Self-Driving Under Uncertainty aa228 Course Project

Paul Shved
pavel.shved@gmail.com

Fall 2019
AA228 class at Stanford University

***Abstract*—This project focuses on building an online policy for a simplified self-driving car problem, and on studying the effects of perception accuracy on the efficiency of the planned route. The planning objective is formulated as Partially Observable Markov Decision Process. As the planning algorithm we use a Monte-Carlo based online policy algorithm POMCPOW that works with continuous state and action spaces. The quality of the planning decisions is evaluated in a simulator on three simple scenarios. We find that with certain fine-tuning, POMCPOW produces driving decisions that result in mission completion albeit suboptimally. However, for more complicated scenarios, the quality of planning decisions degrade and the agent gets stuck without completing the mission. We find that the agent behavior depends more on the domain knowledge embedded into the policy algorithm that on the sensor quality, but detailed exploration of this effect is left to future work.**

I. INTRODUCTION

This project focuses on building an online policy for an autonomous agent that mimics a self-driving car. The environment is very constrained and consists of one lane of road and one "obstacle" car (see section III). The objective of the policy is to complete the mission (drive until the finish line) while not running into the obstacle.

In the literature on autonomy, "perception" typically refers to the software components that observe the state of the world. [5] [1]. The decision of what the car agent ("hero") does next is produced by the "planning" components, based on the output of "prediction", which in turn take results from the perception.

Building policy for self-driving cars is not a solved problem. Replacing personally-owned cars with self-driving autonomous cars will have beneficial effects on climate, productivity, and accident rate. However, many challenges remain; according to [2] perception algorithms are computationally intensive and perform poorly in adverse conditions.

In this work, we aim to study the effects of imperfections of perception algorithms on planning by building a planning policy and studying its performance while varying perception precision as a hyperparameter (see section V)

In the model we use, the agent can use a lidar-like sensor to probe the location of a singular obstacle that may be in or outside the lane. The sensors return noisy data according to the emission model, for which we are going to use a Gaussian as in [9]. The agent builds a belief about the obstacle location and can take actions (e.g. "accelerate", "decelerate", "brake hard", "maintain speed") to drive around it if needed. As per [8], we will also apply a small negative reward at each time step to penalize overly careful policy. We found that further augmentations to the policy are needed; see section III-F.

This project's objective is to study the effects of the sensor accuracy on the quality of planning decisions. In order to achieve this, we formulate the problem as a Partially Observable Markov Decision Process (see section III-A) and use POMCPOW [6], a Monte-Carlo Tree Search-based online policy solver for the autonomous agent (section IV). The online policy is first evaluated on simple scenarios (sections VI-A-VI-B.), and then the study of the perception accuracy is conducted (section VI-C).

II. RELATED WORK

Existing work explores diverse methods for policy planning for autonomous agents. We survey some results that focus on obstacle evasion and policy learning (as opposed to an environment with adversarial agents) and aim to study the efficiency of planning in presence of obstacles.

For example, in [8] the authors build a policy for aircraft that avoids obstacle. The goal of the work is to improve efficiency by proving that rerouting of all air traffic is not required to ensure safe traversal of the airspace. The authors later show that the policy can be encoded into Coordination Tables [7].

In this paper, the state space is disctetized into small quadrants (e.g. the x and y coordinates were discretized with the precision of 5000 ft.) The paper additionally utilizes "Adaptive Space Discretization", improving model precision in the regions where the debris trajectory intercepts the plane trajectory. This discretization allows to build an optimal policy offline and encode it into a simple human-readable representation [7].

Space discretization is appropriate for corase-grained planning for aircraft in airspace (since distances between agents are large and decisions take effect over many timesteps). However, for self-driving in urban environment, decisions take almost immediate effect, and continuous state space is more appropriate.

Markov Decision processes over continuous have applications in self-driving cars specifically. For example, [9] focuses on identifying patterns in the driving behavior. The paper proposes a model for sensor perception of other agents that's based on adding Gaussian noise. The resultant observations are aggregated with the actions that human drivers take, and "primitives" are extracted that are indicative of a certain behavior. While the paper doesn't focus on improving efficiency, this is a successful application of Markov Decision Process to modeling a self-driving car environment.

