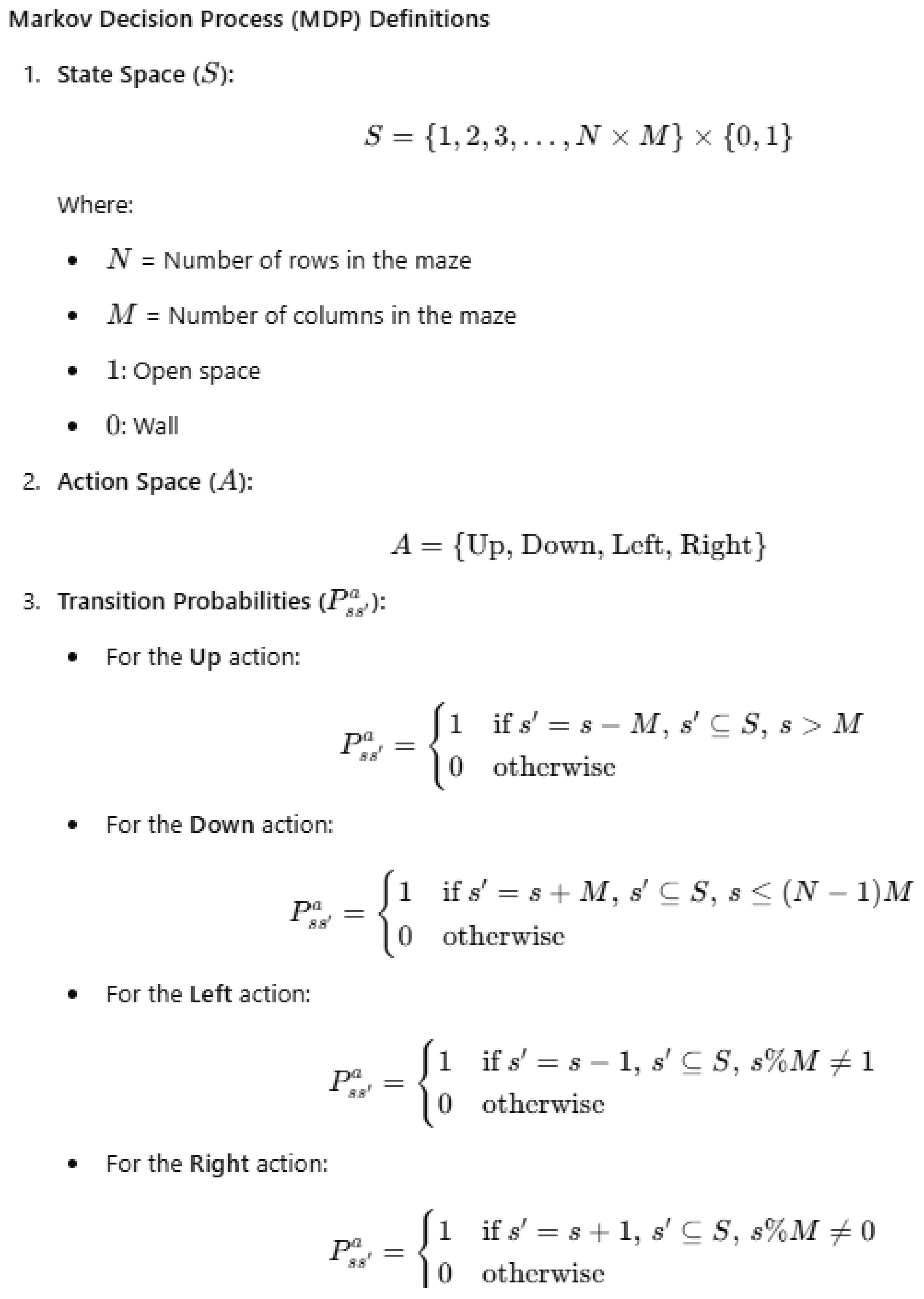
Maze Solver Using Markov Decision Processes

Nicolas Kozachuk
Department of Electrical & Computer Engineering
Lehigh University
ngk324@lehigh.edu

Date: Mar. 31st, 2024

Methodology to Convert the Image into a Markov Decision Process



Markov Decision Process (MDP) Definitions

1. **State Space (S):**

$$S = \{1, 2, 3, \dots, N \times M\} \times \{0, 1\}$$

   Where:

   - N = Number of rows in the maze
   - M = Number of columns in the maze
   - 1: Open space
   - 0: Wall

2. **Action Space (A):**

$$A = \{\text{Up}, \text{Down}, \text{Left}, \text{Right}\}$$

3. **Transition Probabilities ($P^a_{ss'}$):**

   - For the **Up** action:

$$P^a_{ss'} = \begin{cases} 1 & \text{if } s' = s - M,\ s' \subseteq S,\ s > M \\ 0 & \text{otherwise} \end{cases}$$

