Contents

Ant Trail Problem

Introduction

The Ant Trail Problem is
a test configuration aimed at experimenting and developing artificial intelligence agents, specifically reinforcement learning and decision-making under uncertainty in a dynamic, partially observable environment. The core problem is to get an 'ant' traverse a grid, choose to maximize a score by its interaction with the world,
and acquire best strategies from incomplete local knowledge. This problem is an ideal case study to implement various machine learning techniques, including decision trees, multilayer perceptrons, and genetic programming, to control the action of an agent.

Problem Description and Game Rules

The Ant Trail Problem is set on an N x N grid, where N randomly selected cells are initially populated with 'food,' each having a value of +1. All other cells are empty, with a value of 0. The ant, the primary agent in this simulation, begins its journey on any empty cell, with its initial position's value set to -1. The ant's movement is restricted to four cardinal directions: up, down, left, and right. The game's objective is to maximize the ant's accumulated score within a fixed limit of 2N moves.

As the ant traverses the grid, its score is updated based on the value of the cell it enters. Upon entering a cell, the cell's value is decremented by 1, simulating the consumption of resources or a change in the environment due to the ant's presence. A critical rule is the penalty for exiting the grid: if the ant moves outside the boundaries, the game immediately ends, and a substantial penalty of N+2 points is subtracted from its score. This rule encourages the ant to remain within the grid and strategically navigate its environment.

Ant's Field of View and State Representation

A crucial aspect of the Ant Trail Problem is the ant's limited perception of its environment. The ant possesses a 'field of view' (FOV) that corresponds to a square neighborhood of size (2m+1) x (2m+1) centered on its current position, where 'm' represents the 'radius' of its vision. This localized view means the ant's decision-making process must rely solely on the information available within this immediate neighborhood. Consequently, the ant's state at any given moment is entirely described by the values of the cells within its FOV.

This limited perception introduces a significant challenge: the ant must learn to make optimal decisions without complete knowledge of the entire grid. This partial observability is a common characteristic of real-world reinforcement learning problems, making the Ant

Trail Problem a valuable testbed for algorithms designed to operate under such constraints.

Visual Representation

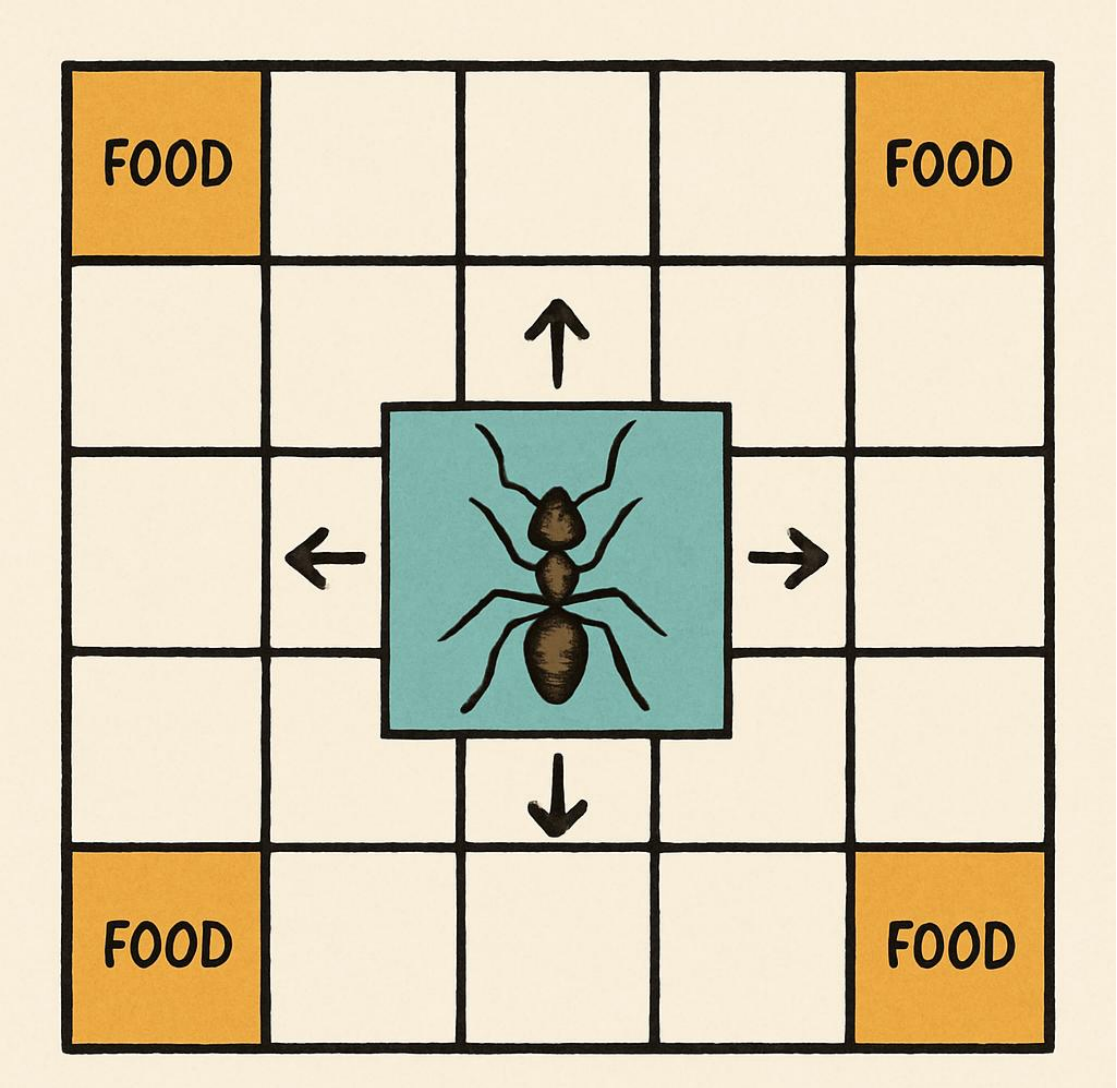


Figure 1This diagram visually depicts the ant on a grid, highlighting its field of view and possible movement directions. The cells marked as "FOOD" represent positive value cells, while the ant's current position and movement affect cell values and the overall

Machine Learning Approaches

The Ant Trail Problem is designed to evaluate the effectiveness of various machine learning algorithms in developing an intelligent agent capable of navigating and optimizing its score within the defined environment. The problem specifically outlines the use of three distinct approaches:

1. Decision Trees

Decision trees are a non-parametric supervised learning method used for classification and regression. In the context of the Ant Trail Problem, a decision tree would be trained to map the ant's field of view (the input attributes) to a specific movement decision (up, down, left, or right). The training data for this would be collected from interactive games, where human or pre-programmed optimal moves are recorded alongside the ant's local state. The decision tree would then learn a set of rules to predict the best move given a particular configuration of cells in the ant's neighborhood.

2. Multilayer Perceptrons (Neural Networks)

Multilayer Perceptrons (MLPs) are a class of feedforward artificial neural networks. For the Ant Trail Problem, an MLP would be configured with an input layer corresponding to the size of the ant's field of view (e.g., 9 inputs for a 3x3 neighborhood or 25 for a 5x5 neighborhood). The hidden layers would process this information, and the output layer would consist of four nodes, each representing a possible movement direction. The MLP would be trained using backpropagation to learn the complex, non-linear relationships between the ant's sensory input and the optimal action, effectively classifying the current state into one of the four movement categories.

3. Genetic Programming (GP)

Genetic Programming is an evolutionary computation technique that automatically generates computer programs to solve problems. In the Ant Trail Problem, GP would evolve programs (represented as parse trees) whose terminals are the cell values within the ant's neighborhood and whose functions include arithmetic operations. The output of these evolved programs would then be interpreted to determine the ant's next move based on predefined thresholds. For example, a program outputting a value within a certain range might correspond to moving 'up,' another range to 'down,' and so on. GP's strength lies in its ability to discover novel and often unintuitive solutions by exploring a vast search space of possible programs.

