Computational Intelligence Log Report

Ahmadreza Farmahini Farahani (s300909@studenti.polito.it)

January 26, 2024

Contents

1 Introduction

This is the log report for the Computational Intelligence exam on February 7, 2024. Throughout the course lectures, I diligently engaged in all the laboratories, challenges, and additional work, focusing on the concurrent development of each lab assignment. In the following sections, I will present a summary of the labs, followed by a discussion on the final project.

In the final project, which involved implementing the Quixo game, I applied Deep Reinforcement Learning and successfully achieved a 95% accuracy rate in winning against a random player after over 4000 iterations.

I extend my sincere gratitude to Professor Squillero and Andrea for their guidance and support throughout the course. It is worth noting that all the code development for the following assignments and the final project was completed independently by me. Throughout the course, I worked solo and did not engage in collaborative efforts with other students. Despite attempting to collaborate with peers, I found that their level of commitment did not align with my own.

2 Lab01

Three approaches have been used to address this problem:

2.1 Depth First

This algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it.

```
frontier = deque()
state = State(set(), set(range(NUM_SETS)))
frontier.append(state)

counter = 0
current_state = frontier.pop()

with tqdm(total=None) as pbar:
    while not goal_check(current_state):
        counter += 1
        for action in current_state[1]:
            new_state = State(
                current_state.taken ^ {action},
                current_state.not_taken ^ {action},
            )
            frontier.append(new_state)
        current_state = frontier.pop()


print(f"Depth first solved in {counter:,} steps ({len(current_state.taken)} tiles)")
```

The results are as following

```
Depth first solved in 13 steps (13 tiles)
```

2.2 Breadth First

It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

```
'''
Due to the high computational cost I was not able to calculate the Breadth First search.
'''
frontier = deque()
state = State(set(), set(range(NUM_SETS)))
frontier.append(state)

counter = 0
current_state = frontier.popleft()
with tqdm(total=None) as pbar:
    while not goal_check(current_state):
        counter += 1
        for action in current_state[1]:
            new_state = State(
                current_state.taken ^ {action},
                current_state.not_taken ^ {action},
            )
            frontier.append(new_state)
        current_state = frontier.popleft()
        pbar.update(1)

print(f"Breadth First solved in {counter:,} steps ({len(current_state.taken)} tiles)")
```

2.3 Greedy Best First

The algorithm works by using a heuristic function to determine which path is the most promising. The heuristic function takes into account the cost of the current path and the estimated cost of the remaining paths. If the cost of the current path is lower than the estimated cost of the remaining paths, then the current path is chosen. This process is repeated until the goal is reached.

```
def f(state):
    missing_size = PROBLEM_SIZE - sum(covered(state))
    return missing_size

frontier = PriorityQueue()
state = State(set(), set(range(NUM_SETS)))
frontier.put((f(state), state))

counter = 0
_, current_state = frontier.get()
with tqdm(total=None) as pbar:
    while not goal_check(current_state):
        counter += 1
        for action in current_state[1]:
            new_state = State(
                current_state.taken ^ {action},
                current_state.not_taken ^ {action},
            )
            frontier.put((f(new_state), new_state))
        _, current_state = frontier.get()
        pbar.update(1)

print(f"Greedy Best First solved in {counter:,} steps ({len(current_state.taken)} tiles)")
```

```
Greedy Best First solved in 5 steps (5 tiles)
```

2.4 Implementing A^*

I tried to create my own optimum A^* to minimize the tiles. In this function I minimize number of True labels existed in current covered indexes existed in not taken states. I also tried to minimize number of False labels in current not covered indexes exists in not taken states.

We have good results and in some cases number of tiles are lower than the first greedy best approach. It also outperforms the proposed cost function by professor.

```
def h_proposed(state):
    coverd_problems = covered(state)
    not_covered_indexes = np.where(np.logical_not(coverd_problems))[0]
    covered_indexes = np.where(coverd_problems)[0]
    not_taken_states = state.not_taken
    missing_size = PROBLEM_SIZE — sum(covered(state))

    tot_false_not_covered = 0
    tot_truth_covered = 0
    for ind_state in not_taken_states:
        tot_false_not_covered += sum(np.logical_not([covered(single_state(ind_state, NUM_SETS))[n]
                                                     for n in not_covered_indexes]))
        tot_truth_covered += sum([covered(single_state(ind_state, NUM_SETS))[n]
                                  for n in covered_indexes])

    return len(state.taken) + tot_truth_covered + tot_false_not_covered

frontier = PriorityQueue()
state = State(set(), set(range(NUM_SETS)))
frontier.put((h_proposed(state), state))

counter = 0
_, current_state = frontier.get()
with tqdm(total=None) as pbar:
    while not goal_check(current_state):
        counter += 1
        for action in current_state[1]:
            new_state = State(
                current_state.taken ^ {action},
                current_state.not_taken ^ {action},
            )
            frontier.put((h_proposed(new_state), new_state))
        _, current_state = frontier.get()
        pbar.update(1)

print(f"A_star objective by me solved in {counter:,} steps ({len(current_state.taken)} tiles)")
```

```
A_star objective solved in 5 steps (5 tiles)
```

2.5 Summary

Here is a quick summary of the results:

1. Depth First: 13 steps (13 tiles)
2. Greedy Best First: 5 steps (5 tiles)
3. Professor Method (admissible heuristic): 443 steps (5 tiles)
4. My Method: 5 steps (5 tiles)

At the first lab, I tried to review other students works and compared it to myself. Here are some results of other students:

1. Tilocca - s305938 : 443 steps (5 tiles)
2. Ferrigno - s316467 : 520 steps (16 tiles)

3 Lab02

In this Lab we needed to develop two agents, one using fixed rules and the other using an ES algorithm. Before I go to developing the agents I tried to develop a method for evaluation. After few run, we notice pure_random algorithm win the optimal solution. To address the issue we first analyze the wining_rate after 50 epochs to measure the performance our evaluation. We consider two procedures:

1. Optimal plays first
2. Pure Random plays first

```
algorithms = {
    'pure_random': pure_random,
    'gabriele': gabriele,
    'optimal': optimal
}


def play_nim(strategy, epochs: int) -> None:
    player_0 = strategy[0]
    player_1 = strategy[1]
    scores = {}
    scores[0] = {}
    scores[1] = {}
    scores[0][player_0] = 0
    scores[1][player_0] = 0
    scores[0][player_1] = 0
    scores[1][player_1] = 0
    strategy = (algorithms[player_0], algorithms[player_1])

    for _ in range(epochs):
        for strat in [0,1]:
            nim = Nim(5)
            player = strat
            while nim:
                ply = strategy[player](nim)
                nim.nimming(ply)
                player = 1 - player

            if player == 0:
                scores[strat][player_0]+=1
            else:
                scores[strat][player_1]+=1

    print(f'==== Comparing "{player_0}" and "{player_1}" algorithms ====\n')
    print(f'If player with "{player_0}" algorthim starts:')
    print(f'Win percentage for "{player_0}" player is:
                {round((scores[0][player_0]/epochs)*100, 2)}%')
    print(f'Win percentage for "{player_1}" player is:
                {round((scores[0][player_1]/epochs)*100, 2)}%\n')
    print(20*'----')
    print(f'\nIf player with "{player_1}" algorthim starts:')
```

```
    print(f'Win percentage for "{player_0}" player is:
                {round((scores[1][player_0]/epochs)*100, 2)}%')
    print(f'Win percentage for "{player_1}" player is:
                {round((scores[1][player_1]/epochs)*100, 2)}%')
```

We defined function *play_nim*() to evaluate future agents. Here are some results of evaluating existing algorithms:

==== Comparing "optimal" and "pure_random" algorithms ====

If player with "optimal" algorthim starts:
Win percentage for "optimal" player is: 76.0%
Win percentage for "pure_random" player is: 24.0%

If player with "pure_random" algorthim starts:
Win percentage for "optimal" player is: 64.0%
Win percentage for "pure_random" player is: 36.0%

==== Comparing "optimal" and "gabriele" algorithms ====

If player with "optimal" algorthim starts:
Win percentage for "optimal" player is: 86.0% Win percentage for "gabriele" player is: 14.0%

If player with "gabriele" algorthim starts:
Win percentage for "optimal" player is: 86.0%
Win percentage for "gabriele" player is: 14.0%

3.1 Rule Based Agent

In the first task we want to optimize an agent using fixed rules based on nim-sum (i.e., an expert system). I have managed to define three algorithms to address NIM game:

1. **min_optimal:** In this method, we select the first move which has the minimum nim_sum and it is not zero (if it exists).
2. **min_optimal_odd:** Like *min_ optimal*, except we consider the minimum odd move.
3. **min_optimal_even:** Like *min_ optimal*, except we consider the minimum even move.

At the end I will try to compare the results for each algorithm with previous algorithms.

```
'''
We now define a new algorithm.
'''



def min_optimal(state: Nim) -> Nimply:
    analysis = analize(state)
    logging.debug(f"analysis:\n{pformat(analysis)}")
    all_moves = [(ply, ns) for ply, ns in analysis["possible_moves"].items()]
    spicy_moves = sorted(all_moves, key=lambda x:x[1])
    lowest_sn_not_zero = 0
    for mov in spicy_moves:
        if mov[1] != 0:
            lowest_sn_not_zero = mov[1]
            break
    new_sm = [mov for mov in spicy_moves if mov[1]==lowest_sn_not_zero]
```

```
    return new_sm[-1][0]

algorithms['min_optimal'] = min_optimal
```

```
==== Comparing "min_optimal" and "pure_random" algorithms ====

If player with "min_optimal" algorthim starts:
Win percentage for "min_optimal" player is: 91.0%
Win percentage for "pure_random" player is: 9.0%

_____

If player with "pure_random" algorthim starts:
Win percentage for "min_optimal" player is: 93.0%
Win percentage for "pure_random" player is: 7.0%
```

With the 'min_optimal' method we always win against proposed 'optimal' method.

```
==== Comparing "min_optimal" and "optimal" algorithms ====
If player with "min_optimal" algorthim starts:
Win percentage for "min_optimal" player is: 100.0%
Win percentage for "optimal" player is: 0.0%

_____

If player with "optimal" algorthim starts:
Win percentage for "min_optimal" player is: 100.0%
Win percentage for "optimal" player is: 0.0%
```

```
def min_optimal_even(state: Nim) -> Nimply:
    analysis = analize(state)
    logging.debug(f"analysis:\n{pformat(analysis)}")
    all_moves = [(ply, ns) for ply, ns in analysis["possible_moves"].items()]
    spicy_moves = sorted(all_moves, key=lambda x:x[1])
    lowest_sn_not_zero = 0
    for mov in spicy_moves:
        if mov[1] != 0 and mov[1]%2 == 0:
            lowest_sn_not_zero = mov[1]
            break
    new_sm = [mov for mov in spicy_moves if mov[1]==lowest_sn_not_zero]
    if len(new_sm)==0:
        return spicy_moves[0][0]
    return new_sm[-1][0]

algorithms['min_optimal_even'] = min_optimal_even
```

The results of the min_optimal_even are as follows:

==== Comparing "min_optimal_even" and "pure_random" algorithms ====

If player with "min_optimal_even" algorthim starts:
Win percentage for "min_optimal_even" player is: 48.0%
Win percentage for "pure_random" player is: 52.0%

If player with "pure_random" algorthim starts:
Win percentage for "min_optimal_even" player is: 58.0%
Win percentage for "pure_random" player is: 42.0%

We obtain very bad results. However, we conclude that selecting even nim_sums lower the performance. I have also defined one function for the odd version.

```
def min_optimal_odd(state: Nim) -> Nimply:
    analysis = analize(state)
    logging.debug(f"analysis:\n{pformat(analysis)}")
    all_moves = [(ply, ns) for ply, ns in analysis["possible_moves"].items()]
    spicy_moves = sorted(all_moves, key=lambda x:x[1])
    lowest_sn_not_zero = 0
    for mov in spicy_moves:
        if mov[1] != 0 and mov[1]%2 == 1:
            lowest_sn_not_zero = mov[1]
            break
    new_sm = [mov for mov in spicy_moves if mov[1]==lowest_sn_not_zero]
    if len(new_sm)==0:
        return spicy_moves[0][0]
    return new_sm[-1][0]

algorithms['min_optimal_odd'] = min_optimal_odd
```

The results of the min_optimal_even are as follows:

==== Comparing "min_optimal_odd" and "pure_random" algorithms ====

If player with "min_optimal_odd" algorthim starts:
Win percentage for "min_optimal_odd" player is: 96.0%
Win percentage for "pure_random" player is: 4.0%

If player with "pure_random" algorthim starts:
Win percentage for "min_optimal_odd" player is: 98.0%
Win percentage for "pure_random" player is: 2.0%

We obtain best results. We managed to win 97% of the times over 200 epoch on average. With the min_optimal_odd method we always win against proposed 'optimal' method.