   - For the **Down** action:

$$P^a_{ss'} = \begin{cases} 1 & \text{if } s' = s + M,\ s' \subseteq S,\ s \leq (N-1)M \\ 0 & \text{otherwise} \end{cases}$$

   - For the **Left** action:

$$P^a_{ss'} = \begin{cases} 1 & \text{if } s' = s - 1,\ s' \subseteq S,\ s\%M \neq 1 \\ 0 & \text{otherwise} \end{cases}$$

   - For the **Right** action:

$$P^a_{ss'} = \begin{cases} 1 & \text{if } s' = s + 1,\ s' \subseteq S,\ s\%M \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

4. Rewards ($R^a_{ss'}$):

$$R^a_{ss'} = \begin{cases} 100 & \text{if } s' \in S,\ \frac{M+N}{2} \leq 10 \\ 100^{\frac{M+N}{20}} & \text{if } s' \in S,\ \frac{M+N}{2} > 10 \\ -10 & \text{if } s' \notin S,\ \frac{M+N}{2} \leq 10 \\ -10^{\frac{M+N}{20}} & \text{if } s' \notin S,\ \frac{M+N}{2} > 10 \end{cases}$$

Termination Criteria

The termination criteria for value iteration are defined as:

$$\left| V^{(k)}(s) - V^{(k-1)}(s) \right| < \varepsilon, \quad \text{where } \varepsilon = 0.0001$$

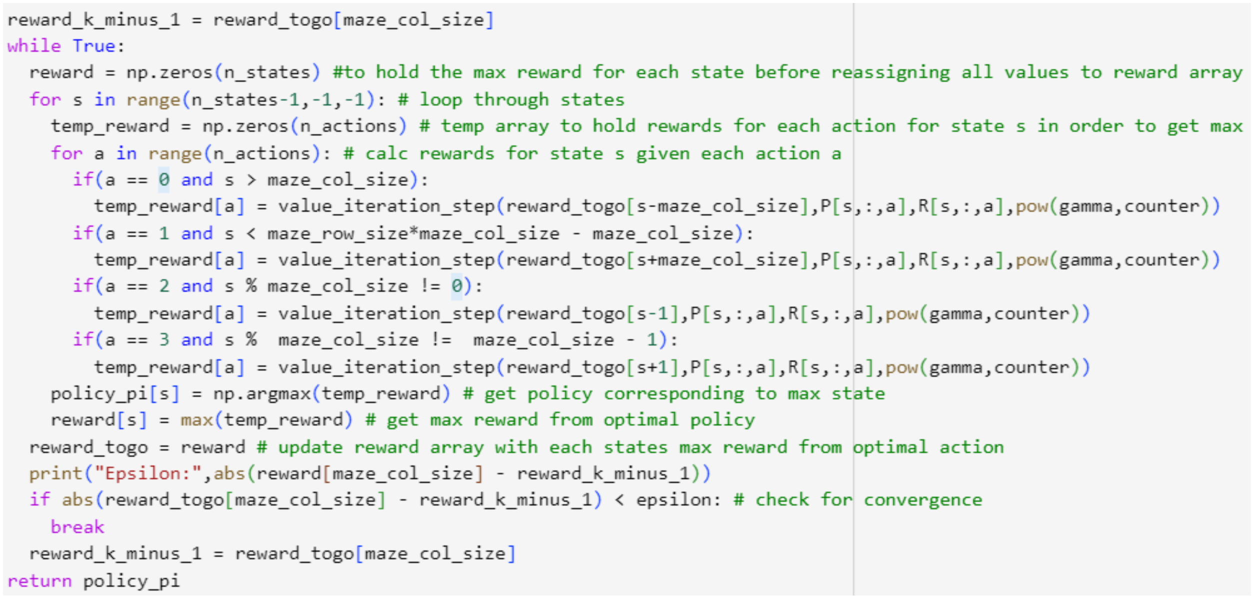
The states are defined as each location of the board and whether or not the location is an open space or wall. Thus, for an NxM sized maze, there are N*M total states, each either being a 0 or 1, where 0 indicates a wall and 1 indicates a space. The states are defined as the first row of the maze array being states 1 to M, the 2nd row being states M+1 to 2*M, etc. There are 4 possible actions, move up, move down, move left, and move right. There are 4 transition matrices and 4 reward matrices, one corresponding to each action. The sizes of each of these matrices as M*N x M*N, as they represent the transition probabilities and rewards for transitioning from each state to each other state. For the transition matrices, there is either all 0s or a single value of 1 and the rest 0s for a given row. This is because when at a state, for a given action that is a move in a single direction, there is either none or 1 transition to another state possible.

