solution

April 13, 2024

0.0.1 1.a

For fully observable navigation, the state space should include all the necessary information for the agent to make decisions. This includes position and orientation. So the state space is a tuple such that:

$statespace = (x_i, y_j, o_k)$ for all i, j, k

where:

- x_i is x position in range 10
- y_j is y position in range 20
- o_k is orientation in where $o_k \epsilon [up, right, left, down]$

For fully observable navigation, the action space should be the movements that only the agent to traverse the grid. So the action space combination of actions the agent can take.

$actionspace = [' turn_{right}, turn_{left}, move_{forward}, move_{backward}]$

0.0.2 1.b

The simulator has been implemented. Below are links to the simulator/envirinment implementation:

- **Fully Observable Environment**

0.0.3 1.c

The Q learner has been implmented and can be found here:

- **Q Learning Agent**

It runs alongside the POMDP agents. You can run the agents with default parameters by typing in : `python -m agents`

However, you can run the program with different paramaters like the given example below:

```
python -m agents --start_state "(1, 1, 'UP')" --episode_num 25 --episode_length
1700 --alpha 0.01 --gamma 1.0 --epsilon 0.35 --num_obstacles 5 --trials 4
```

Once the program finishes, a logfile.txt is provided to check completion time, and the graphs are saved that show the epsiode v rewards.

0.0.4 Results

I ran the program with the following paramaters:

2024-04-13 16:14:50: {'start_state': "(1, 1, 'UP')", 'episode_num': 25, 'episode_length': 2000, 'alpha': 0.1, 'gamma': 1.0, 'epsilon': 0.3, 'num_obstacles': 5, 'trials': 4}

There were four trials with randomized goals and obstacles:

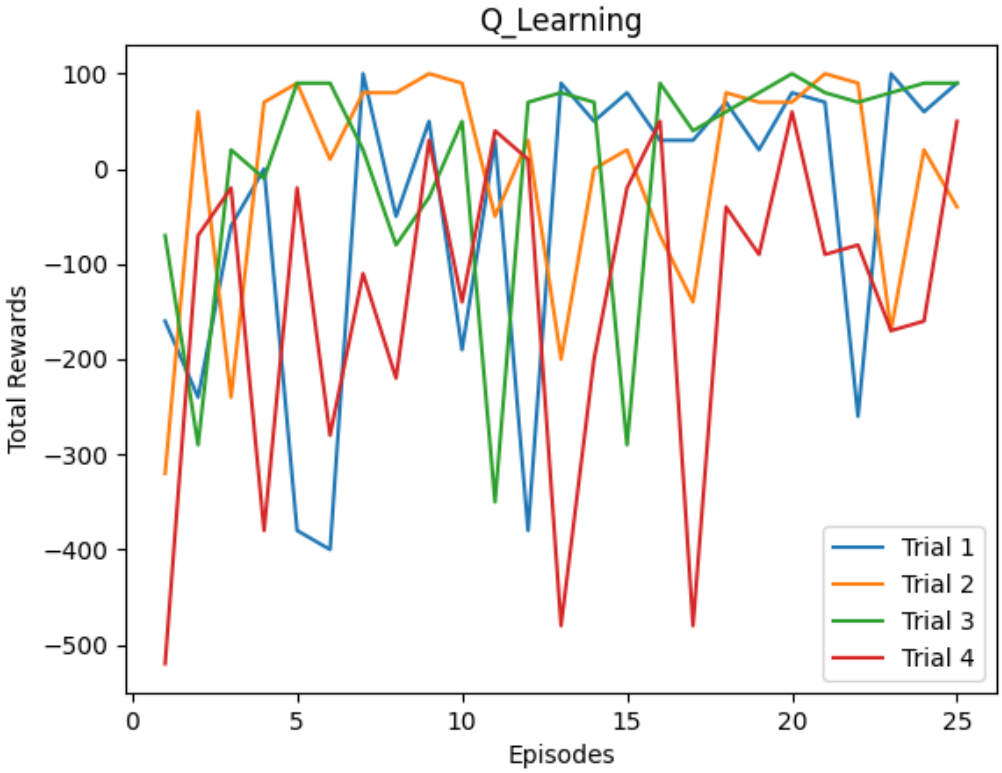
TRIAL 1 - OBSTACLES:[(8, 14), (6, 14), (5, 1), (1, 17), (2, 5)] - GOAL:(7, 4)

