Student: Salvatore Incaviglia s305947

Teacher: Giovanni Squillero

Course: Computational Intelligence

REPORT ON COURSE PROJECTS

The purpose of this report is to describe the main activities carried out during the Computational Intelligence course. In particular, the objective is to take a look at the workshops carried out and understand the solutions adopted to resolve each problem. Furthermore, through the issues carried out starting from laboratory number 2, I will explain what I saw in the implementation of the other people for the same problems .

Laboratory 1: A-Star algorithm on Set Covering Problem

The A* algorithm is a heuristic search algorithm used to find the optimal path between two points in a graph, such as a map or grid. The main objective of the A* algorithm is to find the shortest path from a starting point to a destination point, also considering the costs associated with movements between the various nodes of the graph. The algorithm uses two values for each node:

G(n): the actual cost of reaching the current node from a starting node.

H(n): a heuristic that estimates the minimum cost of reaching the destination node from the current node.

The set covering problem concerns the selection of the smallest subset of sets (or "sets") whose union contains all the elements of a given universe.

The next two functions below are important parts of my implementation of the A* algorithm for set covering. **goal_check(state)** is the function that checks whether a state satisfies the goal of the set covering problem. Then, check whether the union of the sets corresponding to the indices present in state.taken covers all elements of the problem. The new_var variable is calculated using the reduce function with the logical OR operator (np.logical_or). This variable gives us the union of all the sets corresponding to the indices present in state.taken. np.all(new_var) gives us True if all the elements in

new_var are True, and therefore the objective is achieved. Instead, **distance(state)** is the function that provides the "distance" of a state from the initial state. Use the reduce function with the logical OR operator (np.logical_or) to calculate the union of all sets corresponding to the indices in state.taken. This represents the items covered up to this point. The function tells us which set has been covered up to the current point in the search. Now we move on to the definition of the heuristics used which give us an estimate of the minimum cost to reach the optimal solution from the current state.

heuristic(state) is the first heuristic. It begins by calculating the elements covered up to this point using the distance(state) function. If all elements are covered (np.all(covered) is True), it returns 0 because we are already in the goal. Otherwise, it calculates the maximum of the sum of uncovered elements in each set and returns the integer division of the maximum by the number of overall uncovered elements (opt). This tells us how much we may need to cover in the future to reach the solution. **heuristic2(state)** is the second heuristic and it is a function that calculates the number of elements already covered in a state. His logic is based on the fact that a state with more elements covered is probably closer to the solution. Returns the length of the state's taken field, which represents the number of items already covered. **combined_heuristic(state)** is the function that combines the previous two heuristics using a weighted model. The idea is to assign a greater weight to the first heuristic than to the second, but the specific weight is controlled by the weight1 and weight2 variables. The combined result represents the overall estimate of the residual cost to reach the optimal solution from the current state. Finally I used a function crudely called "funzione" which represents the formula f(n)=g(n)+h(n) where I combined the path length and the combined heuristic. In conclusion, after having defined a priority queue as a frontier for the search states and having initialized all the not_taken states at the beginning, after a series of cycles I see how many steps the algorithm has taken to find the solution with the function it also uses my heuristic. Below is a look at the function of the heuristics I used and then combined giving more weight to the former:

```python
def heuristic(state):
    ricoperti=distance(state)

    if np.all(ricoperti):
        return 0
    massimo=max(sum(np.logical_and(s,np.logical_not(ricoperti))) for s in SETS)


    sum_not_covered=PROBLEM_SIZE-sum(ricoperti)

    opt=ceil(massimo/sum_not_covered)
    return opt
```

```python
def heuristic2(state):

    return len(state.taken)
```

```python
def combined_heuristic(state):
    weight1 = 0.7  # weght for the first heuristic
    weight2 = 0.3  # weight for the second heuristic
#I combine two heuristics, it is a way that is often used in computer science to try to optimize problems.
#My aim is to reduce the number of steps to arrive at the solution and to do this I combine two heuristics giving them a we
#For example, in this case I combined one of the heuristics I had developed, which later turned out to be similar to the on
# with another heuristic that calculates the number of taken. It should be all optimistic and the number of steps decreases
#We can observe that one of these two heuristic could be pessimistic (the second in particular) but the result at the end c
# It depends on the weights of different heuristics.
# I know that probably the second heuristic is not optimistic and i chose a weight that is smaller than the weight that i u
    heuristic_value1 = heuristic(state)
    heuristic_value2 = heuristic2(state)

    combined_value = weight1 * heuristic_value1 + weight2 * heuristic_value2


    return ceil(combined_value)
```

HALLOWEEN CHALLENGE-HILL CLIMBING PROBLEMS

The Hill Climbing algorithm is a local technique and can only reach local minimums or maximums. It is a technique that is based on some characteristics such as defining a formal representation of the solutions in the specific problem. An objective function is defined that assigns a value to each possible solution, representing how "good" that solution is compared to the objectives of the problem. An initial solution is then randomly selected. Through a fitness function the values of the objective functions in the current solution are calculated and with the tweak functions the current solutions are slightly modified to obtain a new solution. If the solution obtained now is better than the previous one then it is used as a local solution, otherwise the exploration continues.