For the up action, a transition is done by subtracting M from the current state location, as M is the number of columns, so subtracting M moves one to the state one row above, corresponding to an up move. For the down action, a transition is done by adding M to the current state location, as M is the number of columns, so adding M moves one to the state one row below, corresponding to a move down. For the left action, a transition is done by subtracting 1 from the current state. For the right action, a transition is done by adding 1 to the current state. Additionally, it must be checked that a move doesn't bring one over the edge of the board. An up move is possible(resulting in a probability of 1 in the transition matrix for s to s') if s = 1 and s' = 1, meaning one can move from open space to open space and there is no walls, as well that s' is a subset as s, meaning there are no moves up past the top of the maze. If no up move is possible, the whole row corresponding to s are 0s for the transition matrix. The same is for the down matrix, left matrix, and right matrix, except the check is that the move down doesn't go down past the bottom of the maze, the move left doesn't go past the left edge of the maze, and the move right doesn't go past the right edge of the maze. Checking that a move down doesn't go past the end of the maze is easy as s' would not be a subset as s because it would be at a location that does not exist in the maze. For the left and right move it is a bit trickier, as a move left and right could move one off the board, but due to how the states are defined, a move left or right by adding or subtracting one could move one to the state that is a row above or below. As explained previously the states 1 to M correspond to first row and M + 1 to 2*M corresponds to the second row, so for example, when at state M + 1 in the second row, if subtracting 1 to move left, state M + 1 is actually at the left edge and one moves to M which is a state that is a row above. This can be checked for moving left and right by checking that s % M != 0 for a left move and s % M != M - 1 for a right move, meaning that a left move cannot be made when at the left edge and a right move cannot be made at the right edge.

The reward matrices are defined similar to the transition matrices but with different values. The magnitude of the reward matrices are also defined depending on the size of the maze matrix. In the cases when the transition matrices have a value of 0, the corresponding reward matrix values are negative and when the transition matrices have a value of 1, the reward is positive. The magnitude of these positive and negative values are dependent on the size of the matrix, as large matrices need larger values to properly propagate the reward values through the paths of the maze. To account for varying sizes of M and N, the average maze edge length is taken by calculating (M + N) / 2. If the average edge length is less than or equal to 10, the negative rewards are -10 and the positive rewards are 100. If the average edge length is greater than 10, the negative reward is -10^((M+N)/(2*10)) and the positive reward is 100^((M+N)/(2*10)). I came up with this as I used -10 and 100 as the rewards when I first implemented using a maze with size M=N=10, but I noticed when I doubled the size of the maze the rewards needed to be increased exponentially due to the total number of states increasing exponentially. Thus when testing with a 20x20 maze, 20 is 2 times bigger than 10 and I needed the rewards for a 20x20 maze to be equivalent to the rewards used with 10x10 maze to the power of 2.

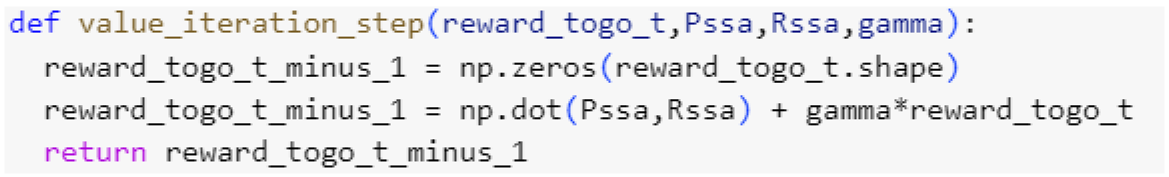
Algorithm Used to Solve for the Optimal Policy

I implemented the maze solver using Markov Decision Processes using value iteration. The first loop is a while loop that loops until the termination criteria is met, the absolute value of the value of a state at step k -1 minus the value of the same state at step k is less than epsilon. Each loop of the while loop goes through each state and calculates the reward for each action at a given state and saves the max reward and the corresponding action.



```
reward_k_minus_1 = reward_togo[maze_col_size]
while True:
  reward = np.zeros(n_states) #to hold the max reward for each state before reassigning all values to reward array
  for s in range(n_states-1,-1,-1): # loop through states
    temp_reward = np.zeros(n_actions) # temp array to hold rewards for each action for state s in order to get max
    for a in range(n_actions): # calc rewards for state s given each action a
      if(a == 0 and s > maze_col_size):
        temp_reward[a] = value_iteration_step(reward_togo[s-maze_col_size],P[s,:,a],R[s,:,a],pow(gamma,counter))
      if(a == 1 and s < maze_row_size*maze_col_size - maze_col_size):
        temp_reward[a] = value_iteration_step(reward_togo[s+maze_col_size],P[s,:,a],R[s,:,a],pow(gamma,counter))
      if(a == 2 and s % maze_col_size != 0):
        temp_reward[a] = value_iteration_step(reward_togo[s-1],P[s,:,a],R[s,:,a],pow(gamma,counter))
      if(a == 3 and s %  maze_col_size !=  maze_col_size - 1):
        temp_reward[a] = value_iteration_step(reward_togo[s+1],P[s,:,a],R[s,:,a],pow(gamma,counter))
    policy_pi[s] = np.argmax(temp_reward) # get policy corresponding to max state
    reward[s] = max(temp_reward) # get max reward from optimal policy
  reward_togo = reward # update reward array with each states max reward from optimal action
  print("Epsilon:",abs(reward[maze_col_size] - reward_k_minus_1))
  if abs(reward_togo[maze_col_size] - reward_k_minus_1) < epsilon: # check for convergence
    break
  reward_k_minus_1 = reward_togo[maze_col_size]
return policy_pi
```

