AA228 — Optional Final Project: Escape Roomba

Gael Colas[1] and Victor Zhang[2]

I. INTRODUCTION

The goal of this project is to design a policy to navigate a Roomba (vacuum cleaner robot) safely out of a room. This problem can be modeled as a Partially Observed Markov Decision Process (POMDP).

II. PROBLEM STATEMENT

The environment is a two-dimensional L-shaped room with one exit (shown in green on Figure 1), and one staircase (shown in red on Figure 1). There are three different room configurations (i.e. with different exit and staircase locations).

The Roomba knows which room it is in (both the shape and the configuration of the room). However it has initially no idea where it is in the room: it is under state uncertainty. It can only interact with its environment using a bumper sensor that indicates when it is in contact with a wall.

The POMDP framework of this problem is as follows:

- **State space** $\mathcal{S} = \{s = (x, y, \theta, \text{status})\}$ where (x, y) denotes the position of the Roomba in the room and θ its heading. The variable "status" indicates whether the Roomba has reached the door or the staircase;
- **Action space** $\mathcal{A} = \{a = (v, \omega)\}$ where v and ω are respectively the velocity and the angular speed of the Roomba;
- **Observation space** $\mathcal{O} = \{o \in \{0, 1\}\}$ where $o = 1$ if the Roomba is in contact with a wall, $o = 0$ otherwise.

The Roomba can navigate the room for $N = 100$ time steps, where the time discretization is: $dt = 0.5s$. Both the door and the staircase are terminal states.

The transition function is based on the discretized dynamics of the problem while taking into account the geometry of the room. However, it is not a purely deterministic function of (s, a) as there is some noise on the actual command (v, ω) applied to the Roomba.

The reward model is:

- each elapsed time-step yields $r = -0.1$;
- wall collision yields $r = -1$;
- staircase collision yields $r = -10$;
- exit collision yields $r = +10$.

Thus, the Roomba must balance the need to collect information about its current position against the goal of safely and efficiently navigating its way out of the room.

[1]Gael Colas is with Stanford University, Department of Aeronautics and Astronautics colasg@stanford.edu

[2]Victor Zhang is with Stanford University, Department of Mechanical Engineering zhangvwk@stanford.edu

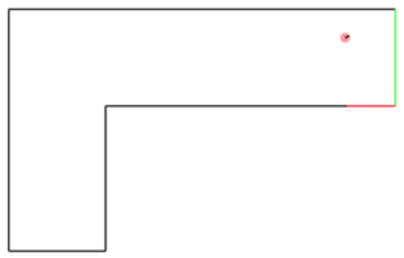

Fig. 1. Roomba environment in room configuration 1

III. APPROACH

A. Baseline

We implemented a heuristic method based on the following observation: if the Roomba turns right and follows the wall every time it bumps into one, the L-shape geometry of the environment makes it converge to the exit from any initial condition.

The first step is to estimate the initial heading of the Roomba to be able to make the first turn. Let α be the true angle between the Roomba's initial direction and the wall it bumps into when it moves straight. A strategy to estimate α is to make the Roomba bump into the same wall twice: first, it moves from its initial heading; then, after bumping into a wall, it rotates by ϕ counterclockwise and moves straight for a distance d_1 that is estimated. Then, it rotates by $\frac{\pi}{2}$ clockwise and moves straight until it bumps into the same wall again after a distance d_2 that is estimated. Finally, α can be estimated using the following relation: $\alpha = \arctan \frac{d_1}{d_2}$, as shown on Figure 2.

We can use this estimate of α to align the Roomba with the wall by rotating it clockwise of $\alpha + \pi$. Then the heuristic depicted above is applied until the Roomba reaches the exit.

Note that there are the following trade-offs to tune:

- angle ϕ needs to be more than $\pi/2$ so that the distance d_2 is long enough to get a better precision on α;
- distance d_1 cannot be too long as there would be a higher chance that the Roomba bumps into a different wall. In this case no useful information can be inferred.

Furthermore, there exist situations in which hitting the same wall twice would require the Roomba to rotate by ϕ clockwise instead of counterclockwise. Nevertheless, it was found that this baseline approach yields reasonable results.

