Toronto Metropolitan University

Data Science and Analytics Program

DS8010 Interactive Learning in Decision Process

Course project

# Empirical Analysis with POMDPs.jl Library: A Comparative Study

Create by

Alice Yang

501149113

April 2023

# Contents

# 1 Introduction

Markov Decision Processes (MDP) is a well-known method for modelling decision-making processes in a perfect environment under certainty where the outcome depends on the current state of the system and the action taken by the decision maker. All states are given to the agent in the MDP problem. Unlike MDP, Partially Observable Markov Decision Processes (POMDPs) is a mathematical framework for modelling an agent decision process under an uncertain environment, where the underlying states are not directly observed. POMDPs problems simulate a real-world environment where directly observing the current state is not possible. For instance, the demand distribution of merchandise is unknown to the seller of a store in real life [4]. POMDPs is a popular tool for modelling decision-making problems in a wide range of applications such as aircraft [19], finance [3], and health care [17].

Solving POMDPs problems can be computationally challenging and time-consuming due to the number of state and action spaces involved in the problem set. We choose Julia programming language for our study. Julia is a high-level, general-purpose dynamic programming language which is designed for parallelism. Because of this design, Julia is renowned for its fast execution time, comparable to languages like C or C++ while offering a high-level approach comparable to Python [11].

Several algorithms have been developed for solving POMDPs, each with its own strengths and weaknesses. In this study, we aim to perform an empirical analysis of multiple algorithms using the package POMDPs.jl [10] written in Julia and examine the performance of each algorithm. POMDPs.jl provides functionality for defining models, simulating problems, and solving them with built-in algorithms.

The objectives of this study are to implement several algorithms using the POMDPs.jl package, compare the performance of these algorithms on several POMDPs problems, and generate a performance table comparing the algorithms on different metrics, such as computation time and collected rewards. The results will be analyzed to identify each algorithm's strengths and weaknesses and determine which algorithm performs best on which type of problem. We also expect to identify areas for future research and improvement.

The contributions of this study can be summarized as follows:

- Seven POMDPs instances are involved in this study. The complexity of each instance differs from each other and provides diverse test sets for our empirical analysis.

- Six algorithms are introduced to solve the POMDPs problems. We provide a comparative analysis of their performance on benchmark problems described above.
- A simulation mechanism is designed to examine the performance of each algorithm using build-in functions in POMDPs.jl.

# 2 Literature Review

This section provides an overview of POMDPs and reviews of recent studies on this topic.

In a standard POMDPs problem, a system can be represented using six components: state space, observation space, action space, transition function, observation function, and reward function. A discrete-time POMDPs problem can formally be described as:

$$P = (S, O, A, T, \Omega, R)$$

where

- $S = \{s_1, s_2, ..., s_n\}$ is a set of partially observable states
- $O = \{o_1, o_2, ..., o_n\}$ is a set of observations
- $A = \{a_1, a_2, ..., a_m\}$ is a set of actions
- $T$ is a set of conditional transition probability from $s \rightarrow s'$, conditioned on the action taken
- $\Omega$ is the observation function
- $R : S \times A$ is a reward function

The first three components are the sets of all possible states the system can be in, all possible observations can be made, and actions can be taken by the agent. The transition function is a probability distribution that describes how the system evolves from one state to another in response to an action. Similarly, the observation function is a probability distribution that describes the likelihood of observing a certain observation given the current state of the system. Lastly, the reward function maps states, actions, and observations to a scalar reward value that indicates how desirable a certain state-action-observation combination is.

## 2.1 Recent works

Although the agent does not have full access to the underlying states in POMDPs setting, we may calculate the most likely states instead and take appropriate action accordingly. A short-term reward value will be received immediately after making an action. The ultimate goal of the POMDPs problem is to learn a policy that maximizes the cumulative long-term reward over time. To achieve this goal, the agent must balance the trade-off between immediate rewards and long-term gains. In some cases, maximizing the long-term cumulative reward may require the agent to disregard the short-term effects. Some of the most popular algorithms used to solve POMDPs problems are summarized in Table 1. We categorize these algorithms as exact solution methods and approximate solution methods.

Table 1: Summary of the relevant literature on POMDPs

| Research | Exact | Approximation | Algorithms | Problem Instance |
|---|---|---|---|---|
| **Litterman et al. (1995) [9]** | No | Yes | QMDP | Mini hallway, tiger, grid world, painting |
| **Hauskrecht (2000) [6]** | Yes | No | FIB | - |
| **Pineau et al. (2003) [12]** | No | Yes | PBVI | - |
| **Kurniawati et al. (2008)[7]** | No | Yes | SARSOP | Rock sample |
| **Silver and Veness (2010) [14]** | No | Yes | POMCP | Rock sample |
| **Sunberg and Kochenderfer (2018) [16]** | No | Yes | POMCPOW | - |

Oftentimes, the exact solution method performs well when there is a small set of states and observations. Finding an exact solution to a POMDPs problem is generally intractable, especially for large and complex problems. The reason is that most exact algorithms use a form of dynamic programming approach, wherein a value function represented by a piecewise linear and convex function is transferred into another. The size of the belief state space grows exponentially with the number of state variables and the horizon of the problem, making it difficult to compute exact solutions. The exact solution method is used in algorithms such as the value iteration algorithm [13], policy iteration algorithm [15], witness algorithm [8], and the IP algorithm [1] that we will examine in our study.

In contrast, the approximate technique uses randomized simulations to approximate the value function and policy. The approximation methods can handle larger POMDPs problems and scale better to high-dimensional state and observation spaces, but they may not guarantee convergence to the optimal solution. Popular algorithms that used approximation methods are listed in Table 1. The choice of approximation technique depends on the problem at hand and the available computational resources.

## 2.2 Solve POMDPs problems in other languages

POMDPs have become an increasingly popular research area in recent years, leading to the development of various solvers using different programming languages other than Julia the one we choose. First, the pomdp_py package [22], written in Python and Cython, is an effective frame for solving POMDPs problems. In addition, the R package pomdp [5] offers algorithms such as enumeration method for solving POMDPs problems. Moreover, the C-written Pomdp-solve [2] package provides a broad range of algorithms, including enumeration, two-pass, witness, and PBVI, to effectively solve POMDPs problems. These different language packages allow researchers and practitioners to apply a variety of tools to solve complex problems. In our study, we will focus on POMDPs.jl as Julia language because Julia enables fast execution times that are comparable to languages such as C or C++ and provides a more intuitive and user-friendly experience like python.

# 3 Methodology

The purpose of this study is to compare the performance of POMDPs.jl algorithms on different POMDPs models. This research aims to answer the research question: which algorithm performs best in terms of computation time and collected rewards. The study will implement seven built-in algorithms: QMDP, FIB, PBVI, SARSOP, POMCP, and POMCPOW. The algorithms were chosen for their popularity and usefulness. The implementation will be in Julia programming language.

