Analysis of Online Learning Algorithms in Repeated Bimatrix Games

Northwestern University,
CS-396: Online Markets

1 Introduction

This analysis evaluates multiple learning algorithms to play in repeated bimatrix games. By performing numerous trials on different combinations of learning rates and learning algorithms on distinct bimatrix games, we can assess whether players will eventually converge to Nash equilibriums. This equilibrium is critical to uncover because it identifies actions in which all opposing players make the most optimal choices such that they have no incentive to deviate from their current strategy. Moreover, extracting learning algorithms with the least regret in bimatrix games can further help future players strategize in real-world settings.

Throughout our analysis, we observed that when learning rates of learning algorithms are similar, so are their payoffs, but algorithms with faster learning rates are capable of taking advantage of algorithms with slower learning rates, approaching vanishing regret and maximizing payoffs faster. We also developed an algorithm that takes advantage of certain online learning algorithms and their weakness in specific types of bimatrix games.

2 Preliminaries

This analysis evaluated the Exponential Weights (EW), Follow the Leader (FTL), and Exponential Weights Multi-Armed Bandit (MAB) algorithms as they competed against each other in bimatrix games and with different learning rates. The algorithms were matched up, their bimatrix payoffs were generated repeatedly, and their decisions were played out repeatedly in trials. Each match-up is generated such that neither algorithm knows the next move of their opponent before choosing their own move, and then both algorithms are given an opportunity to interpret the payoffs they received towards their future decisions.

Exponential Weights is the approach of using cumulative payoffs in hindsight to give each value an exponential weight, where the weights grow proportionate to the learning rate (determined by epsilon value) used with the algorithm. The exponential weights algorithm behaves as follows: given learning rate ϵ, the cumulative utility of action j in round i is $V_j^i = \sum_{r=1}^{i} v_j^r$. Using that, in round i choose action j with probability proportional to $(1+\epsilon)_J^{V_j^{i-1}/h}$.

Follow the Leader considers each action in hindsight, and always pick the action with the highest cumulative payoff so far. We attempted implementation of Follow the Regulated Leader, FTRL, which is a version of FTL that uses a regularizing function when calculating the payoff of each round before deciding cumulative payoffs, but our regularization function added time complexity while giving inferior results to EW, so it will not be elaborated on further in this analysis.

Multi-armed bandit implements an extension of the exponential weights algorithm such that it retrieves weights from online learning and applies an "explore vs exploit" adjustment to ensure there is proper exploration of actions. For each round i, the multi-armed bandit algorithm behaves as follows:

1. $\boldsymbol{\pi} \leftarrow$ OLA
2. draw $j^i \sim \tilde{\boldsymbol{\pi}}$ with

$$\tilde{\pi}^i_j = (1-\epsilon)\,\pi^i_j + {}^{\epsilon}/_{k}$$

3. take action j^i
4. report $\tilde{\mathbf{v}}$ to OLA with

$$\tilde{v}^i_j = \begin{cases} {}^{v^i_j}/_{\pi^i_j} & \text{if } j = j^i \\ 0 & \text{otherwise.} \end{cases}$$

The above graphic shows the steps of Mult-armed bandit when it is incorporated with an OLA algorithm (in this case exponential weights). In step 1, the online algorithm produces recommended weights. In step 2, the algorithm draws from a distribution of these recommended weights with an adjustment according to learning rate (epsilon) and number of possible actions (k). In step 3, an action j^i is drawn based on the adjusted weights. In step 4, the returned payoff is the payoff of that action divided by its adjusted weight, while every other action is given a 0 payoff. All these payoffs are then returned to the exponential weights algorithm and continues to the next round.

An algorithm's success is gauged by the regret it produces, where an algorithm's regret in round $n = Regret_n = \frac{1}{n}[OPT - ALG]$ and OPT is the cumulative payoff in hindsight of the best action, and ALG is the cumulative payoff in hindsight of the algorithm's choice.

We also generated a general heuristic that indicates whether or not an algorithm can consistently reach a Nash Equilibrium, called 'Nash deviation.' Nash deviation is calculated by finding the probability that each action was chosen over the last 10% of the rounds in a given match-up, and comparing that to the probability with which that action would have been chosen by the closest Nash Equilibrium.
$Nash\ deviation_{ALG} = \frac{\sum_{i=1}^{\#rounds}(equilibrium\ deviation_i)}{\#rounds}$, where $equilibrium\ deviation = min((|eq^1_1 - alg^1_1| + \ldots + |eq^1_j - alg^1_j|), \ldots, (|eq^m_1 - alg^m_1| + \ldots + |eq^m_j - alg^m_j|))$, where j is the number of actions and m is the number of Nash Equilibria.

The analysis generated bimatrix games using six methods. Each of the first four methods began by randomly selecting a payoff for each cell in the matrix from the uniform distribution over [0, 1]. The first generative method generated all games with a dominant strategy equilibrium. If a random payoff matrix was generated without a dominant strategy equilibrium, a random payoff matrix cell was selected to be the dominant strategy equilibrium, and each value sharing the column and row was randomly regenerated within a range that would result in a dominant strategy equilibrium for the chosen cell. The next generative method generated all games with at least 1 Pure Nash Equilibrium. If a payoff matrix was generated without a Pure Nash Equilibrium, a Pure Nash Equilibrium was added in the same way the dominant strategies were. The third generative method generated all bimatrix games with only mixed-strategy equilibria. For each randomly generated generated game, if it had any Pure Nash Equilibria, it was regenerated. We know all bimatrix games have at least one Nash Equilibria, so if there are no Pure Nash Equilibria, there must be at least one Mixed Nash Equilibrium. The fourth method randomly drew each payoff as the first 3, and could have any feasible number of dominant strategies, Pure Nash Equilibria, and Mixed Equilibria.

The fifth method generated games according to the 'prisoners' dilemma,' wherein both players had a dominant strategy of playing the same action, but that combination of dominant strategies resulted in the lowest average payoff for both parties. The payoff of the dominant equilibrium was randomly selected from a preset range, the payoff of the cell diagonally opposite the dominant equilibrium was generated from a preset range above that of the dominant cell, and the payoffs where each player chose different actions were 0 for the players that did not choose their dominant strategy action, and generated from a much higher range for those who did choose their dominant strategy action. The ranges were generated s.t. $max(low\ range) < min(med.\ range)$ and $max(med.\ range) < min(high\ range)$.

Method 5	Column Action 1	Column Action 2
Row Action 1	medium range, medium_range	0, high range
Row Action 2	high range, 0	low range, low range

The sixth method generated games slightly differently from the fifth method, specifically to create a situation where the column player does not have a dominant equilibrium, and when the row player and column player do *not* play according to the Nash Equilibrium, the row player gets a higher payoff than if they did play according to the Nash Equilibrium. This created a type of bimatrix game where the algorithm we devised for Part 2 to exploit Exponential Weights could manipulate the opponent into making choices that were more beneficial for itself. By generating this data set, we avoided situations where our exploitative algorithm would be best off when playing for the Nash Equilibrium or the opponent had a dominant strategy, making them not exploitable. Since our exploitative algorithm could be abstracted to perform this strategy when the bimatrix payoffs are exploitable but behave like the fixed learning algorithm when payoffs are not, if the exploitative algorithm has a higher payoff on this method of generation, it is capable of having a higher average payoff than the algorithm it is competing against on random generation.

The algorithm we produced to exploit this, 'Exploitative Exponential Weights' (EEW), first tests actions to populate its own payoff matrix by randomly guessing until its opposing EW algorithm has tried every action, and then considers the payoffs it can receive at each position in the matrix. Let action A for EEW and action B for EW be the combination that results in the highest payoff for EEW, and action C be the action EEW can take to increase EEW's likelyhood of playing B. Each turn thereafter, EEW proceeds to manipulate EW, taking action C, unless EW has taken action B for the last two turns. If EW has taken action B for the last two turns, EEW is moderately convinced that EW will continue to play B, and instead chooses to exploit, taking action A. makes that move twice in a row. At this point, the algorithm will swap from convincing to exploiting, and take the action that has the highest payoff as long as the EW algorithm has made.

Method 6	Column Action 1	Column Action 2
Row Action 1	medium range, highest_range	0, high range
Row Action 2	high range, 0	low range, low range

We hypothesize that algorithms with similar learning rate, whether EW or MAB, will converge to dominant strategy or Pure Nash equilibria easily, and converge to mixed equilibria less accurately but still successfully. However, when the difference in learning rates is drastic, we expect that some algorithms with faster learning rates will earn higher payoffs than algorithms with lower learning rates. We expect the same pattern to hold true for regret, that algorithms with similar learning rates will be more successful in producing vanishing regret.

3 Results

The entirety of the quantitative results calculated by running different trials in different game environments are provided in the appendix section, and figures are labeled there.

4 Conclusions

It should be noted that regret plots for dominant strategy equilibria, Pure Nash equilibria, and mixed Nash equilibria are not included in the results section, but can be generated by the attached code. They were omitted to avoid redundancy, because fully randomized, 'any Nash' bimatrix games resulted in the same patterns and implied the same conclusions as all the other specific Nash generation methods. Therefore, the first conclusion the analysis drew was that the type of Nash equilibria did not have a significant impact on learning algorithm matchups. The only impact regarding algorithm performance caused by different types of Nash equilibria was that algorithms found mixed Nash equilibria less consistently, demonstrated by the lower average *nash deviation* of algorithms on mixed equilibria than pure equilibria.

The result of our analysis confirmed our hypothesis that algorithms with similar learning rates produce similar payoffs, while faster learning rates can take advantage of algorithms with slower learning rates. It should be noted that in our analysis, we are equating a faster reduction of regret to a higher overall payoff. When both players have payoff matrices randomized over the same interval, in repeat trials their average payoffs are equivalent. Therefore, whichever algorithm minimizes regret faster or more effectively is generating a higher overall payoff on average. It should also be noted that in MAB, a low epsilon equates to a higher learning rate, while in EW, a high epsilon equates to a higher learning rate. In all cases where both played algorithms shared the same learning rate (Figure 2, Figure 3), their efficiency and effectiveness at minimizing regret were almost identical. In contrast, when one algorithm had a significantly higher learning rate than its opponent, the algorithm with the higher learning rate approached no regret faster (for example Figure 1).

The analysis of the prisoners' dilemma data generation revealed another important fact about EW, MAB, and FTL. These algorithms all managed to produce vanishing regret and approached their nash equilibrium (as indicated by their *nash deviation*, with MAB(0.5) vs MAB(0.5)'s *nash deviation* = 0.0994 and 0.0930 (lines 88, 89 results.txt from appendix), and EW(0.5) vs EW(0.5)'s *nash deviation* = 0.0100 and 0.0100 (lines 120, 121 results.txt from appendix) and their regret plots (Figure 5, Figure 6). However, in the prisoners' dilemma, both players playing Nash equilibrium generates the worst possible payoff for both of them. Therefore, minimizing regret does not equate to maximizing payoff over the bimatrix game. Minimizing regret only maximizes payoff if we assume that the opponent's decisions are entirely out of our control and are not influenced by our actions.

This motivated our approach to part 2, where we ran a custom exploitative algorithm, Exploit Exponential Weights (EEW) against EW and MAB at different learning rates on an adaptation of the prisoners' dilemma, specifically an adaptation where the column player (played by our learning algorithms) did not have a dominant strategy equilibrium, and thus could be 'exploited' by EEW to move the decisions away from the poor-payoff pure equilibrium.