In my solution for the set covering problem I tried to represent the different variants of the Hill Climbing algorithm with variations in the fitness function for the evaluation phase**.** I used to implement the different variations of Hill Climbing and I implemented different functions to do it: **fitness1(state)** calculates the total cost by summing the state elements. It tests the validity of the state using a sparsely encoded array (sets_sparse).It returns a tuple with the validity value and the negative cost. Validity is a Boolean value, and cost is negative because often in optimization problems we look for the minimum, so it is common to negate the value. **fitness2(state)** is similar to fitness1, but it returns the number of valid elements instead of a Boolean value. It returns a tuple with the number of valid elements and the negative cost. **fitness3(state)** calculates the total cost by summing the state elements. It uses the tweak function to generate a new state configuration. It tests the validity of the new state against a sparsely encoded array. If the new state is valid, it returns the negative cost of the state. Otherwise, it calculates a probability based on the cost difference between the current state and the new state and applies some sort of probabilistic acceptance criterion. If the new state is accepted, the negative cost of the new state is returned; otherwise, the negative value of infinity is returned. **tweak(state)** function creates a copy of the state and randomly reverses the value of one of the elements. **LocalSearch(state)** creates a new state configuration by randomly changing one of the elements. It compares the fitness of the new state with that of the current state using fitness2. If the fitness of the new state is greater than or equal to that of the current state, the current state is returned. Otherwise, the new state is returned. Now let's take a look at the implemented hill climbing variants:

1. SIMPLE HILL CLIMBING:

```
##############################################
#    HILL CLIMBING WITH                      #  ### CON LA FITNESS1 è il miglior metodo mentre con la FITNESS2 il miglic
#    SIMPLE HILL CLIMBING                    #
##############################################
current_state = [choice([False, False, False, False, False, False]) for _ in range(NUM_SETS)]
counter=0
for step in range(10_000):#This is the normal execution used for HILL CLIMBING
    new_state = tweak(current_state)
    if fitness2(new_state) >= fitness2(current_state): #We are calculating the quality of solution with fitness functior
        current_state = new_state

        counter+=1


print(f"The final counter to reach the Hill Climbing with Simple Hill Climbing is {counter}")
```

```
The final counter to reach the Hill Climbing with Simple Hill Climbing is 15
```

Here I used fitness2 to compare the solution of the new modified state with the current state and this solution counts the number of states that have a number of valid elements greater than the current state.

2. SIMULATED ANNEALING:

```
##############################################
#    HILL CLIMBING WITH                      #
#    SIMULATED ANNEALING                     #
##############################################
current_state = [choice([False, False, False, False, False, False]) for _ in range(NUM_SETS)]
counter=0
for step in range(10_000):#This is the normal execution used for HILL CLIMBING
    new_state = tweak(current_state)
    if fitness3(new_state) >= fitness3(current_state): #We are calculating the quality of solution with fitness functior
        current_state = new_state

        counter+=1


print(f"The final counter to reach the Hill Climbing with Simulated Annealing is {counter}")
```

```
The final counter to reach the Hill Climbing with Simulated Annealing is 20
```

We remember that fitness3 is the fitness that includes the definition of evaluation for simulated annealing. The solution is similar to the previous solution with differences in fitness function.

3. ITERATED LOCAL SEARCH: here the solution includes the local search function that is similar to the tweak function but it compares the states in terms of fitness2.

```
###############################################
#    HILL CLIMBING WITH                       #
#    ITERATED LOCAL SEARCH                    #
###############################################
current_state = [choice([False, False, False, False, False, False]) for _ in range(NUM_SETS)]
counter=0
#stato_perturbato = perturbation(current_state)
for step in range(10_000):#This is the normal execution used for HILL CLIMBING
    stato_perturbato = LocalSearch(current_state)
    new_state=tweak(stato_perturbato)
    if fitness1(new_state) >= fitness1(current_state): #We are calculating the quality of solution with fitness functior
        current_state = new_state

        counter+=1


print(f"The final counter to reach the Hill Climbing with Iterated Local Search is {counter}")
```

```
he final counter to reach the Hill Climbing with Iterated Local Search is 96
```

4. TABU SEARCH:

```
###############################################
#    HILL CLIMBING WITH                       #
#    TABU SEARCH                              #
###############################################


current_state = [choice([False, False, False, False, False, False]) for _ in range(NUM_SETS)]
tabu=[]
counter=0
best_state=current_state
for step in range(10_000):

    tmp= [tweak(current_state) for _ in range(NUM_SETS)]
    best = None

    for neighbor in tmp:
        if neighbor not in tabu:
            if best is None or fitness1(neighbor) > fitness1(best):
                best = neighbor

    if best is not None:
        current_solution = best
        tabu.append(current_solution)

        if fitness1(current_solution) > fitness1(best_state):
            best_state = current_solution

    counter += 1
    tabu.pop(0)

print(f"The final counter to reach the Hill Climbing with Tabu Search is {counter}")
```

```
The final counter to reach the Hill Climbing with Tabu Search is 10000
```

You can see how in the tabu search he made an error in the cycle that considers the number of steps to arrive at the optimal solution.

Laboratory 2: NIM GAME WITH EVOLUTIONARY STRATEGY

Nim is a mathematical strategy game played between two players. You have piles with an odd number of elements per increasing pile that starts from 1 and continues every two elements. The game consists of two players who alternate with one move at a time. The player who gets the last item is the winner. By using a good strategy you can always win in this game. The strategy that is required to be implemented is an evolutionary strategy. My idea was to use a strategy that included the moves represented by the Nimply class with the two fields ROW and N_OBJECTS. I thus developed an evolutionary strategy that I included in my solution and which had the aim of creating a population of individuals with a certain number of mutations or recombinations. Then everything was completed with an increase in the population by adding lambda elements into the population. I carried out the selection of the "parents" following the TOURNAMENT SELECTION strategy and from there I calculated the best strategy whose aim was to use the nim_sum function to select the strategy that was best based on the state of the current game. The result obtained is that of an evolutionary mode that adapted to the situation that the game posed. The function I use is "savior" which includes the evolutionary strategy called "es_with_tournment". I had the user play with both Gabriele and Optimal which represent the other two strategies.

