.1 Backup step 1 - Generation of Alpha Planes

```
1: for Action $a$ in $A$ do
2:     for Observation $o$ in $\Omega$ do
3:         for Alpha plane $\alpha_i$ in $V$ do
4:             for State $s$ in $S$ do
5:                 $\alpha_i^{a,o}(s) = 0$
6:                 for Next state $s'$ in $S$ do
7:                     $\alpha_i^{a,o}(s) \mathrel{+}= \gamma\left(O(a, s', o)T(s, a, s')\alpha_i(s)\right)$
8:                 end for
9:             end for
10:            $\Gamma^{a,o} \leftarrow \alpha_i^{a,o}$
11:        end for
12:    end for
13: end for
```

2. Collapse over observations (Equation 2.30)[1]:

Algorithm 3.2 Backup step 2 - Collapse of observations

```
1: for Action $a$ in $A$ do
2:     for Belief $b$ in $B$ do
3:         $\alpha_b^a = \{0, \forall s \in S\}$
4:         for Observation $o$ in $\Omega$ do
5:             $\alpha_b^{a,o} = \text{argmax}_{\alpha_i^{a,o} \in \Gamma^{a,o}} \alpha_i^{a,o} \cdot b$
6:             $\alpha_b^a \mathrel{+}= \alpha_b^{a,o}$
7:         end for
8:         $\Gamma_b^a \leftarrow \alpha_b^a + R(:, a)$
9:     end for
10: end for
```

3. Collapse over actions (Equation 2.31):

Algorithm 3.3 Backup step 3 - Collapse over actions

```
1: for Belief $b$ in $B$ do
2:     $\alpha_b = \text{argmax}_{\alpha_b^a \in \Gamma_b^a} \alpha_b^a \cdot b$
3: end for
```

From this, we can define the operations on matrices. Matrix operations can be run on multiple threads instead of sequentially with multiple nested loops. In Python, the Numpy library allows for multi-threaded matrix operations.

1. Optimization of the generation of alpha planes (Algorithm 3.1):

   With the pre-computed transitional-observation matrix (Section 3.3.1), we can compute with a new set of alpha vectors from the matrix of previous alpha vectors of shape $|V| \times |S|$. Note that the alpha-vectors matrix's $|S|$ dimension is marked by an "n" because the next states are used with the alpha vectors, not the current ones.

```
gamma_a_o = gamma * np.einsum('sano,vn->aovs',
                              transitional_observation_m,
                              alpha_vector_m)
```

   If we want to use reachable states, we have to first select the reachable states for each state-action pair of the alpha vectors, leading to a matrix of shape $|V| \times |S| \times |A| \times |R|$. In Python, we can formulate this as:

[1]We formulate the loop over the states as vector dot products to compute the sum-product of an alpha vector with a belief vector

```
vectors_array_reachable_states = vector_array[
        np.arange(vector_array.shape[0])[:,None,None,None],
        reachable_states[None,:,:,:]]

gamma_a_o = gamma * np.einsum(
        'saro,vsar->aovs',
        reachable_transitional_observation_m,
        vectors_array_reachable_states)
```

2. Optimization of the collapse over observations (Algorithm 3.2):

   We first select the best alpha plane for all belief points using the "argmax", leading to a matrix of indices of shape $|B| \times |A| \times |\Omega|$. Then, we select them from the gamma_a_o_t matrix, leading to a matrix of shape $|B| \times |A| \times |\Omega| \times |S|$. We can then sum over the observation dimension, to which we add the expected rewards matrix ($|A| \times |S|$).

```
best_alpha_ind = np.argmax(np.tensordot(belief_array,
                                        gamma_a_o_t, (1,3)),
                           axis=3)

best_alphas_per_o = gamma_a_o_t[
                        model.actions[None,:,None,None],
                        model.observations[None,None,:,None],
                        best_alpha_ind[:,:,:,None],
                        model.states[None,None,None,:]]

alpha_a = (model.expected_rewards_table.T
        + np.sum(best_alphas_per_o, axis=2))
```

3. Optimization of the collapse over actions (Algorithm 3.3):

   Finally, to remove the loops from the last step of the backup function, we can also formulate it as matrix operations in the same fashion as for step 2. First we find the index of the best alpha vector per action. Following which we select those in the alpha_a matrix.

```
best_actions = np.argmax(np.einsum('bas,bs->ba',
                                   alpha_a,
                                   belief_array),
                         axis=1)

alpha_vectors = np.take_along_axis(alpha_a,
                                   best_actions[:,None,None],
                                   axis=1)[:,0,:]
```

3.3.3 Belief Superiority only Addition

Another essential step of the Point-Based Value Iteration process is choosing what alpha vector has to be added during the value function update. To check whether an alpha vector α_{new} should be added to the value function V or not, based on a set of beliefs B, we can follow the belief dominance check procedure (Section 2.2.7) and formulate it as follows:

Algorithm 3.4 Belief superiority only addition

```
 1: for Belief $b$ in $B$ do
 2:     $best\_value \leftarrow -\infty$
 3:     for Alpha vector $\alpha$ in $V$ do
 4:         $alpha\_value \leftarrow \alpha \cdot b$
 5:         if $alpha\_value > best\_value$ then
 6:             $best\_value \leftarrow alpha\_value$
 7:         end if
 8:     end for
 9:     $alpha\_value \leftarrow \alpha_{new} \cdot b$
10:     if $alpha\_value > best\_value$ then
11:         $\alpha_{new}$ is dominating at b: Add $\alpha_{new}$ to $V$
12:     end if
13: end for
14: $\alpha_{new}$ is not dominating: It is not added to $V$
```

This operation is ideally run at the end of the backup step to check the utility of the newly generated alpha vectors to the value function. This operation is rather heavy to perform, and it also grows with the number of alpha vectors already present in the value function. In the worst case scenario, a total of $|V_{new}| \times (|B| \times |V| \times |S| + |S|)$ operations. Where V_{new} represents the set of new alpha vectors.

Due to the repetitive nature of these operations, the loops can be converted to a Numpy matrix operation to utilize multithreading. In Python, it can be done as follows:

```python
best_value_per_belief = np.sum((belief_array * alpha_vectors),
                               axis=1)
old_best_value_per_belief = np.max(np.matmul(belief_array,
                                             vector_array.T),
                                   axis=1)
dominating_vectors = (best_value_per_belief >
                      old_best_value_per_belief)

```

```python
best_actions = best_actions[dominating_vectors]
alpha_vectors = alpha_vectors[dominating_vectors]
```

3.3.4 Pruning

As seen in Value Function Pruning section (Section 2.2.7), we defined several pruning levels to limit the growth of alpha vectors making up the value function.

First, we defined a straightforward de-duplication pruning operation. It ensures that no copies of the same alpha vector are within the value function. In Python, we found the optimal way of de-duplicating a list of Numpy 1D arrays is to gather the byte string representation of the array and push them in a dictionary with the values being the original arrays. Finally, we gather the dictionary's list of values, now a list of unique alpha vectors.

```python
uniqueness_dict = {alpha_vector.values.tobytes(): alpha_vector
                   for alpha_vector in vector_list}
vector_list = list(uniqueness_dict.values())
```

The second technique we have seen consists of removing alpha vectors point-wise dominated by any other vector. It is a technique similar to the one defined in the Belief Superiority only Addition section (Section 3.3.3). In this technique, each vector is confronted with all other vectors; if one vector is inferior or equal to another at each point, it is said to be dominated. For a vector $\alpha \in V$ checked against a set $V_{comp} := \{\alpha_{comp} \in V | \alpha_{comp} \neq \alpha\}$, this would give:

Algorithm 3.5 Alpha vector absolute dominance

```
 1: function IS_DOMINATED( $\alpha, V_{comp}$ )
 2:     for Alpha vector $\alpha_{comp}$ in $V_{comp}$ do
 3:         $comp\_dominates \leftarrow true$
 4:         for State $s$ in $S$ do
 5:             if $\alpha(s) > \alpha_{comp}(s)$ then
 6:                 $comp\_dominates \leftarrow false$
 7:                 break
 8:             end if
 9:         end for
10:         if $comp\_dominates$ then return $true$
11:         end if
12:     end for
          return $false$
13: end function
```

In Python, we can utilize Numpy's vectorization to perform this check in the following way:

```python
is_dom_by = np.all(alpha_vector_array >= alpha_vector, axis=1)
is_dominated = True if len(np.where(is_dom_by)[0]) == 1 else False
```

Finally, to most accurately check whether a vector is necessary, one should check using the Witness algorithm, which checks whether a combination of vectors dominates a vector. As we have seen, this is done by defining a Linear Program. In Python, we implemented this using the "milp" module of the Scipy library as follows:

```python
# Objective function
c = np.concatenate([np.array([1]), -1*alpha_vect])

# Alpha vector constraints
other_count = len(other_alphas)
A = np.c_[np.ones(other_count),
          np.multiply(np.array(other_alphas), -1)]
alpha_constraints = LinearConstraint(A, 0, np.inf)

# Constraints that sum of beliefs is 1
belief_constraint = LinearConstraint(np.array([0]
                        + ([1]*self.model.state_count)), 1, 1)

# Solve problem
res = milp(c=c,
           constraints=[alpha_constraints, belief_constraint])

# Check if dominated
is_dominated = (res.x[0] - np.dot(res.x[1:], alpha_vect)) >= 0
```

The formulation of the LP is different than what we have defined in the Value Function Pruning section (Section 2.2.7), but it accomplishes the same purpose.

3.3.5 Belief Update Vectorization

If we look at the Belief update equation 2.16, we can identify two steps. First, we must compute the conditional probability $P(o|b, a)$ and then the actual updated belief at each state divided by $P(o|b, a)$.

Algorithm 3.6 Belief update of belief b for action a and observation o

```
1: $P\_bao \leftarrow 0$
2: for State $s$ in $S$ do
3:     for Next state $s'$ in $S$ do
4:         $P\_bao \mathrel{+}= O(a, s', o)T(s, a, s')b(s)$
5:     end for
6: end for
7: $b^{a,o} \leftarrow \{0, \forall s \in S\}$
8: for Next state $s'$ in $S$ do
9:     for State $s$ in $S$ do
10:        $b^{a,o}(s') \mathrel{+}= O(a, s', o)T(s, a, s')b(s)$
11:    end for
12:    $b^{a,o}(s') \leftarrow \frac{b^{a,o}(s')}{P\_bao}$
13: end for
```

To improve this process, we can first use the pre-computed transitional-observation probabilities matrix to multiply with the belief point as a row matrix instead of formulating the operation as loops. Then, instead of computing the probabilities P_bao separately to normalize the updated belief components, we can compute $b^{a,o}$ fully and divide by the sum of its elements. In Python, we have:

```python
new_state_probabilities = np.einsum(
        'sn,s->n',
        transitional_observation_m[:,a,:,o],
        belief)
new_belief /= np.sum(new_state_probabilities)
```

We can also use reachability to reduce the amount of operations required to compute the next state's probabilities. We, however, need a trick to get the probabilities of the reachable states. In Python, we can use the "bincount" function of the Numpy library. This function makes a histogram of how many times a number appears, except we can use the "weights" argument to have a weighted sum of the items on top of that. This function allows us to use the reachable states matrix as the items and the probabilities of the reachable states as the weights in the "bincount" function.

```python
reachable_state_probabilities = (
        reachable_transitional_observation_m[:,a,o,:]
        * belief[:,None])
new_state_probabilities = np.bincount(
        reachable_states[:,a,:].flatten(),
        weights=reachable_state_probabilities.flatten(),
        minlength=model.state_count)
new_state_probabilities /= np.sum(new_state_probabilities)
```

With the alternative implementation, the computation of the probabilities reduces the computational complexity by one order, from $|S|^2$ to $|S|$, and in the worst case will stay the same. However, the additional step with the "bincount" function results in additional $|S|$ operations and, at worst, $|S|^2$. In total, it means that there is one order of computational complexity of improvement. At the same time, in the worst case, the performance decreases but within the same computational order.

3.3.6 Parallelization of Simulations

A simulation based on a solution will consist of an agent evolving in the environment defined by the model and taking actions based on the policy of this solution. The agent starts in a state s_0; it can be picked randomly in the start zone or be set in advance. Under partial observability, the exact state remains unknown to the agent, even though it is used within the simulation to determine the observations the agent can receive.

It starts with an initial belief b_0, having a uniform probability distribution of being in any state within the starting zone. Then, for the actual simulation loop, an action a, associated with an alpha vector, is chosen based on which alpha vector α^* from the value function maximizes the dot product with the belief. The next state s' is then chosen using the transition function. An observation o is then performed; it is picked based on the observation function at state s', and for action a. Finally, the agent updates its belief from the action a it has taken and the observation o it has received, leading to a new belief $b^{a,o}$. The rewards received by the agent are recorded along the path of the simulation. The simulation halts when it reaches the horizon or if it has reached the end state.

Algorithm 3.7 Simulation process

```
1: Initialize $s \leftarrow s_0$
2: Initialize $b \leftarrow b_0$
3: Initialize $rewards = \{\}$
4: while $horizon$ not reached and $s \notin$ end-states do
5:     $\alpha^* \leftarrow \text{argmax}_{\alpha \in V}\, \alpha \cdot b$
6:     $a \leftarrow$ action associated with $\alpha^*$
7:     Sample $s' \in S$ based on $P(s'|s, a)$
8:     Sample $o \in \Omega$ based on $P(o|s', a)$
9:     $r \leftarrow R(s, a, s', o)$
10:    $rewards \leftarrow rewards \cup \{r\}$
11:    $b \leftarrow b^{a,o}$
12:    $s \leftarrow s'$
13: end while
```

From this algorithm, we note that the heavy operation is the choice of the alpha vectors (line 5 in Algorithm 3.7) and the belief update operation (line 11 in Algorithm 3.7). To pick the best alpha vector, we can reformulate the operation in a matrix operation where the value function is represented by a matrix of shape $|V| \times |S|$, where $|V|$ is the number of alpha vectors in the value function. This value function matrix can then be multiplied by the belief b as a row vector of shape $|S| \times 1$, resulting in a column vector of shape $|V| \times 1$. Formulating the operation as a matrix multiplication instead of a sequence of dot products allows it to be multi-threaded on the GPU. The belief update operation, even if heavy, cannot be changed further to improve the speed of the simulation.