Our evaluation of each algorithm will depend on two metrics: computation time and collected rewards. Average computation time measures the time required by an algorithm to compute the optimal policy, while average collected rewards measure the expected reward that an algorithm achieves during the simulation of each problem. We will also report the standard deviation to provide a measure of variability in the results.

## 3.1 POMDPs.jl architecture

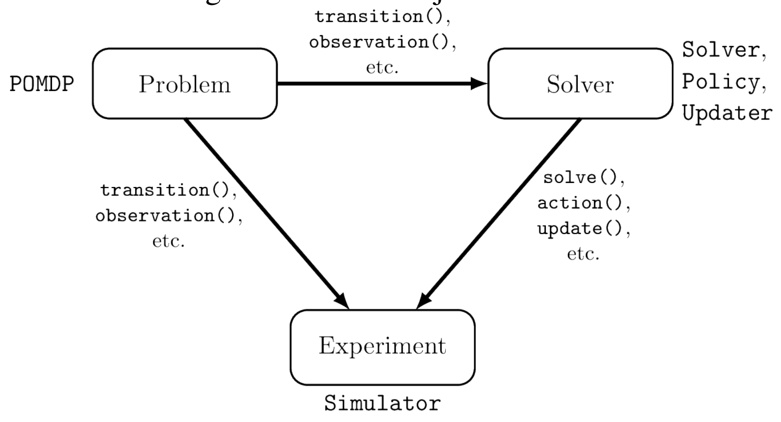
POMDPs.jl developed three main components outlined in Figure 1: the problem, solver, and experiment [23]. The framework of this package defined a set of abstract types and functions that each component must adhere to, for example, transition(). These standardized behaviours allow for seamless integration between the different components of the framework, making it easy to develop, test, and analyze POMDPs models.

We can choose to either use the built-in problem or define our own problem and simulation steps using POMDPs.jl components. Firstly, the problem component is responsible for defining the POMDPs model, including all six components of a POMDPs problem: state space, observation space, action space, transition function, observation function, and reward function. Second, the solver component is responsible for implementing the algorithms that solve the POMDPs model. Lastly, the experiment component is responsible for running simulations of the POMDPs model and analyzing the results.

## 3.2 QMDP

The QMDP algorithm [9] works by first computing an optimal policy for a fully observable MDP that is equivalent to the POMDP. In this way, it is approximating the POMDPs prob-

Figure 1: POMDPs.jl architecture



lem as an MDP problem and derives the upper bound of the Q-function. Then, the QMDP algorithm uses the optimal policy to select the best action to take based on the current belief state. This algorithm is an efficient method as it avoids the need to compute and store a belief state representation of the problem.

The computation time for QMDP is $O(|S^2| \times |A|)$

**Algorithm 1** QMDP Algorithm

```
Initialize the value function $V_0(s)$ for all $s \in S$.
**for** $t = 1$ to $h$ **do**
    Compute the Q-function $Q_t(s,a)$ for all $s \in S$ and $a \in A$:
        $$Q_t(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s'|s,a)V_{t-1}(s')$$
    Compute the value function $V_t(s)$ for all $s \in S$:
        $$V_t(s) = \max_{a \in A} Q_t(s,a)$$
**return** The optimal policy $\pi^*(s) = \arg\max_{a \in A} Q_h(s,a)$ for all $s \in S$.
```

## 3.3 Fast Informed Bound (FIB)

The FIB algorithm [6] was introduced in 2000 and involves iterating between two steps: find all values through the value iteration method [13] and derive the upper bound by using the previous. This is similar to the QMDP method but instead of using the current state to find optimal action, the optimal action depends on the previous state. The update upper-bounds for FIB algorithm are tiger than those for QMDP. The new update is:

$$Q_t(s,a) = R(s,a) + \gamma \sum_{s' \in S} \sum_{o \in \Omega} T(s', o|s,a) V_{t-1}(s')$$

As an exact solution method, the FIB algorithm is computationally expensive compared to the approximation method and is closely related to the standard value iteration [13] and policy iteration algorithms[15]. The computation time for FIB is $O(|S^2| \times |A^2| \times |\Omega|$. Other POMDPs algorithms, such as PBVI described below use different heuristics to improve the efficiency.

## 3.4 Point-Based Value Iteration (PBVI)

PBVI is a point-based value iteration algorithm [12] that approximates the value function of POMDPs by maintaining a finite set of representative belief points. At each iteration, the algorithm selects a subset of the belief points and computes the value function for those points exactly, while approximating the value for the remaining belief points using the previously computed values. The algorithm terminates when the difference between consecutive iterations falls below a certain small threshold.

PBVI is efficient and scalable, making it suitable for solving large-scale POMDPs problems. The algorithm has been extended to handle different types of POMDPs, such as continuous state and action spaces, non-linear dynamics, and non-linear observation models. Full PBVI algorithm can be found in Algorithm 2

**Algorithm 2** PBVI

```
function PBVI
    Initialize belief set B with $b_0$
    while V has not converged to V* do
        IMPROVE(V, B)
        B ← EXPAND(B)
function IMPROVE(V,B)
    repeat
        for each b ∈ B do
            BACKUP(b, V)
            V ← $V \cup \{\alpha\}$
    until V has converged

function EXPAND(B)
    $B_{new}$ ← B
    for each b ∈ B do
        SUCCESSORS(b) ← $\{b^{a,o} | P(o|b,a) > 0\}$
        $B_{new} \leftarrow B_{new} \cup$ argmax $_{b' \in Successors(b)} ||B, b_0||_L$
    return $B_{new}$
```

## 3.5 Successive Approximations of the Reachable Space under Optimal Policies (SARSOP)

SARSOP works by approximating the POMDPs problem using a procedure similar to other point-based algorithms [7], which samples a set of points from the belief states, reduces the size of the belief space and then optimizes the policy. It generates an approximate representation of the space through symbolic approximation which involves clustering the state and observation space of the POMDPs problem to a smaller size. Then, the sampled points and alpha-vectors generated through the approximation are pruned to further improve efficiency.

It is important to note that SARSOP is not guaranteed to find the exact optimal solution, and its performance can depend on the specific problem instance and the choice of parameters.

## 3.6 Partially Observable Monte Carlo Planning (POMCP)

POMCP is a Monte Carlo Tree Search algorithm that solves POMDPs problems by constructing a tree of possible actions and observations using random simulations [14]. It estimates the value of each action given the current state of the environment and the agent's current belief. Details about POMCP Algorithm is described in Algorithm 3

## 3.7 POMCP with observation widening (POMCPOW)

Unlike the POMCP algorithm described above that combines random simulation with tree search, POMCP with observation widening [16] combines Monte Carlo simulations and tree search to approximate the belief state and find an optimal policy. POMCPOW algorithm improves the efficiency of Monte Carlo simulations by using the online method. This generates simulation trajectories that are biased towards regions of the state space that are likely to lead to high-quality solutions.

