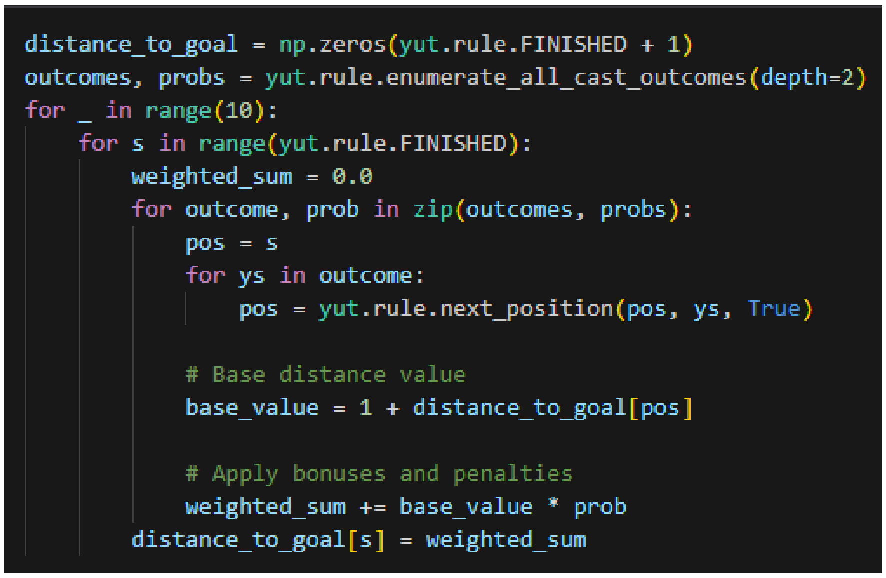
CoE202 FINAL PROJECT REPORT

Team ID - 202411
20210776 Akira Mishima
20220929 Kyaw Ye Thu
20210900 Zainab Sheikh

A. MinimaxPlayer

Our team decided to implement a Minimax algorithm with advanced heuristics and alpha-beta pruning. Our algorithm initializes distance to the goal for each position in the game. The distance itself is then multiplied by its corresponding probability of the outcome to obtain the expected value. We implemented a similar code as the `ExamplePlayer()` but further modified the magnitude of the weight. The initialization value code is as follows:

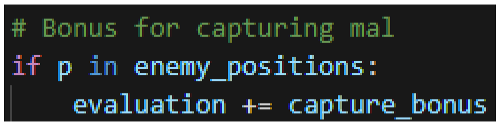


```python
distance_to_goal = np.zeros(yut.rule.FINISHED + 1)
outcomes, probs = yut.rule.enumerate_all_cast_outcomes(depth=2)
for _ in range(10):
    for s in range(yut.rule.FINISHED):
        weighted_sum = 0.0
        for outcome, prob in zip(outcomes, probs):
            pos = s
            for ys in outcome:
                pos = yut.rule.next_position(pos, ys, True)

            # Base distance value
            base_value = 1 + distance_to_goal[pos]

            # Apply bonuses and penalties
            weighted_sum += base_value * prob
        distance_to_goal[s] = weighted_sum
```

The class `MinimaxPlayer(yut.engine.Player)` itself only has three important functions, namely:

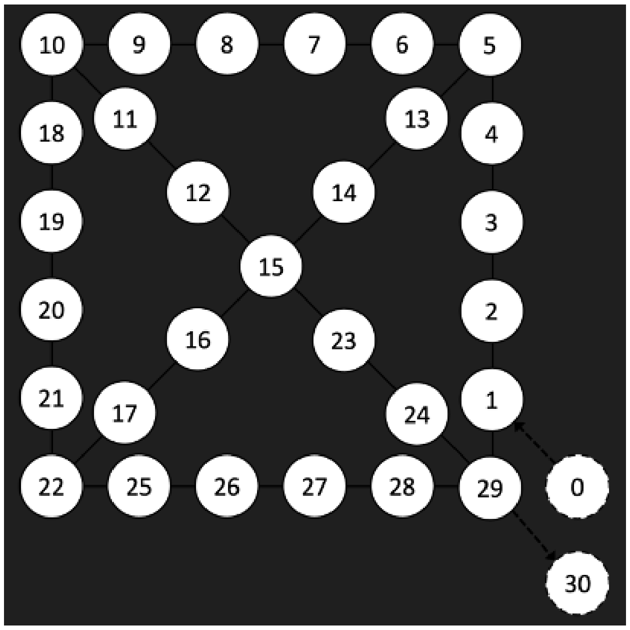
1. `evaluate_board_state()`
   The `evaluate_board_state()` has 4 arguments (including self):
   `my_positions, enemy_positions, mal_caught`

