Strategic Decision-Making in Stochastic Environments: A Comprehensive Analysis of Predator-Prey Dynamics Through Dynamic Games Using Markov Decision Processes and Linear Algebra

Tatyana Ilieva, Anastasiia Merkudanova, Sukhman Parhar, Shreyas Rajaram, Blake Starling

MATHUN2015 Linear Algebra and Probability Fall 2025

1. Introduction

Sequential decision-making in uncertain and dynamic environments is a central problem across mathematics, economics, biology, and artificial intelligence. Many real-world interactions, such as animals competing for survival, security systems adapting to intruders, or autonomous vehicles avoiding collisions, can be modeled as dynamic games (Fang et al., 2021; Chen et al., 2014). In these settings, outcomes depend not only on the choices made in the current moment, but also on how those choices influence future states of the system. The central question we address is: How can we characterize optimal behavior when an agent's actions influence not merely immediate outcomes, but also the probabilistic evolution of the system's future states, particularly when multiple strategic agents interact simultaneously?

This question is especially relevant in ecological systems, where predator-prey interactions exhibit complex feedback loops between current strategic choices and environmental states. This study investigates predator–prey interactions through the lens of Markov Decision Processes (MDPs) and stochastic game theory (Kallenberg, 2011; Littman, 1993), as such interactions cannot be sufficiently modeled using static game theory alone. The repeated nature of the conflict and the stochastic state transitions create path dependencies that alter optimal strategies.

The predator and prey system serves as our primary application domain, modeled within a probabilistic environment defined by a finite set of four states: Far, Near, Caught, and Escaped. (Harten et al., 2019). Caught and Escaped are absorbing states, which create a natural terminal condition for the stochastic process. The predator seeks to maximize long-term rewards by choosing between actions such as chase or wait, while the prey acts to minimize the predator's success by hiding or running. Because the next state depends only on the current state and the actions of the predator/prey, the system fits into a Markov framework. Ultimately, we show that optimal behavior in dynamic predator–prey systems emerges from the interaction of transition structure, long-run rewards, and strategic response.

Linear algebra plays a central role in this analysis. The state transitions are encoded in transition matrices, rewards are represented as vectors, and optimal long-term values are found by solving the Bellman equation:

$$V_{\alpha} = R + \gamma P_{\alpha} V_{\alpha}$$

where V is the value function, P is the transition matrix, R is the reward vector, and gamma is the discount factor (Kallenberg, 2011; Ding et al., 2020). Linear algebra is connected to dynamic decision-making through concepts such as matrix inversion, eigenvalues, and absorbing Markov chains.

The goals of this project are:

1. Construct a one-player MDP model for the predator and compute optimal policies.

2. Extend the model to a two-player dynamic game, where predator and prey choose strategies simultaneously.

3. Analyze how linear algebra tools reveal equilibrium behavior.

2. Background and Game Formulation

2.1 Markov Chains and State Representation

A Markov chain is a stochastic process in which the probability of transitioning to the next state depends only on the current state and not on earlier history (Kallenberg, 2011; Ankan & Panda, 2018). This is known as the Markov property, and it makes these systems mathematically tractable and ideal for modeling dynamic environments with probabilistic outcomes.

We represent our predator–prey environment using a finite set of four states:

1. Far: The predator is far from the prey. This is advantageous for the prey.
2. Near: The predator is near the prey. This is advantageous for the predator.
3. Caught: The predator has caught the prey. This is a terminal state representing success for the predator
4. Escaped: The prey escaped. This is a terminal state representing success for the prey

States 3 and 4 are absorbing states, meaning the system remains in those states with probability 1. Absorbing Markov chains are central to our analysis because long-term outcomes (success vs. failure) are encoded through them (Kallenberg, 2011).

The transition probabilities between states are represented by a transition matrix:

$$P = \begin{matrix} p_{00} & p_{01} & p_{02} & p_{03} \\ p_{10} & p_{11} & p_{12} & p_{13} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

$p_{ij} = P(X_{t+1} = J | X_t = i), \sum_j p_{ij} = 1$

Each sums to 1, which is the total probability.

