Computational Intelligence

Course Work Report

LUCA STURARO – S320062

Here I reported only what I considered to be the most significant snippets of each code with some accompanying explanations.
The codes can be found in their entirety here:
https://github.com/HerryTheBest/Polito---Computational-Intelligence/tree/main

The code of the first lab was realized together with Gabriele Tomatis (s313848), the rest I made independently but I did consult Gabriele Tomatis and Claudu Tcaciuc (s317661) about doubts and improvements

LAB 1 – SET COVERING USING A*

```
PROBLEM_SIZE = 1000
NUM_SETS = 1000
PROBABILITY = 0.3
SETS = tuple(np.array([random() < PROBABILITY for _ in range(PROBLEM_SIZE)]) for _ in range(NUM_SETS))
State = namedtuple('State', ['taken', 'not_taken'])
```

```
if PROBLEM_SIZE <= 100: # if the problem size is more than 100, visualization gets messy
    print("Sets:")
    for num, element in enumerate(SETS):
        print(f'{num}:\t', end='')
        for value in element:
            if value:
                print('1 ', end='')
            else:
                print('0 ', end='')
        print()

max_solution = [0] * PROBLEM_SIZE
for i in range(PROBLEM_SIZE):
    for element in SETS:
        if element[i]:
            max_solution[i] += element[i]

if 0 in max_solution:
    print('Problem not solvable')
    raise Exception('Problem is not solvable')
else:
    print('Problem solvable')
# print(max_solution)
```

```
Problem solvable
```

```
def goal_check(state):
    return np.all(reduce(
        np.logical_or,
        [SETS[i] for i in state.taken],
        np.array([False for _ in range(PROBLEM_SIZE)]),
    ))


# no longer used
def distance(state):
    return PROBLEM_SIZE - sum(
        reduce(
            np.logical_or,
            [SETS[i] for i in state.taken],
            np.array([False for _ in range(PROBLEM_SIZE)]),
        ))

def weight(state):
    # g is the cost of getting to the current node
    g = len(state.taken)
    # h is the heuristic cost of getting to the goal from the current node
    progress = [False] * PROBLEM_SIZE
    for set in state.taken:
        list_set = SETS[set].tolist()
        for i in range(PROBLEM_SIZE):
            progress[i] = progress[i] or list_set[i]
    # h = PROBLEM_SIZE - sum(progress)                    # basic pessimistic heuristic function (works best)
    # h = h_optimistic_1(PROBLEM_SIZE - sum(progress))    # first optimistic heuristic function
    h = h_optimistic_2(PROBLEM_SIZE - sum(progress))      # second optimistic heuristic function
    return g + h
```

```python
def h_optimistic_1(n_missing_elements):
    optimistic_cost_reduction = 0
    # cost is calculated as the sum of the number of sets that can be taken to complete the problem, weighted by the likelyhood that taking that mnany would be enough
    # the probability is not the actual mathematical probability because I don't know hot to implement that function in a generalized manner (mathematically speaking)
    for i in range(n_missing_elements):
        optimistic_cost_reduction += i * (PROBABILITY ** (n_missing_elements - i))

    # failsafe, if the cost comes up too high, we cap it at the max (else A* breaks down)
    if optimistic_cost_reduction >= n_missing_elements:
        optimistic_cost_reduction = n_missing_elements
    return optimistic_cost_reduction


def h_optimistic_2(n_missing_elements):
    # cost is calculated as the maximum cost (the number of sets) minus the sum of the number of sets we could save (meaning that each set covers more than one point)
    # weighted by the chanche that we could actually save that many
    # the probability is not the actual mathematical probability because I don't know hot to implement that function in a generalized manner (mathematically speaking)
    optimistic_cost_reduction = n_missing_elements
    for i in range(n_missing_elements):
        optimistic_cost_reduction -= i * (PROBABILITY ** i)

    return optimistic_cost_reduction
```

```python
frontier = PriorityQueue()
state = State(set(), set(range(NUM_SETS)))
frontier.put((0, state))

counter = 0
_, current_state = frontier.get()
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((weight(new_state), new_state))
    _, current_state = frontier.get()

print(f'Solved in {counter:,} steps ({len(current_state.taken)} tiles)')
print(f'Solution:     Taken: {current_state.taken}')

# observation:
# with a 30% chance to have each position as true in each element, most set coverings tested here with a fixed number of sets of 1000
# are gonna be completed with 7 to 11 tiles, as that corresponds to a chanche to have each element covered of 91.7% to 98%
#
# the takeaway is that the probability increase given by the number of tiles taken is far higher than the probability decrease given by the size of each tile
```

```
Solved in 10 steps (10 tiles)
Solution:     Taken: {1, 452, 393, 442, 338, 276, 26, 443, 28, 959}
```

Lab 1 required to select the minimum number of sets of N 0/1 elements so that if all selected lists were put through a logical elementwise OR, the result would be an all-1 list.

