Snake Game Solving using Reinforcement Learning

Anh Tuan Nguyen - 22014593

***Abstract*—Snake Game, a classic arcade game, presents a dynamic and evolving challenge for [1] Reinforcement Learning (RL) algorithms. Despite its straightforward gameplay, the game requires quick decision-making, efficient path-finding, and adaptive strategies to avoid obstacles, grow the snake, and maximize the score. RL, a branch of machine learning, enables intelligent agents to learn optimal actions by maximizing cumulative rewards. In the context of Snake Game, RL provides a powerful framework for developing strategies to navigate the snake while avoiding collisions with walls or itself. This study investigates the application of five RL algorithms—[2] Q-Learning, [3] SARSA, [4] Monte Carlo, [5] Value Iteration, and Policy Iteration—to solve the [7] Snake Game challenge. The evaluation of these algorithms in a custom-built Snake Game environment highlights their individual strengths and weaknesses. In particular, the experiments cover various game scenarios, including randomized food placements and variable snake speeds, to assess the adaptability of the algorithms. The results indicate that [2] Q-Learning and SARSA excel in finding stable policies, while [4] Monte Carlo, [5] Value Iteration, and [6] Policy Iteration demonstrate effectiveness in long-term planning and state-value optimization. The study concludes by emphasizing the potential of these RL techniques to solve the Snake Game and explores avenues for integrating more advanced RL methods to further optimize gameplay.**

***Index Terms*—Snake Game, [1] Reinforcement Learning, [2] Q-Learning, [3] SARSA, [4] Monte Carlo, [5] Value Iteration, [6] Policy Iteration**

I. Introduction

Snake Game, a classic arcade game, has captivated players for decades with its simple yet addictive game play. The game provides an environment where the player must guide a continuously growing snake to eat food, while avoiding collisions with the limits of the game or the snake's own body. Despite its seemingly straightforward nature, Snake Game presents a dynamic and nontrivial challenge that requires real-time decision-making, strategic planning, and adaptability to changing conditions, making it an excellent platform for exploring [1] Reinforcement Learning (RL) algorithms.

[1] Reinforcement Learning, a subfield of machine learning, focuses on training intelligent agents to make sequences of decisions that maximize cumulative rewards. In the context of Snake Game, RL offers a promising method for enabling agents to learn optimal strategies to navigate the snake through complex environments, collect food, and avoid obstacles. The sparse and evolving nature of the Snake Game's state space—where the game's configuration changes continuously as the snake moves and grows—introduces challenges that test the effectiveness of different RL approaches.

The primary objective of this study is to evaluate the performance of five distinctive RL algorithms—**Q-Learning, SARSA, Monte Carlo, Value Iteration, and Policy Iteration**—in solving the Snake Game challenge. These algorithms were chosen for their widespread use in both model-free and model-based RL and their suitability for environments with unknown or partially known dynamics, such as Snake Game. Each algorithm brings a unique approach to balancing exploration and exploitation and updating policies based on immediate or episodic rewards, as well as state-value estimation.

This paper is organized as follows. **Section 2** introduces our Snake Game environment and explains how states, actions, and rewards are defined. **Section 3** provides an overview of the five algorithms we employed—[2] Q-Learning, [3] SARSA, [4] Monte Carlo, [5] Value Iteration, and [6] Policy Iteration—to solve the problem. **Section 4** presents the experimental results, highlighting the strengths and limitations of each method. **Section 5** concludes the paper, summarizing key insights and proposing future research directions to further enhance the capabilities of RL in game-solving contexts.

'0'	Move up
'1'	Move right
'2'	Move down
'3'	Move left

TABLE I: Actions.

II. Snake Game Environment

Before the agent can effectively learn complex algorithms, it requires a suitable environment in which to interact and explore. For the Snake Game, we use a custom-built environment that allows the agent to navigate, gather food, and grow while avoiding collisions. Although the environment provides a solid foundation for testing [1] Reinforcement Learning algorithms, there are still several issues to address in order to improve flexibility and performance. However, one of the most important aspects influencing the agent's learning process is the reward structure. The rewards received after each action, such as gaining points for consuming food or penalties for collisions, play a crucial role in shaping the agent's strategy and overall gameplay. **The reward structure is summarized in Table II below**, outlining how different actions impact the rewards the agent accumulates throughout the game.