Taking this concept further, instead of running each simulation one after the other, we can consider running the simulation all simultaneously, in parallel. Therefore, a state vector would replace the single state, and the belief would be replaced by a belief matrix B of shape $n \times |S|$, where n is the number of simulations, meaning the number of belief vectors. To find the best alpha vectors, we can take the matrix product between V and B: $V \times B^T$, resulting in a matrix of shape $|V| \times n$. Finally, we can take the *argmax* over the first dimension to find the best alpha vector per belief, leading to a vector of shape $1 \times n$. For the second heavy operation, the belief update, we can formulate it as a matrix operation to update all the beliefs simultaneously instead of one by one.

This parallel implementation leads to a significant time improvement, especially for increasingly large amounts of simulations. The optimizations of the multi-threaded matrix operations contribute to this performance improvement. The following sections will test these claims.

3.4 GPU Implementation

The code framework has been implemented so that running the solver on the GPU or the CPU is as simple as toggling it on. Over the code, depending on whether the GPU toggle is switched on, the functions adapt themselves. The library allowing for the operations to run on the GPU is Cupy. Cupy is a drop-in replacement for Numpy, meaning that almost all operations available with the Numpy library are also available in Cupy. With Cupy, the operations on arrays are run on a CUDA-compatible GPU. The main modifications that need to be implemented are the data movement between the RAM and the GPU memory.

The classes that need support for Cupy are the Model, BeliefSet, and ValueFunction classes, which contain the Numpy arrays used. The flexibility to choose between GPU operations or CPU operations is implemented by adding an "on_gpu" attribute to these classes. Along this, functions called "to_gpu" and "to_cpu" are added to the classes to return an analogous object with the array moved to the GPU/CPU.

Additionally, the various functions using Numpy arrays, for example, in the Solver class, are flexible in determining whether or not the ValueFunction, BeliefSet, and Model objects are on the GPU.

Finally, to reduce the overhead of sending large arrays to and from the GPU memory multiple times, whenever the "to_gpu" or "to_cpu" functions are called, the generated analog object is stored as a reference and returned if already present.

3.5 Theoretical Analysis

For the theoretical analysis, we will analyze the number of operations required as a function of the size of the various components of the POMDP model, such as the states, the actions, the observations, or the reachable states. As a baseline, we will compute the amount of operation in the default definition of the functions; then, we will compare this to the implementation with the concept of reachability in best and worst-case scenarios. We consider a best-case scenario when the number of reachable states from any given state with any action is at most 1, meaning the model is deterministic. On the contrary, in the worst-case scenario, any state of the set of states S can be reached with any state-action pair.

	Base	**Reachable - Best case:** $\lvert R\rvert = 1$	**Reachable - Worst case:** $\lvert R\rvert = \lvert S\rvert$
Update function:			
- Step 1	$O(\lvert A\rvert \times \lvert\Omega\rvert \times \lvert V\rvert \times \lvert S\rvert^2)$	$O(\lvert A\rvert \times \lvert\Omega\rvert \times \lvert V\rvert \times \lvert S\rvert)$	$O(\lvert A\rvert \times \lvert\Omega\rvert \times \lvert V\rvert \times \lvert S\rvert^2)$
- Step 2	$O(\lvert S\rvert \times \lvert\Omega\rvert)$	-	-
- Step 3	$O(\lvert S\rvert)$	-	-
- Total	$O(\lvert A\rvert \times \lvert\Omega\rvert \times \lvert V\rvert \times \lvert S\rvert^2 + \lvert S\rvert \times \lvert\Omega\rvert + \lvert S\rvert)$	$O(\lvert A\rvert \times \lvert\Omega\rvert \times \lvert V\rvert \times \lvert S\rvert + \lvert S\rvert \times \lvert\Omega\rvert + \lvert S\rvert)$	$O(\lvert A\rvert \times \lvert\Omega\rvert \times \lvert V\rvert \times \lvert S\rvert^2 + \lvert S\rvert \times \lvert\Omega\rvert + \lvert S\rvert)$
Belief update	$O(\lvert S\rvert^2 + \lvert S\rvert)$	$O(\lvert S\rvert + \lvert S\rvert + \lvert S\rvert)$	$O(\lvert S\rvert^2 + \lvert S\rvert^2 + \lvert S\rvert)$

Table 3.7: Theoretical analysis of the different parts of the Point-Based Value Iteration process for the base and implementation with the reachability concept in the best and worst-case scenarios.

Overall, for the worst-case scenario of the reachable implementation, it equals the base implementation. In the case of the belief update process, we note that the worst-case scenario is worse than the base-case configuration, but the big-O scale of growth is still n^2.

Now, one area to consider is the placement of the data queried. With the reachable implementation, instead of dealing with contiguous, we select the data only at the reachable state, meaning we go back and forth in the data on memory. When querying a single reachable state per state-action pair, this is not too much of a problem. However, when approaching $|R| = |S|$, the jumping around in the data instead of a continuous read may be detrimental compared to the base implementation.

3.6 Validation of the Implementation

To validate the validity of our theory implementation and compare it to other implementations, we attempted to run the solver on a famous POMDP problem.

3.6.1 Example Problem Setup

The problem used for the validation of our implementation is the Tiger Problem. Cassandra A. et al. [3] define the tiger problem as simulating a gambling game. There are two doors; behind one door, there are 100 dollars, and behind the other, there is a tiger. The participant in this game has to choose which door to open; he can, however, also waste time trying to listen if he manages to hear the tiger, in which case he also loses a small amount of money.

This problem was chosen for the validation as the process is simple enough that the solving process is tractable and makes good use of the belief space.

3.6.2 Example Problem Solution

With our implementation of the problem, a solution is found after less than ten iterations and, at most, ten update passes of the update function per iteration. It leads to the following value function:

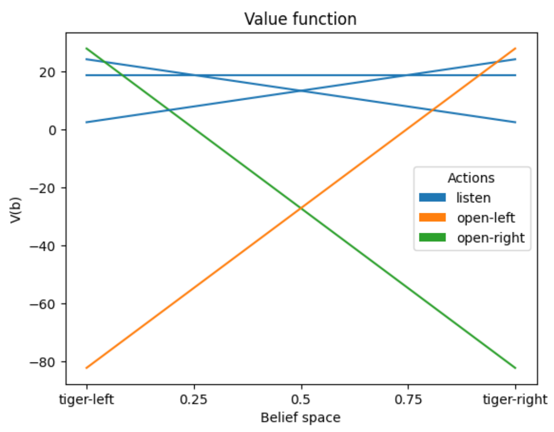


Figure 3.6: Value function of the tiger problem after convergence.

From the value function, we derive a policy. The agent always starts in complete ignorance, then the best action to take is to "listen," which is the belief point closer to either state but not enough to be confident enough to open a door. Therefore, a second listening action should be done to confirm or negate the previous observation, leaving the agent completely ignorant.

We note that this solution is the same as the one found in Cassandra A. et al. [3], and the solving process is similar to ours. This result confirms that our implementation is sound and valid.

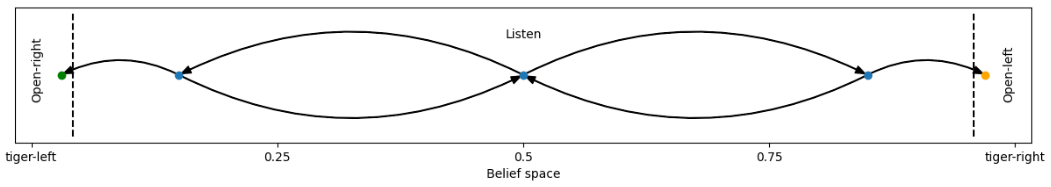


Figure 3.7: Policy of the solution to the tiger problem.

Chapter 4

Navigation as an optimal decision process

In this section, we will explore how Partially-Observable Markov Decision Processes (POMDPs) can be used to find optimal strategies for the problem of Olfactory Navigation. In particular, we consider a single agent, a dog, able to wander while sniffing the ground or stop and take a deeper sniff in the air. This section follows the work of Rigolli N. et al. [9] with our different implementation of a POMDP solver as defined in Chapter 3.

First, we will define the problem of Olfactory Navigation in more detail (Section 4.1). Then, we will define the olfactory navigation problem in terms of a POMDP model (Section 4.2) and what experiments will be done with the solving process (Section 4.3). The solution to the fully observable version will be presented (Section 4.4) as it can be used to solve the partially observable version. Following this, the results of the solving process will be presented as compared to the ones presented in the paper of Rigolli N. et al. [9] (Section 4.5) and diving deeper into the variations displayed with the results (Section 4.6). Finally, the runtimes of the different solving steps and the simulation process will be exposed (Section 4.7 and Section 4.8).

4.1 Olfactory Navigation

The problem of olfactory navigation is for an agent to attempt to find the source of a smell, discarding all other senses. The agent evolves in an environment where, at any instant, it can move up, down, left or right, and smell. The agent's environment is turbulent, either air or water. In nature, this task is accomplished by organisms routinely, but the algorithms employed by the brain still need to be understood better for turbulent environments. Our goal is to implement a simulation, modeling the processes of nature as closely as possible.

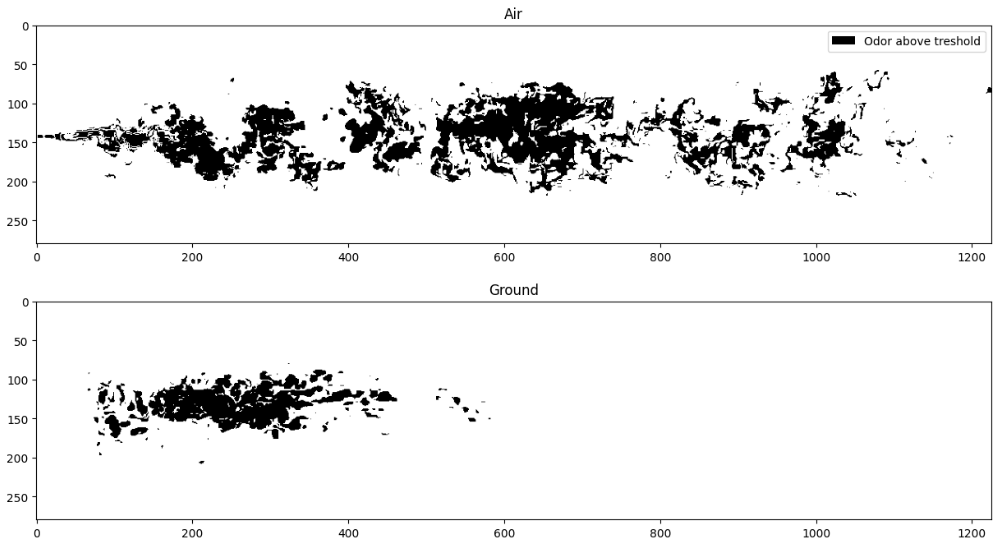


Figure 4.1: An example of the positions on the field, on the ground, and in the air, where the odor cues are above the threshold of $3 \cdot 10^{-6}$ at an instant in time.

At the microscopic scale, the odor field is smooth. In 1986, Segal J. et al. [11], found that bacteria, particularly the E. Coli bacteria, follow the odor gradients to find the source of an attractant. This process of following the odor gradient is called Chemotaxis. As shown by Shraiman B. and Siggia E. in “Scalar turbulence” [13], at larger scales, the odor gradients are broken by turbulence in the fluid. This leads to the impossibility of using odor gradients to perform olfactory navigation at such scales (Balkovsky E. and Shraiman B. [1]). This is why the problem of olfactory navigation in turbulent environments arose.

The problem was first studied by Vergassola M. et al. [16], where the authors explore how an algorithm may be able to find the source of a nutrient in a turbulent environment. In the paper, the authors propose and implement an algorithm called “Infotaxis,” which consists of moving towards maximum information uptake. The agent then takes actions that most reduce the uncertainty about the relative position. At every step, the relative position probabilities are updated using Bayesian updates, similar to the belief update for POMDPs (Equation 2.16).

Infotaxis is a heuristic algorithm that maximizes information about source location. However, maximizing information does not correspond to maximizing the efficiency of reaching the source. Recent approaches move beyond this limitation as they cast turbulent navigation as a POMDP (Rigolli N. et al. [9] and Heinonen A. et al. [5]) and find optimal strategies to reach the source. In Rigolli N. et al. [9], the authors draw inspiration from the observation that during olfactory navigation, dogs alternate smelling in the air while standing still and on the ground while moving. In this paper, they can reproduce a behavior seen in nature where the searcher alternates between phases of casting and surging. Casting is characterized by walking crosswind while intermittently stopping to whiff the air to gain more information about the relative position relative to the source. In comparison, surging consists of walking upwind and getting closer to the source. This is done by modeling the problem as a POMDP and solving it with Forward Search Value Iteration.

4.2 As a POMDP model

Now we can define the tuple $< S, A, T, R, \Omega, O, b_0 >$ for olfactory navigation to reproduce the model used in the paper by Rigolli N. et al. [9]:

- S: States are the searcher's position relative to the source and are assumed to be within a 12m by 2m rectangle. Space is discretized with 30 points per length unit. While in physical space, we consider the source to be at the origin $[0, 0]$ with $x \in [-2, 10]$ and $y \in [-1, 1]$, in the discretized space, we define the origin of the grid at $[-2, -1]$ to avoid negative indices in arrays. This implies that the source is now at coordinates $[60, 30]$ on the discretized grid. The discretized grid consists of 361 units on the x-axis and 61 on the y-axis for 22,021 state points. We decided to add a unit in each direction to make the environment symmetric around the odor source. When visualizing the state space, we will use this discretized grid.