Figure 1: Value Iteration Implementation



```
def value_iteration_step(reward_togo_t,Pssa,Rssa,gamma):
  reward_togo_t_minus_1 = np.zeros(reward_togo_t.shape)
  reward_togo_t_minus_1 = np.dot(Pssa,Rssa) + gamma*reward_togo_t
  return reward_togo_t_minus_1
```

Figure 2: Value evaluation for value iteration step

The while loop begins by defining a reward array that holds the rewards for each state for one loop through of states and each states different actions. Then there is a for loop that loops through each of the states. First in the for loop to loop through states, a temp_reward array is made that is size 4, for each action, as this array holds the reward for a given state for each action and is used to get the max reward for a state over all actions in order to get the optimal policy. Next is the for loop to loop through each action for a state. The reward for each action is defined by calling the value_iteration_step() function as seen in Figure 2, with the parameters of the reward for s', the transition matrix and reward matrix for the given state and action, as well as the gamma value. The if statements for the actions are used to make sure no action is mathematically performed that isn't truly possible, such as moving left when at the leftmost space that would result in going off the board, but mathematically would calculate transitioning to the right most state in the previous row. Additionally, the s' state location calculation used for finding R(s') is done by, for left and right subtracting and adding 1, and for up and down, subtracting and adding the number of columns. The states are defined as the first row being states 1 to M, the 2nd row being states M+1 to 2*M, etc, so subtracting the number of columns(which is M), brings one to the state corresponding to a move up. Once the for loop for the actions is complete, the max reward for the given state over each action is saved in the reward vector for the current state and the optimal action is saved for the current state. This continues as the for loop loops through each state and finds the optimal policy and reward value for the given state. After this is done for every state(completing the for loop), the reward_togo vector is assigned the values of the reward vector. This extra reward vector is needed so that when calculating the rewards during the looping of states, the rewards calculated during the looping process don't affect the reward to go for the next iterations of the current looping. Once the terminal condition is met, the while loop is broken from and the solution for the maze is displayed.

