POMDPy: An Extensible Framework for Implementing Partially-Observable Markov Decision Processes in Python

Patrick Emami[1], Alan J. Hamlet[2], and Carl D. Crane[3]

***Abstract*— As of late, there has been a surge of interest in finding solutions to complex problems pertaining to planning and control under uncertainty. A popular way to approach this task is to formulate the problem at hand as a partially-observable Markov decision process (POMDP). Indeed, a plethora of research has been dedicated towards finding ways to circumvent the curse of dimensionality and curse of history that plagued early approaches to solving POMDPs. The purpose behind creating POMDPy has been to develop an easy-to-use, open-source software framework that will allow researchers to benchmark state-of-the-art solvers on custom problems and to add their own innovative solutions to a growing collection. This work has also been motivated by the desire to see these algorithms integrated into systems that solve real-world problems. POMDPy's default solver uses Monte-Carlo Tree-Search to help it scale to high-dimensional problems.**

We chose to use Python to develop POMDPy due to Python's rapidly-growing popularity in the scientific community, easy-to-read syntax, portability, and the abundance of useful numerical libraries. Python also allows POMDPy to interface easily with many different technologies, including ROS and Tensorflow. Source code for POMDPy can be found at `http://pemami4911.github.io/POMDPy/`

I. Introduction

This article introduces POMDPy, an open-source software framework for solving POMDPs that aims to facilitate further research in planning and control under uncertainty. POMDPy provides a platform for developing, testing, and analyzing new POMDP algorithms. The framework's default solver offers a slightly modified version of the popular POMCP [15] algorithm as an implementation of a discrete POMDP solver. It also contains an implementation of Lark's Pruning Algorithm [5] for performing classic Value Iteration. It should be noted that the framework was designed in such a way as to make it straightforward to implement continuous solvers as well. The framework currently supports testing with two benchmark problems, the RockSample and Tiger problems. A presentation of results on the RockSample problem is provided in Section 3.

Up to now there has been a lack of easy-to-use and extensible platforms for implementing POMDPs. As a result, researchers often resort to implementing their own solvers entirely from scratch. The most widely used framework was created by A. R. Cassandra [4]. One of the main contributions of Cassandra's POMDP software framework was that it defined a unique file format for simplifying the task of providing the parameters required to specify a POMDP problem. This file format has been used by many researchers in an effort to make their experiments repeatable. However, the model for the POMDP must be explicitly defined in this config file, which limits this framework from scaling up to larger problem domains. In contrast, one of POMCP's defining characteristics is that it can interface with a "black-box" simulator for learning transition probabilities. The benefits of using a simulator will be discussed in more detail in Section 2. All of the solvers in Cassandra's framework use the basic dynamic programming approach, and many of the algorithms rely heavily on linear programming.

Another planning and control software framework that has been used in the literature is GPT (General Planning Tool) [2]. It uses a typed logical language for describing problems in a high-level manner. While GPT has been used to solve POMDPs, as seen by the implementation of the approximate POMDP solver RTDP-Bel [3], it is more often used for solving planning problems with A* search and probabilistic planning problems with a heuristic called Real-Time Dynamic Programming.

Other open-source software frameworks of note include the Multi-Agent Decision Process toolbox [17] for solving Dec-POMDPs and the TAPIR toolkit [7], which offers an implementation of the Adaptive Belief Tree (ABT) and Generalized Pattern Search-ABT (GPS-ABT) algorithms. The core of the ABT algorithm also pulls heavily from POMCP; some of the inspiration for the POMDPy framework was derived from TAPIR. More recently, the Approximate POMDP Planning Toolkit (APPL) has gained popularity; it is a solver written in C++ that implements some recent high-performing algorithms such as DESPOT [16].

The rest of the paper is structured as follows. Section 2 provides an introduction to POMDPs and an overview of POMDPy's implementation of the POMCP algorithm. Section 3 presents experimental results from the RockSample benchmark problem. Section 4 details how POMDPy can be used and extended by other researchers in the field, and Section 5 concludes the article with a discussion on how POMDPy will be used to accelerate research in other areas of planning and control.