   In this function, we introduced several new heuristics such as prioritizing capturing the opponent's mal and avoiding risky positions. In order to detect whether the move is capturing a mal or not, we can check from the number `mal_caught`. If `mal_caught > 0`, `score += bonus`. Alternatively, we did use a traditional way of checking it.

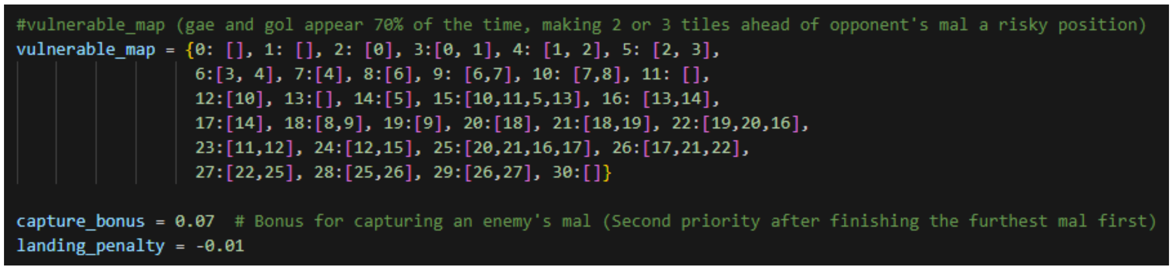


```python
# Bonus for capturing mal
if p in enemy_positions:
    evaluation += capture_bonus
```

To judge whether a tile is risky or not, we mapped each tile with each set of threatening tiles. Since the probability of dice outcome is given, we understood that 2 ('gae', 34.56%), and 3 ('geol', 34.56%) have a really high chance of being rolled. We tried to also incorporate the possibility of getting a yut and its following roll but it turned out to be unnecessary (instead lowering the performance).