The problem was treated as a path search, with the goal being to reach the completion state, and A* was used here to determine the best path to the end by assigning weights to each possible move (in the form of selecting one available set)

LAB 3 – NIM GAME USING EVOLUTIONARY STRATEGIES

Lab 2 consisted in developing an agent able to play the game NIM, based on an evolutionary algorithm

2 algorithms were developed

Match 1 - Relative position strategy

This strategy works with 2 evolutionary parameters, the first indicating which row to select and the second indicating how many elements from that row to take (the parameters work based on percentage to adjust as the elements are taken and the rows empty)

Each set of parameters is used to play 100 games (1 cycle), after which a score is assigned to them in the form of the number of victories achieved The parameters giving the best result are saved and used each cycle as the mean of a gaussian mutation to create a new set of parameters

After 100 cycles, the best set is used for a final test over 100 games

```python
def evolutionary_strategy_1(state: Nim, choice_parameters: list):
    '''makes a choice based on parameters that are modified every evolutionary cycle'''
    remaining_rows =  [r for r, c in enumerate(state.rows) if c > 0]
    row_index = ceil(choice_parameters[0] / 100 * len(remaining_rows)) - 1
    row = remaining_rows[row_index]

    num_objects = ceil(choice_parameters[1] / 100 * state.rows[row])
    return Nimply(row, num_objects)


def evolve_1(choice_parameters):
    '''mutates the parameters used by the evolutionary strategy using a gaussian distribution'''
    sigma = 10
    row_parameter = ceil(np.random.normal(choice_parameters[0], sigma, 1)[0])
    if row_parameter > 100:
        correction = row_parameter - 100
        row_parameter = 100 - correction
    if row_parameter < 0:
        row_parameter = - row_parameter

    num_parameter = ceil(np.random.normal(choice_parameters[1], sigma, 1)[0])
    if num_parameter > 100:
        correction = num_parameter - 100
        num_parameter = 100 - correction
    if num_parameter < 0:
        num_parameter = - num_parameter
    return [row_parameter, num_parameter, 0]
```

```
Final cycle using best parameters
     Scores: optimal 51 - evolutionary_strategy 49
\Best ev_p: (60, 95)
```

This first evolution strategy proved ineffective as it assumed that the parameters obtained after the training process could be used to determine an optimal move at any stage of the game.
However the way the parameters determined the move choice (indicating the row to select and the amount to take both as a percentage) resulted too unreliable.

Match 2 - Ideal Nimsum strategy

This strategy works over a set of 100 evolutionary cycles, each composed of 100 games

It uses a Fitness function for choosing moves that evaluates how close each move comes to bringing the game state to a Nim_sum of X, where x is an evolutionary parameter that evolves over time

Each game, for each move, the strategy generates a population of random possible moves and selects the one (or one of the ones usually) that has the best fitness After every cycle, the evolutionary parameter is assigned a score based on the number of victories achieved From the best evolutionary parameter so far, a new one is generated using a gaussian distribution

After 100 cycles, the best set is used for a final test over 100 games

```python
def evolutionary_strategy_2(state: Nim, ideal_nim_sum: list):
    '''Generates some random moves and picks the best one based on parameters that are modified every evolutionary cycle'''
    '''choice parameter represents the ideal nim_sum the move wants to achireve'''
    N_MOVES = 100

    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    num_objects = random.randint(1, state.rows[row])
    chosen_move = [row, num_objects]
    possible_next_state = deepcopy(state)
    possible_next_state.nimming([row, num_objects])
    best_fitness = abs( nim_sum(possible_next_state) - ideal_nim_sum )

    for _ in range(N_MOVES):
        row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
        num_objects = random.randint(1, state.rows[row])
        possible_move = [row, num_objects]

        possible_next_state = deepcopy(state)
        possible_next_state.nimming([row, num_objects])
        fitness = abs( nim_sum(possible_next_state) - ideal_nim_sum )
        if fitness < best_fitness:
            best_fitness = fitness
            chosen_move = possible_move

    return Nimply(chosen_move[0], chosen_move[1])


def evolve_2(choice_parameters, range):
    '''mutates the parameters used by the evolutionary strategy using a gaussian distribution'''
    sigma = 5
    row_parameter = ceil(np.random.normal(choice_parameters, sigma, 1)[0])
    if row_parameter > range:
        row_parameter = 2 * range - row_parameter
    if row_parameter < 0:
        row_parameter = -row_parameter

    return row_parameter
```

```python
def nim_sum(state: Nim) -> int:
    tmp = np.array([tuple(int(x) for x in f"{c:032b}") for c in state.rows])
    xor = tmp.sum(axis=0) % 2
    return int("".join(str(_) for _ in xor), base=2)
```

```
Final cycle using ev_p: 1
      Scores: optimal 0 - evolutionary_strategy 100
```