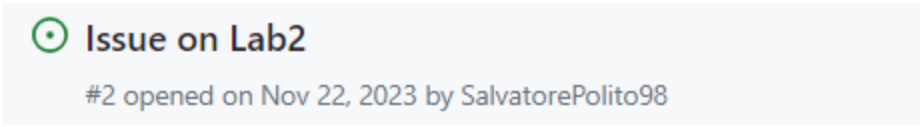
This is the code of evolutionary strategy that i used:

```
def es_with_tournament(state, population_size, mutation_rate, sigma, lambda_value):

    best_strategy = None
    lista=[]

    for _ in range(population_size):
        if random.random() < mutation_rate:
            population = GaussianMutation(state, sigma,mutation_rate)
        else:
            population = Recombination(state)

        vincitore,valore_fitness=tournament_selection(population,state)

        lista.append((vincitore,valore_fitness))

        # Make λ new individuals casually and update it as a population part
        for _ in range(lambda_value):
            row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
            num_objects = random.randint(1, state.rows[row])
            new_individual = (row, num_objects)
            population.append(new_individual)


    lista_ordinata_decrescente = sorted(lista, key=lambda x: x[1], reverse=True)
    buona_strategia=lista_ordinata_decrescente[0][0]
    # Trova la tupla con il secondo elemento massimo

    if buona_strategia[1]<=state.rows[buona_strategia[0]] and buona_strategia is not None:
        best_strategy=buona_strategia

    if best_strategy is None:
        row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
        num_objects = random.randint(1, state.rows[row])
        return (row, num_objects)

    return best_strategy
```

In this code I implemented a gaussian mutation and a recombination with a one cut crossover strategy. As i just said i also used a tournament selection to select the best strategies and i returned the best strategy that is tuple (row, num_objects) that I can use for my final strategy called “salvatore”.

LABORATORY 2 - REVIEWS

As regards the reviews, I carried out 2 reviews taken from the codes of 2 of my colleagues on the course. In the first Review I commented on the code of a colleague of mine(Hossein Kakavand s308581) in which I explained a little what he wanted to do in broad terms and then I also wrote: "This algorithm was designed to play an evolutionary strategy to the NIM game. The population is initialized with randomly chosen parameters that have a value between -1 and 1 and for each individual the fitness function counts how many games are won a total of times. The best strategies are also selected. We then have the function of mutation of the strategies with the subsequent replacement of the mutated population. Everything is repeated for a series of cycles and at the end the strategy with the best fitness of the final population is returned. "evolved_strategy_function" is the function that uses the parameters of the best evolved strategy to make moves in the Nim game based on specific logic.The code is clear and readable and the various comments provide further feedback to better understand all the phases of the algorithm.The use of the evolutionary approach is also clear thanks to the code which is modular with all the functions for mutation, fitness and evolved strategy well explained in the code.`` I did this review on November 22nd as shown in the image below.



```python
def evolved_strategy(state: Nim, param):
    """A strategy function using an evolved parameter to influence move choice."""
    # Convert the parameter to a probability bias
    bias = (param + 1) / 2  # Now it's between 0 and 1

    current_nim_sum = nim_sum(state)

    if current_nim_sum == 0:
        # If the nim-sum is zero, make a move that tries to create a non-zero nim-sum
        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):
                tmp_state = deepcopy(state)
                tmp_state.nimming(Nimply(r, remove))
                if nim_sum(tmp_state) != 0:
                    return Nimply(r, remove)
        return pure_random(state)  # Fallback to random if no such move is found
    else:
        # If the nim-sum is non-zero, find a move that changes it 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):
                tmp_state = deepcopy(state)
                tmp_state.nimming(Nimply(r, remove))
                if nim_sum(tmp_state) == 0:
                    return Nimply(r, remove)
        return pure_random(state)  # Fallback to random if no such move is found

best_strategy_params = evolve_strategy(Nim(4, k=3), population_size=1, generations=10, top_n=5, mutation_rate=0.1)
evolved_strategy_function = lambda state: evolved_strategy(state, best_strategy_params['param'])
```

In the image above I show the strategy used by my colleague given by the evolved_strategy function which allows the player to execute the move according to the rules of the genetic algorithms.