Another approach to working with noisy observations in continouos environments is Kalman Filter and variations [3].

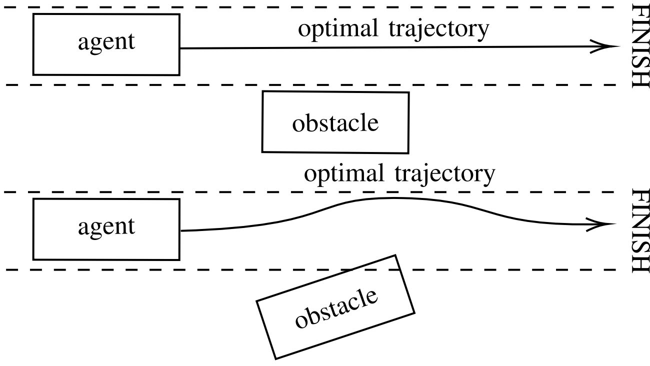

Fig. 1. Illustration of the self-driving problem studied in this project with two different positions of obstacles and oprimal trajectories plotted.

Kalman filters produce a provably optimal policy when the reward function is quadratic, and transition and observation models are linear Gaussian functions. While this approach could be used for the current project too, some simplifications we employed make the model nonlinear. Additionally, this approach might not be applicable for more complex scenarios, whereas Monte-carlo methods in continuous spaces do not have this restriction.

III. Problem Formulation

At a high-level, the problem studied in this project is to drive in a straight line, potentially routing around the obstacles. Figure 1 shows an illustration of two scenarios. The following simplifications are employed:

- The behavior of other agents is non-adversarial.
- The behavior of other agents does not change in response to the actions taken by the hero.
- The environment is two-dimensional.
- Only driving in a straight line is considered (no lane changes).
- The time is discretized to a resolution that is close the perception-planning loop latency of a state-of-art self-driving car.

A. POMDP model

Based on the description, we formulate the partially observable Markov Decision Process problem in figure 2.

B. State Space

The state space we choose to model the problem reflects the position of the agent and of the obstacles.

State space $\mathcal{S}$ is a tuple that describes the state of the agent and the obstacles. It consists of the following elements:

- d_t^x and d_t^y - position of the front right corner of the agent
- b_t^x and b_t^y - position of the rear left corner of the obstacle
- b_t^x and b_t^y - position of the rear left corner of the obstacle
- v_t^x and v_t^y - velocity of the agent
- a_t^x and a_t^y - acceleration of the agent

All state values are continuous and expressed in meters. A state S is terminal if $d_S^x > X$ or if it's a special state $Collision$

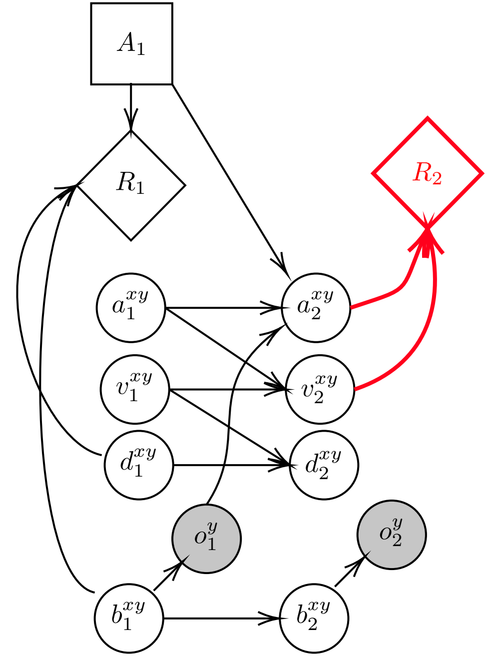

Fig. 2. Driving problem structure. The states represent the position of the agent and the obstacle. The actions represent how controlling acceleration is the only thing the agent can do. Note that R_2 (the "challenge model") is only used to improve performance of the MCTS solver, and is ignored in the final model evaluation; see section III-F.

that's entered after taking an action that results in a collision with an obstacle.