Visualizing the Ant Trial Game

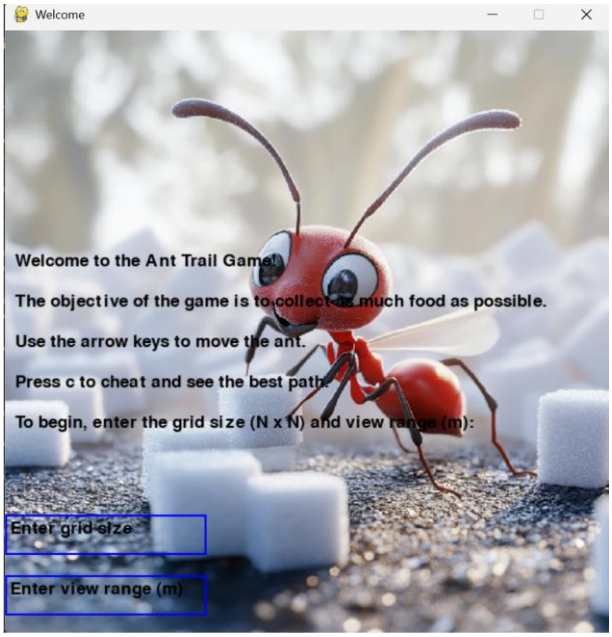


Figure 2The initial game window

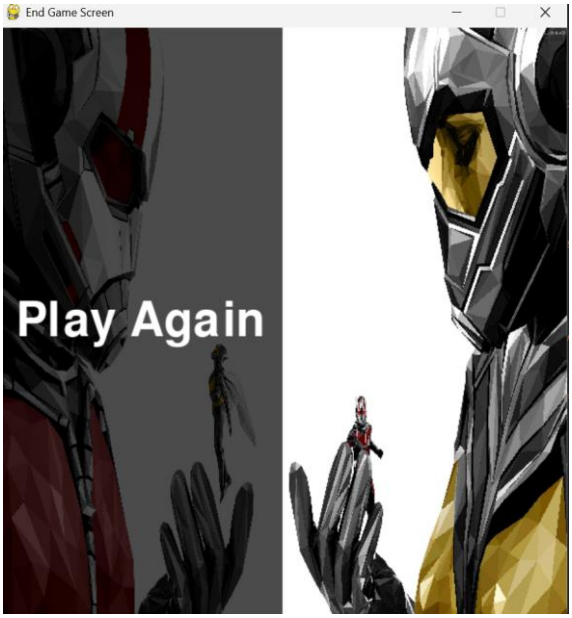


Figure 3The final screen of the game

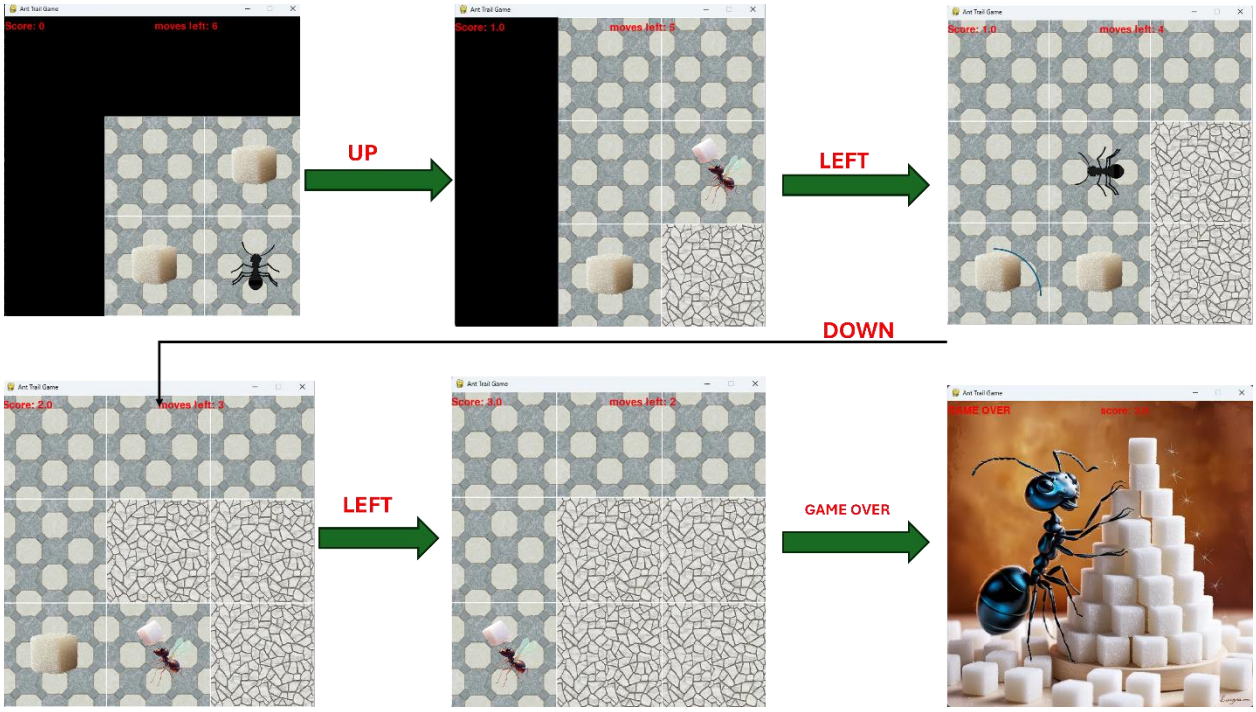


Figure 4The gameplay screen

Getting Training Data: Neighborhood and Direction

Data for Grid >= 6

n the Ant Trail Problem, training a learning agent requires a set of **labeled examples**, where each example consists of:

- The **neighborhood** of the ant's current position (its observable field),
- The **direction** the ant chose to move in that state.

This input-output format supports supervised learning by mapping localized spatial input to optimal movement decisions. However, obtaining high-quality training data is a challenge: random decisions are uninformative, and pure brute-force exploration is computationally infeasible for larger grids.

To address this, I implemented a smart, depth-limited path-finding algorithm that simulates intelligent ant behavior while remaining computationally efficient for grid sizes $N \leq 6$.

Efficient Path Finder: Smart Brute Force (Backtracking)

The algorithm simulates all possible valid paths from the ant's current position but prunes unproductive branches and prioritizes high-reward cells. This is achieved using a recursive backtracking approach with greedy sorting.

For small grids like N=5N = 5N=5 or N=6N = 6N=6, this algorithm explores all high-value paths within a reasonable time (milliseconds to seconds), producing high-quality, near-optimal training data.

Each optimal path can then be replayed to generate training rows like:

- *0,1,0,0,0,1,0,0,-1,right*
- *1,0,0,0,-1,0,0,0,0,down*

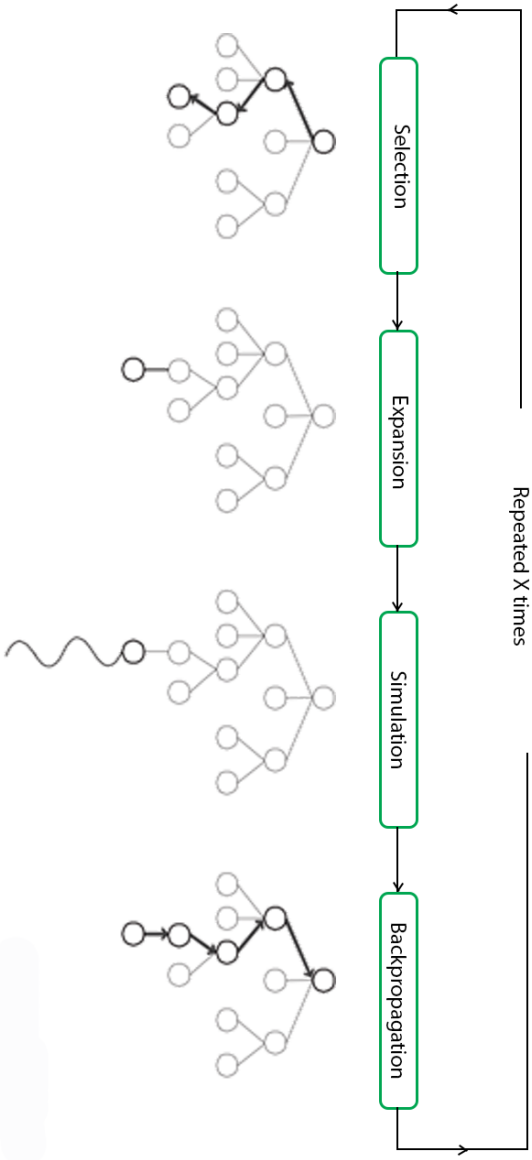
Training Data Collection for Larger Grids: Monte Carlo Tree Search (MCTS)

While small grid sizes (N≤6N \leq 6N≤6) allow the use of recursive backtracking to simulate near-optimal ant behavior within practical time limits, larger grid sizes (N>6N > 6N>6) present a combinatorial explosion of possibilities that make brute-force or depth-limited search computationally infeasible. For example, evaluating all possible sequences of 2N = 20 moves on a 10×10 grid would involve up to 4^20 ≈ 10^12 paths.