3.2 Evolved Agent

In this task I will tried analyze three methods provided by Prof. in lectures. The methods are used:

$$Population\ size\ \ (vanilla) : (1 + \lambda)$$
$$Population\ size\ \ (adaptive) : (1 + \lambda)$$

$$Population\ size\ \ (adaptive) : (\mu + \lambda)$$

Now we start analysis by creating and evoluted agent using Evolutionary Algorithms. I want to develop algorithms using below definitions:

$$f(w) = \frac{1}{2}\left[\left|\sum_{n=1}^{E} w(w)\right|_{s=0} + \left|\sum_{n=1}^{E} w(w)\right|_{s=1}\right]$$

$$g(a) = (row,\ num\_objects,\ future\_nim\_sum)$$

$$s(a, w) = \sum_{j=0}^{D} g(a) \bigotimes w_{ij}$$

In above formulations, $f(w)$ stands for the total fitness of the agent and $w(w)$ represent the winning status of the agent for a specific weight. The total objective will be calculated over E number of epochs. In each epoch agent choose an action based on it's weights associated to it. Each action consist of three chromosomes. The $g(a)$ describe the chromosomes features of an action, where action $a \in R^A$and$A$is number of actions available for the agent. The scores of each step will be calculated by $s(a, w)$. Action with the highest score will be chosen. After the game is over, $w(w)$ will be calculated (0 or 1) for each game. We repeat the process with two possible starts s. The final score is averaged over number of wins. Best weight in population is selected for adding to the new population λ. Moreover, D denotes as the dimension of the weight.

3.3 Vanila-ES $(1 + \lambda)$

First the define following functions:

```
def genome(state: Nim) -> Nimply:
    analysis = analize(state)
    logging.debug(f"analysis:\n{pformat(analysis)}")
    moves = np.array([[ply.row, ply.num_objects, ns]
                      for ply, ns in analysis["possible_moves"].items()])
    return moves

def weights_initalization(weights_dim: int) -> tuple:
    weights = np.random.rand(3,weights_dim) * 5.12 * 2 - 5.12
    projection_weights = np.random.rand(weights_dim,1)
    return (weights, projection_weights)

def create_population(mu, sigma, weights_dim: int):
    population = np.random.rand(mu,3,weights_dim+1)
    population[:,:, :-1] = population[:,:, :-1] * 5.12 * 2 - 5.12
    population[:,:, -1] *= sigma
    return population

def z_norm(outputs: np.array):
    mean = np.mean(output, axis=0).shape
    std = np.std(output, axis=0).shape
    return (output - mean) / std

def es(state, weights):
    moves = genome(state)
    output = moves @ weights
    return Nimply(int(moves[np.argmin(output.sum(axis=1))][0]),
                  int(moves[np.argmin(output.sum(axis=1))][1]))
```

```
def objective(weights_list, opponent_strategy, epochs, nim_dim):
    scores_first = 0
    scores_second = 0
    all_scores = []
    opponent_strategy = algorithms[opponent_strategy]

    for weight in weights_list:
        scores_first = 0
        scores_second = 0
        for _ in range(epochs):
            for strat in [1,0]:
                nim = Nim(nim_dim)
                player = strat
                while nim:
                    if player == 1:
                        ply = opponent_strategy(nim)
                    else:
                        ply = es(nim, weights)
                    nim.nimming(ply)
                    player = 1 - player

                if player == 0:
                    if strat == 0:
                        scores_first+=1

                    else:
                        scores_second+=1
        all_scores.append([(scores_first + scores_second)/2])


    return np.array(all_scores)
```

After we defined our methods, we try to train the models.

```
weights_dim = 5
genome_size = 3
nim_dim = 4
epochs = 20
lambda = 20
sigma = 0.6
opponent_strategy = 'optimal'


weights, _ = weights_initalization(weights_dim)
history = list()
best = np.copy(weights)
current_highest = objective(np.expand_dims(weights, axis=0),
                        opponent_strategy, epochs, nim_dim)[0][0]
with trange(1_000 // lambda) as t:
    for n in t:
        # offspring <- select lambda random points mutating the current solution
        weights_list = (
            np.random.normal(loc=0, scale=sigma, size=(lambda, genome_size, weights_dim))
            + weights
        )
        # evaluarte and select best
        evals = objective(weights_list, opponent_strategy, epochs, nim_dim)
```

```
        weights = weights_list[np.argmax(evals)]
        new_highest = np.max(evals)
        if current_highest < new_highest:
            best = np.copy(weights)
            current_highest = new_highest
            history.append((n, new_highest))

        t.set_postfix({f"Best Average Wins": f"{current_highest}/{epochs}"})

logging.info(f"Best solution: {objective(np.expand_dims(best, axis=0),
                opponent_strategy, epochs, nim_dim)[0][0]}")

if len(history) > 0:
    history = np.array(history)
    plt.figure(figsize=(14, 4))
    plt.plot(history[:, 0], history[:, 1], marker=".")
```

Now let's take a look at the progress of the scores. In the fig. 2 we have have run the model for 50 iteration. Due to high computational cost I was not able to fully train the model.

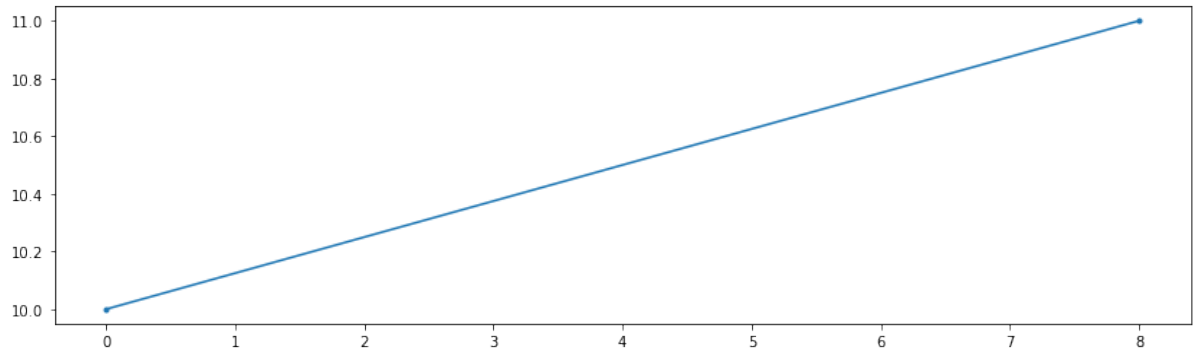


Figure 1: Vanila-ES $(1 + \lambda)$

3.4 Adaptive-ES $(1 + \lambda)$

Evolution Strategies (ES) is a family of optimization algorithms that draw inspiration from natural evolution to find optimal solutions to problems. The terms μ and λ in Adaptive-ES refer to the sizes of two distinct populations used in the optimization process.

In the following listing we have the weights for the Adaptive-ES. Moreover, we define a matrix of dimension three for assigning initial values. These weights will be updated for each evaluaition iteration.

```
weights_dim = 3
genome_size = 3
nim_dim = 4
epochs = 30
lambda = 30
sigma = 0.15
opponent_strategy = 'optimal'


weights, _ = weights_initalization(weights_dim)
history = list()
best = np.copy(weights)
current_highest = objective(np.expand_dims(weights, axis=0),
                            opponent_strategy, epochs, nim_dim)[0][0]
```

```
stats = [0, 0]
with trange(1500 // lambda) as t:
    for step in t:
        weights_list = (
            np.random.normal(loc=0, scale=sigma, size=(lambda, genome_size, weights_dim))
            + weights
        )
        evals = objective(weights_list, opponent_strategy, epochs, nim_dim)
        stats[0] += lambda
        stats[1] += sum(evals > objective(np.expand_dims(weights, axis=0),
                        opponent_strategy, epochs, nim_dim)[0][0])
        weights = weights_list[np.argmax(evals)]
        new_highest = np.max(evals)

        if current_highest < new_highest:
            best = np.copy(weights)
            current_highest = new_highest
            history.append((step, new_highest))

        if (step + 1) % 5 == 0:
            if stats[0] / stats[1] < 1 / 5:
                sigma /= 1.2
            elif stats[0] / stats[1] > 1 / 5:
                sigma *= 1.2
            steps = [0, 0]
        t.set_postfix({f"Best Average Wins": f"{current_highest}/{epochs}"})


logging.info(f"Best solution: {objective(np.expand_dims(best, axis=0),
                                opponent_strategy, epochs, nim_dim)[0][0]}")

if len(history) > 0:
    history = np.array(history)
    plt.figure(figsize=(14, 4))
    plt.plot(history[:, 0], history[:, 1], marker=".")
```

Now let's take a look at the progress of the scores. In the fig. 2 we have have run the model for 50 iteration. Due to high computational cost I was not able to fully train the model.

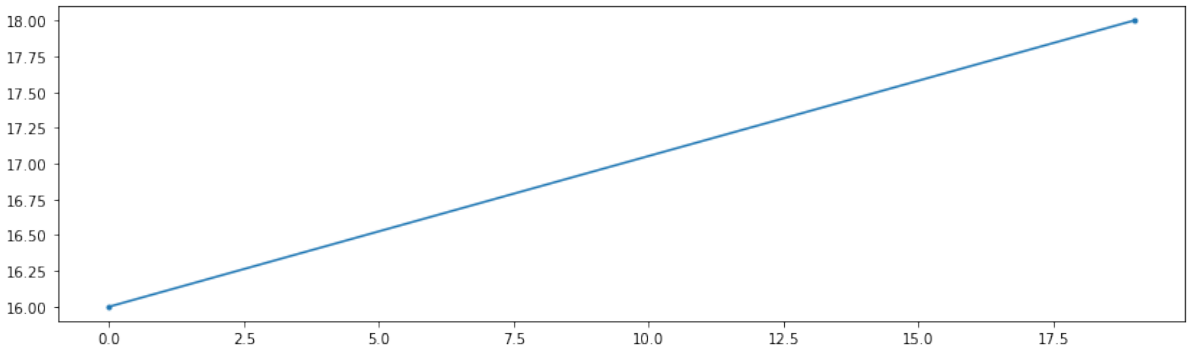


Figure 2: Adaptive-ES(1+λ)

3.5 Adaptive-ES ($\mu + \lambda$)

As we can see, we have also added μ factor to our calculations.

```
mu = 20
lambda = 30
sigma = 1
weights_dim = 3
genome_size = 3
nim_dim = 4
epochs = 30
opponent_strategy = 'optimal'



population = create_population(mu, sigma, weights_dim)

best_fitness = None
history = list()
with trange(1500 // lambda) as t:
    for step in t:
        # offspring <- select lambda random points from the population of mu
        offspring = population[np.random.randint(0, mu, size=(lambda,))]
        # mutate all sigma (last column) and replace negative values with a small number
        offspring[:,:, -1] = np.random.normal(
            loc=offspring[:,:, -1], scale=0.2
        )
        offspring[offspring[:,:, -1] < 1e-5, -1] = 1e-5
        # mutate all v (all columns but the last), using the sigma in the last column
        offspring[:,:, 0:-1] = np.random.normal(
            loc=offspring[:,:, 0:-1], scale=np.expand_dims(offspring[:,:, -1], axis=2)
        )
        # add an extra column with the evaluation and sort
        fitness = objective(offspring[:,:, 0:-1], opponent_strategy, epochs, nim_dim)
        offspring = offspring[fitness.argsort().reshape(-1)]
        # save best (just for the plot)
        if best_fitness is None or best_fitness < np.max(fitness):
            best_fitness = np.max(fitness)
            history.append((step, best_fitness))
        # select the mu with max fitness and discard fitness
        population = np.copy(offspring[-mu:])
        t.set_postfix({f"Best Average Wins": f"{best_fitness}/{epochs}"})



fitness = objective(population[:, 0:-1], opponent_strategy, epochs, nim_dim)
logging.info(
    f"Best solution: {fitness.max()} (with sigma={population[fitness.argmax(), -1]})"
)

if len(history) > 0:
    history = np.array(history)
    plt.figure(figsize=(14, 4))
    plt.plot(history[:, 0], history[:, 1], marker=".")
```

Due to the high computational cost I was not able to train the agent on a proper model. However, as it can be seen in fig. 3 we have progress in overall scores.