  The behavior at the boundary is that the space wraps around. This means that if the agent is at the top edge and takes a step north, it will appear at the bottom edge. This choice has been made so the space appears infinite to the agent, and it cannot use the edges to make a guess about its relative position to the source.
- A: There are six different actions possible:
  - a_0: Move north, sniff the ground
  - a_1: Move east, sniff the ground
  - a_2: Move south, sniff the ground

  - a_3: Move west, sniff the ground
  - a_4: Stand still, sniff the ground
  - a_5: Stand still, sniff the air

- T: The transition probabilities are deterministic and can be defined as:

$$P(s'_{(x=i',y=j')}|s_{(x=i,y=j)}, a) = \begin{cases} 1.0 & \text{, if } (a=0) \text{ and } (i'=i) \text{ and } (j'=j+1) \\ 1.0 & \text{, if } (a=1) \text{ and } (i'=i+1) \text{ and } (j'=j) \\ 1.0 & \text{, if } (a=2) \text{ and } (i'=i) \text{ and } (j'=j-1) \\ 1.0 & \text{, if } (a=3) \text{ and } (i'=i-1) \text{ and } (j'=j) \\ 1.0 & \text{, if } (a=4) \text{ and } (i'=i) \text{ and } (j'=j) \\ 1.0 & \text{, if } (a=5) \text{ and } (i'=i) \text{ and } (j'=j) \\ 0.0 & \text{, otherwise} \end{cases} \tag{4.1}$$

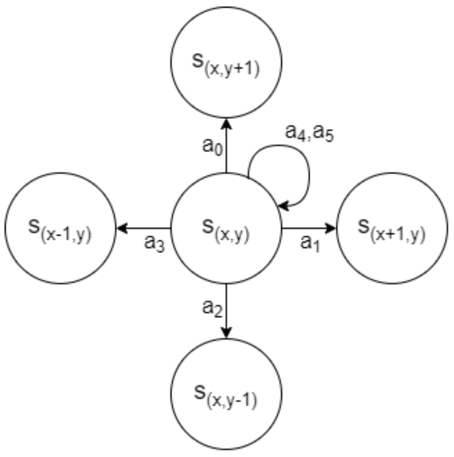


Figure 4.2: State transition based on action

Note: due to the boundary conditions, if $j' > 361$, it becomes 0, and inversely. Likewise for i'; if $i' > 61$, it becomes 0, and inversly.

- R: The reward function is defined as:

$$R(s, a, s') = \begin{cases} 1.0 & \text{, if } s' = s_{goal} \\ 0.0 & \text{, otherwise} \end{cases} \tag{4.2}$$

, where s_{goal} is the state at the position of the source, i.e., at $[0, 0]$.

- Ω: There are three possible observations for olfactory navigation:

1. Nothing
2. Something
3. Goal

- O: The observation function is defined with the probabilities of the odor being above a certain threshold based on the generated turbulent simulation. The odor plume is defined in $x \in [0, 4]$ and $y \in [-0.5, 0.5]$, outside of which observing "Something" has a probability of 0. On the discretized grid, the odor plume is defined in the area: $x \in [60, 315]$, $y \in [15, 45]$.

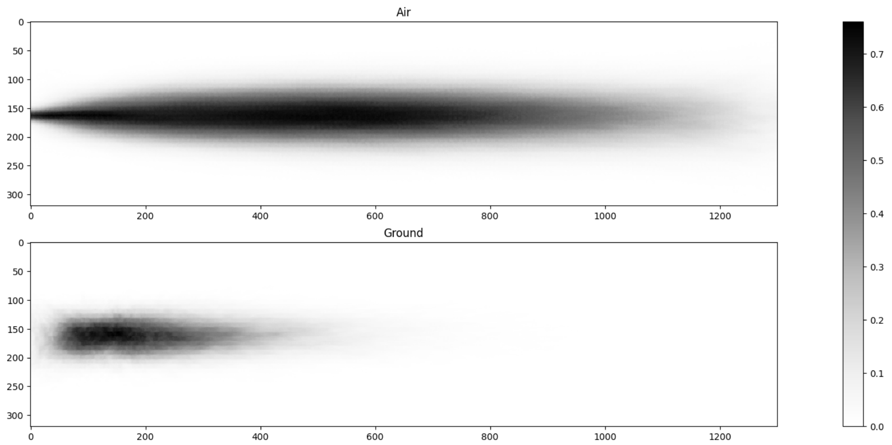


Figure 4.3: The probabilities of observing "Something" in the air or on the ground at $x \in [0, 4]$, $y \in [-0.5, 0.5]$

- b_0: The initial belief, in olfactory navigation, is equivalent to the start probabilities, i.e., all the positions the agents are susceptible to start in. The agent can start anywhere in $x \in [0, 8.5]$ and $y \in [-0.5, 0.5]$, so each state within these ranges has a uniform probability. On the discretized grid: $x \in [60, 315]$, $y \in [15, 45]$.

We can summarize the characteristics of the model in the following visualization (Figure 4.4). The blue rectangle represents the zone where the agent has a uniform probability of starting during simulations. The source, the red dot, is placed at the position $[60, 30]$ in discretized units.

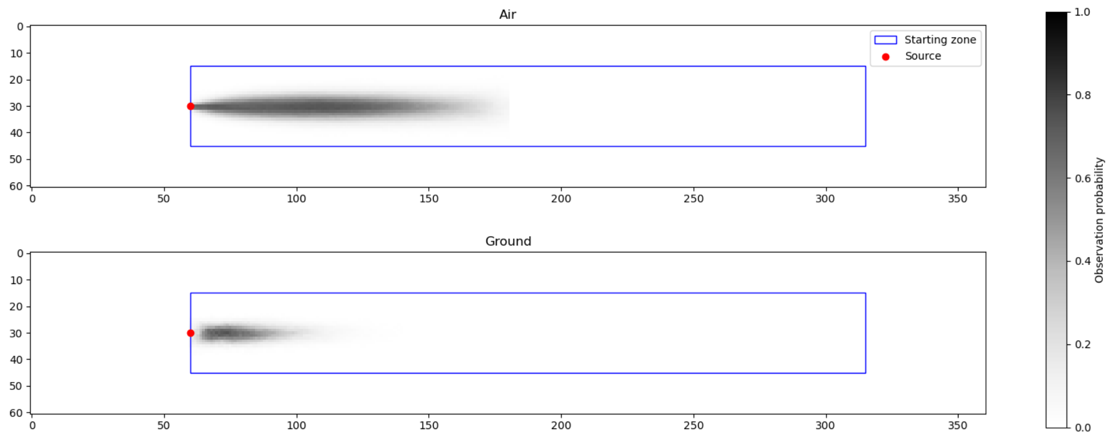


Figure 4.4: A summary of the ground and air model with the source of the odor (red dot), the odor plumes smelling probability in the air and on the ground (greyscale), and the agent's starting zone (blue rectangle).

4.3 Experiments setup

We will run experiments to find the model's optimal strategy (policy). The experiments aim to replicate and expand on the results of the article of Rigolli N. et al. [9] and investigate possible shortcomings of solutions. The experiments will consist of solving the model and running simulations with the solutions resulting from the solving process. Running the simulations will be part of the controlled test that fixes the starting position of the agent at a certain point to draw some conclusions. In this thesis, we will use, in particular, two different test setups: a 3-point test and a grid test.

The solving process for the POMDP model will be done with the Point-Based Value Iteration (PBVI) algorithm (Section 2.2.4). The specific flavor of the PBVI algorithm that will be used in this thesis is the Forward Search Value Iteration (FSVI) algorithm. The depth-first search nature of the expand function of FSVI made it an ideal solver to work on the state space of this scale (22,021 states). As for the specific parameters of the model, we run the algorithm for 300 iterations, as it comes close to what was used by Rigolli N. et al. [9], and more than this would fill the available memory and interrupt the process. For each iteration, a sequence of at most 100 beliefs is generated using the *MDP_explore* algorithm (Algorithm 2.5); this allows for long sequences while not being stopped early too often due to reaching the source. The discount factor (γ) is set to 0.99.

In the paper by Rigolli N. et al. [9], the authors describe a test performed to evaluate the performance of a policy. The test in question consists of running several simulations from 3 different starting points at the opposite side of the starting zone compared to the odor source: $[8, -0.5]$, $[8, 0]$, and $[8, 0.3]$. Then, they look at the simulation's length compared to an optimal trajectory to compute the agent's extra steps to reach the source. For example, starting at $[8, 0]$ on the discretized grid, an optimal path requires 240 steps to reach the source.

In this paper, we also consider this test (we call it the *3-point test*), except the three starting points used are $[8, -0.5]$, $[8, 0]$, and $[8, 0.5]$. Transferred to the discretized grid, they correspond respectively to the points $[300, 15]$, $[300, 30]$, and $[300, 45]$ (Figure 4.5). In this test, 100 simulations are performed starting from each point, for 300 simulations.

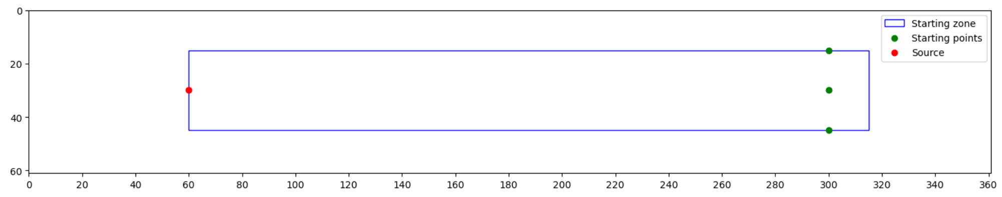


Figure 4.5: Starting points of the simulations relative to the odor source in the 3-point test.

Following this, a more general test called the *grid test* is defined. In this test, we divide the starting space into cells of 10 units square. We make 20 simulations starting at random points within each cell. Given the defined starting zone being $x \in [60, 315]$, $y \in [15, 45]$, we have 26 cells on the x-axis by 3 cells on the y-axis, for a total of 78 cells (Figure 4.6). With 20 simulations per cell, 1560 simulations must be run for a single grid test.

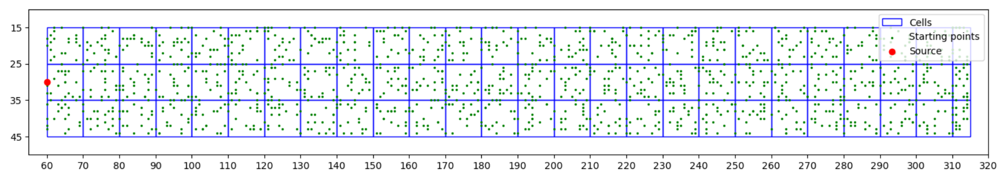


Figure 4.6: An example of starting points for 1560 simulations distributed between the 76 cells of the starting zone for the grid test.

With the 3-point and grid tests, we compute the agent's extra steps to reach the source compared to an optimal path from the simulation's starting point. The optimal path's length is computed by taking the Manhattan distance between the odor source point (os) and the simulation starting point ss: $dist = |x_{os} - x_{ss}| + |y_{os} - y_{ss}|$. We will refer to the performance of a solution as the average amount of extra steps taken for the set of simulations used in the 3-point grid test.

The solver and simulation will be run on a system with the following specifications:

- Processor: 2 AMD EPYC 7413 24-Core running at 2.7GHz (96 threads)
- RAM: 1TB
- GPU: Nvidia A100 with 80GB of graphic memory

If not specified, the solver and simulations are defaulted on the GPU.

4.4 An MDP solution

Taking only the MDP components of the model definition, we can solve the problem as if it were fully observable. The solution can act as an upper bound to the solution obtained in the partially observable setting; however, in our case, the solution will be used to guide the belief expansion of the FSVI solver with the *MDP explore* algorithm (Algorithm 2.5).

To find the solution of the MDP version of the model, we run the Value Iteration algorithm using a discount factor (γ) of 0.99. The algorithm is run until convergence. Convergence is computed by taking the maximum change in the value function. We consider the algorithm is converged if this change is at most $\epsilon = 10^{-6}$. Running the algorithm until convergence, which took 918 iterations and 0.918 seconds, led to the value function and policy in Figure 4.7.

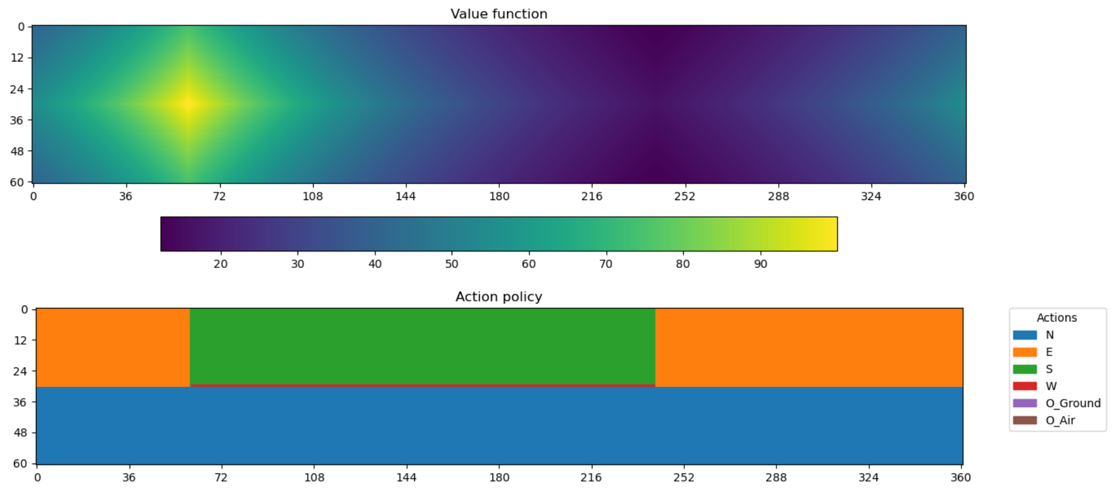


Figure 4.7: MDP Solution of Olfactory Navigation Problem

Note that on a discrete grid, many trajectories lead from any starting position to the source optimally in the minimum number of steps. The algorithm converges to one of the policies, and the choice depends on the order in which the actions are defined. If there is a tie between the values associated with two different actions, the action listed first during the definition will be favored. In our case, the priority is: 1. North; 2. East; 3. South; 4. West. With the actions defined in a different order, the policy map would look different, but the value function would look roughly the same.