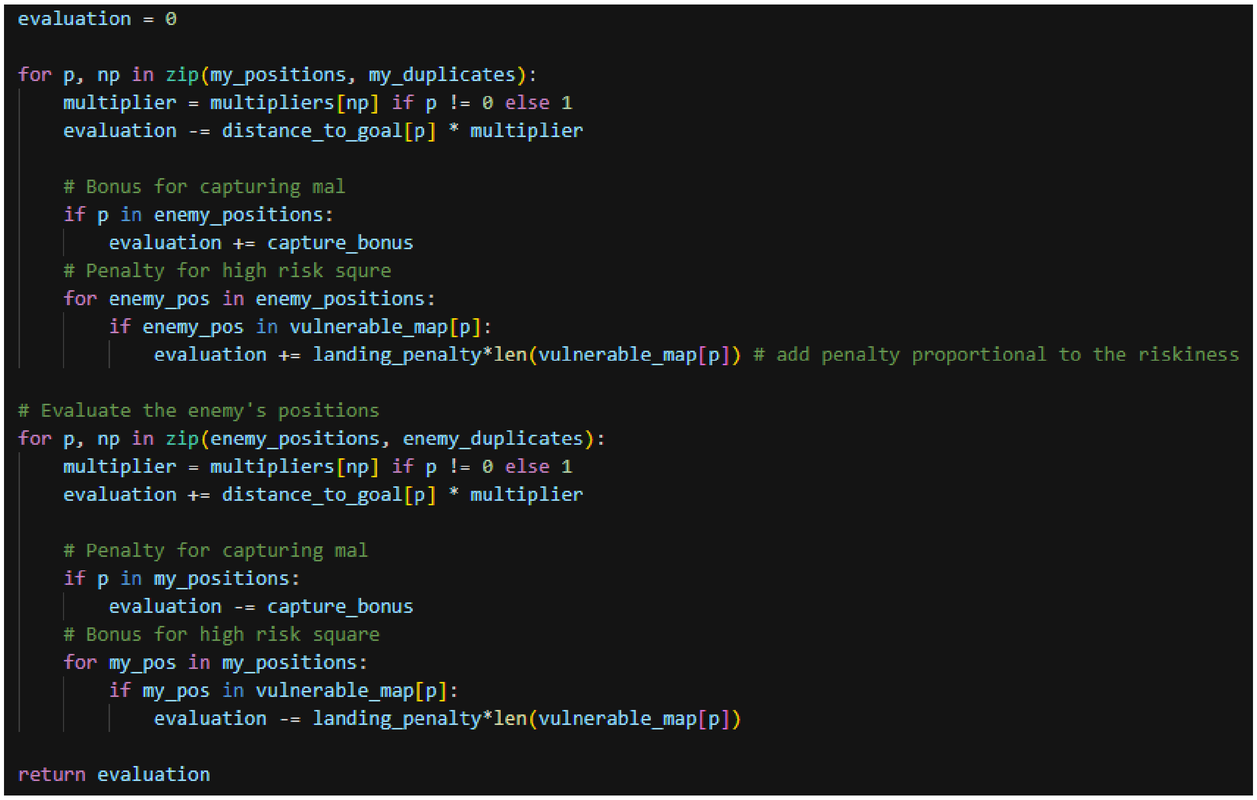


We created a dictionary called `vulnerable_map` which maps each tile to its corresponding weaknesses. For instance, tile 9 is vulnerable to tile 6 and 7 (not because seven ate nine) because 9 is two and three tiles ahead of 7 and 6 respectively. 14 only has weakness tile 5 because it is 2 steps ahead of 5 while tile 4 is not a weakness because rolling a geol at 4 would result in the mal landing on tile 7, not 14. To clearly visualize the map, here is the code for the dictionary.



```python
#vulnerable_map (gae and gol appear 70% of the time, making 2 or 3 tiles ahead of opponent's mal a risky position)
vulnerable_map = {0: [], 1: [], 2: [0], 3:[0, 1], 4: [1, 2], 5: [2, 3],
                  6:[3, 4], 7:[4], 8:[6], 9: [6,7], 10: [7,8], 11: [],
                  12:[10], 13:[], 14:[5], 15:[10,11,5,13], 16: [13,14],
                  17:[14], 18:[8,9], 19:[9], 20:[18], 21:[18,19], 22:[19,20,16],
                  23:[11,12], 24:[12,15], 25:[20,21,16,17], 26:[17,21,22],
                  27:[22,25], 28:[25,26], 29:[26,27], 30:[]}

capture_bonus = 0.07  # Bonus for capturing an enemy's mal (Second priority after finishing the furthest mal first)
landing_penalty = -0.01
```

The scoring is then done by calculating the distance from the goal for both players added by the bonus and penalty from capturing and landing on tiles respectively. The resulting score (evaluation) is the difference between our player's state score and opponent's state score. The optimal move then would be selected by the maximum score (if it's our turn) and minimum score (if it's the opponent's turn) in the minimax function.