2.2 Rewards and the Discount Factor

In an MDP, each state is assigned a reward reflecting the desirability of being in that state. For the predator, we define:

State	Reward	Meaning
Far	–2	Predator is frustrated and making little progress.
Near	–1	Predator is closer but has not succeeded.
Caught	5	Successful capture.
Escaped	–5	Prey permanently escapes.

These rewards form the reward vector:

$$R = \begin{matrix} -2 \\ -1 \\ 5 \\ -5 \end{matrix}$$

Future rewards are discounted by a factor $\gamma \in (0,1)$, which models the idea that immediate success is more valuable than success in the future. This is the standard in MDP, where discounting ensures that the total reward remains finite. (Kallenberg, 2011).

In our model, we set $\gamma = 0.9$.

The literature supports this as it preserves long-term reward structure while favoring immediate outcomes. The specific value of γ is application-dependent; using values close to 1 is standard practice (Ding et al., 2020; Kallenberg, 2011).

2.3 The Bellman Equation and Linear Algebra

The central mathematical equation of an MDP is the Bellman equation, which expresses the value of a state as immediate reward plus expected discounted future reward:

$$V_\alpha = R + \gamma P_\alpha V_\alpha$$

We can then rearrange to solve:

$$(I - \gamma P_\alpha)V_\alpha = R$$
$$V_\alpha = (I - \gamma P_\alpha)^{-1} R \,.$$

This is where linear algebra is needed:

- $I - \gamma PI$ must be invertible.
- Solving for V requires matrix inversion or solving a linear system.
- The structure of P dominates the behavior of V.

The Bellman equation allows us to evaluate how good each state is under a given policy or under strategic play.

2.4 One-Player Game Formulation (Predator Only)

In the one-player case, the predator chooses between two actions:

1. Chase
2. Wait

Each action has its own transition matrix:

$$P_{chase}, P_{wait}$$

Evaluating each policy means computing:

$$V_{chase} = (I - \gamma P_{chase})^{-1} R,$$
$$V_{wait} = (I - \gamma P_{wait})^{-1} R.$$

These value functions determine which action is better in each state.

2.5 Two-Player Game Formulation (Predator vs Prey)

In the two-player extension, both predator and prey choose actions **simultaneously**:

Predator Actions	**Prey Actions**
Chase	Hide
Wait	Run

Each pair of actions produces a different transition matrix:

$P_{chase,\, hide}$
$P_{chase,\, run}$
$P_{wait,\, hide}$
$P_{wait,\, run}$

If the predator uses strategy p (probability of Chase) and the prey uses strategy h (probability of Hide), the resulting transition matrix is the weighted combination:

$$P(p, h) = phP_{chase,\, hide} + p(1 - h)P_{chase,\, run} + (1 - p)hP_{wait,\, hide} + (1 - p)(1 - h)P_{wait,\, run}$$

The value of a given strategy pair is:

$$V(p, h) = (I - \gamma P(p, h))^{-1}R,$$

To find equilibrium strategies, we compute Q-values for each action pair and solve a local 2×2 Nash equilibrium at each state.

This creates a stochastic game, where equilibrium behavior emerges from linear algebra and iterative best responses.

3. One-Player Predator MDP

To model the one-player MDP, we used a Python script, where the transition matrices and reward array were constructed using numpy arrays. For each state, we calculated the value of performing a transition (predator action) using the formula derived from the Bellman equation in section 2.3. See the script below:

```
Python
import numpy as np

# SETUP
gamma = 0.9 # discount factor
states = ["Far", "Near", "Caught", "Escaped"] # states to print later

  # reward vector (far, near, caught, escaped)
R = np.array([
  -2,
  -1,
  5,
  -5
])

  # transition matrices
    # chase
P_chase = np.array([
  [0.6, 0.4, 0, 0],   # from Far
  [0, 0.5, 0.5, 0],   # from Near
  [0, 0, 1, 0],       # from Caught
  [0, 0, 0, 1]        # from Escaped
])
    # wait
P_wait = np.array([
  [0.8, 0.2, 0, 0],   # from Far
  [0.3, 0.6, 0, 0.1], # from Near
  [0, 0, 1, 0],       # from Caught
  [0, 0, 0, 1]        # from Escaped
])

  # bellman equation using V = (I - gamma * P)^-1 x R
def bellman(gamma, P, R):
  I = np.identity(P.shape[0]) # identity matrix, I
  return np.linalg.inv(I - gamma * P) @ R # return V
  # helper function to display values
def display_values(chase_values, wait_values):
  print(f"{'State':<10} | {'Chase Value':<15} | {'Wait Value':<15}") # table headers
  print("-" * 45)
  for i, state in enumerate(states):
    print(f"{state:<10} | {chase_values[i]:<15.2f} | {wait_values[i]:<15.2f}") # values
```

```
# MAIN CODE
if __name__ == "__main__":
    chase_values = bellman(gamma, P_chase, R)
    wait_values = bellman(gamma, P_wait, R)

    display_values(chase_values, wait_values)
```