The second strategy I carried out instead comments on the code of another colleague of mine(Mario Francesco De Pascale s318048), highlighting the fact that his code is quite clear and modular. The comment is this: "The code is very clear upon first reading and using the classes allows the possibility of putting together functions that are then correlated. The comments present allow us to clarify any doubts, if there were any. Furthermore, the fact that there are function type annotations improves the readability of the code and allows you to have no doubts about the return types.At the evolutionary strategy level, all the main functions of an evolutionary algorithm are present such as:

-initialization of the population

-Parent selection mechanisms

-Fitness calculation

-Ordering of the population

-Mutation and Recombination"This review was carried out by me on November 21, 2023 as evidenced by the image below:

LABORATORY 9 - EVOLUTIONARY ALGORITHM

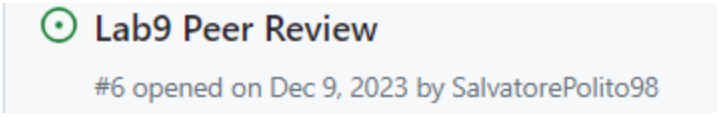
Evolutionary algorithms are a class of optimization algorithms inspired by the principles of biological evolution. The main goal of these algorithms is to solve optimization problems, that is, to find the best or approximately optimal solution in a search space. Their strength lies in the ability to handle complex and non-linear solution spaces, offering a robust and adaptable search strategy. In my case, what was required was a strategy that could include among others an evolutionary algorithm and that implemented in any way a local search for the best result, minimizing if possible the number of calls to the fitness function. It is stated that in my solution, after selecting the parents based on the value of the fitness function and sorting the list of parents subsequently, I then took the evolutionary solution which allowed me in some way to combine a local search for the maximum for the parents and then I developed mutations and crossovers on the population obtained to then have the final result. In my solution I have implemented a class called EA_MIO in which through an internal run() method I can show what I have developed. Inside this run() method I iterate for each generation ordering based on the fitness value and by doing so I find a population that could already represent a local maximum. I take the two best elements and according to my logic I initialize the parents. Subsequently I have to implement an evolutionary algorithm and to do this I took the example seen in the lesson slides of the Mutation strategy (very simple strategy inverting the values of every certain number of elements) + Crossover (multi-point crossover). Finally, after creating the population for generation, I select the best element found based on fitness. In doing so I found the best value, always according to my logic of course. Below I wanted to include the screen of the run() method to show what the modus operandi of my solution was, which constitutes a starting point for an evolutionary algorithm that can partially include local search, even if with all the limitations of the case .

```
def run(self):
    # Initialize population
    self.population = self.start()
    offspring = []
    best_individual = None
    parents=[]

    ################################################################
    #  Strategy Used is Mutation+ recombination similar to Strategy1 #
    ################################################################

    for _ in range(self.generations):

        fitnesses = [self.fitness_function(individual) for individual in self.population]
        lista_ordinata = sorted(self.population, key=lambda x: sum(x), reverse=True)
        ##Here we have parent selection that i did selecting the elements in the population that have a lot of 1 inside.
        parents1=lista_ordinata[0]
        parents2=lista_ordinata[1]
        parents=[parents1,parents2]

        #Mutation of elements of parents
        for i in range(len(parents)):
            if i%3==0:
                mutation_index = randint(0, self.genome_length - 1)
                parents[i][mutation_index] = 1 - parents[i][mutation_index]

        # Crossover (One cut crossover strategy or multi point crossover -> there are two different points but i decided to to multi point)
        while len(offspring) < self.population_size:
            parent1, parent2 = choices(parents, k=2)
            crossover_index = randint(1, self.genome_length - 1)
            crossover_index_list = [randint(1, self.genome_length-1) for _ in range(7)]
            crossover_index_list_ordinata = sorted(crossover_index_list)
            for i in range(len(crossover_index_list_ordinata)-1):
                if random() < self.mutation_rate:
                    child = parent1[:crossover_index_list_ordinata[i]] + parent2[crossover_index_list_ordinata[i]:crossover_index_list_ordinata[i+1]] + parent1[crossover_index_list_ordinata[1+1]:]
                else:
                    child = parent2[:crossover_index_list_ordinata[i]] + parent1[crossover_index_list_ordinata[i]:crossover_index_list_ordinata[i+1]] + parent2[crossover_index_list_ordinata[1+1]:]
            offspring.append(child)
        self.population = offspring
    best_individual = max(self.population, key=self.fitness_function)
```

LABORATORY 9 REVIEWS

For lab 9 I carried out two reviews in which I commented on the code of two of my other colleagues(Raul Gatto s316048 and Salvatore Tilocca s305938); the first says: “The code is well structured and the comments present serve to simplify the understanding of the slightly more complex parts. The developed functions are clear and allow us to develop the genetic functions of the proposed algorithms. A suggestion to improve the code would be to evaluate the use of encapsulation and variables used within a class. This

however is not a major criticism, just a suggestion. For reasons of readability it would be better to remove the commented sections. However, the evaluation for the form of the code is very positive. From the point of view of genetic algorithms, elitism appears to be implemented with the preservation of the best individuals and tournament selection which allows maintaining a good selection of individuals". In my opinion it made sense to highlight a class in which particular cases were made only for a question of readability, given that there are multiple sections that implement perhaps the same thing but in a different way by changing the parameters. As proof of the progress of my review, I report the screen with the date on which I carried it out.



The second review that I did on the same day as the previous says: "The code is well structured, the use of @DataClass also allows you to save in terms of writing code. The methods highlighted in the first part are also clear from the name given to the methods themselves. The code is very clear even if there are no additional comments. In the Mutation Crossover part, the methods introduced to develop the crossover are also clear in the writing of the code and introduce the variability required by the EA. The part relating to the implementation is written with the help of some comments that clarify implementation doubts and the graph shown allows us to graphically understand what has been written in the code. Furthermore, it also shows the results obtained and their progress". The thing that I found most important in this code was the presence of the graph that plots the fitness values based on generations, which makes it clearer what was found in the solution.