The second evolutionary strategy proved much more effective as it would select the best move, amongst a number of randomly generated possible moves, using their ‘nim sum’ as a Fitness function, with a set of parameters determining the ideal value of the sum, evolving the parameters that determined of the ideal ‘nim sum’ every 100 games in order to optimize the number of wins.

LAB 9 – BLACK BOX EVOLUTIONARY ALGORITHM

The lab required finding a 1000-loci genomes that best fits a given (unknown) fitness function through an Evolutionary algorithm, using the minimum possible number of fitness calls

The algorithm starts from a randomly generated population of genomes, sorting the population based on fitness, cutting it down to the top 10% and generating new individuals by splicing together the top individuals and then applying some random mutations.
The algorithm checks after every generation whether improvements have been made, stopping early if no progress has been obtained after a number of generations

```python
def ea(fitness, problem_size : int, pop_size : int, ind_size : int):
    num_survivors = int(pop_size / 10)
    population = create_population(num_survivors, ind_size)
    fitted_population = fitness_check([], population, fitness, num_survivors)
    best_fitness = fitted_population[0][1]
    stagnation_counter = GENERATIONS / 20

    for gen in range(GENERATIONS):

        # create new generation
        pop_copy = [ind[0] for ind in fitted_population]
        offsprings = repopulate(pop_copy, pop_size-num_survivors, best_fitness)

        # sort population based on fitness and discard all but the top 10%
        fitted_population = fitness_check(fitted_population, offsprings, fitness, num_survivors)
        new_best_fitness = fitted_population[0][1]

        print(f"\nProblem size {problem_size}, Generation {gen+1} : {new_best_fitness:.2%}", end='')

        # check if new best fitness
        if best_fitness < new_best_fitness:
            print(f" : Improvement", end='')
            best_fitness = new_best_fitness
            stagnation_counter = GENERATIONS / 100
        else:
            # if we are above a certain fitness treshold, check for stagnation
            if best_fitness >= ACCEPTABLE_FITNESS:
                stagnation_counter -= 1
                if stagnation_counter == 0:
                    return population[0], best_fitness

        # if we reached 100% fitness, end
        if best_fitness == 1:
            return population[0], best_fitness


    return population[0], best_fitness
```

```python
def create_population(number_of_indivinuals, individual_size):
    population = []
    for _ in range(number_of_indivinuals):
        population.append(choices([0, 1], k=individual_size))
    return population


# return the population ordered by fitness, the return is a list of tuples [individual, fitness]
def fitness_check(population_with_fitness : list, new_population, fitness_function, num_survivors):

    for individual in new_population:
        population_with_fitness.append([individual, fitness_function(individual)])

    population_with_fitness.sort(key=lambda x: x[1], reverse=True)
    return population_with_fitness[0 : num_survivors]
```

```python
# creates a number of new individuals through crossover and generates random mutations
def repopulate(population, new_generation_size, best_fitness):
    offsprings = []
    mutation_rate = 0.5
    max_mutation_size = 100
    crossover_parents = 10
    crossover_parent_inheritance = int(len(population[0]) / crossover_parents)

    # generate offspring from random sets of parents, each giving 1/10 of the sequence
    for _ in range(new_generation_size):
        parents = choices(population, k=crossover_parents)
        offspring = []
        for i, parent in enumerate(parents):
            x = i * crossover_parent_inheritance
            offspring += parent[x : x + crossover_parent_inheritance]
        offsprings.append(offspring)

    # generate mutations in the newly created offsprings
    for offspring in offsprings:
        if random() < mutation_rate:
            mutation_size = choice(range(max_mutation_size))
            for _ in range(mutation_size):
                g = choice(range(len(offspring)))
                offspring[g] = 1 - offspring[g]

    return offsprings
```

```python
results = []
for problem_size in [1, 2, 5, 10]:
    fitness_function = lab9_lib.make_problem(problem_size)
    individual, individual_fitness = ea(fitness_function, problem_size, POPULATION_SIZE, INDIVIDUAL_SIZE)
    print(f'Final result : {individual_fitness:.2%}')
    # print(f"{''.join(str(i) for i in individual)} : {fitness_function(individual):.2%}")
    calls = fitness_function.calls
    print(f'fitness calls: {calls}')
    results.append([problem_size, individual_fitness, calls])
```

```
Problen size: 1
Best fitness: 96.80%
Fitness calls: 187570

Problen size: 2
Best fitness: 90.20%
Fitness calls: 572050

Problen size: 5
Best fitness: 46.25%
Fitness calls: 900010

Problen size: 10
Best fitness: 32.49%
Fitness calls: 900010
```