To overcome this limitation, we employed Monte Carlo Tree Search (MCTS), a probabilistic decision-making algorithm that balances exploration and exploitation through statistical sampling and simulation. MCTS enables the agent to estimate the best move at each step by running randomized rollouts and backpropagating rewards through a dynamically built search tree.

How it works:

- **Selection**: Traverse the tree from the root, selecting child nodes based on the **Upper Confidence Bound (UCB1)** formula to balance nodes with high average reward and low visit count.
- **Expansion**: Once a leaf node is reached, expand it by generating child nodes for each possible legal move from the current game state.
- **Simulation (Rollout)**: From the expanded node, simulate the rest of the game using an **ε-greedy** strategy (random with probability ε = 0.1, greedy otherwise).
- **Backpropagation**: After the simulation ends, propagate the outcome (final score and remaining moves) up the tree to update visit counts and cumulative rewards.



Important Observation: Multiple Optimal Solutions per Game

It is crucial to note that a single Ant Trail game instance may have multiple valid solutions that achieve the same final score using the same number of moves. This phenomenon arises from the structure of the problem: since the ant's reward depends only on the values of the cells it visits, not the exact path taken, different movement sequences can lead to identical total scores.

For example, the ant might collect the same amount of food by reaching high-value cells through different routes, especially when those cells are symmetrically placed or equidistant. As long as the visited cells and their consumption order produce the same cumulative reward, the final score remains unchanged—even if the ant’s path varies.

This means that:

- There is not a unique optimal solution for most games.
- Different optimal paths can generate different training data, even if they are equally good.
- Learning models may generalize differently depending on which solution paths were used during data collection.

This variability highlights the importance of collecting diverse training samples and possibly augmenting them with rotated or mirrored versions of equivalent states.

Data Collection Script

collectadata.py

To support supervised learning, I developed a standalone script named collectadata.py. This script automates the generation of training data by simulating a series of ant games and recording the ant's neighborhood and movement decisions.

The goal of the script is to:

- Simulate multiple full games using the **GameLogic** class.
- For each move, extract the ant's local neighborhood as features.

- Record the direction chosen from the precomputed optimal path.
- Save the full dataset as a CSV file for training machine learning models (e.g., decision tree, MLP, or GP).

Exporting Training and Testing Data for Weka: wekadata.py

In addition to collecting data for Python-based machine learning workflows, we created a separate utility script, wekadata.py, to export simulation data in ARFF format (Attribute-Relation File Format). This format is used by Weka, a widely used Java-based machine learning toolkit. The script automatically generates both training and testing datasets, formatted and ready for experiments within Weka.

The main goals of wekadata.py are:

- To simulate multiple games using the GameLogic class.
- To extract ant neighborhood features and the direction taken at each step.
- To convert this data into the ARFF format, with proper metadata (@relation, @attribute, @data).
- To split the dataset into training and test sets for fair evaluation.

Machine Learning Algorithms for Ant Decision Making

Decision Trees

To model the decision-making behavior of the ant, I implemented a supervised learning agent using a Decision Tree Classifier. This approach is interpretable, simple to train, and well-suited for problems where the input features (the neighborhood values) directly map to a limited number of class labels (directions).

Evaluation of Decision Tree Agent

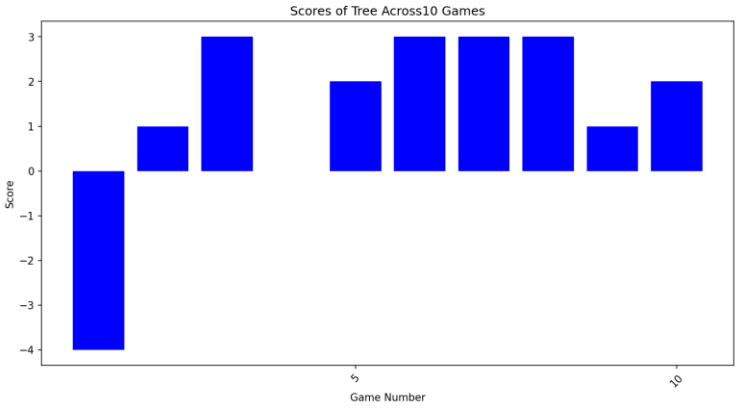
To assess the performance of the Decision Tree model, I evaluated it over 10 test games after training it with datasets generated from different configurations of the game environment and training sessions. Specifically, we varied:

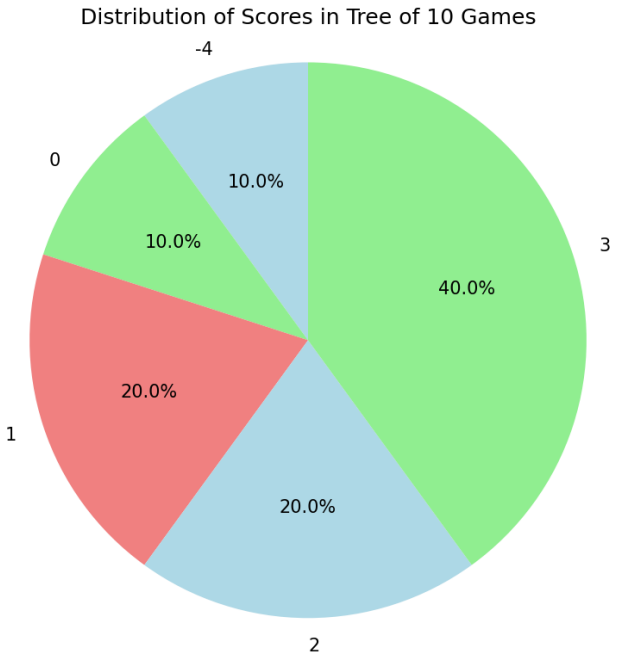
- The number of games used to generate training data (e.g., 1, 5, or 10 games).
- The grid size N (e.g., 6, 8, or 10).

- The ant's neighborhood radius mmm (e.g., 1 or 2).

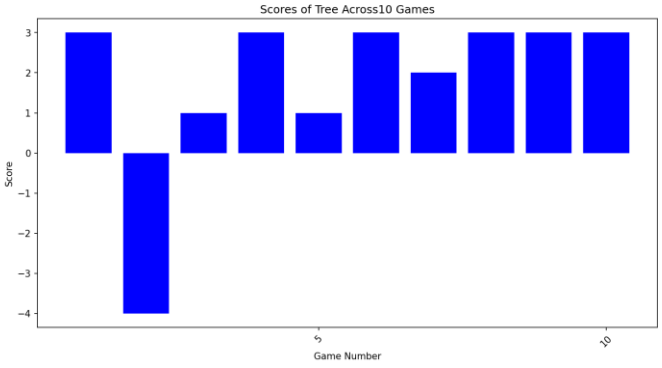
For each configuration, we recorded the score obtained in each test game using the trained model and visualized the results as shown below.