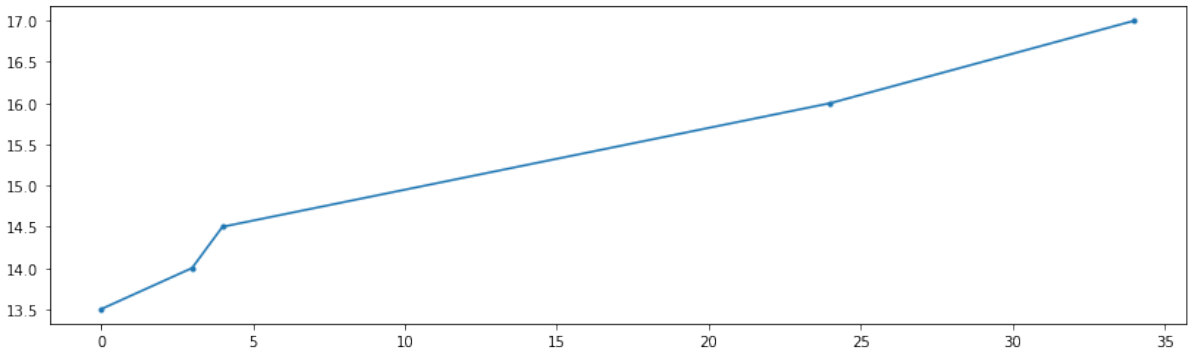
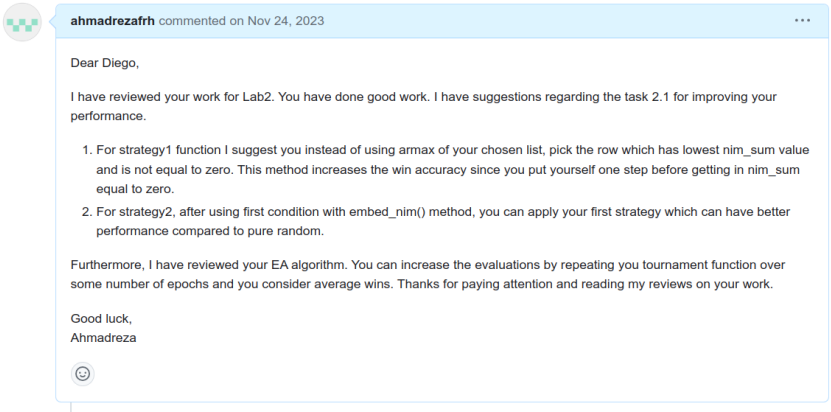


Figure 3: Adaptive-ES(μ+λ)

3.6 Reviews

3.6.1 Opened

I have opened two review for two of my colleagues.



ahmadrezafrh commented on Nov 24, 2023

Dear Diego,

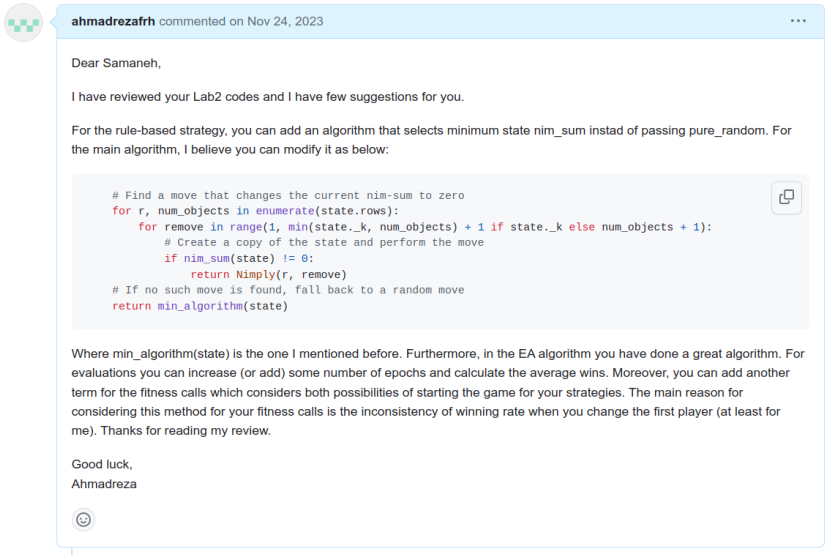
I have reviewed your work for Lab2. You have done good work. I have suggestions regarding the task 2.1 for improving your performance.

1. For strategy1 function I suggest you instead of using armax of your chosen list, pick the row which has lowest nim_sum value and is not equal to zero. This method increases the win accuracy since you put yourself one step before getting in nim_sum equal to zero.
2. For strategy2, after using first condition with embed_nim() method, you can apply your first strategy which can have better performance compared to pure random.

Furthermore, I have reviewed your EA algorithm. You can increase the evaluations by repeating you tournament function over some number of epochs and you consider average wins. Thanks for paying attention and reading my reviews on your work.

Good luck,
Ahmadreza

Figure 4: Lab02 first review



ahmadrezafrh commented on Nov 24, 2023

Dear Samaneh,

I have reviewed your Lab2 codes and I have few suggestions for you.

For the rule-based strategy, you can add an algorithm that selects minimum state nim_sum instad of passing pure_random. For the main algorithm, I believe you can modify it as below:

```
# Find a move that changes the current nim-sum to zero
for r, num_objects in enumerate(state.rows):
    for remove in range(1, min(state._k, num_objects) + 1 if state._k else num_objects + 1):
        # Create a copy of the state and perform the move
        if nim_sum(state) != 0:
            return Nimply(r, remove)
# If no such move is found, fall back to a random move
return min_algorithm(state)
```

Where min_algorithm(state) is the one I mentioned before. Furthermore, in the EA algorithm you have done a great algorithm. For evaluations you can increase (or add) some number of epochs and calculate the average wins. Moreover, you can add another term for the fitness calls which considers both possibilities of starting the game for your strategies. The main reason for considering this method for your fitness calls is the inconsistency of winning rate when you change the first player (at least for me). Thanks for reading my review.

Good luck,
Ahmadreza

Figure 5: Lab02 second review

3.6.2 Received

I have also received one review. I tried to consider both reviews in future works.

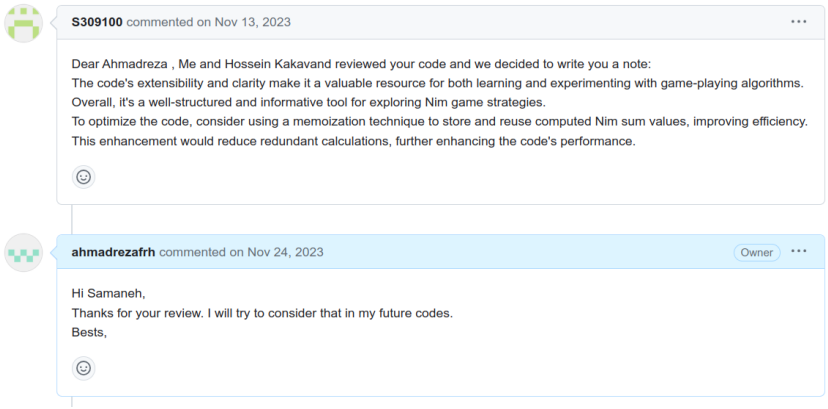


Figure 6: Received review for lab02

4 Lab09

In the Lab09 we were supposed to write a local-search algorithm (eg. an EA) able to solve the *Problem* instances 1, 2, 5, and 10 on a 1000-loci genomes, using a minimum number of fitness calls. That's all.

I have used following hyperparameters to train the ES agent.

```
N_GENERATIONS = 50_000
GENES = 1_000
POPULATION_SIZE = 600
OFFSPRING_SIZE = 400
TOURNAMENT_SIZE = 20
PATIENCE = 10
NUM_MUTS = 10
MUT_SPRINGS = 100
PROBLEMS = 1
```

I added a different functions which I inspired by $set-covering\_ea.ipynb$ existing in the original repository of the course. Moreover, I changed the mutate function that each iteration we can select random genes to mutate them instead of only one.

```
def mutate(ind: Individual) -> Individual:
    offspring = copy(ind)
    mut_point = random.sample(range(1, GENES-1), NUM_MUTS)
    for pos in mut_point:
        offspring.genotype[pos] = abs(offspring.genotype[pos] - 1)
    offspring.fitness = None
    return offspring
```

I also developed $cut\_multiple$ method which allow us to cut different points from genome and developing them.

```
def cut_multiple(ind_list: list) -> Individual:
    cut_points = random.sample(range(1, GENES-1), len(ind_list)-1)
    cut_points.sort()
    cut_points[:0] = [0]
    cut_points.append(GENES)
    offspring = create_offspring(ind_list, cut_points)
```

```
    assert len(offspring.genotype) == GENES
    return offspring
```

In the next steps I tried to achieve the best fitness calls through iterating over EA algorithm.

4.1 Problem size 1

I have repeated the the training with different problem sizes. In the following code you can see an example of the developed EA algorithm.

```
N_GENERATIONS = 50_000
GENES = 1_000
POPULATION_SIZE = 600
OFFSPRING_SIZE = 400
TOURNAMENT_SIZE = 20
PATIENCE = 10
NUM_MUTS = 10
MUT_SPRINGS = 100
PROBLEMS = 1


fitness = make_problem(PROBLEMS)
population = create_population(GENES)
pbar = trange(N_GENERATIONS)
mutation = False
best_fitnesses = []
best_fit = 0
p_count = 0
m_count = 0
tot_mutations=0

for generation in pbar:
    offspring = list()
    for counter in range(OFFSPRING_SIZE):
        if mutation:
            p = select_parent(population)
            o = mutate(p)
            m_count += 1
            if m_count>=MUT_SPRINGS:
                m_count=0
                p_count=0
                mutation=False
        else:
            pops = [select_parent(population) for _ in range(TOURNAMENT_SIZE)]
            o = cut_multiple(pops)
        offspring.append(o)

    for i in offspring:
        i.fitness = fitness(i.genotype)
    population.extend(offspring)
    population.sort(key=lambda i: i.fitness, reverse=True)
    population = population[:POPULATION_SIZE]
    if population[0].fitness <= best_fit:
        p_count+=1
    if p_count == PATIENCE:
        mutation = True
        tot_mutations+=1
```

```
    best_fit = population[0].fitness
    best_fitnesses.append(population[0].fitness)
    pbar.set_postfix({f"fitness": f"{best_fit:.2%}", f"calls": f"{fitness.calls}"})
    pbar.update(0)
    if tot_mutations>10 or best_fit==1:
        break
```

I achieved fitness of 99.4% which was satisfying. The overall fitness calls was 75000.

4.2 Problem size 2

Like problem size 1, we have also achieved good fitness calls in this step. We got the fitness of 83.60%. Although the results are lower than the first one but it was still satisfying. The overall calls reached 227400.