After conducting trials, the algorithm we developed was successful in increasing payoffs at the cost of no longer generating vanishing regret. To exploit the EW algorithm, it would repeatedly make worse decisions from a regret perspective to manipulate EW into moving away from equilibrium and giving it the opportunity to occasionally make decisions with very high payoffs. In doing so, it moved EW away from the mutually detrimental Nash equilibrium and generated a higher average payoff for itself than that of the Nash equilibrium (Figure 8). This is demonstrated in the average payoff table for EEW vs a variety of online learning algorithms, located below Figure 8.

The final graph in Figure 8 demonstrates the manipulation in action, where the orange line (EEW)'s low moves manipulate the blue line (EW) to repeatedly make a move that gives a payoff spike for EEW. Once EEW is confident of the move EW will make, it takes advantage of the opportunity before going back to manipulating.

Overall, the results has led to the following motivated questions:

(1) Would learning algorithms with faster learning rates perform better outside of the 1v1 situation of a bimatrix game, or in environments with many actions?
(2) How could cases where EW can be exploited to the benefit of it's opponent be more rigorously defined to develop a more widely applicable exploitative algorithm?

5 Appendix

Resulting plot figures, table(s), and code for conducting relevant analysis and simulations were completed using the Jupyter Notebook platform and are attached at the end of the report. The last page includes the raw results of the Nash deviations for competing algorithms calculated in each trial.

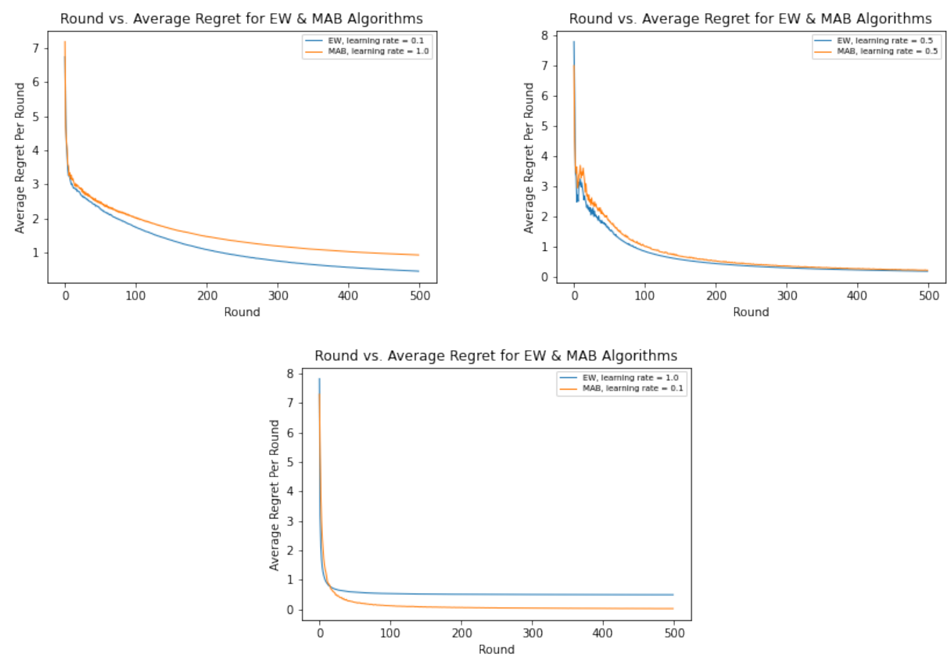

Figure 1: Trials for payoff matrices with any nash equilibria, Exponential Weights algorithm (EW) vs multi-armed bandit algorithm (MAB)

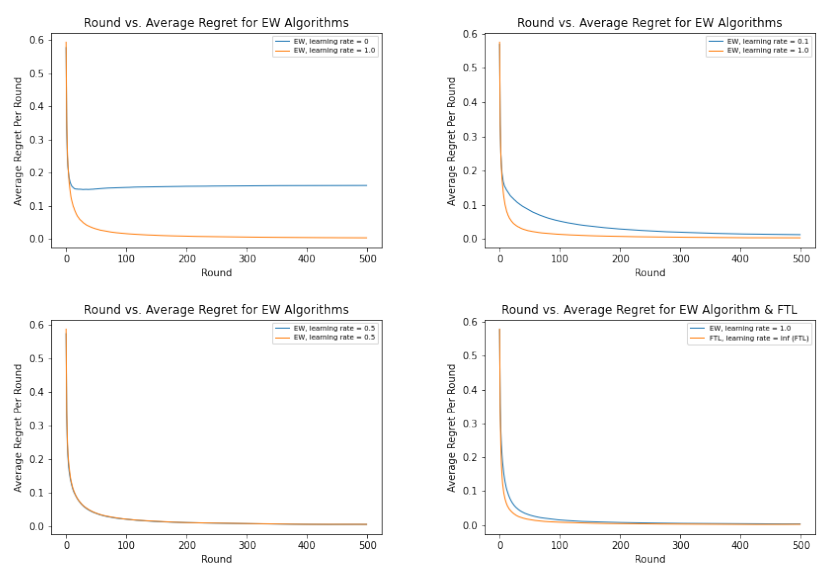

Figure 2: Any Nash Equilibrium. Each plot displays a different combination of learning rates, including 0, 0.1, 0.5, and 1.0. The bottom right plot includes a comparison of FTL algorithm and EW algorithm.

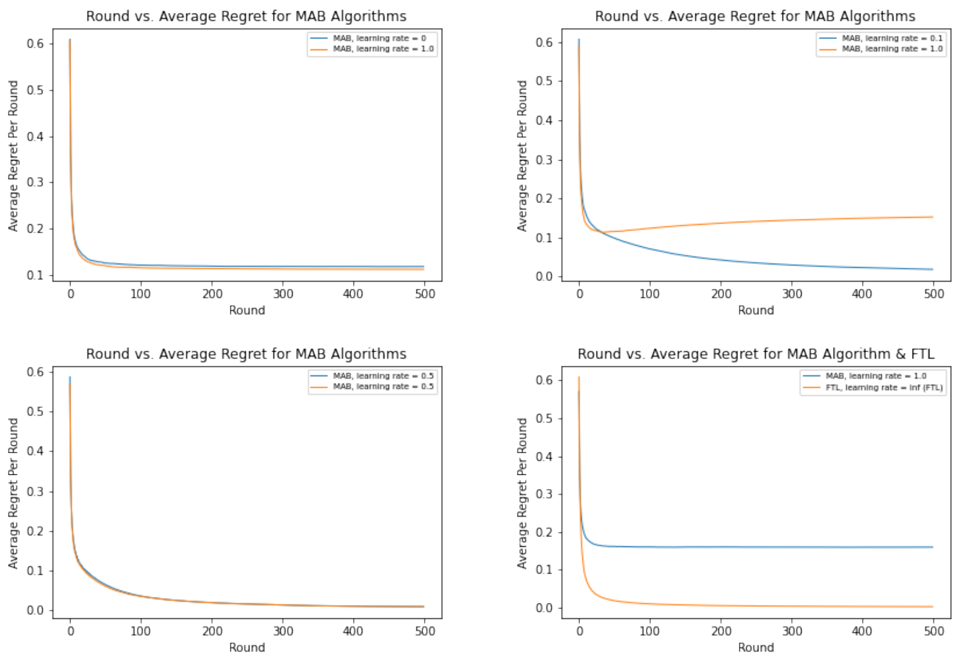

Figure 3: Any Nash Equilibrium. Each plot displays a different combination of learning rates, including 0, 0.1, 0.5, and 1.0. The bottom right plot includes a comparison of FTL algorithm and MAB algorithm.

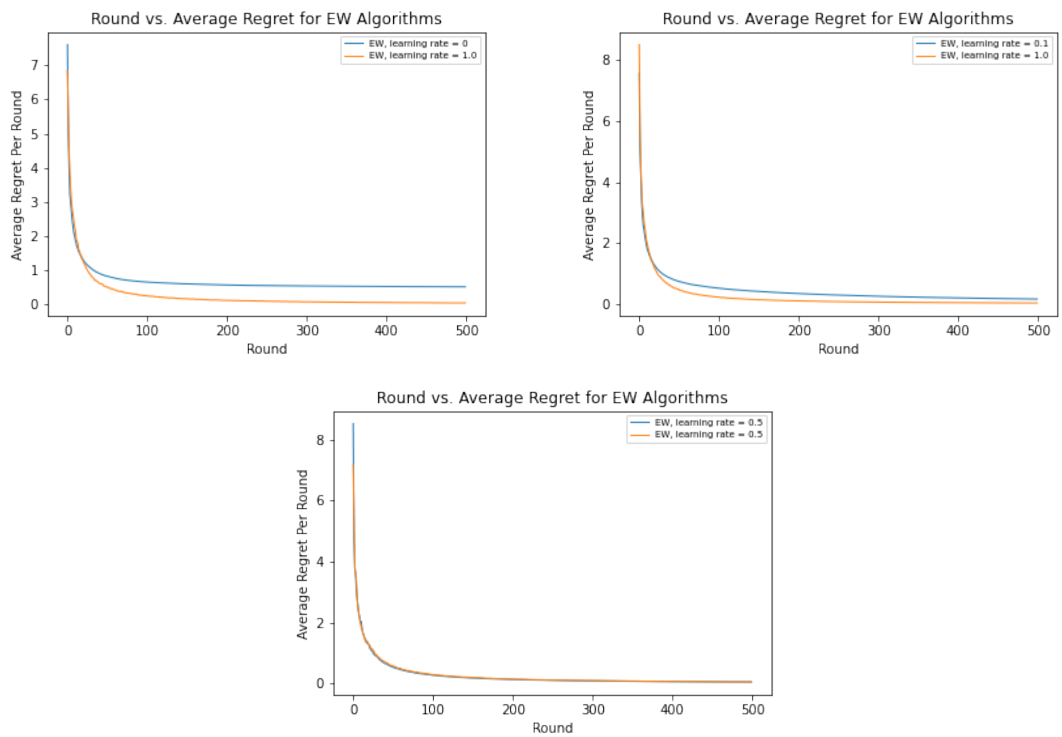

Figure 4: Any Nash Equilibrium. Each plot displays a different combination of learning rates, including 0, 0.1, 0.5, and 1.0.

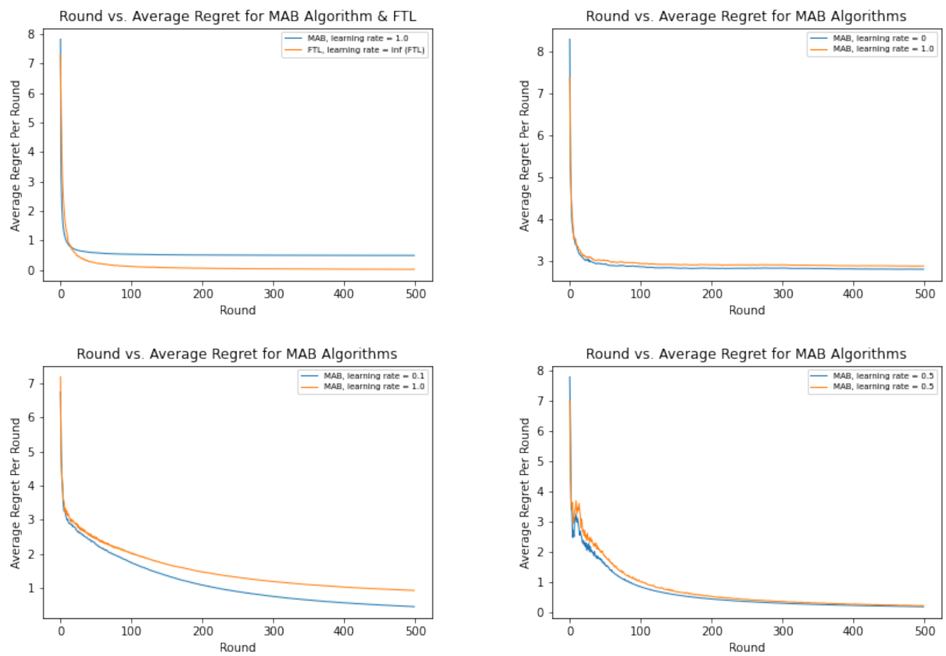

Figure 5: Prisoner Dilemma. Each plot displays a different combination of learning rates, including 0, 0.1, 0.5, and 1.0. The top left plot includes a comparison of FTL algorithm and MAB algorithm.