4. One-Player Results

The script from section 3 produced the following:

```
None
State      | Chase Value     | Wait Value
--------------------------------------------
Far        | 26.25           | -17.76
Near       | 39.09           | -16.51
Caught     | 50.00           | 50.00
Escaped    | -20.00          | -20.00
```

Larger values are preferred for the predator, while smaller numbers are less desirable. As shown, for both near and far states, the value of chasing was greater than the value of waiting, so the optimal strategy for the predator in both states would be to chase. Values in caught and escaped states are trivial because both of these states are terminal (the game has ended), but, as expected, the caught state is always positive, and the escaped state is always negative because the predator is rewarded by catching the prey and punished when the prey escapes.

5. Two-Player Dynamic Game Formulation

Similarly to the one-player MDP, for the two-player dynamic game, numpy was used to create transition matrices. This time, however, 4 transition matrices represented the 4 combinations of strategies (chase/hide, chase/run, wait/hide, and wait/run).

Unlike the previous game, in this game the goal was to find equilibrium strategies. In order to do so, transition matrices had to be evaluated simultaneously, with the policies of the players being a key influence in the evaluations. To do this, we created a function (get_transition_matrix) that constructed a combined transition matrix. This matrix was a weighted combination of all 4 matrices for a specific strategy, with weights determined by the chance of the matrix occurring (i.e., the chances of chasing/waiting and hiding/running). We were then able to create another function (evaluate_policies) to evaluate the policies of the players in one go. This was done using the function derived from the Bellman equation, where the i-th row of the P matrix was constructed using the i-th row of the get_transition_matrix function's output, with arguments for the function being the i-th strategies of each player. We did this to ensure that states were aligned across policies and the P matrix.

Then, we wrapped everything up in the update_policies function, which updated the policies of players given an evaluation of the policies. Using the evaluation (V), it found q-values for each of the strategy pairs (transition matrices) for each state. With the determined values, the optimal strategy for a state was found for both players by solving the nash equilibrium (solve_nash function) of a matrix of the q-values. The solve_nash function was just a check for a dominant strategy, and then a calculation of a mixed strategy to "counter" the opponent if no dominant strategy existed.