B. Offline method

Prior to solving the POMDP, the state and action spaces were discretized as follows:

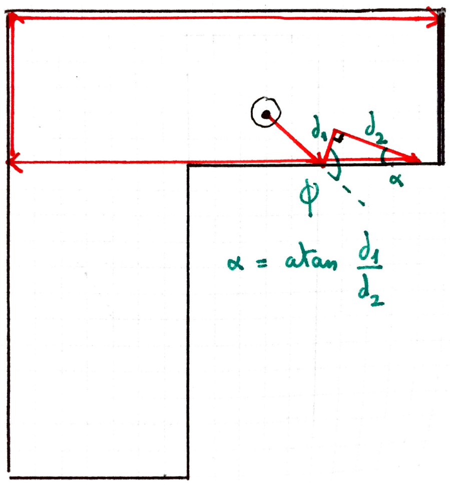

Fig. 2. Heading estimation

- 50 possible x and y coordinates;
- 20 possible θ values;
- 3 possible velocity v and angular speed ω values;

for a total of $3 \cdot 50^2 = 150 \cdot 10^3$ states (as status $\in \{-1, 0, 1\}$) and $3^2 = 9$ actions.

The QMDP method [1] was used to approximate the set of alpha vectors. One alpha vector per action in the discretized action space was initialized: $\alpha_a^{(0)}(s) = 0$ for all $s \in \mathcal{S}$. The update rule is:

$$\alpha_a^{(k+1)}(s) = R(s,a) + \sum_{s'} T(s'|s,a) \max_{a'} \alpha_{a'}^{(k)}(s') \quad (1)$$

We used the POMDPs package [2] and ran the QMDP solver with a maximum number of iterations of 20 and a Bellman tolerance of 10^{-3} to obtain a set of 9 alpha vectors. Using QMDP alone unsurprisingly yields suboptimal results as it only provides an upper bound on the optimal value function.

C. *Online methods*

The upper bound precomputed with QMDP can be used in conjunction with online methods to get a more accurate approximation of the value function and thus a better policy. We experimented with two different methods: Branch and Bound (B&B) and Monte Carlo Tree Search (MCTS).

1) Branch and Bound: The Branch and Bound method [1] is an amelioration of Forward Search because it uses an upper bound (in our case precomputed with QMDP (see III. B.)) to prune actions that can not be optimal. The lower bound we used is the value function associated with a blind policy that selects the same action regardless of the current belief state. The lower bound can be computed using Equation 2. The alpha vectors are initialized with $\alpha_a^{(0)}(s) = \frac{\min_s R(s,a)}{1-\gamma}$.

$$\alpha_a^{(k+1)}(s) = R(s,a) + \sum_{s'} T(s'|s,a) \alpha_a^{(k)}(s') \quad (2)$$

Algorithm 1 Mixture Policy

```
1: function SELECTACTION((b, d))
2:     $\sigma_{\text{pos}} \leftarrow \hat{\sigma}(b)$
3:     if $\sigma_{\text{pos}} > \sigma_{\text{threshold}}$ or inContactWall($b$) then $a \leftarrow$ MCTS($b, d$)
4:     else                                   ▷ switch to controller
5:         $s \leftarrow$ mean($b$)
6:         $\phi_{\text{goal}} \leftarrow \arctan \frac{y_{\text{goal}} - y_s}{x_{\text{goal}} - x_s}$
7:         $\phi_{\text{target}} \leftarrow \phi_{\text{goal}} - \theta_s$
8:         $\omega \leftarrow K_{\text{prop}} \cdot \phi_{\text{target}}$
9:         $a \leftarrow (v_{\text{ev}}, \omega)$
10:    end if
11:    return $a$
12: end function
```

2) Monte Carlo Tree Search: Monte Carlo Tree Search [1] is one of the most successful sampling-based online approaches. It runs simulations from the current belief state to find the best action up to a maximum depth. Each simulation is used to build a History Tree. It updates estimates of the history-action value function $Q(h,a)$ and counts $N(h,a)$ for each sequence h of past observations and actions.

If the simulation reaches a sequence h not currently in the tree, a rollout policy is used to estimate the value function of this new sequence. We compared a random rollout policy with a more reasonable one that uses the QMDP alpha vectors.

We used the POMCP package [2] and ran the MCTS solver with a maximum depth of 20 and a 1000 iterations per action selection.

D. *Controller*

If the Roomba knows exactly its position s, a faster method to navigate to the goal position $(x_{\text{goal}}, y_{\text{goal}})$ is to use a proportional controller with gain K_{prop}.