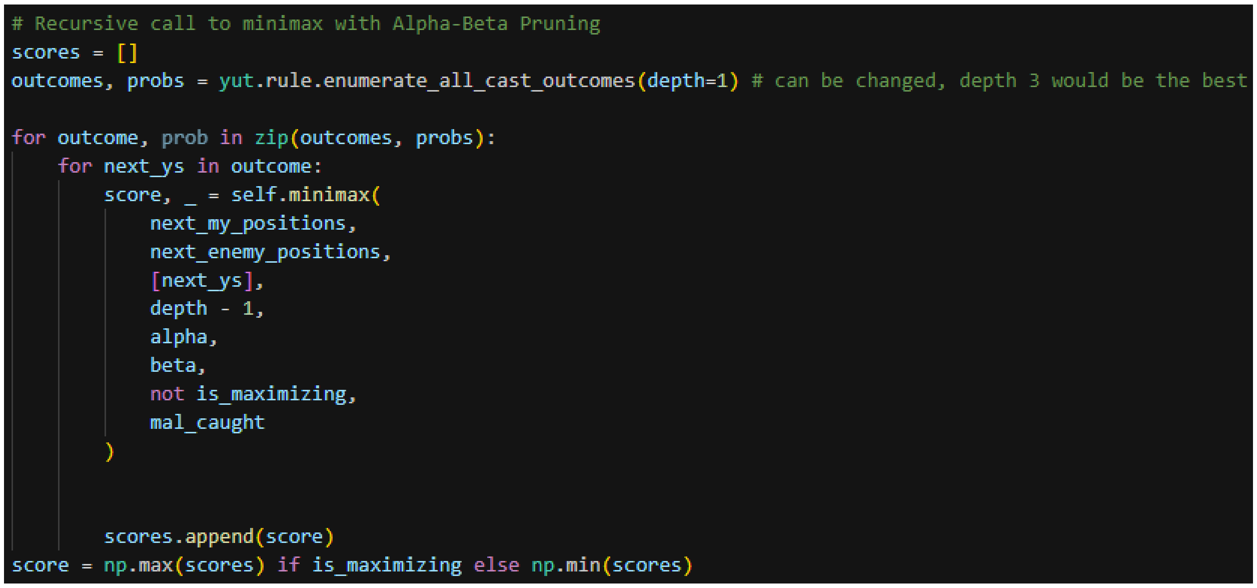
```python
evaluation = 0

for p, np in zip(my_positions, my_duplicates):
    multiplier = multipliers[np] if p != 0 else 1
    evaluation -= distance_to_goal[p] * multiplier

    # Bonus for capturing mal
    if p in enemy_positions:
        evaluation += capture_bonus
    # Penalty for high risk squre
    for enemy_pos in enemy_positions:
        if enemy_pos in vulnerable_map[p]:
            evaluation += landing_penalty*len(vulnerable_map[p]) # add penalty proportional to the riskiness

# Evaluate the enemy's positions
for p, np in zip(enemy_positions, enemy_duplicates):
    multiplier = multipliers[np] if p != 0 else 1
    evaluation += distance_to_goal[p] * multiplier

    # Penalty for capturing mal
    if p in my_positions:
        evaluation -= capture_bonus
    # Bonus for high risk square
    for my_pos in my_positions:
        if my_pos in vulnerable_map[p]:
            evaluation -= landing_penalty*len(vulnerable_map[p])

return evaluation
```

2. `minimax()`

   The `minimax()` function has 9 arguments (including self):
   `my_positions, enemy_positions, available_yutscores, depth, alpha, beta, is_maximizing, mal_caught`

   - `my_positions` and `enemy_positions` are lists of length 4, with each mal's position
   - `available_yutscores` is the number rolled from the dice
   - Depth is the depth of the search for the minimax algorithm
   - Alpha and beta are constants for pruning
   - `is_maximizing` refers to which turn (if `is_mximizing = True`: It's our turn)
   - `mal_caught` how many mals captured (eventually not used)