Action	**Reward**
Eating food	+10
Each action	0
Colliding with wall or itself	-10

TABLE II: Reward structure for Snake Game.

The environment of the Snake Game is implemented as a grid with a fixed size of 320x240 pixels, where each cell is 20x20 pixels. The agent can perform three main actions at each step: maintain the current direction, turn left, or turn

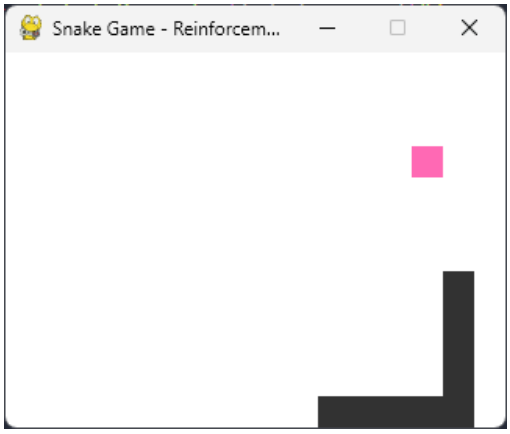

Fig. 1: Map

right. The current state of the game is represented by six values: the relative coordinates of the food compared to the snake's head, the current direction of the snake, and the danger of collision when moving straight, turning left, or turning right. The illustration below (see Figure 1) depicts the game grid structure along with the key state values used during the training process.

III. Reinforcement Learning Algorithms

A. MARKOV'S DECISION PROCESS

1) Definition MDP: A Markov Decision Process (MDP) is a mathematical framework used in artificial intelligence and operations re search to model decision-making processes [1]. It provides a mathematical framework for modeling decision-making in situations where outcomes are partly random andpartly under the control of a decision-maker. MDPs are useful for studying optimization problems solved via dynamic programming. At each time step, the process is in some state, and the decision maker may choose any action that is available in the state. The process responds at the next time step by randomly moving into a new state and giving the decision-maker a corresponding reward. The probability that the process moves into its new state is influenced by the chosen action. An MDP is defined by a 4-tuple (S, A, P, R), with $\gamma \in [0, 1]$, where:

- S is the set of all states.
- A is the set of all actions available.
- P is the transition probability from one state to another.
- R is the reward function obtained from transitioning from one state to another.
- γ is the discount factor.

$$R : S \times A \longrightarrow \mathbb{R}$$

is the reward function.

Figure 2 represents the iteration between the agent and environment. At any given time step t, the agent must base their action, at, on the current state of the environment, st+1, and the agent receives a reward, Rt, from the environment. This cycle of events yields a trajectory (a sequence of states, actions, and rewards) akin to the one depicted in **Figure 3**.

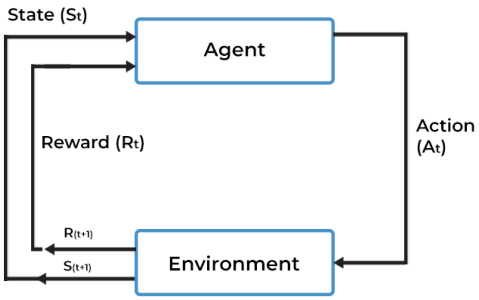

Fig. 2: MDP flowchart cycle

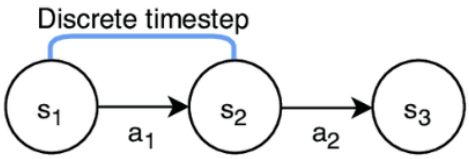

Fig. 3: MDP trajectory

2) Snake Game MDP: To frame Snake Game as an MDP (Markov Decision Process) problem, we need to define all possible states (S), all possible actions (A), the transition function (T), and the reward function (R). In this implementation of the Snake Game environment, the game also includes a maximum number of steps that the agent can take, which serves as an alternate ending condition for the simulation.

