Shiv Nadar University

CSD311: Artificial Intelligence

Dr. Snehasis Mukherjee

Monsoon 2025 Semester

Final Project Report

Snake Game

Group 17

Anisha Pradhan – 2210110851

Chockalingam R – 2210110244

Mananya Bastia – 2210110384

Neradhi Prakash – 2210110734

Subhasri S – 2210110599

Vandana Goyal – 2210110688

ABSTRACT

This report examines automated control methods for the traditional Snake game utilizing search-based planning and deep reinforcement learning. The game is designed as a grid-like setting where an agent must constantly travel from its present head position to food while evading self-collisions and walls in a geometry that changes dynamically. We begin by applying three traditional path-finding algorithms - Breadth-First Search (BFS), Depth-First Search (DFS), and A search - and assess them regarding optimality, time and space complexity, and practical game performance.*
Next, we define Snake as a Markov Decision Process (MDP) and train a Deep Q-Network (DQN) agent that acquires a control policy directly from concise state features.
Experimental findings indicate that A reliably identifies the shortest paths and secures superior scores and survival durations compared to BFS and DFS, whereas the trained DQN agent reaches a consistent average score of about 25 and a maximum score of 72 after multiple hundred games, showcasing strong long-term effectiveness. The research emphasizes the difference between manually designed search strategies and learned policies, while also suggesting multiple paths to expand the work to more intricate environments and more sophisticated reinforcement learning frameworks.*

I. INTRODUCTION

The Snake game is a neat but tough challenge for figuring out how algorithms make decisions. Imagine a snake crawling around on a grid, getting longer every time it eats. The game's over if the snake's head bumps into a wall or its own tail. At first, it might seem like just finding the quickest way to the food. But it's trickier than that. The snake's own body keeps getting in the way, and what you decide to do can unexpectedly block off parts of the grid.
Because of these moving obstacles, the fact that your choices have long-term impacts, and the infrequent rewards, Snake is a great game to test both older search techniques and newer learning methods. If you think about it from a planning angle, you can figure out each move towards the food by searching the grid for a safe path, considering all the current obstacles. If you look at it from a learning angle, the game can be treated like a problem where the agent sees what's happening nearby and has to pick moves that will lead to the best results over time.
We're trying to do three main things here. First, we want to build and check out BFS, DFS, and A* as ways to control the snake, seeing how they perform in theory and practice. Second, we're creating an agent that uses a deep Q-network (DQN) to learn how to play the game directly by interacting with it. Third, we'll compare these methods based on how well they play, how much computing power they need, how they learn, and how good their gameplay is. We'll also talk about what we learn from this that might apply to other games played on grids.

II. LITERATURE SURVEY

A. *Classical Search in Grid Worlds*

When it comes to finding paths, classic methods like BFS, DFS, and A* are essential. BFS explores a graph level by level using a queue, making sure it finds the shortest route in graphs without weights, but it can use up a lot of memory. DFS, on the other hand, uses a stack to go deep down one path before turning back. It uses less memory but doesn't guarantee the best path and can get stuck exploring long, less efficient routes.
A* improves on this by adding a heuristic function, h(n), which guesses the cost from a node to the goal. This is combined with the actual path cost, g(n), to create an evaluation function, f(n)=g(n)+h(n). In grid-like areas, the Manhattan distance is a popular and reliable heuristic. This helps A* find the best paths while typically checking out way fewer spots than BFS. These techniques are commonly used in fields like robotics, game artificial intelligence, and navigation, especially when the surroundings don't change much, making it practical to plan paths over and over.

B. *Reinforcement Learning and Deep Q-Networks*

When it comes to making decisions over time, reinforcement learning is a helpful approach. An agent figures things out by trying things and learning from mistakes to get the most reward possible. Q-learning is one way to do this. It figures out how good it is to take a certain action in a particular situation, assuming you'll make the best choices from then on. If you use deep neural networks to help with this estimation, Q-learning can handle really complex situations.
Deep Q-Networks, or DQNs, which gained popularity thanks to Mnih and his colleagues for their work on Atari games, bring together a neural network with a way to store past experiences and a strategy for exploring different options. The network figures out the expected value of each action for a given situation. Then, the network's settings are adjusted based on the Bellman equation. For the game of Snake, researchers have found that even fairly simple networks, when combined with smart ways of describing the game's state and how rewards are given, can create effective strategies that work well even after multiple attempts.
Generally, when people talk about Deep Q-Networks in research, they mean the specific method Mnih's team developed for Atari. This method uses both storing past experiences and a separate network that's updated less frequently. In this project, though, we're using a more straightforward version of deep Q-learning. We're using just one neural network, storing past experiences, and using a strategy that randomly picks actions some of the time. We also calculate the target values directly from the network we're currently using. So, to be precise, our approach is more like "deep Q-learning with function approximation" rather than the full Atari DQN method.