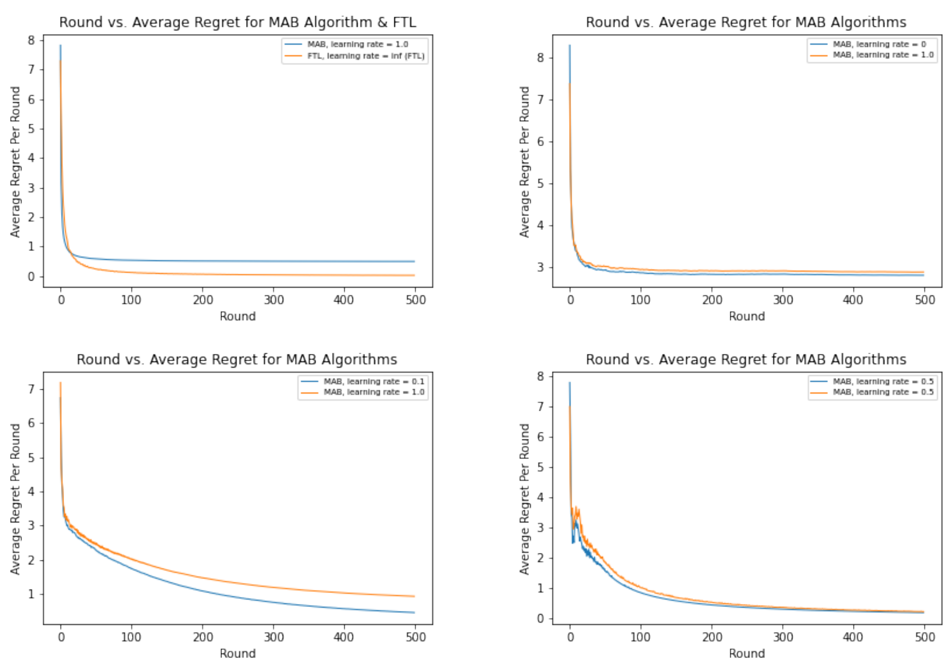

Figure 6: Prisoner Dilemma. Each plot displays a different combination of learning rates, including 0, 0.1, 0.5, and 1.0. The top left plot includes a comparison of FTL algorithm and MAB algorithm.

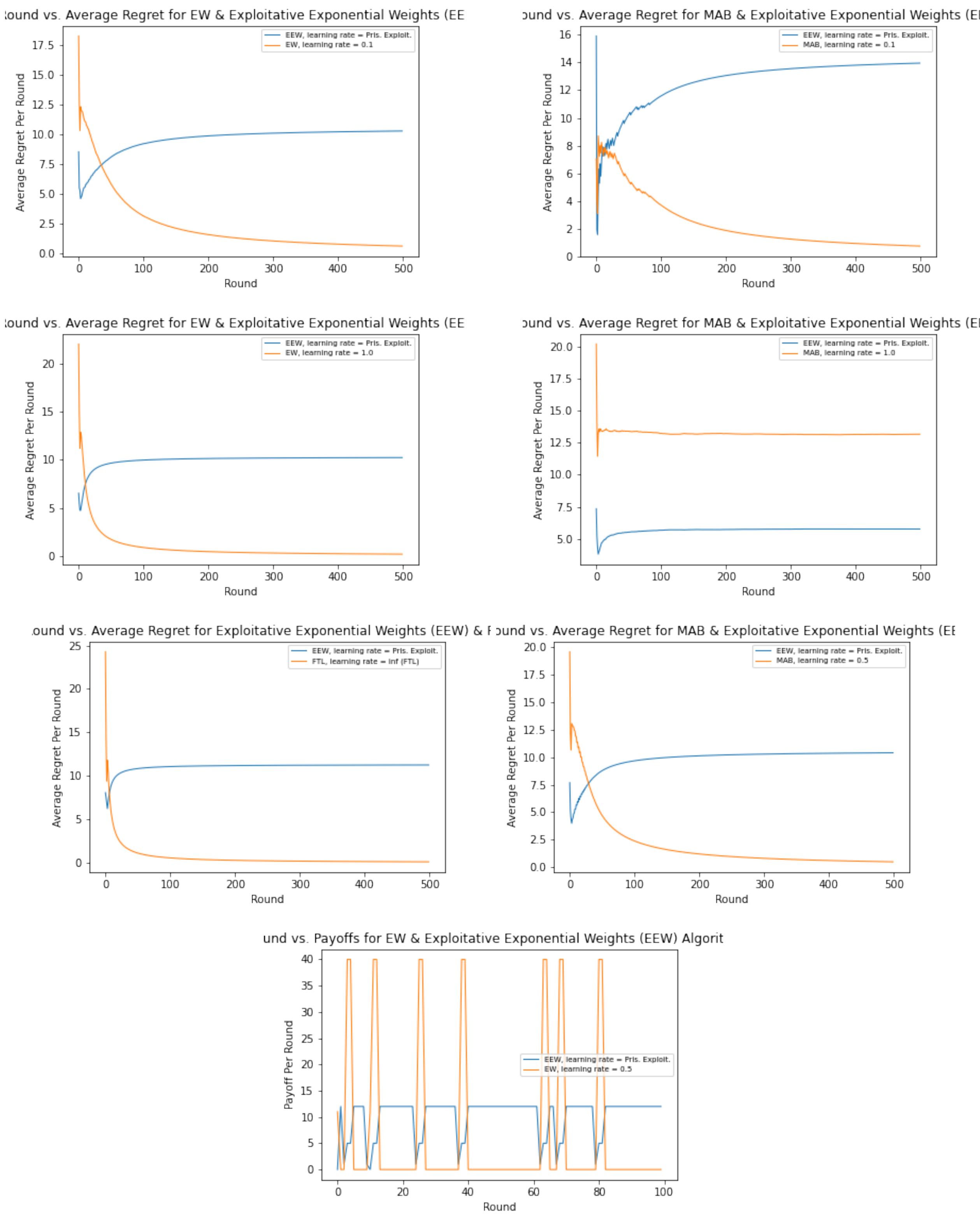

Figure 7: Prisoner Dilemma. Left-side plots represent a comparison between Exploitative Exponential Weights algorithm and Expected Weights algorithm, with the bottom left plot incorporating the Follow-The-Leader algorithm. The right-side plots represent a comparison between multi-armed bandit algorithm and Exploitative Exponential Weights algorithm. The bottom plot compares Exponential Weights algorithm with Exploitative Exponential Weights algorithm by measuring payoffs per round. Each plot displays a different combination of learning rates, including 0, 0.1, 0.5, and 1.0.

Table 1: Comparison of both player's respective payoffs from EEW, EW, and MAB algorithms

EEW Payoff Table	EEW Payoff	Opponent Payoff
EEW vs EW(epsilon=0.1)	EEW payoff: 6.2332	EW payoff: 0.82062
EEW vs EW(epsilon=0.5)	EEW payoff: 5.1886	EW payoff: 0.9592
EEW vs EW(epsilon=1.0)	EEW payoff: 4.9400	EW payoff: 1.0112
EEW vs FTL	EEW payoff: 6.2734	FTL payoff: 1.030
EEW vs MAB(epsilon=0.1)	EEW payoff: 6.9243	MAB payoff: 0.6858
EEW vs MAB(epsilon=0.5)	EEW payoff: 6.1193	MAB payoff: 0.8266
EEW vs MAB(epsilon=1.0)	EEW payoff: 7.7910	MAB payoff: 0.6023

Bimatrix games, different equilibria - Generate list of matrices (m1 = round 1) - Pure nash - Mixed nash - Prisoners' dilemma - RPS - Skip coarse-correlated equilibriums

FTL, OL, FTRL (regularized based on how recent the feedback was - *constant/i) A B X AX BX Y AY BY Online Learning - Given opponent took action X, we give alg AX, BX MAB - Given opponent took action X and we took action A, we give MAB just AX

In [ ]:

```
import matplotlib.pyplot as plt
import seaborn as sns
```

In [ ]:

```
import sys

import random
import nashpy as nash
import numpy as np

def rand_decimal():
    return random.randrange(0, 99)/100

def find_max_payoffs(payoff_matrix):
    max_row_payoff, max_col_payoff = 0, 0
    for row in payoff_matrix:
        for payoffs in row:
            row_payoff = payoffs[0]
            col_payoff = payoffs[1]
            if row_payoff > max_row_payoff: max_row_payoff = row_payoff
            if col_payoff > max_col_payoff: max_col_payoff = col_payoff
    return max_row_payoff, max_col_payoff

def generate_dominant_strategy(num_actions=2, num_rounds=1):
    row_dominant, col_dominant = random.randrange(0, num_actions), random.randrange(0, num_actions)
    #print(row_dominant, col_dominant)
    #generate randomized payoff matrix
    payoff_matrix = [[[rand_decimal(), rand_decimal()] for i in range(num_actions)] for i in range(num_actions)]

    #overwrite payoffs of dominant row and col with 'dominant' payoffs (random values that are higher than the max payoff)
    max_row_payoff, max_col_payoff = find_max_payoffs(payoff_matrix)
    for row in payoff_matrix:
        row[col_dominant][1] = random.randrange(int(max_col_payoff*100), 100)/100
    for payoff in payoff_matrix[row_dominant]:
        payoff[0] = random.randrange(int(max_row_payoff*100), 100)/100

    return payoff_matrix

def is_pure_nash(row, col, payoff_matrix, num_actions):
    row_player_val, col_player_val = payoff_matrix[row][col][0], payoff_matrix[row][col][1]
    for i in range(num_actions):
        if payoff_matrix[row][i][1] > col_player_val: return False
        if payoff_matrix[i][col][0] > row_player_val: return False
    return True

def add_pure_nash(payoff_matrix, num_actions):
    #print('pre-added')
    #print(payoff_matrix)
    pnash_row, pnash_col = random.randrange(0, num_actions), random.randrange(0, num_actions)
    old_row_val, old_col_val = payoff_matrix[pnash_row][pnash_col][0], payoff_matrix[pnash_row][pnash_col][1]
    row_max, col_max = 0, 0
    row_max_index, col_max_index = None, None
    for i in range(num_actions):
        if payoff_matrix[pnash_row][i][1] > col_max:
            col_max = payoff_matrix[pnash_row][i][1]
            col_max_index = i

        if payoff_matrix[i][pnash_col][0] > row_max:
            row_max = payoff_matrix[i][pnash_col][0]
            row_max_index = i

    col_max_loc = payoff_matrix[pnash_row][col_max_index]
    row_max_loc = payoff_matrix[row_max_index][pnash_col]
    col_max_loc[1], payoff_matrix[pnash_row][pnash_col][1] = old_col_val, col_max
    row_max_loc[0], payoff_matrix[pnash_row][pnash_col][0] = old_row_val, row_max
    #print('added')
    return [pnash_row, pnash_col]


def generate_pure_nash(num_actions=2, num_rounds=1):
    payoff_matrix = [[[rand_decimal(), rand_decimal()] for i in range(num_actions)] for i in range(num_actions)]
    pure_nash_list = []
    for row in range(num_actions):
        for col in range(num_actions):
            if is_pure_nash(row, col, payoff_matrix, num_actions): pure_nash_list.append([row, col])
    # if no pure nash randomly generated, recreate one
    if pure_nash_list == []:
        new_nash = add_pure_nash(payoff_matrix, num_actions)
        pure_nash_list.append(new_nash)

    #print(payoff_matrix)
    #print(pure_nash_list)
    return payoff_matrix

def generate_mixed_nash(num_actions=2, num_rounds=1):
    pure_nash_list = None
    while pure_nash_list != []:
        payoff_matrix = [[[rand_decimal(), rand_decimal()] for i in range(num_actions)] for i in range(num_actions)]
        pure_nash_list = []
        for row in range(num_actions):
            for col in range(num_actions):
                if is_pure_nash(row, col, payoff_matrix, num_actions): pure_nash_list.append([row, col])
    return payoff_matrix

def generate_any_nash(num_actions=2, num_rounds=1):
    #generate randomized payoff matrix, may have pure or mixed nash equilibrium(s)
    payoff_matrix = [[[rand_decimal(), rand_decimal()] for i in range(num_actions)] for i in range(num_actions)]
    return payoff_matrix