LABORATORY 10- REINFORCEMENT LEARNING ALGORITHM

Reinforcement Learning is a machine learning paradigm where an agent learns to make decisions by interacting with an environment. The main goal of reinforcement learning is to make the agent learn to take actions in order to maximize a performance measure, called "reward". The learning process occurs through trial and error, where the agent explores the environment, takes actions and receives rewards.

Reinforcement learning uses specific algorithms, such as Q-learning which is the method I tried to implement in the aforementioned lab. In particular, in my case a general Reinforcement learning strategy must be implemented which has the aim of winning against any player in the tic-tac-toe game.

In my implementation of the Q_learning Reinforcement Learning algorithm it is important to take a look at the various parameters that I used to then understand the final solution:

Q_table is a data structure representing the Q function, which associates a pair (state, action) with a numerical value representing the expected utility of performing that action in that state. **learning_rate** is instead an alpha parameter, which determines how much the agent learns from new data compared to how much it remembers from old data. **discount_factor** is the phi parameter, which indicates how much the agent values future rewards over immediate rewards. **exploration_prob** is the probability with which the agent explores new actions instead of following the current policy. **num_episodes** is the total number of learning episodes, where an episode represents an entire game or sequence of actions up to a terminal state. **initialize_Q** is a function that initializes the Q-table for a specific state. Other functions of interest in my implementation, before showing how the algorithm moves, are:

select_player_action--> function that selects player action based on exploration/exercise policy. During the initial episodes, the agent explores new actions with high probability. As time passes (episodes), the probability of exploration decreases.

select_opponent_action--> function that selects the opponent's action randomly. This simulates an opponent making random moves. Now we can see the final representation of the algorithm:

```
wins_X = 0
wins_O = 0
deuce=0
# Algoritmo Q-learning con giocatore che inizia con valore della X prima
for episode in tqdm(range(num_episodes)):
    # Reset del gioco
    board = [' '] * 9
    reward = 0
    while True:
        # Stato corrente
        state = get_state()
        # Selezione dell'azione per il giocatore
        action = select_player_action(state,episode)
        # Esegui la mossa del giocatore
        make_move(action, player_symbol)
        # Valuta lo stato dopo la mossa del giocatore
        result = evaluate_state()

        if result is not None:
            reward = result
            if reward == 1:
              wins_X += 1
            elif reward == -1:
              wins_O += 1
            else:
              deuce+=1
            break  # Il gioco è terminato
        # Stato successivo
        next_state = get_state()
        # Selezione dell'azione dell'avversario (casuale)
        opponent_action = select_opponent_action()
        # Esegui la mossa dell'avversario
        make_move(opponent_action, opponent_symbol)
        # Valuta lo stato dopo la mossa dell'avversario
        result = evaluate_state()

        if result is not None:
            reward = result
            if reward == 1:
             wins_X += 1
            elif reward == -1:
             wins_O += 1
            else:
              deuce+=1
            break  # Il gioco è terminato

        # Stato successivo
        new_state = get_state()

        # Aggiornamento della tabella Q solo per il giocatore
        initialize_Q(state)
        initialize_Q(new_state)
        Q_table[state][action] = (1 - learning_rate) * Q_table[state][action] + learning_rate * (reward + discount_factor *
total_games = num_episodes
win_percentage_X = (wins_X / total_games) * 100
win_percentage_O = (wins_O / total_games) * 100
```

The algorithm executes a number of episodes defined by num_episodes, where an episode represents a complete match between the agent and the adversary. For each episode, the game is reset and the agent makes moves following the policy learned from the Q-table. The select_player_action(state, episode) function is used to select the action of player "X" based on the exploration policy. The agent finally executes the move and evaluates the resulting state.

If the game ends (one of the players wins or ends in a draw), the win count is updated and the game ends. Then it is the turn of opponent "O" to make random moves, and the process is repeated. After each move of agent "X" and adversary "O", the Q-table update is performed based on the Q-learning update formula seen in class. At the end of the cycle, the win percentages for player "X" and player "O" are calculated over the total number of games (total_games). In my last case, the strategy for ‘X’ with the Q Learning algorithm achieved a win percentage of around 50% versus a 37% win rate for ‘O’.

LABORATORY 10 - REVIEWS

I have included two reviews regarding laboratory 10. In the first regarding the code of a colleague of mine(Mario Francesco De Pascale s318048) I state as follows: "This little project uses a class TicTacToe to manage the game state, players, and moves. The code initializes the game with a hidden board, representing values associated with each move, and an empty playable board. Players take turns making moves, and the game checks for a winner or a draw after each move. The code presents an implementation of a machine learning agent for the game of tic-tac-toe, based on model-free Q-learning with some minmax strategies. The agent plays against itself to learn and then competes against a human player or performs performance evaluations against random choices. Among the positive characteristics we can include that the agent implements Q-learning, a model-free machine learning technique, to improve its moves in the tic-tac-toe game. The code incorporates exploration and deepening strategies via the epsilon parameter, allowing the agent to balance the exploration of random moves and the use of acquired knowledge. The code manages the flow of the game, keeping track of the agent's states, moves and training rewards. The code includes a loop of performance tests in which the agent competes against random players as both the first and second players. To enhance the agent's evaluation, it would be beneficial to incorporate specific metrics during its training. For instance, we could monitor its learning ability over time by observing how scores or the percentage of wins against random opponents improve. A more detailed analysis of its evolution could involve tracking metrics such as the variation in Q-values, the frequency of random choices versus Q-value-based choices, and the understanding of specific strategies learned during the training process." The second review that I did to my colleague (Samaneh