III. Methodology

A. Game Environment

The environment is an implementation of the classic Snake game on a rectangular grid. The game's status at any moment includes where the snake's body parts are, which way it's moving, and where the single piece of food is, which shows up randomly in an empty spot. When you play, you can either go straight, turn right, or turn left, and that moves the game forward one step. If the snake hits the edge of the grid or itself, the game ends.
You get points for every piece of food you eat. We also keep track of how many steps you survive to see how long a game plan can last without ending, even if it's not getting a lot of food. Both the methods that search for controllers and the DQN agent use these same measurements.

B. Breadth First Search

The Breadth-First Search (BFS) algorithm in the Snake game explores each state in a level-by-level manner. It uses two core data structures:

1) **Queue (FIFO):** A queue storing tuples `(position, path)`. This implements First-In-First-Out ordering.
2) **Visited Set:** A set of visited positions, initialized as `visited = {initial_position}`. This prevents revisiting previously explored cells.

Pseudocode:

Algorithm 1 Breadth-First Search for Snake Pathfinding

```
INITIALIZE queue ← empty_deque
INITIALIZE visited ← empty_set
ENQUEUE(queue, (head, []))
ADD head TO visited
while queue ≠ ∅ do
    (current_pos, path) ← DEQUEUE(queue)
    if current_pos = target then
        if path ≠ [] then
            return path[0]
        end if
    end if
    for each direction ∈ {UP, DOWN, LEFT, RIGHT} do
        next_pos ← current_pos + direction
        if is_valid(next_pos) and next_pos ∉ visited and
        next_pos ∉ snake_body then
            ADD next_pos TO visited
            new_path ← path + [direction]
            ENQUEUE(queue, (next_pos, new_path))
        end if
    end for
end while
return safe_move()
```

The algorithm begins by setting up two essential data structures. The queue serves as a FIFO (First-In-First-Out) container that maintains the exploration frontier—positions waiting to be processed. Initially, it contains only the snake's head position paired with an empty path list. The visited set tracks all positions already explored, preventing cycles and redundant work. The head position is immediately marked as visited since it's our starting point.

The main loop continues as long as the queue contains unexplored positions. In each iteration, the algorithm dequeues the oldest position from the front of the queue using the DEQUEUE operation, which in practical implementations corresponds to the `popleft()` method. This operation is fundamental to BFS's correctness and efficiency for several reasons.

The DEQUEUE/`popleft()` Operation

The `DEQUEUE` operation removes and returns the element at the front (left end) of the queue, implementing First-In-First-Out (FIFO) semantics. This is in direct contrast to stack-based algorithms like DFS, which use `pop()` to remove from the back (Last-In-First-Out). The distinction is not merely implementational; it fundamentally determines the exploration order and thus the algorithm's behavior.

When positions are added to the queue, they are enqueued at the back. The FIFO property ensures that positions added earlier (which are closer to the start) are processed before positions added later (which are farther from the start). It ensures that the algorithm explores level by level and all nodes are explored in finding the optimal path. By dequeuing from the front, we guarantee that all distance-1 positions are processed before any distance-2 position enters the processing stage.

Key Properties:

1) **FIFO ordering:** The queue's FIFO property ensures breadth-first exploration, processing all positions at distance d before any at distance $d + 1$.
2) **Early termination:** The goal check (in the pseudocode) occurs immediately upon dequeuing, enabling early exit once the shortest path is found.
3) **Path construction:** Each queue entry maintains its complete path history, allowing immediate path reconstruction upon goal discovery without backtracking.
4) **Visited marking:** Positions are marked visited when added to the queue, not when dequeued, preventing duplicate queue entries and ensuring linear-time complexity.
5) **Completeness:** If a path exists, the exhaustive level-by-level exploration guarantees its discovery.

