Optimally Escaping a Room Using POMDPs

Zachary Farnsworth
Stanford University
450 Serra Mall
zacharyf@stanford.edu

Charles Furrer
Stanford University
450 Serra Mall
cfurrer@stanford.edu

Adam Thorne
Stanford University
450 Serra Mall
athorne@stanford.edu

Abstract

Partially Observable Markov Decision Processes (POMDPs) are becoming more and more ubiquitous as technology continues to develop throughout the world. Examples include Professor Kochenderfer's Aircraft Collision Avoidance algorithm [1], autonomous robots, marketing policies, and militaristic applications (moving target search, search and rescue, etc.)[2]. This paper focuses on documenting POMDP solving techniques applied to the following hypothetical scenario: A Roomba robot is placed in a room and has the goal of navigating its way to a goal wall (labelled green in the simulation). The robot knows the layout of the room (dimensions and spacing), but doesn't know where it is placed. The robot is equipped with a bumper sensor that can tell when it hits a wall, but that is its only way to collect information on its location. A reward of -0.1 is given for every time step in the simulation and -1.0 for every time a wall is hit. The simulation continues until the robot either reaches a portion of the wall designated as the "stairs" (labelled in red in the simulation), resulting in a reward of -10, or reaches the goal, corresponding to a reward of +10. This paper describes the implementation and evaluation of ARDESPOT, QMDP, and continuous and discrete space Monte Carlo Tree Search solving techniques applied to the previously described POMDP scenario.

1. Introduction

In a Markov Decision Process (MDP), an agent chooses action a_t at time t, which results in a transition to state s_t and a reward r_t [3]. In a POMDP, the state of the agent is uncertain, and the agent must make choose actions to take based on observations that it makes. In both POMDPs and MDPs, policies are created that aim to maximize the agent's cumulative reward in the state space or state and observation space described in the model.

1.1. POMDP Solving Methods

Two main methods exist for solving POMDPs: offline and online methods. Offline methods compute a policy through implementation of whatever solving algorithm is being used prior to execution of the policy in the model, while online methods compute the optimal policy in real time, based in the current state of the model.

Offline methods typically find an approximate optimal solution unless value iteration is being used. POMDP value iteration is slightly different than MDP value iteration, as in the POMDP implementation, alpha vectors are calculated for incremental one step plans, with dominated alpha vectors (alpha vectors that result in a policy with a lower utility than any other policy at a given belief state) being thrown out and not used for calculation in the consequent one step plan. Other offline POMDP solving techniques include QMDP, Fast Informed Bound (FIB) and Point Based Value Iteration.

Online solving methods plan an optimal policy for the POMDP based on the given belief state the model is in (typically up to a certain depth specified in online method algorithms). Methods include Forward Search, One-Step Lookahead, and Monte Carlo Tree Search.

2. Related Work

We were initially inspired by "*Robust Online Belief Space Planning in Changing Environments: Application to Physical Mobile Robots*" by Agha-mohammadi et. al. [4] In the paper, online methods are used and extended to try to help a robot navigate out of a room full of obstacles. Online methods were shown to have very high success rates. Additionally, the researchers implemented a novel policy (a rollout-based extension of FIRM

(Feedback-based Information RoadMap)), which had even better results. This convinced us that online methods would likely be a successful approach to our own Roomba problem.

In “Predictive Autonomous Robot Navigation” [5], the researchers used online POMDP methods to help a robot navigate through a crowded space full of moving obstacles. While we do not have moving obstacles, the same principles of online methods still apply to our research, and the success of this research furthers our desire to explore online methods as a solution.

Another group of researchers attempted to use neural networks to help a robot navigate autonomously, in a paper titled “Real-time autonomous robot navigation using VLSI neural networks” [6]. The results of these experiments were quite successful, however, it should be noted that these robots were equipped with infrared sensors, while our Roomba will only be using bumper sensors. Nevertheless, the success of these experiments would make neural networks an interesting approach to explore despite the differences in the constraints of the problems.

3. Methods

3.1. Environment Setup

We used a code base that set up the problem as a POMDP, including the declaration of a sensor and action spaces for the roomba, as well as declarations of the possible states and observations of the room. The observation space is defined by implementing a particle filter. The code base also sets up functions to simulate the performance of the robot based on a calculated policy. When executing a given policy in the environment, the belief state is also updated by means of the particle filter based on the observations of the Roomba through its movement and bumper sensors.