Example Outputs with Mazes of Varying Sizes

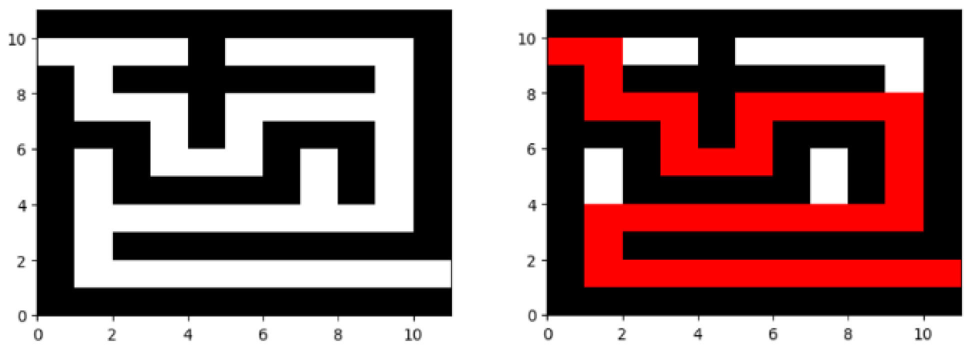


Figure 1: 5x5 Starting Maze and MDP Solution Path

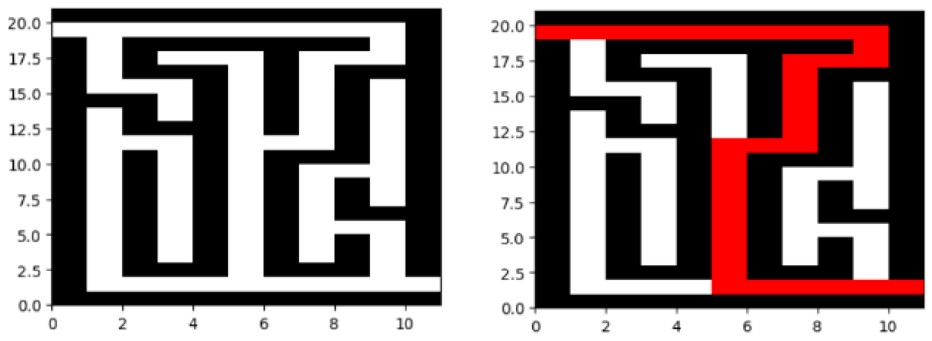


Figure 2: 10x5 Starting Maze and MDP Solution Path

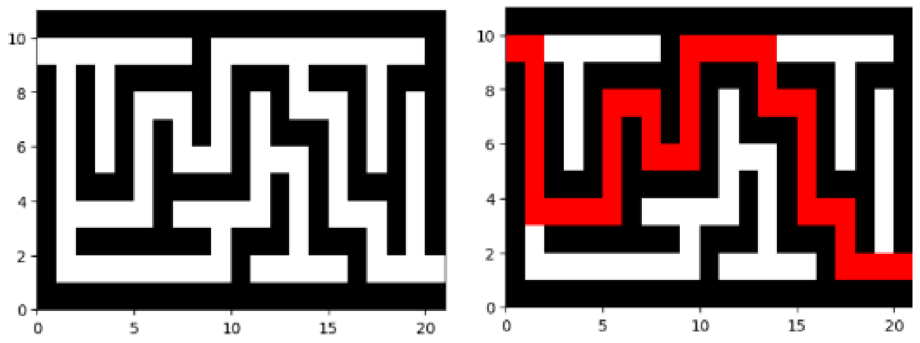


Figure 3: 5x10 Starting Maze and MDP Solution Path

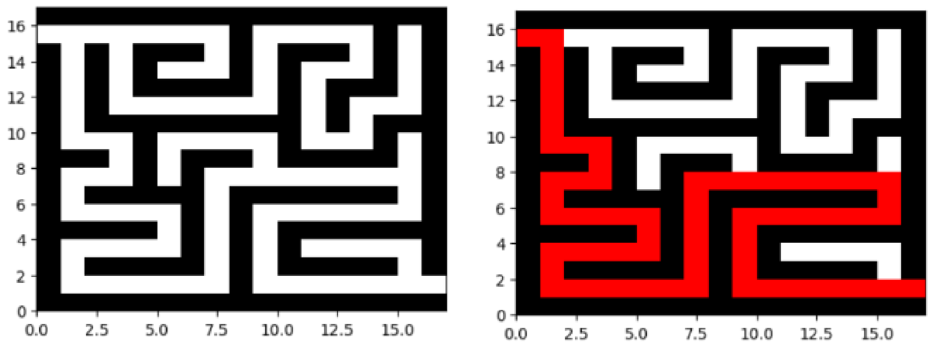


Figure 4: 8x8 Starting Maze and MDP Solution Path

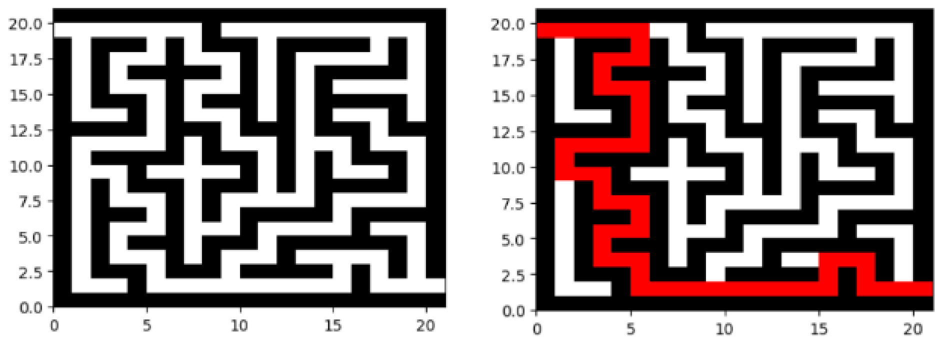


Figure 5: 10x10 Starting Maze and MDP Solution Path

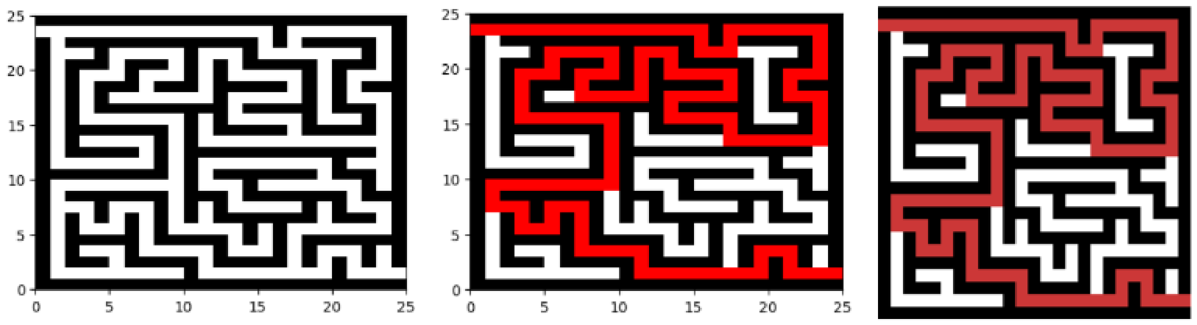


Figure 6: 12x12 Starting Maze, MDP Solution Path, and Provided Solution Path(Dark Red)

