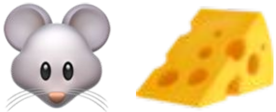
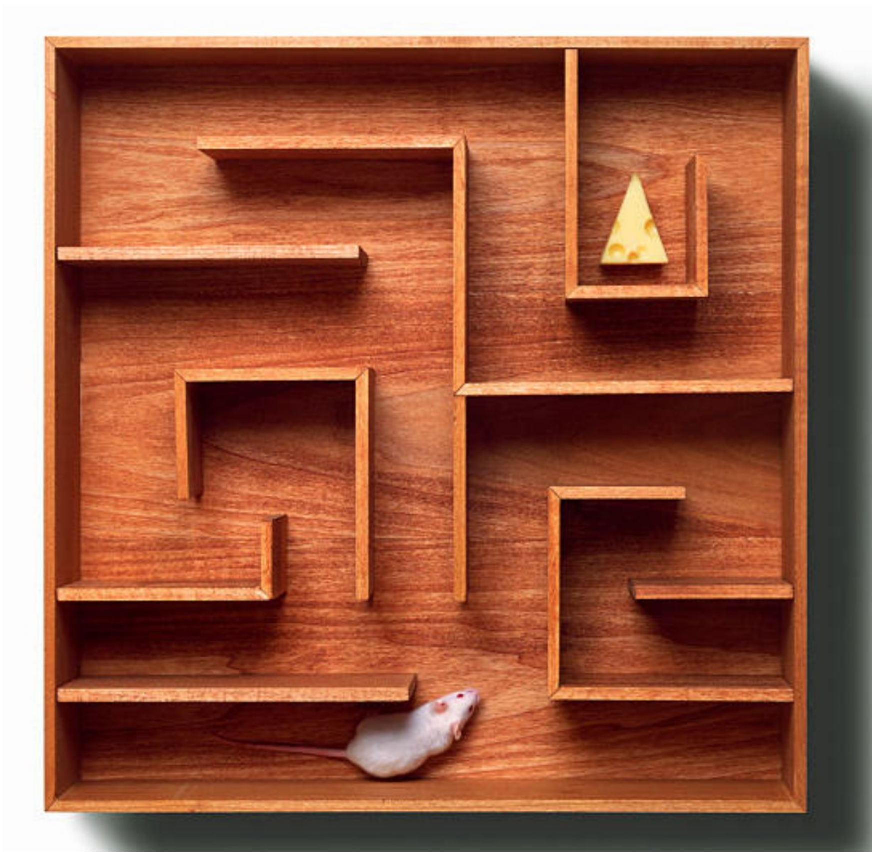
Mouse Maze Simulation



Donya Fozoonmayeh

Agenda

- Background
- Problem Setup
- Modeling
- Demo
- Results



PSYCHOLOGY & NEUROSCIENCE

Research Question

“Can I train a reinforcement learning agent(🐭) to learn to navigate through a maze to find the ultimate reward(🧀)?”