4.5 Reproduction of Alternation Results

This thesis aims to reproduce and extend the results shown in the paper by Rigolli N. et al. [9]. To this end, we have run 20 instances of the Forward-Search Value Iteration (FSVI) algorithm to obtain 20 approximately optimal navigation policies.

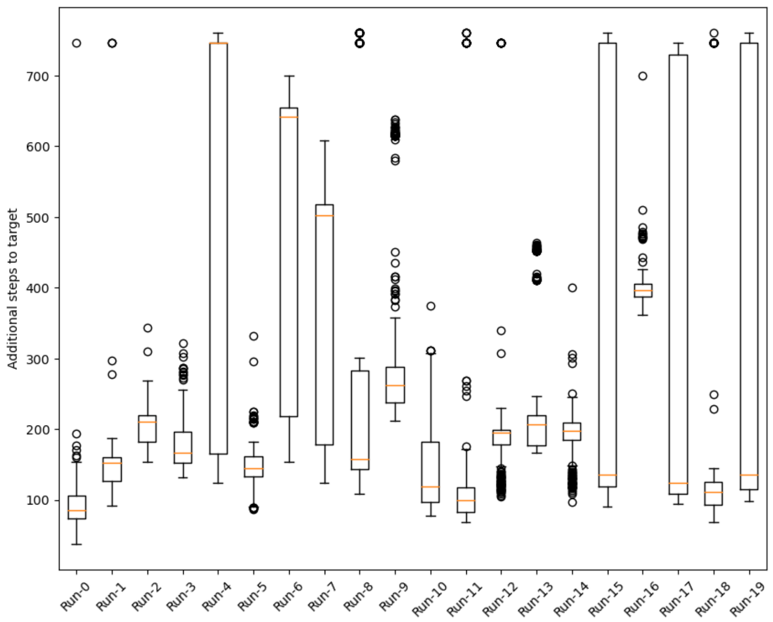


Figure 4.8: Performance variation for the ground and air problem across 20 solving runs using FSVI with 300 iterations each at most 100 belief growth per iteration. Performance computed using the 3-point test with 100 simulations per point.

As shown in Figure 4.8, the performance between the solutions of different solving runs varies significantly. The details of the runs are summarized in Table A.1. Run-2's policy is an example from this set of runs, with good performance without being overly good.

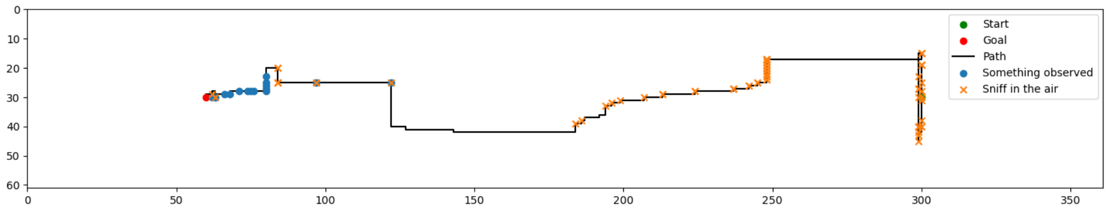


Figure 4.9: Path taken by an agent following the policy that came out as a solution to the Run-2. The agent starts at the green dot ($x = 300$, $y = 30$) and ends at the red dot; the orange crosses represent the agent stopping to sniff the air, and the blue dots are the steps in which some odor cue is observed.

From Figure 4.9, we can see that the agent alternates between phases of casting, where it sweeps the field crosswind while sniffing in the air(column $x = 300$, column $x = 250$), followed by phases where it surges against the wind toward the source(from $x = 300$ to $x = 250$). The lengths of one such surge correspond to the length of the odor plume (it spans 4 units in the x-direction, which translates to 120 discretized units).

Then, we can reproduce the plot "Empirical characterization of the alternation between olfactory sensory modalities" from the paper by Rigolli N. et al. [9]. This visualization is meant to explore the rates of sniffing in the air while casting/surging and far/close to the source. Also, it explores the utility of stopping to sniff in the air by comparing the performances across various simulations and finally seeing how the entropy (information) and value of beliefs evolve throughout a simulation. In more details, the sub-plots are:

(a) Sniffing rate in the air for Close or Far zones.

An agent is considered far from the source outside the air-borne odor plume, where the probability of smelling something in the air is zero. Inversely, it is considered close if the probability of detecting odor is above zero. The rate of sniffing in the air is computed by dividing the number of times the agent has sniffed in the air by the number of steps while the agent is close/far from the source. This rate is computed for simulations; hence, it is visualized as a Probability Density Function (PDF) histogram.

(b) Performance comparison when removing the ability of the agent to smell the air.

We quantify the value of stopping and sniffing the air by defining a new model where action 5 (*Stand still, smell the air*) is removed. This "Ground-only" model can then be solved in the same way as the original model.

(c) Rates of sniffing in the air during Casting and Surging periods.

Casting is a back-and-forth crosswind movement of the agent. We define a series of steps as a *casting* sequence if it contains consecutive steps of movement up, down, or standing still (actions 0, 2, 4, or 5) and is at least three steps long. We consider *surging* a series of consecutive actions upwind or standing still (actions 1, 3, 4, or 5). We count the number of times the agent sniffs the air during cast/surge sequences and divide by the total amount of steps spent casting/surging. This procedure yields one sniffing rate while casting and one sniffing rate while surging for each simulation. Like with the close and far sniff rates, we compute them for simulations and summarize them in a PDF.

(d) Belief entropy and value throughout a simulation.

We can compute the agent's uncertainty at a given time by computing the entropy of the belief. The entropy is computed using Shannon's entropy formula: $-\sum_{s \in S} b_{(s)} ln(b_{(s)})$. We can evaluate the value of a belief by taking: $\max_{\alpha \in V} b \cdot \alpha$. We can evaluate the belief's entropy and value along the simulation steps and plot these computed values in a graph on top of one another.

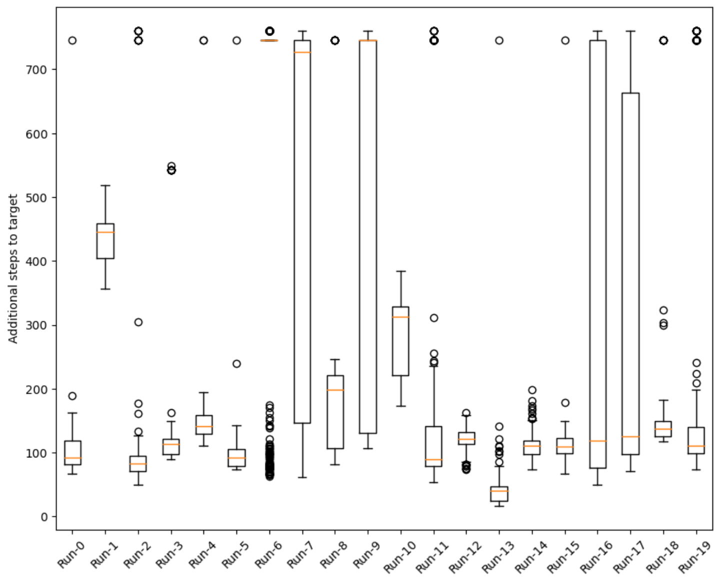
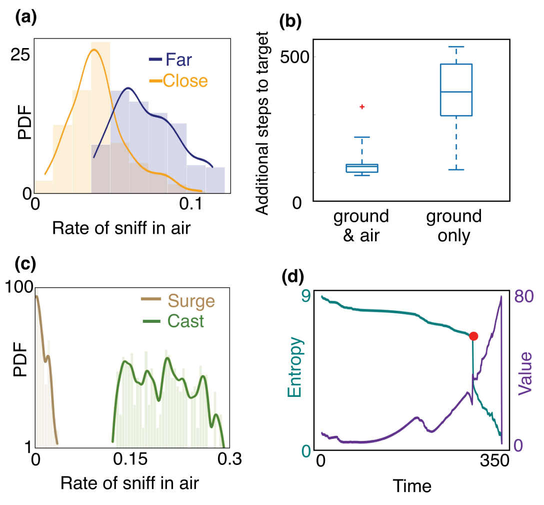


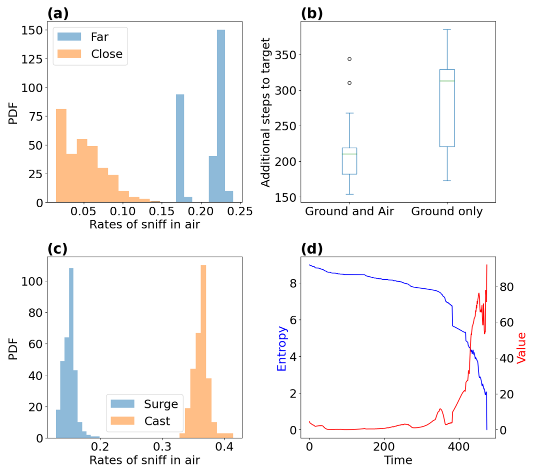
Figure 4.10: Performance for the ground-only problem across 20 solving runs using FSVI with 300 iterations each at most 100 belief growth per iteration. Performance computed using the 3-point test with 100 simulations per point.

Similarly as for the ground and air model, the FSVI algorithm is run to solve the ground-only model in 20 runs. The solutions are then tested using the 3-point test. This leads to the following performances in Figure 4.10.

From all the runs in Figure 4.10, a single solution is picked to make the comparison plot with the Rigolli N. et al. [9] paper. With the same reasoning as for the ground and air model, the solution of Run-10 is used as it displays average performances.



(i) Rigolli N. et al. [9] results



(ii) This paper's recreation of the results

Figure 4.11: Empirical characterization of the alternation between olfactory sensory modalities. Comparison between Rigolli N. et al. [9] and this paper's implementation.

We note that the results are overall reproduced with some minor variations. For example, the shapes of the PDFs of plots **(a)** and **(c)** are different as they are tied to the learned policy for the problem, which is different every time due to the randomness of the initial conditions during the solving. However, the spirit of the plots remains the same: the rate of sniffing in the air is higher when being far away from the source and while casting compared to when the agent is close to the source or surging. As shown in Appendix A, the spirit of the plots **(c)** (Figure A.3) and **(d)** (Figure A.4) retain the same spirit no matter the performance of the run. The plot **(a)** (Figure A.2) tend to vary more between runs, but generally, the "Rate of sniff in air" tends to be higher when far away from the source compared to when being close the source.

We can also note that to reproduce the results of Rigolli N. et al. [9], we picked a typical solution from the 20 solutions we obtained. As stated, the solution of Run-2 was picked for the ground and air model, while the solution of Run-10 was chosen for the ground-only model. This choice mainly affected the characteristics displayed by plot **(b)**. As we can see from Figure 4.8 and Figure 4.10, the performance varies drastically between solving runs. The reason for these variations will be explored in the following section.

4.6 Variation of solver performance

To inspect the variation in performance, we can take the solutions generated by specific runs and see how they fare across a more comprehensive array of starting positions. This more comprehensive test was described as the grid test (Section 4.3). We choose three runs displaying various performance characteristics with the 3-point test and apply the grid test to their solutions. We can then plot the average performance over each cell as a grid.

The runs chosen based on their performance, as shown in Figure 4.8, are the following:

- Run-0 is a run with excellent performance compared to the rest.
- In Run-9, the performances are more average, and we can see three performance clusters.
- Run-16 displays poor performances compared to the rest of the runs.

Note that the points used in the 3-point on $x = 300$ and at $y \in \{15, 30, 45\}$. The points are associated with the cells $[305, 20]$, $[305, 30]$, and $[305, 30]$ in the grid test.