II. Background

A. POMDPs

When a robot, or *agent*, is attempting to solve a planning or control problem, it often uses sensors to obtain noisy measurements of its environment. By modeling this problem

[1]Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, U.S.A `pemami@ufl.edu`

[2]Department of Mechanical and Aerospace Engineering, University of Florida, Gainesville, FL, U.S.A. `AJHamlet@ufl.edu`

[3]Department of Mechanical and Aerospace Engineering, University of Florida, Gainesville, FL, U.S.A. `Carl.Crane@gmail.com`

at a high level as a POMDP, the agent is able to account for the inherent uncertainty in the measurements it obtains. More specifically, a POMDP can be described by the following [11], [13], [18]:

For simplicity, all events that occur at timestep t will be denoted by a subscript.

- $State$, The current state of the world, denoted by $s \in S$.
- $Action$, The agent executes certain actions, denoted by $a \in A$.
- $Observation$, Through sensors, the agent can obtain a noisy measurement $o \in O$ of the world's state.
- $Reward$, In each state s, the agent receives a reward $r(s) \in R$. In some cases, however, the reward is also dependent on the the action a taken from s, and the reward is given by $r(s, a)$.

POMDPs are further characterized by the following distributions:

- The $transition\ model$, A specification of the outcome probabilities for each action in each possible state, denoted by $T(s_{t+1}, a_t, s_t)$. We will assume that transitions are Markovian, in the sense that reaching state s' after selecting action a is only contingent on state s, and not on the history of earlier states.
- The $observation\ model$, A specification of the probabilities of perceiving the observation o in state s after choosing action a, denoted by $O(s_t, a_t, o_t)$.
- The $initial\ state\ distribution$, $Pr(s_0)$, which specifies the initial distribution of states at time t = 0.

A policy π is a solution to a POMDP that details each action an agent should choose given a belief state. A belief state b is a probability distribution over all possible states. Given the current belief state $b(s)$, the next belief state is calculated by a belief update given by:

$$b_{t+1}(s_{t+1}) = \alpha O(s_t, a_t, o_t) \sum_s T(s_{t+1}, a_t, s_t) b(s_t) \quad (1)$$

where α is a normalizing constant to make the belief state sum to 1.

For the discounted infinite-horizon problem setting, the overall goal of solving the POMDP is to find an $optimal\ policy$ that maximizes an agent's expected total future reward. The planning horizon can be set by the discount factor $\gamma \in (0, 1)$. The value function for a policy π is thereby computed as

$$V(b, \pi) = \sum_{t=0}^{\infty} E[\gamma^t R(s_t, a_t)|b, \pi] \quad (2)$$

Hence, the agent seeks a policy π^* that maximizes $V(b, \pi)$:

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, V(b, \pi) \quad (3)$$

Discrete real-world planning problems tend to have incredibly large state spaces; a simple 10×10 grid world with 10 available actions at each state has a transition model of magnitude 10^5. A continuous planning problem has an infinite state space. This rapid blow-up of the magnitude of the state space for a planning problem is known as the $curse\ of\ dimensionality$. Consequently, the transition and observation models are not able to be specified explicitly for these problems and solving (1) becomes computationally intractable.

The $curse\ of\ history$ that inevitably befalls many POMDP solvers is due to the fact that the number of histories that must be evaluated is exponential in the planning horizon. A history H is a sequence of states, actions, observations, and rewards. A single history entry, $h_t \in H$, consists of a tuple (s_{t+1}, a_t, o_t, r_t).

A large amount of research has been dedicated towards breaking both of these curses. A popular approach is to use Monte-Carlo sampling from a simulator [8], [9], [15], [20], [18]. The POMCP algorithm introduced in [15] uses this method; experimentation showed that it scaled significantly better than SARSOP [10], a top-performing full-width offline planner, on the Rocksample problem [15]. POMCP's promising results were why we chose to use an implementation of it as the default algorithm for the POMDPy framework.

B. Overview of POMDPy's Implementation of POMCP

For a full description of POMCP and proofs of convergence, see [15]. The defining characteristics of this algorithm that will be mentioned in this paper are the simulator and the use of Monte-Carlo Tree Search (MCTS) [6].

1) Learning Transitions from Simulation: Consider a simulator S that provides a sample of a successor state, observation, and reward given a state and action, $(s_{t+1}, o_{t+1}, r_{t+1}) \sim S(s_t, a_t)$. This simulator is used to generate sequences of experiences, known as history sequences or episodes. In turn, these are used to update the value function without ever having to look inside the black box that acts in place of an explicit transition and observation model. The simulator is a key component of POMCP and many other modern day Reinforcement Learning algorithms; solvers that do not explicitly define a transition and observation model are able to navigate around the curse of dimensionality effectively. If a simulator is able to be obtained for a highly-dimensional planning or control problem, then the added computational complexity induced by the size of the state space becomes negligible.

