COL778

ASSIGNMENT 2

Planning in a Markov Decision Process

Shubh Goel *(2020EE10672)*
Link to Plots

Date of submission of Report: March 3, 2024

```
class Domain:
    def __init__(self, grid, p = 0.8, gamma = 0.9, living_reward = 0,
        hole_reward = 0, goal_reward = 1, snow = False):
        self.p = p
        self.gamma = gamma
        self.living_reward = living_reward
        self.hole_reward = hole_reward
        self.goal_reward = goal_reward
        self.grid = grid
        self.snow = snow

    def transition_prob(self, curr_state, action, next_state):
        d = {'r':(0,1), 'd':(1,0), 'l':(0,-1), 't':(-1,0)}
        x_curr,y_curr = curr_state
        r,c = self.grid.shape
        x_int = max(min(x_curr+d[action][0],r-1),0)
        y_int = max(min(y_curr+d[action][1],c-1),0)
        pr = {}
        if(self.snow):
            opp = {'r':'l','d':'t','l':'r','t':'d'}
            pr[(x_int,y_int)] = 1/3
            for key,value in d.items():
                if(key == action):
                    continue
                x = max(min(x_curr+value[0],r-1),0)
                y = max(min(y_curr+value[1],c-1),0)
                if pr.get((x,y)) is None:
                    if(key == opp[action]):
                        pr[(x,y)] = 0
                    else:
                        pr[(x,y)] = 1/3
                else:
                    if(key == opp[action]):
                        pr[(x,y)] += 0
                    else:
                        pr[(x,y)] += 1/3
            return pr[next_state]

        pr[(x_int,y_int)] = self.p

        for key,value in d.items():
            if(key == action):
                continue
            x = max(min(x_curr+value[0],r-1),0)
            y = max(min(y_curr+value[1],c-1),0)
            if pr.get((x,y)) is None:
                pr[(x,y)] = (1 - self.p)/3
            else:
                pr[(x,y)] += (1 - self.p)/3

        return pr[next_state]

    def get_reward(self,state):
        x,y = state

        if(self.grid[x][y] == 'F' or self.grid[x][y] == 'S'):
            return self.living_reward
        elif (self.grid[x][y] == 'H'):
```

```
                return self.hole_reward
        else:
                return self.goal_reward

    def get_state_actions(self,state):
        x,y = state
        r,c = self.grid.shape

        states = list(set([(x,min(y+1,c-1)),(min(x+1,r-1),y),(x,max(y-1,0)),(max(x-1,0),y)]))
        actions = ['r','d','l','t']

        return states, actions
```

Listing 1: Domain Class

Part A: Solving for optimal policy

Value Iteration

Algorithm 1: Value Iteration

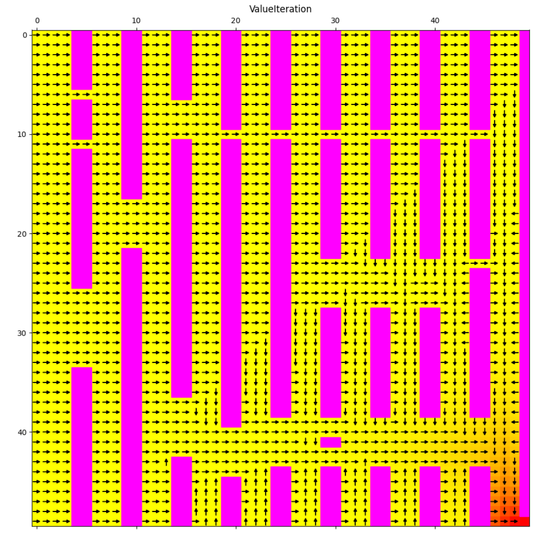
Data: $MDP\{S, A(s), P(s|s', a), R(s|s', a), \gamma\}$, ϵ(maximum error allowed in the utility of any state)

Result: $V(s)$(The optimal utility of each state)

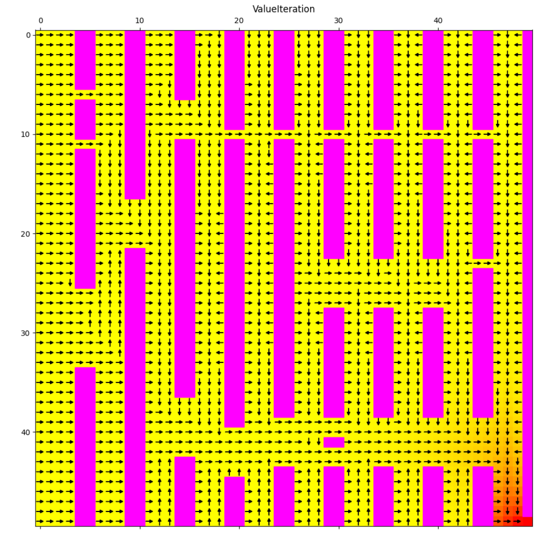
$num\_iterations \leftarrow 0$;
repeat
    $V \leftarrow V'$;
    $\delta \leftarrow 0$;
    for *each state* $s \in S$ **do**
        $V'[s] \leftarrow \max_{a \in A(s)} \sum_{s' \in S} P(s'|s, a)(R(s'|s, a) + \gamma V[s'])$;
        if $|V'[s] - V[s]| > \delta$ **then**
            $\delta \leftarrow |V'[s] - V[s]|$;
        end
    end
    $num\_iterations \leftarrow num\_iterations + 1$;
until $\delta < \epsilon(1 - \gamma)/\gamma$;

For $\epsilon = 10^{-8}$, the algorithm converged to a sensible optimal policy.