Despite that the agent and the obstacle are driving within one lane of traffic, we still model the lateral movement within the lane using two-dimensional continuous coordinate space.

Choosing the front right and the rear left corners to represent the state of the agent and of the obstacle respectively allows for easier collision detection algorithm.

C. Action Space

The action space, denoted $\mathcal{A}$, is a tuple that consists of the following elements:

- "maintain" the current heading and velocity
- "accelerate" and "brake" set $a_{t+1}^x \leftarrow \pm\alpha$ respectively where α is constant.
- "slide left" and "slide right" represent taking and recovering from an evasive maneuver and respectively change d_{t+1}^y to either $\delta = 0.5$ or back to 0.

The obstacle is not an agent and does not take actions. However, in the simulation and evaluation code, the subroutines to emulate the agent movement is reused to represent the obstacle's movement.

D. Transition Model

In this work, most actions succeed as we focus on modeling observational uncertainty. The "accelerate" and "brake" actions change the acceleration at the next time step with 100% certainty, which propagates to velocity v^{xy} and d^{xy} as per laws

of motion without friction. Time discretization step $\Delta t = 0.1$ is used.

As an exception to that, "slide left" action has only 25% chance to succeed, modeling the potential obstruction in the left lane:

$$T(d^x_{t+1} = \delta | d^x_t = 0, \text{slide right}) = 0.25$$

The obstacle transitions to the next state according to a scenario. The scenarios used to model the roadside situations are described in section V.

E. Observation Model

For simplicity, we assume full perfect observability of b^y_t, and only allow uncertainty in one dimension.

The agent observes the state of the obstacle with a certain degree of accuracy σ and precision ρ. The accuracy σ models a noisy sensor return:

$$P(o^y_t | b^y_t) = \mathcal{N}(b^y_t, \sigma) \tag{1}$$

The returns are then also discretized. Not only this models the actual sensor behavior which has limited precision, discretizing observation space is **critical** for the POMCPOW algorithm to produce results different from random policy, as mentioned in Section 4.4 in [6].

If we denote the sensor precision as ρ, we modify o^y_t as

$$o^y_t \leftarrow \lfloor o^y_t \cdot \rho \rfloor / \rho \tag{2}$$

This presented a problem explored in section VI-C1, and we recommend modifying this algorithm for further work as described in section VII.

For the remainder of this project, we thus derive the probability weighting function for POMCPOW from (1) and (2):

$$Z(o^y_t | b^y_t, a) = \lfloor \mathcal{N}(b^y_t, \sigma) \Delta t \cdot \rho \rfloor / \rho \tag{3}$$

F. Reward Model

In this project, we use two reward models. One reward model (R_1, we call it "benchmark model") is used for evaluation and defines good driving. However, the solvers used were unable to attain good results on that model directly. We augment this reward model and use model R_2 ("challenge model") when applying POMCPOW.

In the **benchmark model**, a large positive reward ($R_M = +10000$) is given in step t for transitioning into the end state $d^x_t > X$. A small negative reward is applied at every time step to encourage progress (-1), a moderate negative reward for lateral moves or for driving not in the lane (-2000 for each timestem when $d^y \neq 0$), and a large negative reward (-10^9) for collision with the obstacle.

In the **challenge model**, the large positive reward for mission completion R_M is "spread out" across the state space, and we reward high velocity

$$r_d(S, A, S') = R_M \frac{d'^x - d^x}{2X} \tag{4}$$

$$r_v(S, A, S') = \begin{cases} R_V ||v'||_2^\nu & \text{if } ||v'|| < \text{Speed Limit} \\ -R_{\text{Speeding}} & \text{otherwise} \end{cases} \tag{5}$$

$$R(S, A, S') = R_1 + r_d + r_v \tag{6}$$

The motivation for introducing r_d is the limitation of depth-bounded online tree search methods. If the large reward is invisible when the rollout depth is smaller than $\frac{2X}{\Delta t}$. This equals to 300 in our simulations, and running an online solver with this depth is intractable.