RL Problem Setup

Agent: 🐭

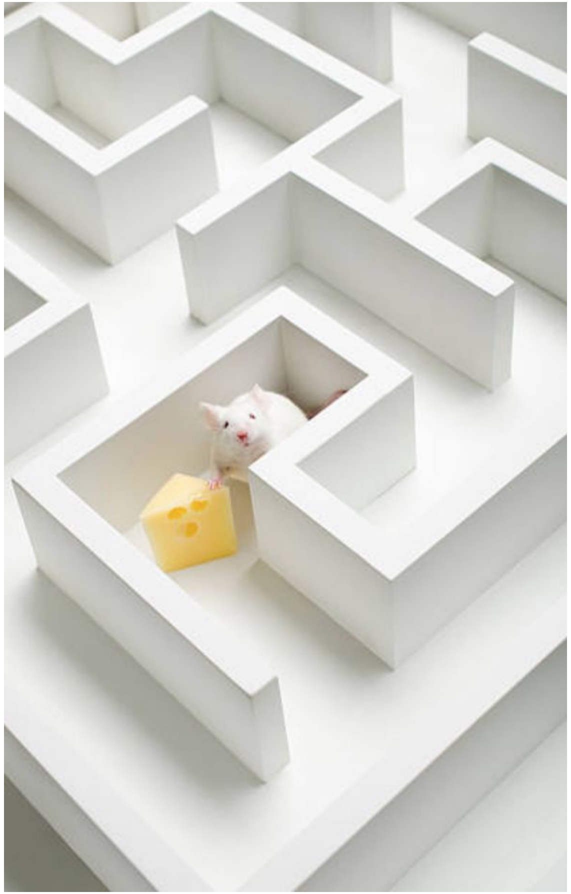
Reward: 🧀 or -0.1/Number of cells

Environment: OpenAI Gym maze

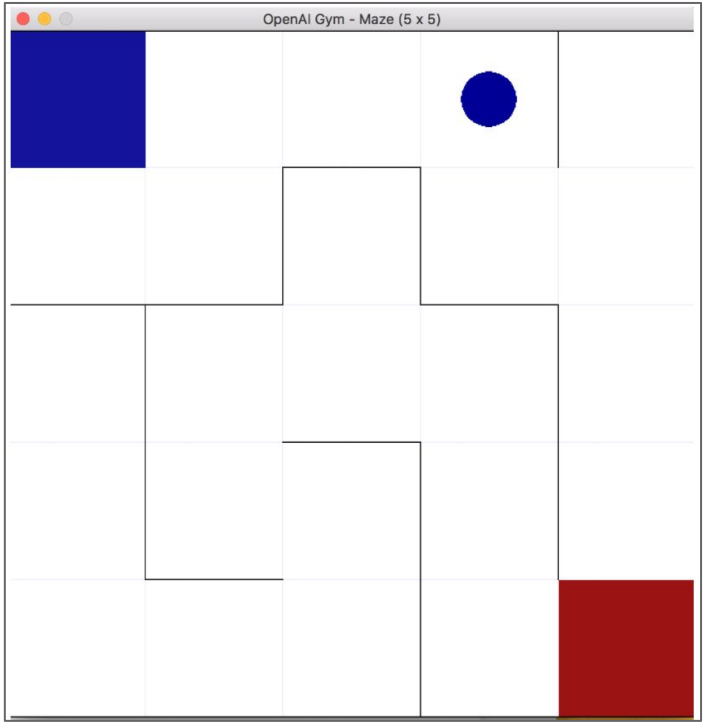
State: Location on the maze

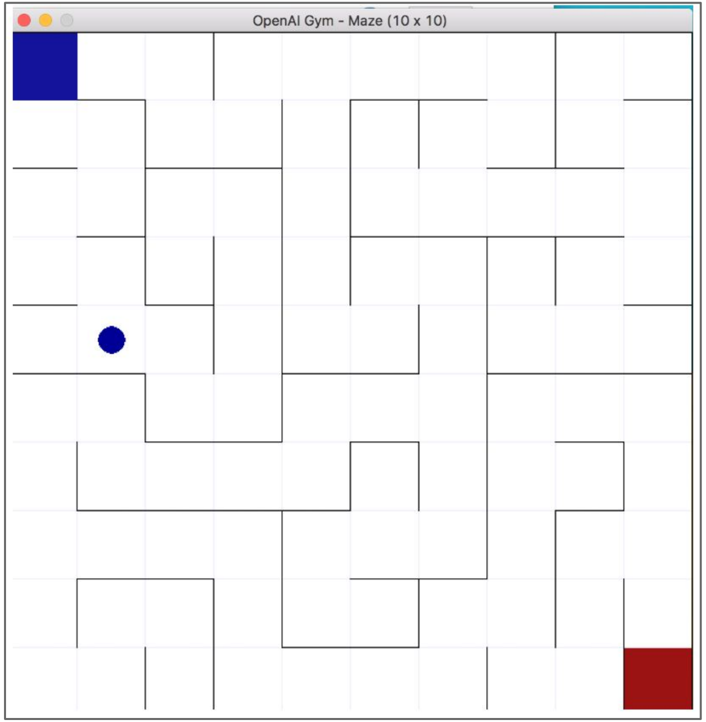
Action: North, South, East, West

Sequential: Yes ✔️



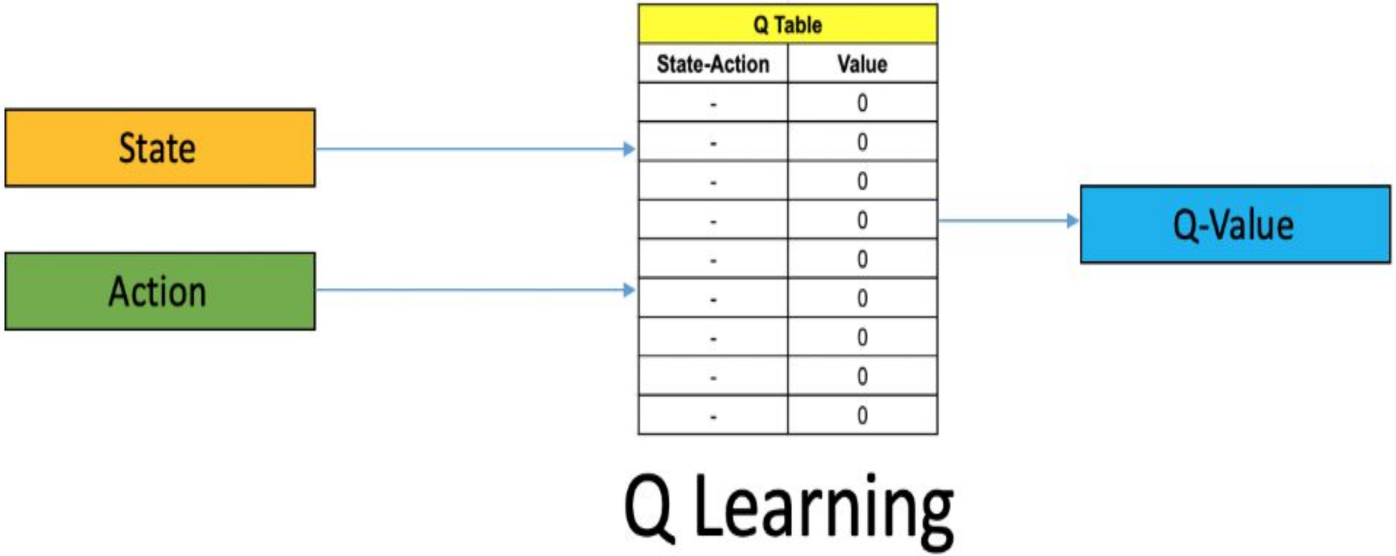
Environment





Modeling

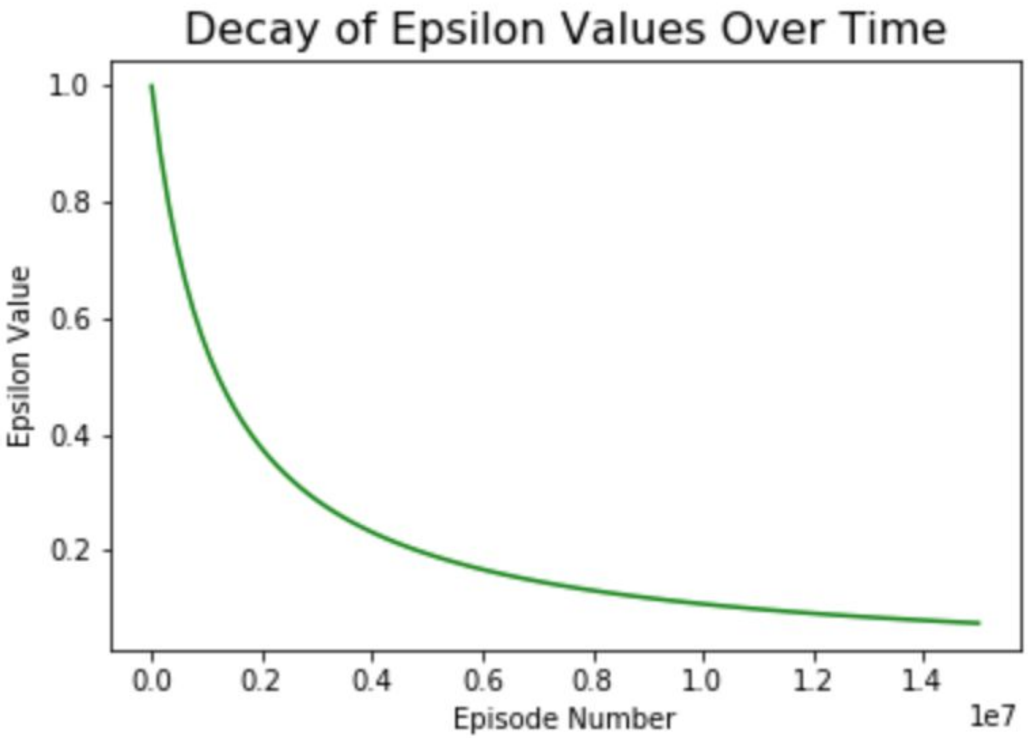
- Q-Learning



```python
class QlearningMouse:
    """
    The QlearningMouse object is a reinforcement learning
    mouse agent that uses Q-learning to work its way through
    an Open-ai maze to get to the end of the maze where the
    reward (the hypothetical cheese) is.

    This object has attributes such as state, decay factor,
    discount rate, exploration rate and a Q-table.
    It also has methods for stepping (deciding what the next
    action is using the epsilon-greedy approach) and updating
    the Q-table, as well as initializing a new episode.

    This object uses a decaying epsilon and learning rate,
    which improves its performance and makes it converge
    faster.
    """

    def __init__(self,
                 maze_width,
                 maze_height,
                 epsilon=0.9,
                 learning_rate=0.9,
                 decay_factor=10,
                 gamma=0.90):

        self.epsilon = epsilon
        self.state = (0., 0.)
        self.learning_rate = learning_rate
        self.decay_factor = decay_factor
        self.gamma = gamma
        self.episode = 0

        # initializing q-table
        self.q_table = {}
        for i in range(maze_width):
            for j in range(maze_height):
                self.q_table[(i, j)] = [0., 0., 0., 0.]
```