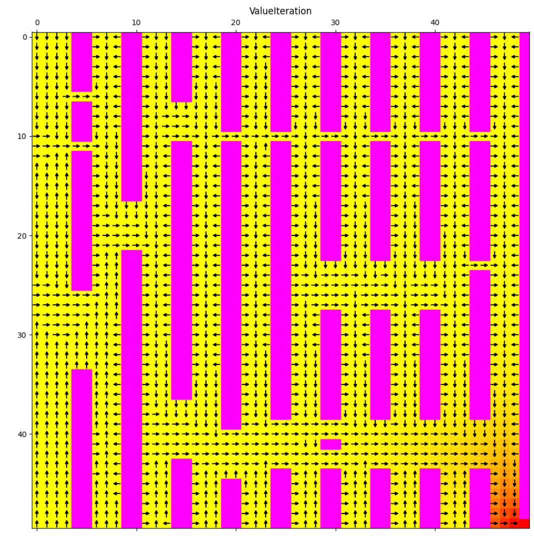
Experiments with ϵ



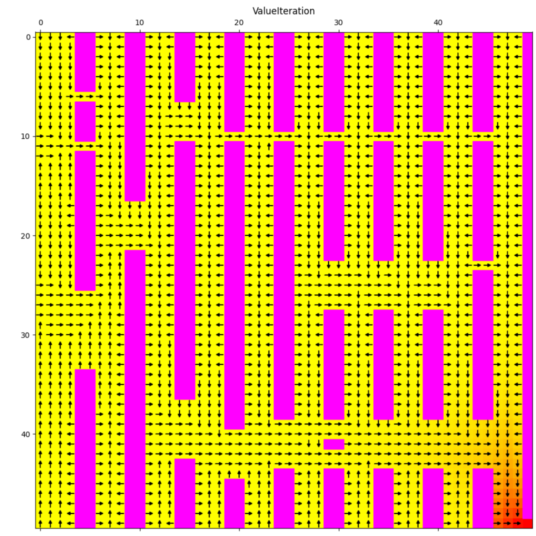
(a) $\epsilon = 10^{-2}$, number of iterations = 43



(b) $\epsilon = 10^{-4}$, number of iterations = 79



(c) $\epsilon = 10^{-6}$, number of iterations = 117



(d) $\epsilon = 10^{-8}$, number of iterations = 152

Figure 1: Optimal policy heatmaps for different values of ϵ

Asynchronous Updates

Row Major Sweep

Algorithm 2: Row Major Sweep

Data: $MDP\{S, A(s), P(s|s', a), R(s|s', a), \gamma\}$, ϵ(maximum error allowed in the utility of any state)

Result: $V(s)$(The optimal utility of each state)

```
r, c ← goal_state;
num_iterations ← 0;
repeat
    δ ← 0;
    for i ∈ {r − 1, r − 2, . . . , 0} do
        for j ∈ {c − 1, c − 2, . . . , 0} do
            s ← S[i][j];
            value ← max_{a∈A(s)} ∑_{s'∈S} P(s'|s, a)(R(s'|s, a) + γV[s']);
            if |value − V[s]| > δ then
                δ ← |value − V[s]|;
            end
            V[s] ← value
        end
    end
    num_iterations ← num_iterations + 1;
until δ < ε(1 − γ)/γ;
```

Prioritized Sweep

Algorithm 3: Prioritized Sweep

Data: $MDP\{S, A(s), P(s|s', a), R(s|s', a), \gamma\}$, ϵ(maximum error allowed in the utility of any state)

Result: $V(s)$(The optimal utility of each state)

```
Initialize priority queue q;
num_updates ← 0;
num_iterations ← 0;
repeat
    s, δ ← q.pop();
    for each state s'|∃a s.t P(s|s', a) > 0 do
        priority(s) ← max_{a∈A(s)} ∑_{s'∈S} P(s|s', a)(R(s|s', a) + γV[s]);
        q.push(s, priority(s))
    end
    num_updates ← num_updates + 1;
    if num_updates%|S| == 0 then
        num_iterations ← num_iterations + 1;
    end
until δ < ε(1 − γ)/γ;
```

Convergence speed analysis between Asynchronous and Synchronous updates

Table 1: Number of Iterations($\epsilon = 10^{-8}$)

Algorithm	Large Map	Small Map
`Value Iteration`	152	33
`Row Major Sweep`	57	19
`Prioritized Sweep`	18	18

We can observe from the table 1 that the convergence speed for vanilla Value Iteration algorithm is significantly higher than it's asynchronous variants. The difference becomes quite notable for the larger grid.

Policy Iteration

Algorithm 4: Policy Evaluation

Data: $MDP\{S, A(s), P(s|s', a), R(s|s', a), \gamma\}$, Policy π ,ϵ(maximum error allowed in the utility of any state)

Result: $V(s)$(The utility of each state following the policy π)

for *each state* $s \in S$ **do**
  $V'[s] \leftarrow 0$
end
repeat
  $V \leftarrow V'$;
  $\delta \leftarrow 0$;
  for *each state* $s \in S$ **do**
    $V'[s] \leftarrow \sum_{s' \in S} P(s'|s, \pi[]))(R(s'|s, \pi[s]) + \gamma V[s'])$;
    if $|V'[s] - V[s]| > \delta$ **then**
      $\delta \leftarrow |V'[s] - V[s]|$;
    end
  end
until $\delta < \epsilon$;

Algorithm 5: Policy Iteration

Data: $MDP\{S, A(s), P(s|s', a), R(s|s', a), \gamma\}$, ϵ(maximum error allowed in the utility of any state)

Result: π(The optimal policy)

1 **for** *each state* $s \in S$ **do**
2     $i \leftarrow rand(0, |A(s)|)$;
3     $\pi[s] \leftarrow a_i$
4 **end**
5 $V' \leftarrow$ `POLICY_EVALUATION`(MDP, π, ϵ);
6 $num\_iterations \leftarrow 0$;
7 **repeat**
8     $V \leftarrow V'$;
9     $\delta \leftarrow 0$;
10   **for** *each state* $s \in S$ **do**
11         $\pi[s] \leftarrow argmax_{a \in A(s)} \sum_{s' \in S} P(s'|s, a)(R(s'|s, a) + \gamma V[s'])$;
12   **end**
13     $V' \leftarrow$ `POLICY_EVALUATION`(MDP, π, ϵ);
14   **for** *each state* $s \in S$ **do**
15         $\delta \leftarrow \delta + |V'[s] - V[s]|$
16   **end**
17     $num\_iterations \leftarrow num\_iterations + 1$;
18 **until** $\delta < \epsilon$;