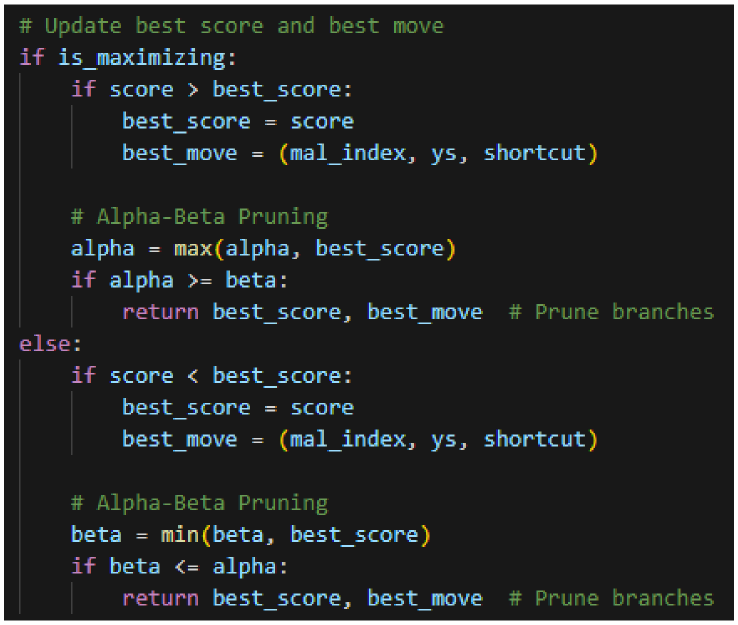
   The function starts with the base case where it checks the remaining depth to search, when it's zero, we are done. The function then proceeds to do a search on every outcome of the dice with `depth=1`. For every outcome, it evaluates recursively until the minimax hits `depth=0` and adds scores of every state it searched to the list. Depending on the `is_maximizing`, it selects the maximum or minimum if `is_maximizing` is True and False respectively.



```python
# Recursive call to minimax with Alpha-Beta Pruning
scores = []
outcomes, probs = yut.rule.enumerate_all_cast_outcomes(depth=1) # can be changed, depth 3 would be the best

for outcome, prob in zip(outcomes, probs):
    for next_ys in outcome:
        score, _ = self.minimax(
            next_my_positions,
            next_enemy_positions,
            [next_ys],
            depth - 1,
            alpha,
            beta,
            not is_maximizing,
            mal_caught
        )


        scores.append(score)
score = np.max(scores) if is_maximizing else np.min(scores)
```

After the score is selected, update the alpha beta values for pruning. If the score is out of the pruning range, it will be automatically returned (forgotten).

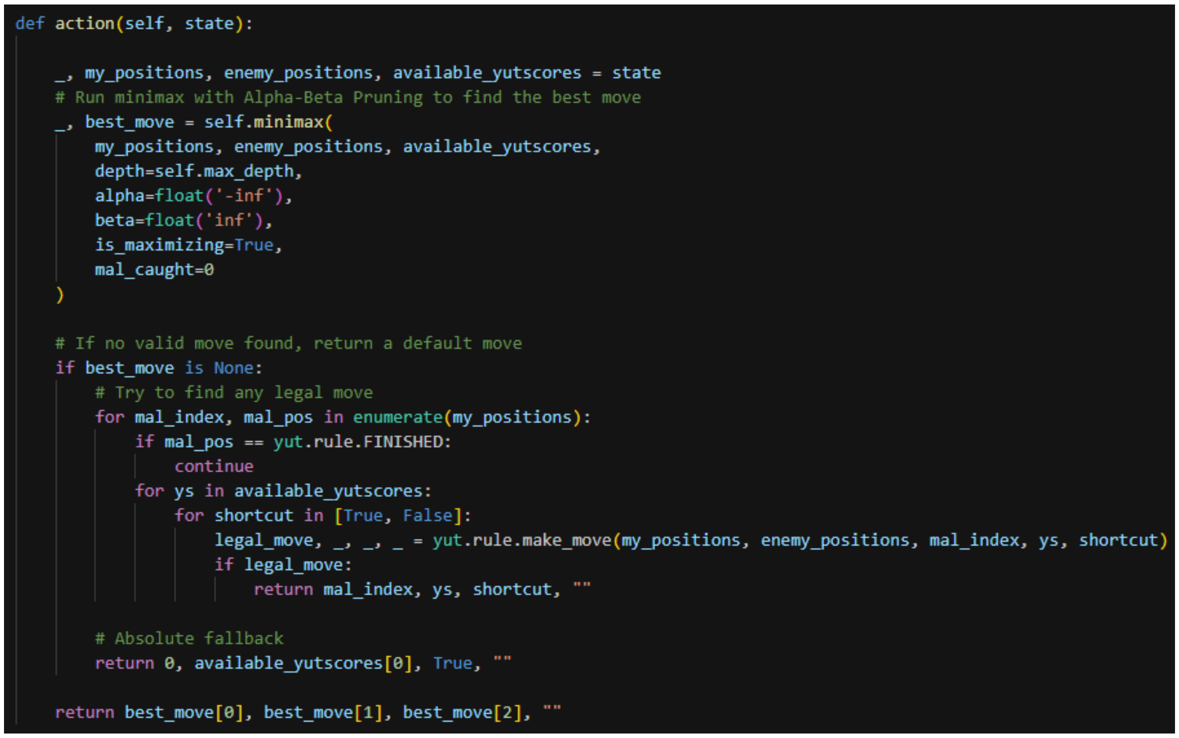
As we tried different values of the depth parameter of the minimax recursive function, it's observed that higher depth than 2 has a diminishing return. This can be because the more steps the program looks ahead, the further its imagination is from reality. Although higher depth in minimax is helpful in purely strategic and deterministic games like chess, it is not the case in games like Yut Nori where the next stage at a particular time depends on the yut that we receive. After all, the program can only approximate which yut it will get when looking ahead, which diverges from reality more, the further it looks ahead.



```
# Update best score and best move
if is_maximizing:
    if score > best_score:
        best_score = score
        best_move = (mal_index, ys, shortcut)

    # Alpha-Beta Pruning
    alpha = max(alpha, best_score)
    if alpha >= beta:
        return best_score, best_move  # Prune branches
else:
    if score < best_score:
        best_score = score
        best_move = (mal_index, ys, shortcut)

    # Alpha-Beta Pruning
    beta = min(beta, best_score)
    if beta <= alpha:
        return best_score, best_move  # Prune branches
```