```

```python
def generate_prisoners():
    row_cooperate_payoff, col_cooperate_payoff = random.randrange(3, 6), random.randrange(3, 6)
    row_betray_payoff, col_betray_payoff = random.randrange(10, 20), random.randrange(10, 20)
    row_double_betray_payoff, col_double_betray_payoff = random.randrange(0, 3), random.randrange(0, 3)
    payoff_matrix = [
        [[row_cooperate_payoff, col_cooperate_payoff], [0, col_betray_payoff]],
        [[row_betray_payoff, 0], [row_double_betray_payoff, col_double_betray_payoff]]
    ]
    return payoff_matrix

def generate_rps():
    rock_win_payoff = random.randrange(10, 20)
    paper_win_payoff = random.randrange(10, 20)
    scissors_win_payoff = random.randrange(10, 20)
    tie_payoff = random.randrange(0, 3)
    rock_loss_payoff = random.randrange(5, 10)
    paper_loss_payoff = random.randrange(5, 10)
    scissors_loss_payoff = random.randrange(5, 10)
    payoff_matrix = [
        [[tie_payoff, tie_payoff], [rock_loss_payoff, paper_win_payoff], [rock_win_payoff, scissors_loss_payoff]],
        [[paper_win_payoff, rock_loss_payoff], [tie_payoff, tie_payoff], [paper_loss_payoff, scissors_win_payoff]],
        [[scissors_loss_payoff, rock_win_payoff], [scissors_win_payoff, paper_loss_payoff], [tie_payoff, tie_payoff]]
    ]

    return payoff_matrix

generate_any_nash()
generate_prisoners()
generate_rps()
```

In [ ]:

```python
## Multi-Armed Bandit Online Learning Algorithm
```

In [ ]:

```python
class MAB:

    def __init__(self, epsilon, num_actions=2):
        self.weights_vector = [[((1 / num_actions) * 100) for i in range(num_actions)]]
        self.totals_by_round = []
        self.partial_totals_by_round = []
        self.payoffs_by_round = []
        self.choices_by_round = []
        self.pi_tilda = []
        self.actions_list = [i for i in range(num_actions)]
        self.epsilon = epsilon
        self.num_actions = num_actions

    def reset_instance(self, epsilon=None, num_actions=2):
        self.weights_vector = [[((1 / num_actions) * 100) for i in range(num_actions)]]
        self.totals_by_round = []
        self.partial_totals_by_round = []
        self.payoffs_by_round = []
        self.choices_by_round = []
        self.pi_tilda = []
        self.actions_list = [i for i in range(num_actions)]
        self.num_actions = num_actions
        if epsilon == None:
            self.epsilon = self.epsilon
        else:
            epsilon = None

    def choose_action(self, max_payoff):
        # find weights
        current_weights = [None for i in range(self.num_actions)]
        for action in range(self.num_actions):
            if self.choices_by_round == []:
                #print(self.choices_by_round)
                current_weights = self.weights_vector[0]
            else:
                #print(self.weights_vector)
                #print(self.choices_by_round)
                total_weights = sum(self.weights_vector[-1])
                V_last = self.partial_totals_by_round[-1][action]
                exp = V_last / max_payoff
                current_weights[action] = (pow(1 + self.epsilon, exp) / total_weights) * 100
        #convert probabilties to new MAB distribution
        mab_weights = []
        for i in range(len(current_weights)):
            mab_weights.append(((1 -  self.epsilon) * (current_weights[i] / 100) +
                                (self.epsilon / self.num_actions)) * 100)

        # randomly select from actions using weights from MAB
        selected_action = random.choices(self.actions_list, weights=mab_weights, k=1)[0]
        self.pi_tilda.append(mab_weights[selected_action])
        self.weights_vector.append(current_weights)
        self.choices_by_round.append(selected_action)

        return selected_action

    def process_payoff(self, selected_payoff, payoff_list):
     # add new payoffs to totals, add payoff choice this round to payoffs matrix
        #self.payoffs_by_round.append(selected_payoff/self.pi_tilda[-1])
        self.payoffs_by_round.append(selected_payoff)

        if self.totals_by_round == []:
            temp_totals = []
            for i in range(self.num_actions):
                if i == self.choices_by_round[-1]:
                    temp_totals.append(selected_payoff/self.pi_tilda[-1])
                else:
                    temp_totals.append(0)
            self.partial_totals_by_round.append(temp_totals)
            self.totals_by_round.append([payoff_list[i] for i in range(self.num_actions)])
        else:
            last_round_totals = self.totals_by_round[-1]
            curr_payoffs = []
            for i in range(self.num_actions):
                if i == self.choices_by_round[-1]:
                    curr_payoffs.append(selected_payoff/self.pi_tilda[-1])
                else:
                    curr_payoffs.append(0)
            self.partial_totals_by_round.append([(last_round_totals[i] + curr_payoffs[i]) for i in
                                                 range(self.num_actions)])
            self.totals_by_round.append([last_round_totals[i] + payoff_list[i] for i in range(self.num_actions)])

        #print(self.totals_by_round)
        #print(self.payoffs_by_round)
        #NOTE: totals_by_round[-1] at the end of the simulation will help find 'OPT'
```

In [ ]:

```
class FTLRegularization:

    def __init__(self, num_actions=2):
        self.weights_vector = [1 for i in range(num_actions)]
        self.totals_by_round = []
        self.payoffs_by_round = []
        self.choices_by_round = []
        self.all_payoffs_by_round = []
        self.actions_list = [i for i in range(num_actions)]
        self.epsilon = 1000
        self.num_actions = num_actions

    def reset_instance(self, epsilon=None, num_actions=2):
        self.weights_vector = [1 for i in range(num_actions)]
        self.totals_by_round = []
        self.payoffs_by_round = []
        self.choices_by_round = []
        self.all_payoffs_by_round = []
        self.actions_list = [i for i in range(num_actions)]
        self.num_actions = num_actions
        if epsilon == None:
            self.epsilon = self.epsilon
        else:
            epsilon = None

    def find_ftlr_vector(self):
        vector = [0 for i in range(self.num_actions)]
        for index in range(len(self.all_payoffs_by_round)):
            for action in range(self.num_actions):
                #print(action, index, self.all_payoffs_by_round)
                vector[action] += self.all_payoffs_by_round[index][action] * (index / len(self.all_payoffs_by_round))
        return vector


    def choose_action(self, max_payoff):
        # find weights
        current_weights = [None for i in range(self.num_actions)]
        ftlr_vector = self.find_ftlr_vector()
        for action in range(self.num_actions):
            if self.totals_by_round == []:
                V_last = 0
            else:
                V_last = ftlr_vector[action]
            exp = V_last / max_payoff
            current_weights[action] = pow(1 + self.epsilon, exp)
        # randomly select from actions using weights as probabilities
        selected_action = random.choices(self.actions_list, weights=current_weights, k=1)[0]
        self.choices_by_round.append(selected_action)
        self.weights_vector.append(current_weights)
        return selected_action

    def process_payoff(self, selected_payoff, payoff_list):
        # add new payoffs to totals, add payoff choice this round to payoffs matrix
        self.payoffs_by_round.append(selected_payoff)
        self.all_payoffs_by_round.append(payoff_list)
        if self.totals_by_round == []:
            self.totals_by_round.append([payoff_list[i] for i in range(self.num_actions)])
        else:
            last_round_totals = self.totals_by_round[-1]
            self.totals_by_round.append([last_round_totals[i] + payoff_list[i] for i in range(self.num_actions)])

    #NOTE: totals_by_round[-1] at the end of the simulation will help find 'OPT'
```

Algorithm Classes

In [ ]:

```python
class ExponentialWeights:

    def __init__(self, epsilon, num_actions=2):
        self.weights_vector = [1 for i in range(num_actions)]
        self.totals_by_round = []
        self.payoffs_by_round = []
        self.choices_by_round = []
        self.actions_list = [i for i in range(num_actions)]
        self.epsilon = epsilon
        self.num_actions = num_actions

    def reset_instance(self, epsilon=None, num_actions=2):
        self.weights_vector = [1 for i in range(num_actions)]
        self.totals_by_round = []
        self.payoffs_by_round = []
        self.choices_by_round = []
        self.actions_list = [i for i in range(num_actions)]
        self.num_actions = num_actions
        if epsilon == None:
            self.epsilon = self.epsilon
        else:
            epsilon = None

    def choose_action(self, max_payoff):
        # find weights
        current_weights = [None for i in range(self.num_actions)]
        for action in range(self.num_actions):
            if self.totals_by_round == []:
                V_last = 0
            else:
                V_last = self.totals_by_round[-1][action]
            exp = V_last / max_payoff
            current_weights[action] = pow(1 + self.epsilon, exp)
        # randomly select from actions using weights as probabilities
        selected_action = random.choices(self.actions_list, weights=current_weights, k=1)[0]
        self.choices_by_round.append(selected_action)
        self.weights_vector.append(current_weights)
        return selected_action

    def process_payoff(self, selected_payoff, payoff_list):
        # add new payoffs to totals, add payoff choice this round to payoffs matrix
        self.payoffs_by_round.append(selected_payoff)
        if self.totals_by_round == []:
            self.totals_by_round.append([payoff_list[i] for i in range(self.num_actions)])
        else:
            last_round_totals = self.totals_by_round[-1]
            self.totals_by_round.append([last_round_totals[i] + payoff_list[i] for i in range(self.num_actions)])


    #NOTE: totals_by_round[-1] at the end of the simulation will help find 'OPT'
```