- **Stopping Criterion:** Lines 14,15,16 and 18 of the Algorithm 5 represents the stopping criterion used to determine the convergence of the policy iteration. After, the policy evaluation step, we take the sum of the change in values for each state and compare it with $\epsilon(10^{-8})$
- We take the sum of the change in the value function of each state because we want to ensure that the policy is stable and the value function converges to the optimal value function. By taking the sum of the changes, we can monitor the overall progress of the algorithm and determine when the policy is stable and the value function has converged to the optimal value function.

Analysis of the Algorithms

Convergence

Table 2: Convergence Metrics

Algorithm	Large Map		Small Map	
	Wall Time (ms)	**Num Iterations**	**Wall Time (ms)**	**Num Iterations**
`Value Iteration`	30192.57	152	47.32	33
`Row Major Sweep`	11863.34	57	30.48	19
`Priority Sweep`	20679.42	18	109.42	18
`Policy Iteration`	70173.51	8	62.92	3

- **Priority Sweep:** This algorithm has the lowest number of iterations (18[1]) among the tested algorithms for the larger environment. However, it's wall time (20679.42 ms) is significantly more than the wall time of Row-Major Sweep (11863.34 ms). The high wall

[1]Note how num_iterations is defined in each algorithm.

time may pose scalability challenges for extremely large environments. However, it still seems more scalable than Value Iteration and Policy iteration in terms of time efficiency.

- **Row Major Sweep:** This algorithm has the lowest wall time (20679.42 ms) among the tested algorithms for the larger environment. Additionally, it requires a relatively fewer iterations (57), indicating a good balance between time efficiency and convergence speed. This suggests that Row Major Sweep may be more scalable for larger environments compared to Value Iteration and Policy Iteration.

- **Value Iteration:** While Value Iteration has a moderate wall time (30782.82 ms) for the larger environment, it requires a reasonable number of iterations (152). However, it seems less efficient than Row Major Sweep based on both wall time and number of iterations.

- **Policy Iteration:** Policy Iteration has the highest wall time (70173.51 ms) among all the algorithms for the larger environment, indicating poor scalability in terms of execution time. Despite requiring a relatively low number of iterations (8), the high wall time suggests inefficiency in handling larger environments.

Based on these observations, **Row Major Sweep** appears to be the most scalable algorithm for larger environments among the ones provided. It strikes a good balance between wall time and number of iterations, indicating efficient convergence for larger problem sizes. However, if we are working in a finite and small policy space, Policy Iteration might be a better choice since it is guaranteed to converge in a finite number of iterations. Value iteration(and its variants) does not have the same guarantees.

Start state estimated value plots

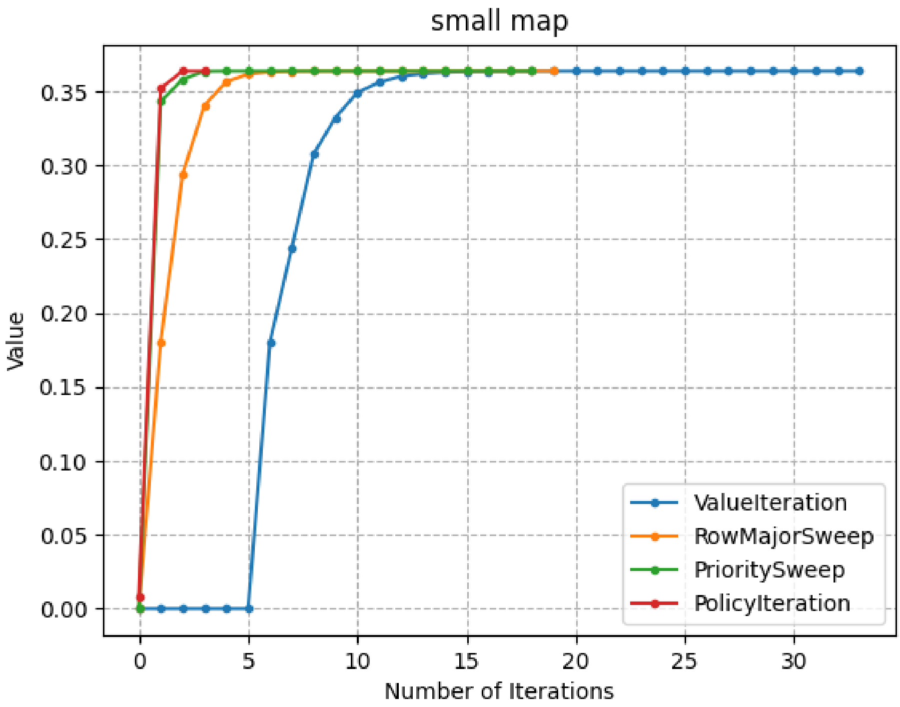


Figure 2: Line graph showing the evolution of the value of the starting state for small map