Ghareh Dagh Sani s309100) said: “This code implements a tic-tac-toe (Tic-Tac-Toe) game in which two agents play against each other. One of the agents follows a machine learning approach called Q-learning to improve its moves over time. The QLearningAgent class implements an agent that learns using the Q-learning algorithm. The agent tries to maximize its score by choosing actions that lead to a more advantageous state. The State class is a good choice to represent the state of the game. Using objects makes your code more object-oriented and organized. Using defaultdict to handle Q values of states reduces code complexity. It eliminates the need to check for a key before updating it. One really important thing is that Q-learning agent parameters such as epsilon, alpha and gamma are configurable, allowing for greater flexibility in experiment and parameter optimization . Finally we can say that the code is clear, readable and it uses good programming practices that make it easy to use even for an external user who sees it for the first time”. Below is the representation of the class implementing the Q_learning strategy:

```python
class QLearningAgent:
    def __init__(self, epsilon=0.1, alpha=0.1, gamma=0.9):
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma
        self.q_values = defaultdict(float)

    def get_q_value(self, state, action):
        return self.q_values[(state.x,state.o, action)]

    def choose_action(self, state, valid_actions):
        if not valid_actions:
            return None  # No valid actions available
        elif np.random.rand() < self.epsilon:
            return choice(valid_actions)
        else:
            q_values = [self.get_q_value(state, a) for a in valid_actions]
            return valid_actions[np.argmax(q_values)]

    def update_q_value(self, state, action, reward, next_state):
        current_q = self.get_q_value(state, action)
        max_next_q = max([self.get_q_value(next_state, a) for a in range(1, 9 + 1)])
        new_q = (1 - self.alpha) * current_q + self.alpha * (reward + self.gamma * max_next_q)
        self.q_values[(state.x,state.o, action)] = new_q

    def play_against_random_agent(self):
        state = State(frozenset(), frozenset())
        while True:
            # Q-learning agent's turn
            valid_actions_q = list(set(range(1, 9 + 1)) - (state.x.union(state.o)))
            action_q = self.choose_action(state, valid_actions=valid_actions_q)
            if action_q is None:
                return "It's a tie! (No valid actions remaining)"
            state= State(state.x.union({action_q}), state.o)
            if win(state.x):
                return "QLearningAgent wins!"
            elif not valid_actions_q:
                return "It's a tie!"
```

```
            # Random agent's turn
            valid_actions_random = list(set(range(1, 9 + 1)) - (state.x.union(state.o)))
            action_random = random_agent(state)
            state = State(state.x, state.o.union({action_random}))
            if win(state.o):
                return "Random Agent wins!"
            elif not valid_actions_random:
                return "It's a tie!"
# Training with Q-Learning
value_dictionary = defaultdict(float)
hit_state = defaultdict(int)
agent = QLearningAgent(epsilon=0.001, alpha=0.05, gamma=0.09)
q_agent_wins=0
for steps in tqdm(range(10000)):
    trajectory = random_game()
    for i in range(len(trajectory) - 1):
        state = trajectory[i]
        next_state = trajectory[i + 1]
        action = agent.choose_action(state, valid_actions = list(set(range(1, 9 + 1))-(state.x.union(state.o))))  # Choose 
        final_reward = state_value(next_state)
        agent.update_q_value(state, action, final_reward, next_state)

# Display the top 10 states based on their Q-values
top_states = sorted(agent.q_values.items(), key=lambda e: e[1], reverse=True)[:10]
for state_action, q_value in top_states:
    state_x, state_o, action = state_action
    print(f"State X: {state_x}, State O: {state_o}, Q-value: {q_value}")
result = agent.play_against_random_agent()
print(result)
```

My other colleague answered saying: “Dear Salvatore, Thank you for your encouraging feedback on my Tic-Tac-Toe project. I'm glad the code's structure and the QLearningAgent's implementation came across clearly. Your appreciation for the use of defaultdict and the adjustable Q-learning parameters is much appreciated. It's great to hear that the code is user-friendly and approachable. Thanks again for your kind words and helpful insights! Best,Samaneh”

QUIXO

Quixo is an abstract strategy game, suitable for 2-4 players, with a duration of approximately 15 minutes. The game consists of placing 5 cubes in a row, moving the lines of cubes horizontally or vertically. The first player to place 5 cubes in a row wins. Strategies are a fundamental element for this game and the development of the strategies that a player can employ determines how good a player is or not. In my case, I developed two different strategies to understand which was the best: the first is genetic in nature and therefore considers the fundamental genetic processes for selecting the best move in a given state of the game. The second strategy is the one that instead leads to the evaluation of the move according to the MinMax algorithm in which through the in-depth exploration of a move exploration tree, considering a current state, an attempt is made to achieve victory.

MY MIN MAX STRATEGY ON QUIXO

MinMax is a decision algorithm used in games such as chess or tic-tac-toe. Our goal with MinMax is to find the best move for ourselves while benefiting ourselves and minimizing the opponent instead. It is usually a recursive algorithm that develops in depth on the decision tree and considers all the moves that can be made at a given moment by the player who uses it. Each node is therefore a state of the game while the terminal nodes are the outcomes of the game. In my implementation. In my implementation I tried to highlight the main aspects of the algorithm and in particular I tried to use the pruning technique with the alpha and beta parameters to ensure that the algorithm is, as far as possible, optimized. Now I will explain the main parts of my algorithm trying to highlight the most important aspects. I used a class called MinMaxPlayer which extends the properties of the previous Player class. This is initialized with the depth of the search tree and then with the identifier of the player who is using this strategy, since the code must be general and possibly usable both by a player on the first move and on the second move. We use various methods that are intended to analyze the state of our game and the state of the board. These are useful for deciding which strategy to adopt or not and which strategy is convenient to use at a given moment of the game.