In [ ]:

```python
class FTL:

    def __init__(self, num_actions=2):
        self.totals_by_round = []
        self.payoffs_by_round = []
        self.choices_by_round = []
        self.actions_list = [i for i in range(num_actions)]
        self.num_actions = num_actions

    def reset_instance(self, num_actions=2):
        self.totals_by_round = []
        self.payoffs_by_round = []
        self.choices_by_round = []
        self.actions_list = [i for i in range(num_actions)]
        self.num_actions = num_actions

    def choose_action(self, max_payoff):
        # randomly select from actions using highest total payoff so far
        if self.totals_by_round != []:
            selected_action = self.totals_by_round[-1].index(max(self.totals_by_round[-1]))
            self.choices_by_round.append(selected_action)
            return selected_action
        else:
            selected_action = random.randrange(0, self.num_actions)
            return selected_action

    def process_payoff(self, selected_payoff, payoff_list):
        # add new payoffs to totals, add payoff choice this round to payoffs matrix
        self.payoffs_by_round.append(selected_payoff)
        if self.totals_by_round == []:
            self.totals_by_round.append([payoff_list[i] for i in range(self.num_actions)])
        else:
            last_round_totals = self.totals_by_round[-1]
            self.totals_by_round.append([last_round_totals[i] + payoff_list[i] for i in range(self.num_actions)])


    #NOTE: totals_by_round[-1] at the end of the simulation will help find 'OPT'
```

Matchup Simulator

In [ ]:

```
# helpers to find regret of an algorithm
def sum_to_round_i(alg_payoffs, current_round):
    total = 0
    for i in range(current_round):
        total += alg_payoffs[i]
    return total

def individual_regrets(alg_payoffs, round_totals):
    final_payoffs = round_totals[-1]
    opt_action = final_payoffs.index(max(final_payoffs))
    #print(opt_action)
    individual_regrets = [0 for i in range(len(alg_payoffs))]
    for round in range((len(alg_payoffs))):
        individual_regrets[round] = (round_totals[round][opt_action] - sum_to_round_i(alg_payoffs, round))
        / (round + 1)
    return individual_regrets

#takes two instantiations of algorithm classes as inputs
def matchup_simulator(alg1, alg2, payoff_matrix, num_rounds, max_payoff):
    num_actions = len(payoff_matrix)
    for round in range(num_rounds):
        # determine which action each algorithm picks
        alg1_action = alg1.choose_action(max_payoff)
        alg2_action = alg2.choose_action(max_payoff)

        # determine the payoffs and payoff lists for the algorithm combination
        payoff_cell = payoff_matrix[alg1_action][alg2_action]
        alg1_payoff, alg2_payoff = payoff_cell[0], payoff_cell[1]
        alg1_payoff_list, alg2_payoff_list = [], []
        for i in range(num_actions):
            alg1_payoff_list.append(payoff_matrix[i][alg2_action][0])
            alg2_payoff_list.append(payoff_matrix[alg1_action][i][1])

        # process the payoffs for the algorithm combination to prep alg1, alg2 for the next round
        alg1.process_payoff(alg1_payoff, alg1_payoff_list)
        alg2.process_payoff(alg2_payoff, alg2_payoff_list)
    #print(alg1.choices_by_round)
    #print(alg2.choices_by_round)
    # find the regret at each round, return the regret list for each algorithm
    alg1_regrets = individual_regrets(alg1.payoffs_by_round, alg1.totals_by_round)
    alg2_regrets = individual_regrets(alg2.payoffs_by_round, alg2.totals_by_round)
    #print(alg2.payoffs_by_round)
    #print(alg2.totals_by_round)
    return alg1_regrets, alg2_regrets

payoff_matrix = generate_dominant_strategy()
alg1 = MAB(0.5)
alg2 = MAB(0.1)
#alg2 = FTLRegularization()
#print(alg2.weights_vector)
#print(alg2.choose_action(1))
#alg2.choose_action(1)
matchup_simulator(alg1, alg2, payoff_matrix, 100, 1)
```

In [ ]:

```
## Delete contents of result file ###

# DO NOT RUN INDIVIDUALLY #

file = open("results.txt","r+")
file.truncate(0)
file.close()
```

Visualization of Regrets

In [ ]:

```
def visualize_regret(alg_results, rounds, lr_1, lr_2, plot_title, alg_1_name, alg_2_name, trial_type):

    file_name = trial_type + '_' + alg_1_name + alg_2_name + "_" + f'{lr_1}' + "_" + f'{lr_2}' + '.png'

    x = np.array(list(range(0, rounds)))
    y_1 = np.array(alg_results[0])
    y_2 = np.array(alg_results[1])
    plt.plot(x, y_1, label='{alg_1_name}, learning rate = {lr_1}'.format(alg_1_name=alg_1_name, lr_1 = lr_1),
             linewidth=1)
    plt.plot(x, y_2, label='{alg_2_name}, learning rate = {lr_2}'.format(alg_2_name=alg_2_name, lr_2 = lr_2),
             linewidth=1)
    plt.xlabel("Round")
    plt.ylabel("Average Regret Per Round")
    plt.title(plot_title)
    plt.legend(loc='best', prop={'size': 7})

    plt.savefig(file_name)

    plt.show()

    file1 = open("results.txt", "a")  # append mode
    file1.write(file_name + ", alg1" + ": " + f'{alg_results[2]}' + "\n")
    file1.write(file_name + ", alg2" + ": " f'{alg_results[3]}' + "\n")
    file1.close()
```

Matchup Trials

Matchup Trials

In [ ]:

```python
# matchup trial helpers
def update_avg_regrets(alg1_avg_regret_per_round, alg2_avg_regret_per_round, n, new_alg1_regrets, new_alg2_regrets):
    if alg1_avg_regret_per_round == None:
        alg1_avg_regret_per_round = new_alg1_regrets
    else:
        for i in range(len(alg1_avg_regret_per_round)):
            alg1_avg_regret_per_round[i] = ((n * alg1_avg_regret_per_round[i]) + new_alg1_regrets[i]) / (n + 1)

    if alg2_avg_regret_per_round == None:
        alg2_avg_regret_per_round = new_alg2_regrets
    else:
        for i in range(len(alg2_avg_regret_per_round)):
            alg2_avg_regret_per_round[i] = ((n * alg2_avg_regret_per_round[i]) + new_alg2_regrets[i]) / (n + 1)

def find_bimatrix_equilibria(payoff_matrix):
    row_player_payoffs = []
    col_player_payoffs = []
    for row in payoff_matrix:
        new_cplayer_row = []
        new_rplayer_row = []
        for payoff in row:
            new_cplayer_row.append(payoff[1])
            new_rplayer_row.append(payoff[0])
        row_player_payoffs.append(new_rplayer_row)
        col_player_payoffs.append(new_cplayer_row)

    A = np.array(row_player_payoffs)
    B = np.array(col_player_payoffs)
    game = nash.Game(A, B)
    equilibria = game.support_enumeration()
    return equilibria

# calculate what percent deviation alg1 and alg2 had from the closest nash equilibrium to their decisions
def dev_from_nash(alg1_last_choices, alg2_last_choices, payoff_matrix):
    num_actions = len(payoff_matrix)
    equilibria = find_bimatrix_equilibria(payoff_matrix)
    alg1_choice_averages = [0 for i in range(num_actions)]
    for action in range(num_actions):
        for choice in alg1_last_choices:
            if choice == action: alg1_choice_averages[action] += 1
    alg2_choice_averages = [0 for i in range(num_actions)]
    for action in range(num_actions):
        for choice in alg2_last_choices:
            if choice == action: alg2_choice_averages[action] += 1

    for index in range(len(alg1_choice_averages)):
        alg1_choice_averages[index] = alg1_choice_averages[index] / len(alg1_last_choices)
    for index in range(len(alg2_choice_averages)):
        alg2_choice_averages[index] = alg2_choice_averages[index] / len(alg2_last_choices)


    alg1_min_diff = float('inf')
    alg2_min_diff = float('inf')
    for eq in equilibria:
        alg1_eq, alg2_eq = eq[0], eq[1]
        alg1_curr_diff = abs(alg1_eq[0] - alg1_choice_averages[0]) + abs(alg1_eq[1] - alg1_choice_averages[1])
        alg2_curr_diff = abs(alg2_eq[0] - alg2_choice_averages[0]) + abs(alg2_eq[1] - alg2_choice_averages[1])
        if alg1_curr_diff < alg1_min_diff: alg1_min_diff = alg1_curr_diff
        if alg2_curr_diff < alg2_min_diff: alg2_min_diff = alg2_curr_diff

    return alg1_min_diff, alg2_min_diff


def matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds):
    alg1_avg_regret_per_round, alg2_avg_regret_per_round = None, None
    alg1_dev_from_nash_list, alg2_dev_from_nash_list = [], []

    for payoff_matrix in payoff_matrix_list:
        # find which trial number we are on
        n = payoff_matrix_list.index(payoff_matrix)

        #find max payoff (h)
        max_payoff = 0
        for row in payoff_matrix:
            for payoff in row:
                if payoff[0] > max_payoff: max_payoff = payoff[0]
                if payoff[1] > max_payoff: max_payoff = payoff[1]

        # run matchup and find regret lists
        new_alg1_regrets, new_alg2_regrets = matchup_simulator(alg1, alg2, payoff_matrix, num_rounds, max_payoff)

        # update average regret lists with new regret lists
        #update_avg_regrets(alg1_avg_regret_per_round, alg2_avg_regret_per_round, n, new_alg1_regrets, new_alg2_regrets)
        if alg1_avg_regret_per_round == None:
            alg1_avg_regret_per_round = new_alg1_regrets
        else:
            for i in range(len(alg1_avg_regret_per_round)):
                alg1_avg_regret_per_round[i] = ((n * alg1_avg_regret_per_round[i]) + new_alg1_regrets[i]) / (n + 1)

        if alg2_avg_regret_per_round == None:
            alg2_avg_regret_per_round = new_alg2_regrets
        else:
            for i in range(len(alg2_avg_regret_per_round)):
                alg2_avg_regret_per_round[i] = ((n * alg2_avg_regret_per_round[i]) + new_alg2_regrets[i]) / (n + 1)
        #TODO: take final stored nash values, check if they are nash equilibrium, update average deviation from nash
```

```
        alg1_last_actions = alg1.choices_by_round[-(int(num_rounds/10)):]
        alg2_last_actions = alg2.choices_by_round[-(int(num_rounds/10)):]
        alg1dev, alg2dev = dev_from_nash(alg1_last_actions, alg2_last_actions, payoff_matrix)
        alg1_dev_from_nash_list.append(alg1dev)
        alg2_dev_from_nash_list.append(alg2dev)

        # reset alg1 and alg2 internally stored values
        alg1.reset_instance()
        alg2.reset_instance()

    # calculate average deviation from nash equilibria
    alg1_avg_nash_dev = sum(alg1_dev_from_nash_list) / len(alg1_dev_from_nash_list)
    alg2_avg_nash_dev = sum(alg2_dev_from_nash_list) / len(alg2_dev_from_nash_list)

    return [alg1_avg_regret_per_round, alg2_avg_regret_per_round, alg1_avg_nash_dev, alg2_avg_nash_dev]


payoff_matrix_list = []
for i in range(1000):
    payoff_matrix_list.append(generate_dominant_strategy())
alg1 = ExponentialWeights(0.5)
alg2 = ExponentialWeights(1.0)
num_rounds = 500
matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

```

Run Trials on Payoff Matrix Types

In [ ]:

```
# Constants
NUM_TRIALS = 1000
NUM_ROUNDS = 500

#
# Trials for payoff matrices with RPS
#
payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_rps())
alg1 = ExponentialWeights(1.0)
alg2 = ExponentialWeights(1.0)
num_rounds = NUM_ROUNDS
rps_result_array = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

print(rps_result_array[0])
print(rps_result_array[1])
```