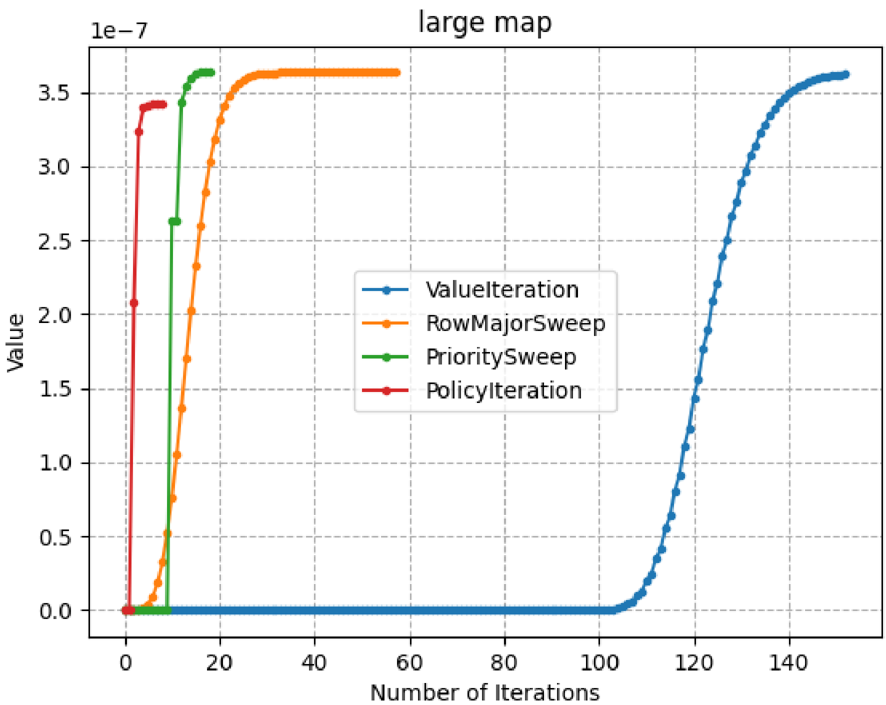
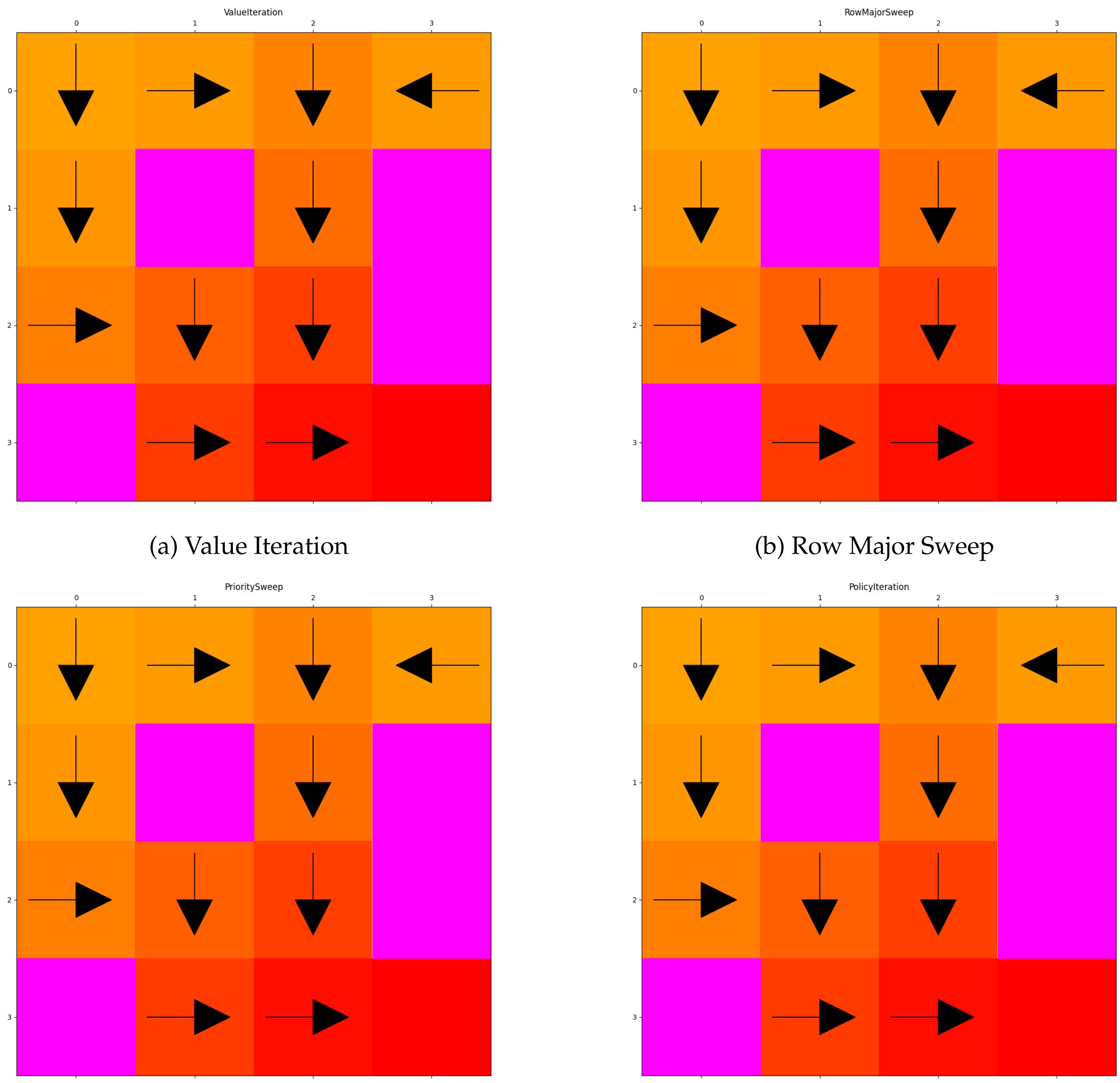


Figure 3: Line graph showing the evolution of the value of the starting state for large map