G. Prior information

Since POMCPOW starts with a belief, it is important to provide a prior belief that is consistent with the observation. That is, instead of a uniform (uninformed) prior, the observation needs to contain an informed prior. In order to initially seed our particle filter, we sample o^y_0 uniformly. from $[-3\sigma; +3\sigma]$

H. Belief update

The model uses Particle Filter without Rejection as a way to maintain belief state with $N = 10$ particles.

Since the obstacle might move in the next step, and we are not sure if it will, we incorporate this uncertainty into the model as well. At every step, we replace αN particles with new observation while we keep the remaining $(1 - \alpha)N$ particles from the previous step. This is inspired by Feynman-Kac particle filter [4] but does not strictly follow the method.

IV. Policy Solver

Since the state space is continuous and not discretized, we use a Monte-Carlo tree search-based algorithm, POMCPOW [6]. It is a fixed-depth tree search algorithm that uses Double Progressive Widening to explore continouos actions space, and observation binning to constrain the continuous observation space.

The algorithm allows incorporating the prior information, and the default priors (e.g. random rollouts) were not sufficient to achieve high rewards. The default "random" rollout estimate erased the difference between the effects of the actions explored close to the root of the tree. We first disabled rollouts and tried using 0 as expected reward instead. This alone was insufficient as well: the policy would instruct the car to accelerate (optimizing R_2) even when an obstacle is impeding.

Instead, we considered a policy "always brake" as the rollout policy; this produced policies that stopped before impeding obstacles. However, the policy

We addressed this problem by using **two rollout policies**: always brake and always maintain, comparing the expected rewards, and choosing the action produced by the highest policy.

Action	POMCPOW count	POMCPOW %	Optimal count	Optimal %
slide_right	57	**9.9**	0	0
slide_left	15	**2.6**	0	0
brake	8	**1.4**	0	0
maintain	51	**8.8**	3	6.25
accelerate	447	69.69	45	**93.75**

Fig. 3. Scenario 1a (driving without obstacles): count and percentage of actions taken by our and optimal policy across 10 runs. Larger percentage value is highlighted in bold. Optimal policy is to accelerate until the speed limit is reached (which is almost at the end of the 40-meter wide world) and maintain speed.

Action	Scenario 1a count	Scenario 1a %	Scenario 1b count	Scenario 1b %
slide_right	57.0	9.9	81.0	**10.8**
slide_left	15.0	2.6	45.0	**6.0**
brake	8.0	1.4	90.0	**12.0**
maintain	51.0	8.8	88.0	**11.7**
accelerate	447.0	**77.3**	448.0	59.6

Fig. 4. Scenario 1b and 1a comparison: count and percentage of actions taken across 10 runs by the POMCPOW algorithm. Larger percentage value is highlighted in bold.

V. Evaluation

We've evaluated the performance of the driving code on multiple scenarios, aiming to achieve the satisfactory performance on the simple ones before moving onto the more complex. We additionally conducted the study of sensor accuracy on the rewards the policy is available to collect.

The scenarios used for evaluation are:

- **Scenatio 1a: Driving in Straight Line**: $X = 20$, obstaclre is visibly off the road, $\sigma = 0$.
- **Scenatio 1b: Driving around stationary obstacle**: $X = 20$, obstacle is visibly on the road, but is passable but evasive maneuver is required ($d_t^y = 0.2$) $\sigma = 0.1$.
- **Scenario 2: Stop Before Obstacle**: the obstacle is blocking the road and is impassible ($d_t^y = 1.0$); the agent needs to stop in front of it and stay.
- **Scenario 3: Accuracy Study**: the obstacle is randomly placed $d_t^y \sim [0; 1]$ and the sensor accuracy is given by

$$\sigma = \begin{cases} \beta\sigma_0 & \text{when } |d^x| > 5 \\ \sigma_0 & \text{when } |d^x| \leq 5 \end{cases} \tag{7}$$

  The reward produced by the policy on the evaluation model is compared for $\beta \in 1, 2, 5, 10$.

In all scenarios, the obstacle is at 0 on the x axis.

The following parameters were used for POMCPOW: max_depth 20, time limit 60s per solver run (however, the average time was only 1.67 seconds on a Core i9-9700k CPU), and $c = 10$ for the upper confidence bound.

The code is available at https://github.com/pshved/aa228.