2) Monte-Carlo Tree Search: MCTS is an algorithm that consists of iteratively running random simulations from the root position of a search tree [6]. Central to the success of MCTS is the action-selection process; if exploration is handled appropriately, then MCTS converges to the optimal policy. Figures 1 and 2 are illustrations of POMDPy's implementation of MCTS.

POMDPy creates a belief search tree Bt consisting of belief nodes Bn and action nodes An. Each Bn represents a belief state.The root node of Bt represents the agent's current belief state, which is approximated by a point-set of state particles generated by sampling from the initial state distribution $Pr(s_0)$. The Bn that are direct descendants of the root node are candidates for the agent's next future belief

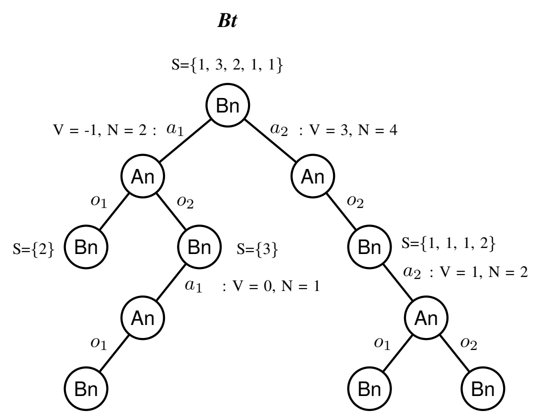

Fig. 1. A depiction of the POMDPy belief search tree in an environment containing 3 states, 2 actions and 2 observations. The root of the search tree is the agent's current belief state; the agent performs simulations to expand the tree. The expected values of choosing actions a_1 and a_2 from the current belief state are given by their mean return.

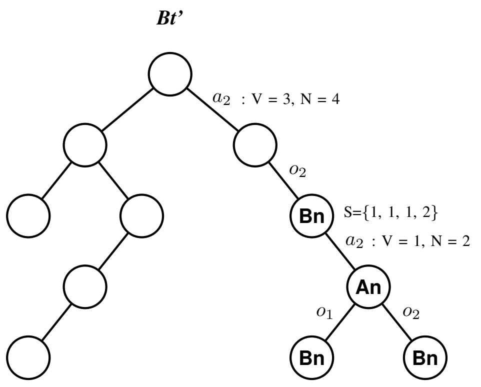

Fig. 2. The agent uses the belief search tree to select a real action (a_2) and real observation (o_2) from the current belief state in order to carry out a belief update. The remainder of the tree is pruned, and the new root of the search tree becomes the selected belief node. Particle reinvigoration may be needed to replenish the state particles for the new root of the search tree.

state. The path through Bt from the root node to a leaf node represents a history sequence h.

The belief search tree Bt is constructed by running multiple simulations. Each simulation is carried out by sampling a random state s_t from the current belief state distribution and generating a history sequence by means of continuously running state samples through the generative model. The sampled state s_t is added to the particle set of the first new belief node Bn encountered during that simulation. The agent's current belief state is updated through a particle-filtering process after a specified number of simulations have been run or a time-limit has been reached. At this point, an action a_t extending from the agent's current belief state is chosen by an action-selection strategy and executed. A corresponding observation o_{t+1} is observed and a new belief state is reached, corresponding to one of the belief nodes Bn added to Bt during the simulations. The other unreachable nodes in Bt are pruned away. The new belief state's particle set is reinvigorated to ensure that it contains a minimum number of particles. This is done by uniformly sampling a state s_t from the old belief state and passing s_t into the simulator to obtain a successor state s_{t+1} and observation o_{t+1} from $(s_{t+1}, o_{t+1}, r_t) \sim S(s_t, a_t)$. If the real observation and simulated observations match, the successor state s_{t+1} is added to the particle set of the new belief state. This approximation to the belief state approaches the true belief state in the limit of the number of state particles used in the particle set.