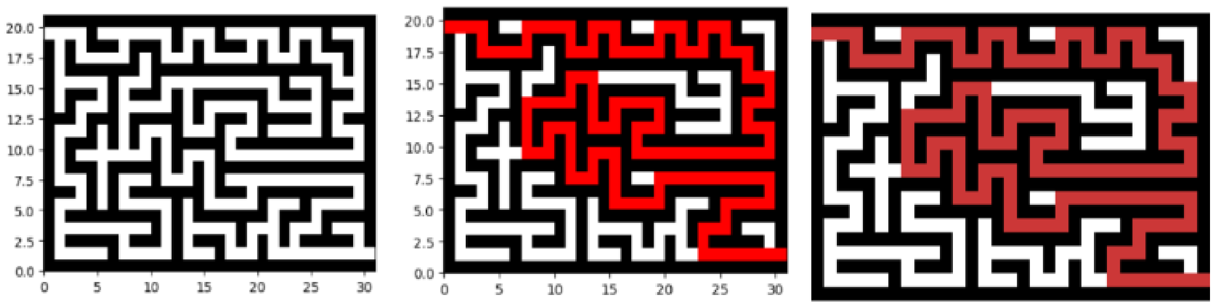


Figure 7: 10x15 Starting Maze, MDP Solution Path, and Provided Solution Path(Dark Red)

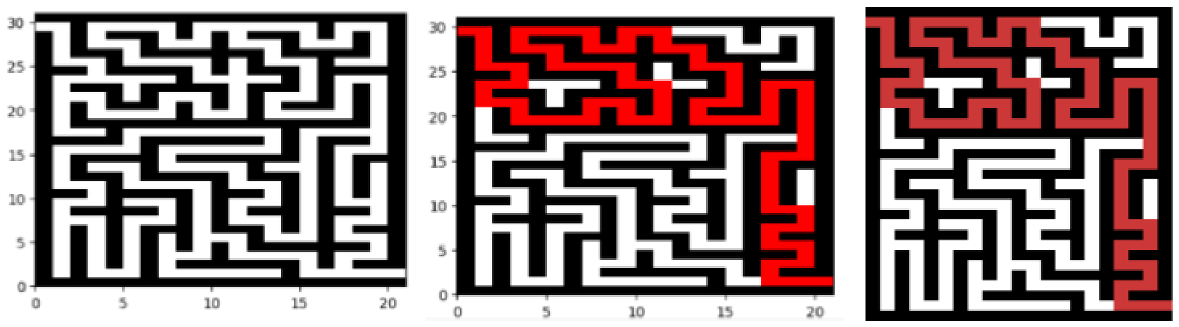


Figure 8: 15x10 Starting Maze, MDP Solution Path, and Provided Solution Path(Dark Red)

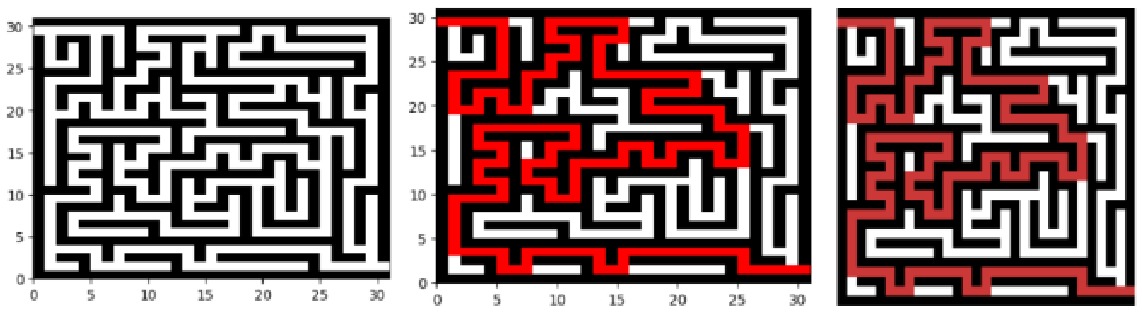


Figure 9: 15x15 Starting Maze, MDP Solution Path, and Provided Solution Path(Dark Red)