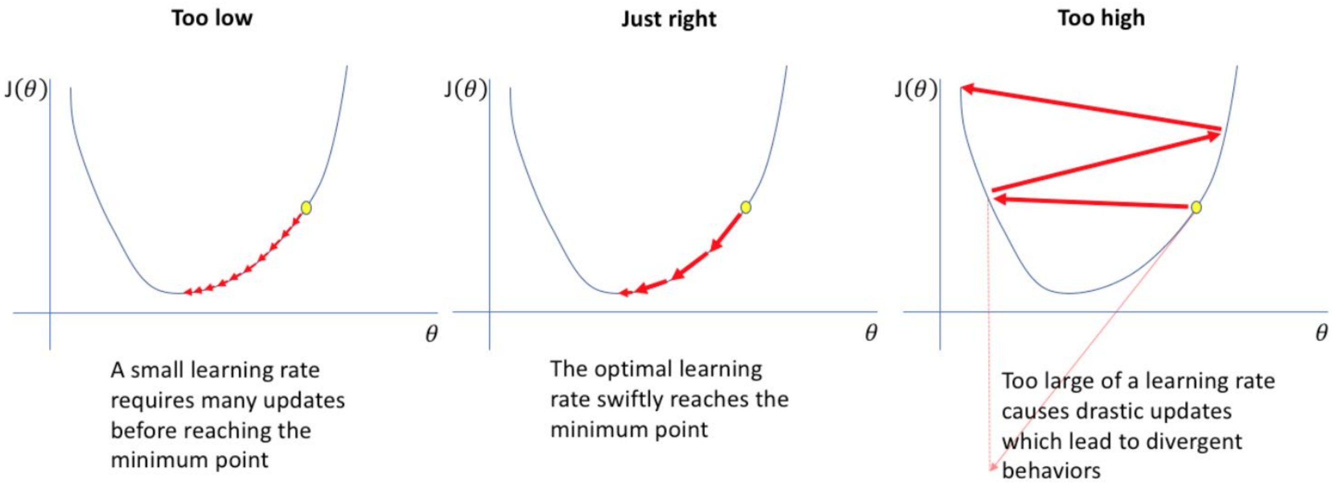
```python
def step(self):
    """
    Returns the next action, exploiting (1-epsilon)% of the time and
    exploring epsilon% of the time.
    """
    if random.random() < self.decayed_epsilon():
        return env.action_space.sample()
    else:
        action_idx = np.argmax(
            self.q_table[(self.state[0], self.state[1])])
        return int(action_idx)
```

```python
def decayed_epsilon(self):
    """
    Given the number of episodes, returns epsilon.
    Decayed epsilon is used because it makes the convergance much
    faster. The epsilon is high at first, making the agent explore
    more and it decreases as the number of episodes increase and the
    agent exploits more.
    """
    # exploration rate is never smaller than 0.001 and never greater than 0.9
    return max(0.001, min(0.9, 1.0 - np.log10(self.episode/self.decay_factor)))
```



Decay of Epsilon Values Over Time
Epsilon Value
1.0
0.8
0.6
0.4
0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1e7
Episode Number