POMCPOW algorithm also uses a modified form of Upper Confidence Bounds (UCB) to balance exploration and exploitation during the tree search. This takes into account the history of the search, including the actions and observations that have been made, to guide the exploration more effectively.

The POMCPOW algorithm utilizes a belief representation that is gradually expanded and weighted to update its beliefs. This is done by assigning a higher number of particles to beliefs that have a greater likelihood of being reached by the optimal policy. The algorithm simulates a state and inserts it into the weighted particle collection that represents the belief, followed by sampling a new state. Efficient re-sampling is achieved using binary search, resulting in a computational complexity of $O(|S| \times d \times log(|S|))$. Full POMCPOW algorithm can be found in Algorithm 4

**Algorithm 3** POMCP Algorithm

```
procedure SEARCH(h)
    repeat
        if h = empty then $s \sim I$
        else $s \sim B(h)$
        SIMULATE(s,h,d=0)
    until TIMEOUT
    return argmax V(hb)
procedure ROLLOUT(s,h,d)
    if $\gamma^d < \epsilon$ then
     return 0
    $a \sim \pi_{rollout}(h, \dot{})$
    $(s', o, r) \sim G(s, a)$
    return r+ $\gamma$ ROLLOUT($s', hao, d+1$)

procedure SIMULATE(s,h,d)
    if $\gamma^d < \epsilon$ then return 0
    if h $\notin T$ then
        for all $a \in A$ do
            $T(ha) \leftarrow (N_{init}(ha), V_{init}(ha), )$
      return ROLLOUT(s,h,d)
    $a \leftarrow argmaxV(hb) + c\sqrt{\frac{logN(h)}{N(hb)}}$
    $(s', o, r) \sim G(s, a)$
    R $\leftarrow r + \gamma$SIMULATE$(s', hao, d+1)$
    $B(h) \leftarrow B(h) \cup \{s\}$
    $N(h) \leftarrow N(h) + 1$
    $N(ha) \leftarrow N(ha) + 1$
    $V(ha) \leftarrow V(ha) + \frac{R - V(ha)}{N(ha)}$
    return R
```

**Algorithm 4** POMCPOW Algorithm

```
procedure SEARCH(h)
    repeat
        if h = empty then
            $s \sim I$
        else
            $s \sim B(h)$
        SIMULATE($s, h, d = 0$)
    until TIMEOUT
    return argmax V(hb)
procedure ROLLOUT($s, h, d$)
    if $\gamma^d < \epsilon$ then
        return 0
    $a \sim \pi_{rollout}(h, \dot{)}$
    return r+ $\gamma$ ROLLOUT($s', hao, d + 1$)
procedure SIMULATE($s, h, d$)
    if $d = 0$ then
        return 0
    $a \leftarrow$ ACTIONPROGWIDEN($h$)
    $s', o, r \leftarrow G(s, a)$
    if $|C(ha)| \leq k_o N(ha)^{\alpha_o}$ then
        $M(hao) \leftarrow M(hao) + 1$
        if $o \notin C(ha)$ then
            $C(ha) \leftarrow C(ha) \cup \{hao\}$
            return $\leftarrow r + \gamma$ROLLOUT($s', hao, d - 1$)
    else
        $o \leftarrow$ select $o \in C(ha)$ w.p. $\frac{M(hao)}{\sum_o M(hao)}$
    append $s'$ to $B(hao)$
    append $Z(o|s, a, s')$ to $W(hao)$
    $s' \leftarrow$ select $B(hao)[i]$ w.p. $\frac{W(hao)[i]}{\sum_{j=1}^{m} W(hao)[j]}$
    $r \leftarrow R(s, a, s')$
    $total \leftarrow r + \gamma$SIMULATE($s', hao, d - 1$)
    $N(h) \leftarrow N(h) + 1$
    $N(ha) \leftarrow N(ha) + 1$
    $Q(ha) \leftarrow Q(ha) + \frac{total - Q(ha)}{N(ha)}$
    return $total$
```

# 4 Experimental Setup

To assess the effectiveness of each algorithm, we will utilize a set of seven POMDP models. These models encompass the crying baby problem, tiger problem, paint problem, query problem, mini hallway problem, rock sample problem, simple grid world problem, and T-maze problem. The selection of these models was based on their differing levels of intricacy.

The decision horizon in a POMDPs problem refers to the number of time steps for which a decision must be made. The decision horizon can be either finite or infinite, depending on the problem itself. In our study, we set the limit for each simulation in a POMDPs problem to a finite horizon of 100-time steps. Therefore, the agent must be able to efficiently gather information about the environment and update its beliefs in a timely manner to make optimal decisions.

We will conduct our experiments on a personal computer with an Intel Core i5 processor and 8GB of RAM running the Windows operating system. We will use the latest version of the Julia programming language version 1.8.1 and version 0.9.5 of the POMDPs.jl package.

## 4.1 Problems

In general, POMDPs problems can be considered complex due to the following factors: large state space, non-stationary environments, and etc. The experiments conducted in this study are centred around expanding the seven well-known POMDPs problem instances listed in Table 2, with increasing state space. We are taking the simplified version of each problem to reduce destructive run-time issues and repeat the experiment for 10 rounds and take an average to minimize the effect of randomness on the performance indicator, namely average cumulative discounted reward and average run time for a round. For each round, we will simulate 1000 games from initial state to a terminal state of a problem.

Further information on the problems examined in this study and their POMDPs extensions is provided in the section below.

### 4.1.1 Crying Baby Problem

The problem involves an agent tasked with calming a crying baby, but the agent is only able to observe noisy and ambiguous cues about the baby's needs, such as the baby's cries.

Table 2: POMDPs problem instances

| Problem | States | Actions | Observations | Reference |
|---|---|---|---|---|
| **Crying baby** | 2 | 2 | 2 | Wu et al. (2018) [20] |
| **Tiger** | 2 | 3 | 2 | Cassandra et al. (1997) [1] |
| **Query** | 9 | 2 | 2 | Xuan et. al. [21] |
| **Paint** | 4 | 4 | 2 | Cassandra et al. (1997) [1] |
| **Mini hallway** | 13 | 4 | 9 | Litterman et al. (1995) [8] |
| **T-Maze problem (length =10)** | 13 | 4 | 5 | Wierstra et al. (2007) [18] |
| **Rock Sample(3 $\times$ 3)** | 37 | 4 | 2 | Sunberg and Kochenderfer (2018) [16] |

However, the baby might cry even when he's not hungry. The agent must infer the underlying state of the baby, such as whether they are hungry or not hungry to take effective actions. The goal is to identify the baby's needs and take action, for example, feeding the baby, that will soothe them. To model the crying baby problem in a POMDPs environment, states representing the baby's underlying needs are either "hungry" or "not hungry". Actions like "feed" or "not-feed" the baby represents the agent's interventions. Agent's noisy observations of the baby's behaviour, namely crying or not crying are the two observations involved in this question. The transitions between states depend on the agent's actions and the baby's needs. For instance, if the agent offers a bottle, the baby stop being hungry at the next time step.

