Intelligent Systems - Snake Game

1st David Mendonça
DETI, University of Aveiro
Masters Robotics and Intelligent Systems
Aveiro, Portugal
dmsmua.pt

2nd Rafael Morgado
DETI, University of Aveiro
Masters Robotics and Intelligent Systems
Aveiro, Portugal
rafa.morgado@ua.p

3rd Salomé Marie
DETI, University of Aveiro
Masters Robotics and Intelligent Systems
Aveiro, Portugal
salome.marie@ua.p

***Abstract*—This paper presents the architecture and design of an autonomous agent developed to play the game Snake. The agent uses tree search strategies and adaptive decision-making techniques to improve gameplay performance in both single-player and multiplayer modes. The implemented system balances real-time decision optimization, collision avoidance, and competitive behavior using a modular Python architecture. A detailed analysis of the agent's architecture, strategies, and experimental results is presented, providing insights into decision latency, resource efficiency, and competitive behavior.**

***Index Terms*—Snake Game, Artificial Intelligence, Autonomous Agents, Tree Search, Pathfinding, Adaptive Agents, Multiplayer AI.**

I. Introduction

The Snake game has long served as a benchmark for AI testing due to its dynamic constraints and decision-making challenges. This project aims to develop an autonomous agent capable of playing Snake using heuristic search techniques, pathfinding strategies, and adaptive decision-making. The agent is optimized for both single and multiplayer environments. The goal is to maximize performance by balancing food collection, avoiding collisions, and handling limited visibility constraints effectively.

II. Ease of Use

The Snake AI agent was designed with a strong emphasis on ease of integration, modularity, and maintainability to support both educational use and further development. The architecture ensures the following advantages:

- **Clear Separation of Functional Components:** The system is divided into well-defined modules, including the agent, client, server, and map manager, ensuring minimal code dependencies and simplified debugging.
- **Intuitive Communication Interfaces:** All components communicate via standardized WebSocket protocols, making it easier to expand the system or integrate external tools.
- **Comprehensive Documentation:** The codebase includes detailed comments, function descriptions, and a structured API reference, ensuring clarity for developers working with the project.
- **Plug-and-Play Visualization:** The agent integrates seamlessly with the Pygame-based visualizer, enabling real-time monitoring and debugging without complex setup.
- **Extensibility for Further Research:** The modular design facilitates the testing of alternative decision-making strategies, such as reinforcement learning or different search algorithms, with minimal code adjustments.

III. System Architecture and Development

The Snake AI project was developed with a modular architecture, emphasizing clarity, reusability, and efficiency. The system is structured into multiple components that work together to ensure seamless gameplay and decision-making.

A. Information Flow

- The **Server** sends the **Game State** to the **Client**, which forwards it to the **Agent**.
- The **Agent** processes the information, computes the next move, and sends **Commands** back to the **Server**.
- The **Game** consults the **Map Manager** for environmental validation and applies **Modifications** to the game state.
- The updated game state is sent to the **Viewer** for visualization and to the **Client** for further processing by the agent.

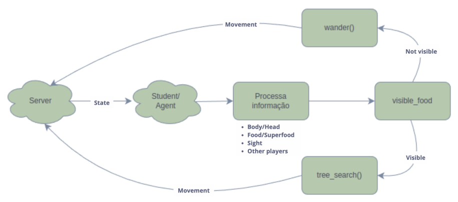

Fig. 1. Flow chart.

B. Agent Module

The `agent.py` module implements the core decision-making logic for the Snake AI. It uses a tree search algorithm combined with a heuristic evaluation to determine the optimal next move. The agent follows these key steps:

- **State Management:** The agent tracks its current position, body, food locations, and obstacles.
- **Decision Process:** A depth-limited tree search algorithm (depth = 10) is used to explore possible moves and their consequences.

Identify applicable funding agency here. If none, delete this.