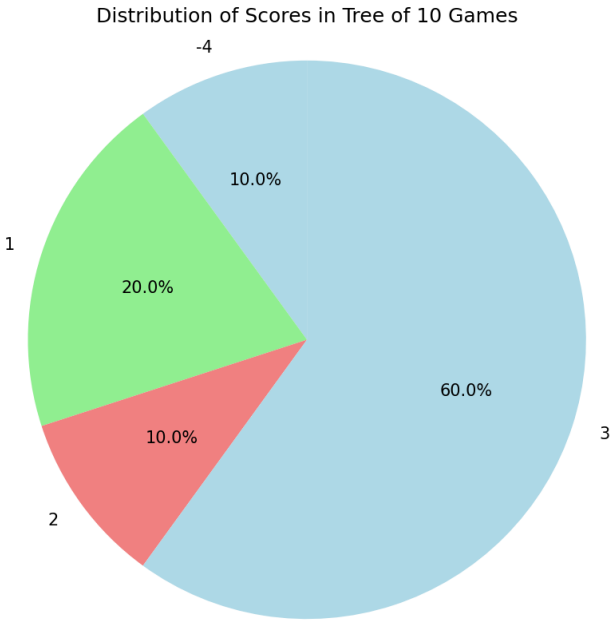
N = 3, m = 1, g(number of played games) = 5.



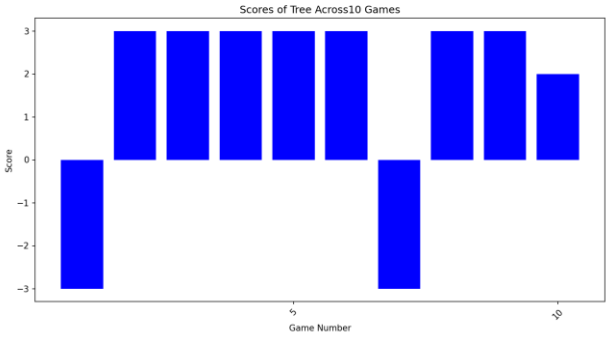


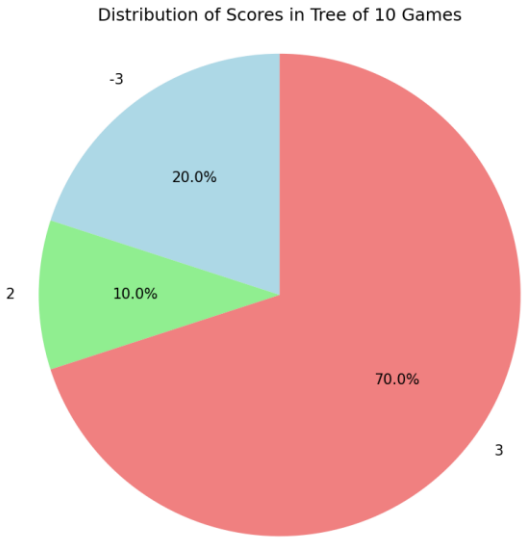
N = 3, m = 2, g = 5.



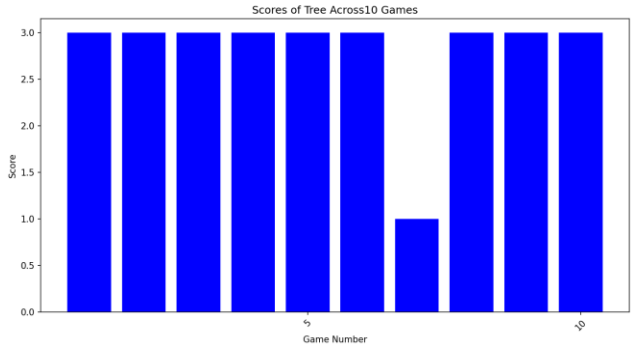


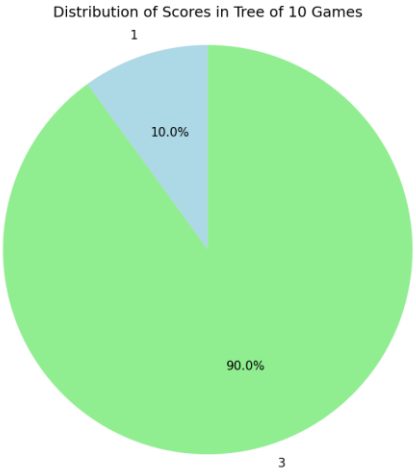
N = 3, m = 1, g = 10.



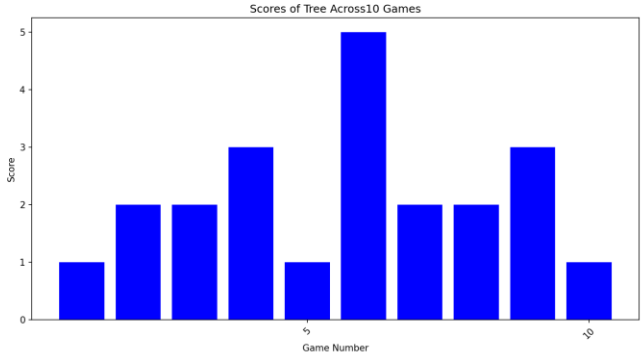


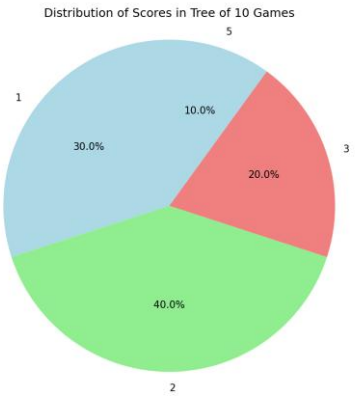
N = 3, m = 2, g = 10.