Heat Maps



(a) Value Iteration (b) Row Major Sweep

(c) Prioritized Sweep (d) Policy Iteration

Figure 4: Small Grid Heat Maps

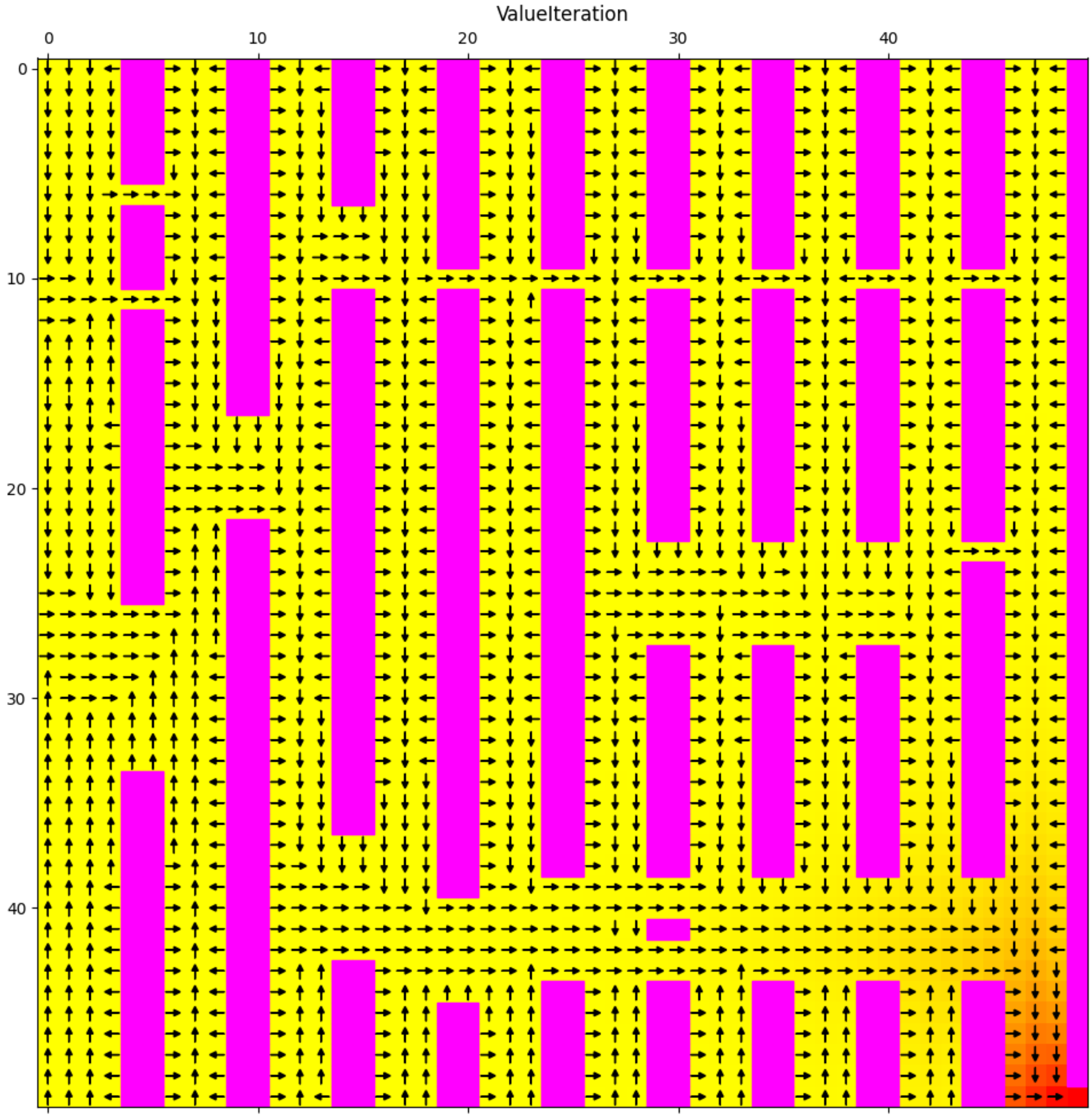


Figure 5: Value Iteration

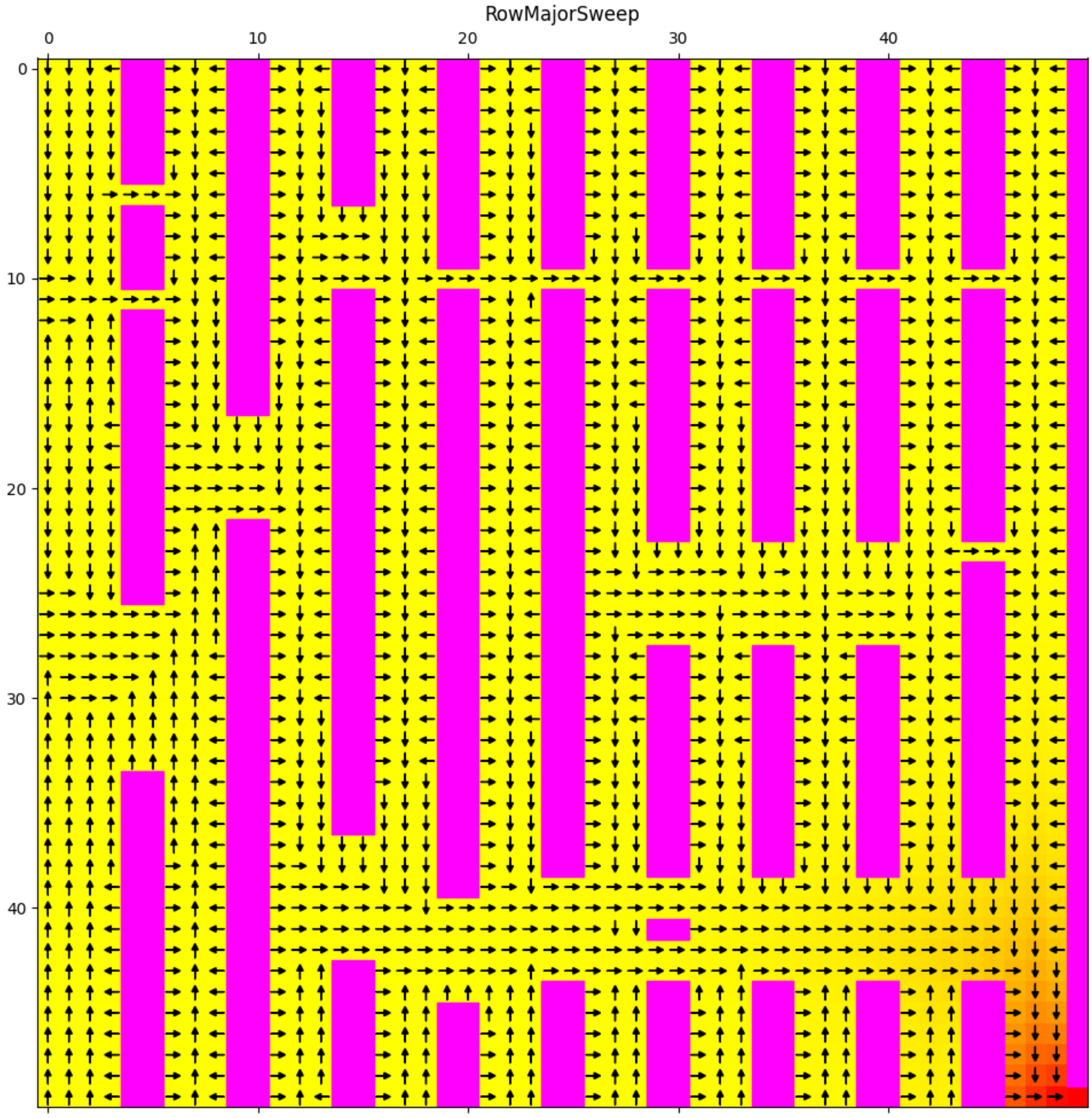