The evaluate_board's method calculates a score for the current game configuration. Its purpose is to evaluate the rows, columns and diagonals of the chessboard, assigning scores based on the alignments of zeros and ones. Rewards the presence of 3 or more zeros or ones in a 3x3 submatrix. This is due to the theory of possession of the center of the board in Quixo which means that if a player has more squares inside the internal 3x3 board then he has a greater chance of winning. evaluate_line and legal_piece are two other important methods: evaluate_line is the function that assigns scores based on the count of zeros and ones in a row or column. legal_piece instead verifies whether a piece can be moved from its position. The submatrix_verify method verifies whether a 3x3 submatrix has enough zeros or ones to consider it a strategic move. Then we also have get_legal_moves which generates a list of legal moves considering the legality of the move and the number of zeros or ones in the submatrices. The last supporting method is find_best_move_position which determines the best position for a move by searching among random positions that meet the legality and strategy conditions. The key method is minimax(..) which uses recursion to evaluate all possible moves up to the specified depth. Uses alpha and beta parameters for alpha-beta cut processing, improving the efficiency of the MinMax algorithm. With evaluate_board you evaluate the goodness of the moves. Then with the make_move method the minimax method is called for the entire depth of the tree and the best move is chosen with the find_best_move_position method. Now I post the screen of the minimax method and the make_move method to give a complete view of what I just explained.

```
def make_move(self, game):
    alpha = float('-inf')
    beta = float('inf')
    _, move = self.minimax(game, self.max_depth, alpha, beta, True)
    return self.find_best_move_position(game, move), move
```

```python
def minimax(self, game, depth, alpha, beta, maximizing_player):
  gamecopy = deepcopy(game)
  alpha
  if depth == 0 or game.check_winner() != -1:
    return self.evaluate_board(game), None

  legal_moves = self.get_legal_moves(gamecopy)

  if maximizing_player:
    max_eval = float('-inf')
    best_move = None

    for move in legal_moves:
        game_copy = deepcopy(game)
        position = self.find_best_move_position(game, move)
        game_copy._Game__move(position, move, game_copy.current_player_idx)
        eval, _ = self.minimax(game_copy, depth - 1, alpha, beta, False)

        if eval > max_eval:
            max_eval = eval
            best_move = move

        alpha = max(alpha, eval)
        if alpha >= beta:
            break

    return max_eval, best_move
```

```python
  else:
    min_eval = float('inf')
    best_move = None

    for move in legal_moves:
        game_copy = deepcopy(game)
        position = self.find_best_move_position(game, move)
        game_copy._Game__move(position, move, game_copy.current_player_idx)
        eval, _ = self.minimax(game_copy, depth - 1, alpha, beta, True)

        if eval < min_eval:
            min_eval = eval
            best_move = move

        beta = min(beta, eval)
        if alpha >= beta:
            break

    return min_eval, best_move
```

In the end, after having implemented the method 100 times over and over again, we can say that its effectiveness stands at around 80% of cases of victory versus 20% of defeat, whether you start first or second in developing the moves.

MY GENETIC STRATEGY ON QUIXO

Genetic algorithms are an optimization and search technique inspired by the natural evolutionary process. These algorithms are used to solve complex problems, obtain optimal or approximate solutions through imitation of genetic and evolutionary principles. Each genetic algorithm is composed of different phases and methods whose aim is to generate a population of individuals who can then be used for different purposes. We start by initializing the population with candidate solutions. Subsequently, a fitness test is used to understand how good a candidate solution is or not. Then a selection criterion of individuals is implemented, made according to different criteria, and which in our case is based on the fitness value which, if maximum, leads to a better selection. Then the crossover and mutation phases of the solutions occur, after however a population of parents has been taken which allows the previous population to be initialized. These criteria can be varied and can lead to an improvement or worsening of the solutions. Finally, the process is iterated for a number of generations until the final population is obtained. It is also good practice to take and increase the population of an alpha parameter to allow there to be heterogeneity in the discovery of new solutions. In my solution I tried to implement these maxims to have a strategy that was good for the Quixo game. The population considers the moves and tries to analyze, mutate and generate them, in order to ultimately find the best solution. Below I briefly analyze the purposes of the various methods of the GeneticPlayer class specifically built for the genetic strategy of my implementation. I have also implemented methods whose aim is to analyze the effectiveness of the solutions found with the genetic strategy. After initializing my population with the classic init method of a class we have the method acceptable_move(self, from_pos) which checks if a move is acceptable, i.e. if the starting position is at the edge of the game table. check_condition(self, matrix, row, col) is then the method that checks whether a given condition is satisfied in the game matrix based on the player's ID.