Finally, we obtained results by setting default policies and looping to evaluate the policies and update them until an equilibrium was found. We did this for two different scenarios: one with no mixed strategies and one with mixed strategies. The difference in outcomes resulted from modifications to the transition matrices, where some strategy combinations were made more or less punishing to the predator. See the script for no mixed strategies below, followed by the transition matrices used to find mixed strategies:

```python
import one_player as op
import numpy as np

# SETUP
gamma = op.gamma
states = op.states
R = op.R

    # transition matrices
        # chase (predator) and hide (prey)
P_chase_hide = np.array([
    [0.6, 0.4, 0, 0],       # from far
    [0.1, 0.8, 0.1, 0],     # from near
    [0, 0, 1, 0],           # prey caught
    [0, 0, 0, 1]            # prey escaped
])
        # chase (predator) and run (prey)
P_chase_run = np.array([
    [0.2, 0.1, 0, 0.7],     # from far
    [0.2, 0.1, 0.1, 0.6],   # from near
    [0, 0, 1, 0],           # prey caught
    [0, 0, 0, 1]          # prey escaped
])
        # wait (predator) and hide (prey)
P_wait_hide = np.array([
    [0.9, 0.1, 0, 0],      # from far
    [0.1, 0.9, 0, 0],      # from near
    [0, 0, 1, 0],          # prey caught
    [0, 0, 0, 1]           # prey escaped
])
        # wait (predator) and run (prey)
P_wait_run = np.array([
    [0.6, 0, 0, 0.4],      # from far
    [0.8, 0, 0, 0.2],      # from near
    [0, 0, 1, 0],          # prey caught
    [0, 0, 0, 1]           # prey escaped
])

transition_matrices = [P_chase_hide, P_chase_run, P_wait_hide, P_wait_run] # create a list of
matrices for use in functions

    # helper function to get a single transition matrix by combining multiple weighted transition
matrices
def get_transition_matrix(chase_prob, hide_prob, transition_matrices):
    wait_prob = 1 - chase_prob
    run_prob = 1 - hide_prob

    # return a weighted combination of matrices
    return (chase_prob * hide_prob * transition_matrices[0] + # weighted chase-hide transition
matrix
            chase_prob * run_prob * transition_matrices[1]  + # weighted chase-run transition matrix
            wait_prob * hide_prob * transition_matrices[2]  + # weighted wait-hide transition matrix
            wait_prob * run_prob * transition_matrices[3]     # weighted wait-run transition matrix
    )

    # evaluate (return V) a pair of policies. Policies are assumed to be lists with probabilities of
primary
    # actions (chase/hide) at different states: [far, near, caught, escaped]
def evaluate_policies(pred_strategy, prey_strategy, transition_matrices):
    P = np.zeros((4,4)) # empty 4x4 matrix that will be populated with state-specific rows
    for i in range(4):
        P[i] = get_transition_matrix(pred_strategy[i], prey_strategy[i], transition_matrices)[i] # set
the i-th row of P to the
        # corresponding row in the matrix. Starts with far, then near, then caught, and finally escaped

    I = np.identity(P.shape[0]) # identity matrix, I
    return np.linalg.inv(I - gamma * P) @ R # return V
    # return new optimal strategies for the current step

    # helper function that returns Nash equilibrium strategies from a matrix of payoffs
def solve_nash(q_matrix):
    chase_prob = 0
    hide_prob = 0
    # unpack matrix
    q_chase_hide = q_matrix[0, 0]
    q_chase_run = q_matrix[0, 1]
    q_wait_hide = q_matrix[1, 0]
    q_wait_run = q_matrix[1, 1]

    # find optimal predator action
        # check for dominant strategies (one strategy is always better or equal)
    if (q_chase_hide >= q_wait_hide) and (q_chase_run >= q_wait_run):
        chase_prob = 1 # always chase
    elif (q_chase_hide <= q_wait_hide) and (q_chase_run <= q_wait_run):
        chase_prob = 0 # always wait
    else: # there is no action that will always lead to an optimal outcome, so try to statistically
counter the prey
         # do this by setting up the equality: q if prey hides = q if prey runs
         # => p * q_chase_hide + (1 - p) * q_wait_hide = p * q_chase_run + (1 - p) * q_wait_run
         # => p = (q_wait_run - q_wait_hide) / ((q_chase_hide + q_wait_run) - (q_chase_run +
q_wait_hide))
        chase_prob = (q_wait_run - q_wait_hide) / ((q_chase_hide + q_wait_run) - (q_chase_run +
q_wait_hide))

    # find optimal prey action
        # check for dominant strategies
    if (q_chase_hide <= q_chase_run) and (q_wait_hide <= q_wait_run):
        hide_prob = 1 # always hide
    elif (q_chase_hide >= q_chase_run) and (q_wait_hide >= q_wait_run):
        hide_prob = 0 # always run
    else: # counter predator: q if predator chases = q if predator waits
        # => p * q_chase_hide + (1 - p) * q_chase_run = p * q_wait_hide + (1 - p) * q_wait_run
        # => p = (q_wait_run - q_chase_run) / ((q_chase_hide + q_wait_run) - (q_chase_run +
q_wait_hide))
        hide_prob = (q_wait_run - q_chase_run) / ((q_chase_hide + q_wait_run) - (q_chase_run +
q_wait_hide))

    return np.clip(chase_prob, 0, 1), np.clip(hide_prob, 0, 1)

    # return updated policies for predator and prey using a value matrix
def update_policies(V):
    pred_strategy = [0, 0, 0, 0]
    prey_strategy = [0, 0, 0, 0]

    # calculate q-values for each state
    for i in range(4):
        # calculate q-values for each strategy combination (chase/hide, for example) using q =
immediate reward (R) + future reward
        q_chase_hide = R[i] + gamma * (transition_matrices[0][i] @ V)   # chase/hide transition
matrix
         q_chase_run = R[i] + gamma * (transition_matrices[1][i] @ V)    # chase/run transition
matrix
        q_wait_hide = R[i] + gamma * (transition_matrices[2][i] @ V)    # wait/hide transition
matrix
        q_wait_run = R[i] + gamma * (transition_matrices[3][i] @ V)     # wait/run transition matrix

        # construct a matrix to solve (nash)
        q_matrix = np.array([
            [q_chase_hide, q_chase_run],
            [q_wait_hide, q_wait_run]
        ])

        chase_prob, hide_prob = solve_nash(q_matrix) # solve the matrix

        # update strategies
        pred_strategy[i] = chase_prob
        prey_strategy[i] = hide_prob

    return pred_strategy, prey_strategy
```

```
# MAIN CODE
if __name__ == "__main__":
    # set initial policies
    pred_policy = np.array([0.75, 1, 0, 0])
    prey_policy = np.array([0.75, 0.25, 0, 0])
    print()

    # create table
    print(f"{'Iteration':<10} | {'Chase (Far)':<15} | {'Chase (Near)':<15} | {'Hide (Far)':<15} |
{'Hide (Near)':<15}")
    print("-" * 80)
    print(f"{0:<10} | {pred_policy[0]:<15.2f} | {pred_policy[1]:<15.2f} | {prey_policy[0]:<15.2f} |
{prey_policy[1]:<15.2f}")
    for i in range(10):
        V = evaluate_policies(pred_policy, prey_policy, transition_matrices)
        pred_policy, prey_policy = update_policies(V)
        print(f"{i + 1:<10} | {pred_policy[0]:<15.2f} | {pred_policy[1]:<15.2f} |
{prey_policy[0]:<15.2f} | {prey_policy[1]:<15.2f}")
    print()
```