Every action $a_i, i = 1, ..., N$ where N is the number of legal actions that can be taken from a belief node, specifies a mapping from a belief node Bn to an action node An. An action node An is mapped to future belief states by observations $o_j, j = 1, ..., M$ where M is the number of observations that can be obtained from the current action node An. Each action-mapping keeps track of its current value $Q(b, a)$ and the number of times the action has been tried $N(b, a)$. An overall count $N(b) = \sum_a N(b, a)$ is also kept, which contains the number of times any action has been chosen from the belief node in question.

A uniform rollout strategy is used to expand unexplored belief nodes during simulation; once all actions from a belief node have been attempted at least once, the UCB1 algorithm [8] can be used for action-selection. The idea behind the UCB1 algorithm is to view each belief node in Bt as a multi-armed bandit; therefore, an exploration bonus is added to actions that have been tried relatively few times [1]. The augmented action value $Q^*(b, a)$ is given by $Q(b, a)+c\sqrt{logN(b)/N(b, a)}$. The scalar constant c controls the ratio of exploration to exploitation during action selection [15]; when c is small, the exploration bonus is negligible and UCB1 algorithm acts greedily. Once all of the actions from a belief node have been tried, UCB1 selects the action maximizing the augmented action value, $\operatorname{argmax}_a Q^*(b, a)$. For a proof of the convergence of UCB1 algorithm to the optimal value function, see [1]. During experimentation, it was observed that UCB1 with a c value set to 3.0 showed a significant speed-up over greedy action-selection when the problem size was very large.

At the end of each simulation, the rewards observed after each step of the generated history sequence are propagated back up the belief tree in the form of a Q-update. Let Q represent $Q(b, a)$, Q' represent $Q(b', a')$, N represent $N(b, a)$, and r represent $r(b, a)$. The Q-update is performed by using the Q-learning rule, as seen in [13]:

$$Q := Q + \frac{\alpha}{1 + N}(r + \gamma \operatorname*{argmax}_{a'} Q' - Q) \tag{4}$$

Here, γ is the discount factor and $r(b, a)$ is the immediate reward obtained by choosing action a while in belief state b. The system was naturally quite sensitive to the choice of α, the step-size, and selection of its value required significant fine-tuning.

III. Experimentation

To show off the functionality of POMDPy, we applied our solver to the benchmark RockSample problem. For each problem size and each action-selection strategy, we ran POMDPy for 12 hours of total computation time. Experiments were ran on a Intel Core i7-3610QM @ 2.30 GHz with 8 GB of RAM. The metrics used to assess performance were the total average discounted and undiscounted reward, and the average time per run. With 1000 simulations per step for a given history sequence, the solver took approximately 6 seconds to calculate the next action to during a run. No comparisons were made with other state-of-the-art POMDP solvers, because the purpose of the experimentation was mainly to act as a proof of concept for POMDPy. Results comparing POMCP with other solvers can be found in [15]. The performance benchmark used for our experimentation was simply carrying out the simulations without any tree. A rollout algorithm was used the entire time, where the agent would choose the legal action that maximized their average expected return after sampling all actions that could be taken from each state. Our results are presented in Table 1.

The UCB1 exploration constant for our solver was set to $c = 3.0$. The discount horizon was set to 0.01, which equates to $\gamma = 0.95$. The choice of step-size varied with the size of the problem. For RockSample(7, 8), we found that $\alpha = 0.4$ provided the best results. For Rocksample(11, 11) and RockSample(15, 15), we used $\alpha = 0.6$. These values were hand-tuned after extensive experimentation.

The RockSample(n, k) problem simulates a rover exploring unknown terrain. The environment is discretized as an $n\ x\ n$ grid containing k rocks. The task is to determine which rocks are valuable, take samples of the valuable rocks, and then leave the map to the east when no other rocks are deemed worthwhile to investigate. Additionally, the agent has a sensor that allows it to check rocks near itself to assess whether there are rocks worth exploring nearby. In practice, the agent's initial belief state is a uniformly-sampled vector that encodes the current estimation of the state of each of the k rocks.

On RockSample(7, 8) and RockSample(11, 11), the UCB1 and greedy strategies performed comparably well. However, on RockSample(15, 15), the solver using UCB1 was able to complete 20 runs compared to only 13 runs by the solver using the greedy action-selection strategy. On average, the UCB1 solver finished runs 30 minutes faster for this problem size. Since the UCB1 algorithm encourages exploration, it allows the agent to explore the state space much more effectively than the greedy strategy does; this advantage is crucial for problems such as this one that have over 7 million states.