Figure 6: Row Major Sweep

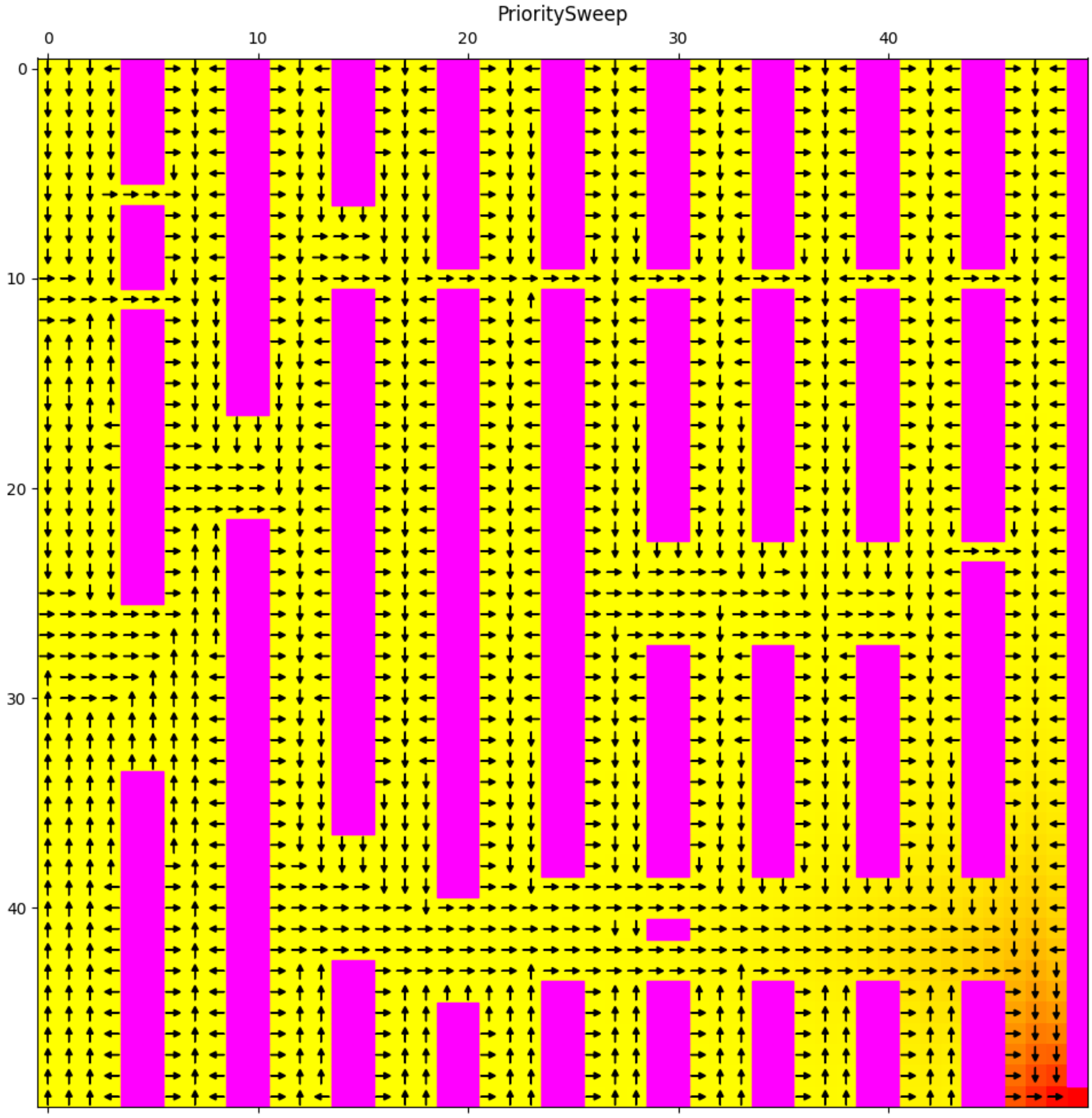


Figure 7: Prioritized Sweep

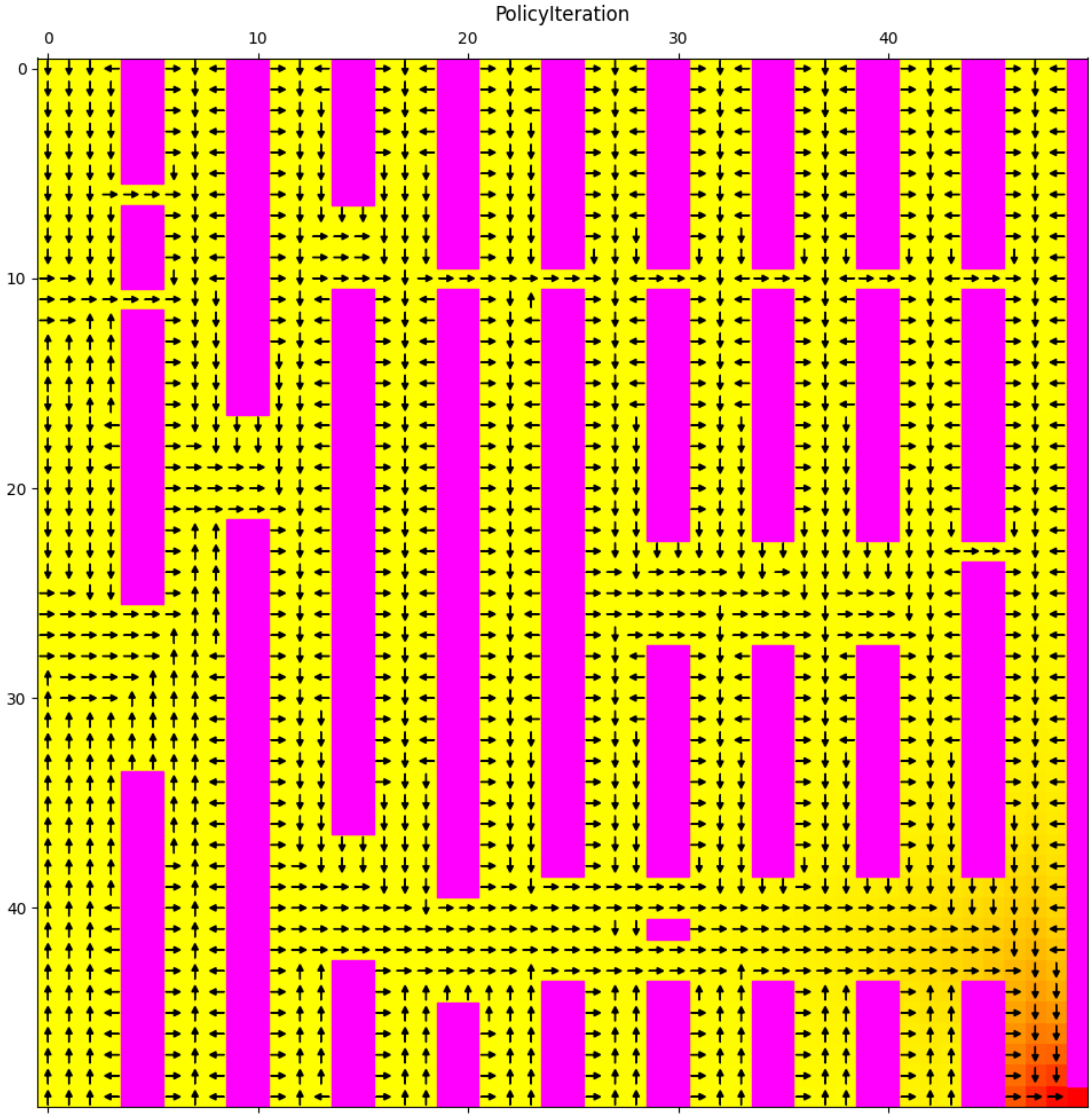


Figure 8: Policy Iteration

From Figure 4, 5, 6, 7 and 8 we observe that all the algorithms