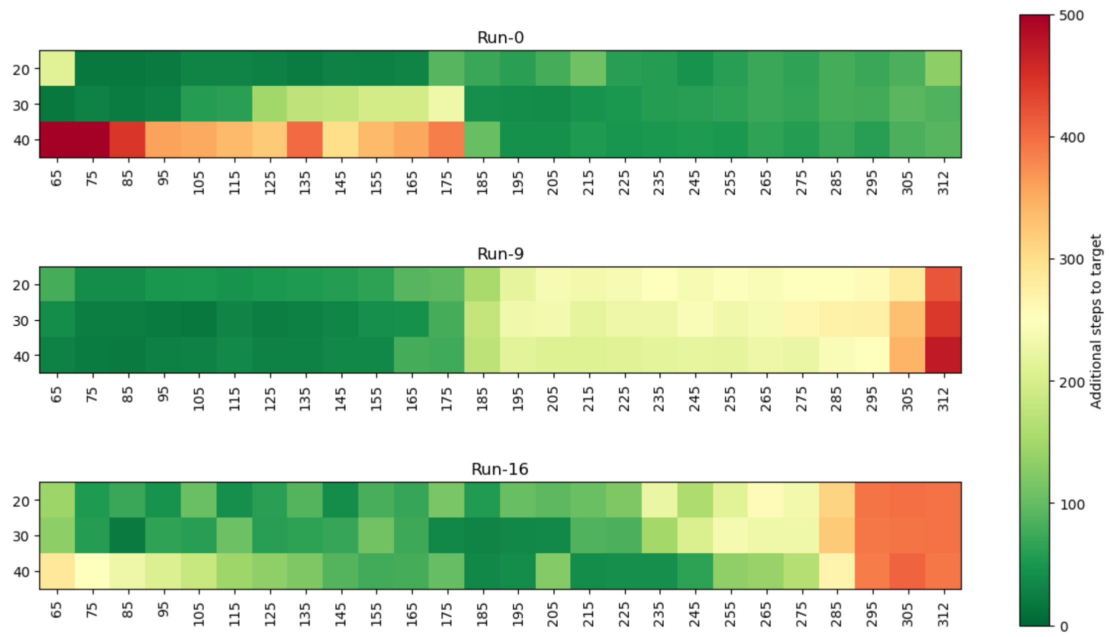
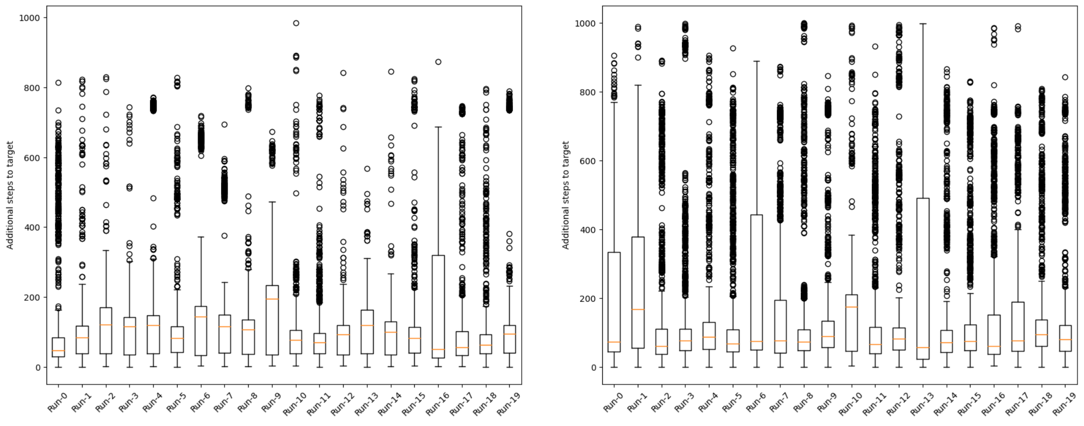


Figure 4.12: Average performance of 20 simulations in cells of 10 by 10 discretized units using the solutions of runs 0, 9, and 16

Overall, the performance shown by the 3-point test is not representative of the general performance of a solution. The 3-point test represents a local performance metric, arbitrarily chosen to be at $x = 300$, on the furthest possible side from the source in the possible starting locations. It is a good metric if we only consider the possibility of the agent starting in $x = 300$. However, we want to consider a more general metric that encompasses the performance from any possible starting point. For example, in Run-0, the performance shown by the 3-point test was outstanding, but now, with the grid test, we can see a starting area under the source where the performance is very poor. Meanwhile, in Run-16, the performance is worse across the starting space, but the points displaying bad performance show fewer bad performances than the worse ones of Run-0.

A notable characteristic of the performances across all the possible starting positions is that some zones perform significantly worse than the grid average, and such for all runs picked. We can wonder why this pattern of having a poorly performing zone, like the Achilles heel, appears for all the solutions. This underperformance will be explored further later in this thesis.

Since the grid test has been done for all the runs, we can plot the spread of the extra steps required to reach the source in box plots in the same fashion as Figure 4.8 and Figure 4.10.



(i) Ground and air (ii) Ground only

Figure 4.13: Performance variation across 20 different runs with the performance being evaluated with the grid test (20 simulations per cell of 10 by 10 discretized units)

From this new plot, the number of extra steps is less stable for the ground-only model than for the ground-and-air model. The medians, however, have stood in about the same zone (between 100 and 200 extra steps). To get a better appreciation of the average performances of the various runs for both models, we can generate a box plot of the averages of the amounts of extra steps to the source:

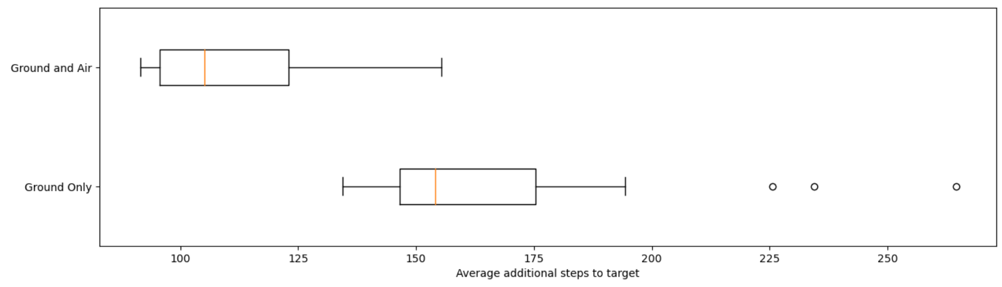


Figure 4.14: Spread of the average performances with the grid test comparing the ground and air model and the ground-only model.

Comparing the averages, we can interpret that, most of the time, a solving run of the ground and air model will produce a solution performing better on average than the solution of a solving run of the ground-only model. This better performance confirms that, as stated in the Rigolli N. et al. [9] paper, the ability of the agent to pause and sniff the air to gain more information is a valuable action most of the time. The performance variation between the solutions of different runs can still be asked.

To understand the reason for the performance discrepancies, we hypothesize that it could either be caused by the solving process not having reached convergence yet or caused by the randomness involved in the FSVI solving procedure.

4.6.1 Is the Forward Search Value Iteration Process Converging?

To investigate the convergence of the solving process, we save the solution every 10 iterations and let the FSVI solving process run for 600 iterations instead of the usual 300 iterations we used thus far. We run the grid test for every recorded solution, so 1560 simulations. This set of simulations gives us a set of performance values every 10 iterations. We can then plot the average performances with the error bars representing the 20 and 80 percentiles:

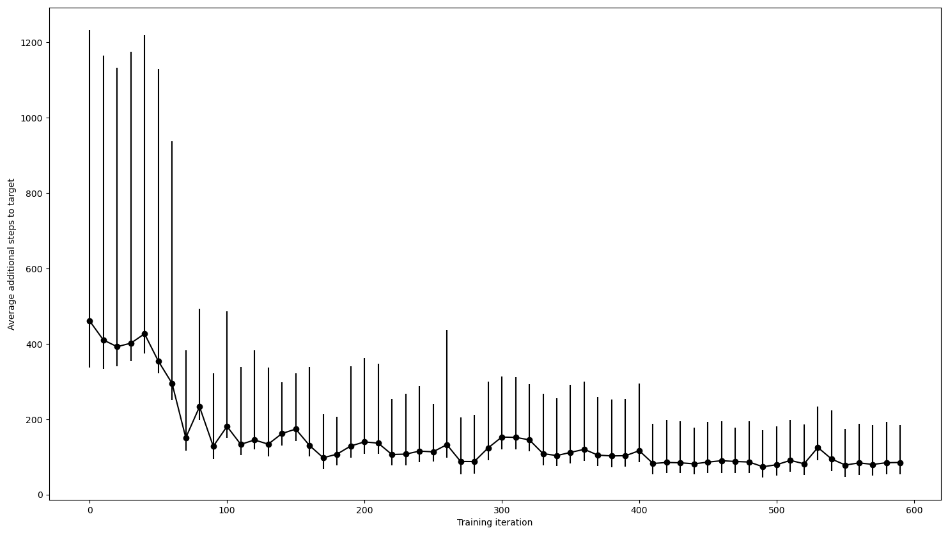


Figure 4.15: Performance measured using the grid test every 10 iterations of the training process with FSVI. The solving is run for 600 iterations with at most 100 beliefs generated in each iteration. The error bars represent the 20th and 80th percentiles of the performance values.

From iteration 80, the solution's performance has reached a satisfactory point, but the 80th percentile is still relatively high. After the 80th iteration, we can see that the average performance stays around that level or decreases slightly. We can also note an improvement in the 80th percentile of the performance after 400 iterations. In conclusion, we do not sacrifice much performance when stopping the solving process after 300 iterations.

4.6.2 How does the Randomness in the Forward Search Value Iteration Affect the Solving Process?

Regarding the second hypothesis, which is that the variation in performance could come from the randomness in the solving process, we can visualize the belief sequences generated. As described previously, beliefs are generated by performing a simulation from a random state and following the policy of the MDP version of the problem. It leads to a sequence of states and actions, and with observations sampled from the observation probabilities for the given states and actions, we can find a sequence of beliefs.

We can visualize the trajectories of states generated throughout the FSVI-solving process by selecting 300 random initial points corresponding to the 300 iterations. From these points, trace the states generated following the MDP policy until the source is reached or for 100 steps.

In Figure 4.16, the green points represent the random starting points, and the black lines represent the trajectories taken from these initial points. The trajectories stop either when reaching the odor source, the red dot, or when reaching 100 steps. In this figure, the background colors represent the policy of the MDP solution, as shown in Figure 4.7, with the following color-action mappings: blue-north, orange-east, green-south, and red-west.

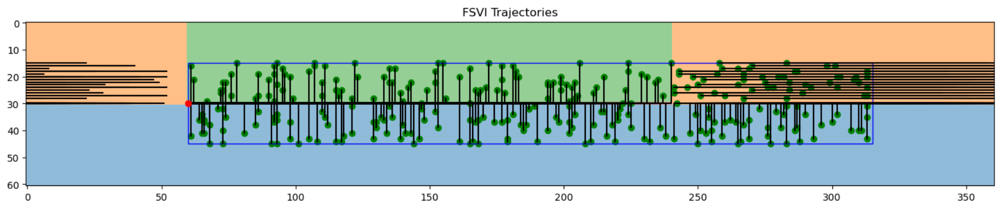


Figure 4.16: Set of trajectories generated from 300 initial states (green dots), for at most 100 steps, following the policy of the MDP solution to the ground and air problem. The MDP policy is represented by the overlayed colors (blue-north, orange-east, green-south, and red-west).

We note that the sequences generated are repetitive due to the simplicity of the policy of the MDP model. Another randomness factor in FSVI belief generation is the observations picked along the path. This randomness leads to the sequences of beliefs generated from an equal initial state to differ depending on the observations chosen along the path.

Another notable characteristic that can be raised from Figure 4.16 is the set of sequences generated from points starting in the northeast corner of the starting zone. Since the boundary conditions of the model allow for the agent to traverse the east edge of the field and appear at the west, the MDP model's policy dictates that the fastest way to reach the source when this far east is to walk east until column 60 and then walk south until the source is found. It is an unwanted behavior, as it does not reflect how it would be in reality (the agent would not be placed downwind from the source and expected to follow the wind to find the source). It is also a behavior that generates a sequence of beliefs that leads to no gain in information about the agent's position. As the odor plume is in $x \in [60, 180]$, we can see that the trajectories will never cross the plume and never gain information from smelling something. Moreover, since the sequences of actions always go east, with no variation (it never enters the odor plume, so no observation is made), the sequences of beliefs generated from these points will all be the same.

As a lucky hazard, we can note that the movement to the east could lead to some positive behavior from the agent, as it could incorporate in the solution a mechanism where the agent could backtrack on its steps if there is the possibility that it went too far and passed the source.

4.6.3 Proposed Solutions

Alternative solving processes should be investigated to resolve the first potential issue. Such a solving technique should keep a notion of learning the optimal trajectory using the MDP policy. However, some exploration should allow the agent to learn an information-gathering strategy. In our case with FSVI, the actions used to generate the sequences will always be one of the four first actions (move north, east, south, or west while sniffing the ground), as when the environment is fully observable, the action of stopping is detrimental to the policy. However, it might be beneficial to incorporate the other action of pausing and sniffing the air into the belief sequences explored. The belief generation process of FSVI is greedy; therefore, an exploratory mechanism could be incorporated to allow for better coverage of the belief space. Furthermore, we could implement an epsilon-greedy solver to perform this trade-off between exploration and exploitation.

Then, for the second issue of allowing the agent to approach the source from the west, we propose implementing an alternative model where the east and west boundary conditions are modified. Instead of allowing the agent to be sent to the other side of the field, we could implement these boundaries as walls. However, we have to pay attention to whether there is enough padding between the starting belief (starting zone) and the actual border for the agent not to learn to use the border to its advantage to gain information about its relative position to the source.

4.7 Runtimes of the Solution

With this paper's implementation of the PBVI solver, which can utilize multi-threading and the GPU, we also wanted to compare the runtime performances to the C++, single-threaded implementation of the Rigolli N. et al. [9] paper. A new solving run is performed with FSVI on the ground and air model to compare the two implementation's runtimes.

As stated in the Rigolli et al. [9] paper, the solution for 310 iterations took with their implementation around 2 days. With this implementation, 300 iterations can be run between 3 and 4 minutes.

The runtime of the expand function during each iteration is, on average, 0.06 seconds, and it generates, on average, 81 belief points. A more interesting concept to visualize is how long the backup function will take over the course of the solving, as the function depends on the size of the value function.

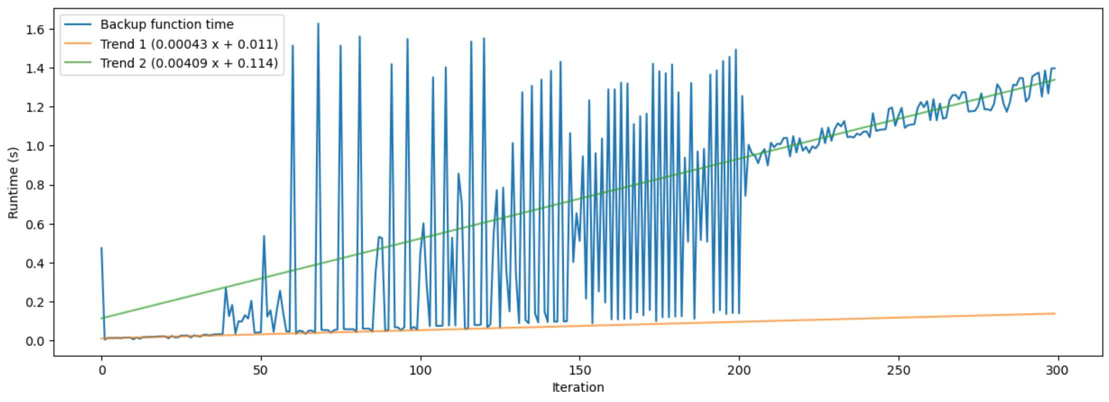


Figure 4.17: Runtimes of the backup operation throughout 300 iterations of the FSVI solving process of the ground and air model with at most 100 beliefs per iteration.