N = 5, m = 2, g = 10

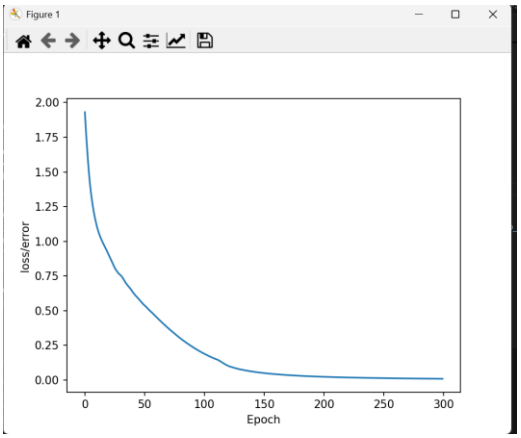


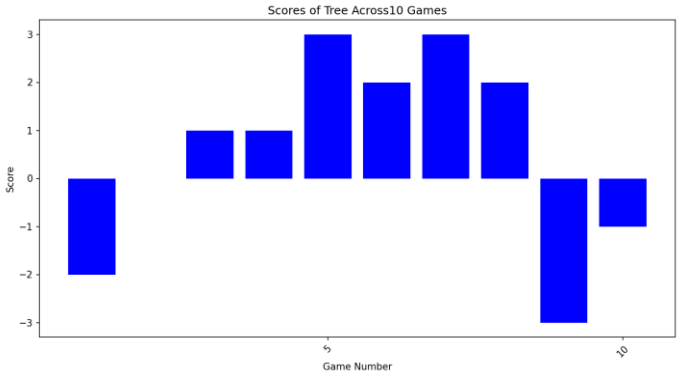


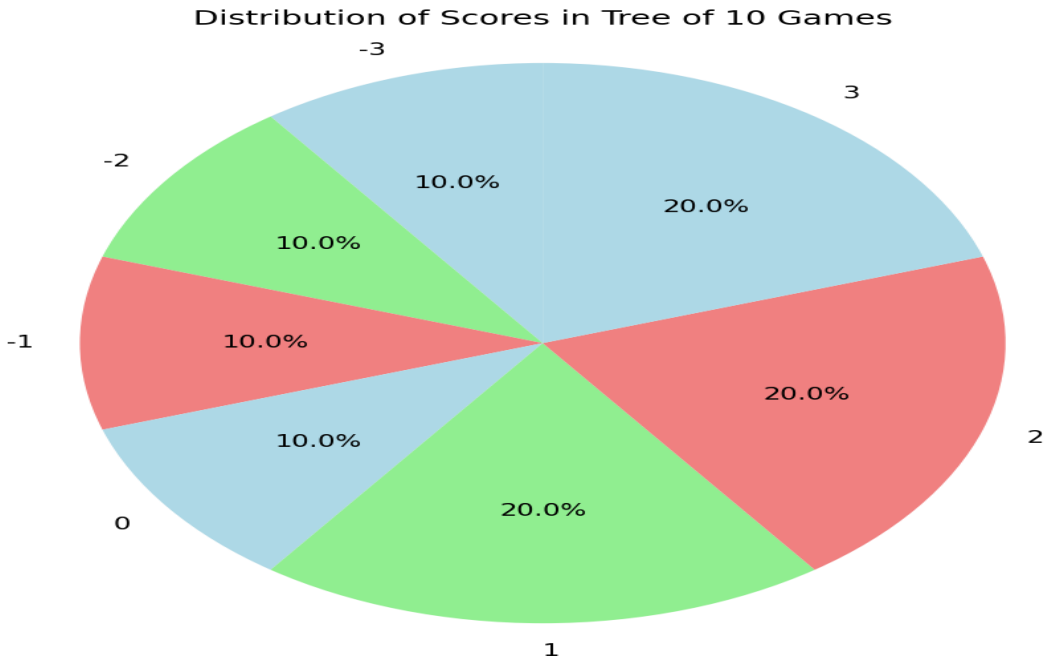
Neural Network Agent

To enable more flexible and expressive learning, I implemented an agent that uses a feedforward neural network to predict movement directions based on the ant's local neighborhood. This approach can capture non-linear patterns and complex feature interactions that may be difficult for rule-based systems or shallow models to learn.

N = 3, m = 1, g(number of played games) = 5.







N = 3, m = 2, g = 5.

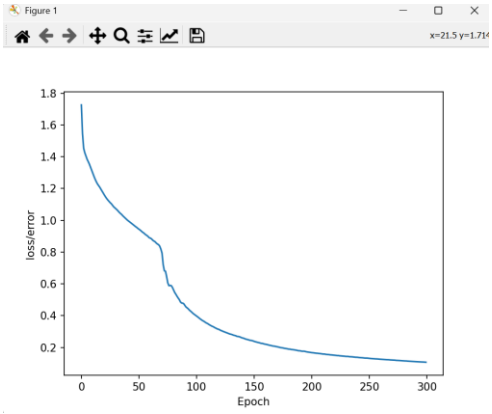
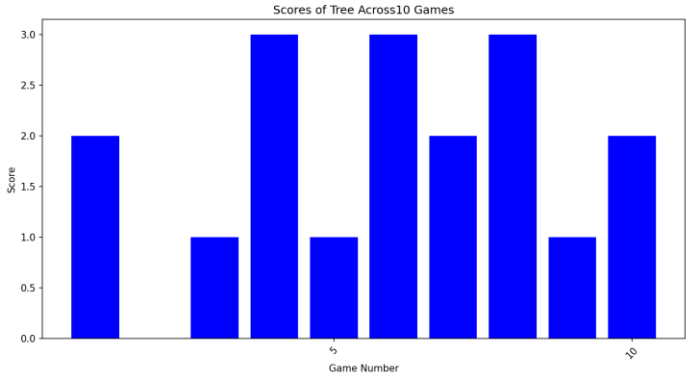
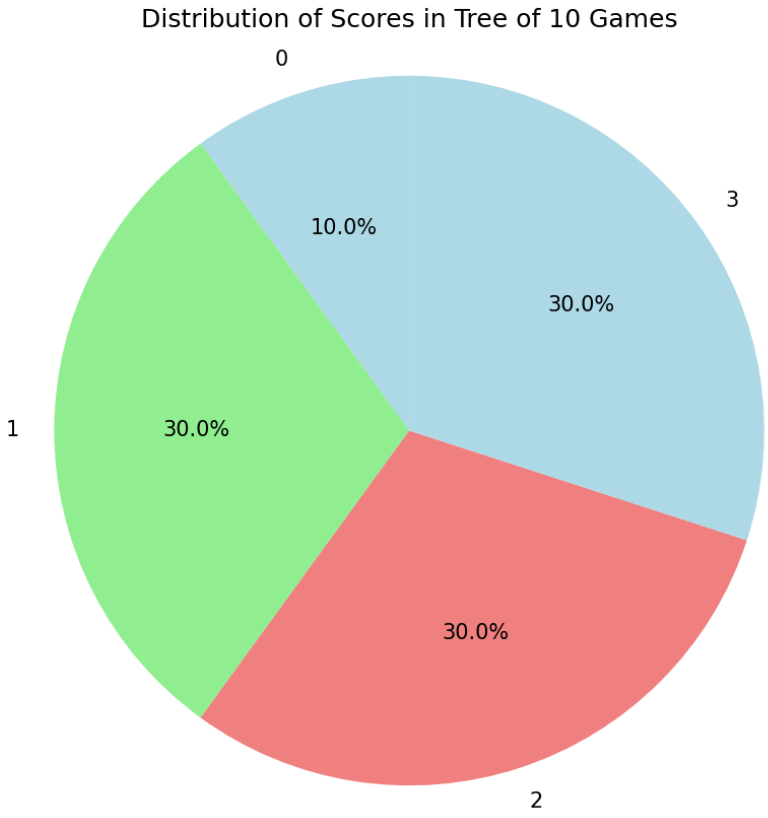


Figure 1
x=21.5 y=1.714
loss/error
Epoch


Scores of Tree Across10 Games
Score
Game Number


Distribution of Scores in Tree of 10 Games
0
3
1
2
10.0%
30.0%
30.0%
30.0%

N = 3, m = 1, g = 10.

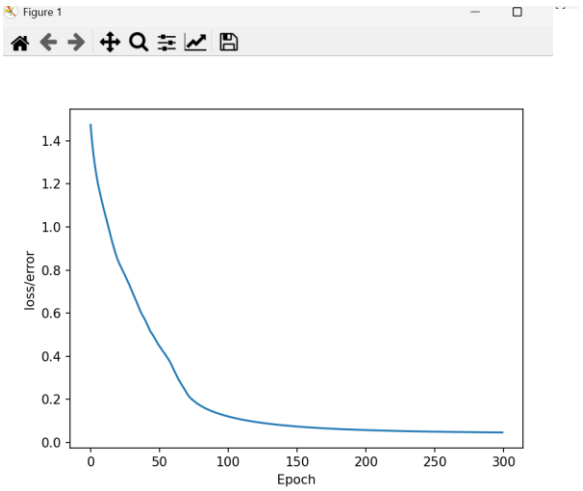
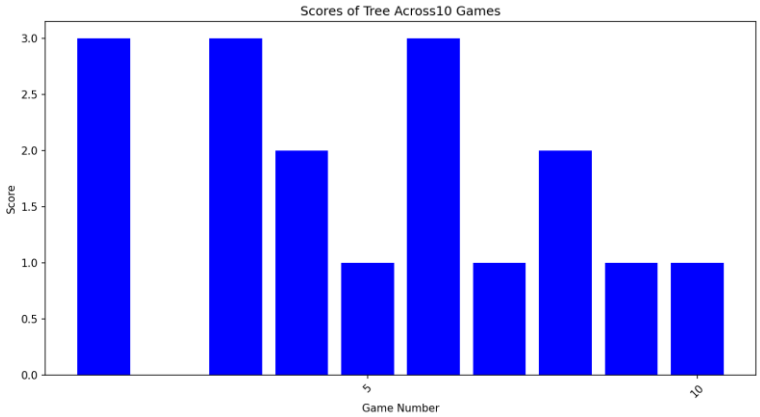
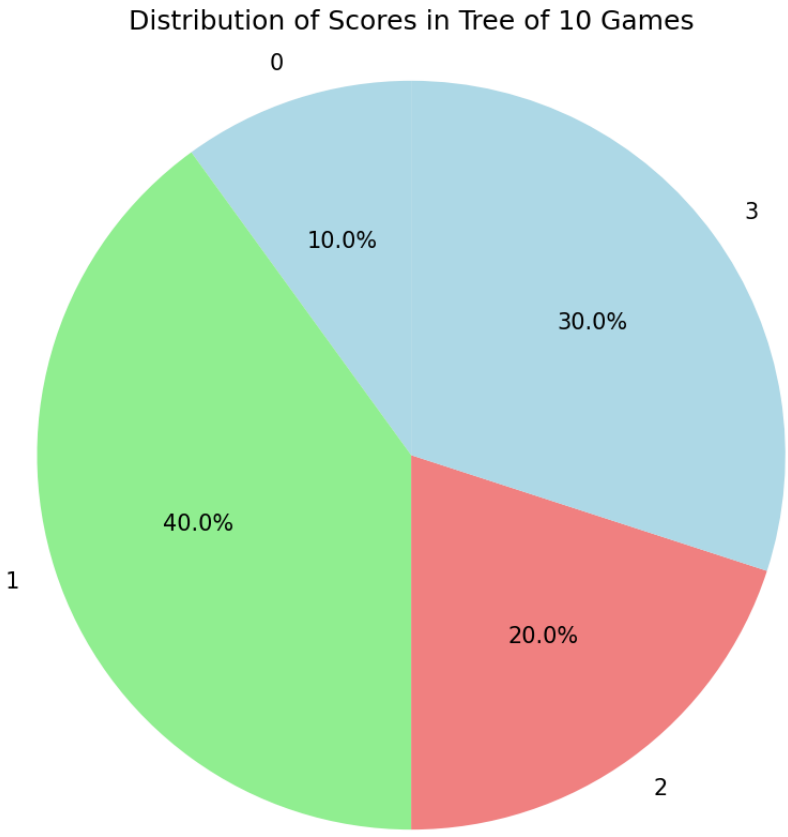