The agent needs to determine whether to feed a crying baby to maximize the expected reward, which could be a true signal of hungry or a false alarm that wastes the agent's time. The reward for this problem is always negative whereas the "feed" action will result in a negative reward of -5 and the baby in a "hungry" state will result in a negative reward of -10. The terminal state for this problem is set to the time when the agent takes the action to feed the baby no matter the baby's current state.

The initial state for the crying baby problem is always not hungry. In a normal setting, the baby might have a 10% chance of getting hungry and a great probability of 0.8 for crying when hungry. The chance of crying when not hungry is low with a probability of 0.1. The discounted rate for this problem is 0.9 by default.

### 4.1.2 Tiger Problem

The problem scenario involves an agent positioned in front of two doors. Behind one door, there is a tiger, and opening it results in a significant negative reward. In contrast, opening the other door leads to a large positive reward. The problem is represented using two states, tiger-left and tiger-right. The agent has three available actions: "listen", "open-left", and "open-right".

If the agent chooses the "listen" action, it incurs a reward of -1 and receives an observation of either "tiger-left" or "tiger-right", which can help decrease the uncertainty in the belief state. The action of opening a door leads to rewards of -100 and +10, respectively, for the doors with and without the tiger behind them. Notably, the "listen" action is helpful in reducing the belief state's uncertainty by providing 85% accurate information about whether there is a tiger behind either door. In practical terms, it is advisable to keep listening until the belief state is heavily biased towards one door, indicating the tiger's location. The discount factor for this problem is 0.95 by default, and we restart the game each time when we open a door.

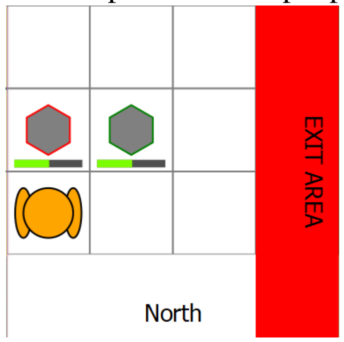
### 4.1.3 Paint Problem

The paint problem is a task that involves painting parts within an automated factory system. The goal is to ensure that only flawless and blemish-free products are shipped, while any parts with imperfections are rejected. The environment consists of four states that provide information on the product's condition: "NFL-NBL-NPA", "NFL-NBL-PA", "FL-NBL-PA", and "FL-BL-NPA", where NFL/FL represents non-flawed/flawed, BL/NBL represents blemished/unblemished, and PA/NPA represents painted/unpainted.

To achieve this goal, the automated system can choose to perform one of four actions: "paint", "inspect", "ship", or "reject" a part. If the system ships a painted, unblemished, non-flawed part (state "NFL-NBL-PA"), it is rewarded with +1. However, if the system ships a part in any other state, it incurs a reward of -1. Additionally, the system is rewarded with +1 for rejecting a part that is flawed, blemished, and unpainted. The "inspect" action have a probability of 85% for detecting the correct items to ship or reject. The initial state of the problem is always unpainted, namely a random selection between state "NFL-NBL-NPA" and "FL-BL-NPA". Discounted rate for this problem is 0.95. Terminal state is when the agent decides to ship or reject the item.

### 4.1.4 Rock Sample Problem

Figure 2: Set up of rock sample problem



In this problem, a robot is tasked with collecting rock samples from a grid world. We consider a simplified version of a 3 by 3 grid containing 2 rocks in our study. The robot is rewarded when it samples a good rock but is penalized on the contrary. To detect the quality of rocks, the robot is equipped with a sensor. The robot can move in 4 directions, it can sample the rock that it is standing on, and it can check each of the rocks using its sensor, for a total of 5 actions. The robot can also occupy any of the squares on the grid, of which there are 9. The total number of states of the 2 rocks is 4. There is a terminal state, which is reached by moving east off of the right side of the grid, hence the total number of states is $9 \times 4 + 1 = 37$.

Whenever the robot takes a step, it incurs a minor penalty of -1, except when it enters the terminal state. If the robot enters the terminal state, it receives a reward of +20 and is reset to the initial state and belief in the next epoch. The goal of the robot is to collect as many good rocks as possible while taking a minimal number of steps. If the robot samples a good rock, it is awarded +10; instead, it is penalized by -10. If the robot moves against the wall, it will remain in its current position and have no reward. The discounted reward is 0.95.

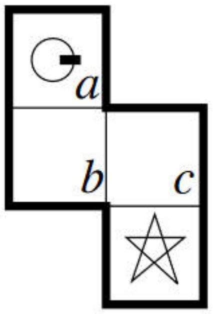
### 4.1.5 Query problem

The server problem involves two servers with different speeds and failure probability. Server A and Server B can be either "loaded", "unloaded", or "down" at any given time. Both servers have the same success rate, meaning that when a request is sent to the other, it will respond successfully with a probability of 0.85. If both servers are loaded, it will have a great chance to have a fast response. When one of the servers is unloaded, there is a moderate probability of getting a fast response. When bother servers are unloaded, a slow response for the server is of high probability.

The goal is to determine the status of both servers at each time step, given a sequence of requests and responses. However, because the servers' responses are probabilistic and can be affected by various factors, the status of the servers is only partially observable. The agent is trying to minimize the time takes to process the queries. It gives no reward for no response server, +3 reward for a slow response, and +10 for a fast response situation. At the initial state, both servers have an equal chance in any state and the discounted factor for this problem is 0.95 by default. We will terminate the trial when there is no response from the server.