We can see the variation in runtimes between iterations 50 and 190; it correlates to the times when the GPU memory fills up, leading to the garbage collection being called to free memory blocks. Then, after this, the GPU memory becomes so full that the garbage collection works constantly at the cost of time but decreases the number of peaks. In Figure 4.17, we can see two trends. Trend 1 corresponds to the iterations where the garbage collection system is not involved, resulting in a growth of runtimes by 0.4ms per iteration. In trend 2, the growth is a magnitude higher, with a 4ms increase in runtime per iteration.

We can see from Figure 4.17 that, besides the spikes, the runtime increases linearly with iteration counts and, therefore, with the number of alpha vectors in the value function. This correlation is also confirmed by Figure 4.18, where the runtimes are divided by the number of alpha vectors in the value function. These figures also show that when the garbage collection system is not involved in the runtimes, we have a runtime of 0.03 ms per iteration per alpha vector. Then, with the garbage collector running, the runtimes are 0.2 ms per iteration per alpha vector in the value function.

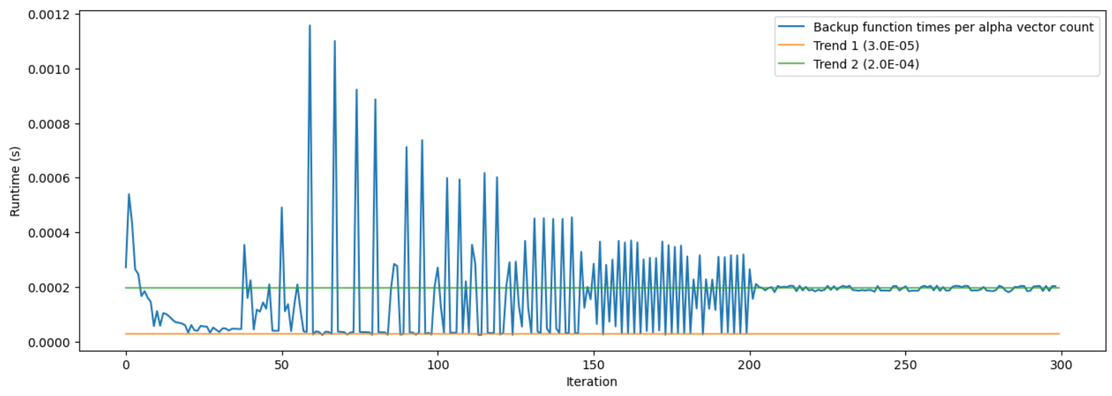


Figure 4.18: Runtimes of the backup operation scaled by the number of alpha vectors in the value function. It is plotted over 300 iterations of the FSVI solving process of the ground and air model with at most 100 beliefs per iteration.

The garbage collection system has to be triggered due to the memory limit of the GPU, which is 80GB. What would these performance trends look like if run without memory limitation? To do this, we can run the solving on the CPU only; the server on which it is run has 1TB of RAM, allowing for the memory limitation to be lifted. Running on the CPU leads to a performance trade-off, however, as the highly optimized matrix operations of GPU cannot be utilized anymore. However, the algorithm is still multi-threaded, allowing for faster performance than the implementation of Rigolli N. et al. [9], as it finishes the 300 iterations of FSVI in 49.72 minutes.

We can see in Figure 4.19 that the runtimes follow a single, almost linear trend. Once again, we can see from Figure 4.20 that if adjusted for the number of alpha vectors in the value function, we see a trend of 2.8 ms per iteration per alpha vector present in the value function.

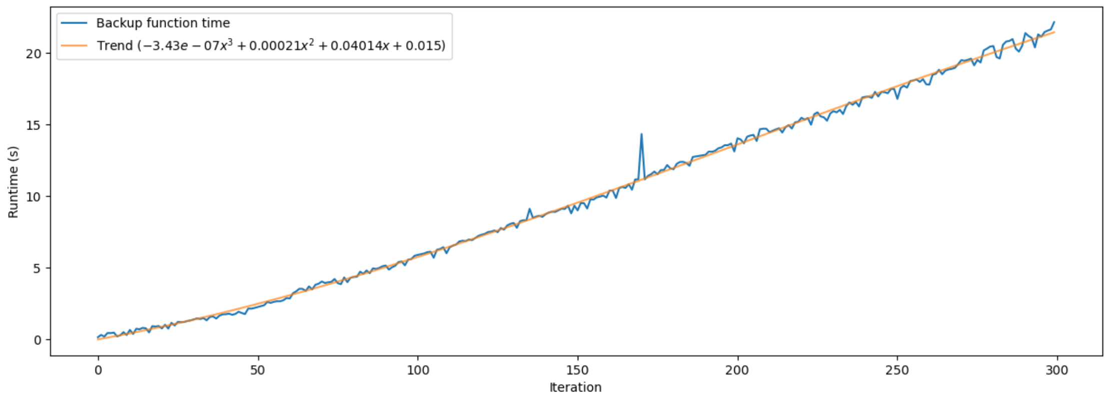


Figure 4.19: Runtimes of the backup function throughout 300 iterations of the FSVI solving process of the ground and air model, with at most 100 beliefs generated per iteration.
The solving process ran exclusively on the CPU.

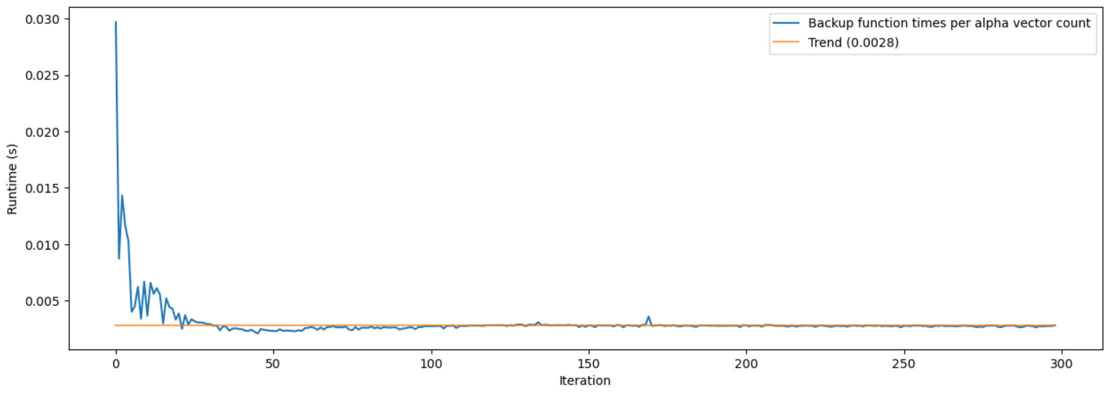


Figure 4.20: Runtimes of the backup operation scaled by the number of alpha vectors in the value function. It is plotted over 300 iterations of the FSVI solving process of the ground and air model with at most 100 beliefs per iteration.
The solving process ran exclusively on the CPU.

4.8 Runtimes of the Simulations

The solution generated by the solver can be used to perform a simulation, in other words, having an agent apply the policy learned when put in the environment. Rigolli N. et al. [9] report that running such a simulation with their implementation takes up to 10 hours. On the other hand, our implementation runs a single simulation in half a second on average (Figure 4.21) or 45.2 seconds for 1000 simulations in parallel.

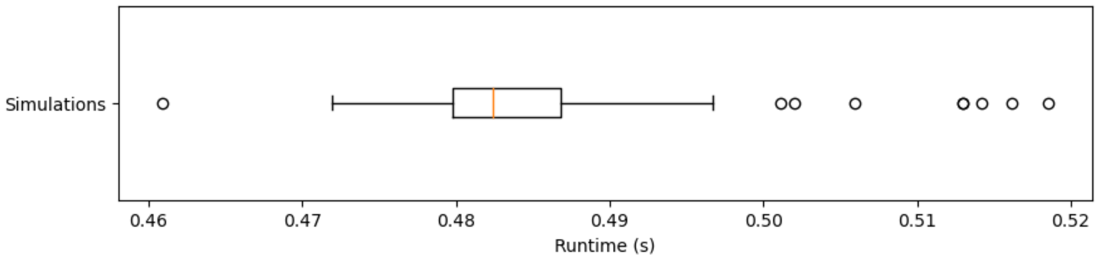


Figure 4.21: Runtimes of 100 simulations run sequentially with the solution of Run-0, an initial state of $[300, 60]$, and a maximum of 300 steps.

Figure 4.22 shows that our parallel simulation implementation displays runtimes one order of magnitude better. This different implementation proves that the advantages of the parallel implementation outweigh the additional cost of the more significant computations required. We trade memory usage with runtime as the stored value function has to be fetched from memory only once in the parallel implementation but for a heavier operation.

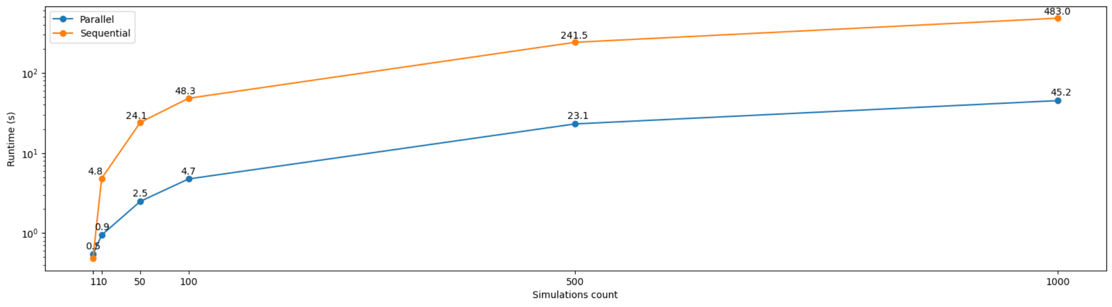


Figure 4.22: Runtime comparison for a set of simulations (1, 10, 50, 100, 500, and 1000) sequentially or in parallel with the solution of Run-0, an initial state of $[300, 60]$ and a maximum of 300 steps. The sequential implementation assumes a linear increase in runtime for each additional simulation, with 0.483 seconds per simulation.

Chapter 5

Conclusion

5.1 Results

In this thesis, we successfully reproduced the results displayed in the article by Rigolli N. et al. [9]. The comparison confirms that our version of the Point-Based Value Iteration (PBVI) algorithm is sound and valid. We managed to show the agent exhibiting casting and surging behaviors. Following this, we display a plot showing various characteristics of alternation for a set of simulations; this plot is directly compared to an equal one in [9] (Figure 4.11). Even though we observe considerable variations in performances between runs, the casting and surging behavior characteristics remain similar across the various runs.

More thorough tests could be performed thanks to the improved runtime performance of this thesis's PBVI implementation. The first extended test performed was to perform multiple soluting runs and compute the performances with sets of simulations for each solution. This set of simulations consists of 100 agents, starting from 3 different starting points far from the source (3-point test). The second extended test is an alternative to the simulation set. Instead of considering only 3 points from which to start the simulations, we divide the starting space into equal cells and start ten simulations randomly inside each cell (grid-test).

The first extended experiment shows that the performances vary considerably between solving runs. These significant variations lead us to question where this variation comes from. The second experiment was designed to explore where the variance could come from. From this later experiment, we noted that the 3-point test does not represent the complete picture of how an agent will perform. In the complete starting zone, we can see that depending on the area the agent starts in, the extra steps needed to reach the source have significant differences. Then, although some variance still exists, the average performances are more stable under this test.

Then, we checked whether the Point-Based Valued Iteration had converged by letting it run for twice the number of iterations. Due to the memory limitation of the GPU, this had to be performed on the CPU instead. Evaluating the performance every ten iterations, we could see that it had already plateaued before 300 iterations. This result led us to conclude stopping the algorithm after 300 iterations was valid.

We also showed that the runtime of the backup function grows linearly with the size of the value function. We also showed that the garbage collection system occupies a significant portion of the runtimes of the backup step. It represents iterations running one order of magnitude slower than when the garbage collection is not involved.

Finally, we showed that the parallel implementation of the simulation process proposed by this thesis displays runtimes that are an order of magnitude better than the sequential implementation.

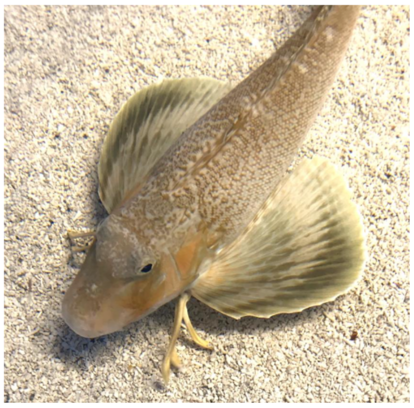
5.2 Future work

Possible work to further the research beyond this thesis results includes continuing the investigation into the significant performance variations. As suggested in Chapter 4, we could explore alternative POMDP model definitions. One alternative model could have different boundary conditions; instead of wrapping around in both axes, the agent could be allowed to wrap around only on the crosswind axis. Then, the amount of states in the margins could be changed to see how it affects the performance variations. Other solving strategies could also be used to find a solution, such as different flavors of the PBVI algorithm. We proposed changing the FSVI algorithm to include epsilon-greedy action choice to generate beliefs and allow for a more complete exploration.