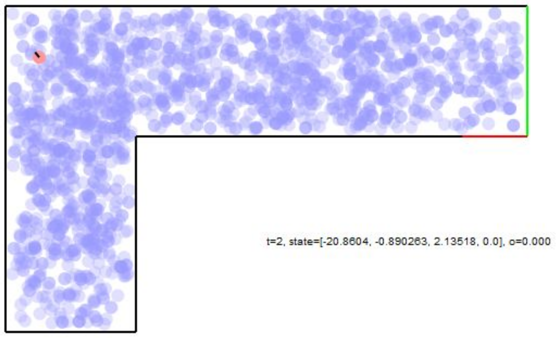

Figure 1: Roomba and its belief state (blue circles) after 2 time steps. Goal marked in green, stairs marked in red.

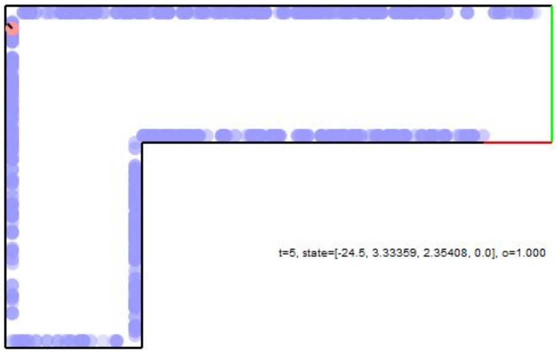

Figure 2: The same simulation after 5 time steps. Notice the belief state is much smaller due to the Roomba having hit a wall

3.2. Evaluation Metric

We created a custom evaluation metric in order to be able to compare performance between different policies created by different POMDP solving algorithms (or our baseline policy). Additionally, it should be noted that we will be evaluating all of our policies using a maximum of 100 time steps.

If a roomba has not reached a terminal state by the end of 100 steps or so, it is possible that it never will, and quite likely that even if it does it will not for many steps more, incurring many more negative penalties along the way. Because of this, it seemed to us that the fact that a policy successfully completes after a large fixed number of steps is the most important thing in selecting the "best policy". Because of this, we define our evaluation metric as follows:

$$score = \left(\frac{\#\ succesful\ trials}{total\ trials}\right)^2 * MTR$$

where MTR is the mean total rewards for the successful trials. This metric stresses the importance of successfully finding the goal state so as to "stop the bleeding" of endlessly looping in dead ends, since in a real simulation, the Roomba would not stop incurring negative rewards after 100 time steps.

3.3. Baseline Policy

The roomba calculates its state as the mean of all of its belief states, and then computes the angle between its state and the goal state before moving in the direction it believes the goal to be. However, the roomba will also, for 4 (consecutive) time steps out of every 20 time steps, turn to the left and move forward as an exploratory strategy.

The idea here is that the roomba will often get stuck if it only gains information by bumping into walls. By deterministically turning and moving 20% of the time, this should help the roomba remove itself from dead ends. We chose this policy because it is quite naive, yet still produces a good result a fair amount of the time. The state and action spaces used for this policy were both continuous.

3.4. ARDESPOT

ARDESPOT is an online search algorithm similar to forward search and branch and bound algorithms in that it searches for an optimal policy for a tree (based on a given belief state) up to a given depth. This is done by looping over the entire action and observation spaces for a given node and summing the expected rewards for taking a certain action and seeing a certain observation from the given node. A weakness of this approach is that calculating the optimal policy for a very large depth and extremely large state, observation and action spaces can become intractable. This is where a DESPOT comes into play.

A Determinized Sparse Observable Tree (DESPOT) essentially "prunes" the decision tree used in forward search by only evaluating policies based on K sampled scenarios, where a scenario is a set of belief spaces included in the tree while the others are left out.

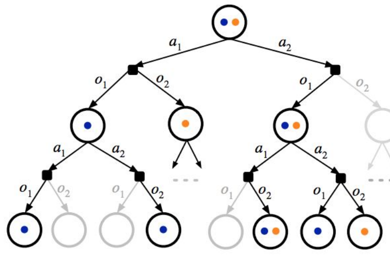

Figure 3: Illustration of a DESPOT from Somani et. al. [7]. The black and gray lines together represent the entire tree (used in forward search or branch and bound algorithms), while the black lines only represent a DESPOT, where only certain belief states (labeled as any combination of a single blue circle, single orange circle or blue and red circle) are evaluated in the decision tree.