- **Heuristic Function:** The heuristic prioritizes proximity to food while avoiding collisions with obstacles or its own body.
- **Collision Avoidance:** The function `is_safe_move()` checks if the next move would result in a collision, considering both the agent's body and the map boundaries.
- **Pathfinding Strategies:** The agent uses a basic wander strategy when no food is visible and switches to a targeted search when food is detected within its visible range.

C. Invalid State Handling

The agent ensures the game state remains valid by identifying and avoiding conditions that would render the state invalid:

- **Collisions with own body:** Moving into a previously occupied cell.
- **Collisions with superfoods:** Avoids hazardous items marked as dangerous.
- **Increased Distance to Food:** Discourages moves that increase distance from the nearest food item.
- **Reverse Movements:** Prevents movements directly opposite the current direction.
- **Collisions with Other Snakes:** In multiplayer mode, collisions with other agents are also avoided.

D. Tree Search

The `tree_search()` method implements a limited-depth tree search strategy to determine the optimal next move for the snake agent.

Objective: Explore possible moves from the current position, predicting the consequences of each decision over multiple steps (limited depth) and selecting the best move based on a scoring system.

Functioning:

- **Input:** Search depth, head position, body coordinates, and visible food positions.
- **State Expansion:** At each step, four possible moves are generated (`w`, `a`, `s`, `d`). Moves leading to collisions or increased distance from food are discarded.
- **Evaluation:** The `heuristic()` function assigns scores, penalizing unsafe moves and rewarding proximity to food.
- **Recursion:** The search continues until reaching the maximum depth, progressively evaluating the game state.
- **Return:** The move with the best score is selected. If no move is safe, a default move (`d`) is returned to avoid stagnation.

E. Wander (Controlled Exploration)

The `wander()` method is used when no food is visible within the agent's sight range. It generates a systematic exploration pattern to keep the snake moving safely and prevent it from becoming idle.

Objective: Keep the snake moving safely while searching for food, avoiding collisions, and maintaining exploration.

Functioning:

- **Input:** Sight range, wall traversal mode, body, and hazardous food items.
- **Logic:** The snake initially moves horizontally and then shifts downward upon reaching a boundary. If collision risk is detected, the pattern is dynamically adjusted.
- **Collision Prevention:** The `is_safe_move()` function checks for collision risks and ensures safe movement. The `is_traverse` functionality, while implemented, is currently unused because the agent proactively avoids superfoods, making reverse movements unnecessary in this context. However, in future scenarios where the agent might need to consume a superfood to survive (e.g., in a constrained environment or with limited resources), the `is_traverse` implementation would be crucial to provide strategic flexibility.

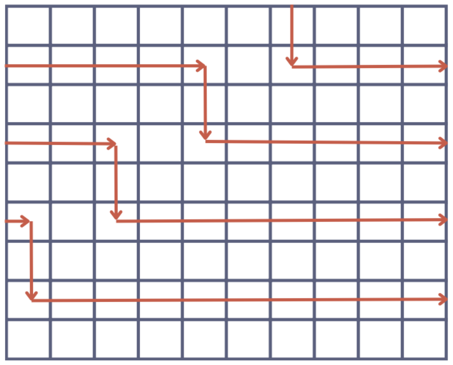

Fig. 2. Wander strategy movement pattern.

F. Heuristic Function

The `heuristic()` function evaluates the quality of each game state during the tree search process, guiding the agent towards favorable decisions.

Objective: Assess the quality of each expanded state during the decision-making process based on food proximity and collision safety.

Functioning:

- **Input:** Head position, body, and food list.
- **Logic:** Calculates the Manhattan distance between the snake's head and the nearest food item. Closer food receives higher priority while collisions are severely penalized.
- **Return:** A negative score proportional to food proximity, with strong penalties for collisions or movements that increase distance from food.

These strategies collectively ensure that the agent can make informed decisions during gameplay, balancing food collection, collision avoidance, and continuous movement.