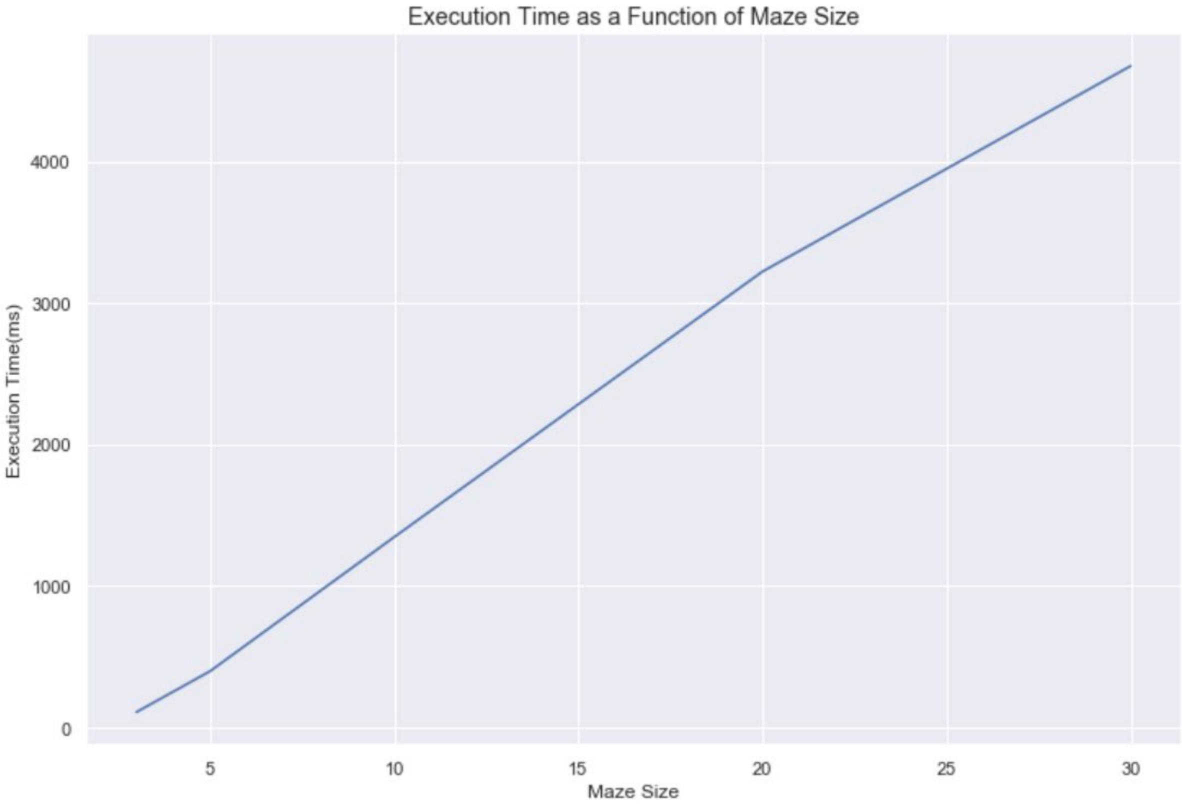
```python
def decayed_learning_rate(self):
    """
    Given the number of episodes, returns learning rate.
    Similar to the decayed_epsilon() function, this function returns
    the learning rate depending on the number of episodes. The learning
    rate is high at first and it decreases as the number of episodes
    increase, resulting in better learning performance.
    """
    # learning rate is never smaller than 0.2 and never greater than 0.9
    return max(0.2, min(0.9, 1.0 - np.log10(self.episode/self.decay_factor)))
```


Too low
Just right
Too high
J(θ)
J(θ)
J(θ)
θ
θ
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

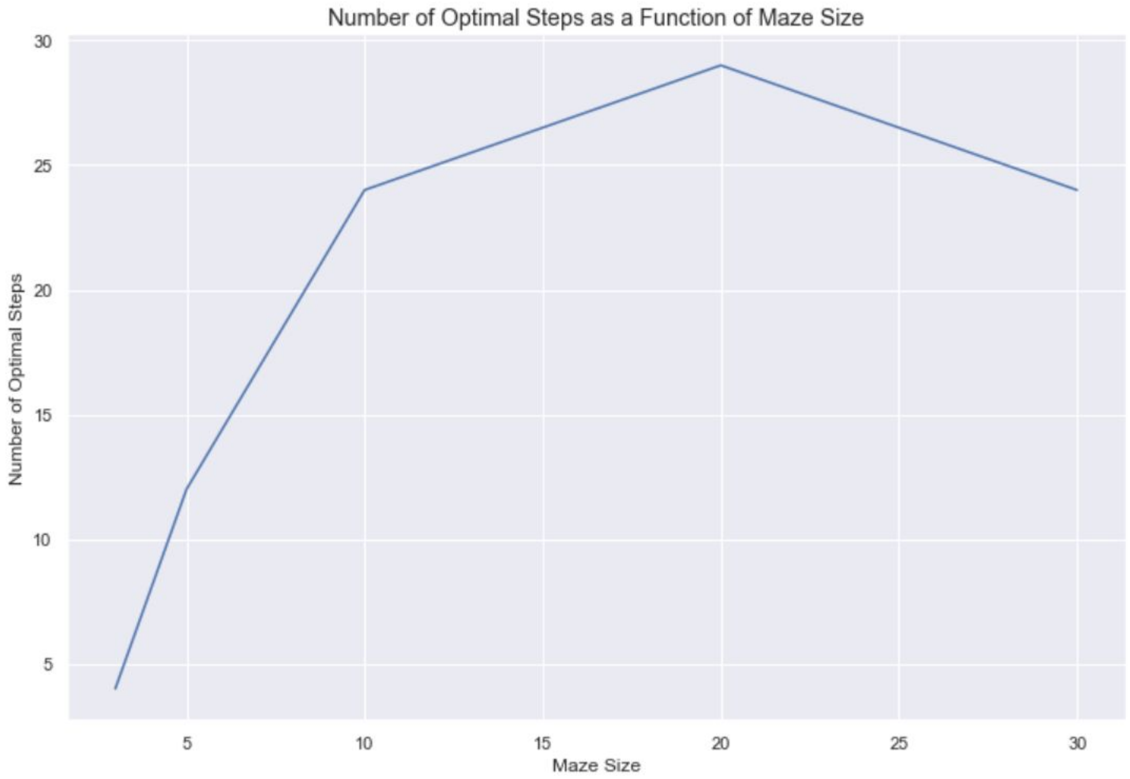
```python
def update(self, action, observation, reward):
    """
    Updates q-table based on an action, observation and reward.
    """
    best_q = np.amax(
        self.q_table[(int(observation[0]), int(observation[1]))])

    q_table_key = (self.state[0], self.state[1])

    # updating Q-table using the Q-learning formula
    self.q_table[q_table_key][action] += \
        self.decayed_learning_rate() * (reward + self.gamma * (best_q) -
                                        self.q_table[q_table_key][action])

    self.state = [int(observation[0]), int(observation[1])]
```