C. Depth First Search

The Depth-First Search (DFS) algorithm forms an alternative decision-making mechanism to guide the snake toward the food. Like A*, it operates on the same grid-based structure where the snake moves in four discrete directions: up, down, left, and right. However, unlike A*, which balances optimality with efficiency through heuristic evaluation, DFS follows a much simpler paradigm: it explores one entire path as deeply as possible before considering alternatives. This makes DFS

computationally lightweight and easy to implement, though it does not guarantee the shortest route to the food.

The algorithm begins by pushing the snake's head into a *stack*, which governs DFS's characteristic **Last-In-First-Out (LIFO)** behavior. Each entry in this stack stores two things:

1) the current cell's coordinates, and
2) the full sequence of moves (path) taken to reach that cell.

At the start of each iteration, DFS pops the top-most element from the stack. This popped element acts as the current expansion node in the implicit search tree. DFS then checks whether this cell corresponds to the food. If it does, the algorithm immediately terminates and returns the first direction in the recorded path—this becomes the snake's next movement. If not, DFS examines all four neighboring cells in a predetermined order.

A neighboring cell is considered valid only if:

- it lies within the grid boundaries,
- it does not overlap with any segment of the snake's body, and
- it has not already been visited during the current search.

Valid neighbors are pushed onto the stack along with their updated path histories. Since the stack processes the most recently added nodes first, DFS naturally dives deeper along one direction, exploring full branches before backtracking to earlier decision points. As a result, the search tree formed by DFS is typically **long and narrow**, following a depth-oriented exploration pattern rather than a broad, level-by-level expansion like BFS or A*.

Unlike algorithms that compute path cost or use heuristics, DFS does not evaluate which move is "better." It simply explores legal moves in the order presented. This means DFS will often find a valid but not necessarily shortest path to the food. In practice, this behavior yields unpredictable path shapes, making DFS-driven gameplay appear more exploratory and less optimal than A*. However, the algorithm's simplicity provides a distinct computational advantage: DFS has low overhead, requires minimal memory, and can traverse large portions of the grid extremely quickly.

If DFS exhausts all reachable states without finding a route—usually occurring when the snake forms a near-enclosed loop—the algorithm invokes a fallback safety mechanism. The *safe-move function* checks all four directions and chooses any movement that avoids immediate collision risks. This prevents abrupt termination and gives the snake a last opportunity to survive.

Overall, DFS provides a straightforward, deterministic method for pathfinding in the Snake environment. While it lacks the optimality and precision of A*, it compensates with simplicity, speed, and ease of implementation. Its depth-oriented search style also makes it valuable for analyzing how different exploration strategies influence gameplay, where exploration and survival can outweigh path optimality.

D. A Alogorithm*

The A* algorithm is the core decision-making system that drives the snake's movement toward the food. Since the entire game runs on a grid and the snake is restricted to four movements (up, down, left, right), A* becomes a natural choice because it balances accuracy with computational efficiency. The algorithm begins by inserting the snake's head into a priority queue (implemented with Python's heapq).