Results obtained were good for low problem size; to attempt to improve the results, I created an alternate version of the algorithm where each individual would remember which genes had been mutated during its creation (the idea being that if the individual survived as part of the top 10% its offsprings would try and mutate different genes, to avoid stagnation) Overall, this approach had the opposite result as expected as all problems stopped because of stagnation

```
Problen size: 1
Best fitness: 70.00%
Fitness calls: 500050

Problen size: 2
Best fitness: 66.40%
Fitness calls: 500050

Problen size: 5
Best fitness: 46.77%
Fitness calls: 500050

Problen size: 10
Best fitness: 35.86%
Fitness calls: 500050
```

```python
# creates a number of new individuals through crossover and generates random mutations
def repopulate(population, new_generation_size, best_fitness):
    offsprings = []
    mutation_size = 100
    mutation_rate = 0.5
    crossover_parents = 2
    crossover_parent_contribution = int(len(population[0][0]) / crossover_parents)

    for _ in range(new_generation_size):
        parents = choices(population, k=crossover_parents)
        offspring = []
        mutation_history = []
        for i, parent in enumerate(parents):
            x = i * crossover_parent_contribution
            offspring += parent[0][x : x + crossover_parent_contribution]
            [mutation_history.append(g) for g in range(x, x + crossover_parent_contribution) if g in parent[1]]
        offsprings.append([offspring, mutation_history])
```

```python
# generate mutations in the newly created offsprings
for offspring in offsprings:
    new_mutation_history = []
    if random() < mutation_rate:
        mutation_history = offspring[1]
        mutation_pool = choices(range(len(offspring[0])), k=mutation_size)
        mutating_genes = [g for g in mutation_pool if g not in mutation_history]
        # mutation_size = choice(range(max_mutation_size))
        for g in mutating_genes:
            offspring[0][g] = 1 - offspring[0][g]
            new_mutation_history.append(g)
    offspring[1] = new_mutation_history
return offsprings
```

LAB 10 – TIC-TAC-TOE USING REINFORCEMENT LEARNING

For lab 10, I chose to utilize the Q-learning technique

Q_learning_player

The Reinforced learning player is characterized by 3 factors that help it make choices and adapt

- *learning rate* determines the impact of newly acquired informations compared to the old ones: at 0 the system doesn't learn anything, at 1 the system only considers the last acquired information
- *discount factor* determines the impact of future rewards: at 0 the system only considers immediate reward (greedy), as the value increases the system considers more and more future rewards
- *exploration_rate* determines the chance the system will explore the environment rather than exploiting it (acts as a percentage chance)

The Q_learning_player decides every move wether to explore (make a random move) or exploit (make a move it knows the reward for). For exploiting it keeps a dictionary of all encountered game states, and to each state corresponds a list possible moves from that state and a list containing the associated rewards

```
class random_player():
    def make_move(self, game_state: State):
        return random.choice(available_moves(game_state))

    def name(self):
        return 'Random_player'


class Q_learning_player():
    def __init__(self, learning_rate, discount_factor, exploration_rate, role):
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_rate = exploration_rate
        self.learning_dictionary = defaultdict(int)
        self.role = role
        # because the rewards are set up to be positive if x wins and negative if o wins,
        # the player must know if it's playing as x or o to know if it must maximize on minimize the rewards

    def make_move(self, game_state: State):
        hashable_state = (frozenset(game_state.x), frozenset(game_state.o))
        if random.uniform(0, 1) > self.exploration_rate and self.learning_dictionary[hashable_state] != 0:
            # exploit, aka choose a move based on what it has learned so far
            moves, rewards = self.learning_dictionary[hashable_state] # get a list of possible moves from the current state
            if self.role == 'x':
                best_move_index = rewards.index(max(rewards))
            else:
                best_move_index = rewards.index(min(rewards))
            return moves[best_move_index]
        else:
            # explore, aka make a new random move outside of what it has learned so far
            return random.choice(available_moves(game_state))
```

```
def update(self, game_log, reward):
    for game_state, move, next_state in game_log: # for each move it took during the game

        hashable_state = (frozenset(game_state.x), frozenset(game_state.o))
        hashable_next_state = (frozenset(next_state.x), frozenset(next_state.o))
        existing_state = self.learning_dictionary[hashable_state]
        if existing_state == 0:
            self.learning_dictionary[hashable_state] = [[], []]

        previous_moves: list = self.learning_dictionary[hashable_state][0]
        if move in previous_moves:
            i = previous_moves.index(move)
            current_reward = self.learning_dictionary[hashable_state][1][i]
            if self.learning_dictionary[hashable_next_state] != 0:
                optimal_next_reward = max(self.learning_dictionary[hashable_next_state][1])
            else:
                optimal_next_reward = 0
            # self.learning_dictionary[hashable_state][1][i] = previous_reward * self.learning_rate + (reward - previous_reward) * self.discount_factor
            self.learning_dictionary[hashable_state][1][i] = current_reward * (1 - self.learning_rate) + self.learning_rate * (reward + self.discount_factor * optimal_next_reward)
        else:
            move_reward = reward * self.discount_factor
            self.learning_dictionary[hashable_state][0].append(move)
            self.learning_dictionary[hashable_state][1].append(move_reward)

def name(self):
    return 'Q_learning_player'

def data(self):
    return self.role, self.learning_rate, self.discount_factor, self.exploration_rate
```