Figure 1
loss/error
Epoch
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
50
100
150
200
250
300


Scores of Tree Across10 Games
Score
Game Number
0.0
0.5
1.0
1.5
2.0
2.5
3.0
5
10


Distribution of Scores in Tree of 10 Games
0
3
1
2
10.0%
30.0%
40.0%
20.0%

N = 3, m = 2, g = 10.

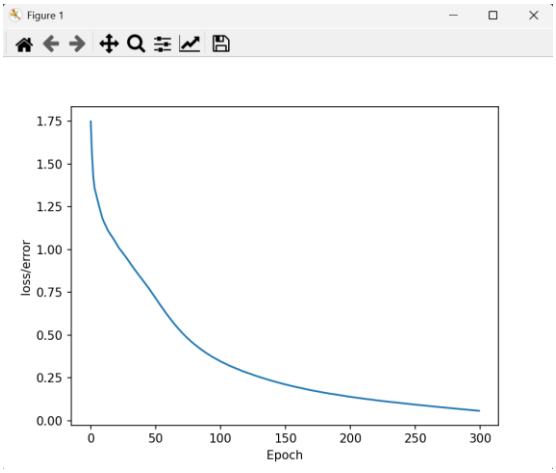
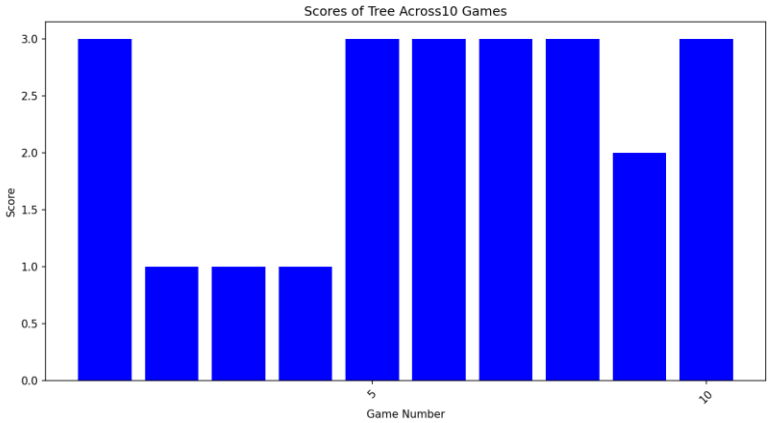
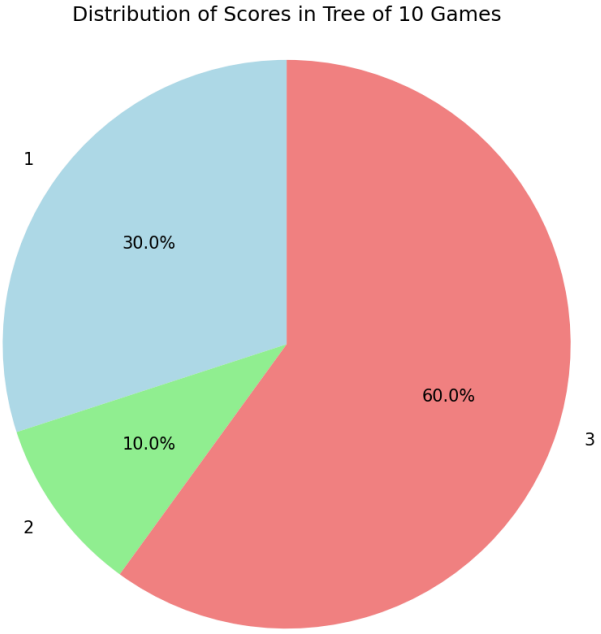
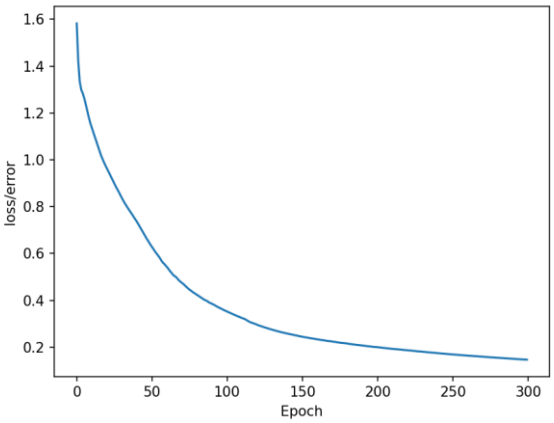


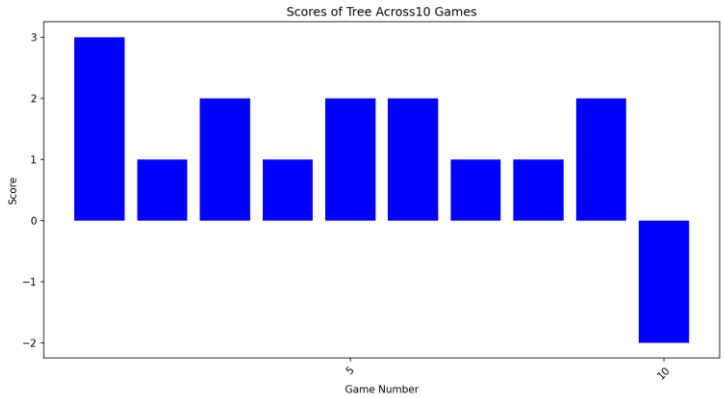
Figure 1
loss/error
Epoch
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
0
50
100
150
200
250
300