3. `action()`

   The action function is straightforward with only state as an argument. The function initializes values for the minimax search and returns the best move, `yutscore_to_use`, `shortcut`, `debug_msg`. If the search does not yield any valid move, return default move

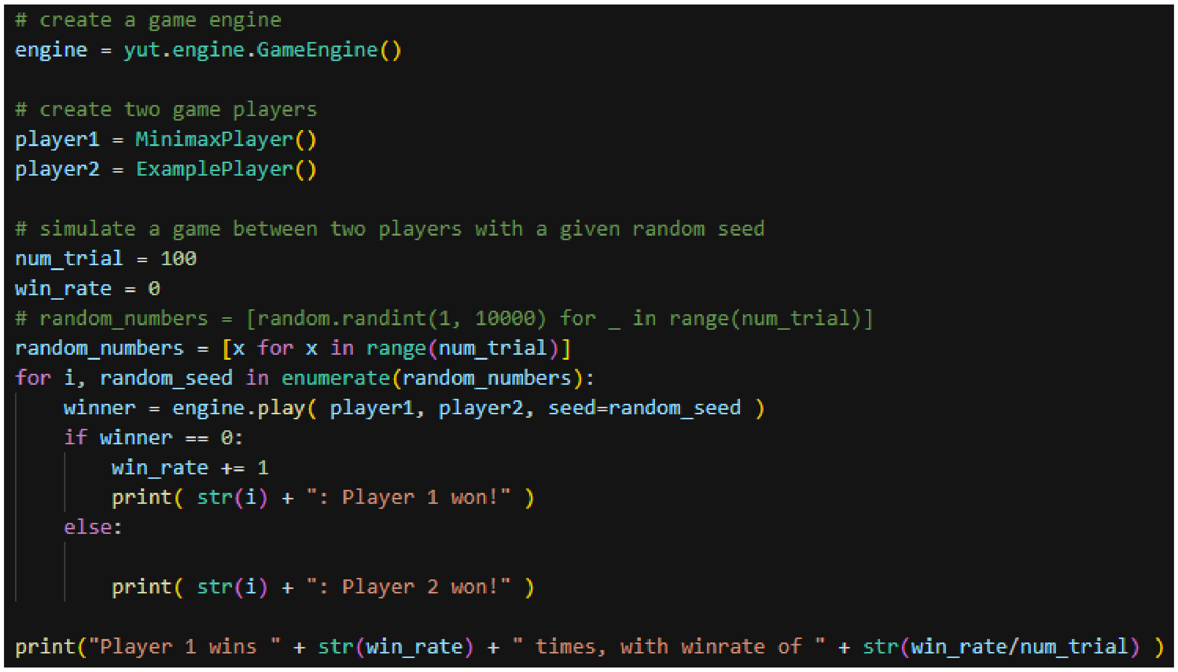


```
def action(self, state):

    _, my_positions, enemy_positions, available_yutscores = state
    # Run minimax with Alpha-Beta Pruning to find the best move
    _, best_move = self.minimax(
        my_positions, enemy_positions, available_yutscores,
        depth=self.max_depth,
        alpha=float('-inf'),
        beta=float('inf'),
        is_maximizing=True,
        mal_caught=0
    )

    # If no valid move found, return a default move
    if best_move is None:
        # Try to find any legal move
        for mal_index, mal_pos in enumerate(my_positions):
            if mal_pos == yut.rule.FINISHED:
                continue
            for ys in available_yutscores:
                for shortcut in [True, False]:
                    legal_move, _, _, _ = yut.rule.make_move(my_positions, enemy_positions, mal_index, ys, shortcut)
                    if legal_move:
                        return mal_index, ys, shortcut, ""

        # Absolute fallback
        return 0, available_yutscores[0], True, ""

    return best_move[0], best_move[1], best_move[2], ""
```