Dominant Strategy EW Trials

In [ ]:

```
#
# Trials for payoff matrices with dominant equilibria
#

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_dominant_strategy())
alg1 = ExponentialWeights(0.5)
alg2 = ExponentialWeights(0.5)
num_rounds = NUM_ROUNDS
ew_dominant_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(ew_dominant_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for EW Algorithms',
                 'EW', 'EW', 'DomStr')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_dominant_strategy())
alg1 = ExponentialWeights(0.1)
alg2 = ExponentialWeights(1.0)
num_rounds = NUM_ROUNDS
ew_dominant_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(ew_dominant_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for EW Algorithms',
                 'EW', 'EW', 'DomStr')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_dominant_strategy())
alg1 = ExponentialWeights(1.0)
alg2 = FTL()
num_rounds = NUM_ROUNDS
ew_dominant_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(ew_dominant_result_array3, num_rounds, 1.0, 'inf (FTL)',
                 'Round vs. Average Regret for EW Algorithm & FTL', 'EW', 'FTL', 'DomStr')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_dominant_strategy())
alg1 = ExponentialWeights(0)
alg2 = ExponentialWeights(1.0)
num_rounds = NUM_ROUNDS
ew_dominant_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(ew_dominant_result_array4, num_rounds, 0, 1.0,
                 'Round vs. Average Regret for EW Algorithms', 'EW', 'EW', 'DomStr')

#print(ew_dominant_result_array4[2])

#print(ew_dominant_result_array4[3])

#print(ew_dominant_result_array4[0])

#print(ew_dominant_result_array4[1])

#print(num_rounds)
```

Pure Nash EW Trials

In [ ]:

```python
#
# Trials for payoff matrices with Pure Nash equilibria
#

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_pure_nash())
alg1 = ExponentialWeights(0.5)
alg2 = ExponentialWeights(0.5)
num_rounds = NUM_ROUNDS
ew_pure_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(ew_pure_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for EW Algorithms',
                 'EW', 'EW', 'Pure Nash')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_pure_nash())
alg1 = ExponentialWeights(0.1)
alg2 = ExponentialWeights(1.0)
num_rounds = NUM_ROUNDS
ew_pure_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(ew_pure_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for EW Algorithms',
                 'EW', 'EW', 'Pure Nash')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_pure_nash())
alg1 = ExponentialWeights(1.0)
alg2 = FTL()
num_rounds = NUM_ROUNDS
ew_pure_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(ew_pure_result_array3, num_rounds, 0.1, 'inf (FTL)',
                 'Round vs. Average Regret for EW Algorithm & FTL', 'EW', 'FTL', 'Pure Nash')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_pure_nash())
alg1 = ExponentialWeights(0)
alg2 = ExponentialWeights(1.0)
num_rounds = NUM_ROUNDS
ew_pure_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(ew_pure_result_array4, num_rounds, 0, 1.0, 'Round vs. Average Regret for EW Algorithms',
                 'EW', 'EW', 'Pure Nash')
```

Mixed Nash EW Trials

In [ ]:

```
#
# Trials for payoff matrices with Mixed Nash equilibria
#

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_mixed_nash())
alg1 = ExponentialWeights(0.5)
alg2 = ExponentialWeights(0.5)
num_rounds = NUM_ROUNDS
mn_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mn_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for EW Algorithms',
                 'EW', 'EW', 'Mix Nash')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_mixed_nash())
alg1 = ExponentialWeights(0.1)
alg2 = ExponentialWeights(1.0)
num_rounds = NUM_ROUNDS
mn_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mn_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for EW Algorithms',
                 'EW', 'EW', 'Mix Nash')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_mixed_nash())
alg1 = ExponentialWeights(1.0)
alg2 = FTL()
num_rounds = NUM_ROUNDS
mn_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mn_result_array3, num_rounds, 1.0, 'inf (FTL)', 'Round vs. Average Regret for EW Algorithm & FTL',
                 'EW', 'FTL', 'Mix Nash')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_mixed_nash())
alg1 = ExponentialWeights(0)
alg2 = ExponentialWeights(1.0)
num_rounds = NUM_ROUNDS
mn_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mn_result_array4, num_rounds, 0, 1.0, 'Round vs. Average Regret for EW Algorithms',
                 'EW', 'EW', 'Mix Nash')

```

Any Nash EW Trials

In [ ]:

```python
#
# Trials for payoff matrices with Any Nash Equilibria
#

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_any_nash())
alg1 = ExponentialWeights(0.5)
alg2 = ExponentialWeights(0.5)
num_rounds = NUM_ROUNDS
an_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(an_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for EW Algorithms',
                 'EW', 'EW', 'Any Nash')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_any_nash())
alg1 = ExponentialWeights(0.1)
alg2 = ExponentialWeights(1.0)
num_rounds = NUM_ROUNDS
an_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(an_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for EW Algorithms',
                 'EW', 'EW', 'Any Nash')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_any_nash())
alg1 = ExponentialWeights(1.0)
alg2 = FTL()
num_rounds = NUM_ROUNDS
an_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(an_result_array3, num_rounds, 1.0, 'inf (FTL)', 'Round vs. Average Regret for EW Algorithm & FTL',
                 'EW', 'FTL', 'Any Nash')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_any_nash())
alg1 = ExponentialWeights(0)
alg2 = ExponentialWeights(1.0)
num_rounds = NUM_ROUNDS
an_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(an_result_array4, num_rounds, 0, 1.0, 'Round vs. Average Regret for EW Algorithms',
                 'EW', 'EW', 'Any Nash')

```

Prisoners' Dilemma EW Trials

In [ ]:

```
#
# Trials for payoff matrices with Prisoners' Dilemma
#

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_prisoners())
alg1 = ExponentialWeights(0.5)
alg2 = ExponentialWeights(0.5)
num_rounds = NUM_ROUNDS
p_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(p_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for EW Algorithms', 'EW',
                 'EW', 'Pr Dil')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_prisoners())
alg1 = ExponentialWeights(0.1)
alg2 = ExponentialWeights(1.0)
num_rounds = NUM_ROUNDS
p_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(p_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for EW Algorithms',
                 'EW', 'EW', 'Pr Dil')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_prisoners())
alg1 = ExponentialWeights(1.0)
alg2 = FTL()
num_rounds = NUM_ROUNDS
p_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(p_result_array3, num_rounds, 1.0, 'inf (FTL)', 'Round vs. Average Regret for EW Algorithm & FTL',
                 'EW', 'FTL', 'Pr Dil')
payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_prisoners())
alg1 = ExponentialWeights(0)
alg2 = ExponentialWeights(1.0)
num_rounds = NUM_ROUNDS
p_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(p_result_array4, num_rounds, 0, 1.0, 'Round vs. Average Regret for EW Algorithms',
                 'EW', 'EW', 'Pr Dil')
```

Dominant Strategy MAB Trials

In [ ]:

```
#
# Trials for payoff matrices with dominant equilibria
#

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_dominant_strategy())
alg1 = MAB(0.5)
alg2 = MAB(0.5)
num_rounds = NUM_ROUNDS
mab_dominant_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_dominant_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for MAB Algorithms',
                 'MAB', 'MAB', 'Dom Str')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_dominant_strategy())
alg1 = MAB(0.1)
alg2 = MAB(1.0)
num_rounds = NUM_ROUNDS
mab_dominant_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_dominant_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for MAB Algorithms',
                 'MAB', 'MAB', 'Dom Str')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_dominant_strategy())
alg1 = MAB(1.0)
alg2 = FTL()
num_rounds = NUM_ROUNDS
mab_dominant_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_dominant_result_array3, num_rounds, 1.0, 'inf (FTL)',
                 'Round vs. Average Regret for MAB Algorithm & FTL', 'MAB', 'FTL', 'Dom Str')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_dominant_strategy())
alg1 = MAB(0)
alg2 = MAB(1.0)
num_rounds = NUM_ROUNDS
mab_dominant_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_dominant_result_array4, num_rounds, 0, 1.0, 'Round vs. Average Regret for MAB Algorithms',
                 'MAB', 'MAB', 'Dom Str')
```

Pure Nash MAB Trials

In [ ]:

```
#
# Trials for payoff matrices with pure nash equilibria
#

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_pure_nash())
alg1 = MAB(0.5)
alg2 = MAB(0.5)
num_rounds = NUM_ROUNDS
mab_pn_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_pn_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for MAB Algorithms',
                 'MAB', 'MAB', 'Pure Nash')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_pure_nash())
alg1 = MAB(0.1)
alg2 = MAB(1.0)
num_rounds = NUM_ROUNDS
mab_pn_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_pn_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for MAB Algorithms',
                 'MAB', 'MAB', 'Pure Nash')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_pure_nash())
alg1 = MAB(1.0)
alg2 = FTL()
num_rounds = NUM_ROUNDS
mab_pn_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_pn_result_array3, num_rounds, 1.0, 'inf (FTL)',
                 'Round vs. Average Regret for MAB Algorithm & FTL', 'MAB', 'FTL', 'Pure Nash')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_pure_nash())
alg1 = MAB(0)
alg2 = MAB(1.0)
num_rounds = NUM_ROUNDS
mab_pn_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_pn_result_array4, num_rounds, 0, 1.0,
                 'Round vs. Average Regret for MAB Algorithms', 'MAB', 'MAB', 'Pure Nash')

```

Any Nash MAB Trials

In [ ]:

```
#
# Trials for payoff matrices with pure nash equilibria
#

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_any_nash())
alg1 = MAB(0.5)
alg2 = MAB(0.5)
num_rounds = NUM_ROUNDS
mab_an_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_an_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for MAB Algorithms',
                 'MAB', 'MAB', 'Any Nash')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_any_nash())
alg1 = MAB(0.1)
alg2 = MAB(1.0)
num_rounds = NUM_ROUNDS
mab_an_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_an_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for MAB Algorithms',
                 'MAB', 'MAB', 'Any Nash')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_any_nash())
alg1 = MAB(1.0)
alg2 = FTL()
num_rounds = NUM_ROUNDS
mab_an_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_an_result_array3, num_rounds, 1.0, 'inf (FTL)',
                 'Round vs. Average Regret for MAB Algorithm & FTL', 'MAB', 'FTL', 'Any Nash')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_any_nash())
alg1 = MAB(0)
alg2 = MAB(1.0)
num_rounds = NUM_ROUNDS
mab_an_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_an_result_array4, num_rounds, 0, 1.0, 'Round vs. Average Regret for MAB Algorithms',
                 'MAB', 'MAB', 'Any Nash')

```

Prisoners' Dilemma Trials

In [ ]:

```
#
# Trials for payoff matrices with pure nash equilibria
#

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_prisoners())
alg1 = MAB(0.5)
alg2 = MAB(0.5)
num_rounds = NUM_ROUNDS
mab_p_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_p_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for MAB Algorithms',
                 'MAB', 'MAB', 'Pris Dil')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_prisoners())
alg1 = MAB(0.1)
alg2 = MAB(1.0)
num_rounds = NUM_ROUNDS
mab_p_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_p_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for MAB Algorithms',
                 'MAB', 'MAB', 'Pris Dil')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_prisoners())
alg1 = MAB(1.0)
alg2 = FTL()
num_rounds = NUM_ROUNDS
mab_p_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_p_result_array3, num_rounds, 1.0, 'inf (FTL)',
                 'Round vs. Average Regret for MAB Algorithm & FTL', 'MAB', 'FTL', 'Pris Dil')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_prisoners())
alg1 = MAB(0)
alg2 = MAB(1.0)
num_rounds = NUM_ROUNDS
mab_p_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_p_result_array4, num_rounds, 0, 1.0,
                 'Round vs. Average Regret for MAB Algorithms', 'MAB', 'MAB', 'Pris Dil')