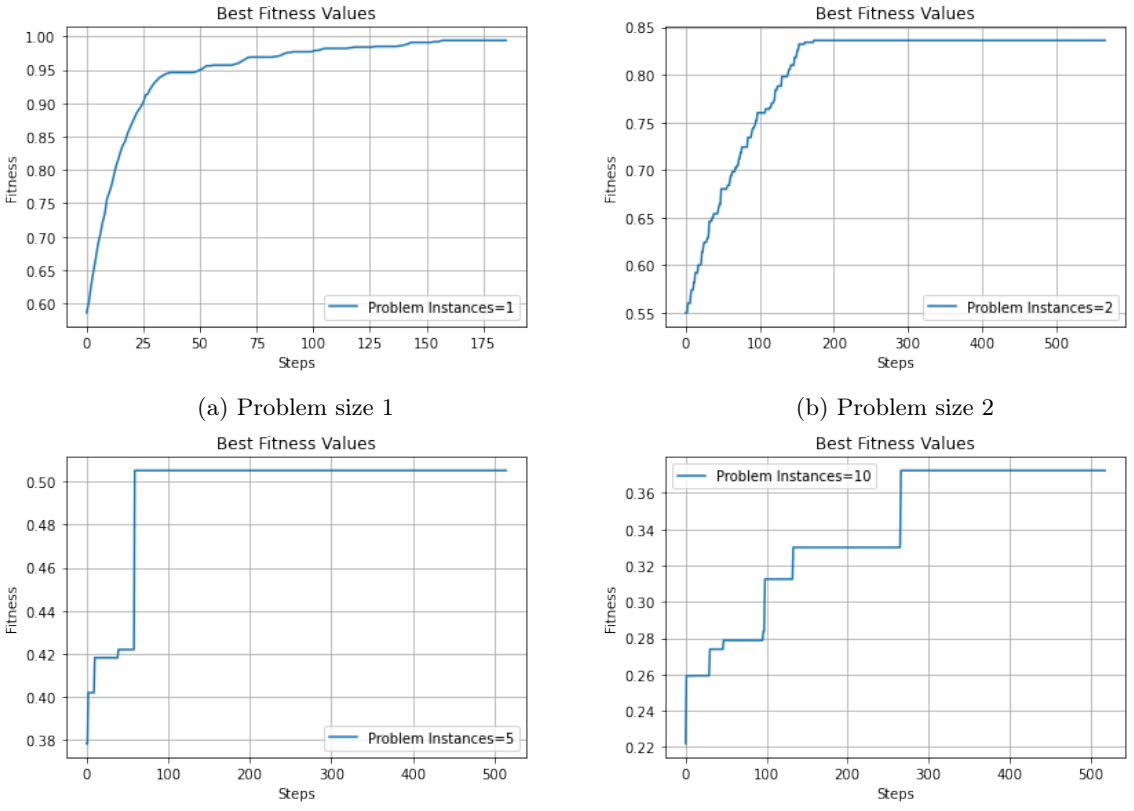
4.3 Problem size 5

I couldn't achieve desired fitness in this size. The overall fitness achieved 50.50%. The overall calls was 361400.

4.4 Problem size 10

Like problem size 5, we did not get good results in this size. The fitness achieved 37.23$ with 312400 calls.

In the fig. 7 we see the figures of each run.



(a) Problem size 1

(b) Problem size 2

(c) Problem size 5

(d) Problem size 10

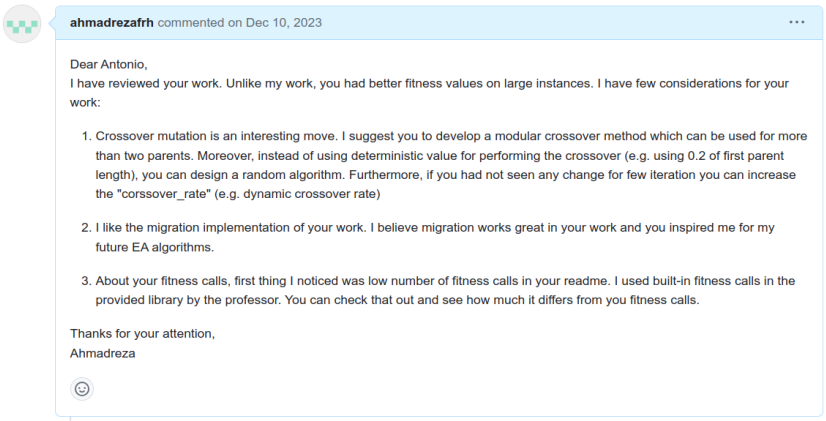
Figure 7: Progress figures of fitness calls

We can see from the figures that our algorithm could not reach better results than we got. For having better results we needed to develop more complex algorithms.

4.5 Reviews

4.5.1 Opened

I have opened two review for two of my colleagues.



ahmadrezafrh commented on Dec 10, 2023

Dear Antonio,
I have reviewed your work. Unlike my work, you had better fitness values on large instances. I have few considerations for your work:

1. Crossover mutation is an interesting move. I suggest you to develop a modular crossover method which can be used for more than two parents. Moreover, instead of using deterministic value for performing the crossover (e.g. using 0.2 of first parent length), you can design a random algorithm. Furthermore, if you had not seen any change for few iteration you can increase the "corssover_rate" (e.g. dynamic crossover rate)
2. I like the migration implementation of your work. I believe migration works great in your work and you inspired me for my future EA algorithms.
3. About your fitness calls, first thing I noticed was low number of fitness calls in your readme. I used built-in fitness calls in the provided library by the professor. You can check that out and see how much it differs from you fitness calls.

Thanks for your attention,
Ahmadreza

Figure 8: Lab09 first review

ahmadrezafrh commented on Dec 10, 2023

Dead Luca,

I have reviewed your work. I believe you have done perfect in paying attention to all different parts and algorithms individually. We have also some common implementations (e.g. we have almost same "n_cuts_xover"). It is interesting for me to see different implementation of my own ideas and I appreciated it and learned from your point of view. I try to write my view on your different algorithms:

1. In the "n_cuts_xover" I have also implemented a method which can use as many parents and Individuals I want. My suggestion is as follows (you can change total number of Individuals in the list which you pass it to the function). In the the example below I tried to modify your code based on my own suggestion:

```
def n_cuts_xover(ind_list: list) -> Individual:

    k = len(ind_list[0].genotype)
    cut_points = random.sample(range(1, k-1), len(ind_list)-1).sort()
    cut_points[:0] = [0]
    cut_points.append(k)

    offspring_genotype = []
    for i in range(len(ind_list)):
        offspring_genotype.extend(ind_list[i].genotype[cut_points[i]:cut_points[i+1]])

    offspring = Individual(fitness=None,
                           genotype=offspring_genotype)

    assert len(offspring.genotype) == k

    return offspring
```

2. For the mutation, you can add another method which mutate "n" number of positions (you can also perform it with swap).
3. In any steps of your code you can add a "patience" argument. If after some iterations you did not face any progress, you can change your hyperparameters such as "POPULATION_SIZE" and "MUTATION_PROBABILITY".

Thanks for your attention,
Ahmadreza

Figure 9: Lab09 second review

4.5.2 Received

I have also received one review. I tried to consider both reviews in future works.

5 Lab10

The purpose of the Lab10 was to reinforcement learning to devise a tic-tac-toe player.

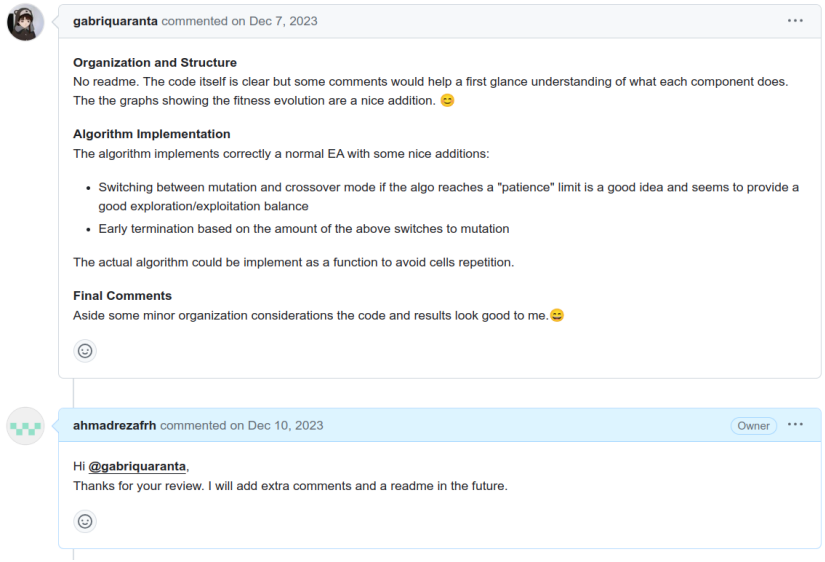


Figure 10: Received review for lab02

5.1 Evaluation method

Before initializing the reinforcement learning policy for playing the game, I define a metric that measures the winning rate when playing against a random player. The method considers both possibilities of starting the game, regardless of which player goes first. Furthermore, I attempt to create a class for selecting the next action from all available choices.

```
class Evaluation:
    def __init__(self, x_policy, o_policy):
        self.x_policy = x_policy
        self.o_policy = o_policy
        self._empty()

    def _empty(self):
        self.trajectory = list()
        self.state = State(set(), set())
        self.available = set(range(1, 9+1))

    def _get_stats(self, reward_list):
        return Counter(reward_list)


    def play(self, switch):
        while self.available:
            if not switch:
                x = self.x_policy.get(self.state, self.available)
            else:
                x = self.o_policy.get(self.state, self.available)

            self.state.x.add(x)
            self.trajectory.append(deepcopy(self.state))
            self.available.remove(x)
            if win(self.state.x) or not self.available:
                break

            if not switch:
```

```
                o = self.o_policy.get(self.state, self.available)
            else:
                o = self.x_policy.get(self.state, self.available)

            self.state.o.add(o)
            self.trajectory.append(deepcopy(self.state))
            self.available.remove(o)
            if win(self.state.o):
                break

        traj = deepcopy(self.trajectory)
        self._empty()
        return traj


    def evaluate(self, n_games):
        rewards = []
        for _ in tqdm(range(n_games)):
            tr = self.play(switch=False)
            reward = state_value(tr[-1])
            rewards.append(reward)
        stats = self._get_stats(rewards)


        print(f'total number of plays: {n_games}\n')
        print(f'winning accuracy of {self.x_policy.typ}: {stats[1]/n_games}\n')
        print(40*'_')
```

We have tried the evaluation method with random policies on both sides. We got a winning accuracy of 58% which we expected. The results are as following

```
x_policy = RandomPolicy()
o_policy = RandomPolicy()

eval = Evaluation(x_policy, o_policy)
eval.evaluate(1000)
```

```
total number of plays: 1000 winning accuracy of ranodm: 0.589
```

5.2 Policy and Value Iteration

Now we try to optimize the next action selection by value iteration. In the laboratory of the course, we played the game for many times to obtain "value_dictionary". Now we try to find optimum policy π^* that selects best action based on best values

Value iteration is an algorithm used in reinforcement learning to find the optimal value function and policy for a Markov decision process (MDP). The algorithm iteratively updates the value function for each state until convergence.

Update Equation

$$V_{k+1}(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V_k(s') \right)$$

Optimal Policy

$$\pi^*(s) = \arg\max_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^*(s') \right)$$

Value iteration converges to the optimal values and policy, making it a key algorithm in solving MDPs.

5.3 Optimum Policy

```
class OptimumPolicy:
    def __init__(self,):
        self.vd = defaultdict(float)
        self.typ = 'optimum'

    def set_vd(self, value_dictionary):
        self.vd = value_dictionary

    def get_vd(self):
        return self.vd

    def get(self, state: State, available: Set[int]) -> int:
        temp_val = {}
        for mov in available:
            new_state = deepcopy(state)
            new_state.x.add(mov)
            temp_val[mov] = self.vd[(frozenset(new_state.x), frozenset(new_state.o))]

        max_value = max(temp_val, key=temp_val.get)
        return max_value

    def iteration(self, n_steps):
        for steps in tqdm(range(n_steps)):
            trajectory = self._play()
            final_reward = state_value(trajectory[-1])
            for state in trajectory:
                hashable_state = (frozenset(state.x), frozenset(state.o))
                self.vd[hashable_state] = self.vd[
                    hashable_state
                ] + epsilon * (final_reward - self.vd[hashable_state])



    def _play(self):
        trajectory = list()
        state = State(set(), set())
        available = set(range(1, 9+1))

        while available:
            x = self.get(state, available)
            state.x.add(x)
            trajectory.append(deepcopy(state))
            available.remove(x)
            if win(state.x) or not available:
                break

            o = self.get(state, available)
            state.o.add(o)
```

```
            trajectory.append(deepcopy(state))
            available.remove(o)
            if win(state.o):
                break
        return trajectory
```

After I developed the class I tried to evaluate it with evaluation class I developed earlier.

```
x_policy = OptimumPolicy()
x_policy.set_vd(value_dictionary)
o_policy = RandomPolicy()


eval = Evaluation(x_policy, o_policy)
eval.evaluate(10_000)
```

The results are as following:

```
total number of plays: 10000 winning accuracy of optimum: 0.9893
```

Now we try to evaluate the π^* policy against random policy. We can also use value iteration to update the value dictionary to have better winning accuracy. All the rules implemented based on the assumption that our agent plays first.