### 4.1.6 Mini hallway problem

Figure 3: Mini Hallway Environment



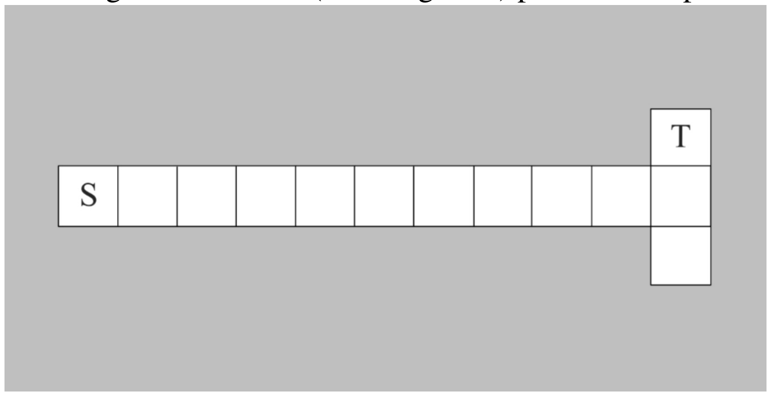
The Mini Hallway problem illustrated in Figure 3 is a simple navigation task that involves an agent moving through a narrow hallway to reach a target location with minimal steps. The agent's movements are restricted by the narrow hallway, which makes navigation challenging. The transition model of the Mini Hallway problem is deterministic, meaning that the agent's movement is always accurate, and the transition function is known. When the agent takes an action, it moves in the direction specified by the action, unless the movement would take it outside the grid or through a wall. If the movement is successful, the agent's new location is determined by the transition function, which maps the agent's current location and action to a new location.

The Mini Hallway problem has a finite number of states, actions, and observations. The states in this problem represent the agent's location in the hallway and the target location. The hallway is modelled as shown in diagram **??**, and the target location is fixed at the bottom-right corner of the grid. The agent's starting location is from the top-left corner of the grid. The states are partially observable, and the agent's sensor can take four possible readings: up, down, left, or right. These observations give the agent a sense of the direction it is moving in. The reward function of the Mini Hallway problem is designed to encourage the agent to reach the target location with as few steps as possible. The agent receives a zero reward for each movement it makes, and a reward of +1 when it reaches the target location. The goal of the agent is to maximize its cumulative reward while minimizing the number of steps it takes to reach the target location.

### 4.1.7 T-Maze problem

T-maze problem: the agent must navigate a T-shaped maze and choose the path that leads to a reward while avoiding the path that leads to a penalty.

Figure 4: T-Maze (arm length 10) problem setup



In the T-Maze problem, the agent navigates a T-shaped maze consisting of a stem and two arms, illustrated in Figure 4. The goal is to reach the end of one of the arms, which leads to a reward while avoiding the other arm, which leads to a penalty. The agent can move in all 4 directions, north, east south and west. The arms of the maze have a fixed length of 10 units. The agent receives a reward of +4 if it reaches the end of the left arm. The reward of -0.1 is received when the agent bumps against the wall or reaches the wrong arm. Discounted rate for this problem is 0.99 by default.

## 4.2 Solver Options: Keyword Arguments

Each solver in POMDPs.jl use different algorithms to determine an optimal policy for an agent operating in an uncertain environment. Each solver provides a set of keyword arguments that allow the user to customize the behaviour of the solver and fine-tune its performance. Understanding the available keyword argument choices is important for achieving optimal performance with these solvers. We describe the available keyword arguments for each solver and discuss their impact on solver performance below:

- `max_iterations = 20`

  Default maximum number of iterations to run the solver is 100 for QMDP, FIB, PBVI, and SARSOP algorithms. We believe having more iterations will not improve the result and thus choose 20 iterations to be the sufficient number of iterations for discrete state space problems.

- `tree_queries=100` POMCP and POMCPOW algorithm is different from the other algorithms where they implement a Monte Carlo tree search. The number of iterations during each action call defaults to be 1000 but choose 100 tree queries for simplicity. Increasing the number of tree_queries will not improve the performance.

- `max_depth =10`: Rollouts and tree expansion will stop when this depth is reached for POMCP and POMCPOW algorithm

# 5 Results

To evaluate the performance of the algorithms, we conducted experiments using seven POMDPs problems of varying levels of complexity. For each problem, we ran each algorithm ten times and recorded the average cumulative reward and cumulative discounted reward obtained over 1000 trials. We also measured the average computation time required to solve each problem.

We set up the experiment so that each trial starts from the initial state and ends at the terminal state. The number of steps taken within each trial is different for different trials. In this experiment, we want to examine the quality of the experiments first through an un-discounted reward and compare it with discounted reward.