```

EW vs. MAB Trials

In [*]:

```
#
# Trials for payoff matrices with pure nash equilibria
#

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_any_nash())
alg1 = ExponentialWeights(0.5)
alg2 = MAB(0.5)
num_rounds = NUM_ROUNDS
ew_mab_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_p_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for EW & MAB Algorithms',
                 'EW', 'MAB', '')

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_any_nash())
alg1 = ExponentialWeights(0.1)
alg2 = MAB(1.0)
num_rounds = NUM_ROUNDS
ew_mab_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_p_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for EW & MAB Algorithms',
                 'EW', 'MAB', '')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_any_nash())
alg1 = ExponentialWeights(1.0)
alg2 = MAB(0.1)
num_rounds = NUM_ROUNDS
ew_mab_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_p_result_array3, num_rounds, 1.0, 0.1, 'Round vs. Average Regret for EW & MAB Algorithms',
                 'EW', 'MAB', '')

```

Part 2

In [ ]:
```python
def generate_asymmetric_prisoners():
    row_cooperate_payoff, col_cooperate_payoff = random.randrange(3, 6), random.randrange(3, 6)
    row_betray_payoff, col_betray_payoff = random.randrange(10, 20), random.randrange(10, 20)
    row_double_betray_payoff, col_double_betray_payoff = random.randrange(0, 3), random.randrange(0, 3)
    payoff_matrix = [
        [[row_cooperate_payoff, 10*col_cooperate_payoff], [0, col_betray_payoff]],
        [[row_betray_payoff, 0], [row_double_betray_payoff, col_double_betray_payoff]]
    ]
    return payoff_matrix
```

In [ ]:
```python
class EWPrisonersExploitation:
    def __init__(self, num_actions=2):
        self.totals_by_round = []
        self.payoffs_by_round = []
        self.choices_by_round = []
        self.actions_list = [i for i in range(num_actions)]
        self.payoff_matrix = [None for i in range(num_actions)]
        self.confess = None
        self.deny = None
        self.opponent_confess_vals = None
        self.opponent_deny_vals = None
        self.num_actions = num_actions

    def reset_instance(self, num_actions=2):
        self.totals_by_round = []
        self.payoffs_by_round = []
        self.choices_by_round = []
        self.actions_list = [i for i in range(num_actions)]
        self.payoff_matrix = [None for i in range(num_actions)]
        self.confess = None
        self.deny = None
        self.opponent_confess_vals = None
        self.opponent_deny_vals = None
        self.num_actions = num_actions

    def choose_action(self, max_payoff):

        # if within first 3 actions of game, or have not yet built our payoff matrix, guess randomly
        if len(self.payoffs_by_round) <= self.num_actions or None in self.payoff_matrix:
            selected_action = random.randrange(0, self.num_actions)
            self.choices_by_round.append(selected_action)
            return selected_action

        # If for the last 2 rounds the opponent confessed, deny
        if self.payoffs_by_round[-1] in self.opponent_confess_vals and self.payoffs_by_round[-2] in
        self.opponent_confess_vals:
            selected_action = self.deny
            self.choices_by_round.append(selected_action)
            return selected_action

        # otherwise, confess to bait opponent into higher probability of confessing
        selected_action = self.confess
        self.choices_by_round.append(selected_action)
        return selected_action


    def process_payoff(self, selected_payoff, payoff_list):
        # find selected action
        selected_action = payoff_list.index(selected_payoff)
        if selected_action not in self.payoff_matrix:
            self.payoff_matrix[selected_action] = payoff_list

        # if payoff matrix is full, find which action is confess, which action is deny
        if self.confess == None or self.deny == None:
            if payoff_matrix[0][0] > payoff_matrix[1][1]:
                self.confess = 0
                self.deny = 1
                self.opponent_confess_vals = [payoff_matrix[0][0][0], payoff_matrix[1][0][0]]
                self.opponent_deny_vals = [payoff_matrix[1][1][0], payoff_matrix[0][1][0]]
            else:
                self.confess = 1
                self.deny = 0
                self.opponent_confess_vals = [payoff_matrix[1][1][0], payoff_matrix[0][1][0]]
                self.opponent_deny_vals = [payoff_matrix[0][0][0], payoff_matrix[1][0][0]]

        # add new payoffs to totals, add payoff choice this round to payoffs matrix
        self.payoffs_by_round.append(selected_payoff)
        if self.totals_by_round == []:
            self.totals_by_round.append([payoff_list[i] for i in range(self.num_actions)])
        else:
            last_round_totals = self.totals_by_round[-1]
            self.totals_by_round.append([last_round_totals[i] + payoff_list[i] for i in range(self.num_actions)])


    #NOTE: totals_by_round[-1] at the end of the simulation will help find 'OPT'
```

Asymmetric Prisoners' Dilemma EW Exploitation Trials

In [ ]:

```
#
# Trials against EQ
#

payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_asymmetric_prisoners())
alg1 = EWPrisonersExploitation()
alg2 = ExponentialWeights(0.1)
num_rounds = NUM_ROUNDS
mab_p_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
print(mab_p_result_array1)

visualize_regret(mab_p_result_array1, num_rounds, 'Pris. Exploit.', 0.1,
                 'Round vs. Average Regret for EW & Exploitative Exponential Weights (EEW)', 'EEW', 'EW', 'Pris Dil')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_asymmetric_prisoners())
alg1 = EWPrisonersExploitation()
alg2 = ExponentialWeights(0.5)
num_rounds = NUM_ROUNDS
mab_p_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_p_result_array2, num_rounds, 'Pris. Exploit.', 0.5,
                 'Round vs. Average Regret for EW & Exploitative Exponential Weights (EEW)', 'EEW', 'EW', 'Pris Dil')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_asymmetric_prisoners())
alg1 = EWPrisonersExploitation()
alg2 = ExponentialWeights(1.0)
num_rounds = NUM_ROUNDS
mab_p_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_p_result_array3, num_rounds, 'Pris. Exploit.', 1.0,
                 'Round vs. Average Regret for EW & Exploitative Exponential Weights (EEW)', 'EEW', 'EW', 'Pris Dil')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_asymmetric_prisoners())
alg1 = EWPrisonersExploitation()
alg2 = FTL()
num_rounds = NUM_ROUNDS
mab_p_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_p_result_array4, num_rounds, 'Pris. Exploit.', 'inf (FTL)',
                 'Round vs. Average Regret for Exploitative Exponential Weights (EEW) & FTL', 'EEW', 'FTL', 'Pris Dil')


#
# Trials against MAB
#
payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_asymmetric_prisoners())
alg1 = EWPrisonersExploitation()
alg2 = MAB(0.1)
num_rounds = NUM_ROUNDS
mab_p_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
print(mab_p_result_array1)

visualize_regret(mab_p_result_array1, num_rounds, 'Pris. Exploit.', 0.1,
                 'Round vs. Average Regret for MAB & Exploitative Exponential Weights (EEW)', 'EEW', 'MAB', 'Pris Dil')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_asymmetric_prisoners())
alg1 = EWPrisonersExploitation()
alg2 = MAB(0.5)
num_rounds = NUM_ROUNDS
mab_p_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_p_result_array2, num_rounds, 'Pris. Exploit.', 0.5,
                 'Round vs. Average Regret for MAB & Exploitative Exponential Weights (EEW)', 'EEW', 'MAB', 'Pris Dil')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
    payoff_matrix_list.append(generate_asymmetric_prisoners())
alg1 = EWPrisonersExploitation()
alg2 = MAB(1.0)
num_rounds = NUM_ROUNDS
mab_p_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_p_result_array3, num_rounds, 'Pris. Exploit.', 1.0,
                 'Round vs. Average Regret for MAB & Exploitative Exponential Weights (EEW)', 'EEW', 'MAB', 'Pris Dil')


payoff_matrix_list = []
for i in range(NUM_TRIALS):
```

```
    payoff_matrix_list.append(generate_asymmetric_prisoners())
alg1 = EWPrisonersExploitation()
alg2 = FTL()
num_rounds = NUM_ROUNDS
mab_p_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)

visualize_regret(mab_p_result_array4, num_rounds, 'Pris. Exploit.', 'inf (FTL)',
                 'Round vs. Average Regret for Exploitative Exponential Weights (EEW) & FTL', 'EEW', 'FTL', 'Pris Dil')

```

Prisoner's Dilemma EW Exploit. Trial Payoff Visualization Function

In [ ]:

```
def visualize_payoff(payoff1, payoff2, rounds, lr_1, lr_2, plot_title, alg_1_name, alg_2_name, trial_type):

    file_name = trial_type + '_' + alg_1_name + alg_2_name + "_" + f'{lr_1}' + "_" + f'{lr_2}' + '.png'

    x = np.array(list(range(0, rounds)))
    y_1 = np.array(payoff1)
    y_2 = np.array(payoff2)
    plt.plot(x, y_1, label='{alg_1_name}, learning rate = {lr_1}'.format(alg_1_name=alg_1_name, lr_1 = lr_1),
             linewidth=1)
    plt.plot(x, y_2, label='{alg_2_name}, learning rate = {lr_2}'.format(alg_2_name=alg_2_name, lr_2 = lr_2),
             linewidth=1)
    plt.xlabel("Round")
    plt.ylabel("Payoff Per Round")
    plt.title(plot_title)
    plt.legend(loc='best', prop={'size': 7})

    plt.savefig(file_name)

    plt.show()
```

Prisoner's Dilemma EW Exploitation Sample Trial

In [ ]:

```
payoff_matrix = generate_asymmetric_prisoners()
alg1 = EWPrisonersExploitation()
alg2 = ExponentialWeights(0.5)
num_rounds = 100
max_payoff = 0
for row in payoff_matrix:
    for payoff in row:
        if payoff[0] > max_payoff: max_payoff = payoff[0]
        if payoff[1] > max_payoff: max_payoff = payoff[1]
regret1, regret2 = matchup_simulator(alg1, alg2, payoff_matrix, num_rounds, max_payoff)
payoffs1, payoffs2 = alg1.payoffs_by_round, alg2.payoffs_by_round
for row in payoff_matrix:
    print(row)

visualize_payoff(payoffs1, payoffs2, num_rounds, 'Pris. Exploit.', 0.5,
                 'Round vs. Payoffs for EW & Exploitative Exponential Weights (EEW) Algorithms', 'EEW', 'EW',
                 'Pris Dil')

#print(payoffs1)
#print(payoffs2)
```