Now let's take a look at the results regarding to policy iteration.

```
x_policy = OptimumPolicy()
x_policy.set_vd(value_dictionary)
o_policy = RandomPolicy()


eval = Evaluation(x_policy, o_policy)
eval.evaluate(10_000)
```

The results are as following:

```
total number of plays: 10000 winning accuracy of optimum: 0.9904
```

5.4 Q-Learning

We can see now after 500000 iteration, we got better optimum policy. Now we try to implement Q-Lerarning technique for further investigation. First let's discuss about Q-values.

In reinforcement learning, the Q-value $Q(s, a)$ represents the expected cumulative reward when an agent takes action a in state s. It is defined by the Bellman equation:

$$Q(s, a) = R(s, a) + \gamma \max_{a'} Q(s', a')$$

Here, $R(s, a)$ is the immediate reward, s is the next state, γ is the discount factor, and $\max_{a'} Q(s', a')$ is the maximum Q-value in the next state. Q-values are crucial in algorithms like Q-learning for optimizing an agent's policy. Based on the "values_dictionary" which we obtained during previous iteration, we consider immediate values of each $R(s, a)$ based on new learning of the values with respect to the action made in each state.

```
def new_random_game():
    trajectory = list()
    state = State(set(), set())
```

```
    available = set(range(1, 9+1))

    while available:
        x = choice(list(available))
        state.x.add(x)
        trajectory.append((deepcopy(state), x))
        available.remove(x)
        if win(state.x) or not available:
            break

        o = choice(list(available))
        state.o.add(o)
        trajectory.append((deepcopy(state), x))
        available.remove(o)
        if win(state.o):
            break
    return trajectory

q_table = defaultdict(float)
hit_state = defaultdict(int)
epsilon = 0.001

for steps in tqdm(range(1_000_000)):
    trajectory = new_random_game()
    final_reward = state_value(trajectory[-1][0])
    for state, action in trajectory:
        hashable_state = (frozenset(state.x), frozenset(state.o), action)
        hit_state[hashable_state] += 1
        q_table[hashable_state] = q_table[
            hashable_state
        ] + epsilon * (final_reward - q_table[hashable_state])
```

To develop the above equations, we initially defined another random game. Subsequently, we endeavored to construct a new dictionary. Following the training of our agent with the Q values, we created QPolicy to be evaluated alongside previous policies.

```
class QPolicy:
    def __init__(self,):
        self.q_table = defaultdict(float)
        self.typ = 'Q'

    def set_q_table(self, q_table):
        self.q_table = q_table

    def get_q_table(self):
        return self.q_table

    def get(self, state: State, available: Set[int]) -> int:
        temp_val = {}
        cp_state = deepcopy(state)
        for mov in available:
            temp_val[mov] = self.Q(cp_state, mov)
        max_value = max(temp_val, key=temp_val.get)
        return max_value

    def iteration(self, n_steps):
        for steps in tqdm(range(n_steps)):
```

```
        trajectory = self._play()
        final_reward = state_value(trajectory[-1][0])
        for state, action in trajectory:
            hashable_state = (frozenset(state.x), frozenset(state.o), action)
            self.q_table[hashable_state] = self.q_table[
                hashable_state
            ] + epsilon * (final_reward - self.q_table[hashable_state])



def _play(self):
    trajectory = list()
    state = State(set(), set())
    available = set(range(1, 9+1))

    while available:
        x = self.get(state, available)
        state.x.add(x)
        trajectory.append((deepcopy(state), x))
        available.remove(x)
        if win(state.x) or not available:
            break

        o = self.get(state, available)
        state.o.add(o)
        trajectory.append((deepcopy(state), x))
        available.remove(o)
        if win(state.o):
            break
    return trajectory


def R(self, state: State, action: int) -> float:
    return self.q_table[(frozenset(state.x), frozenset(state.o), action)]

def Q(self, state: State, action: int, gamma: float = 0.9) -> float:

    s = deepcopy(state)
    s.x.add(action)
    current_r = self.R(s, action)
    taken = s.x.union(s.o)
    available = set(MAGIC) - taken
    temp_val = []
    if available:
        for mov in available:
            next_state = deepcopy(s)
            next_available = deepcopy(available)
            next_state.o.add(mov)
            next_available = next_available - {mov}

            if next_available:
                for next_mov in next_available:
                    st_cpy = deepcopy(next_state)
                    n_val = self.Q(st_cpy, next_mov)
                    temp_val.append(n_val)
```

```
        if len(temp_val)==0:
            return current_r
        else:
            max_value = max(temp_val)
            return current_r + gamma*max_value
```

I ran the evaluation with the following code:

```
o_policy = RandomPolicy()
x_policy = QPolicy()
x_policy.set_q_table(q_table)

eval = Evaluation(x_policy, o_policy)
eval.evaluate(40)
```

Due to the limited hardware resources, I was unable to run the model for a high number of iterations. I conducted the model runs with 40 iterations, and the results are as follows:

```
total number of plays: 40 winning accuracy of Q: 1.0
```

5.5 Game Implementation

At the end of the lab I decided to create a simple GUI game where the opponent is the selected policy. The structure of the codes are as follows:

```
class color:
   PURPLE = '\033[95m'
   CYAN = '\033[96m'
   DARKCYAN = '\033[36m'
   BLUE = '\033[94m'
   GREEN = '\033[92m'
   YELLOW = '\033[93m'
   RED = '\033[91m'
   BOLD = '\033[1m'
   UNDERLINE = '\033[4m'
   END = '\033[0m'

def print_board(pos):
    """Nicely prints the board"""
    for r in range(3):
        for c in range(3):
            i = r * 3 + c
            if MAGIC[i] in pos.x:
                if c==2:
                    print(f'|{color.BOLD}X{color.END}|', end='')
                else:
                    print(f'|{color.BOLD}X{color.END}', end='')

            elif MAGIC[i] in pos.o:
                if c==2:
                    print(f'|{color.BOLD}O{color.END}|', end='')
                else:
                    print(f'|{color.BOLD}O{color.END}', end='')
            else:
                if c==2:
                    print(f'|{MAGIC[i]}|', end='')
                else:
```

```
                print(f'|{MAGIC[i]}', end='')
        print()
    print()
```

Moreover, two other methods have been defined for running the game:

```
def get_input(available):
    x = int(input(f"Enter your move from {available}:  "))
    if x not in available:
        print(f'{x} is not available')
        x = get_input(available)
    return x

def experiment(policy):

    trajectory = list()
    state = State(set(), set())
    available = set(range(1, 9+1))

    while available:
        print(f'\ncurrent board with available positions in\n')
        print_board(state)
        x = get_input(available)
        state.x.add(x)
        trajectory.append(deepcopy(state))
        available.remove(x)
        if win(state.x):
            print(f"\nCongrats! you win the game.\n")
            print(30*'_')
            break
        if not available:
            print(f"\nit's draw.\n")
            print(30*'_')
            break

        o = policy.get(state, available)
        state.o.add(o)
        trajectory.append(deepcopy(state))
        available.remove(o)
        if win(state.o):
            print(f"\nYou messed up unfortunately!\n")
            print(30*'_')
            break

        print(30*'_')
    print('\n')

    play_again = str(input("Do you want to play again? (y/n) :"))
    if play_again=="y":
        experiment(policy)
```

To run the game we should use the following codes:

```
x_policy = OptimumPolicy()
x_policy.set_vd(value_dictionary)
experiment(x_policy)
```

In the fig. 11, we have a sample screen shot of how the game is ran with the jupyter notebook.

```
_____________________________

current board with available positions in

|O|7|6|
|9|5|X|
|4|3|8|

Enter your move from {3, 4, 5, 6, 7, 8, 9}:  5
_____________________________

current board with available positions in

|O|7|6|
|9|X|X|
|4|O|8|

Enter your move from {4, 6, 7, 8, 9}:  9

Congrats! you win the game.

_____________________________
```

Figure 11: Tic-Tac-Toe Gui Interface

5.6 Reviews

5.6.1 Opened

Due to the Christmas holidays I missed the deadline.

5.6.2 Received

I have also received one review. I tried to consider both reviews in future works.

Figure 12: Received review for lab02

6 Holloween Challenge

I participated in the Halloween challenge. In the first step I tried to set an optimization method for getting optimum solutions. I added count variable in fitness call to achieve the correct number of fitness calls.

```
def optimization(current_state, num, curr_fit_val=(0,0), steps=10_000, show_log=False):



    for step in range(steps):
        new_state = tweak(current_state, num)
        fit_val = fitness(new_state)
        if fit_val >= curr_fit_val:
            current_state = new_state
            curr_fit_val = fit_val
            if show_log:
```

```
                print(f'Fitness call: {count}')
                print(f'New state fitness: {fit_val}\n')


    return current_state
```

I defined a window size for windowing the solution size instead of doing one by one to increase the performance. The following function is a hill climbing method with considering window technique.

```
def hill_climbing_windowed(x, win_size, fitness_count=0):
    hc_states = []
    while fitness(hc_states)[0]!=x.shape[0]:
        current_val=0
        val_list = hc_states.copy()
        val_window = [randint(0, x.shape[0]—1) for p in range(0, win_size)]
        for ins in val_window:
            if ins not in hc_states:
                val_list.append(ins)
                val,_ = fitness(val_list)
                if val>current_val:
                    index=ins
                    current_val=val
                val_list.remove(ins)

        hc_states.append(index)
    return hc_states
```

In the next attempt, I tried to modify tweak method, I add two features into this method:

- Popping randomly from current state (to increase the exploration rate and avoiding local optimums)
- Creating a random index vector from original indexes. Choosing the indexes that have lower similarity with existing indexes (based on how many intersection they have.)

```
def tweak_modified(state, problem_size, add_step=2, pop_num=1, win_size=30):
    new_state = state.copy()
    shuffle(new_state)
    if pop_num<len(new_state):
        for _ in range(pop_num): new_state.pop()

    if len(new_state)==0:
        index = np.random.randint(0, problem_size — 1)


    val_window = [randint(0, x.shape[0]—1) for p in range(0, win_size)]
    min_diff = np.inf
    for i in val_window:
        for n in new_state:
            intsec = np.intersect1d(np.unique(np.any(x[[n]]).nonzero()[1]),np.unique(np.any(x[[i]]).nonzero()[1]))
            if intsec.shape[0] < min_diff:
                min_diff = intsec.shape[0]
                index = i


    new_state.remove(index) if index in new_state else new_state.append(index)

    return new_state
```

Unfortunately, this method adds extra time which not worth it. However, we obtain better results compared to two first methods.

Now we start analyzing all results together. We use three main methods:

- Method provided by professor
- Optimized hill climbing (my method)

```
num_points = [(10,100), (10,1_000), (10,5_000)] # first element of the tuples is window size
density_list = [.3, .7]
methods = ['basic', 'windowed_hc']

'''
Basic:   Professor method (tweaking from initial empty list)
Mine :   Windowed hill climbing
'''


for num in num_points:
    for density in density_list:
        for method in methods:
            print(f'Method        : {method}\nDensity
                : {density}\nProblem size  : {num[1]}\n')

            count=0
            x = make_set_covering_problem(num[1], num[1], density)
            hc_states = hill_climbing(x)
            print(f'Greedy fitness call: {count}')
            print(f'Greedy optimum: {fitness(hc_states)}\n')

            count=0
            if method=='basic':
                current_state = []
            if method=='windowed_hc':
                current_state = hill_climbing_windowed(x, win_size=num[0])
                print(f'Fitness call: {count}')
                print(f'New state fitness: {fitness(current_state)}\n')

            # Calculating greedy optimums with hill hill climbing

            opt_state = optimization(current_state, num=x.shape[0],
                                     show_log=True)
```

The final results are shown in fig. 14. I was able to achieve good performance in each density and covering sizes.

