Specification

Adversarial turn-based game inspired by the fighting game Footsies.

Basic rules

A round happens in successive turns, where each of the players select an option. How the options interact determines whether the game continues for another turn, or ends with one of the players winning. Playing the game can happen over multiple rounds or a single one.

Actions

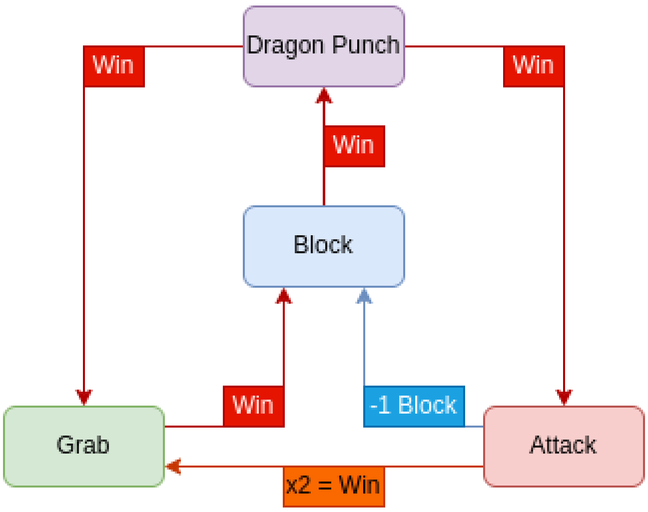
There are four possible options available at the start of the game :

- Attack
- Block
- Grab
- Dragon Punch

Block is limited to three uses per round against Attacks, as will be covered in the next section.

Interaction

Interaction can be summarized by this chart:



Attack wins only against Grabs, and if a player lands two attacks in succession, they win. If the opponent Blocks, the game continues and one use of Block is deducted from them. If the opponent Dragon Punches, the player loses.

Block completely stops both Attacks and Dragon Punches. If the opponent Attacks, the game continues, and one use is consumed, out of three per round. If the opponent Dragon Punches, the player counters it and wins. If the opponent grabs, the player loses.

Grab wins only against Block, and will instantly grant a win. As mentioned earlier, Grabs lose to attacks, and if hit twice, the player loses.

Finally, Dragon Punches are special attacks that win instantly against Attacks and Grabs, and lose instantly to Blocks.

Implementation

The game is implemented inside the `game.py` file, then experimented on in the `footsies.ipynb` notebook.

Problems encountered

Nash Player

One of the first strategies we tried to implement as a simple solution was establishing a Nash Equilibrium, a set of probabilities for each move calculated using linear programming from what's called a payoff matrix. To be more specific, a Nash Equilibrium is a set of stragegies for both players from which neither would want to deviate.

In our case, the obtained payoff matrix looked like this :

```
np.array([
    [  0,   1, -1,   1],   # Attack       vs (Attack, Block, Grab, Dragon Punch)
    [ -1,   0,   1, -1],   # Block        vs (Attack, Block, Grab, Dragon Punch)
    [  1, -1,   0,   1],   # Grab         vs (Attack, Block, Grab, Dragon Punch)
    [  1, -1,   1,   0],   # Dragon Punch vs (Attack, Block, Grab, Dragon Punch)
])
```

In terms of game theory, it is considered "degenerate" : There exists situations in which two responses could be best. This is why it's considered a bluffing game, since if our game had a designated best option for every other option, it would amount to rock-paper-scissors (with an extra choice). However, as we weren't aware before digging deeper, this actually makes the computation of a Nash Equilibrium impossible, since in most situations, players will want to change up their options. As such, we dropped this player type.

game.py

```
from abc import ABC,abstractmethod
from dataclasses import dataclass

class Move:
    def __init__(self, value: int, name: str):
        self.value = value
        self.name = name

@dataclass
class State:
    other_previous_move: Move
    own_blocks: int
    other_blocks: int
    own_has_attack: bool
    other_has_attack: bool
    rounds_left: int

MoveSelection = {
    "a": Move(1, "Attack"),
    "b": Move(2, "Block"),
    "g": Move(4, "Grab"),
    "dp": Move(8, "Dragon Punch")
}

class Player(ABC):
    @property
    @abstractmethod
    def name(self):
        pass

    @abstractmethod
    def act(self, game_state: State) -> Move:
        pass

class Footsies:
    def __init__(self, player1: Player, player2: Player, rounds: int = 1, blocks: int = 3, attackstowin: int = 2, timeout: int = 0):
        self.p1 = player1
        self.p2 = player2
        self.rounds = rounds
        self.attacks = attackstowin

        self.p1_blocks = blocks
        self.p2_blocks = blocks
        self.p1_has_attack = False
        self.p2_has_attack = False
        self.p1_lose = False
        self.p2_lose = False
```

```python
        self.p1_previous: Move = None
        self.p2_previous: Move = None

        self.timeout = False
        self.timeout_rounds = 0
        self.current_round = 1
        if timeout > 0:
            self.timeout = True
            self.timeout_rounds = timeout
            self.current_round = 0

    def start(self) -> int:
        '''Starts the game loop until a player wins or there's a timeout. Returns the number of the player that won. '''

        def no_timeout():
            return True

        def timeout():
            self.current_round += 1
            print(f"Round {self.current_round}/{self.timeout_rounds}")
            return self.current_round <= self.timeout_rounds

        condition = None
        if self.timeout:
            condition = timeout
        else:
            condition = no_timeout

        while condition():
            rounds_left = self.timeout_rounds - self.current_round
            p1_state = State(self.p2_previous, self.p1_blocks, self.p2_blocks, self.p1_has_attack, self.p2_has_attack, rounds_left)
            p2_state = State(self.p1_previous, self.p2_blocks, self.p1_blocks, self.p2_has_attack, self.p1_has_attack, rounds_left)
            move1 = self.p1.act(p1_state)
            move2 = self.p2.act(p2_state)
            self.p1_previous = move1
            self.p2_previous = move2

            print(f"{self.p1.name} chose {move1.name}. {self.p2.name} chose {move2.name}.")

            p1_hit_attack = False
            p2_hit_attack = False

            match (move1.value - move2.value):
                case 0:
                    print("Same option chosen!")
                case 1:
                    print("Player 1 blocks a hit!")
```

```python
                    self.p1_blocks -= 1
                case -1:
                    print("Player 2 blocks a hit!")
                    self.p2_blocks -= 1
                case 2:
                    print("Player 2 gets thrown!")
                    self.p2_lose = True
                case -2:
                    print("Player 1 gets thrown!")
                    self.p1_lose = True
                case 3:
                    print("Player 2 lands a hit!")
                    if self.p2_has_attack:
                        self.p1_lose = True
                    p2_hit_attack = True
                case -3:
                    print("Player 1 lands a hit!")
                    if self.p1_has_attack:
                        self.p2_lose = True
                    p1_hit_attack = True
                case 6:
                    print("Player 2 blocks the Dragon Punch and counters!")
                    self.p1_lose = True
                case -6:
                    print("Player 1 blocks the Dragon Punch and counters!")
                    self.p2_lose = True
                case _:
                    if move1.value > move2.value:
                        print("Player 1 lands a Dragon Punch!")
                        self.p2_lose = True
                    else:
                        print("Player 2 lands a Dragon Punch!")
                        self.p1_lose = True

            self.p1_has_attack = p1_hit_attack
            self.p2_has_attack = p2_hit_attack

            if self.p1_lose:
                print("Player 2 wins")
                return 2

            if self.p2_lose:
                print("Player 1 wins")
                return 1

        return 0
```

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:
```

```
        """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 [ ]:
```
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)
```