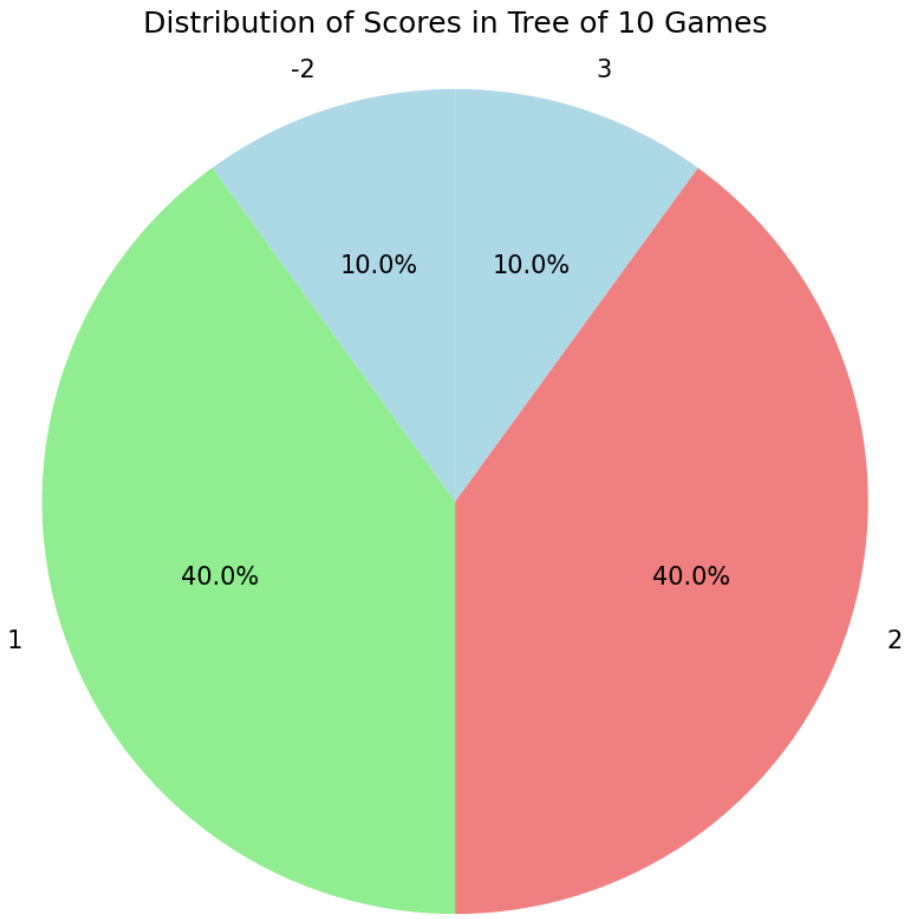

Scores of Tree Across10 Games
Score
Game Number
0.0
0.5
1.0
1.5
2.0
2.5
3.0
5
10


Distribution of Scores in Tree of 10 Games
1
30.0%
60.0%
10.0%
3
2

N = 5, m = 2, g = 10.

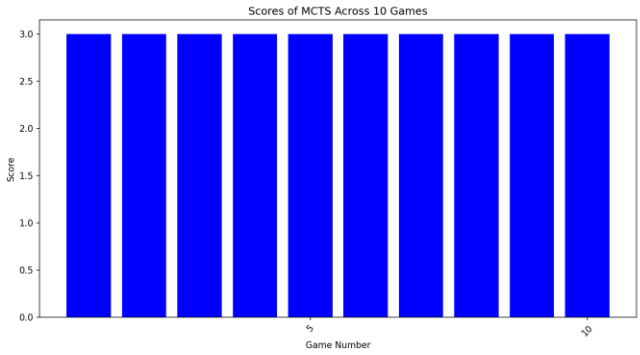

loss/error
Epoch
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0
50
100
150
200
250
300

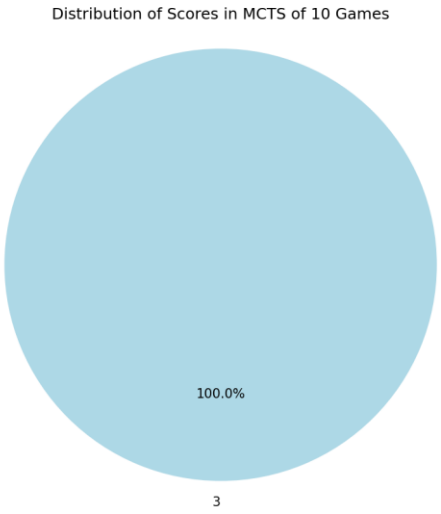

Scores of Tree Across10 Games
Score
Game Number
3
2
1
0
−1
−2
5
10


Distribution of Scores in Tree of 10 Games
-2
3
10.0%
10.0%
40.0%
40.0%
1
2

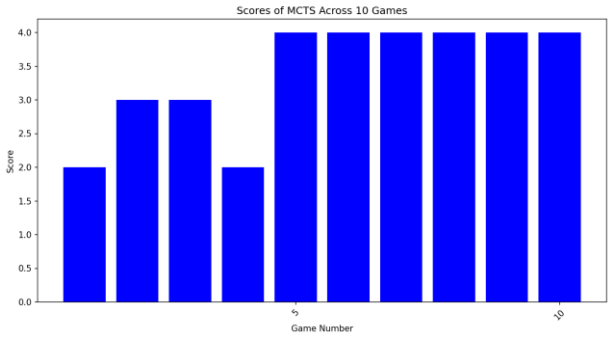
Monte Carlo Tree Search (MCTS)

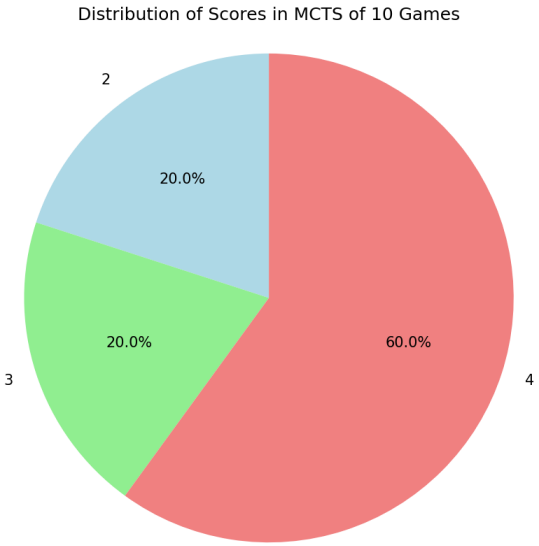
N = 3, m = 1, g = 10.



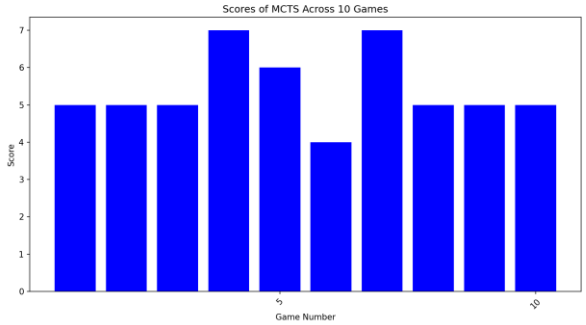


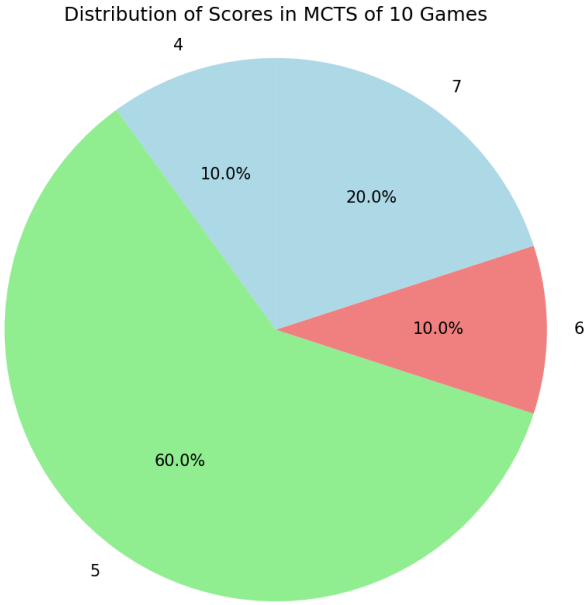
N = 5, m = 2, g =10.