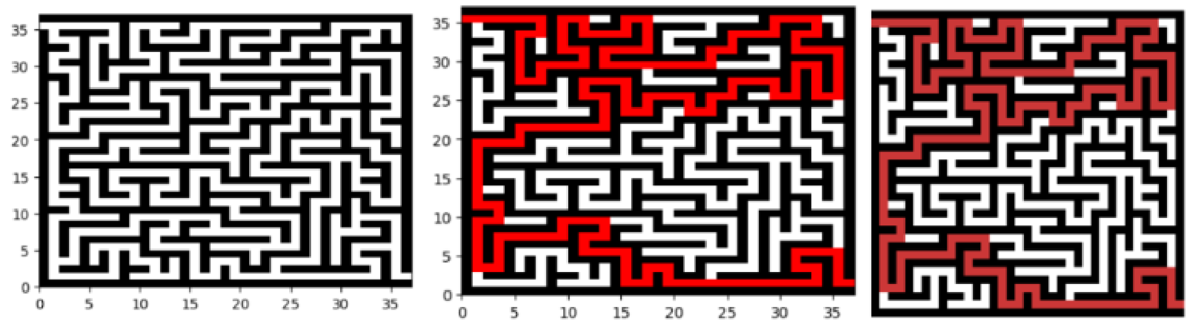


Figure 10: 18x18 Starting Maze, MDP Solution Path, and Provided Solution Path(Dark Red)

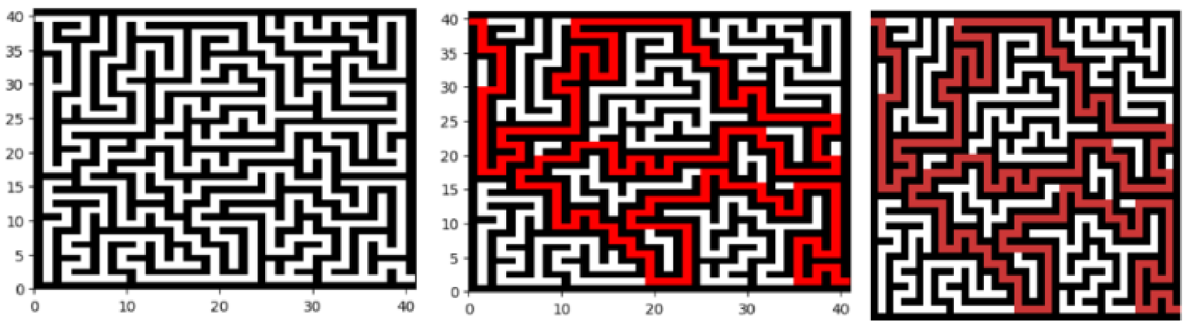


Figure 11: 20x20 Starting Maze, MDP Solution Path, and Provided Solution Path(Dark Red)

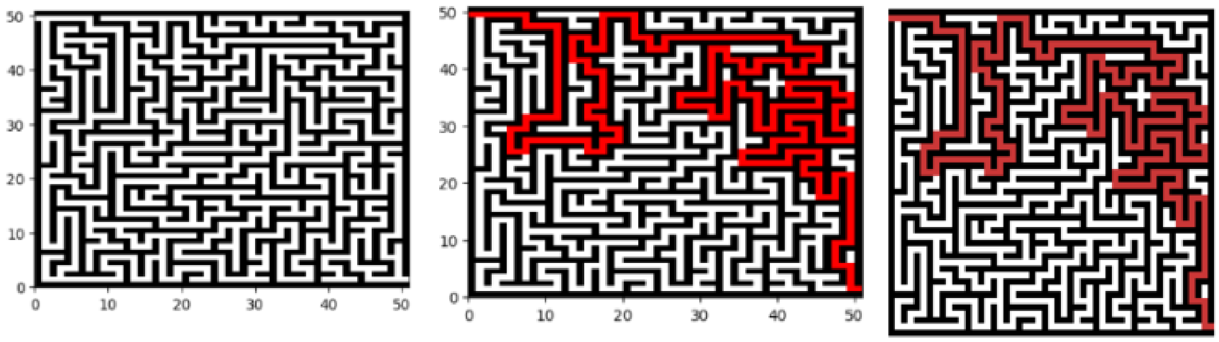


Figure 12: 25x25 Starting Maze, MDP Solution Path, and Provided Solution Path(Dark Red)

Study on Execution Time

Maze Size	Maze 1	Maze 2	Maze 3	Maze 4	Maze 5	Average
5x5	1s	1s	1s	1s	2s	1.2s
5x10	11s	10s	10s	9s	9s	9.8s
10x5	11s	10s	10s	10s	10s	10.2s

8x8	12s	12s	12s	13s	13s	12.4s
10x10	22s	20s	21s	21s	22s	21.2s
12x12	33s	34s	34s	34s	35s	34s
10x15	40s	39s	38s	42s	41s	40s
15x10	41s	41s	42s	42s	42s	41.6s
15x15	65s	63s	65s	66s	65s	64.4s
18x18	119s	118s	118s	129s	123s	121.4s
20x20	167s	165s	169s	168s	167s	167.2s
25x25	424s	432s	431s	427s	430s	428.8s