One weakness of DESPOTs is the optimal policy computed is only an approximate estimation of the true optimal policy if the policy space is small and the subtrees for the DESPOT are small. Regularized DESPOTS or RDESPOTs remedy this by implementing default policies for subtrees in the DESPOT that are over a certain size.

Anytime RDESPOTs (ARDESPOTs) further improve on RDESPOTs by greedily ruling out subtrees that result in a lower bound value for the algorithm, and greedily choosing values above an upper bound for the algorithm.

We implemented ARDESPOT with a continuous state space and discretized action and observation spaces. The action space was limited to combinations of a velocity of 5.0 and angular turning velocities in the set {-0.5, 0, 0.5}. The observation space is limited to whether you hit a wall or not, {0, 1}. The upper bound of the algorithm was 0 (as any reward above 0 is desirable) and the lower bound was -9 (allowing for traversing the room and hitting a few walls) in our implementation.

3.5. QMDP

The second algorithm we experimented with was QMDP, an offline method that creates a set of alpha vectors for each action based on the the state-action value function under full observability [3]. These alpha vectors can be computed using value iteration, and used to estimate the value function. Hitting a wall provides quite a bit of information to the Roomba, so it is likely that QMDP may have issues with this problem, since it tends to have issues with problems with information-gathering actions.

We discretized both the state and the action spaces for QMDP. The state space divides the room into 100 x-points, 100 y-points, and 20 theta-points. The action space is the set of combinations of a velocity of 3.0 and angular turning velocities in the set {-0.5, -0.25, 0, 0.25, 0.5}. We chose this particular discretization because it offered a good range of options while still allowing the algorithm to complete in a reasonable amount of time. We also decided that the Roomba should always be moving, so we restricted all actions to have a velocity of 3.0.

3.6. POMCP

The third algorithm we used was Partially Observable Monte-Carlo Planning, or POMCP for short [8]. POMCP builds on the Monte-Carlo search to provide a high-performance online method for approximating best actions. A traditional Monte-Carlo approach involves sampling only start states from the belief state to overcome the curse of dimensionality. POMCP's innovation is to use a flavor of Monte-Carlo sampling to histories as well. It uses a simulator to create history samples and reduce dimensionality in that spacel, allowing it to perform much faster than a typical Monte-Carlo search.

We discretized the action space for POMCP, and used a continuous state space. The action space is the set of combinations of a velocity of 5.0 and angular velocities in the set {-0.5,0,0.5}. We chose this particular discretization because POMCP seemed to be better at learning where it was trying to go, so we decided that giving it a fixed high velocity of 5.0 was ideal. We had to limit the number of options for the turn-rate due to the high computational requirements that running this online algorithm entails. Expanding the range of actions led to much slower run times.

3.7. Modifications to POMDP Policies

After testing our policies for ARDESPOT, QMDP, and POMCP on random data, the results were somewhat mediocre, and we could see from visualizing our simulations that the Roomba would frequently get stuck in walls. Because of this, we made a simple modification to our policies. For ARDESPOT, QMDP, and POMCP, we select an action for the Roomba based on the corresponding policy, unless the Roomba is currently in contact with a wall, in which case we turn with an angular velocity of $-\pi$ and move with velocity of 5.0. This simple modification greatly improved the performance of all of our POMDP policies. Thus, it should be noted in the following sections that our experiments on ARDESPOT, QMDP, and POMCP are actually using this modified policy.

4. Experiments

We ran each of our 4 algorithms (baseline, ARDESPOT, QMDP, POMCP) on 3 different room configurations (different goal and stairs locations, but the same room shape). For each run on each configuration, the policy was tested on a batch of 100 different random initializations for the Roomba, i.e., the Roomba starts at a different location and orientation within the room on each of the 100 runs. We then computed the score for each of these batches on each room configuration and for each algorithm, using our previously defined evaluation metric to compute each score.

