AI Project Proposal

Samank Gupta (G41566207)

Classic Snake Game with a random enemy snake

[Github Link]

Problem Description

We all have played the snake game on a Nokia 3310. In this project, we build an AI agent to play a modified version of the classic snake game. In this version, a small enemy snake appears randomly on the grid for a few seconds. If our snake touches this enemy snake, the game ends. The agent needs to eat food to grow longer and survive as long as possible while avoiding itself, and this enemy.

Uncertainties Involved

The enemy snake appears at a random time and at a random location for a few seconds. The movement of the enemy snake also follows a random pattern. The agent does not know in advance when or where the enemy will appear. The size of the enemy snake will be small but also random. The size of the grid can be given by the user. These make the game unpredictable and more challenging for the AI to plan.

Why This Problem is Non-Trivial

The game becomes dynamic and unpredictable due to the random enemy. Basic search algorithms like BFS or A* cannot handle this kind of uncertainty well. The agent must make decisions over time, thinking not just about the current move but also about future risks and rewards. It needs to balance between eating food quickly and avoiding danger.

Existing Solution Methods

- Policy Iteration / Value Iteration: to find the best moves in a known environment.

- Utility-Based Agents: to make smart decisions based on rewards and risks.

Plan for modeling and solving the problem

State Space: Snake's position and direction, food location, enemy location (if visible), grid size.

Action Space: Turn Left, Move Forward, Turn Right.

Observations: Local view of the grid (like a 5x5 area around the snake), direction of food, enemy snake (if visible).

Rewards:

- +10 for eating food
- -100 for dying (crash or touching enemy)
- -1 for each time step (to avoid waiting too long)

Algorithm:

- Model the game as an MDP and solve using Policy Iteration or Value Iteration on a simplified/smaller grid.
- For larger or more complex grids, use Temporal Difference (TD) Learning to let the agent learn by playing.
- The agent will use utility-based decision-making to choose actions that give the best long-term reward.

State Space Description

Natural language description of state space

In this AI-controlled Snake Game, the agent (our snake) operates in a dynamic grid environment where it must navigate toward food, avoid colliding with itself, and evade a randomly appearing enemy snake. The agent's state is defined by:

- The position and direction of the snake's head and body.
- The location of the food pellet.
- The presence, position, and size of the enemy snake (if currently visible).
- The overall dimensions of the grid.

The agent has limited vision - typically a local 5×5 observation window, and must make decisions based on partial and dynamic information. The enemy's sudden and random appearances introduce uncertainty, making it necessary for the agent to plan with incomplete data.

Complete mathematical description of states, transitions, actions, and observations

Let the environment be a grid of size G = (W, H) where W is the width and H is the height.

State Space (S)

A state $s \in S$ can be defined as:

s = (p_snake, d_snake, p_food, p_enemy, t_enemy) where:

- p_snake = [p_1, p_2, ..., $p_{\square}$] is an ordered list of the snake's body positions (with p_1 being the head).
- d_snake $\in$ {N, E, S, W} is the current direction of the snake (North, East, South, West).
- p_food $\in$ G is the position of the food on the grid.
- p_enemy = [e_1, e_2, ..., $e_{\square}$] is the set of positions occupied by the enemy snake. If the enemy is not visible, p_enemy = null.
- t_enemy $\in$ {0, 1, ..., T} is a timer showing how many steps the enemy will remain visible (0 if not currently present).

Action Space (A)

A = {Turn Left, Move Forward, Turn Right}

Transition Function (T)

- The snake moves deterministically based on the action a.
- The appearance and movement of the enemy snake are stochastic (random).
- If the snake eats food, its body grows.
- If the snake hits the wall, itself, or the enemy snake, the game ends.

Observation Space (O)

At each step, the agent receives an observation:

o = (V_5(p1), d_food, d_enemy) where:

- V_5(p1) is a 5×5 grid centered at the snake's head p1.
- d_food is the relative direction to the food.
- d_enemy is the relative direction to the enemy snake (if visible).

Reward Function (R)

The reward function R(s, a) is defined as:

- +10 for eating food
- -100 for crashing into a wall, itself, or the enemy
- -1 for each time step (to encourage faster decisions)

State Space Implementation

Successor State Generation

In the developed Snake Game model, successor states are generated by applying an action (either from a human player or an AI agent) to the current game state. The **move()** function within the GameState class defines the transition mechanism.

When an action is taken:

- The snake's head moves one step forward based on the current direction, which may be updated by a user keypress or by the AI's decision-making process.
- If the snake collides with its own body or with the randomly moving enemy snake, the game transitions into a terminal state, resulting in the end of the game.
- If the snake's new head position coincides with a food item, the snake grows by one segment and a new food pellet is randomly generated on the grid.
- The snake's body continuously follows the head's movement, maintaining its structure and integrity.
- Wrapping around walls is handled using modular arithmetic, allowing the snake to reappear from the opposite side of the screen upon crossing boundaries.

Thus, the successor state is deterministically generated based on the applied action and the current environment.

Observations from State-Action Transitions

After each state-action transition, observations are collected to monitor the outcome of the action and inform future decision-making.

The key observations recorded after a move are:

- Whether the snake remains alive or whether it has died due to collision with itself or with the enemy snake.
- Whether the snake has eaten a food item, prompting body growth and score increment.
- The new position of the snake's head after movement.
- Whether a collision occurred with the randomly moving enemy snake.
- Whether the victory condition has been achieved by reaching a predefined score (e.g., eating 25 food pellets).

These observations are derived from the updated game state immediately after executing the move() function.

The observed information can be conceptually summarized as:

- Alive status (True/False)
- Food consumption status (True/False)
- Current head position (x, y)
- Enemy collision status (True/False)
- Current food position (fx, fy)
- Current score (integer value)

Although these observations are not explicitly packaged into a single return object, they are accessible through the updated game state variables and are used internally to determine subsequent moves, update the score display, and handle winning or game-over conditions.