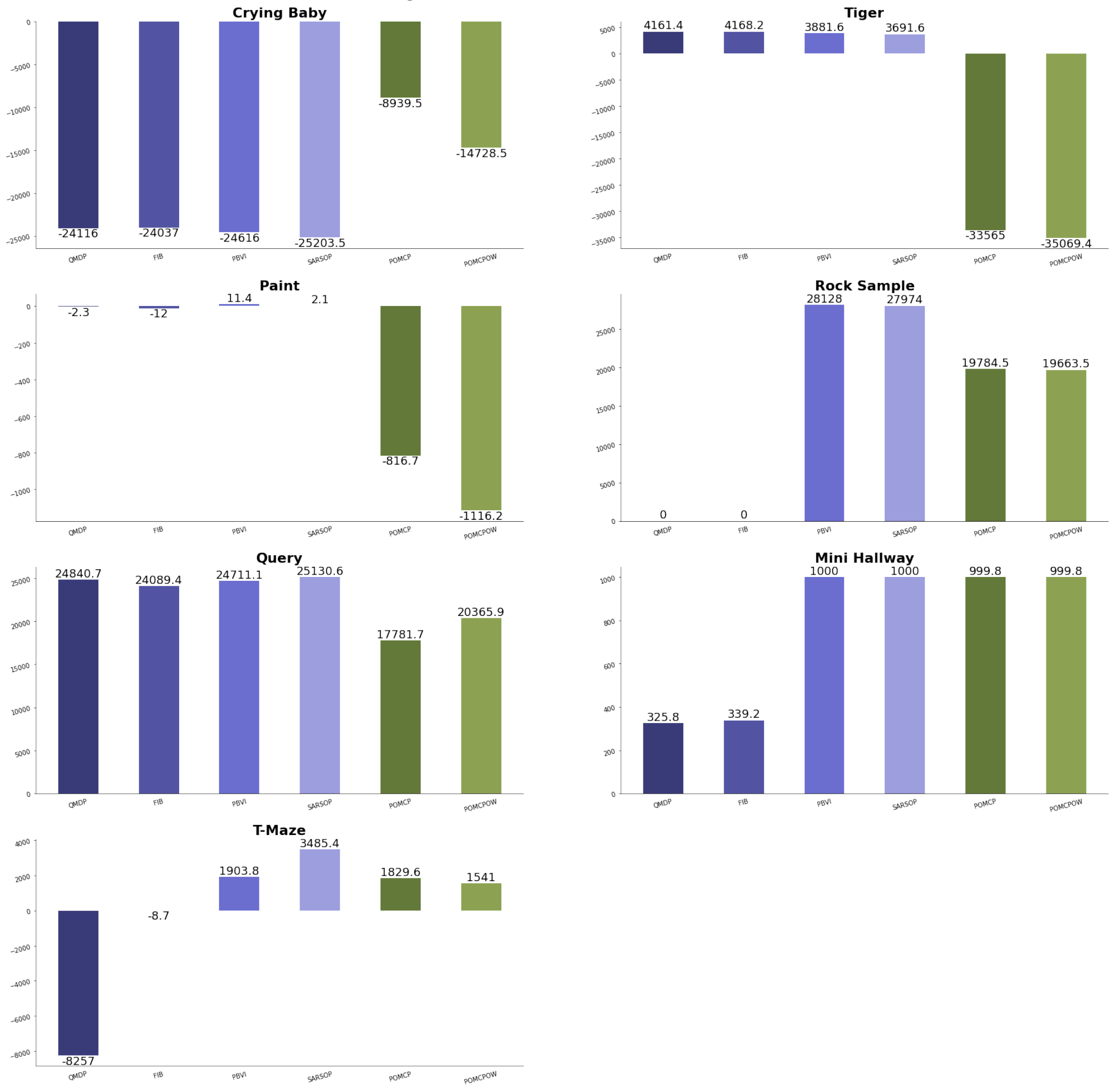
## 5.1 Validation

Figure 5 is a grouped bar plot for cumulative un-discounted reward. In this case, we have six grouped bar plots, with each plot representing the rewards obtained from seven problems using one of six different algorithms. The y-axis of each bar plot represents the rewards obtained from the problem, and the x-axis shows the algorithm used.

Looking at the first problem, we observe that the crying baby problem has a minimal reward of -8939.5 and a maximum reward of 23203.5. There is a high variance between the result for each algorithm. In most cases, the agent might first observe the baby's crying and take action to feed the baby in the next round. This will give us a cumulative reward of $(-10 - 10 - 5) \times 1000 = 25000$ for 1000 trials. On the other hand, the agent might feed the baby when he is not hungry. This will give us optimal cumulative reward but does not indicate a good performance as it is not acting based on the environment but acting based on randomness. The reward for such a case will give us the cumulative reward of $-5 \times 1000 = -5000$. From the result, we can conclude that POMCP and POMCPOW are not performing reasonably.

The perfect case for the tiger problem cumulative reward happens when the agent always opens the correct door, which is $10 \times 1000 = 10000$; however, this might be hard to achieve in real life. Therefore, we might want to consider the case when the agent opens the correct door with an accuracy of 95%. The cumulative reward will then be $10.95 - 100 \times 1000 \times 0.05 =$

Figure 5: Cumulative Reward



$4500$. When the cumulative reward reaches 4500, then the agent is having a high accuracy of $95\%$. The worst case for the tiger problem is when the agent has a 50% chance of opening the correct door. The reward for such a "wild guess" scenario is $10\times10)0\times0.5-100\times100\times0.5 = -45000$. QMDP, FIB, PBVI, and SARSOP are performing with an accuracy close to $95\%$. POMCP and POMCPOW algorithms are performing poorly.

Similar to the previous problem, the paint bar plot is giving a similar trend where POMCP and POMCPOW give low rewards. The agent is expected to take at least one action before rejecting or shipping the corresponding item. The expected cumulative reward for this problem is $(-1+1)\times1000 = 0$. This means algorithms such as QMDP, FIB, PBVI, and SARSOP which have rewards around 0 are acceptable. POMCP and POMCPOW algorithm takes an extra step of "paint" or "inspect" before making a decision to ship or reject. The reward for this is approximately $(-1 + 1 - 1) \times 1000 = -1000$

For the Rock Sample problem, QMDP and FIB are not performing well as it keeps moving forward and backward and is not able to find the exit or find a rock to gain a reward. Since there is no penalty for moving within grids with no rock or not exiting grids. In the best case, the robot is able to find 2 rocks and then exit the grid to maximize cumulative reward. This gives a reward of $(10 + 10 + 20) \times 1000 = 40000$. Unfortunately, none of the algorithms can collect 2 rocks. In most cases, the robot can detect 1 rock and then exit the grid which gives cumulative rewards of $(10 + 20) \times 1000 = 30000$. In our experiment, PBVI and SARSOP are able to give rewards of 28128 and 27974 close to this expectation. POMCP and POMCPOW algorithms approximately give average rewards of around 20000, indicating that they can either find both rocks but not exit the grid or are unable to find the rocks but exit the grid.

All algorithms are giving relatively similar rewards for query problems, while some algorithms can give higher accuracy in choosing which server to query to get a faster response. In the worst case, the server will get at least one fast response and one slow response before getting no response. The reward for the worst case is $(10 + 3) \times 1000 = 13000$. All the algorithms are able to meet this minimal requirement.

The mini hallway problem is set up so that it receives a reward of 1 when reaches the goal and zero rewards for moving within the grids. We are finding the solution to this problem in a finite horizon of 100. If the agent is not able to find the solution within 100 steps, it will then receive a reward of zero and exit the trial. Because of this design, the optimal solution is that the algorithm can find the goal within 100 steps and gives a reward of $1 \times 1000 = 1000$. PBVI, SARSOP, POMCP, and POMCPOW have average reward close to 1000 which meet this expectation. In contrast, QMDP and FIB algorithms are not performing well as they have rewards of around 320 indicating that they have a low chance of 32% for finding a goal.

Lastly, the cumulative reward for T-Maze problem has high variance. The initial state of the problem can be at any grid that is not the terminal state. Assuming in the perfect world, the agent is one step away from the goal, near the end of the T-maze, the expected reward for this scenario is $(-0.1 + 4) \times 1000 = 3900$. Suppose the agent got unlucky and is far away from the goal, it takes 10 steps to reach the goal which leads to a expected reward of $(-0.1 \times 10 + 4) \times 1000 = 3000$. In our experiment, SARSOP can find the goal with a minimal number of steps and gives a reward of 3485.4. PBVI, POMCP, and POMCPOW receive an average reward of around 1500 to 2000 meaning that they will take extra steps and might move against the wall before reaching the goal. The worst case for this problem is when the agent can't find the goal within 100 steps and gives reward of $(-0.1 \times 100) \times 1000 = 10000$. QMDP is not performing well since a negative reward of -8257 indicates the agent is unable to find the goal in most cases and keeping get the negative reward of -0.1 for 100 steps and exiting the game.

## 5.2 Performance: Reward and Runtime

We have discussed whether the result getting from the experiment are reasonable or not using cumulative un-discounted reward. Now, we will use average discounted reward to examine how much the reinforcement learning algorithm takes care of rewards in the distant future relative to those in the immediate future and decide the most suitable algorithm for problems with different levels of complexity.