IV. Contributions to the Community

A. Using POMDPy

The POMDPy software framework currently has a number of different uses. First and foremost, it allows users of the software to rapidly implement and test their own POMDP problems with POMDPy's implementation of the POMCP solver. Users are able to choose how they would like to specify the set of all actions, states, and observations; by default, actions and observations are represented by an enumerated set. The majority of the programming work that the researcher must undertake is to simply to extend an abstract Model class that describes the simulator, and optionally to extend an abstract HistoryData class. Defining a concrete implementation of the Model class allows users to generate history sequences for their problem. The simulator for the RockSample problem parses a text file that contains the $n\ x\ n$ grid with the points of interest marked accordingly; this encodes all of the transition probabilities of the POMDP, saving the implementer a large amount of work. The HistoryData class allows the user to encapsulate data obtained by observations and define domain-specific behaviors that affect belief updates; however, in some instances the effect of an observation on generating the next history entry can be handled entirely by the generative model. In the future, POMDPy's public API will be simplified further to make it even easier to use, by providing users with the ability to specify a problem and desired solver through a single interface.

POMDPy's design is intended to make it simple to test state-of-the-art solvers against each other. The framework is abstracted in such a way that there is a separation between the abstract components that make up a general POMDP and the various concrete implementations. Therefore, if a user would like to test their own solver against one of POMDPy's solvers, they only need to extend the abstract POMDP components as needed to implement their own algorithm. Additionally, a user has the ability to change the granularity of customization; if they would like to compare an action-selection strategy against the UCB1 algorithm, they are also able to easily swap out action-selection strategies within the framework and compare results. POMDPy conveniently keeps track of statistics during run-time, which can be easily plotted with Python's *matplotlib* package.

In summary, POMDPy is a work-in-progress that aims to bring POMDPs closer to integration with real-world problems. The next section describes the steps that are being taken to achieve this.

B. Extending POMPDy

POMDPy fills the need for an open-source, community-driven platform that will keep POMDPs relevant. Naturally, one of the main attractions of POMDPy is its extensibility. It is our intention that this software framework be continuously built up over time by the community so that it remains on the cutting-edge of POMDP research. For example, the default POMDP implementation currently supports only discrete planning and control problems; however, the framework is designed in such a way that adding support for continuous POMDPs is straightforward. As benchmark problems move more towards real-world robotic planning and control problems, continuous-space POMDPs will begin to be seen more regularly.

The POMDPy framework will also benefit from the addition of new POMDP solvers and heuristics as more research is conducted in this area. A potential addition to

	RockSample(7, 8)		RockSample(11, 11)		RockSample(15, 15)	
$States\|S\|$	12,544		247,808		7,372,800	
$Action - selection$	UCB1	Greedy	UCB1	Greedy	UCB1	Greedy
Ave. Time/Run (sec)	230.88	207.07	711.17	686.67	2489.75	3866.15
Runs	188	209	61	63	20	13
Ave. Discounted Return	8.29 ± 0.478	7.58 ± 0.422	3.42 ± 0.56	4.35 ± 0.47	2.06 ± 0.398	2.07 ± 0.621
Ave. Undiscounted Return	27.9 ± 0.984	25.0 ± 0.875	34.9 ± 2.04	34.9 ± 1.45	44.0 ± 2.39	41.5 ± 4.85

TABLE I

RESULTS FROM RUNNING ROCKSAMPLE WITH THREE DIFFERENT CONFIGURATIONS. EACH EXPERIMENT RAN FOR 12 HOURS WITH 1000 SIMULATIONS PER STEP.

the framework would be an implementation of SARSOP [10], DESPOT [16], ABT [11] and GPS-ABT [14] from Kurniawati et al. SARSOP is an excellent offline POMDP solver to use as a benchmark for testing new solvers against. ABT provides real-time POMDP re-planning functionality that makes these planning algorithms much more useful when tested on real-world applications. GPS-ABT provides an initial look at how current state-of-the-art POMDP solvers can be extended to continuous action spaces.

Finally, adding on to POMDPy's library of sample problems will provide researchers with a thorough testing suite for their contributions to the literature. Currently, POMDPy has two popular benchmark POMDP problems to test against; the RockSample problem and Tiger Problem. See [19] for an explanation of the Tiger Problem.