This Controller uses a fixed evolution speed v_{ev} and computes the angular speed using the relations given in Equation (3).

$$\begin{cases} \phi_{\text{goal}} & = \arctan \frac{y_{\text{goal}} - y_s}{x_{\text{goal}} - x_s} \\ \phi_{\text{target}} & = \phi_{\text{goal}} - \theta_s \\ \omega & = K_{\text{prop}} \cdot \phi_{\text{target}} \end{cases} \quad (3)$$

Two major issues don't allow to only use a proportional controller to navigate the Roomba:

1) At the beginning, the Roomba does not know where it is in the room. The controller does not have a reasonable estimate of its position to compute the right action;
2) The controller does not take the walls into account while computing the action. If there is a wall on the trajectory, then the Roomba will get stuck against that wall until the end of the simulation.

E. *Final "Mixture" method*

In our final method, we leveraged the best of both worlds:

TABLE I
HYPERPARAMETERS OF THE "MIXTURE" METHOD

Hyperparameters		Value
QMDP	Maximum number of iterations	20
	Bellman tolerance	10^{-3}
MCTS	Maximum depth	20
	Number of iterations to select each action	1000
Controller	Proportional gain K_{prop}	2.0
	Evolution velocity v_{ev}	5 m/s
	Standard deviation threshold $\sigma_{\text{threshold}}$	0.5

- QMDP alpha vectors are computed offline to provide an upper bound on the value function;
- Online: the MCTS method computes the best action for a belief b up to a maximum depth d. Precomputed QMDP alpha vectors are used as a reasonable rollout policy;
- Online: when Roomba becomes "sufficiently confident" about its position in the room, the controller uses the mean position to navigate the Roomba more quickly towards the goal.

The criterion "sufficiently confident" is estimated with $\hat{\sigma}_{\text{pos}}$, the normalized position standard deviation of the filter's particle. $\hat{\sigma}_{\text{pos}}$ is a three-dimensional vector composed of: $\hat{\sigma}_x(b)$, $\hat{\sigma}_y(b)$ and $\hat{\sigma}_\theta(b)$, which are the normalized standard deviations associated with x, y and θ respectively. The Roomba's confidence about its position is the sum of all three terms and this sum is then compared to a threshold value $\sigma_{\text{threshold}}$.

Before switching to the controller to navigate to the goal, the Roomba checks that it is not currently in contact with a wall. As the controller does not take the walls into consideration when computing the action, this heuristic avoids getting stuck against a wall.

Our method is outlined in more detail in Algorithm 1.

The hyperparameters of the "Mixture" method were tuned (see Table I) over development seeds in room configuration 1 to provide the best results.

IV. RESULTS

To compare our policies, we used the following metrics:

- **Mean Total Return (TR)** where one total return is the sum of the individual rewards accumulated during a simulation. This metric indicates how well the policy does on average.
- **Standard Deviation of the Total Return** which is the standard deviation of TR. This metric indicates whether or not the policy consistently performs well.

For consistency, we averaged the Total Return over 50 test seed simulations from different Roomba starting positions in room configuration 1, which is the trickiest one as the exit and the staircase are directly adjacent (as on Figure 1).

Apart from the baseline, we used discretized state and action spaces to test our online methods. Table II summarizes our simulation parameters.

TABLE II
SIMULATION PARAMETERS

Parameters		Value
Room configuration		1
Number of simulations		50
Number of time steps		100
Time step		0.5 s
Discretization	State	$(n_x = 50, n_y = 50, n_\theta = 20)$
	Action	$(v, \omega) \in \{0; 5; 10\} \times \{-1; 0; 1\}$
Action noise coefficients	v	2.0
	ω	0.5
Number of particles		5000

TABLE III
COMPARISON OF TOTAL RETURNS OVER THE SIMULATIONS

Method	Best index	Mean	Standard deviation
Lidar	14.0 %	-4.00	9.31
Baseline	8.0 %	-8.11	9.97
B&B + QMDP	14.0 %	-4.80	9.54
MCTS Vanilla	16.0 %	-6.82	8.26
MCTS + QMDP	20.0 %	-0.14	6.68
"Mixture"	28.0 %	2.32	5.27

Table III draws a comparison between "Lidar" (the baseline method using a Lidar sensor originally given by the course staff), "Baseline" (our own baseline method described in III. A.), "B&B + QMDP" (the branch and bound method using QMDP as described in III. C. 1)), "MCTS Vanilla" (Monte Carlo Tree Search with a random rollout policy), "MCTS + QMDP" (Monte Carlo Tree Search with a rollout policy using QMDP alpha vectors as described in III. C. 2)), and "Mixture" (as described in III. E.).