G. Client Module

The `client.py` module handles the communication between the Snake AI agent and the game server. It uses the WebSocket protocol to exchange game state information and agent decisions:

- Connects to the server and requests the current game state.
- Forwards the game state to the agent for decision-making.
- Sends the selected move back to the server for execution.

H. Server Module

The `server.py` module is responsible for managing the game logic, including:

- Handling multiple players and maintaining a synchronized game state.
- Detecting collisions and managing scoring logic.
- Broadcasting the updated game state to all connected clients.
- Managing player registration and game termination.

I. Map Management Module

The `mapa.py` module is responsible for handling the representation of the game grid, including food, obstacles, and the snake's body. Key functions include:

- Generating the map grid with defined boundaries.
- Randomly placing food and obstacles.
- Updating the map based on the agent's movements.

J. Viewer Module

The `viewer.py` module provides a graphical representation of the game using Pygame. It allows real-time visualization of the Snake game and serves as a debugging tool for the agent's performance. Key features include:

- Displaying the game grid, snake, and food.
- Real-time updates based on the game state received from the server.
- Visualizing the agent's decisions and collisions.

K. Decision-Making Process

The decision-making process is based on a combination of tree search and heuristic evaluation. The `tree_search` method explores potential moves and scores them using a heuristic function. Steps include:

1) Generate all possible actions (`w`, `a`, `s`, `d`).
2) Expand each state and evaluate its score using a heuristic that prioritizes proximity to food and collision safety.
3) Prune invalid states where collisions would occur.
4) Return the action associated with the highest score.

L. Collision Handling and Safety

The `is_safe_move()` function ensures the agent does not collide with itself or obstacles. It checks:

- If the next move would place the snake's head on a previously occupied tile.
- If the move would result in moving into a wall or another player's body.
- If the next move violates the current movement direction (e.g., moving directly backward).

M. Multiplayer Considerations

The `SNAKE_final_multiplayer` version of the agent extends the original implementation to support multiple players. Additional considerations include:

- **Detection of Opponents:** The agent considers the positions of other snakes using the `get_other_snake_positions()` function.
- **Avoidance Strategies:** The agent adjusts its pathfinding strategy to avoid collisions with opponents.
- **Competitive Behavior:** The heuristic was modified to account for racing opponents to food and strategic blocking.

N. Optimization and Improvements

Future enhancements and optimizations could include:

- Implementing a more advanced heuristic function using A* search instead of basic tree search.
- Incorporating reinforcement learning techniques to adapt the agent's strategy over time.
- Enhancing the collision detection mechanism to account for more complex multiplayer interactions.

The modular design and clean codebase make the Snake AI agent a strong foundation for both educational purposes and advanced research in autonomous agent behavior.

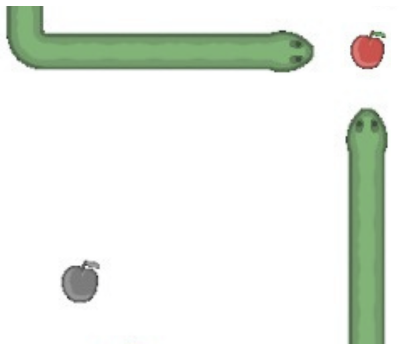

Fig. 3. Snake Game.

IV. Conclusion and Future Work

The autonomous agent demonstrated effective decision-making and competitive performance using tree search and adaptive strategies. The developed modular architecture and adaptive strategies allowed the agent to excel in dynamic game environments.

Future work includes:

- Incorporation of Reinforcement Learning for dynamic strategy adaptation.
- Advanced Cooperative and Competitive Multiplayer Tactics.
- Further Optimization for Reduced Latency.
- Integration with Neural Networks for Complex Decision Making.

References

[1] IEEE Conference Templates: https://www.ieee.org/conferences/publishing/templates.html

[2] Project Repository: https://github.com/detiuaveiro/group-project-snake-2025-group

[3] Pygame Documentation: https://www.pygame.org/docs/