V. CONCLUSION

The purpose of this research is to introduce a novel, extensible POMDP software framework. POMDPy offers an implementation of the popular POMCP algorithm as a default solver. It is our intention for POMDPy to act as a centralized location for future POMDP research. Some other areas of POMDP research that we believe should be integrated into POMDPy include heirarchical POMDPs [20] and the computation of large-scale MCTS in parallel on GPUs [12]. NVIDIA offers CUDA Python, which allows scientists to utilize NVIDIA GPUs while maintaining the CUDA layer underneath Python. Taking a heirarchical approach to framing robotic planning and control problems allows highly-complex tasks to be broken down into more manageable sub-problems. The concept of breaking down a planning problem into a structured heirarchy of smaller planning problems is reminiscent of certain cognitive theories that attempt to explain the way in which our own brains process tasks as simple as reaching for and picking up small wooden blocks. This helps to prove the intuitiveness of this branch of robotics. Research on parallelizing large-scale MCTS will be especially useful for finding solutions to continuous-space POMDPs, and is currently being looked at to be added to POMDPy. Artificial neural-networks have been previously employed in order to approximate the highly-nonlinear Q-function for a continuous planner; much excitement has been generated as of late over deep learning networks due to the performance speed-ups offered by GPUs. This is another area that we hope POMDPy will cover as well. There are many promising and exciting avenues of research on POMDPs mentioned in this paper; many others were not mentioned. We hope that POMDPy will contribute signficantly to the task of integrating these ideas in order to accelerate the widespread adoption of POMDPs to real-world robotic planning and control problems. POMDPy will be continuously improved upon so that it facilitates research in all of these areas.

REFERENCES

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[2] B. Bonet and H. Geffner. Gpt: a tool for planning with uncertainty and partial information. In *Proc. IJCAI-01 Workshop on Planning with Uncertainty and Partial Information*, pages 82–87, 2001.

[3] B. Bonet and H. Geffner. Solving pomdps: Rtdp-bel vs. point-based algorithms. In *IJCAI*, pages 1641–1646, 2009.

[4] A. Cassandra. Partially observable markov decision process, 2009.

[5] A. Cassandra, M. L. Littman, and N. L. Zhang. Incremental pruning: A simple, fast, exact method for partially observable markov decision processes. In *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*, pages 54–61. Morgan Kaufmann Publishers Inc., 1997.

[6] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and games*, pages 72–83. Springer, 2007.

[7] D. Klimenko and J. S. and. H. Kurniawati. Tapir: A software toolkit for approximating and adapting pomdp solutions online. In *Proc. Australasian Conference on Robotics and Automation*, 2014.

[8] L. Kocsis and C. Szepesvari. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.

[9] H. Kurniawati, Y. Du, D. Hsu, and W. S. Lee. Motion planning under uncertainty for robotic tasks with long time horizons. *The International Journal of Robotics Research*, 30(3):308, 2011.

[10] H. Kurniawati, D. Hsu, and W. S. Lee. Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In *Robotics: Science and Systems*, volume 2008. Zurich, Switzerland, 2008.

[11] H. Kurniawati and V. Yadav. An online pomdp solver for uncertainty planning in dynamic environment. In *Proc. Int. Symp. on Robotics Research*, 2013.

[12] K. M. Rocki. Large scale monte carlo tree search on gpu. 2012.

[13] S. Russell and P. Norvig. *Artificial intelligence: a modern approach (2nd edition).* Prentice Hall.

[14] K. M. Seiler, H. Kurniawati, and S. P. Singh. An online and approximate solver for pomdps with continuous action space. 2015.

[15] D. Silver and J. Veness. Monte-carlo planning in large pomdps. In *Advances in Neural Information Processing Systems*, pages 2164–2172, 2010.

[16] A. Somani, N. Ye, D. Hsu, and W. S. Lee. Despot: Online pomdp planning with regularization. In *Advances in neural information processing systems*, pages 1772–1780, 2013.

[17] M. T. Spaan and F. A. Oliehoek. The multiagent decision process toolbox: software for decision-theoretic planning in multiagent systems. In *Proc. of the AAMAS Workshop on Multi-Agent Sequential Decision Making in Uncertain Domains (MSDM)*, pages 107–121, 2008.

[18] S. Thrun. Monte carlo pomdps. In *NIPS*, volume 12, pages 1064–1070, 1999.

[19] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[20] N. A. Vien and M. Toussaint. Hierarchical monte-carlo planning. 2015.