For training we play the game and every time the learning system makes a move it saves the state and the move it took in the game_log for the learning process After every game, if the player won or lost it gains a positive or negative reward associated with the moves it took at a given state (in case of a draw it learns nothing)

```python
def training(players, epochs, learning_player):
    if learning_player == 'x':
        learning_player = 0
    if learning_player == 'o':
        learning_player = 1

    for _ in range(epochs):
        game_over = False
        game_log = []
        game_state = State(set(), set())
        player_turn = 0
        while not game_over:

            move = players[player_turn].make_move(game_state)
            if learning_player == player_turn:
                current_state = deepcopy(game_state)
            if player_turn == 0:
                game_state.x.add(move)
            else:
                game_state.o.add(move)
            if learning_player == player_turn:
                game_log.append((current_state, move, deepcopy(game_state)))

            if winning_state(game_state) or len(available_moves(game_state)) == 0:
                game_over = True

            player_turn = 1 - player_turn

        if winning_state(game_state):
            players[learning_player].update(game_log, winning_state(game_state))
```

I ran some tests to find the best combination of hyperparameters

```python
results_x = []
results_o = []
#lr, df, er = 0.1, 0.9, 0.1
for lr in [0.1, 0.5, 0.9]:
    for df in [0.1, 0.5, 0.9]:
        for er in [0.1, 0.5, 0.9]:
            for role in ['x', 'o']:
                print(f'learning_rate, discount_factor, exploration_rate = {lr}, {df}, {er}\nplaying as = {role}')
                player_1 = Q_learning_player(lr, df, er, role)
                player_2 = random_player()
                train_epochs = 100_000
                test_epochs = 1_000
                if role == 'x':
                    players = [player_1, player_2]
                elif role == 'o':
                    players = [player_2, player_1]

                training(players, train_epochs, role)

                win_rate, data = testing(players, test_epochs, role)
                if data[0] == 'x':
                    results_x.append([win_rate, data])
                if data[0] == 'o':
                    results_o.append([win_rate, data])
                print('\n')
```

The best results obtained were

```
learning_rate, discount_factor, exploration_rate = 0.9, 0.9, 0.1
playing as = x
Final results out of 100000 games:
        Q_learning_player win rate: 87.6 %
        Random_player win rate 2.56 %
        Draw rate 9.84 %
```

QUIXO

For the exam code the request was to produce an agent able to play a game of Quixo.

For this I developed 3 agents, each better than the last.