6.1 Reviews

I have received a review for this challenge. The colleague suggested to update my fitness count. Inn the updated version of Halloween challenge in my repository I considered his suggestion.

7 Final Project

For the final Quixo project, I employed two primary methods. Initially, I implemented a deterministic algorithm that does not require any training and serves as a straightforward solution.

Subsequently, for the second method, I employed deep reinforcement learning algorithms to develop an intelligent agent. Within this method, I utilized three optimization techniques, which I will briefly explain in this section.

Problem Size	Density	Method	Fitness Calls	Solution Size
100	0.3	Basic Approach	12	10
100	0.3	Windowed Hill Climbing	**100**	**8**
100	0.3	Greedy Hill Climbing	592	6
100	0.7	Basic Approach	4	4
100	0.7	Windowed Hill Climbing	**125**	**3**
100	0.7	Greedy Hill Climbing	301	3
1000	0.3	Basic Approach	314	14
1000	0.3	Windowed Hill Climbing	**141**	**12**
1000	0.3	Greedy Hill Climbing	9966	10
1000	0.7	Basic Approach	6	6
1000	0.7	Windowed Hill Climbing	**56**	**5**
1000	0.7	Greedy Hill Climbing	3999	4
5000	0.3	Basic Approach	602	19
5000	0.3	Windowed Hill Climbing	**210**	**19**
5000	0.3	Greedy Hill Climbing	64936	13
5000	0.7	Basic Approach	7	7
5000	0.7	Windowed Hill Climbing	**67**	**6**
5000	0.7	Greedy Hill Climbing	24996	5

Figure 13: Holloween Challenge Results

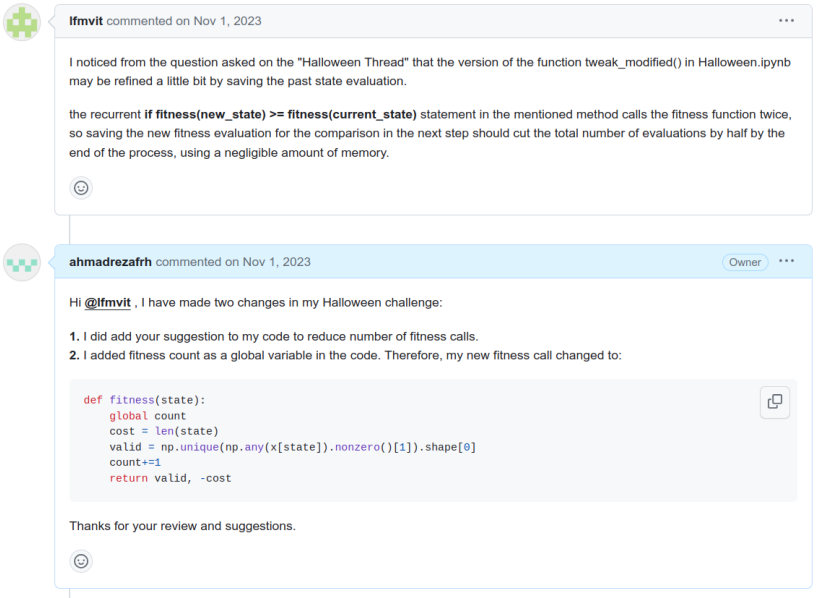


Figure 14: Holloween Challenge Review

Additionally, I attempted to implement value iteration, as demonstrated in Lab 10; however, I encountered challenges and did not achieve any significant results.

7.1 Deterministic Agent

The agent behaves randomly, but when it identifies an action that could lead to a winning position, it prioritizes and executes that action.

```
class DeterministicPlayer(Player):
    def __init__(self) -> None:
        super().__init__()
        self.pos_ranges = [range(0,5),range(0,5)]
        self.all_pos = list(itertools.product(*self.pos_ranges))
        self.available_pos = []
        for pos in self.all_pos:
            row, col = pos
            from_border = row in (0, 4) or col in (0, 4)
            if from_border:
                self.available_pos.append(pos)

    def check_winner(self, board, player_id) -> int:
        '''Check the winner. Returns the player ID of the winner if any, otherwise returns -1'''
        player = player_id
        winner = -1
        for x in range(board.shape[0]):
            if board[x, 0] != -1 and all(board[x, :] == board[x, 0]):
                winner = board[x, 0]
        if winner > -1 and winner != player:
            return winner
        for y in range(board.shape[1]):
            if board[0, y] != -1 and all(board[:, y] == board[0, y]):
                winner = board[0, y]
        if winner > -1 and winner != player:
            return winner
        if board[0, 0] != -1 and all(
            [board[x, x]
                for x in range(board.shape[0])] == board[0, 0]
        ):
            winner = board[0, 0]
        if winner > -1 and winner != player:
            return winner
        if board[0, -1] != -1 and all(
            [board[x, -(x + 1)]
             for x in range(board.shape[0])] == board[0, -1]
        ):
            winner = board[0, -1]
        return winner


    def get_moves(self, game):
        available_actions = []
        for pos in self.available_pos:
            new_pos = deepcopy((pos[1], pos[0]))
            if game._board[new_pos] != game.current_player_idx and game._board[new_pos] != -1:
                continue
            available_slides = acceptable_slides(new_pos)
            for slide in available_slides:
                available_actions.append((pos, slide))
        return available_actions
```

```
    def get_action_results(self, game, available_actions):
        no_wins = []
        for action in available_actions:
            pos = action[0]
            mov = action[1]
            axis_0 = pos[1]
            axis_1 = pos[0]
            cp_board = deepcopy(game._board)
            cp_board[(pos[1], pos[0])] = game.current_player_idx

            if mov == Move.RIGHT:
                cp_board[axis_0] = np.roll(cp_board[axis_0], -1)
            elif mov == Move.LEFT:
                cp_board[axis_0] = np.roll(cp_board[axis_0], 1)
            elif mov == Move.BOTTOM:
                cp_board[:, axis_1] = np.roll(cp_board[:, axis_1], -1)
            elif mov == Move.TOP:
                cp_board[:, axis_1] = np.roll(cp_board[:, axis_1], 1)

            winner = self.check_winner(cp_board, game.current_player_idx)
            if winner==game.current_player_idx:
                return action
            if winner==-1:
                no_wins.append(action)


        return no_wins



    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        moves = self.get_moves(game)
        best_actions = self.get_action_results(game, moves)
        if type(best_actions)==list:
            action = random.choice(best_actions)
        else:
            action = best_actions

        return action
```

This agent is a variation of RandomPlayer which is not reliable, so I decided to develop more advanced agent.

7.2 Value Iteration

I attempted to develop a value dictionary. However, a significant challenge of this method was the vast number of available states. The calculation of the values for each state was not feasible, given that the number of required states is equal to $3^{25} - 12$.

To address this issue, I designed a player capable of capturing a trajectory for each game and subsequently back-propagating the state values in the dictionary.

```
class ValuePlayer(Player):
    def __init__(self) -> None:
        super().__init__()

        self.pos_ranges = [range(0,5),range(0,5)]
```

```
        self.all_pos = list(itertools.product(*self.pos_ranges))
        self.available_pos = []
        for pos in self.all_pos:
            row, col = pos
            from_border = row in (0, 4) or col in (0, 4)
            if from_border:
                self.available_pos.append(pos)

        self.set_trajectory()

    def set_trajectory(self):
        self.trajectory = []

    def get_moves(self, game):
        available_actions = []
        for pos in self.available_pos:
            new_pos = deepcopy((pos[1], pos[0]))
            if game._board[new_pos] != 1 and game._board[new_pos] != -1:
                continue
            available_slides = acceptable_slides(new_pos)
            for slide in available_slides:
                available_actions.append((pos, slide))
        return available_actions


    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        moves = self.get_moves(game)
        from_pos, move = random.choice(moves)
        self.trajectory.append(deepcopy(game._board))
        return from_pos, move
```

I have developed a value class which is as follows:

```
class ValueClass:
    def __init__(
        self,
        player1: Player,
        player2: Player
    ):
        self.player1 = player1
        self.player2 = player2
        self.set_value_dictionary()

    def set_value_dictionary(
        self,
    ):
        self.value_dictionary = defaultdict(float)
        self.model_exist = False

    def train(
        self,
        epsilon: float = 0.001,
        steps: int = 5_000_000,
        checkpoints: int = 50_000,
        resume: str = None
    ) -> None:
```

```
        if not self.model_exist:
            if resume:
                self.value_dictionary = defaultdict(float,load_dict(resume))
            for steps in tqdm(range(steps)):
                g = Game()
                self.player2.set_trajectory()
                final_reward = g.play(self.player1, self.player2)
                for state in self.player2.trajectory:

                    hashable_state = "".join(map(str, list(state.reshape(1, -1)[0])))
                    self.value_dictionary[hashable_state] = round(self.value_dictionary[
                        hashable_state
                    ] + epsilon * (final_reward - self.value_dictionary[hashable_state]), 5)

                if steps%checkpoints==0 and steps!=0:
                    self.model_exist = True
                    self.save(file_name=f'cpt-{steps}.json', exist_ok=True)
            self.model_exist = True
        else:
            print('model exists, try to empty the value dictionary first')

    def save(
        self,
        models_path: str = './models',
        file_name: str = 'model.json',
        exist_ok: bool = False

    ) -> None:

        if self.model_exist:
            file_path = f'{models_path}/{file_name}'
            os.makedirs(models_path, exist_ok=True)
            if os.path.isfile(file_path):
                if exist_ok:
                    save_dict(self.value_dictionary, file_path)
                    print(f'Value dictionary have been rewritten in {file_name}')
                else:
                    print('Value dictionary exists')

            else:
                save_dict(self.value_dictionary, file_path)
                print(f'Value dictionary have been created in {file_name}')

        else:
            print('Model is not trained, train the model first')


    def load(
        self,
        file_path: str = './models/value_dictionary.json'
    ) -> None:

        assert os.path.isfile(file_path)
        self.model_exist = True
        self.value_dictionary = load_dict(file_path)
        print('value dictionary loaded')
```

```
    def top(
        self,
        n: int = 5,
        reverse: bool = True
    ):
        print(sorted(self.value_dictionary.items(), key=lambda e: e[1], reverse=reverse)[0:n])
```

Despite all the effort I could not reach any results.

7.3 Gym Envronment

Before advancing to the final phase of developing the agent using deep reinforcement learning, I had to create an environment for the Quixo game. Building this environment is essential for training faster and more modular models that can be easily modified in future development.