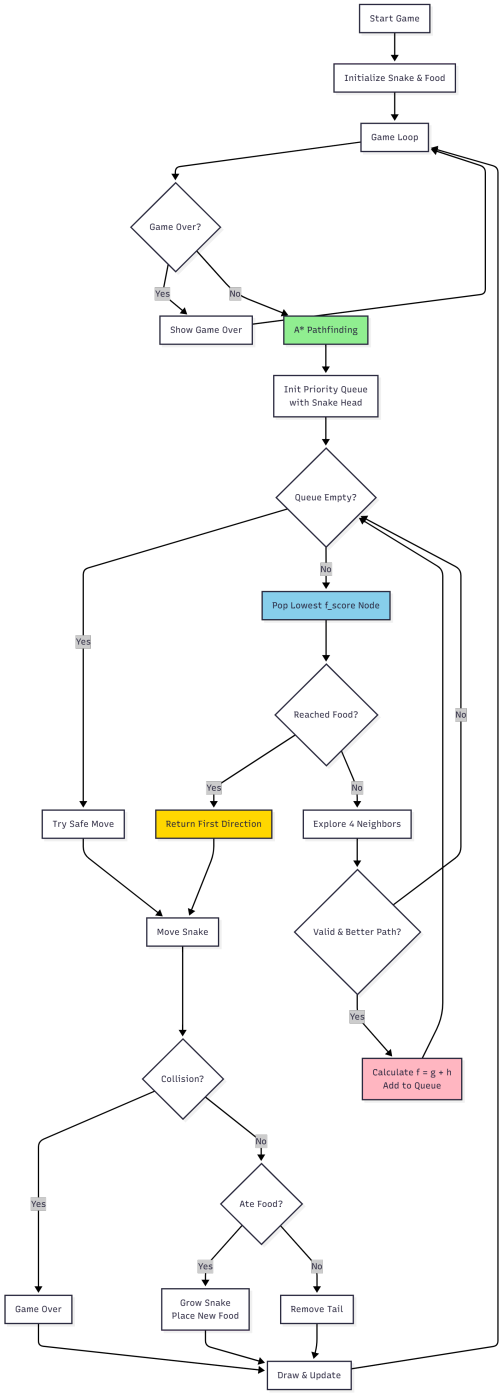

Fig. 1: Working of A* Algorithm

Every entry in this queue stores:

- the f-score (total estimated cost),
- a tie-breaker counter,
- the current cell position, and
- the full path taken to reach that cell.

A* evaluates each move using the formula f(n) = g(n) + h(n).

- g(n) represents the number of steps taken from the snake's head to the current position.
- h(n) is the estimated distance from the current position to the food.

For h(n), the code uses Manhattan distance. This is ideal because it measures distance strictly through horizontal and vertical movement - exactly how the snake moves. It never overestimates the actual path length, which ensures A* always produces the shortest possible path if a solution exists. This keeps the AI both optimal and predictable.

During the search, A* repeatedly takes the position with the lowest f-score from the queue and checks whether it has reached the food. If not, all four neighboring cells are examined. A neighbor is only considered valid if it lies within the grid and does not collide with the snake's body. For each valid neighbor, the algorithm calculates new g, h, and f values and pushes it back into the priority queue if this new route is better than previously recorded ones. If A* cannot find a path - usually when the snake is surrounded - the game gracefully falls back to a safe-move function, which chooses any non-fatal direction. This ensures the game continues smoothly even in tight situations.

E. Deep Q-Networks

Markov Decision Process & State Representation

We cast the Snake environment as an MDP defined by the tuple (S,A,P,r,). The state space S is encoded into an 11-dimensional binary feature vector:

- 3 features indicating danger straight, danger right, and danger left relative to the current direction of motion (1 if the next step in that direction would cause collision, 0 otherwise).
- 4 one-hot features encoding the current movement direction (left, right, up, down).
- 4 one-hot features encoding the relative position of the food with respect to the snake head (food left, right, up, down).

This representation abstracts away the full grid but retains local safety and directional information necessary for effective control. For example, [0,0,0, 0,1,0,0 0,1,0,0] implies:

- No danger straight, right, or left
- Snake is moving RIGHT
- Food is to the RIGHT of the head

The action space A=0,1,2 contains three relative actions: go straight, turn right, or turn left. The transition function PPP is implicitly defined by the game dynamics: given a state vector and an action, the environment updates the snake's position and returns the next state.

The reward function is shaped to accelerate learning:

- +10 when the snake eats food,
- 10 when the snake dies,
- 0.1 for every time step otherwise. The small per-step penalty discourages aimless wandering and encourages the agent to reach food quickly rather than merely surviving.

Transition Function

In the Markov Decision Process (MDP) formulation, the environment dynamics are described by the transition function:

$$\mathcal{P}(s' \mid s, a)$$

which gives the probability that the next state is s when the agent takes action a in state s.

In our Snake environment the dynamics are deterministic: for each pair(s,a) there is exactly one successor state s. Equivalently, the transition kernel satisfies:

$$P(s' \mid s, a) = \begin{cases} 1, & \text{if } s' = f(s, a), \\ 0, & \text{otherwise.} \end{cases}$$

for some deterministic transition function f. Rather than representing P explicitly, we implement this function through the game engine. Conceptually, the environment applies:

$$(s_t, a_t) \mapsto (s_{t+1}, r_{t+1}, \text{done}),$$

That is, the current state vector together with the chosen action fully determines the next state, the immediate reward, and whether the episode terminates.