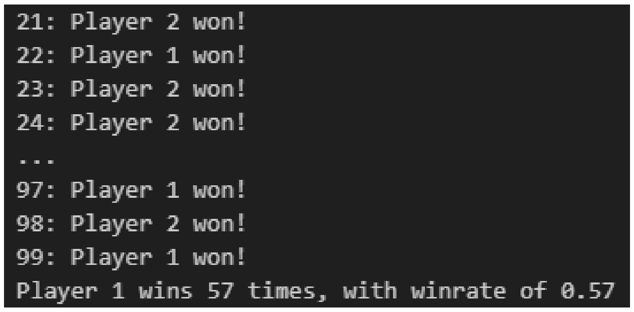
B. Results

Our bot consistently plays well against `RandomPlayer()` with around 89-94% win rate. This is slightly lower than the `ExamplePlayer()` performance due to it having more complex decision-making alongside the randomness involved in Yutnori. However, after analyzing the game history, the move our bot made is mostly optimal and `RandomPlayer()` only wins by mere luck.

Comparing our bot to `ExamplePlayer()`, we are slightly better with a win rate varying between 48-60% in most of our experiments. However, the result is still strongly dependent on the seed number (which is the random nature of this game). Regardless, after checking the game history, their decisions are near optimal.



```python
# create a game engine
engine = yut.engine.GameEngine()

# create two game players
player1 = MinimaxPlayer()
player2 = ExamplePlayer()

# simulate a game between two players with a given random seed
num_trial = 100
win_rate = 0
# random_numbers = [random.randint(1, 10000) for _ in range(num_trial)]
random_numbers = [x for x in range(num_trial)]
for i, random_seed in enumerate(random_numbers):
    winner = engine.play( player1, player2, seed=random_seed )
    if winner == 0:
        win_rate += 1
        print( str(i) + ": Player 1 won!" )
    else:

        print( str(i) + ": Player 2 won!" )

print("Player 1 wins " + str(win_rate) + " times, with winrate of " + str(win_rate/num_trial) )
```



```
21: Player 2 won!
22: Player 1 won!
23: Player 2 won!
24: Player 2 won!
...
97: Player 1 won!
98: Player 2 won!
99: Player 1 won!
Player 1 wins 57 times, with winrate of 0.57
```