Footsies

This notebook implements different algorithms as player types for a simple adversarial turn-based game inspired by the fighting game Footsies.

The game is implemented in text-based form in the `game.py` file. Players are defined by an abstract class, and all contain an `act()` method that takes as input the current state of the game and outputs a Move.

In [ ]:
```
import math
import random
from game import Move, State, Player, MoveSelection, Footsies
```

Manual player

First, we define a player type that can be controlled through keyboard input. It also informs the human player of the state of the game in a more detailed way ; Later algorithms are provided the same information.

In [ ]:
```
class ManualPlayer(Player):
    name = ""
    def __init__(self, name: str = "Manuel"):
        self.name = name

    def act(self, game_state: State) -> Move:
        print(f"Your turn, {self.name}!{f' {game_state.rounds_left} rounds l
        print(f"You have {game_state.own_blocks} blocks left{' and have lanc
        print(f"Your opponent has {game_state.other_blocks} blocks left{' ar

        move_str = ""
        while move_str not in MoveSelection:
            move_str = input("Choose your move: ")

        return MoveSelection[move_str]
```

Random player

As a test, and as a way to measure the effectiveness of the different algorithms, we implement a class that selects a move at random.

In [ ]:
```
class RandomPlayer(Player):
    name = "The Chaotic"

    def act(self, game_state: State) -> Move:
        return random.choice(list(MoveSelection.values()))
```

Simple counter strategy

Since some specific decisions in the game states are definite wins or losses, we can create a simple algorithm that avoids these bad decisions at all costs, at the risk of predictability.

In [ ]:
```
class CounterPlayer(Player):
    name = "The Simple-Minded"

    def act(self, game_state: State) -> Move:
        if game_state.other_has_attack:
            return MoveSelection["b"]  # Always block if the opponent has at

        if game_state.other_blocks == 0:
            return MoveSelection["a"]  # If they can't block, attack them

        if game_state.other_previous_move == None:
            return random.choice(list(MoveSelection.values()))

        if game_state.other_previous_move == MoveSelection["a"]:
            return MoveSelection["b"]  # If they attacked last, block
        if game_state.other_previous_move == MoveSelection["b"]:
            return MoveSelection["g"]  # If they blocked last, grab
        if game_state.other_previous_move == MoveSelection["g"]:
            return MoveSelection["a"]  # If they grabbed last, attack
```

Bayes Inferences

In [ ]:
```
from collections import defaultdict

class BayesianPlayer(Player):
    name = "The Statistician"

    def __init__(self, alpha: float = 1.0, risk_threshold: float = 0.2):
        """Alpha is the smoothing factor for Bayesian updates. Risk threshol
        self.alpha = alpha
        self.risk_threshold = risk_threshold
        self.opponent_history = defaultdict(lambda: self.alpha)  # Prior wit
        self.total_moves = self.alpha * len(MoveSelection)  # Initial sum of

    def update_beliefs(self, opponent_move: Move):
        """Updates the belief distribution based on the opponent's last move
        self.opponent_history[opponent_move] += 1
        self.total_moves += 1

    def predict_opponent_move(self) -> Move:
        """Predicts the opponent's next move using Bayesian inference."""
        probabilities = {move: count / self.total_moves for move, count in s
        return max(probabilities, key=probabilities.get)  # Most probable mo

    def best_response(self, predicted_move: Move) -> Move:
```

```python
        """Chooses the best response to the predicted move, considering Drag
        if predicted_move == MoveSelection["a"]:
            return MoveSelection["b"]  # Block an attack
        if predicted_move == MoveSelection["b"]:
            return MoveSelection["g"]  # Grab a blocker
        if predicted_move == MoveSelection["g"]:
            return MoveSelection["a"]  # Attack a grabber

        # Introduce Dragon Punch when the opponent is too predictable
        highest_prob = max(self.opponent_history.values()) / self.total_move
        if highest_prob >= self.risk_threshold:
            return MoveSelection["dp"]  # Risky but rewarding option

        return random.choice(list(MoveSelection.values()))  # Default: mix i

    def act(self, game_state: State) -> Move:
        if not game_state.other_previous_move:
            return random.choice(list(MoveSelection.values()))

        self.update_beliefs(game_state.other_previous_move)
        predicted_move = self.predict_opponent_move()
        return self.best_response(predicted_move)
```

MCTS

A Monte Carlo Tree Search relies on simulating the game starting from its current state to determine which decision will lead to the best outcome. This allows for a probabilistic approach that would usually require some reinforcement training.

In [ ]:

```python
from collections import defaultdict

class MCTSPlayer(Player):
    name = "The Clairvoyant"

    def __init__(self, simulations: int = 100, exploration: float = 1.4):
        self.simulations = simulations
        self.exploration = exploration
        self.wins = defaultdict(int)
        self.visits = defaultdict(int)

    def simulate(self, move: Move, state: State) -> float:
        """ Runs a short simulation and returns a score (-1, 0, or 1). """
        p1_blocks, p2_blocks = state.own_blocks, state.other_blocks
        p1_attack, p2_attack = state.own_has_attack, state.other_has_attack
        rounds_left = state.rounds_left

        # First move interaction
        opponent_move = random.choice(list(MoveSelection.values()))
        result = self.evaluate_move(move, opponent_move, p1_blocks, p2_block
        if result != 0: return result  # Immediate win/loss

        # Rollout for a few rounds ahead
        for _ in range(min(5, rounds_left)):  # Look ahead 3 rounds max
```

```
            move = random.choice(list(MoveSelection.values()))
            opponent_move = random.choice(list(MoveSelection.values()))
            result = self.evaluate_move(move, opponent_move, p1_blocks, p2_b
            if result != 0:
                return result  # Stop if we determine a clear outcome

        return 0  # Default to neutral if inconclusive

    def evaluate_move(self, move: Move, opponent_move: Move, p1_blocks, p2_b
        """ Determines the outcome of a move interaction. """
        match (move.value - opponent_move.value):
            case 1: p1_blocks -= 1
            case -1: p2_blocks -= 1
            case 2: return -1
            case -2: return 1
            case 3: return -1 if p2_attack else 0.5
            case -3: return 1 if p1_attack else 0.5
            case 6: return -1
            case -6: return 1
            case _: return 1 if move.value > opponent_move.value else -1

        return -1 if p1_blocks <= 0 else (1 if p2_blocks <= 0 else 0)

    def act(self, game_state: State) -> Move:
        """ Selects the best move using MCTS with UCB1 exploration. """
        total_simulations = sum(self.visits.values()) + 1

        for _ in range(self.simulations):
            move = random.choice(list(MoveSelection.values()))  # Explore al
            result = self.simulate(move, game_state)
            self.wins[move] += result
            self.visits[move] += 1

        return max(MoveSelection.values(), key=lambda move: self.ucb1(move,

    def ucb1(self, move: Move, total_simulations: int) -> float:
        """ UCB1 formula to balance exploration & exploitation. """
        if self.visits[move] == 0:
            return float("inf")  # Always explore unvisited moves
        win_rate = self.wins[move] / self.visits[move]
        return win_rate + self.exploration * math.sqrt(math.log(total_simula
```

Testing

In [ ]:

```
player1 = RandomPlayer()
player2 = BayesianPlayer()
game = Footsies(player1, player2)

player1success = 0
player2success = 0
for i in range(0, 10000):
    game = Footsies(player1, player2)
    result = game.start()
    if result == 1:
```

```
        player1success += 1
    else:
        player2success += 1

print(player1success, player2success)
```