In practice, we never compute P explicitly; we simply call `play_step`, which implements the deterministic function $f(s, a)$, and `get_state`. The call `game.play_step(action)` applies the transition function to the internal game state given the selected action, and returns the corresponding reward r_{t+1}, terminal flag `done`, and updated score. The call `agent.get_state(game)` encodes the updated game configuration into the next state vector.

Discount Factor

A discount factor γ=0.9 balances immediate and future rewards: the agent is encouraged to seek food but also to avoid states leading to delayed death.The discount factor specifies how much the agent values future rewards relative to immediate rewards. For a trajectory starting at time t, the (discounted) return is defined as:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

When γ=0, the agent is completely myopic and optimises only the immediate reward. As approaches 1, future rewards contribute almost as much as present rewards, and the agent becomes more "patient", willing to sacrifice short-term gain for long-term benefit. Choosing =0.9 therefore encourages the Snake agent to consider the long-term consequences of its moves (e.g. avoiding self-traps) while still slightly preferring rewards that can be obtained sooner.

Network Architecture

The Q-network is a fully connected feed-forward network with:

- Input layer of size 11 (one neuron per state feature)
- Single hidden layer with 256 neurons and a non-linear activation
- Output layer with 3 neurons corresponding to the Q-values for the three actions

$$f_\theta : \mathbb{R}^{11} \to \mathbb{R}^3$$

where the 11-dimensional input is the state vector and the three outputs correspond to the three possible actions (go straight, turn right, turn left). Each output f is the estimated Q-value Q(s,ai), i.e. the predicted long-term return if action is taken in state s and the current policy is followed thereafter.
The hidden layer with 256 units learns an internal feature representation of the state. These features are not interpreted individually, rather are latent variables that allow the network to approximate the nonlinear mapping from state to action values. During decision making, the agent feeds the current state into this network and selects the action with the highest Q-value, unless exploration is triggered by the ϵ-greedy policy.

```
class Linear_QNet(nn.Module):
    def __init__(self, input_size, hidden_size
        , output_size):
        super().__init__()
        self.linear1 = nn.Linear(input_size,
            hidden_size)
        self.linear2 = nn.Linear(hidden_size,
            output_size)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = self.linear2(x)
        return x

self.model = Linear_QNet(11, 256, 3)    # 11
    inputs -> 3 actions
```

Exploration Strategy

The agent uses an ϵ-greedy policy. At game n, the probability of a random move is :

$$P(\text{random action}) = \frac{\epsilon}{200}$$

The exploration was implemented through:

```
if random.randint(0, 200) < self.epsilon:
```

At game n, we set:

$$\epsilon = \max(0,\ 80 - n)$$

so that epsilon decays linearly from 80 to 0 over the first 80 games. Because the condition is "randint(0,200) < epsilon", the probability of choosing a random action is epsilon/200. Early in training (large epsilon) the agent mostly explores, while after game 80 we have epsilon = 0 and the policy becomes fully greedy with respect to the learned Q-values.. For each decision step:

- With probability proportional to e, a random action is sampled to encourage exploration.
- Otherwise, the action with the highest Q-value is selected.

This schedule forces strong exploration in an initial "childhood" phase, followed by gradual transition to exploitation as the policy improves. The behaviour is illustrated in the epsilon-decay and exploration-versus-exploitation plots across training games.