Next, we can ask how robust an agent trained with Point-Based Value Iteration is. We could test the robustness by changing the signal the agent receives during simulation by changing the observation probabilities. We could imagine the wind blowing the scent more or less far than the agent thinks. This difference would mean the likelihood of making an observation comes from a probability distribution stretched or squished in the direction of the wind compared to what the agent has in its model. We can also imagine the odor spreads more or less across the wind than the agent thinks. We can then test the robustness of the agent using the grid test for a range of increases and decreases in the probability distribution width and height. The test would also be able to give information about the starting position of the agent relative to the source for both the original odor probabilities and the modified probabilities.

Another potential future research could be to see how the two reinforcement learning paradigms compare against one another for the problem of olfactory navigation. In this thesis, model-based reinforcement learning was investigated, but how could we formulate olfactory navigation as a model-free problem? In a model-free setting, we are in a case where the agent has no prior information about the environment. The agent, therefore, has to learn through experience and cannot plan. In a model-free setting, this learning process can happen through tabular q-learning. We can imagine that the training, although simpler for the model-free problem, could take longer, while the choice of what action to take during simulations could take more time for the model-based problem due to how the value function is built, as we have seen in this thesis. The comparison could happen more systematically by comparing the computational and storage complexity of the training and simulation steps. Then, the solutions could be compared using other criteria, such as the conditions in which one excels compared to the other.

Ongoing research within the PiMLB unit consists of studying Sea Robins. Sea Robins are fish with filaments akin to legs, which they use to forage under the sand. We ask how they can track food under sand and alternate sensing with their legs versus their nose. Experiments were performed where the fish was observed trying to find a piece of food under the sand. The Model-based Reinforcement Learning technique of Point-Based Value Iteration can be used to investigate this question; conceptually, the fish can be seen as an agent that can alternate between two sensory modalities, as in the problem discussed in this thesis.


(i) Sea Robin (credit: C. Allard, N. Bellono (unpublished))

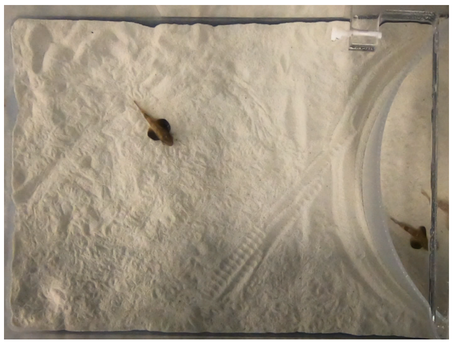

(ii) Experiment with Sea Robin (credit: D. Lagomarsino-Oneto, N. Rigolli, C. Allard, A. Seminara, N. Bellono (unpublished))

Figure 5.1: Illustration of a sea robin fish, showing their leg-like filaments (i). Then, a still frame of a video part of an experiment where the sea robin attempts to find a piece of food hidden under the sand (ii).

Bibliography

[1] Eugene Balkovsky and Boris I Shraiman. “Olfactory search at high Reynolds number”. In: *Proceedings of the national academy of sciences* 99.20 (2002), pp. 12589–12593.

[2] Jacques F Benders. “Partitioning procedures for solving mixed-variables programming problems”. In: *Numer. Math* 4.1 (1962), pp. 238–252.

[3] Anthony R Cassandra, Leslie Pack Kaelbling, and Michael L Littman. “Acting optimally in partially observable stochastic domains”. In: *Aaai.* Vol. 94. 1994, pp. 1023–1028.

[4] Milos Hauskrecht. “Value-function approximations for partially observable Markov decision processes”. In: *Journal of artificial intelligence research* 13 (2000), pp. 33–94.

[5] Robin A Heinonen, Luca Biferale, Antonio Celani, and Massimo Vergassola. “Optimal policies for Bayesian olfactory search in turbulent flows”. In: *Physical Review E* 107.5 (2023), p. 055105.

[6] Michael L Littman, Anthony R Cassandra, and Leslie P Kaelbling. *Efficient dynamic-programming updates in partially observable Markov decision processes.* 1995.

[7] Joelle Pineau, Geoff Gordon, Sebastian Thrun, et al. “Point-based value iteration: An anytime algorithm for POMDPs”. In: *Ijcai.* Vol. 3. 2003, pp. 1025–1032.

[8] Jolle Pineau, Geoffrey Gordon, and Sebastian Thrun. “Point-based approximations for fast POMDP solving”. In: *Proc. Int. Symp. on Robotics Research.* 2005.

[9] Nicola Rigolli, Gautam Reddy, Agnese Seminara, and Massimo Vergassola. “Alternation emerges as a multi-modalstrategy for turbulent odor navigation”. In: *elife* (2022). DOI: `https://doi.org/10.7554/eLife.76989`.

[10] Stuart J Russell and Peter Norvig. *Artificial intelligence a modern approach.* Third. London, 2010.

[11] Jeffrey E Segall, Steven M Block, and Howard C Berg. “Temporal comparisons in bacterial chemotaxis.” In: *Proceedings of the National Academy of Sciences* 83.23 (1986), pp. 8987–8991.

[12] Guy Shani, Ronen I Brafman, and Solomon Eyal Shimony. “Forward Search Value Iteration for POMDPs.” In: *IJCAI*. Citeseer. 2007, pp. 2619–2624.

[13] Boris I Shraiman and Eric D Siggia. “Scalar turbulence”. In: *Nature* 405.6787 (2000), pp. 639–646.

[14] Trey Smith and Reid Simmons. “Heuristic search value iteration for POMDPs”. In: *arXiv preprint arXiv:1207.4166* (2012).

[15] Matthijs TJ Spaan and Nikos Vlassis. “Perseus: Randomized point-based value iteration for POMDPs”. In: *Journal of artificial intelligence research* 24 (2005), pp. 195–220.

[16] Massimo Vergassola, Emmanuel Villermaux, and Boris I Shraiman. “‘Infotaxis’ as a strategy for searching without gradients”. In: *Nature* 445.7126 (2007), pp. 406–409.

[17] Erwin Walraven and Matthijs Spaan. “Accelerated vector pruning for optimal POMDP solvers”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 1. 2017.

Appendix A

Detailed Simulation Results

This appendix contains tables and figures that give the complete results of the ground-and-air solving runs using Forward Search Value Iteration (FSVI). First, we will show the statistics of the solving runs (Table A.1).

Then, with the solutions of the 20 runs, the first 500 steps of a random simulation from the point $[300, 30]$ are shown (Figure A.1). It illustrates the behavior an agent will take with a given solution.

Additional results of the 3-point test simulations are then shown. These additional figures reproduce the Rigolli N. et al. [9] paper results for all 20 solutions. Figure A.2 reproduces the sub-graph **(a)** and shows the sniffing rates in the air when the agent is far/close to the source. Figure A.3 reproduces the sub-graph **(c)** showing the rates of sniffing in the air during casting/surging sequences. Figure A.4 reproduces the sub-graph **(d)** showing the belief entropy versus its value along a simulation.

Finally, Figure A.5 shows the average number of extra steps for each 10 by 10 cells in the starting zone for every 20 solutions.

Run	Average beliefs per iteration	Total solve time (s)	Final alpha vector count	Iterations
0	84.76	193.9558	7402	291
1	82.02	205.8844	7194	300
2	85.26	205.5065	7282	279
3	84.24	191.9555	6737	284
4	85.87	213.2578	7069	289
5	85.26	219.8330	7583	300
6	85.56	210.4253	7234	300
7	85.08	169.3911	6491	269
8	80.22	215.3880	7221	300
9	82.20	200.3131	7045	300
10	83.15	224.5018	7659	300
11	83.76	220.9853	7469	300
12	85.03	222.1257	7896	298
13	86.51	215.0722	7008	300
14	86.22	217.0440	7496	300
15	84.39	180.2524	6729	275
16	87.54	224.5297	7760	291
17	82.35	192.3629	7018	286
18	82.40	191.9101	6913	276
19	84.32	198.7912	6976	300

Table A.1: Ground and air model, statistics of 20 runs of FSVI with at most 300 iterations, at most 100 beliefs generated per iteration, and $\gamma = 0.99$. It is applied to the ground and air model.

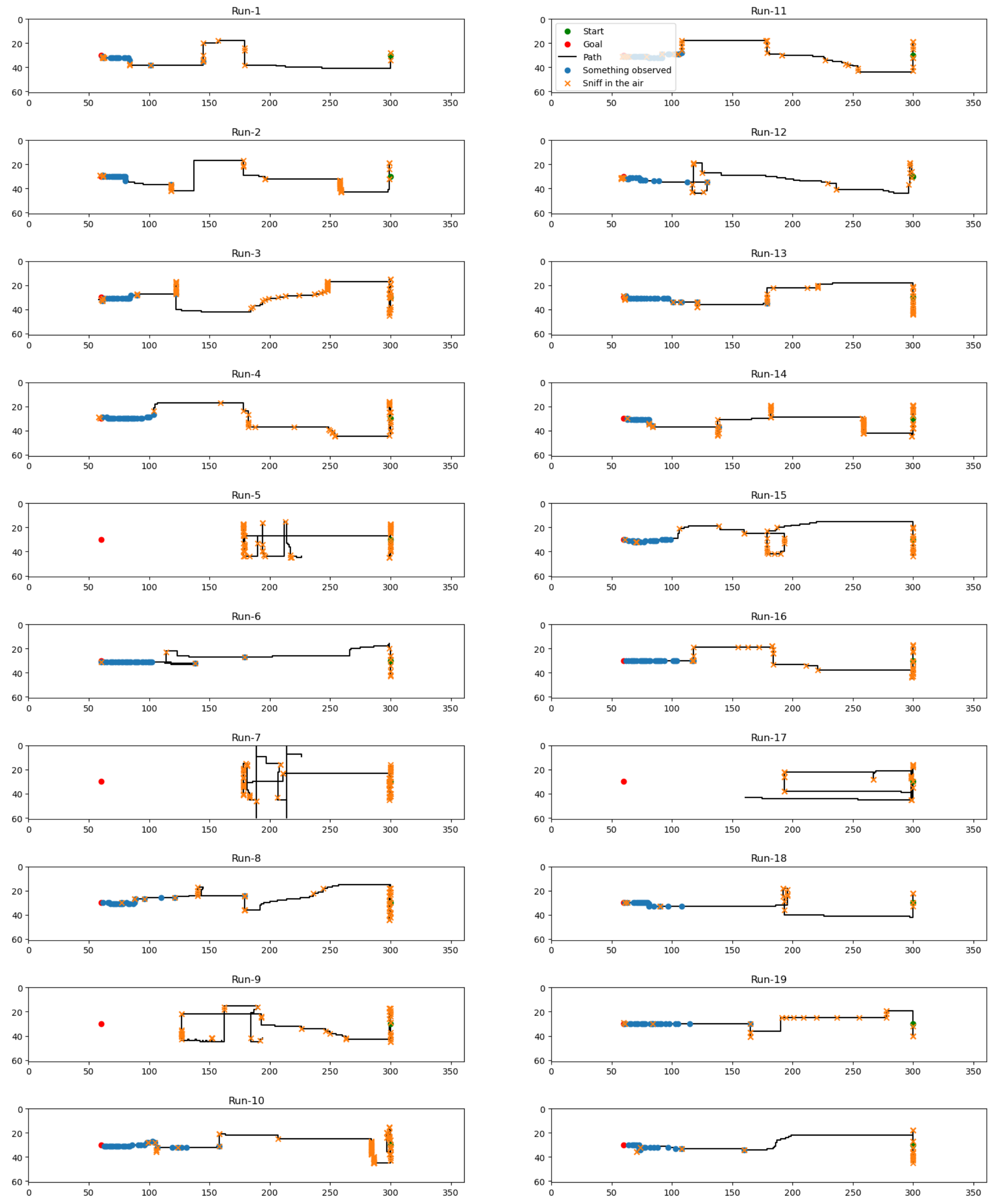


Figure A.1: Simulations up to 500 steps following the policy of the solution of 20 runs of FSVI with at most 300 iterations, at most 100 beliefs generated per iteration, and $\gamma = 0.99$. It is applied to the ground and air model.