1ST AGENT - Q LEARNING

```python
class Qbot(Player):
    def __init__(self, training_epochs, role, exploration_logic=RandomPlayer(), training_opponent=RandomPlayer(),
                 learning_rate=0.9, discount_factor=0.9, exploration_rate=0.1):
        super().__init__()
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_rate = exploration_rate
        self.learning_dictionary = defaultdict(int)
        self.game_log = list()
        self.training_epochs = training_epochs
        self.role = role
        self.exploration_logic = exploration_logic
        self.training_opponent = training_opponent
        self.training = True
        self.unencountered_states = 0

        self.train(training_epochs)


    def train(self, epochs : int):
        print(f'Training Qbot over {epochs} epochs')
        for _ in tqdm(range(epochs)):
            g = Game()
            if self.role == 0:
                winner = g.play(self, self.training_opponent)
            elif self.role == 1:
                winner = g.play(self.training_opponent, self)
            self.update(winner)

        self.training = False
        self.exploration_rate = 0
```

```python
def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
    hashable_state = tuple(map(tuple, game.get_board()))

    if random.uniform(0, 1) > self.exploration_rate and self.learning_dictionary[hashable_state] != 0:
        # exploit, aka choose a move based on what it has learned so far
        moves, rewards = self.learning_dictionary[hashable_state] # get a list of possible moves from the current state
        best_move_index = rewards.index(max(rewards))
        if max(rewards) > 0:
            from_pos, move = moves[best_move_index]
        else:
            #no memorized option has lead to victory, explore a new one
            from_pos, move = self.exploration_logic.make_move(game)
    else:
        # explore, aka make a new random move outside of what it has learned so far
        from_pos, move = self.exploration_logic.make_move(game)
        if not self.train:
            self.unencountered_states += 1
            print(self.unencountered_states)

    if self.training:
        self.game_log.append((deepcopy(game.get_board()), (from_pos, move)))
    return from_pos, move
```

```python
def update(self, victory):
    reward = 1 if (victory == self.role) else -1

    for board, move in self.game_log: # for each move taken during the game
        hashable_state = tuple(map(tuple, board))

        existing_state = self.learning_dictionary[hashable_state]
        if existing_state == 0:
            self.learning_dictionary[hashable_state] = [[], []]

        previous_moves: list = self.learning_dictionary[hashable_state][0]
        if move in previous_moves:
            i = previous_moves.index(move)
            current_reward = self.learning_dictionary[hashable_state][1][i]
            optimal_next_reward = 0

            #get the board state after this move
            next_opponent_board = simulate_move(board, self.role, move[0], move[1])
            #simulate all possible moves the opponent might make
            for opponent_move in get_all_valid_moves(next_opponent_board, (1 - self.role)):
                next_board = simulate_move(next_opponent_board, (self.role - 1), opponent_move[0], opponent_move[1])
                hashable_next_state = tuple(map(tuple, next_board))
                if self.learning_dictionary[hashable_next_state] != 0:
                    if max(self.learning_dictionary[hashable_next_state][1]) > optimal_next_reward:
                        optimal_next_reward = max(self.learning_dictionary[hashable_next_state][1])

            # self.learning_dictionary[hashable_state][1][i] = previous_reward * self.learning_rate + (reward - previous_reward) * self.discount_factor
            self.learning_dictionary[hashable_state][1][i] = current_reward * (1 - self.learning_rate) + self.learning_rate * (reward + self.discount_factor * optimal_next_reward)
        else:
            move_reward = reward * self.discount_factor
            self.learning_dictionary[hashable_state][0].append(move)
            self.learning_dictionary[hashable_state][1].append(move_reward)

    self.game_log = list()
```

My first attempt was to utilize the Q learning algorithm developed for lab 10 and adapt it for quixo, various iteration lead me to use a variable exploration logic accepting any other player (by default a random player) to take decisions for exploration (mostly used during training, while the internal dictionary is being created)

The performance of this 'Qbot' player was mixed (these results are based on up to 100_000 training epochs):
- when tested against a random player while using random exploration logic, it would result in a 50% win rate

- when tested against a random player while using a more advanced exploration logic, it would consistently win

- when tested against a more intelligent player (Gbot or SimpleBot) it would consistently loose, regardless of internal logic

My conclusions are that the algorithm would probably need a very high number of training cycles to develop a robust enough internal dictionary to obtain good results, but for the exam I considered it to be too computationally expensive to produce good results.