VI. Analysis

Before analyzing the behavior of specific scenarios,

MCTS due to random sampling policy doesn't select the best action. For example, the best action is Accelerate, but since any trace would have an equal mix of all actions, MCTS doesn't see the difference.

(!) Plot velocity over time for an obstacle and compare it with velocity over time with no obstacle on one plot.

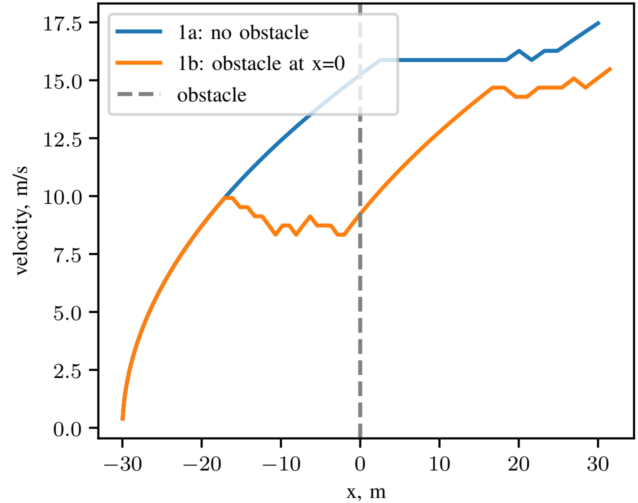

Fig. 5. Scenario 1b and 1a comparison: velocity plotted against the agent position on the x axis for 1 random run for each of the two scenarios.

A. Scenario 1: Driving in Straight Line

The evaluation results for Scenario 1a are presented on 3. We see that while our solution does eventually accelerate, POMCPOW also "produces" a variety of other actions. The larger impact on that is the relative indifference which action to take at the end of the simulation: when there is no obstacle between the agent and the reward, all actions seem similar (see Fig.5). There's also some exploration affecting the choice of actions at the earlier stages.

In scenario 1b, optimal policy is harder to deduce, so we compare the behavior of the agents across scenarios 1a and 1b (see Figure 4). More telling are velocity patterns (see Figure 5): we can indeed see that the agent learns to approach the obstacle more carefully to ensure (a) that it can swerve around; (b) to better localize it.

Note how the behavior towards the end of the simulation exposes a problem with our challenge policy: the penalty for speeding is not large enough so speeding to complete the mission faster is indeed optimal.

Depth	Avg. reward	Avg. time (s)
10	-95454555	1.03
20	-362	1.60
40	-301	2.79
80	-282	3.09

Fig. 6. Scenario 2: comparison of average reward over 300 steps as function of the exploration depth when the obstacle is reliably and observably blocking the path. Note that depth of -10 (unlike larger depths) is not sufficient to avoid collisions.

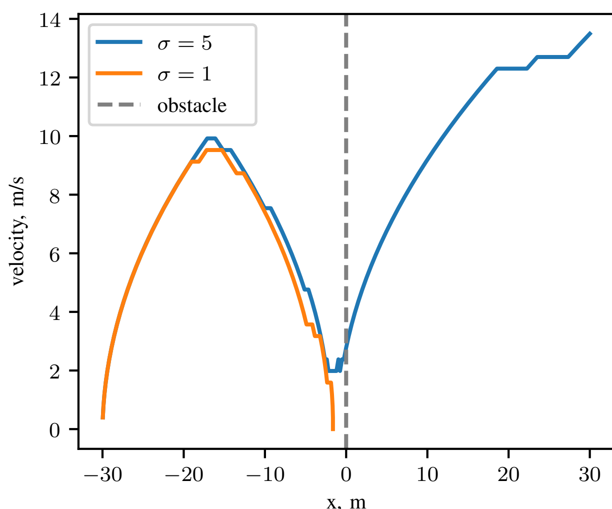

Fig. 7. Scenario 3 comparison of two runs with better accuracy (corresponds to $\sigma = 1$) and worse accuracy (corresponds to $\sigma = 5$) with the obstacle at $b^y = 0.32$ in both graphs. We can see that the agents get often confused about the position of the obstacle, and better a5ccuracy doesn't result in better policy. The line for $\sigma = 1$ also serves as an accurate representation of Scenario 2 behavior.