check_condition_zero(self, matrix) checks whether a sequence of zeros is present in each row or column of the matrix, based on the player ID. verify_submatrix(self, matrix) is the method that checks whether the matrix is at least 5x5 and checks the number of elements equal to 1 or 0 in a central 3x3 submatrix. tournament_selection(self, candidates, game) is the function within the GeneticPlayer class that implements tournament selection, typical for evolutionary algorithms, to choose parents for reproduction. evaluate_fitness(self, strategy, game) then evaluates the fitness of a strategy based on the moves made during the game. We tend to penalize strategies that lead to the defeat of the current player and highlight those that block the opponent or that respect the criteria of the functions we have just seen above. Finally, i present the make_move method which is the heart of the solution I proposed:

```
def make_move(self, game):
    offspring = []
    gamecopy = deepcopy(game)
    parents_mutated = []
    ok=False
    lista_finale_strategie=[]
    lista_combinata_fitness_popolazione=[]
    lista_best_strategy=[]
    lista_ulteriore=[]
    parents=[]

    # DOBBIAMO QUA FAR ANDARE AVANTI IL NOSTRO METODO PER UN NUMERO X=? DI GENERAZIONI
    for _ in range(self.generations):
        lista = self.initialize_population()
        lista.sort(key=lambda x: self.evaluate_fitness(x,game),reverse=True)
        # Tournament selection---> applicandola il codice sta diverso tempo a runnare ma arriva a risultato
        for _ in range(len(lista)):
             tournament_candidates = random.sample(lista,5)
             best_strategy_index = self.tournament_selection(tournament_candidates,game)
             parent = tournament_candidates[best_strategy_index]
             parents.append(parent)
        #parents1 = lista[0]
        #parents = [parents1]
```

```
    #Mutation of elements of parents
    if random.random()<0.4:
     for i in range(len(parents)):
        element = parents[i]  #lista interna visto che io considero una lista di liste di azioni che faccio nella board
        mutation_type =random.choice(["replace_move","replace_position","replace_all"])
        if mutation_type == "replace_move":
            #Replace a random move in the strategy with a new random move
            random_index = random.randint(0, len(element) - 1)
            element[random_index] = (random.choice(list(Move)), element[random_index][1])
        elif mutation_type=='replace_position':
            random_index = random.randint(0, len(element) - 1)
            element[random_index] = (element[random_index][0], (random.randint(0, 4), random.randint(0, 4)))
        elif mutation_type=='replace_all':
           random_index = random.randint(0, len(element) - 1)
           element[random_index] = (random.choice(list(Move)), (random.randint(0, 4), random.randint(0, 4)))
        parents_mutated.append(element)
     offspring=parents_mutated
    else:
      #Crossover di tipo one cut crossover
      parents_mutated=parents
      while len(offspring) < len(self.population):
         parent1,parent2 = random.choices(parents_mutated, k=2)
         crossover_index = random.randint(1, min(len(parent1), len(parent2)) - 1)
         child = parent1[:crossover_index] + parent2[crossover_index:]
         offspring.append(child)

    self.population = offspring ## la nuova popolazione composta dai figli
```

```
     ###Incremento il numero di individui nella poplazione di un fattore lambda
     for _ in range(self.lamda):
        strategy = [(random.choice(list(Move)), (random.randint(0, 4), random.randint(0, 4))) for _ in range(30)]
        self.population.append(strategy)

     fitness_scores = [self.evaluate_fitness(strategy, gamecopy) for strategy in self.population]
     for i,strategia in enumerate(self.population):
         lista_combinata_fitness_popolazione.append((fitness_scores[i],strategia))

lista_combinata_fitness_popolazione.sort(key=lambda x:x[0],reverse=True)
i=0
while ok==False and i<=(len(lista_combinata_fitness_popolazione[0][1])-1):
   # Scegli la mossa dalla strategia migliore
   move, position = lista_combinata_fitness_popolazione[0][1][i]
   i=i+1
   # Applica la mossa allo stato attuale del tavolo se è tutto ok
   ok = gamecopy._Game__move(position, move, gamecopy.current_player_idx)
   if gamecopy.check_winner()==self.id:
     return position,move
if ok==False:
   while ok==False:
     move,position=(random.choice(list(Move)), (random.randint(0, 4), random.randint(0, 4)))
     ok=gamecopy._Game__move(position, move, gamecopy.current_player_idx)
     if gamecopy.check_winner()==self.id:
        return position,move
return position, move
```

The make_move method seen above in the screens implements a genetic strategy that interposes mutation and crossover. At the beginning, the tournament selection of the parents who constitute the initial population is applied. After taking the initial parents, either the mutation or the crossover is applied,

and this is decided via the clause that wants a random number to be chosen. Based on the value of this random number the mutation is carried out or not. The mutation, if carried out, can be of 3 types and changes the tuple that constitutes our population of individuals. You can change the moves or the positions of the moves or both parameters of the tuple. The crossover is very simple. It is of the one cut crossover type and when it is carried out it mixes the genetic characteristics of the two randomly chosen parents. After creating the final population, it is incremented with a lambda parameter. This is crucial as it allows me to increase genetic diversity within my population. After finishing the predefined generations, we move on to evaluate the solutions found which are evaluated according to the scores of the fitness function. The strategies are sorted and evaluated and then the latter are sorted from best to worst. Finally, I carried out further checks as sometimes we encountered less than perfect solutions and with these checks we avoid overflows in the code and generate a move that can be more or less good. In my implementation both the number of players and the generations are small, because due to the size of the problem it was difficult to get tangible results for a high number of generations in the code. The lambda parameter is also small, but it is intended to increase genetic diversity. However, I did a number of iterations equal to 10 for the code I presented, and with both that and the one in which there are 100 iterations, the results are good and range between 92 and 99 percent wins for the genetic strategy.