2ND AGENT - GENETIC ALGORITHM

```python
class Gbot(Player):
    def __init__(self):
        super().__init__()
        self.generations = 10
        self.population_size = 20
        self.survivors = int(0.2 * self.population_size)
        self.mutation_rate = 0.5


    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        population = self.generate_population(game)
        fitted_population = list()
        for generation in range(self.generations):
            #fitness
            new_fitted_population = self.fitness(population, game)
            fitted_population += new_fitted_population

            #selection
            fitted_population.sort(key=lambda x: x[1], reverse=True)
            top_population = fitted_population[0 : self.survivors]
            if generation+1 < self.generations: #don't reproduce on the last generation
                #reproduction & mutation
                population = self.reproduction(top_population)

        #select top
        from_pos, move = top_population[0][0]
        return from_pos, move
```

```
def generate_population(self, game : Game):
    population = list()
    for _ in range(self.population_size):
        from_pos, move = valid_random_move(game)
        population.append((from_pos, move))
    return population
```

```
def fitness(self, population : list(), game : Game):
    fitted_population = list()
    player = game.get_current_player()
    for individual in population:
        #simulate move
        move_result = numpy.array(simulate_move(game.get_board(), player, individual[0], individual[1]))
        fitness_score = 0

        #fitness1: promote longer sequences, not necessarely continuous
        for row in move_result:
            fitness_score += numpy.sum(row == player)**2
        for col in move_result.T:
            fitness_score += numpy.sum(col == player)**2
        main_diagonal = move_result.diagonal()
        fitness_score += numpy.sum(main_diagonal == player)**2
        secondary_diagonal = numpy.fliplr(move_result).diagonal()
        fitness_score += numpy.sum(secondary_diagonal == player)**2


        #fitness2: prefer to remove opponent pieces, not own pieces
        if game.get_board()[individual[0][1]][individual[0][0]] == game.get_current_player():
            fitness_score -= 1
        elif game.get_board()[individual[0][1]][individual[0][0]] == (1 - game.get_current_player()):
            fitness_score += 1


        fitted_population.append((individual, fitness_score))
    return fitted_population
```

```
def reproduction(self, fitted_population):
    offsprings = list()
    functional_offsprings = list()
    num_offsprings = self.population_size - len(fitted_population)

    #crossover
    for _ in range(num_offsprings):
        parent1, parent2 = random.sample(fitted_population, 2)
        offsprings.append((parent1[0][0], parent2[0][1]))
```

```
    #discard nonfunctional offsprings
    for offspring in offsprings:
        acceptable = True
        if offspring[0][0] == 0 and offspring[1] == Move.LEFT:
            acceptable = False
        if offspring[0][0] == 4 and offspring[1] == Move.RIGHT:
            acceptable = False
        if offspring[0][1] == 0 and offspring[1] == Move.TOP:
            acceptable = False
        if offspring[0][1] == 4 and offspring[1] == Move.BOTTOM:
            acceptable = False
        if acceptable:
            functional_offsprings.append(offspring)


    return functional_offsprings
```

For my second attempt, I adapted the lab 9 algorithm to generate a number of random moves, weight them based on a Fitness function and discard all but the top % of solution, before generating new moves by mixing the ones from the previous generation (initially I also applied mutations, but I realized that most results I would obtain through mutation I already obtained through crossover, this is the realization that led me to develop the 3rd agent).

This agent obtained much better results due to the fitness function which promotes moves that lead to longer sequences.

3RD AGENT – SIMPLE FITNESS

```python
for individual in population:
    #simulate move
    move_result = numpy.array(simulate_move(game.get_board(), player, individual[0], individual[1]))
    fitness_score = 0
    lines = [row for row in move_result]
    lines += [col for col in move_result.T]
    lines += [move_result.diagonal(), numpy.fliplr(move_result).diagonal()]

    #fitness1: promote longer sequences, not necessarely continuous
    for line in lines:
        fitness_score += numpy.sum(line == player)**2


    #fitness2: prefer to remove opponent pieces, not own pieces
    if game.get_board()[individual[0][1]][individual[0][0]] == game.get_current_player():
        fitness_score -= 1
    elif game.get_board()[individual[0][1]][individual[0][0]] == (1 - game.get_current_player()):
        fitness_score += 1

    #fitness3: don't make a move that makes the adversary win
    losing_move = False
    for line in lines:
        if numpy.sum(line == adversary) == 5:
            losing_move = True
    if losing_move:
        fitness_score -= 1000

    fitted_population.append((individual, fitness_score))
```

For this last agent, I realized that the Genetic algorithm was a waste of resources for this problem, as the game only has up to 44 possible moves a player can make during their turn, thus it was faster to generate them all immediately.
The moves are still ranked based on a fitness function (which is an improved version of the one from the 2nd agent)
Overall, the main improvement of the 3rd agent over the 2nd is in terms of efficiency rather then performance.

LAB REVIEWS

I'm not sure if they should be reported but just in case (I'm not reporting here their entire code for a matter of logistics, but I'll leave links to the relevant codes)

LAB 2

HTTPS://GITHUB.COM/YALDAMOBARGHA/COMPUTATIONAL-INTELLIGENCE/TREE/MAIN/LABS/2

Intro

As a general note, the code colud use some more comments to clarify the purpose of some of the functions (not a big deal since the code is farely short, but always nice).

Output feedback

I think it might have been useful to have the ES function output the various states of the Evolution (things like best individual and win rate) at every cycle, both for feedback during developement and for future observers to get a better visual rapresentation of the problem and its evolution.

ESstrategy function

I belive the inclusion of some more possibile moves for the problem to choose from would be useful, giving more freedom of action and possibly leading to better results. As it stands now, I belive the possible moves the algorithm chan make are farely restrictive.

Parameter variation

Another possible improvement would be to run the problem multiple times with varying mutation parameters (mu and lambda), seeing if this couldn't lead to better results.