TRIAL 2 - OBSTACLES:[(2, 14), (10, 4), (10, 19), (7, 5), (10, 2)] - GOAL:(6, 5)

TRIAL 3 - OBSTACLES:[(2, 17), (6, 20), (1, 10), (3, 12), (5, 9)] - GOAL:(6, 2)

TRIAL 4 - OBSTACLES:[(4, 10), (6, 4), (2, 3), (4, 11), (10, 2)] - GOAL:(9, 6)

Below is the graph for the **Q Learner** I ran and the respective 4 different trials:



Each trial performed as expected. Trial 3 did the best overall likely because the goal (6,2) was the closest to the start state. Trial 4 did the worst likely because it was the furthest (9,6) from the start state. All trials evetually performed better over the runtime of the program.

- 2024-04-13 16:15:00: Trial: 1. Finished Q Learning. Goal finding efficiency: 0.88
- 2024-04-13 16:17:24: Trial: 2. Finished Q Learning. Goal finding efficiency: 0.96
- 2024-04-13 16:21:34: Trial: 3. Finished Q Learning. Goal finding efficiency: 0.88
- 2024-04-13 16:25:09: Trial: 4. Finished Q Learning. Goal finding efficiency: 0.8

0.0.5 2.a

Now we have an POMDP. The agent doesnt have access to the true state of the environment. It only knows if has hit a wall, obstacle, goal, or none of the options. So to design an observation fucntion, we need to design it in the environemnt. This **observation function** will 'filter' the true state and return the what the agent sees.

$observation = Observation_{func}(state)$

In the below code, the environemnt has the true state and returns the `observed_state` to the agent. In practice, this should mimic how an agent would only obtain certain infromation from its sensors. Here is the **observation function**

```
    def observation_function(self, next_state):

        x_2, y_2, orien_2 = next_state
        observed_state = None

        # Return observation from agent's view, given the actual state
        if (x_2, y_2)in self.obstacles:
            observed_state = HIT_OBSTACLE

        elif (x_2,y_2) == self.goal_state:
            observed_state = HIT_GOAL

        elif x_2 > X_DIRECTION or y_2 > Y_DIRECTION or x_2 < 1 or  y_2 < 1:
            observed_state = HIT_WALL

        else:
            observed_state = HIT_NONE

        return observed_state
```

On top of the observation function, we need an observation probability function to obtain O, or the set opf conditional observation probabilities. Since we assumed the environment was deterministic, action a has a 100% of leading to state s' from state s, our **observation probability function** becomess much more simple; there is only one possible observation for every state-action pair.

So out **observation probability** function becomes:

```
def observation_probs(self, state, observation, action):
    chance_correct = 1
    chance_wrong = 1 - chance_correct
    expected = self.observation_function(state)

    if observation == expected:
        return chance_correct
    else:
        return chance_wrong
```

Later on, when we caluclate belief updates, this simplifies our calculations since traditionally the

belief update is:

$b'(s') = nO(o|s', a) \sum T(s'|s, a)b(s)$

But since the environment is deterministic, the probability that an observation matches the expected outcome given a state and action is effectively 1 and, conversely, the probability of any other observation is 0. Now making the **belief update**:

$b'(s') = n \sum T(s'|s, a)b(s)$

Here is the link to my functions:

Observations

0.0.6 2.b

The code has been expanded to handle observations and we added some new functions to do this:

```
def observation_probs(self, state, observation, action):
def observation_function(self, next_state):
def init_belief(start_state):
def update_belief(observation, agent):
def compute_Q_b_a(action, agent):
```

0.0.7 2.c

0.0.8 Results

I ran the program with the following paramaters:

2024-04-13 16:14:50: {'start_state': "(1, 1, 'UP')", 'episode_num': 25, 'episode_length': 2000, 'alpha': 0.1, 'gamma': 1.0, 'epsilon': 0.3, 'num_obstacles': 5, 'trials': 4}

There were four trials with randomized goals and obstacles:

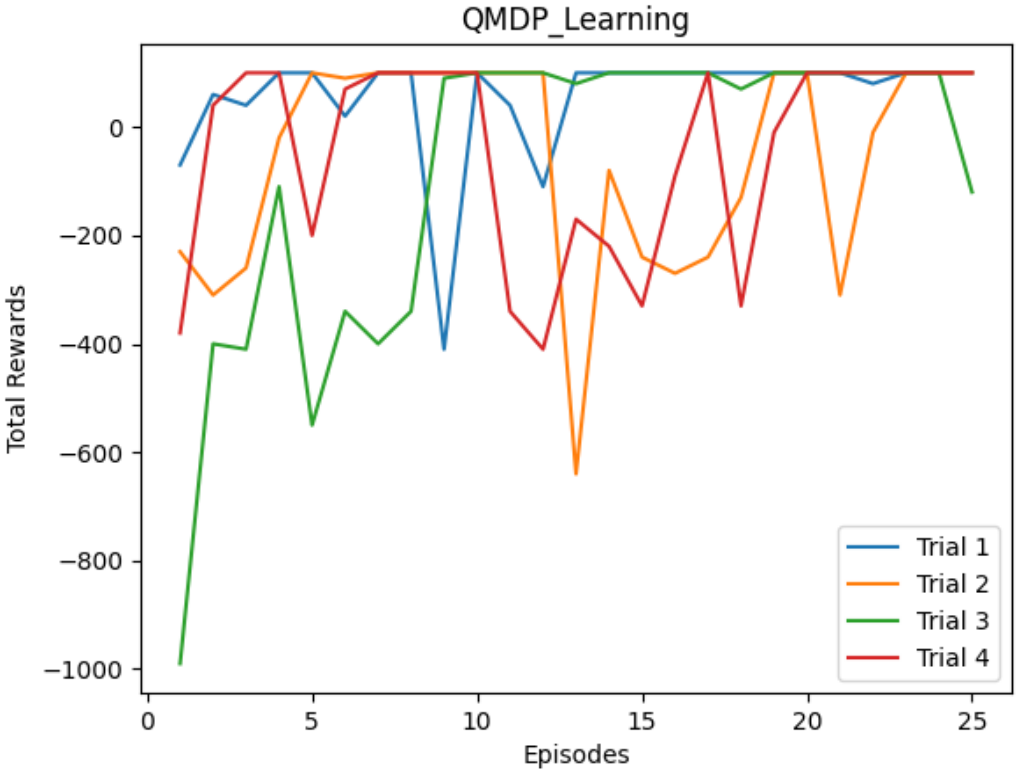
TRIAL 1 - OBSTACLES:[(8, 14), (6, 14), (5, 1), (1, 17), (2, 5)] - GOAL:(7, 4)

TRIAL 2 - OBSTACLES:[(2, 14), (10, 4), (10, 19), (7, 5), (10, 2)] - GOAL:(6, 5)

TRIAL 3 - OBSTACLES:[(2, 17), (6, 20), (1, 10), (3, 12), (5, 9)] - GOAL:(6, 2)

TRIAL 4 - OBSTACLES:[(4, 10), (6, 4), (2, 3), (4, 11), (10, 2)] - GOAL:(9, 6)

Below is the graph for the Q_{MDP}. I ran and the respective 4 different trials:



The Q_{MDP}, despite not being a full POMDP because its a hybrid, did relatively well at aproximating the POMPD values. Unlike the Q original learner in the fully observable mdp, the Q_{MDP} has less variance in the graph. It was also much better at learning and had several epsiodes in a row that near perfect 100 reward. This is likely because it had access to the previous MDP's $q(s, a)$ paramaters

- 2024-04-13 16:17:17: Trial: 1. Finished Starting QMDP Learning. Goal finding efficiency: 1.0
- 2024-04-13 16:21:27: Trial: 2. Finished Starting QMDP Learning. Goal finding efficiency: 0.64
- 2024-04-13 16:24:54: Trial: 3. Finished Starting QMDP Learning. Goal finding efficiency: 0.68
- 2024-04-13 16:30:49: Trial: 4. Finished Starting QMDP Learning. Goal finding efficiency: 0.72

0.0.9 2.d

I ran the program with the following paramaters:

2024-04-13 16:14:50: {'start_state': "(1, 1, 'UP')", 'episode_num': 25, 'episode_length': 2000, 'alpha': 0.1, 'gamma': 1.0, 'epsilon': 0.3, 'num_obstacles': 5, 'trials': 4}

There were four trials with randomized goals and obstacles:

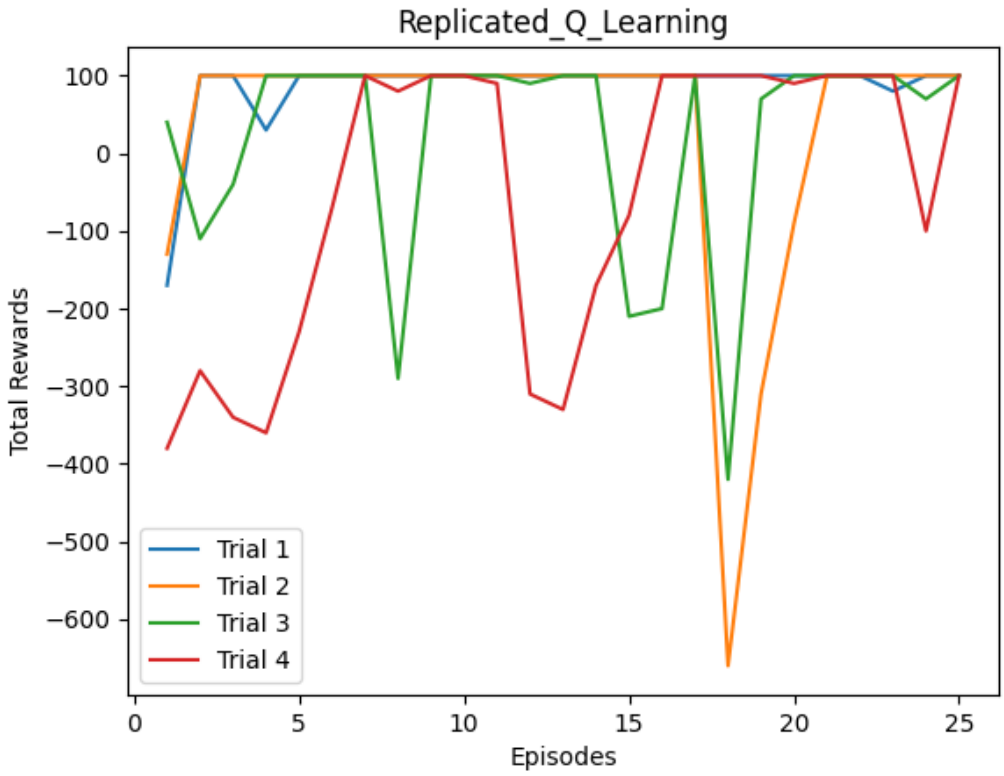
TRIAL 1 - OBSTACLES:[(8, 14), (6, 14), (5, 1), (1, 17), (2, 5)] - GOAL:(7, 4)

TRIAL 2 - OBSTACLES:[(2, 14), (10, 4), (10, 19), (7, 5), (10, 2)] - GOAL:(6, 5)

TRIAL 3 - OBSTACLES:[(2, 17), (6, 20), (1, 10), (3, 12), (5, 9)] - GOAL:(6, 2)

TRIAL 4 - OBSTACLES:[(4, 10), (6, 4), (2, 3), (4, 11), (10, 2)] - GOAL:(9, 6)

Below is the graph for Replicated Q Learning the I ran and the respective 4 different trials:



Unlike the Q_{MPD}, the replicated q learner did not have access to the $q(s, a)$ paramaters, so it had to learn in a completely POMDP environment. Despite this, it had pretty solid performance. It had a perfect 100 reward for several epsidoes in a row. At epsidoe 17 it ended up hadving -600 reward, but overall had a pretty solid learning curve.

- 2024-04-13 16:16:51: Trial: 1. Finished Replicated Learning Q Learning. Goal finding efficiency: 1.0
- 2024-04-13 16:19:44: Trial: 2. Finished Replicated Learning Q Learning. Goal finding efficiency: 0.92
- 2024-04-13 16:23:19: Trial: 3. Finished Replicated Learning Q Learning. Goal finding efficiency: 0.92
- 2024-04-13 16:29:06: Trial: 4. Finished Replicated Learning Q Learning. Goal finding efficiency: 0.68

0.0.10 2.e

I ran the program with the following paramaters:

2024-04-13 16:14:50: {'start_state': "(1, 1, 'UP')", 'episode_num': 25, 'episode_length': 2000, 'alpha': 0.1, 'gamma': 1.0, 'epsilon': 0.3, 'num_obstacles': 5, 'trials': 4}

There were four trials with randomized goals and obstacles:

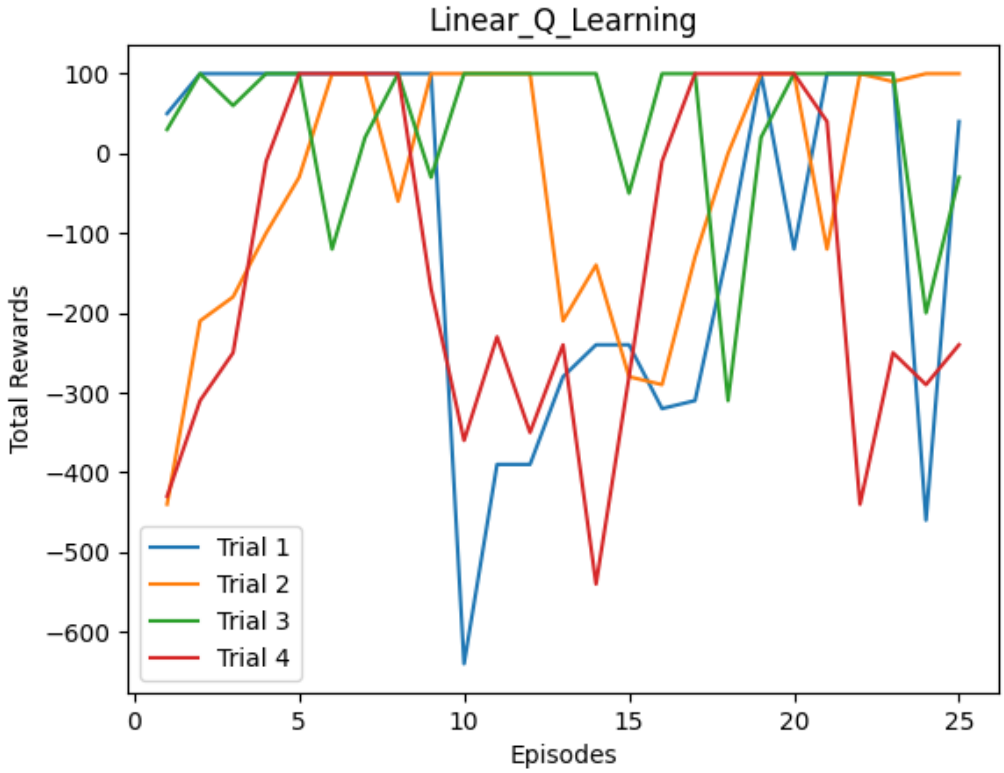
TRIAL 1 - OBSTACLES:[(8, 14), (6, 14), (5, 1), (1, 17), (2, 5)] - GOAL:(7, 4)

TRIAL 2 - OBSTACLES:[(2, 14), (10, 4), (10, 19), (7, 5), (10, 2)] - GOAL:(6, 5)

TRIAL 3 - OBSTACLES:[(2, 17), (6, 20), (1, 10), (3, 12), (5, 9)] - GOAL:(6, 2)

TRIAL 4 - OBSTACLES:[(4, 10), (6, 4), (2, 3), (4, 11), (10, 2)] - GOAL:(9, 6)

Below is the graph for Linear Q Learning the I ran and the respective 4 different trials:



Unlike the Q_{MPD}, the linear q learner did not have access to the $q(s, a)$ paramaters, so it had to learn in a completely POMDP environment. The learner had solid performance at the begining, bad performance halfway, then good perfance at the end. The graphs over all trials kind of make a 'U' shape. There was solid performance all around except for trial 4, which is likely because of that goal being located farther away from the start state than the others.

- 2024-04-13 16:16:41: Trial: 1. Finished Linear Q Learning. Goal finding efficiency: 0.68
- 2024-04-13 16:19:13: Trial: 2. Finished Linear Q Learning. Goal finding efficiency: 0.64
- 2024-04-13 16:22:32: Trial: 3. Finished Linear Q Learning. Goal finding efficiency: 0.88
- 2024-04-13 16:27:35: Trial: 4. Finished Linear Q Learning. Goal finding efficiency: 0.44

0.0.11 Comparing POMDP Agents

Similarities Every learner did well on trial 3. Every learner did well on trial 2.

Differences Replicated Q learning had the best performance. Linear Q learning had the worst performance between all POMDP learners. Q_{MDP} had the lowest reward minimum.

Conclusions Linear Q learning seemed to be unstable on the episode number I used, replicated learning seemed to perform the best, and QMDP had good middle ground performance. In parially observable environments, replicated q learning would be the option I would use.