The results of our experiments are summarized in 3. Our results indicate that PBVI and SARSOP performed consistently well across all problems, achieving high average discounted rewards and relatively fast computation times. For example, PBVI achieved high discounted rewards on the mini hallway and the T-Maze problem, while SARSOP achieved an average discounted reward of 34.3 on the tiger problem. Both algorithms also had relatively fast computation times, with PBVI requiring an average of 1.0194 seconds to solve

Table 3: Performance Matrix

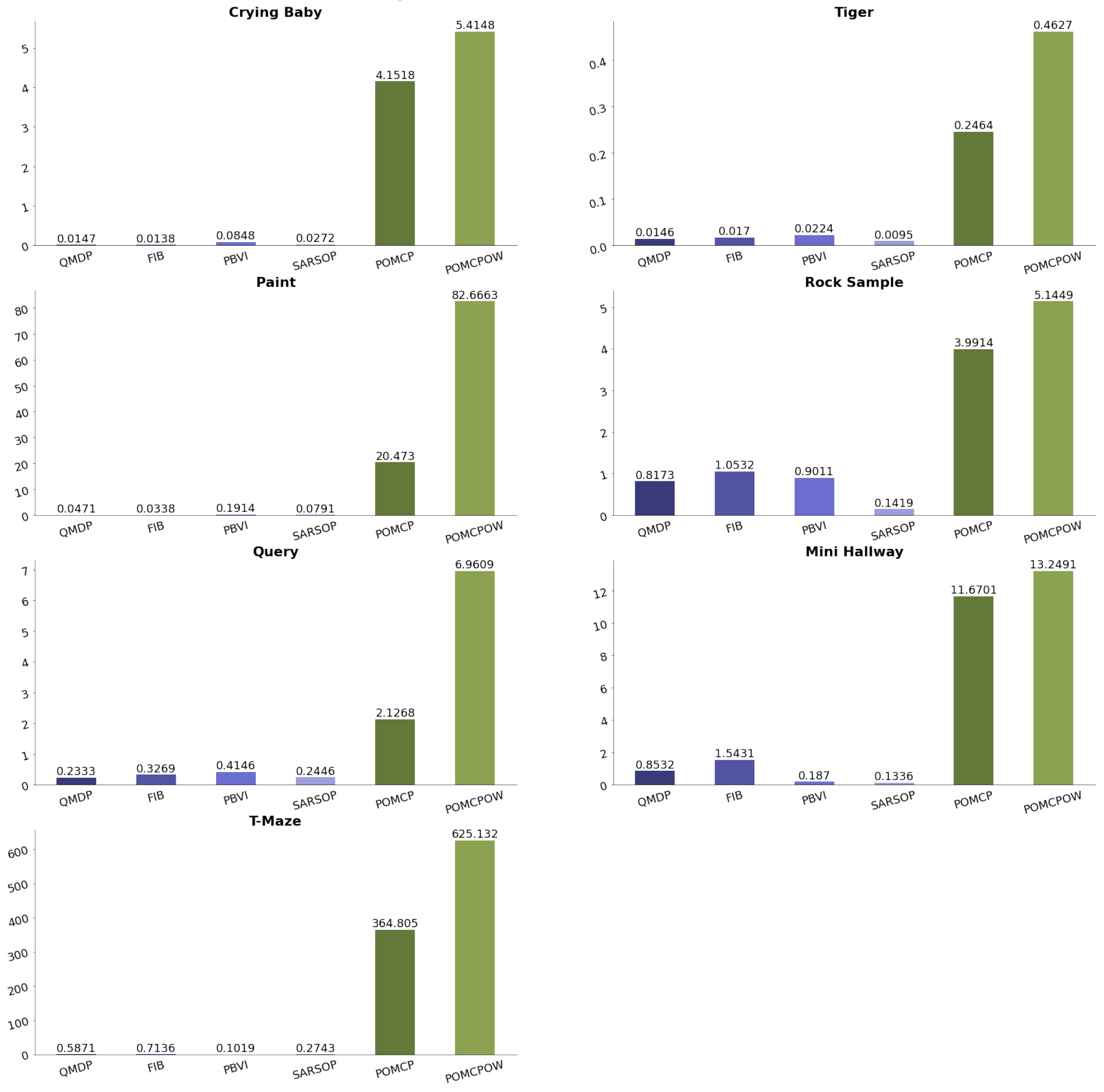
| | | QMDP | FIB | PBVI | SARSOP | POMCP | POMCPOW |
|---|---|---|---|---|---|---|---|
| **Discounted Reward** | **Crying Baby** | **-12.9** | -19.5 | -14.3 | -13.4 | -20.3 | -18.6 |
| | **Tiger** | 25.9 | 24.6 | 31.9 | **34.3** | -372.8 | -328.7 |
| | **Paint** | -0.5 | **0.4** | -1.6 | -1.3 | -5.5 | -5.3 |
| | **Rock Sample** | 0.0 | 0.0 | 67.3 | 66.6 | 68.6 | **68.9** |
| | **Query** | 95.8 | **100.6** | 96.8 | 98.2 | 82.3 | 78.6 |
| | **Mini Hallway** | 0.0 | 0.1 | **2.2** | 2.0 | 0.4 | 0.4 |
| | **T-Maze** | -8.6 | 0.0 | **29.9** | 24.7 | 16.2 | 9.9 |
| **Runtime** | **Crying Baby** | 0.0147 | **0.0138** | 0.0848 | 0.0272 | 4.1518 | 5.4148 |
| | **Tiger** | 0.0146 | 0.017 | 0.0224 | **0.0095** | 0.2464 | 0.4627 |
| | **Paint** | 0.0471 | **0.0338** | 0.1914 | 0.0791 | 20.473 | 82.6663 |
| | **Rock Sample** | 0.8173 | 1.0532 | 0.9011 | **0.1419** | 3.9914 | 5.1449 |
| | **Query** | **0.2333** | 0.3269 | 0.4146 | 0.2446 | 2.1268 | 6.9609 |
| | **Mini Hallway** | 0.8532 | 1.5431 | 0.187 | **0.1336** | 11.6701 | 13.2491 |
| | **T-Maze** | 0.5871 | 0.7136 | **0.1019** | 0.2743 | 364.8055 | 625.1324 |

the T-Maze problem, and SARSOP requiring the least amount of time across the tiger, rock sample, and mini hallway problem.

In contrast, QMDP and FIB algorithms struggled with some of the more complex problems with a larger number of states and actions, achieving low average rewards and relatively slow computation times of less than 15 seconds. For example, FIB and QMDP achieved optimal discounted rewards for problems with a smaller number of states such as crying baby, paint and query problems. However, QMDP achieved zero rewards on the rock sample and mini hallway problem, while FIB achieved zero average rewards on the rock sample, mini hallway, and t-maze problem.

POMCP and POMCPOW, on the other hand, showed great promise in handling problems with large state spaces, achieving good average rewards on rock sample, query, mini hallway, and t-maze problems. However, it struggled with problems requiring precise control and small state spaces, achieving negative average rewards on tiger and paint problems. It also had relatively slow computation times across all problems compared to other algorithms, see Figure 6. For example, POMCPOW required an average of 5 seconds to solve the crying baby problem with 2 states only, while other algorithms only require less than 0.1 seconds.