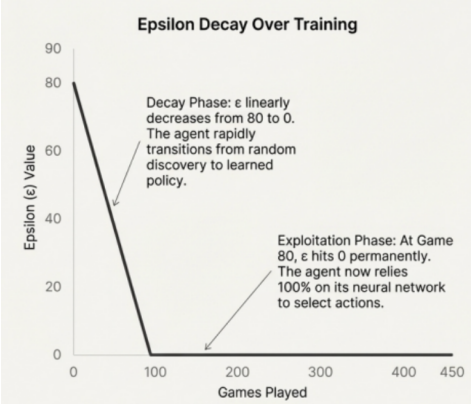

Fig. 2: Epsilon Decay Over Training

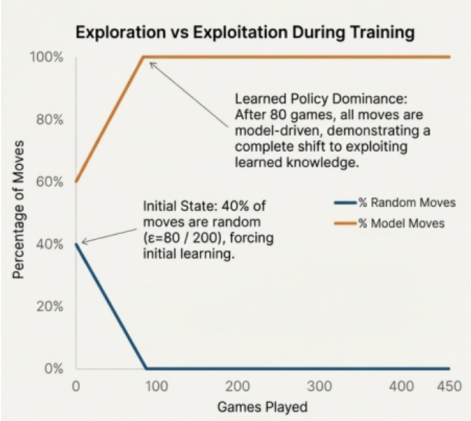

Fig. 3: Exploration vs Exploitation During Training

Experience Replay & Training

During interaction, the agent collects experience tuples $(s, a, r, s', \text{done})$ after each step. These are stored in a replay buffer implemented as a bounded deque. The training process uses two complementary update mechanisms:

1) **Short-memory training:** immediately after each step, the agent performs an update using the most recent tuple.
2) **Long-memory training:** when a game ends, a batch of past experiences is sampled from the replay buffer to perform additional updates, decorrelating consecutive experiences and improving data efficiency.

For each sampled transition, the target Q-value is computed using the Bellman update:

$$Q_{\text{target}} = r + \gamma \max_{a'} Q(s', a')$$

where

$$Q_{\text{target}} = \begin{cases} r, & \text{if done,} \\ r + \gamma \max_{a'} Q(s', a'), & \text{otherwise.} \end{cases}$$

The network parameters are then adjusted via gradient descent to minimise the difference between current predictions and these targets. This drives the network toward a fixed point approximating the optimal action-value function over games.

We also log the score of each game to a CSV file and keep track of the highest score obtained so far. Whenever a new record is achieved, the corresponding network parameters are saved to disk as `best_model.pth`. For evaluation runs, we reload this checkpoint and run the agent without exploration, so that it plays purely according to the learned Q-values.

IV. Results

Search-Based Algorithms

The evaluation known as the "Algorithm Arena" assesses BFS, DFS, and A* based on their theoretical characteristics and empirical gameplay metrics.

Both BFS and A* are optimal regarding path length on unweighted grids, in contrast to DFS, which does not share this property. While BFS and DFS exhibit an O(V+E) time complexity, they differ in terms of space usage: BFS necessitates the storage of entire wavefronts of nodes, leading to significant memory consumption; on the other hand, DFS utilizes memory in proportion to the maximum depth, rendering it more space-efficient. A* incurs additional costs associated with maintaining a priority queue but typically explores fewer nodes due to its heuristic guidance.

The scoreboard provides a summary of the average score and survival time across various runs. A* secures the highest score and the longest survival time, indicative of its capability to identify near-optimal routes that circumvent unnecessary detours and minimize the risk of the snake becoming trapped. BFS achieves moderate scores and survival times: it consistently identifies the shortest path, yet its lack of foresight regarding potential future self-collisions can occasionally result in traps. Conversely, DFS ranks lowest on both metrics; its indiscriminate deep exploration frequently produces long, convoluted paths that intersect with the snake's own body or lead it into corners.

A qualitative analysis of movement further underscores these distinctions: BFS is inclined to generate rigid and somewhat mechanical movements, DFS results in erratic trajectories susceptible to collisions, while A* demonstrates fluid and strategically directed movement towards goals. In summary, among the deterministic search controllers, A* is distinctly the superior choice for Snake, both in theoretical and empirical performance.

Deep Q-Network Performance

Training the DQN agent over several hundred games results in a distinct learning curve. In the initial training phase (approximately games 0 - 80), significant exploration leads to considerable volatility: numerous low-scoring games interspersed with occasional successes as the agent identifies rewarding behaviors.

After the exploration parameter ϵ diminishes to 0 around game 80, the agent transitions into a phase of pure exploitation. During this time, both the highest score achieved and the running mean score show a consistent increase. The record score reaches 72 by approximately game 145, while the mean score across games stabilizes around 25 after about 300 games. The convergence plateau signifies that the agent has acquired a policy that is both stable and effective given the architecture and state representation. Subsequent games display ongoing variability in raw scores - partly due to the random placement of food - yet there is no systematic drift in the average. The learning curve plots illustrate this behavior clearly: the record curve levels off, the mean curve stabilizes, and the per-game score remains erratic but within bounds. Qualitatively, the trained agent exhibits behaviors that are challenging to encode through static search methods alone. It tends to evade immediate self-traps, maneuvers around its body to access food from safer angles, and maintains a balance between greedily pursuing nearby food and preserving maneuverability. These behaviors emerge solely from the reward signal and the dynamics of the environment, without the need for explicit path-planning code.