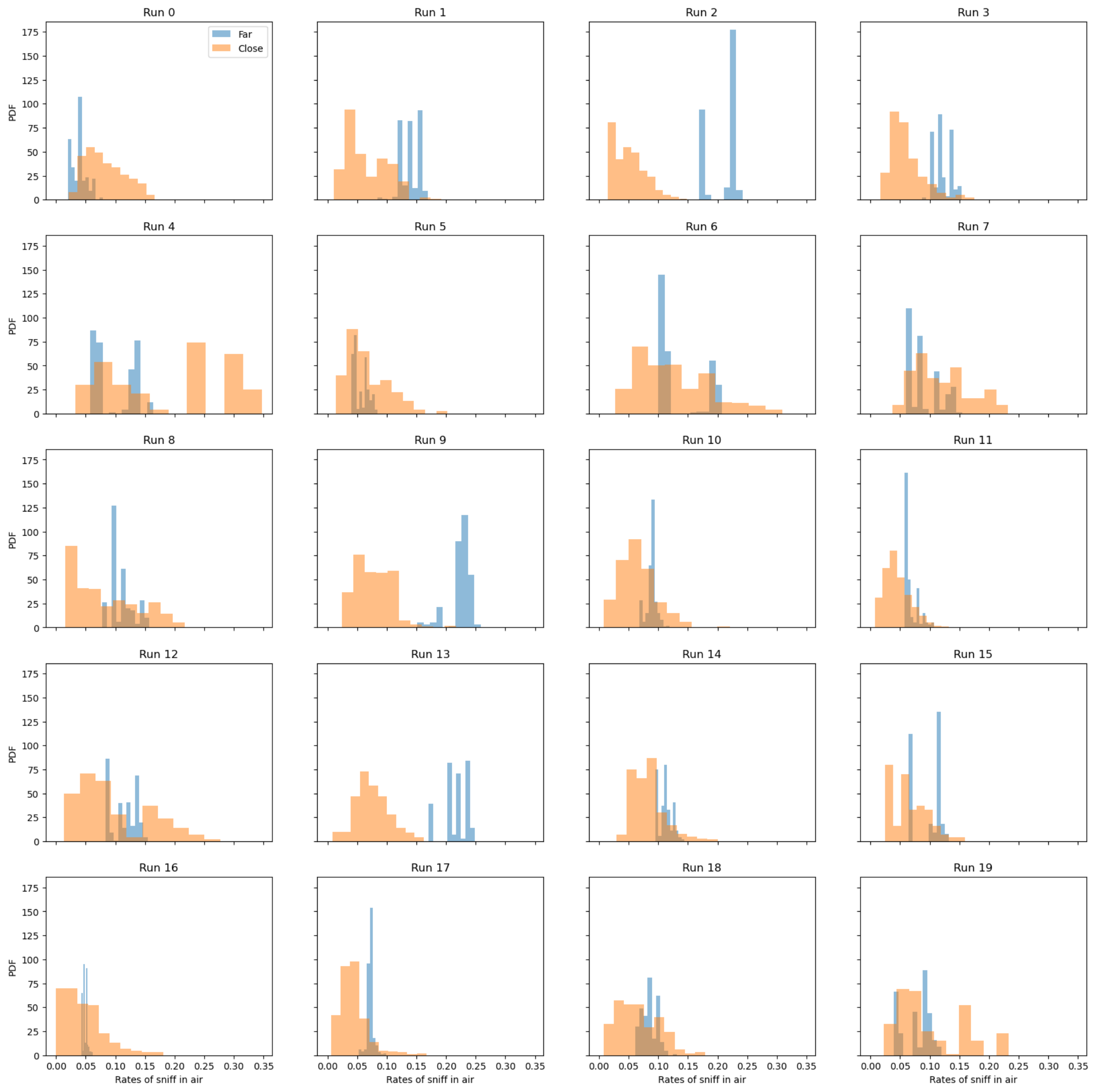


Figure A.2: Histograms of the rates of sniffing in the air while the agent is far/close to the source (sub-plot **(a)**) for the 20 run's solutions. An agent is considered close to the source if it is within the probability plume of smelling an odor.

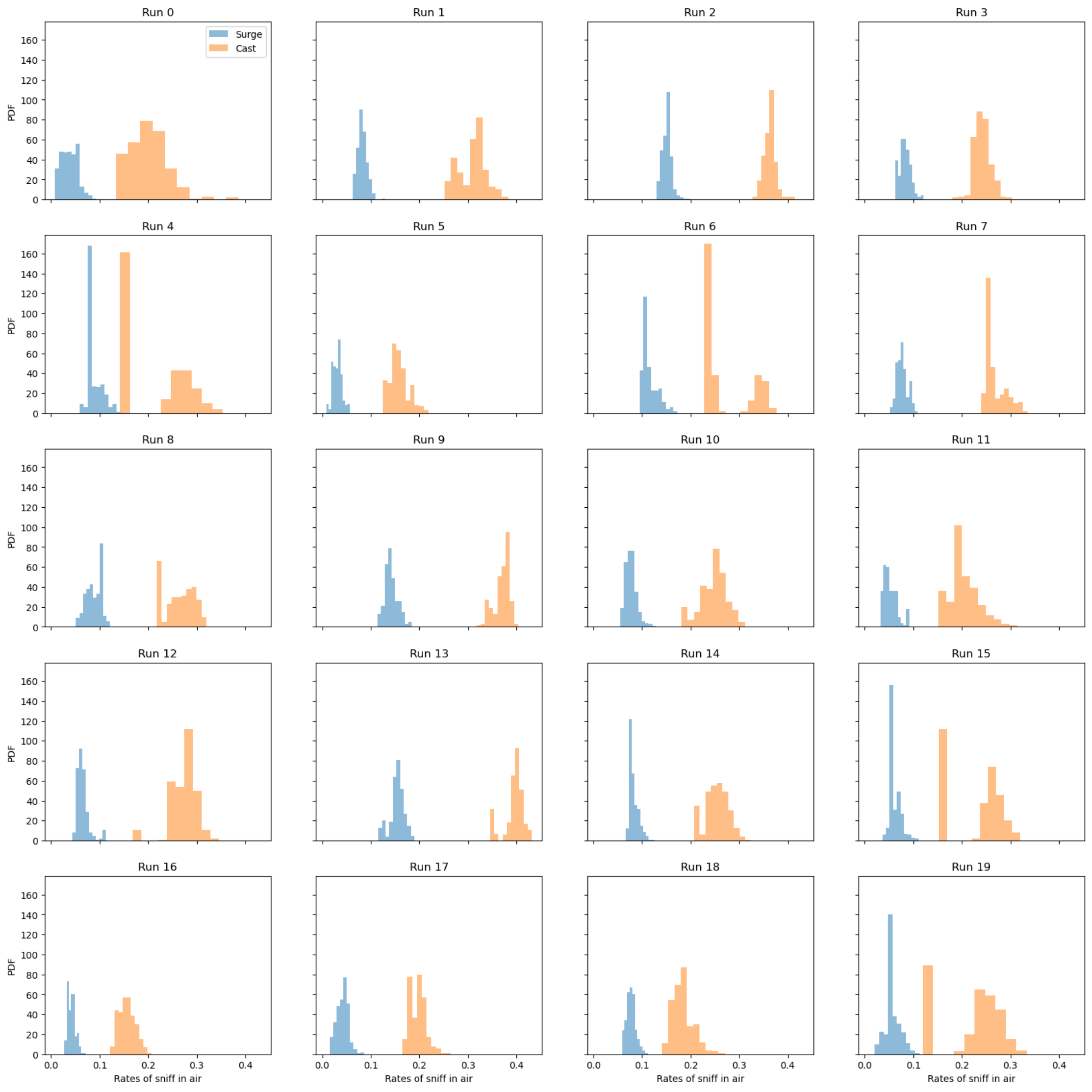


Figure A.3: Histograms of the rates of sniffing in the air while the agent is in a casting/surging sequence (sub-plot **(c)**) for the 20 run's solutions. An agent is considered in a casting sequence if it moves crosswind for 3 steps or more in a row and surging if it goes upwind for at least 3 steps.

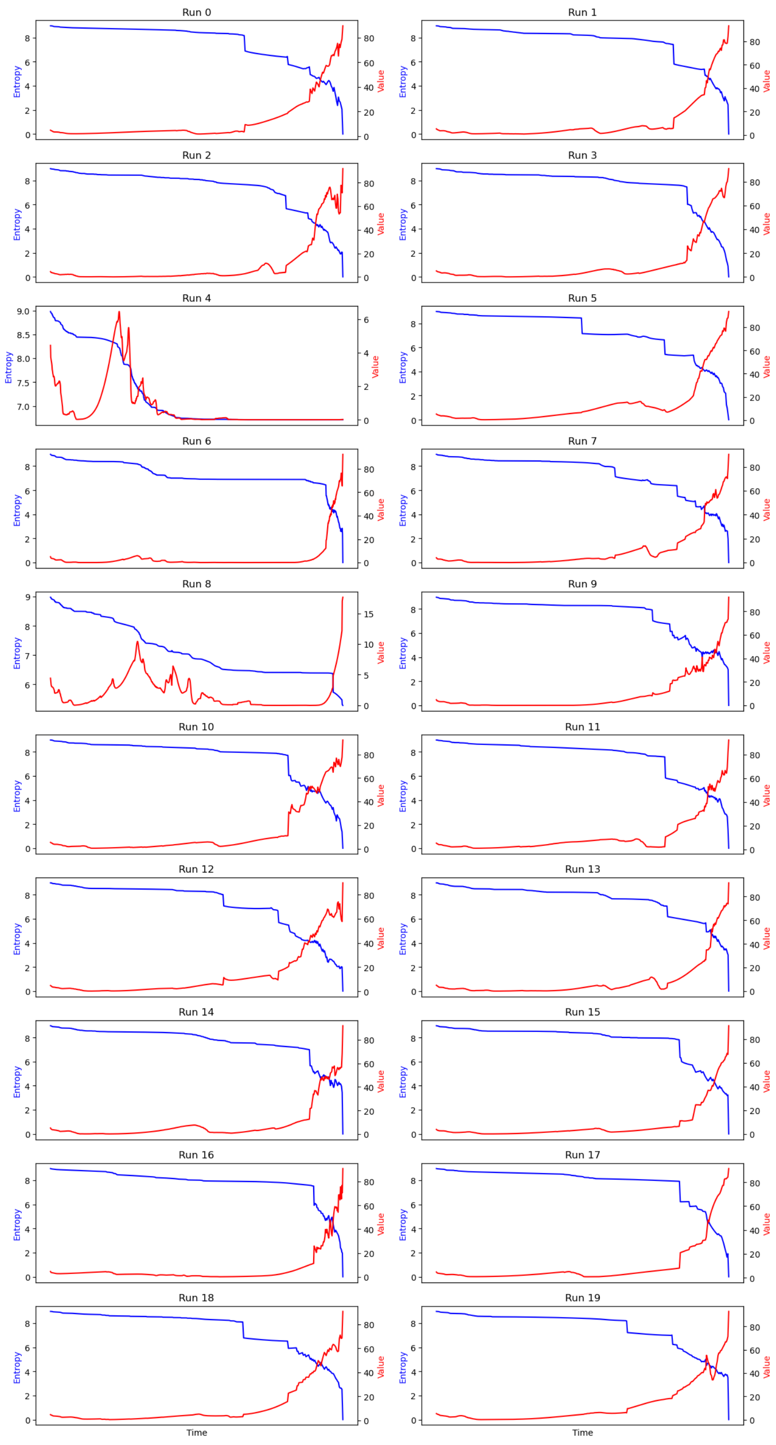


Figure A.4: Belief value and entropy over the course of a simulation for a random simulation (sub-plot **(d)**) for the 20 run's solutions.

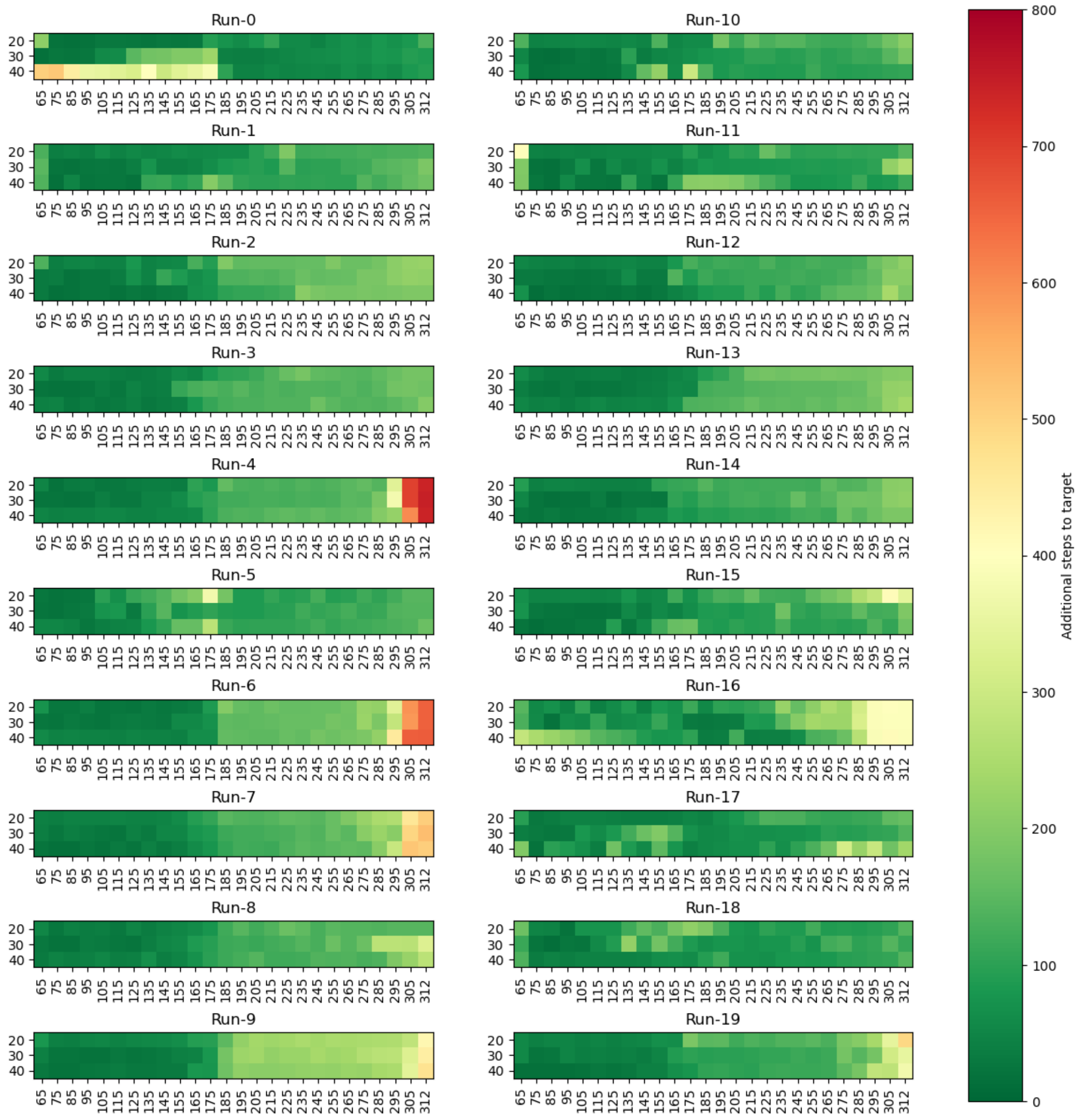


Figure A.5: Grid test performance of 20 runs of FSVI with at most 300 iterations, at most 100 beliefs generated per iteration, and $\gamma = 0.99$. It is applied to the ground and air model.