I utilized Gym, an open-source Python library designed for developing and comparing reinforcement learning algorithms. Gym provides a standard API for seamless communication between learning algorithms and environments, along with a standardized set of environments compliant with that API. Since its release, Gym's API has become the industry standard for these purposes.

```
class QuixoEnv(Env):
    def __init__(self, player: int):
        self.actions = get_available_actions()
        self.action_space = Discrete(len(self.actions))
        self.observation_space = Box(
            low=-30, high=30, shape=(len(self.actions)+5*5, ), dtype=np.int32)
        self.board = -np.ones((5, 5), dtype=np.int8)
        self.player = player
        self.opposite = 0 if self.player == 1 else 1
        if self.player == 1:
            result = self.op_move()
            assert result == None

    def slide(self, pos, slide):
        axis_0 = pos[1]
        axis_1 = pos[0]

        if slide == Move.RIGHT:
            self.board[axis_0] = np.roll(self.board[axis_0], -1)
        elif slide == Move.LEFT:
            self.board[axis_0] = np.roll(self.board[axis_0], 1)
        elif slide == Move.BOTTOM:
            self.board[:, axis_1] = np.roll(self.board[:, axis_1], -1)
        elif slide == Move.TOP:
            self.board[:, axis_1] = np.roll(self.board[:, axis_1], 1)

    def get_moves(self, player):
        a_actions = []
        for i in range(len(self.actions)):
            if self.board[(self.actions[i][0][1], self.actions[i][0][0])] != player:
                a_actions.append(i)

        return a_actions

    def get_action_results(self):
        rewards = []
        count = 0
```

```
        neg_count = 0
        for i in range(len(self.actions)):
            pos = self.actions[i][0]
            mov = self.actions[i][1]
            if self.board[(pos[1], pos[0])] == self.player:
                rewards.append(0)
            else:
                axis_0 = pos[1]
                axis_1 = pos[0]
                cp_board = deepcopy(self.board)
                cp_board[(pos[1], pos[0])] = self.opposite

                if mov == Move.RIGHT:
                    cp_board[axis_0] = np.roll(cp_board[axis_0], —1)
                elif mov == Move.LEFT:
                    cp_board[axis_0] = np.roll(cp_board[axis_0], 1)
                elif mov == Move.BOTTOM:
                    cp_board[:, axis_1] = np.roll(cp_board[:, axis_1], —1)
                elif mov == Move.TOP:
                    cp_board[:, axis_1] = np.roll(cp_board[:, axis_1], 1)

                winner = self.check_winner(cp_board, self.player)
                if winner == self.opposite:
                    rewards.append(—1)
                    neg_count += 1
                elif winner == self.player:
                    rewards.append(2)
                    count += 1
                else:
                    rewards.append(1)

        return rewards, count, neg_count

    def step(self, action):
        self.ep_count += 1
        result = self.player_move(action)
        if result:
            return result
        result = self.op_move()
        if result:
            return result

        self.action_results, count, neg_count = self.get_action_results()

        if self.ep_count >= 35:
            self.score = 1 + count + (35 — self.ep_count)
        else:
            self.score = 1 + count

        info = {}
        self.observation = np.append(self.board.flatten(), np.array(
            self.action_results, dtype=np.int32))
        winner = —1
        info['winner'] = winner
        info['detail'] = "no one won"
        return self.observation, self.score, self.done, info
```

```
    def player_move(self, action):
        mapped_action = self.actions[action]
        pos = mapped_action[0]
        mov = mapped_action[1]

        if self.board[(pos[1], pos[0])] == self.opposite:
            self.score = -5
            self.action_results = len(self.actions) * [0]
            self.observation = np.append(self.board.flatten(), np.array(
                self.action_results, dtype=np.int32))
            info = {}
            winner = -1
            info['winner'] = winner
            info['detail'] = "selected position in board taken"
            return self.observation, self.score, self.done, info

        self.board[(pos[1], pos[0])] = self.player
        self.slide(pos, mov)

        winner = self.check_winner(self.board, self.player)
        if winner == self.player:
            #             if self.ep_count<10:
            #                 self.score = 100
            if self.ep_count < 20:
                self.score = 80
            elif self.ep_count < 30:
                self.score = 60
            elif self.ep_count < 40:
                self.score = 40
            elif self.ep_count < 60:
                self.score = 20
            else:
                self.score = 10
            self.action_results = len(self.actions) * [20]
            self.done = True
            info = {}
            self.observation = np.append(self.board.flatten(), np.array(
                self.action_results, dtype=np.int32))
            info['winner'] = winner
            info['detail'] = f"player {self.player} won with his own move"
            return self.observation, self.score, self.done, info

        elif winner == self.opposite:
            self.score = -10
            self.action_results = len(self.actions) * [15]
            self.done = True
            info = {}
            self.observation = np.append(self.board.flatten(), np.array(
                self.action_results, dtype=np.int32))
            info['winner'] = winner
            info['detail'] = f"player {self.opposite} won with oponent move"
            return self.observation, self.score, self.done, info

        return None

    def op_move(self):
        op_actions = self.get_moves(self.player)
```

```
        op_action = random.choice(op_actions)
        op_pos, op_mov = self.actions[op_action]
        self.board[(op_pos[1], op_pos[0])] = self.opposite
        self.slide(op_pos, op_mov)
        winner = self.check_winner(self.board, self.opposite)
        if winner == self.opposite:
            self.score = -5
            self.action_results = len(self.actions) * [-10]
            self.done = True
            info = {}
            self.observation = np.append(self.board.flatten(), np.array(
                self.action_results, dtype=np.int32))
            info['winner'] = winner
            info['detail'] = f"player {self.opposite} won with his own move"
            return self.observation, self.score, self.done, info

        elif winner == self.player:
            self.score = 0
            self.action_results = len(self.actions) * [-5]
            self.done = True
            info = {}
            self.observation = np.append(self.board.flatten(), np.array(
                self.action_results, dtype=np.int32))
            info['winner'] = winner
            info['detail'] = f"player {self.player} won with oponent move"
            return self.observation, self.score, self.done, info

        return None

    def check_winner(self, board, player_id) -> int:
        '''Check the winner. Returns the player ID of the winner if any, otherwise returns -1'''
        player = player_id
        winner = -1
        for x in range(board.shape[0]):
            if board[x, 0] != -1 and all(board[x, :] == board[x, 0]):
                winner = board[x, 0]
        if winner > -1 and winner != player:
            return winner
        for y in range(board.shape[1]):
            if board[0, y] != -1 and all(board[:, y] == board[0, y]):
                winner = board[0, y]
        if winner > -1 and winner != player:
            return winner
        if board[0, 0] != -1 and all(
            [board[x, x]
                for x in range(board.shape[0])] == board[0, 0]
        ):
            winner = board[0, 0]
        if winner > -1 and winner != player:
            return winner
        if board[0, -1] != -1 and all(
            [board[x, -(x + 1)]
             for x in range(board.shape[0])] == board[0, -1]
        ):
            winner = board[0, -1]
        return winner
```

```
    def reset(self):
        self.done = False
        self.ep_count = 0
        self.score = 0
        self.board = —np.ones((5, 5), dtype=np.int32)
        self.action_results = len(self.actions)*[1]
        if self.player == 1:
            result = self.op_move()
            assert result == None
            self.action_results, _, _ = self.get_action_results()

        self.observation = np.append(self.board.flatten(), np.array(
            self.action_results, dtype=np.int32))
        return self.observation

    def update_board(self, board):
        self.board = deepcopy(board)

    def get_obs(self):
        action_results, _, _ = self.get_action_results()
        observation = np.append(self.board.flatten(), np.array(
            action_results, dtype=np.int32))
        return observation

    def experiment(self):
        t = 0
        observation = self.reset()
        tot_reward = 0
        while True:
            t += 1
            action = self.action_space.sample()
            observation, reward, done, info = self.step(action)
            tot_reward += reward
            if done:
                break

        print(f"episode length: {t}")
        print(f"total reward: {tot_reward}")
        print(f"winner: player {info['winner']}")
```

In developing the environment, it was crucial to define rewards and observations. The observation space is a one-dimensional vector containing the flattened version of the board, and the rewards for each available action.

Rewards for available actions are assigned based on their feasibility. The reward is determined by the winning status of each action. The actual rewards returned in each step aid the agent in learning and identifying parameters that can maximize total rewards. In the initial training attempts, I assigned rewards to the agent whether it won or not. If the agent lost, it received negative rewards, while winning actions received positive rewards.

However, other factors were also important. One crucial factor was the selection of an action that was available. For this purpose, if the agent selected an action not available on the board, it received a negative reward (less than the negative reward for losing).

After implementing the changes above, I observed that the agent selected the right actions but sometimes chose actions that did not lead to a winning position (for itself and the opponent). To address this issue, I adjusted the winning rewards to be dynamic and dependent on the episode length. If the agent won in the early steps, it received higher rewards.

Since the opponent makes a move in the same step, the agent needed to select actions that put the opponent in a non-winning position (to improve performance against a deterministic player).

7.4 Stable-baselines3

Stable Baselines3 (SB3) is a set of reliable implementations of reinforcement learning algorithms in PyTorch. It is the next major version of Stable Baselines. For developing future agents I used the original documentations of this framework. Moreover, It is also compatible with gym environment.

7.5 Actor-Critic Models

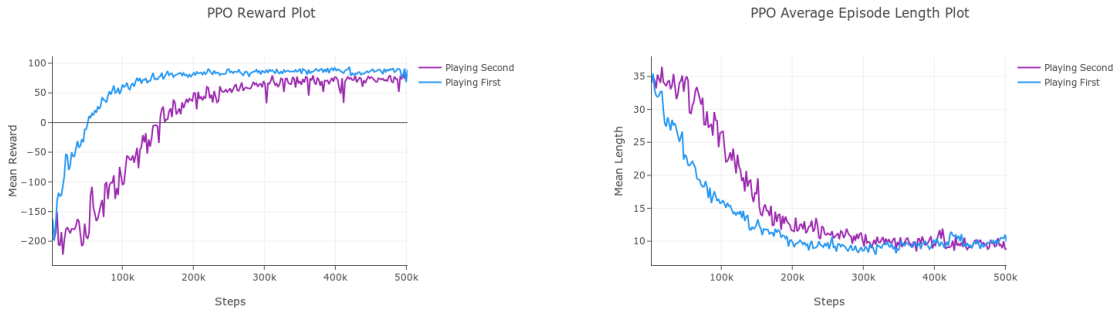
In deep reinforcement learning, the Actor-Critic framework combines elements of policy-based and value-based methods. The "actor" learns the policy to choose actions, and the "critic" learns the value function to evaluate states or actions. The actor is trained to increase the likelihood of actions leading to higher estimated values, guided by the critic's evaluation. This hybrid approach offers stability and flexibility, leveraging neural networks to handle both discrete and continuous action spaces.

NOTE: The following algorithm explanation are not written by me, it is just a brief introduction to the algorithms. I researched through different algorithms and I selected the following three algorithms.

7.5.1 Proximal Policy Optimization (PPO)

PPO is a reinforcement learning algorithm designed to optimize policy functions. It aims to strike a balance between stability and sample efficiency. We have trained yhe model for each starting position.

In the fig. 15 we have the progress of rewards in the training. We can see good progress in the received rewards. Model works well in both starting positions.



(a) Episode Mean Reward

(b) Episode Mean Length

Figure 15: PPO plots over 500000 iteration

Objective Function:

The objective function for PPO is defined as the clipped surrogate objective:

$$L_t(\theta) = \hat{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right],$$

where:

- $L_t(\theta)$ is the objective function at time step t.
- θ represents the policy parameters.
- $\hat{E}_t$ is the empirical expectation over a batch of samples.
- $r_t(\theta)$ is the ratio of new and old policy probabilities.
- $\hat{A}_t$ is the advantage function.

- ϵ is a hyperparameter controlling the extent of policy change.

Optimization:

PPO employs optimization methods, such as stochastic gradient ascent, to maximize the objective function with respect to the policy parameters θ.

Clipping:

The clipping term ensures that the policy update does not deviate too far from the old policy, promoting stability during training.

```
class PPOWrapper:
    def __init__(self, players: list[int] = [0,1], models_dir: str = './models/ppo',):
        self.players = [0,1]
        self.models_dir = models_dir

    def train(
        self,
        net_arch: dict = dict(pi=[1024, 512, 256, 128], vf=[32, 32, 32 , 32]),
        policy = 'MlpPolicy',
        verbose = 1,
        progress_bar = True,
        ts = int(2e6),
        callbacks = 50_000
    ):

        for player in self.players:
            checkpoint_callback = CheckpointCallback(
                save_freq=callbacks,
                save_path=f"{self.models_dir}/player_{player}",
                name_prefix="quixo_dqn",
                save_replay_buffer=False,
                save_vecnormalize=False,

            )
            policy_kwargs = dict(net_arch=net_arch)
            env = QuixoEnv(player=player)
            with open(os.devnull, "w") as f, contextlib.redirect_stdout(f):
                model = PPO(policy, env, policy_kwargs=policy_kwargs,
                            verbose=verbose, tensorboard_log="./logs/ppo")
            model.learn(total_timesteps=ts, progress_bar=progress_bar,
                        callback=checkpoint_callback, tb_log_name=f"./quixo_ppo_{player}")
            model.save(f"{self.models_dir}/quixo_ppo_{player}")
            del model

    def experiment(
        self,
        trials=100

    ):
        trials = trials
        for player in self.players:
            env = QuixoEnv(player=player)
            with open(os.devnull, "w") as f, contextlib.redirect_stdout(f):
                model = PPO.load(
                    f"{self.models_dir}/quixo_ppo_{player}", env=env)
            winners = 0
```

```
            done = False
            f = 0
            for i in range(trials):
                obs = env.reset()
                done = False
                while not done:
                    action, _states = model.predict(obs, deterministic=True)
                    moves = env.get_moves(env.opposite)
                    if action not in moves:
                        action = random.choice(moves)
                        f += 1
                    obs, rewards, done, info = env.step(action)
                if player == 0 and info["winner"] == 0:
                    winners += 1
                elif player == 1 and info["winner"] == 1:
                    winners += 1
            print("\n")
            if player == 0:
                print("playing first")
            else:
                print("playing second")
            print(f"win percentage: {winners/trials}")
            print(f"total wrong actions: {f}\n")
            print("-----")
```

For training an agent with PPO algorithm we have used stable-baselines3 to create a wrapper for the agent and the training phase. I have changed the final linear network architecture. We have now 4 linear layers for selecting final action.