V. Conclusion

This report provides a comparative analysis of search-based and learning-based control strategies for the Snake game. Classical planners such as BFS, DFS, and A* were employed to compute a path to food at each decision-making step, while a Deep Q-Network agent was trained to derive a control policy directly from compact state features.

The evaluation revealed that A* stands out as the most effective among the classical algorithms, merging optimality with efficient heuristic guidance. It achieves superior scores and longer survival rates compared to BFS and DFS, producing trajectories that are both purposeful and smooth. In contrast, the DQN agent learns to attain an average score of approximately 25, with a peak score of 72, illustrating that a relatively small neural network utilizing a straightforward 11-dimensional state representation can significantly master the game.

This comparison underscores a crucial trade-off. Search-based methods offer theoretical guarantees and transparent behavior but must resolve a planning problem at each time step, often struggling with long-term consequences that extend beyond the immediate snapshot. Conversely, the trained DQN policy executes swiftly and implicitly considers long-term rewards, yet it necessitates extensive data and meticulous tuning of state features, rewards, and exploration schedules.

VI. Future Extensions

Improved State Representations

Rather than utilizing manually created 11-dimensional vectors, one might input a raw grid representation into a convolutional neural network (CNN). This approach allows the agent to uncover more complex spatial features, such as patterns in the entire layout of the snake's body and its closeness to corners.

Advanced RL Algorithms

The original Atari DQN algorithm utilizes a distinct target network, with its parameters being updated solely at predetermined intervals. In contrast, our current agent does not incorporate a target network; the addition of such a network would align the method more closely with conventional DQN and generally enhance stability. Furthermore, strategies like Double DQN, Dueling Networks, and Prioritized Experience Replay could further mitigate over-estimation bias and speed up convergence. Additionally, actor-critic methods (such as A2C or PPO) can produce more refined policies with improved credit assignment during extended episodes.

Hybrid Planning - Learning Agents

A potential avenue for advancement is the integration of local A* planning with a high-level learned policy. For instance, the policy might determine the appropriate moments to engage A* in pursuit of various intermediate objectives, or it could employ search techniques to enhance decision-making in high-risk scenarios. This approach would combine the optimality assurances provided by search with the flexibility inherent in reinforcement learning.

Richer Environments and Objectives

Extensions of Snake that incorporate multiple food items, moving obstacles, or partial observability would evaluate the general applicability of the acquired strategies. Multi-objective reward functions could provide a balance between score, survival duration, and path smoothness. Curriculum learning, which begins with smaller grids and progressively increases in difficulty, may enhance the sample efficiency of the training process.

Safety and Robustness Analysis

A thorough assessment involving adversarial initializations, randomized food distributions, or limitations on computation time would yield a more profound insight into the robustness of each controller in real-time scenarios.

In summary, the Snake game remains a concise yet profound area for investigating the relationship between traditional algorithms and contemporary reinforcement learning. The findings presented here indicate that each paradigm possesses unique advantages, and it is probable that future systems will gain from integrating search and learning in mutually beneficial manners.

Acknowledgments

We sincerely thank Dr. Snehasis Mukherjee for his guidance and support throughout this project.

References

[1] GeeksforGeeks, "Markov Decision Process (MDP) in Machine Learning," *GeeksforGeeks*, Available: https://www.geeksforgeeks.org/machine-learning/markov-decision-process/.

[2] GeeksforGeeks, "Bellman Equation in Machine Learning," *GeeksforGeeks*, Available: https://www.geeksforgeeks.org/machine-learning/bellman-equation/.

[3] GeeksforGeeks, "Breadth-First Search (BFS) for a Graph," *GeeksforGeeks*, Available: https://www.geeksforgeeks.org/dsa/breadth-first-search-or-bfs-for-a-graph/.

[4] N. S. D. Appaji, "Comparison of Searching Algorithms in AI Against Human Agent in Snake Game," 2025.

[5] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.