The role of the Markov Decision Process (MDP) in the Snake Game is to model the decision-making process of the player (agent) who attempts to navigate the snake (object) in order to consume food (goal) while avoiding collisions with the walls or its own body within a grid-based environment. The MDP framework can help the agent learn an optimal policy (strategy) to maximize the expected cumulative reward (score) over time. By solving the MDP, the agent aims to achieve long-term success by efficiently navigating the game environment.

B. VALUE ITERATION

The [5] Value Iteration Algorithm is a dynamic programming algorithm that can be used to find the optimal policy for a Markov Decision Process (MDP). In Snake Game, the algorithm can be used to find the optimal policy for the game by finding the optimal value function for each state in the game. It also determines which moves will lead to the highest reward and which moves will lead to lower rewards. This **psedocode** below shows a complete algorithm with this kind of termination condition (**Algorithm 1**)

C. POLICY ITERATION

1) Definition: The [6] Policy Iteration Algorithm (a combination of Policy Evaluation and Policy Improvement) is another dynamic programming algorithm used to find the optimal policy. In the Snake Game, this algorithm can be applied to iteratively improve the value function and the policy until convergence. **Policy Evaluation** involves determining

Algorithm 1 Value Iteration Algorithm, for estimating $\pi = \pi^*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation

Initialize $V(s)$, for all $s \in S^+$, arbitrarily except that $V(\text{terminal}) = 0$

loop

$\Delta \leftarrow 0$

for each $s \in S$ **do**

$v \leftarrow V(s)$

$V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

end for

until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi^*$, such that:

$$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$$

the value function for a given policy, which represents the expected long-term reward for each state under that policy. **Policy Improvement** refers to enhancing the current policy by making it greedy with respect to the value function, meaning that at each state, the action that yields the highest expected long-term reward (according to the current value function) is chosen.

2) Formula: Once a policy, π, has been improved using v_π to yield a better policy, π', we can then compute $v_{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_* \tag{1}$$

where $\xrightarrow{E}$ denotes a *policy evaluation* and $\xrightarrow{I}$ denotes a *policy improvement*

A complete algorithm is **pseudocode** given below (**Algorithm 2**).

Algorithm 2 Policy Iteration Algorithm

1) **Initialization**
   $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
   $V(\text{terminal}) = 0$
2) **Policy Evaluation**
   loop
   $\Delta \leftarrow 0$
   Loop for each $s \in S$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} P(s', r|s, \pi(s))[r + \gamma V(s')]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   end loop
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)
3) **Policy Improvement**
   policy-stable $\leftarrow$ true
   For each $s \in S$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} P(s', r|s, a)[r + \gamma V(s')]$
   If old-action $\neq \pi(s)$, **then** policy-stable $\leftarrow$ false
   end loop
   If policy-stable, **then** stop and return $V \approx V^*$ and $\pi \approx \pi^*$;
   else go to 2.

D. First Visit Monte Carlo

I define the [4] Monte Carlo method used in this project as stochastic. [4] Monte Carlo is a model-free approach for learning the state-value function. A 'model-free' method refers to one that does not require prior knowledge of the state transition probabilities. The value of each state is determined by the total reward an agent can expect to accumulate in the future, starting from that state. The First-Visit [4] Monte Carlo method estimates this value function by averaging the returns following all first visits to a given state. The state-value function is defined as follows:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2}$$

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T \tag{3}$$

We have:

- π: policy.
- G_t: weighted return.
- S_t: state at the time step t.
- s: state.
- $\gamma \in [0; 1]$: discount rate.
- R_t: reward obtained at time step t.

So we can simplify this equation to:

$$V(s) \leftarrow V(s) + \frac{1}{N(s)} (G_t - V(s)) \tag{4}$$

Which $N(s)$ is the number of times the agent visits state s. So, after looping in a very large amount of time, we can extract the optimal policy π so the agent can complete the puzzle.