We can see that "Mixture" yields the highest best index, which is the percentage of best total returns. Furthermore, "Mixture" is the only method yielding a positive mean total return: mean(TR) $= 2.32$. It is the also most consistent one with the smallest standard deviation: $\sigma(\text{TR}) = 5.27$.

Figure IV sums up graphically Table III and shows further interesting results: not only does "Mixture" (shown in brown) outperforms MCTS + QMDP (shown in red), it also resolves several outlier cases observed with the MCTS + QMDP method. This shows that adding our controller increases the robustness of MCTS + QMDP.

Table IV provides more insightful indicators to evaluate the best method "Mixture" and which can be used to compare it with the best policies found by other groups.

TABLE IV
FINAL "MIXTURE" METHOD PERFORMANCES

Indicator	Value
Average time used	33.0 s
Average bumps	3.7
End status	Reaches goal 96% / Gets stuck 0% / Dies 4%

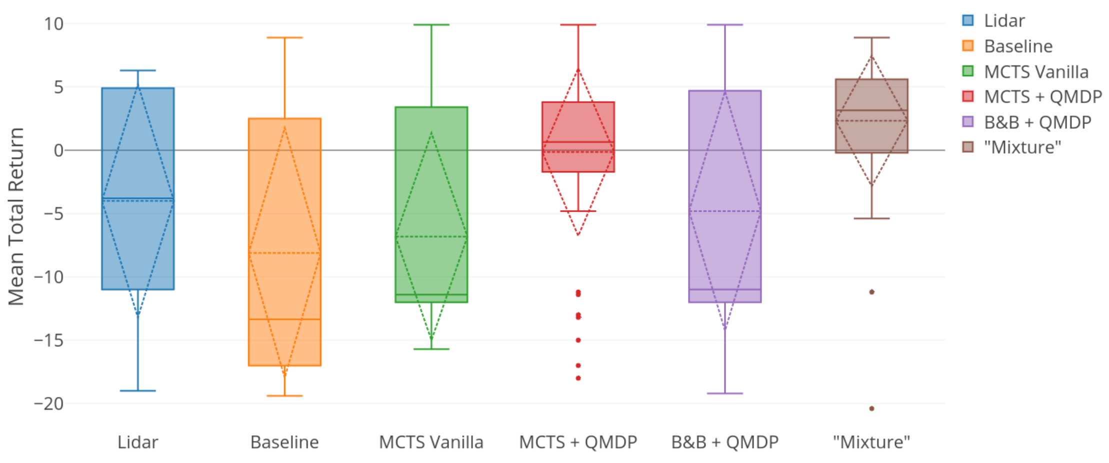

Fig. 3. Box and whisker plots of the Mean Total Returns for the different methods

V. CONCLUSION

Navigating a Roomba inside a room using only a bumper sensor is not a trivial problem, even if the map is known in advance. The bumper sensor does not provide the Roomba with a lot of information about its environment.

Our “Mixture” method works well in this environment because it leverages the advantages of two different approaches. First, it uses Monte Carlo Tree Search to balance exploitation versus safe exploration. The exploration is considered to be safe as it uses its current belief to avoid the high penalty of falling into the stairs. Starting off MCTS with a reasonable rollout policy based on QMDP alpha vectors helps to get a good policy while keeping the depth relatively small.

Then, it switches to a Controller when the Roomba is confident enough about its position. This helps navigating more efficiently to the goal in the last steps: less bumps and time steps to reach the goal.

All our code for this project can be found here [4].

REFERENCES

[1] Mykel Kochenderfer, “Decision Making Under Uncertainty, Theory and Application”, *MIT Press*, 2015.
[2] POMDPs and POMCP Julia packages, https://github.com/JuliaPOMDP
[3] Roomba User Manual https://store.irobot.com/default/robot-vacuum-roomba/
[4] Project GitHub repository https://github.com/ColasGael/AA228FinalProject/tree/masterendthebibub rliography