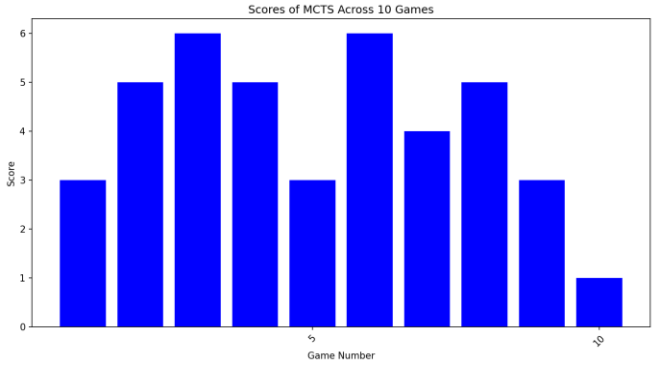


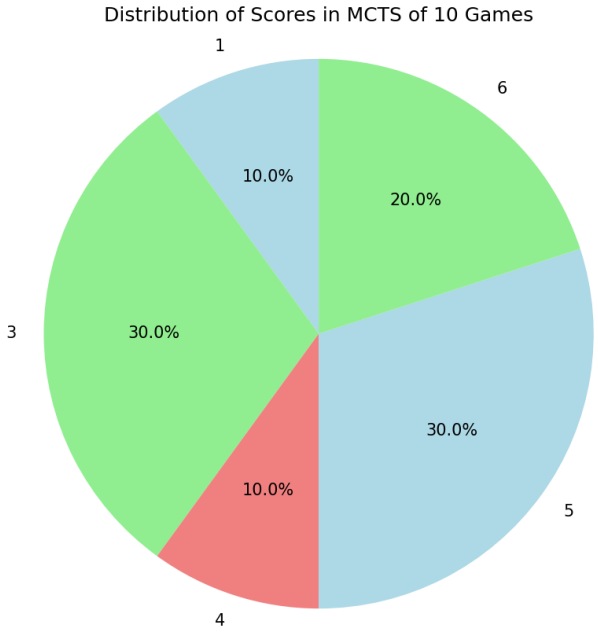
N = 10, m= 1, g = 10.





N = 10, m = 2, g = 10.





Genetic Programming

To solve the Ant Trail problem, we employed a Genetic Programming (GP) approach that evolves a population of candidate movement strategies to guide an artificial ant through a grid-based environment. Each candidate solution is encoded as a **genome**, a fixed-length sequence of binary strings where each pair of bits represents a directional move (00: up, 01: down, 10: left, 11: right). The goal is to maximize the ant's score by efficiently navigating the environment and collecting food or visiting designated tiles.

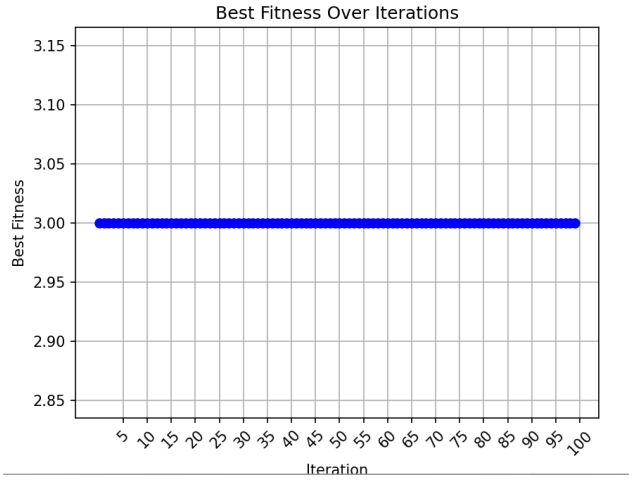
The GP algorithm operates over multiple generations using the standard evolutionary cycle:

- **Initialization**: A random population of genomes is generated.
- **Evaluation (Fitness Function)**: Each genome is decoded and simulated using a game environment ('*GameLogic'* class), and its fitness is computed based on the amount of food collected and the number of moves remaining.
- **Selection**: Tournament selection is used to probabilistically favor higher-performing genomes.
- **Crossover**: Pairs of parent genomes undergo one-point crossover with a fixed probability to create offspring.
- **Mutation**: Individual genes (movement codes) in the genome are mutated at a low rate by flipping their binary codes.

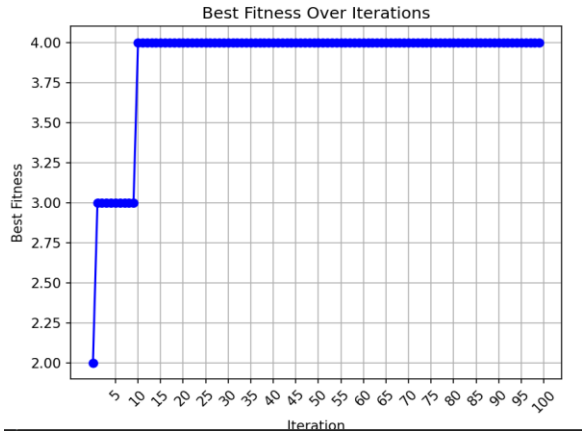
The algorithm keeps track of the best-performing genomes across generations and plots the evolution of fitness over time. The '*real_score*' function also ensures that the final solution's effectiveness is measured accurately by re-evaluating the best genome without considering remaining moves as part of the score.

This method enables the ant to learn optimized paths through evolutionary adaptation. While it does not provide real-time adaptability during navigation, it effectively discovers movement strategies that perform well in static environments over time.

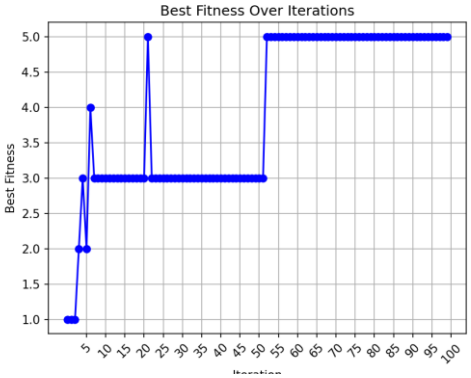
N = 3, m = 1.



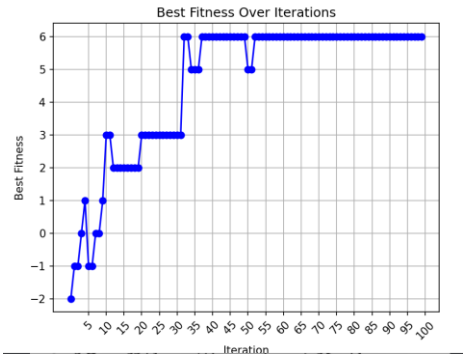
N = 5, m = 1.



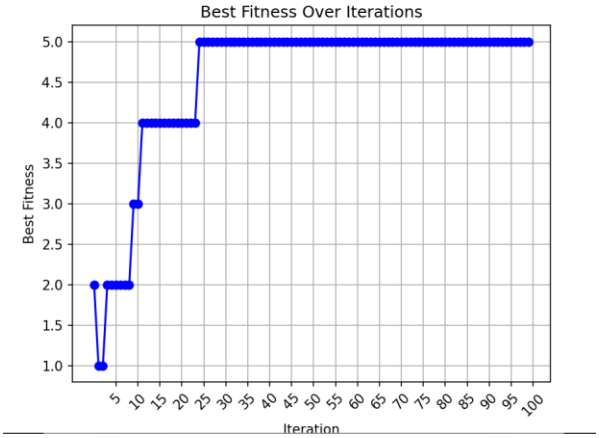
N = 5, m = 2.



N = 10, m = 1.



N = 10, m = 2.



Q-Learning-Based Path Optimization in a Grid World

In this project, I implemented a Q-learning reinforcement learning algorithm to guide an agent (an ant) through a grid-based environment, with the goal of maximizing its score while navigating efficiently. The environment is managed by the GameLogic class, and the agent's progress is visualized using the GameGrphic class. Over 10,000 episodes, the agent learns optimal policies using a Q-table, which is iteratively updated based on the reward received for each action taken. The reward function takes into account both the score achieved and the number of moves remaining, encouraging both effective and efficient solutions. The algorithm also incorporates an exploration-exploitation strategy using a decaying exploration rate to balance learning and performance. The best-performing paths are recorded, and at the end of training, the most successful path is animated to visually demonstrate the learned behavior. This implementation highlights how reinforcement learning can be applied to solve navigation and optimization problems in a dynamic environment.

Conclusion

Overall, I thoroughly enjoyed working on the Ant Trial Problem, particularly exploring and applying two powerful reinforcement learning techniques—Monte Carlo Tree Search (MCTS) and Q-learning. This project not only allowed me to develop a deeper understanding of these algorithms but also gave me hands-on experience in implementing them to solve a non-trivial pathfinding problem. Observing how the agent improved over time and discovered more efficient paths was both rewarding and insightful. This experience has strengthened my interest in reinforcement learning and its potential to solve complex decision-making tasks.