B. Scenario 2

In scenario 2, we found, unsurprisingly, that if the depth of the tree search is not enough to explore the state space until the impeding obstacle, the collisions are imminent. Thanks to the "always brake" rollout policy, even the depth of 20 was able to stop the car in front of an obstacle despite that 25 steps are required.

We found also that (a) the average time to compute the next step does **not** grow exponentially, and (b) that the returns quickly diminish with the growth of depth. We've conducted all our other experiments with depth 20.

C. Scenario 3

In Scenario 3, we aimed to study how the driving behavior degrades when sensor

We found out, however, that the simulation provided in this paper is not sufficiently accurate to perform this study. Figure 7 demonstrates that the agent fails to drive around an obstacle when the sensor accuracy is better, but succeeds when it's worse. One explanation could be that the problem statement or the solver do not exhibit sufficient performance. Another explanation would be that the more sensor produces such outlandish observations that the agent gets through by pure luck. We leave the detailed exploration to future work.

1) Particle Filter and Precision Failure Mode: During experiments in Scenario 3, we found a problem worth noting with (2) when particle filters are used. Assume $b_t^x = 0.55$; then $o_t^y = 0.5$. When sampling successor states from the generative model starting from o_t^y without adding noise, it would be feasible to drive past the obstacle after taking action "slide left", while in reality this results in a collision.

Modifying observation model such that it assumes the obstacles are strictly larger then they are would mitigate that.

VII. Future Work

One of the things to explore could be a more rigorous approach to particle filtering that still combines the previous observations with the current. We also can incorporate more rollout policies into the ensemble policy.

VIII. Conclusion

We applied online Monte-Carlo tree search to building policy for autonomous agents emulating driving a self-driving car around the obstacle. Specifically, we applied POMCPOW [6], a variation of Monte-Carlo Tree Search for continouos action and observatino spaces.

This method produces viable results in simple cases even when state space discretization is not used (only with time discretization). However we found that a carefully crafted rollout policy and spreading the rewards across the world are required to make that mehtod to work.

Other drawbacks of the method include its inability to learn from prior experience and produce an offline policy (unlike policy gradient methods or Q-learning) which results in unnecessary computation in simple situations.

References

[1] Claudine Badue, Ranik Guidolini, Raphael Vivacqua Carneiro, Pedro Azevedo, Vinicius Brito Cardoso, Avelino Forechi, Luan Ferreira Reis Jesus, Rodrigo Ferreira Berriel, Thiago Meireles Paixão, Filipe Wall Mutz, Thiago Oliveira-Santos, and Alberto Ferreira de Souza. Self-driving cars: A survey. *CoRR*, abs/1901.04407, 2019.

[2] Rui Fan, Jianhao Jiao, Haoyang Ye, Yang Yu, Ioannis Pitas, and Ming Liu. Key ingredients of self-driving cars. *CoRR*, abs/1906.02939, 2019.

[3] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[4] P Del Moral. An introduction to feynman-kac particle methods in statistical learning and rare event simulation, 2014. http://people.bordeaux.inria.fr/pierre.delmoral/Sydney-Uni.March-28-2014.pdf.

[5] Alexandru Constantin Serban, Erik Poll, and Joost Visser. A standard driven software architecture for fully autonomous vehicles. In *ICSA Companion*, pages 120–127. IEEE Computer Society, 2018.

[6] Zachary Sunberg and Mykel J. Kochenderfer. POMCPOW: an online algorithm for pomdps with continuous state, action, and observation spaces. *CoRR*, abs/1709.06196, 2017.

[7] R. E. Tompa, B. Wulfe, M. P. Owen, and M. J. Kochenderfer. Collision avoidance for unmanned aircraft using coordination tables. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–9, Sep. 2016.

[8] Rachael Tompa and Mykel Kochenderfer. Optimal aircraft rerouting during space launches using adaptive spatial discretization. pages 1–7, 09 2018.

[9] Wenshuo Wang and Ding Zhao. Extracting traffic primitives directly from naturalistically logged data for self-driving applications. *CoRR*, abs/1709.03553, 2017.