E. Q-Learning

Q-Value Update Rule: The Q-value for a particular state action pair is updated based on the [2] Q-Learning formula:

$$Q(s, a) = (1-\alpha) \cdot Q(s, a) + \alpha \cdot (R + \gamma \cdot \max(Q(s', a'))) \tag{5}$$

- $Q(s, a)$: Q-value for state s and action a.
- α: Learning rate, determining the influence of new information on the current estimate.
- R: Immediate reward obtained after taking action a in state s.
- γ: Discount factor, reflecting the agent's preference for immediate rewards over delayed ones.
- $Q(s', a')$: The maximum Q-value for the next state s' among all possible actions a'.

Epsilon-Greedy Policy: Q-learning employs an epsilon-greedy exploration policy. With probability ϵ, the agent explores by selecting a random action, and with probability

$(1-\epsilon)$, it exploits its current knowledge by selecting the action with the highest Q-value.

In summary,[2] Q-Learning aims to iteratively update Q-values for state-action pairs based on the rewards received and the maximum expected future rewards. Over time, these updates converge to the optimal Q-values that guide the agent to ward making the best decisions in the given environment. The epsilon-greedy strategy balances exploration (trying new actions) and exploitation (choosing known best actions) to gradually improve the policy.

F. SARSA

[3] SARSA is an on-policy [1] Reinforcement Learning algorithm aimed at estimating the optimal action-value function Q(s,a), similar to [2] Q-Learning. The key distinction lies in how it updates the Q-values based on the agent's interaction with the environment. Below is the SARSA algorithm formula:

$$Q(s,a) = (1-\alpha)\cdot Q(s,a) + \alpha\cdot(R + \gamma\cdot\max(Q(s',a'))) \quad (1)$$

where:

- $Q(s,a)$: Q-value for state s and action a.
- α: Learning rate, controlling the step size of the Q-value updates.
- R: Immediate reward obtained after taking action a in state s.
- γ: Discount factor, emphasizing the importance of future rewards.
- $Q(s',a')$: Q-value for the next state-action pair (s',a').

Key Differences between [3] SARSA and [2] Q-Learning:

1) **On-Policy vs. Off-Policy**:
   - 3 SARSA is an on-policy algorithm, meaning it learns the Q-values based on the policy it is currently following during exploration.
   - 2 Q-Learning is an off-policy algorithm, which learns Q-values for an optimal policy but explores using a different policy, typically an epsilon-greedy policy.
2) **Q-Value Update**:
   - 3 SARSA updates the Q-value based on the action actually taken in the next state, following the current exploration policy.
   - 2 Q-Learning updates the Q-value using the action that maximizes the Q-value in the next state, regardless of the action taken during exploration.
3) **Exploration vs. Exploitation**:
   - 3 SARSA uses the same epsilon-greedy exploration policy for both action selection and Q-value updates.
   - 2 Q-Learning typically explores using epsilon-greedy, but it exploits the action with the highest Q-value during the Q-value update.
4) **Stability and Convergence**:
   - 3 SARSA tends to be more conservative, leading to a more stable policy but possibly slower convergence.
   - 2 Q-Learning can be more aggressive in exploration, which may lead to faster convergence but can also result in instability during the learning process.

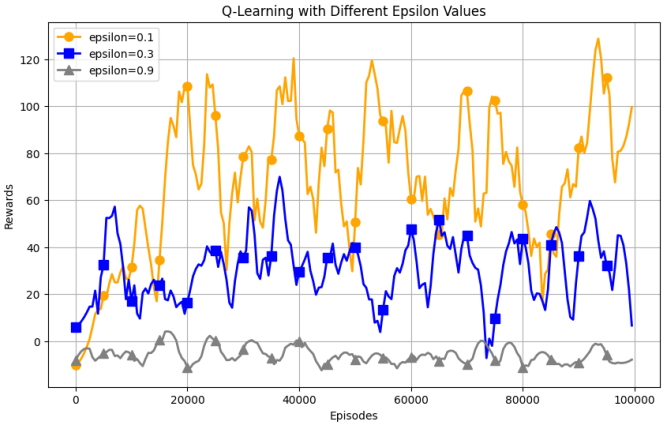

Fig. 4: Q-Learning with Different Epsilon Values

In summary, both [3] SARSA and [2] Q-Learning are widely used [1] Reinforcement Learning algorithms for estimating optimal action-value functions. The key difference lies in how they handle exploration and Q-value updates, with [3] SARSA sticking to the policy it is learning about, while [2] Q-Learning aims for an optimal policy while exploring using a different strategy.