5. Analysis of Results

5.1. QMDP

QMDP had the most consistent results of all the models. The success rates were 93.0%, 84.0%, and 76%, with scores of 1.31, .674, and .316 for configs 1, 2 and 3, respectively. This was expected, as QMDP is a well known algorithm that converges to an approximately optimal solution. Overall, QMDP had the best success rate performance, and produced the most consistent performance for each configuration. The only outlier performance was on room number three, where it scored very low relative to the other configurations.

5.2. ARDESPOT

ARDESPOT performed the worst out of all of the implemented algorithms (even worse than the baseline algorithm). The success rates were 21.0%, 14.0% and 11.0% with scores of 0.374, 0.156 and 0.091 for configs 1, 2 and 3, respectively. We did not expect this algorithm to behave more poorly compared to other online and offline methods that we implemented.

We suspect that our implementation of ARDESPOT was not optimal. We passed in float limits into the function (-9.0 and 1.0 for the upper and lower bounds), and what was likely needed for a more successful implementation was passing in a heuristic function to the solver. ARDESPOT.jl uses a random rollout policy to calculate the lower bound - we would try to implement this as a starting point in the future to see if this algorithm could be more effective.

Although ARDESPOT was designed to scale up to large state spaces better than DESPOT or RDESPOT algorithms, Somani et. al. explains that when implemented with extremely large state spaces ARDESPOT can work if a good, small policy exists [7]. The best policies for this POMDP are not going to be small, as we have 200,000 states that are possible with discretization alone (this implementation uses a continuous state space, but the relatively large discretized state size illustrates that optimal policies would be complex). This would lead to extremely complex policies, which would make ARDESPOT a poor algorithm to use for this POMDP.

5.3. POMCP

POMCP performed the best out of all of the implemented algorithms in terms of score. The success rates were 84.0%, 20.0% and 66.0% for configs 1, 2 and 3, respectively. It scored 2.598, .265, and 1.669 on each configuration. Overall, POMCP had the best score performance, which supports the hypothesis that online methods are better suited for this problem than offline methods.

5.4. Graphs

To analyze our results consistently across all of the configurations, we divided each model's score by the maximum score observed for that configuration. This resulted in a "Norm Score" metric that presents a configuration score as a percentage of the best score observed. This led to a nice set of graphs, where we compared success percentage against Norm Score. The graph results can be seen in figs 4 through figs 7 below.