Transition matrices used to find mixed strategies:

```
Python
P_chase_hide = np.array([
    [0.6, 0.4, 0, 0],       # from far
    [0.1, 0.6, 0.3, 0],     # from near
    [0, 0, 1, 0],           # prey caught
    [0, 0, 0, 1]            # prey escaped
])
        # chase (predator) and run (prey)
P_chase_run = np.array([
    [0, 0.1, 0, 0.9],       # from far
    [0, 0.1, 0.1, 0.8],     # from near
    [0, 0, 1, 0],           # prey caught
    [0, 0, 0, 1]            # prey escaped
])
        # wait (predator) and hide (prey)
```

```
P_wait_hide = np.array([
    [0.9, 0.1, 0, 0],       # from far
    [0.1, 0.9, 0, 0],       # from near
    [0, 0, 1, 0],          # prey caught
    [0, 0, 0, 1]           # prey escaped
])
        # wait (predator) and run (prey)
P_wait_run = np.array([
    [0.2, 0.5, 0, 0.3],     # from far
    [0.3, 0.4, 0.2, 0.1],   # from near
    [0, 0, 1, 0],          # prey caught
    [0, 0, 0, 1]           # prey escaped
])
```

6. Two-Player Results

The first script from section 5 produced the following:

```
None
Iteration  | Chase (Far)   | Chase (Near)   | Hide (Far)    | Hide (Near)
-----------------------------------------------------------------------------
0         | 0.75         | 1.00         | 0.75         | 0.25
1         | 0.00         | 1.00         | 0.00         | 0.00
2         | 0.00         | 1.00         | 0.00         | 0.00
3         | 0.00         | 1.00         | 0.00         | 0.00
4         | 0.00         | 1.00         | 0.00         | 0.00
5         | 0.00         | 1.00         | 0.00         | 0.00
6         | 0.00         | 1.00         | 0.00         | 0.00
7         | 0.00         | 1.00         | 0.00         | 0.00
8         | 0.00         | 1.00         | 0.00         | 0.00
9         | 0.00         | 1.00         | 0.00         | 0.00
10         | 0.00         | 1.00         | 0.00         | 0.00
```

In this scenario, dominant strategies arose for both players across near and far states. For the predator in the far state, waiting was the dominant strategy. In the near state, it was to chase. For the prey, running was the dominant strategy for both states.