Figure 6: Cumulative Runtime



Overall, our results suggest that the choice of algorithm can have a significant impact on the effectiveness and efficiency of POMDPs solvers for various problem domains. PBVI and SARSOP are strong performers across all problems, while QMDP and FIB may struggle with more complex problems. POMCP and POMCPOW show promise for problems with large state spaces but may struggle with problems requiring precise control.

In summary, our study provides a valuable comparative analysis of the performance of six POMDPs algorithms implemented in the POMDPs.jl package. Our results highlight the importance of careful algorithm selection when solving POMDPs and provide insights into the strengths and weaknesses of each algorithm for different problem domains. These results can serve as a valuable resource for practitioners working with POMDPs and can inform the development of future POMDPs solvers.

# 6 Conclusion

In conclusion, our study provides insights into the performance of different algorithms for solving POMDPs using built-in solvers in the package POMDPs.jl. Our results suggest that the choice of algorithm can significantly impact the effectiveness and efficiency of POMDPs solvers for various problem domains.

- QMDP and FIB handles simple model with a small number of state spaces well but have difficulty working with problems with larger state spaces
- PBVI and SARSOP have outstanding discounted across all discrete problems with different levels of complexity
- Computation time for QMDP, FIB, PBVI, and SARSOP are relatively similar, usually able to complete 1000 games in less than 2 seconds for all problems
- POMCP and POMCPOW have the advantage of solving complex problems but are unable to process problems with small state space that require more precise control. The computation time for POMCP and POMCPOW is at least 10 times slower than other algorithms

It is worth noticing that our comparative study has some limitations. Firstly, we focused only on the algorithms implemented in the POMDPs.jl package and did not explore other state-of-the-art POMDPs algorithms that may offer improved performance. Secondly, the problems we selected do not represent the full range of possible POMDPs, and it is possible that some algorithms may perform differently on other problem domains. Lastly, we only considered problems with the discrete state, action, and observation spaces and performance on continuous problems can be different from our result.

Despite these limitations, our study provides a valuable starting point for practitioners seeking to solve POMDPs. By comparing the performance of different algorithms across multiple problem domains, our study can help guide the selection of algorithms for specific POMDPs applications. Additionally, our findings highlight the need for continued research and development of POMDPs solvers, particularly for solving large-scale problems with complex dynamics.

Future work could explore the performance of other POMDPs solvers, as well as investigate the use of hybrid algorithms that combine different techniques for improved performance. Moreover, it may be beneficial to extend the analysis to include other types of POMDPs problem, such as problems with state spaces greater than 100 and continuous-state.

In summary, our study provides a comprehensive evaluation of the performance of eight different algorithms implemented in the Built-In POMDPs.jl package, across a range of problem domains. While some algorithms performed better than others, our study underscores the importance of careful algorithm selection in tackling complex POMDPs problems. We hope that our findings will inspire further research in this area, and contribute to the development of more effective and efficient POMDPs solvers.

## References

[1] Anthony Cassandra, Michael L Littman, and Nevin L Zhang. Incremental Pruning: A Simple, Fast, Exact Method for Partially Observable Markov Decision Processes. 1997.

[2] Anthony R. Cassandra. *pomdp-solve: POMDP Solver Software*, 2015.

[3] Xi-Ren Cao De-Xin Wang. Event-Based Optimization for POMDPs and Its Application in Portfolio Management. *ScienceDirect*, 44:3228–3233, January 2011.

[4] Tianhu Deng, Zuo-Jun Max Shen, and J. George Shanthikumar. Statistical Learning of Service-Dependent Demand in a Multiperiod Newsvendor Setting. *Operations Research*, 62(5):1064–1076, October 2014.

[5] Michael Hahsler and Anthony R. Cassandra. *pomdpSolve: Interface to 'pomdp-solve' for Partially Observable Markov Decision Processes*, 2023.

[6] M. Hauskrecht. Value-Function Approximations for Partially Observable Markov Decision Processes. *Journal of Artificial Intelligence Research*, 13, August 2000.

[7] Hanna Kurniawati, David Hsu, and Wee Sun Lee. SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces. June 2008.

[8] Michael Littman. The witness algorithm: Solving partially observable markov decision processes. Feburary 1995.

[9] Michael L. Littman, Anthony R. Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. pages 362–370, 1995.

[10] Edward Balaban Tim A. Wheeler Jayesh K. Gupta Maxim Egorov, Zachary N. Sunberg and Mykel J. Kochenderfer. POMDPs.jl: A Framework for Sequential Decision Making under Uncertainty. *Journal of Machine Learning Research*, 18, April 2017.

[11] Tomáš Omasta. Efficient MDP Algorithms in POMDPs.jl. 2021.

[12] Joelle Pineau, Geoff Gordon, and Sebastian Thrun. Point-based value iteration: An anytime algorithm for POMDPs. 2003.

[13] Katsushige Sawaki and Akira Ichikawa. Optimal control for partially observable markov decision processes over an infinite horizon. *Journal of The Operations Research Society of Japan*, 21:1–16, 1978.

[14] David Silver and Joel Veness. Monte-Carlo Planning in Large POMDPs. 2010.

[15] Edward Sondik. The optimal control of partially observable markov process over the infinite horizon: Discounted costs. *Operations Research*, 26:282–304, 04 1978.

[16] Zachary Sunberg and Mykel J. Kochenderfer. Online algorithms for POMDPs with continuous state, action, and observation spaces. September 2018.

[17] Tarek Taha, Jaime Valls Miro, and Gamini Dissanayake. Wheelchair Driver Assistance and Intention Prediction Using POMDPs. December 2007.

[18] Daan Wierstra, Alexander Förster, Jan Peters, and Jürgen Schmidhuber. Solving Deep Memory POMDPs with Recurrent Policy Gradients. 4668:697–706, September 2007.

[19] Travis B. Wolf and Mykel J. Kochenderfer. Aircraft Collision Avoidance Using Monte Carlo Real-Time Belief Space Search. *Jornal of Intelligent  Robotic Systems*, January 2011.

[20] Yueh-Hua Wu and Shou-De Lin. A Low-Cost Ethics Shaping Approach for Designing Reinforcement Learning Agents. September 2018. arXiv:1712.04172 [cs].

[21] Ping Xuan, Victor Lesser, and Zilberstein. Communication Decisions in Multi-agent Cooperation: Model and Experiments. 2001.

[22] Kaiyu Zheng and Stefanie Tellex. pomdp_py: A framework to build and solve pomdp problems. In *ICAPS 2020 Workshop on Planning and Robotics (PlanRob)*, 2020.

[23] zsunberg. Pomdps.jl: Concepts and architecture.