converge to the same optimal policy on both the maps. Therefore, no algorithm is better than the other in terms of optimal policy.

Part B: Analysis

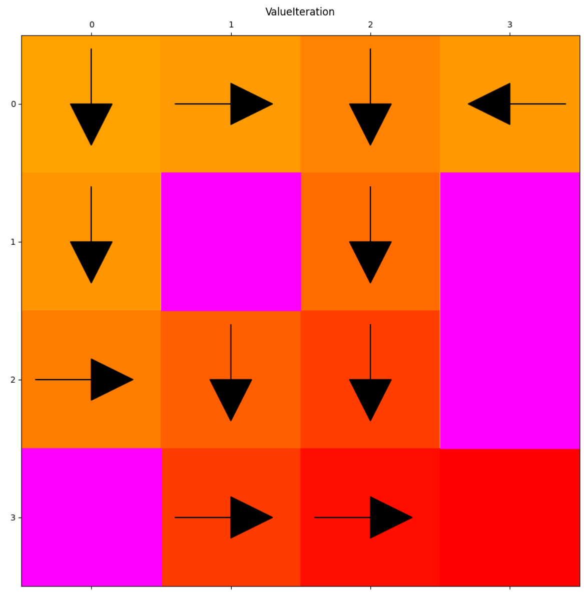
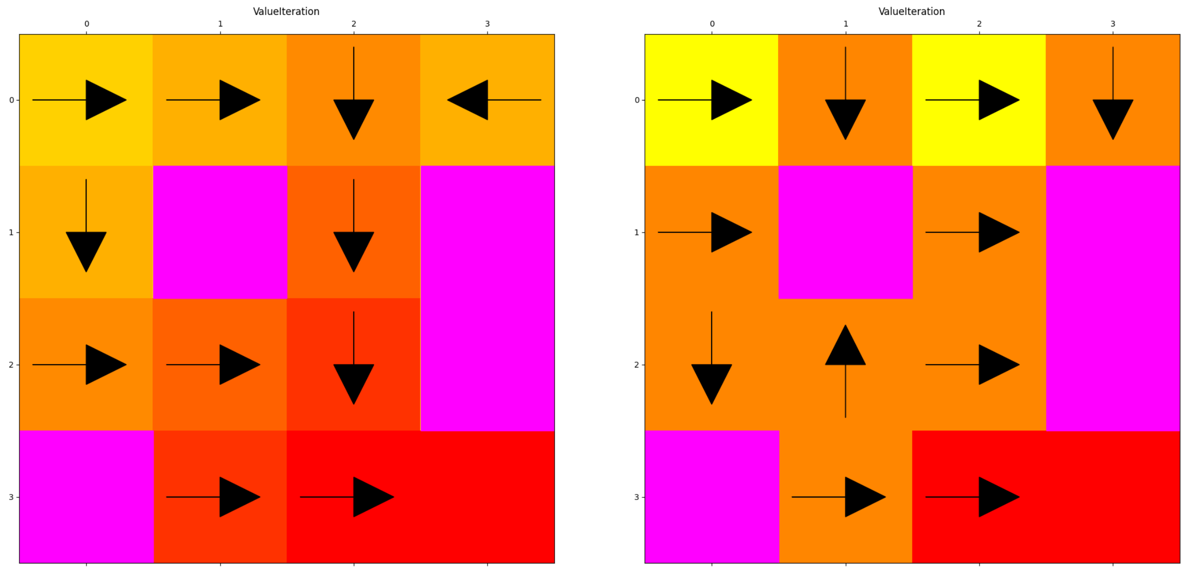