Since values are highly dependent on whether the agent plays first or second, I have trained two agents for starting first and second. The final PPOPlayer is as follows:

```
class PPOPlayer(Player):
    def __init__(self) -> None:
        super().__init__()
        self.env0 = QuixoEnv(player=0)
        self.env1 = QuixoEnv(player=1)
        self.env0.reset()
        self.env1.reset()

        self.models = []
        with open(os.devnull, "w") as f, contextlib.redirect_stdout(f):
            self.models.append(
                PPO.load("./models/ppo/quixo_ppo_0", env=self.env0))
            self.models.append(
                PPO.load("./models/ppo/quixo_ppo_1", env=self.env1))

    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        if game.current_player_idx == 0:
            action = self.get_action(game, self.env0, self.models[0])
        else:
            action = self.get_action(game, self.env1, self.models[1])
        return action

    def get_action(self, game, env, model):
        opposite = 0 if env.player == 1 else 1
        env.update_board(game._board)
        obs = env.get_obs()
```

```
        action, _ = model.predict(obs, deterministic=True)
        env.update_board(game._board)
        moves = env.get_moves(opposite)
        if action not in moves:
            action = random.choice(moves)
        return env.actions[action]
```

7.5.2 Deep Q-Network (DQN)

DQN is a reinforcement learning algorithm that combines Q-learning with deep neural networks to approximate and learn a policy for decision-making in an environment.

Q-Learning Background:

In traditional Q-learning, an agent learns a Q-function, denoted as $Q(s, a)$, representing the expected cumulative future rewards for taking action a in state s. The Q-function is updated based on the Bellman equation:

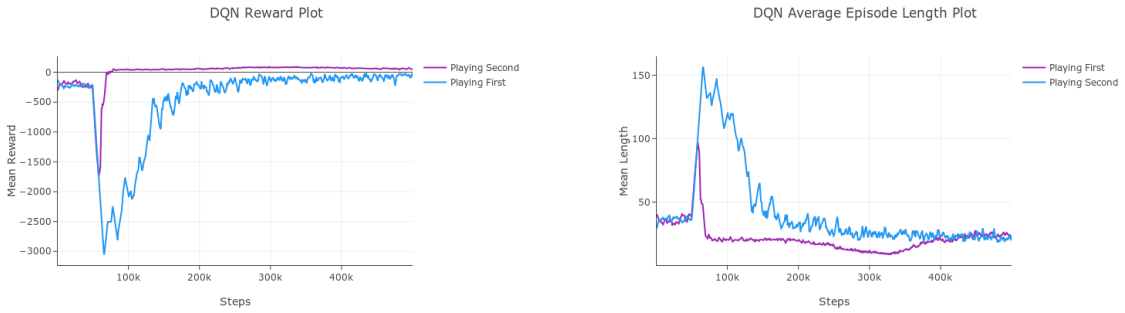
$$Q(s, a) = R + \gamma \max_{a'} Q(s', a')$$

where R is the immediate reward, γ is the discount factor, and s' is the next state.

DQN Architecture:

DQN extends Q-learning to handle complex state spaces using a deep neural network to approximate the Q-function. The network takes the state as input and outputs Q-values for each possible action. The Q-network is trained to minimize the temporal difference error.

In the fig. 16 we have the progress of rewards in the training. We can see the results are not satisfying compared to PPO. This algorithm was not successful.



(a) Episode Mean Reward

(b) Episode Mean Length

Figure 16: DQN plots over 500000 iteration

Experience Replay:

DQN employs experience replay, storing past experiences in a replay buffer. During training, random batches of experiences are sampled to break correlations and stabilize learning.

Target Network:

To improve stability, DQN uses two Q-networks: the target network and the online network. The target network's parameters are updated less frequently, providing a stable target for the temporal difference error.

Loss Function:

The DQN loss function is a combination of the temporal difference error and regularization to prevent overestimation bias:

$$L(\theta) = E\left[\left(Q(s,a;\theta) - (r + \gamma \max_{a'} Q(s',a';\theta^-))^2\right)\right]$$

where θ represents the parameters of the online network, and θ^- represents the parameters of the target network.

Exploration vs. Exploitation:

DQN often uses an epsilon-greedy strategy to balance exploration and exploitation, exploring with probability ϵ and exploiting the current best policy otherwise.

We have developed an DQN agent wrapper and a DQNPlayer class for evaluation. The structure of the classes are the same and are reported in my original final project repository in GitHub.

7.5.3 Advantage Actor-Critic (A2C)

A2C is a reinforcement learning algorithm that combines elements of both policy-based and value-based methods for more stable and efficient training.

Objective Function:

The A2C objective function is a combination of policy and value functions:

$$L(\theta) = \hat{E}_t\left[\nabla_\theta \log(\pi_\theta(a_t|s_t)) \cdot A_t + \beta \cdot (V_\phi(s_t) - V_{\phi_{\text{old}}}(s_t))^2\right],$$

where:

- $L(\theta)$ is the objective function.
- θ represents the policy parameters.
- $\pi_\theta(a_t|s_t)$ is the policy distribution.
- A_t is the advantage function.
- β is a hyperparameter.
- $V_\phi(s_t)$ is the value function.
- ϕ represents the value function parameters.

Actor-Critic Structure:

A2C employs an actor-critic structure where the actor (policy) and critic (value function) are neural networks. The actor is responsible for selecting actions, and the critic evaluates the state-value.

Advantage Function:

The advantage function, A_t, measures how much better or worse an action is compared to the average action in a given state. It is a crucial component in the objective function.

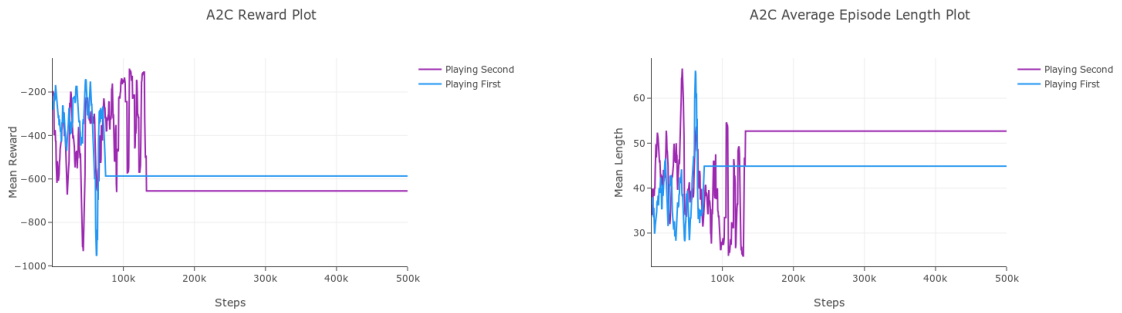
In the fig. 17 we have the progress of rewards in the training. We can see the results are not satisfying compared to PPO. This algorithm was not successful.

Training:

During training, the policy and value functions are updated using the gradient of the objective function with respect to their respective parameters.

Entropy Regularization:

A2C often includes entropy regularization ($-\beta \cdot H(\pi_\theta)$) in the objective function to encourage exploration:



(a) Episode Mean Reward

(b) Episode Mean Length

Figure 17: A2C mean reward over 500000 iteration

$$L(\theta) = \hat{E}_t \left[\nabla_\theta \log(\pi_\theta(a_t|s_t)) \cdot A_t + \beta \cdot (V_\phi(s_t) - V_{\phi_{\text{old}}}(s_t))^2 - \beta \cdot H(\pi_\theta) \right],$$

where $H(\pi_\theta)$ is the entropy of the policy distribution.

Like DQN and PPO, we have developed an A2C agent wrapper and a A2CPlayer class for evaluation. The structure of the classes are the same and are reported in my original final project repository in GitHub.

7.6 Results

After I developed the models, I tried to evaluate all players with each other. I have used the following function for the evaluation:

```
def evaluate(
    eplayer: Player,
    opponent: Player,
    enum: int = 10,
):
    """
    Evaluate the performance of a player against an opponent.

    Parameters:
    – eplayer (Player): Player to evaluate.
    – opponent (Player): Opponent player.
    – enum (int): Number of evaluation trials.

    Returns:
    – None
    """
    win1 = 0
    win2 = 0
    with tqdm(total=2 * enum) as pbar:

        for i in range(enum):
            g = Game()
            winner = g.play(eplayer, opponent)
            if winner == 0:
                win1 += 1
            pbar.update(1)

        for i in range(enum):
```

```
            g = Game()
            winner = g.play(opponent, eplayer)
            if winner == 1:
                win2 += 1
            pbar.update(1)

    print(f"TOTAL # of trials:              {2 * enum}")
    print(f"WIN RATIO —— PLAYING SECOND:    {round(win1 / enum, 3)}")
    print(f"WIN RATIO —— PLAYING FIRST:     {round(win2 / enum, 3)}")
    print(f"TOTAL WIN RATIO:                {round((win2 + win1) / (2 * enum), 3)}")
```

As we discussed earlier, it is important whether which player starts as the first. In the table. 1 in the main player column we have the player we wanted to do evaluation on. We have winning percentage of playing as first and second. Moreover we have the total winning ratio.

Table 1: Overall results of developed players in Quixo game

Main Player	Opponent Player	Iterations	Playing First	Playing Second	Total Winning
RandomPlayer	RandomPlayer	1000	0.56	0.61	0.58
DeterministicPlayer	RandomPlayer	1000	0.89	0.87	0.88
PPOPlayer	RandomPlayer	1000	**0.96**	**0.91**	**0.93**
PPOPlayer	DeterministicPlayer	1000	0.87	0.62	0.74
DQNPlayer	RandomPlayer	1000	0.65	0.00	0.32
DQNPlayer	DeterministicPlayer	1000	0.27	0.00	0.14
A2CPlayer	RandomPlayer	1000	0.82	0.77	0.80
A2CPlayer	DeterministicPlayer	1000	0.58	-0.51	0.55

Our results indicate superior performance when playing as the first player. As demonstrated in the table, the most favorable outcomes were achieved with the PPOPlayer, outperforming in both possible positions.

To enhance results, especially for the DQN and A2C players, a thorough review of the rewarding function in the developed environment is necessary. It may be beneficial to write different versions of the environment tailored to each algorithm.

Additionally, custom feature extractors can be written for these players. The Stable-baselines3 library provides comprehensive documentation on creating these custom feature extractors. Adjusting the final linear network architecture (linear layers after feature extractors) is another avenue to explore for performance improvement.

In summary, the best player is found within the PPOPlayer. You can run the *main.py* file in the project repository to view the results of winning. Furthermore, we can use following command in terminal:

```
$ python3 evaluate.py
```

This command evaluates of all the models. Moreover, we can use the following command to train all the developed models:

```
$ python3 train.py
```

All the arguments and hyperparameters for each function and class are well-documented within the code. For developing your own models with your specified hyperparameters, you can leverage the developed classes.

8 Conclusion

Throughout the course, I gained a better understanding of the concepts of reasoning and training a smart agent, which can process information, think logically, and act rationally. Initially uncertain about

whether I made the right decision at the beginning of the course, I gradually acquired more knowledge about the subject over time.

A noticeable aspect of my work is the progress I made over various labs. Even my coding style improved as I studied contributions from other students. The presentations by fellow students were instrumental in helping me understand different problem-solving approaches.

Participating in challenges, such as the Halloween challenge, was particularly interesting for me. I suggest incorporating more challenges and perhaps offering rewards for the best results. The competition enhances our ability to think more algorithmically and fosters continuous improvement in our problem-solving skills. Thank you.