Figure 13: Table of Maze Sizes and Corresponding Execution Time

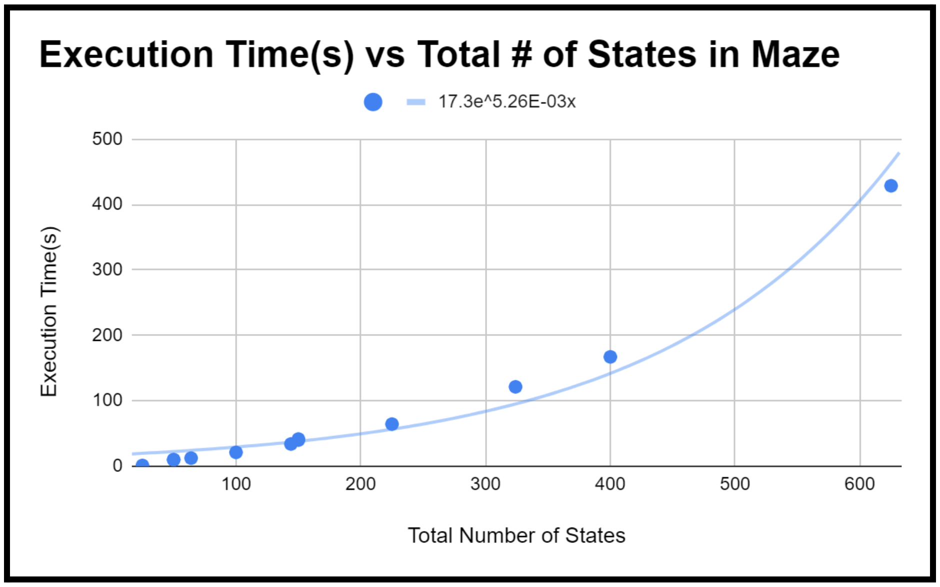


Figure 14: Plot of Executive Time(s) vs Total Number of States in Maze

Figure 13 shows each of the different sizes of NxM maze arrays that were tested and the corresponding execution times for five different mazes of each size, as well as their averages. It can be seen in Figure 13 that as the size of the maze gets bigger, the jump in the execution time becomes larger. Figure 14 is a plot of the execution times in seconds of the mazes vs the number of states in the maze. For example, for a 5x5 maze, there are 25 states. It can be seen in Figure 14 that the execution time increases exponentially with the number states, which makes sense. Value iteration contains looping over every state and the possible actions for every state, so the computation time grows exponentially with the size of the number of states. Additionally, since for an NxN maze, the total number of states increases exponentially as N increases, it can be noted that increasing the size of a maze's row/column dimension causes a great exponential increase in execution time.

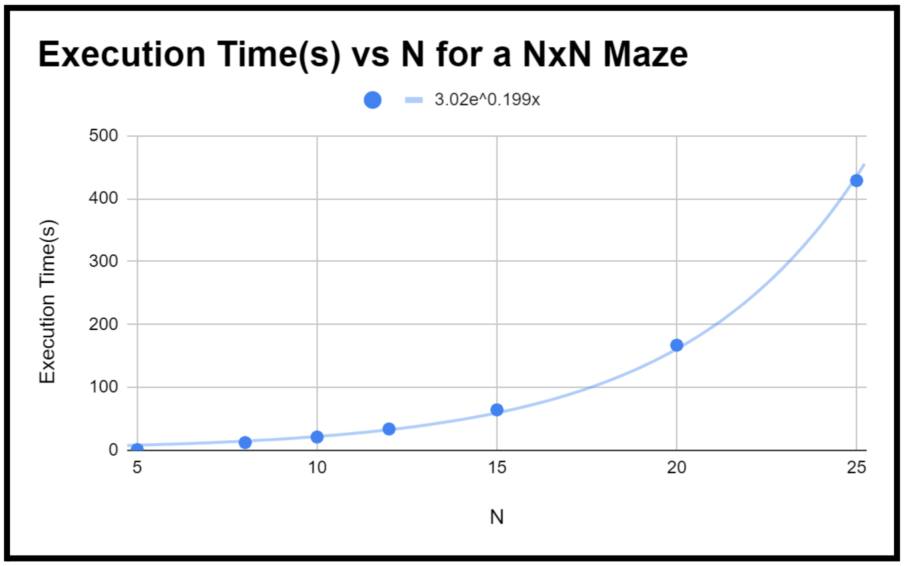


Figure 15: Plot of Executive Time(s) vs N for NxN Maze

This great exponential increase in execution time can be seen in Figure 15, where only the NxN mazes are plotted, when looking at the execution time vs N for NxN mazes, the execution time increases exponentially as N increases. The exponential trend line for Figure 15 is surprisingly "nice" and I can conclude that the execution time in seconds for the maze solver increases as a function of 3e^(2N), where N is the row/column size of an NxN maze.