Figure 9: Default Setting Policy

B1. Living reward analysis

Negative Living Rewards



(a) living reward = -0.1 (b) living reward = -0.9

Figure 10: Converged Policies with negative living rewards

- When comparing the default setting policy (Figure 9) with a policy where the living reward is set to -0.1 (Figure 10a), it's observed that the converged values of the initial few states are lower in the latter case, as indicated by lighter colors in the heatmap representation. This observation aligns with expectations since in the modified policy, the agent receives a negative reward in each free state, whereas in the default setting, the agent received a reward of 0. Consequently, the overall reward accumulated by the agent during its exploration of the environment is reduced.

- A more significant change is observed when the living reward is further decreased to -0.9. In this scenario, the converged policy (Figure 10b) indicates a distinct behavior where, from every free state, the agent tends to move towards neighboring terminal hole states. This behavior stems from the fact that the expected discounted reward that the agent receives from any free state (except those close to the goal state) becomes significantly less than the rewards obtained from the hole states, as the reward in the hole states is kept unchanged at 0. Consequently, the agent prioritizes moving towards the terminal hole states, despite them not being the goal state, as it perceives them as more beneficial in terms of expected future rewards.

Positive Living Rewards

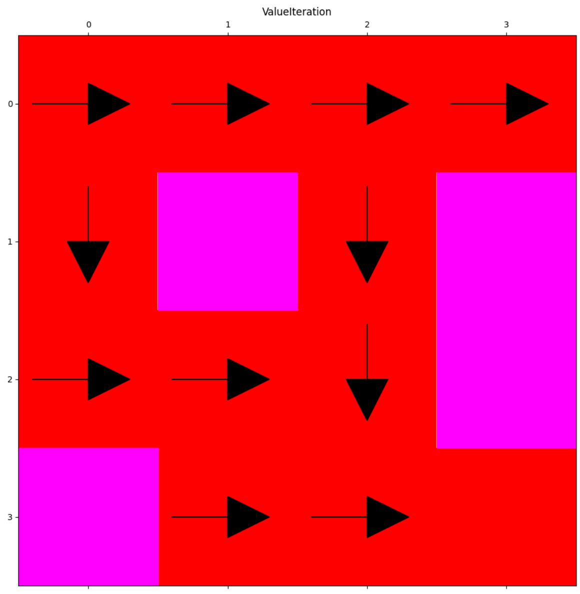


Figure 11: living reward = 0.001, $\gamma = 0.999$

- From Figure 11 we observe that the reward setting in this case is such that all the free states converged to the same value of 1.
- When the reward model in a MDP is such that after running value iteration, all states converge to the same value, it suggests that the reward model is not providing enough discrimination between states. In other words, the rewards are not providing enough information to distinguish between different states in terms of their value or desirability.
- In this scenario, all states becoming equivalent means that regardless of the state the agent is in, it expects the same cumulative reward over the long term. Consequently, the agent does not have any preference for one state over another, as they all provide the same expected return. As a result, the algorithm converges to a randomly picked policy by the agent.

B2. Changing transition probabilities

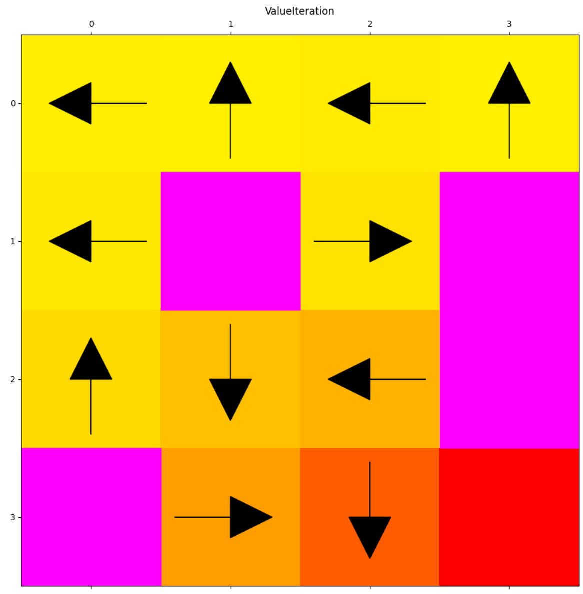


Figure 12: Converged Policy

From Figure 12 we observe that the transition model is such that in every state, the agent chooses the action that has the least probability of ending up in a terminal hole state. For example, in the state (1,2), the agent chooses the 'right' action which has a 33% chance of going into the hole state, where as the actions 'down' and 'top' have 66% chance of going into the hole state. Thus in every free state, the agent selects the action that maximizes the likelihood of remaining in the current state or transitioning to a non-hole state.