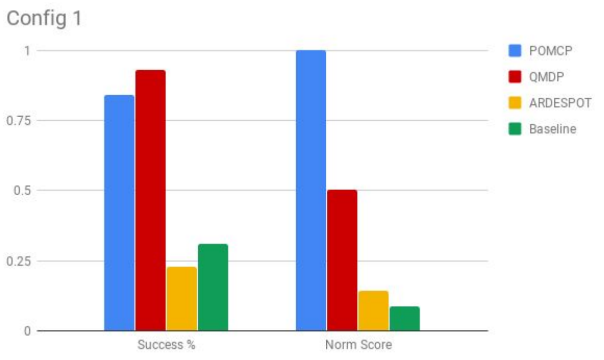

Figure 4: Performance of each algorithm on configuration one

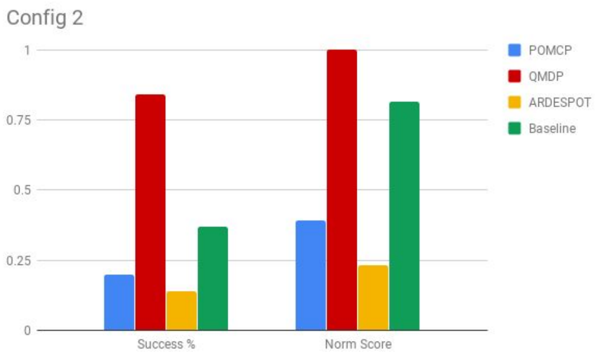

Figure 5: Performance of each algorithm on configuration two

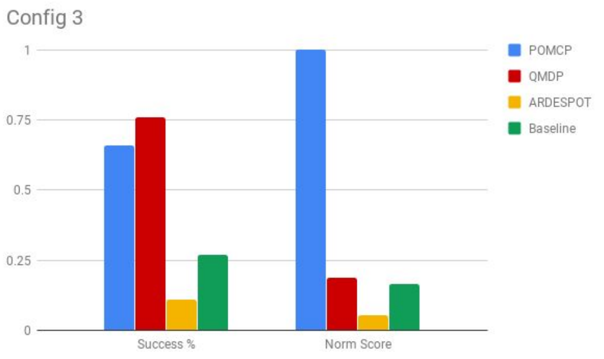

Figure 6: Performance of each algorithm on configuration three

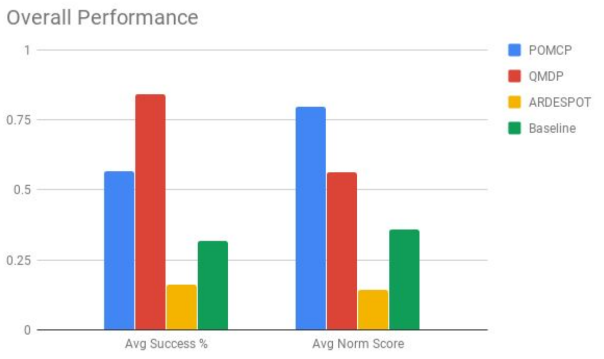

Figure 7: Average performance of each model

As you can see in Figure 7, the best performing models were POMCP and QMDP. QMDP had the highest success percentage across all configurations, with an average of around 85% success rate. However, for our proposed scoring metric, POMCP performed the best with an average normalized score of .8 . This means that POMCP might have been successful less often than QMDP, but when it was successful it found its way to the goal very quickly.

6. Conclusion

POMDPs are a powerful tool for solving problems in which there exists a great deal of uncertainty. Because of this, it should come as no surprise that POMDP methods perform quite well at helping a Roomba navigate out of a hypothetical room in an optimal way. Even with a minimal amount of observable data using very basic sensors, both offline and online methods were proven to be much more effective than random or naive policies. Overall, we conclude that the success of POMDPs in this simple experiment could translate into success in more advanced areas of autonomous control, such as self-driving cars.

6.1. Future Work

Given more time and resources, other areas that would be interesting to explore would be the use of neural networks in solving such a problem. This could be quite difficult to implement, but given the success of neural networks in other areas of AI, it would be worth exploring. The use of larger datasets and scenarios with multiple goals/stairs would also be an interesting extension to this problem. The variations and possible solutions to this problem are quite extensive, and could produce valuable results for autonomous control technologies.

6.2. Contributions

Zachary did much of the groundwork for formulating a baseline policy and establishing an evaluation metric. He also worked extensively on running and evaluating QMDP methods, as well as explored relevant literature.

Charlie worked on applying POMCP to the problem, training, and evaluating it. He did lots of research into saving and standardizing the models produced. He also aggregated the results and produced visuals and analysis.

Adam did initial research into narrowing down which algorithms would be best to implement for the Roomba POMDP and how to implement them in code.. He spent extensive time implementing, researching and documenting results for the ARDESPOT algorithm. He also maintained the GitHub for the project.

References

[1] Bai, Haoyu, et al. "Unmanned aircraft collision avoidance using continuous-state POMDPs." *Robotics: Science and Systems VII* 1 (2012): 1-8.

[2] Cassandra, Anthony R. "A survey of POMDP applications." *Working notes of AAAI 1998 fall symposium on planning with partially observable Markov decision processes*. Vol. 1724. 1998.

[3] Kochenderfer, Mykel J. *Decision making under uncertainty: theory and application*. MIT press, 2015.

[4] Agha-Mohammadi, Ali-Akbar, et al. "Robust online belief space planning in changing environments: Application to physical mobile robots." ICRA. 2014.

[5] Foka, Amalia F., and Panos E. Trahanias. "Predictive autonomous robot navigation." *Intelligent Robots and Systems*, 2002. IEEE/RSJ International Conference on. Vol. 1. IEEE, 2002.

[6] Tarassenko, Lionel, et al. "Real-time autonomous robot navigation using VLSI neural networks." *Advances in neural information processing system*s. 1991.

[7] Somani, Adhiraj, et al. "DESPOT: Online POMDP planning with regularization." *Advances in neural information processing systems*. 2013.

[8] Silver, D., & Veness, J. (2010). Monte-Carlo Planning in Large POMDPs. In *Advances in neural information processing systems* (pp. 2164–2172).