By swapping the transition matrices with the code block from section 5, the following was produced:

None
```
Iteration  | Chase (Far)    | Chase (Near)   | Hide (Far)     | Hide (Near)
--------------------------------------------------------------------------------
0          | 0.75           | 1.00           | 0.75           | 0.25
1          | 0.00           | 0.27           | 0.00           | 0.48
2          | 0.00           | 0.12           | 0.00           | 0.61
3          | 0.00           | 0.10           | 0.00           | 0.63
4          | 0.00           | 0.09           | 0.00           | 0.63
5          | 0.00           | 0.09           | 0.00           | 0.63
6          | 0.00           | 0.09           | 0.00           | 0.63
7          | 0.00           | 0.09           | 0.00           | 0.63
8          | 0.00           | 0.09           | 0.00           | 0.63
9          | 0.00           | 0.09           | 0.00           | 0.63
10         | 0.00           | 0.09           | 0.00           | 0.63
```

In this scenario, two dominant and two mixed strategies were found. For the predator in the far state, waiting was the dominant strategy. In the near state, there was no dominant strategy, so the predator countered the prey by chasing 9% of the time and waiting 91% of the time. For the prey in the far state, running was the dominant strategy. In the near state, there was no dominant strategy, so the prey countered the predator by hiding 63% of the time and running 37% of the time.

7. Discussion

The analysis shows important findings related to optimal behavior. First, we can conclude that the optimal behavior is driven by long-run expectations rather than immediate rewards. Even though we defined "Near" with a corresponding reward of -1, the "Chase" action is the most optimal, as it increases the chance of transitioning to the "Caught" state, which yields the greatest reward.

Second, we can conclude that the transition probabilities strongly shape the value of each action. The modeling accuracy of our game is critical, as a shift in any of the probabilities that influence the action of the prey could alter the policy conclusions.

It is also important for us to consider that in the one-player (predator only) model, the environment relies on external factors. However, the two-player model allows the prey to make choices, which ultimately leads to a Markov Perfect Equilibrium. However, there may be potential divergence between the game-theoretical model and MDP-optimal policies. This is because MDP-optimal policies assume that the environment is not strategic, and that equilibrium policies reflect best mutual responses.

7.1 Why Do Pure Strategies Emerge?

In our results, we see a pure strategy emerge instead of mixed strategies. This is because the structure of the game does not incentivize randomness; instead, we see that in each state, there is one action that is more favorable than the other:

- When Near waiting gives a worse result because it increases the chance of the prey escaping.
- When Far chasing provides little benefit because the prey's stochastic movement tends to draw the system to near, regardless of what the predator does. This means that the risk of the prey escaping when the predator chooses to chase when far is too high.

Given these strong outcomes, the players do not benefit from mixing their actions; rather, the game simplifies to a deterministic strategy.

7.2 Limitations

1. **Simplified geometry:**
    a. The model uses only two distances (Far, Near). A grid or continuous distance would be more realistic.
    b. Transition probabilities are static in time, can implement some exponential decay to imitate likelier prey transition Near -> Far the longer the predator does not catch it (i.e. predator running out of energy or having to sleep)

2. **Fixed reward structure:**
    a. Prey could be given its own reward vector, making the game fully two-sided.
    b. Reward structure could be time dependent (i.e. more negative utility to predator the longer it does not eat).

3. **Deterministic prey preferences:**
    a. In real life, prey would have mixed strategies. In our project, our transition probabilities encourage equilibrium.

4. **No learning:**
    a. We compute equilibrium directly; reinforcement learning could allow predator and prey to adapt.