Demo

Results

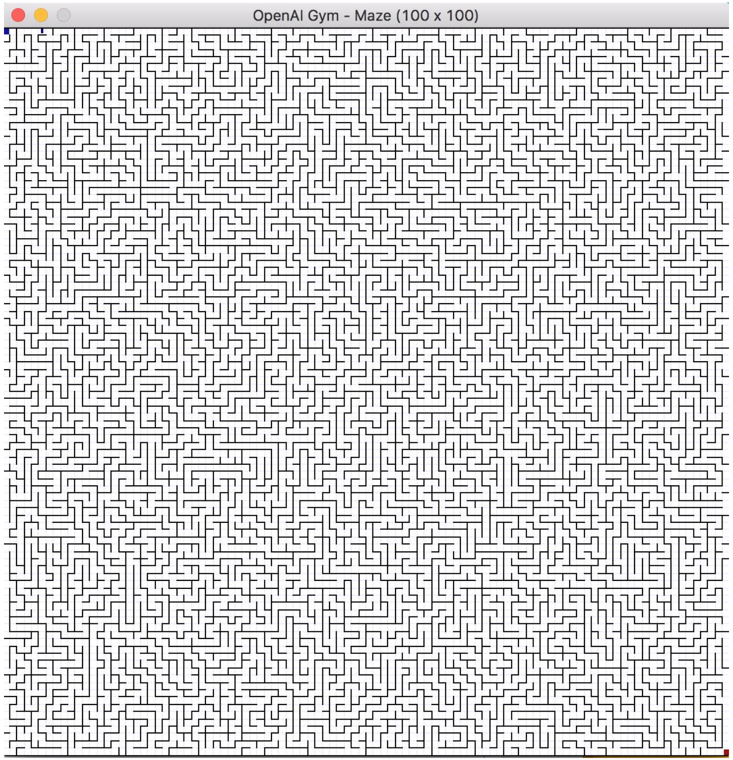


Execution Time as a Function of Maze Size
Execution Time(ms)
0
1000
2000
3000
4000
5
10
15
20
25
30
Maze Size


Number of Optimal Steps as a Function of Maze Size
Number of Optimal Steps
Maze Size
30
25
20
15
10
5
5
10
15
20
25
30

Future Directions

- Deep Q-Learning
- Double Q-Learning
- Larger mazes




THANK YOU

Questions?