IV. EXPRERIMENT AND EVALUATION

In this section, I delve into the implementation of five [1] Reinforcement Learning algorithms to tackle the Snake Game using custom game environments. Our evaluation metrics include tracking the Q-value differences across training episodes for [4] Monte Carlo, [2] Q-Learning, and [3] SARSA, while also monitoring the return values for each training episode under different epsilon settings. For both [2] Q-Learning, [3] SARSA and [4] Monte Carlo, the learning rate is set to 0.1, and the discount factor is set to 0.9. Unfortunately, we have excluded the results of [5] Value Iteration and [6] Policy Iteration due to their comparatively lower performance in this context. This section is focused on the practical implementation and analysis of the selected [1] Reinforcement Learning algorithms, offering insights into their effectiveness in solving the Snake Game challenge. Our primary aim is to understand the behavior of these algorithms by comparing Q-values and assessing return values, while also acknowledging the limitations observed in [5] Value Iteration and [6] Policy Iteration methods. Below, we present some of the key results from our experiments:

V. CONCLUSION

In this study, I successfully implemented all five fundamental [1] Reinforcement Learning algorithms: [5] Value Iteration, [6] Policy Iteration, [4] Monte Carlo, [2] Q-Learning, and SARSA. After conducting experimental trials and training processes, the results from SARSA, [2] Q-Learning, and [4] Monte Carlo demonstrated promising outcomes in effectively solving different Snake Game environments. Additionally, [2] Q-Learning, which emerged as the most successful algorithm, was applied consistently across various scenarios with stable

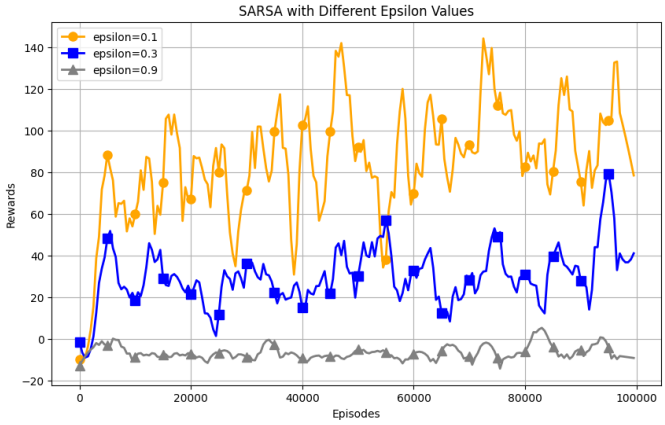

Fig. 5: SARSA with Different Epsilon Values

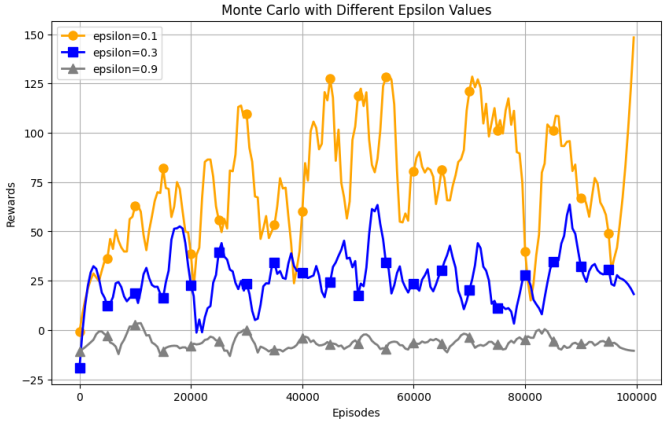

Fig. 6: Monte Carlo with Different Epsilon Values

success. Moving forward, we plan to explore advanced [1] Reinforcement Learning techniques to further enhance our approach in solving this game.