Conclusion

I hope you can find these comments useful

HTTPS://GITHUB.COM/RIDEN15/COMPUTATIONAL-INTELLIGENCE/TREE/MASTER/LAB%202

Intro

As a general note, the code is well laid out and the comments make it quite straight forward to follow, only the make_move function in the NimAgent class could have used some more clarification for how the choice is actually made.

Output

While it clearly would not have been feasible to display the entire population at the end of every cycle, maybe you could have shown the best individual, to show the progressive evolution, together with the win rate (not necessary but it would have given a better visual rapresentation in my opinion).

make_move function

I only have one real issue with the code which stems from the way individuals are made up of probabilities of operating on each row, but no parameter exists to decide how many elements to take from each row (its determined as a function of the row itself).

Because of this, the algorithm looses a degree of freedom, where it may be optimal to take some elements from a row and then move on to the next one.
This cannot be done as the individual will always choose the row with the highest probability, regardless of how many elements are left in it, thus it will always prioritize to drain the preferred row before moving to the next.

I belive adding some parameter to each individual to decide the number of elements to take independently of the row might lead to better results.

Conclusion

I hope you'll find these comments helpful.

LAB 9

HTTPS://GITHUB.COM/FILIPPOBERTOLOTTI0/MY_COMPUTATIONAL_INTELLIGENCE_317811/TREE/MAIN/LABS/L09

Hey filippo, first off your readme notes make the code quite straight forward to follow and review (I apreciate that).

Looking over your code, I find it is a quite straightforward and effective implementation of an evolutionary algorytm, I can suggest a couple of additions/changes to improve your results:
First you could try to implement a flexible crossover function to create new individuals from recombining a variable number of parents (this doesn't necessarely improve the solution, but it can create interesting results I think are worth considering)
Second, while you might have already done so while developing the code, it could be useful to see the results of running the code with different parameters (changing things like Population size, Standard deviation for the mutation function and the no_improvement counter) to see how the results change with each set of parameters.

Overall the code is good and performs well, so these are just some suggestions for things to change to improve it.
I hope you found this review helpful.

HTTPS://GITHUB.COM/ALLEGRAROBERTO/COMPUTATIONAL-INTELLIGENCE/TREE/MAIN/LAB3

Hey Roberto, first off I have to say the Readme notes and explanations make the code very easy to follow and analyze.

I think its great you run the evolution code multiple times with different parameters to determine the most impactful ones, one addition I can suggest is to implement a variable parent number, meaning creating new individuals by splicing together genome material from more than 2 parents (which represents an additional parameter to tweak to look for the best solution).

Looking over your code I can think of 2 possible improvements:
First it would be some kind of stagnation control to stop the evolution process when it reached a point of stagnation and its not making meaningful progress towards a better fitness (although looking at your results I'm not sure if it would have been useful in your case).
Second, if I'm reading your code correctly you recalculate the fitness value of your population at the start of each generation to sort them, this is good for the new individuals but you could save on recalculating the fitness of the elite of the previous generation by saving the fitness when you calculate it each time (a way to do this would be to transform your population into a list of tuples containing Individual and fitness).

I hope this review can help, good job overall.

LAB 10

HTTPS://GITHUB.COM/MATTEMARTINI/COMPUTATIONAL-INTELLIGENCE/TREE/MAIN/LAB10

Hey matteo, first off I like your code, its well organized and easy to read with your comments.

Honestly the results of your learning agent speak for themselves, so there isn't much to say on how to improve on your code.
One of the only things I can think of is you could try and experiment with different values as rewards for your q_learning agent to see if they can slightly improve your results (your alpha and gamma parameters, seen as you already experiment with epsilon).

Beside that, you could try and adapt the system to play as Second as well as First and see if it does need some changes or optimizations when playing under different starting conditions.

Really, there isn't much to say as you already did a great job. Good luck for the exam :)

HTTPS://GITHUB.COM/KINEPO/CI-POLITO

Hey Alexandre, as a first note I want to say that your code is somewhat disorganized, it could use some more comments and cutting down on a few needless repetitions of code to make it more intuitive to follow.

On the code itself (mind you, this is what I could understand from reading through your code myself so if I misread I apologize) it seems to me you don't associate the reward of each move to the state, meaning the system is only learning what the best moves are in general terms, regardless of what the board looks like right now; if I'm correct you should try to implement some such form of learning, associating state and move reward, else there is no long term learning.

Additionally you should have your learning player train and play as second player as well as first to try and see if there are telling changes between the two or if you should optimize it differently for the different start.

I hope this can help you for the exam ahead.