DomStr_EWEW_0.5_0.5.png, alg1: 0.3
DomStr_EWEW_0.5_0.5.png, alg2: 0.3
DomStr_EWEW_0.1_1.0.png, alg1: 0.7
DomStr_EWEW_0.1_1.0.png, alg2: 0.1
DomStr_EWFTL_1.0_inf (FTL).png, alg1: 0.3
DomStr_EWFTL_1.0_inf (FTL).png, alg2: 0.0
DomStr_EWEW_0_1.0.png, alg1: 0.8
DomStr_EWEW_0_1.0.png, alg2: 0.3
Pure Nash_EWEW_0.5_0.5.png, alg1: 0.6
Pure Nash_EWEW_0.5_0.5.png, alg2: 0.5
Pure Nash_EWEW_0.1_1.0.png, alg1: 1.0
Pure Nash_EWEW_0.1_1.0.png, alg2: 0.4
Pure Nash_EWFTL_0.1_inf (FTL).png, alg1: 0.0
Pure Nash_EWFTL_0.1_inf (FTL).png, alg2: 0.2
Pure Nash_EWEW_0_1.0.png, alg1: 0.7
Pure Nash_EWEW_0_1.0.png, alg2: 0.5
Mix Nash_EWEW_0.5_0.5.png, alg1: 0.8741895309188061
Mix Nash_EWEW_0.5_0.5.png, alg2: 1.0116734166956458
Mix Nash_EWEW_0.1_1.0.png, alg1: 0.9870260954432399
Mix Nash_EWEW_0.1_1.0.png, alg2: 0.8918917063907437
Mix Nash_EWFTL_1.0_inf (FTL).png, alg1: 0.7666120144684581
Mix Nash_EWFTL_1.0_inf (FTL).png, alg2: 0.8212776067139398
Mix Nash_EWEW_0_1.0.png, alg1: 0.9416043589000577
Mix Nash_EWEW_0_1.0.png, alg2: 0.9495599774303429
Any Nash_EWEW_0.5_0.5.png, alg1: 0.6
Any Nash_EWEW_0.5_0.5.png, alg2: 0.5
Any Nash_EWEW_0.1_1.0.png, alg1: 0.9705208981436998
Any Nash_EWEW_0.1_1.0.png, alg2: 0.3884269298656388
Any Nash_EWFTL_1.0_inf (FTL).png, alg1: 0.545757605885862
Any Nash_EWFTL_1.0_inf (FTL).png, alg2: 0.14869797721460126
Any Nash_EWEW_0_1.0.png, alg1: 0.8954022988505747
Any Nash_EWEW_0_1.0.png, alg2: 0.3621621621621621
Pr Dil_EWEW_0.5_0.5.png, alg1: 0.5
Pr Dil_EWEW_0.5_0.5.png, alg2: 0.2
Pr Dil_EWEW_0.1_1.0.png, alg1: 0.3
Pr Dil_EWEW_0.1_1.0.png, alg2: 0.2
Pr Dil_EWFTL_1.0_inf (FTL).png, alg1: 0.1
Pr Dil_EWFTL_1.0_inf (FTL).png, alg2: 0.0
Pr Dil_EWEW_0_1.0.png, alg1: 0.5
Pr Dil_EWEW_0_1.0.png, alg2: 0.2
Dom Str_MABMAB_0.5_0.5.png, alg1: 1.0
Dom Str_MABMAB_0.5_0.5.png, alg2: 1.2
Dom Str_MABMAB_0.1_1.0.png, alg1: 1.1

Dom Str_MABMAB_0.1_1.0.png, alg2: 1.0
Dom Str_MABFTL_1.0_inf (FTL).png, alg1: 1.2
Dom Str_MABFTL_1.0_inf (FTL).png, alg2: 0.0
Dom Str_MABMAB_0_1.0.png, alg1: 0.8
Dom Str_MABMAB_0_1.0.png, alg2: 1.0
Pure Nash_MABMAB_0.5_0.5.png, alg1: 1.1
Pure Nash_MABMAB_0.5_0.5.png, alg2: 0.8
Pure Nash_MABMAB_0.1_1.0.png, alg1: 0.8
Pure Nash_MABMAB_0.1_1.0.png, alg2: 1.0
Pure Nash_MABFTL_1.0_inf (FTL).png, alg1: 1.1
Pure Nash_MABFTL_1.0_inf (FTL).png, alg2: 0.3
DomStr_EWEW_0.5_0.5.png, alg1: 0.016439999999999996
DomStr_EWEW_0.5_0.5.png, alg2: 0.019159999999999993
DomStr_EWEW_0.1_1.0.png, alg1: 0.07216
DomStr_EWEW_0.1_1.0.png, alg2: 0.004080000000000001
DomStr_EWFTL_1.0_inf (FTL).png, alg1: 0.028719999999999998
DomStr_EWFTL_1.0_inf (FTL).png, alg2: 0.0
DomStr_EWEW_0_1.0.png, alg1: 0.9905200000000008
DomStr_EWEW_0_1.0.png, alg2: 0.0003600000000000001
Pure Nash_EWEW_0.5_0.5.png, alg1: 0.016239999999999994
Pure Nash_EWEW_0.5_0.5.png, alg2: 0.012028571428571427
Pure Nash_EWEW_0.1_1.0.png, alg1: 0.05189246753246748
Pure Nash_EWEW_0.1_1.0.png, alg2: 0.03211999999999999
Pure Nash_EWFTL_0.1_inf (FTL).png, alg1: 0.0072
Pure Nash_EWFTL_0.1_inf (FTL).png, alg2: 0.002
Pure Nash_EWEW_0_1.0.png, alg1: 0.919597735187389
Pure Nash_EWEW_0_1.0.png, alg2: 0.2881733333333334
Mix Nash_EWEW_0.5_0.5.png, alg1: 0.4253902279882111
Mix Nash_EWEW_0.5_0.5.png, alg2: 0.42387568530937864
Mix Nash_EWEW_0.1_1.0.png, alg1: 0.28425298144208844
Mix Nash_EWEW_0.1_1.0.png, alg2: 0.5288270294542836
Mix Nash_EWFTL_1.0_inf (FTL).png, alg1: 0.450922521593311
Mix Nash_EWFTL_1.0_inf (FTL).png, alg2: 0.5129439983739663
Mix Nash_EWEW_0_1.0.png, alg1: 0.45645449601024285
Mix Nash_EWEW_0_1.0.png, alg2: 0.9898919006511173
Any Nash_EWEW_0.5_0.5.png, alg1: 0.06146016818534966
Any Nash_EWEW_0.5_0.5.png, alg2: 0.07241710822202245
Any Nash_EWEW_0.1_1.0.png, alg1: 0.08072620956993154
Any Nash_EWEW_0.1_1.0.png, alg2: 0.10268596252398027
Any Nash_EWFTL_1.0_inf (FTL).png, alg1: 0.06096150866184486
Any Nash_EWFTL_1.0_inf (FTL).png, alg2: 0.07078136260135606
Any Nash_EWEW_0_1.0.png, alg1: 0.8613203045235875
Any Nash_EWEW_0_1.0.png, alg2: 0.42985534119583135
Pr Dil_EWEW_0.5_0.5.png, alg1: 0.09972

Pr Dil_EWEW_0.5_0.5.png, alg2: 0.09788000000000005
Pr Dil_EWEW_0.1_1.0.png, alg1: 0.2972800000000005
Pr Dil_EWEW_0.1_1.0.png, alg2: 0.0
Pr Dil_EWFTL_1.0_inf (FTL).png, alg1: 0.24635999999999986
Pr Dil_EWFTL_1.0_inf (FTL).png, alg2: 0.0
Pr Dil_EWEW_0_1.0.png, alg1: 0.9552799999999992
Pr Dil_EWEW_0_1.0.png, alg2: 0.0
Dom Str_MABMAB_0.5_0.5.png, alg1: 0.004960000000000002
Dom Str_MABMAB_0.5_0.5.png, alg2: 0.007640000000000003
Dom Str_MABMAB_0.1_1.0.png, alg1: 0.007080000000000002
Dom Str_MABMAB_0.1_1.0.png, alg2: 1.0039199999999977
Dom Str_MABFTL_1.0_inf (FTL).png, alg1: 0.9960799999999994
Dom Str_MABFTL_1.0_inf (FTL).png, alg2: 0.0
Dom Str_MABMAB_0_1.0.png, alg1: 0.994
Dom Str_MABMAB_0_1.0.png, alg2: 0.9945200000000004
Pure Nash_MABMAB_0.5_0.5.png, alg1: 0.012839999999999997
Pure Nash_MABMAB_0.5_0.5.png, alg2: 0.01048
Pure Nash_MABMAB_0.1_1.0.png, alg1: 0.3137491677808849
Pure Nash_MABMAB_0.1_1.0.png, alg2: 0.9392729211738772
Pure Nash_MABFTL_1.0_inf (FTL).png, alg1: 0.9243866144791808
Pure Nash_MABFTL_1.0_inf (FTL).png, alg2: 0.26839999999999997
Pure Nash_MABMAB_0_1.0.png, alg1: 0.9330566596183434
Pure Nash_MABMAB_0_1.0.png, alg2: 0.9368665600980408
Any Nash_MABMAB_0.5_0.5.png, alg1: 0.06942592232914993
Any Nash_MABMAB_0.5_0.5.png, alg2: 0.06677141951461779
Any Nash_MABMAB_0.1_1.0.png, alg1: 0.36329707054974
Any Nash_MABMAB_0.1_1.0.png, alg2: 0.8584407290748491
Any Nash_MABFTL_1.0_inf (FTL).png, alg1: 0.879171135066243
Any Nash_MABFTL_1.0_inf (FTL).png, alg2: 0.3749868026355111
Any Nash_MABMAB_0_1.0.png, alg1: 0.860669154188545
Any Nash_MABMAB_0_1.0.png, alg2: 0.8570694350458758
Pris Dil_MABMAB_0.5_0.5.png, alg1: 0.09944000000000006
Pris Dil_MABMAB_0.5_0.5.png, alg2: 0.09300000000000003
Pris Dil_MABMAB_0.1_1.0.png, alg1: 0.0002800000000000002
Pris Dil_MABMAB_0.1_1.0.png, alg2: 0.9658799999999995
Pris Dil_MABFTL_1.0_inf (FTL).png, alg1: 0.9641599999999986
Pris Dil_MABFTL_1.0_inf (FTL).png, alg2: 0.0
Pris Dil_MABMAB_0_1.0.png, alg1: 0.9611599999999991
Pris Dil_MABMAB_0_1.0.png, alg2: 0.9661199999999984
_EWMAB_0.5_0.5.png, alg1: 0.09944000000000006
_EWMAB_0.5_0.5.png, alg2: 0.09300000000000003
_EWMAB_0.1_1.0.png, alg1: 0.0002800000000000002
_EWMAB_0.1_1.0.png, alg2: 0.9658799999999995
_EWMAB_1.0_0.1.png, alg1: 0.9641599999999986

_EWMAB_1.0_0.1.png, alg2: 0.0
Pris Dil_EEWEW_Pris. Exploit._0.1.png, alg1: 2.0
Pris Dil_EEWEW_Pris. Exploit._0.1.png, alg2: 1.33
Pris Dil_EEWEW_Pris. Exploit._0.5.png, alg1: 2.0
Pris Dil_EEWEW_Pris. Exploit._0.5.png, alg2: 1.308
Pris Dil_EEWEW_Pris. Exploit._1.0.png, alg1: 2.0
Pris Dil_EEWEW_Pris. Exploit._1.0.png, alg2: 1.36
Pris Dil_EEWFTL_Pris. Exploit._inf (FTL).png, alg1: 2.0
Pris Dil_EEWFTL_Pris. Exploit._inf (FTL).png, alg2: 1.316
Pris Dil_EEWMAB_Pris. Exploit._0.1.png, alg1: 2.0
Pris Dil_EEWMAB_Pris. Exploit._0.1.png, alg2: 1.35
Pris Dil_EEWMAB_Pris. Exploit._0.5.png, alg1: 2.0
Pris Dil_EEWMAB_Pris. Exploit._0.5.png, alg2: 1.294
Pris Dil_EEWMAB_Pris. Exploit._1.0.png, alg1: 2.0
Pris Dil_EEWMAB_Pris. Exploit._1.0.png, alg2: 0.963719999999999
Pris Dil_EEWFTL_Pris. Exploit._inf (FTL).png, alg1: 2.0
Pris Dil_EEWFTL_Pris. Exploit._inf (FTL).png, alg2: 1.35