VI. REFERENCES

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd edition.
[2] S. Ravichandiran, *Deep Reinforcement Learning with Python*, 2nd edition.
[3] C. J. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.
[4] N. Sprague and D. Ballard, "Multiple-goal reinforcement learning with modular SARSA (0)," 2003.

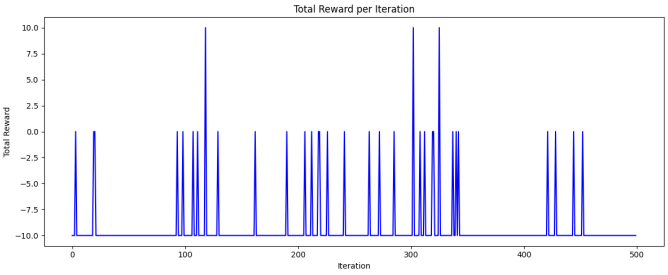

Fig. 7: Value Iteration with iterations=500

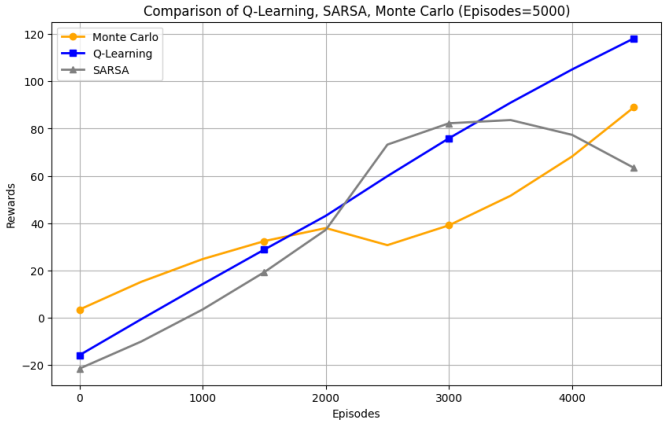

Fig. 8: Comparison of Q-Learning, SARSA, Monte Carlo (Episodes=5000)

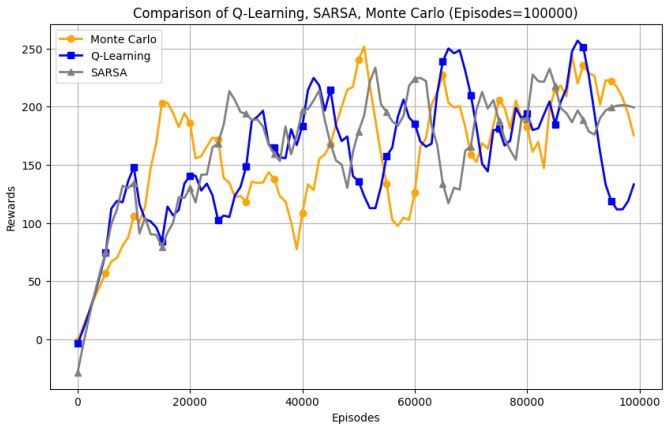

Fig. 9: Comparison of Q-Learning, SARSA, Monte Carlo (Episodes=100000)

[5] Y. Iba, "Population monte carlo algorithms," *Transactions of the Japanese Society for Artificial Intelligence*, vol. 16, no. 2, pp. 279–286, 2001.
[6] S. P. Meyn, "The policy iteration algorithm for average reward Markov decision processes with general state space," *IEEE Transactions on Automatic Control*, vol. 42, no. 12, pp. 1663–1680, 1997.
[7] K. Chatterjee and T. A. Henzinger, "Value iteration," in *25 Years of Model Checking: History, Achievements, Perspectives*. Springer, 2008, pp. 107–138.

REFERENCES