7.3 Possible Extensions

1. Introduce a prey reward vector.
2. Use more states (Far, Mid, Near, Very Near).
3. Use dynamic (in time) transition probabilities.
4. Implement learning into policies.
5. Add stochastic/dynamic rewards or penalties.

6. Model multiple predators or multiple prey.

8. Conclusion

Our project concludes that for the predator, the aggressive pursuit, "Always Chase," is the most optimal strategy for the one-player MDP. The structure of the transition probabilities shows that the waiting action is dominated in each nonterminal state. By choosing to always chase, the predator receives substantially higher long-term payoff accumulation as opposed to waiting. This can likely be made non-Pareto by disincentivizing the chase with a time-increasing disutility to the predator or by linking the terminal "escape" state to Near/Far while chasing.

At a deeper level, our findings highlight how MDP-optimal actions are not only shaped by reward magnitudes, but more importantly, by the directional flow of transition probabilities. For example, the "Chase" action creates a high-probability forward path, e.g., Far → Near → Caught, which results in a fast convergence to positive absorbing outcomes. Waiting, however, creates stagnation, trapping the predator in an unfavorable region of the state space, giving the prey opportunities to escape. Therefore, the dominance of the chase policy does not only reflect the anticipation of immediate reward, but rather is a consequence of the long-run stochastic environment. As we transition to a two-player stochastic game, the interaction of two agents' decisions transforms our system into coupled Bellman equations, which is a multi-dimensional fixed-point problem whose solutions correspond to the Markov Perfect Equilibrium. In this scenario, the interplay between the two agents creates a higher-dimensional linear system where equilibrium arises from mutually consistent value vectors instead of unilateral optimization.

We must consider that our conclusion of "Always Chase" as the optimal action is based on a simplified model, where there are no risk-based penalties for aggressive pursuit, and no energy or fatigue-related penalties. Introducing these nuances would shift the radius of γP^{α}, which therefore changes the long-run directional tendencies of the process. Additional nuance would also introduce opportunity costs into the environment, potentially resulting in more complex optimal pursuits.

In sum, our analysis provides a strong foundation for extending our study to a full, two-player stochastic game, developing a framework for exploring strategically richer pursuit-evasion environments.

9. References

Ankan, A., & Panda, A. (2018). *Hands-on Markov models with Python: Implement probabilistic models for learning complex data sequences using the Python ecosystem*. Packt Publishing.

Chen, Y., Gao, Y., Jiang, C., & Liu, K. R. (2014). Game theoretic Markov decision processes for optimal decision making in social systems. In *2014 IEEE Global Conference on Signal and Information Processing (GlobalSIP)* (pp. 268–272). IEEE. https://doi.org/10.1109/GlobalSIP.2014.7032171

Ding, Z., Huang, Y., Yuan, H., & Dong, H. (2020). Introduction to reinforcement learning. In H. Dong, Z. Ding, & S. Zhang (Eds.), *Deep reinforcement learning* (pp. 1–35). Springer. https://doi.org/10.1007/978-981-15-4095-0_2

Fang, F., Liu, S., Basak, A., Zhu, Q., Kiekintveld, C. D., & Kamhoua, C. A. (2021). Introduction to game theory. In C. A. Kamhoua, C. D. Kiekintveld, F. Fang, & Q. Zhu (Eds.), *Game theory and machine learning for cyber security* (pp. 17–42). Wiley. https://doi.org/10.1002/9781119723950.ch2

Harten, L., Miodownik, S., Stern, S., & Mandelik, Y. (2019). *Predator–prey foraging cycles are driven by resource renewal and depletion. Journal of the Royal Society Interface*, 16(155), 20190087. https://doi.org/10.1098/rsif.2019.0087

Kallenberg, L. (2011). *Markov decision processes* (Lecture notes). Leiden University.

Littman, M. L. (1993). Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning* (pp. 157–163). Morgan Kaufmann.

10. Self-Evaluation Score

Sukhman Parhar

5/5

I helped contribute to the initial creation of the topic as well as the overall description and creation of the game. I was in charge of describing the game as well as helping create the vectors that ended up being added to the code. I did not write the code for the game, but I was the one who created the idea for a predator-prey game and outlined it.