Activity Report - Riccardo Cardona s319441

Table of Contents

1. Lab 01 — Set Covering

Description of the problem

Given the number of sets `NUM_SETS` and the number of elements inside each set `PROBLEM_SIZE`, determine, if possible, the collection of sets through which all the elements are available.

A state is made up of the sets of items that I take and the sets that I don't take.

- State=({1,3,5}, {0,2,4,6,7}) -> I'm taking the second array of items from `SETS`, the fourth and the sixth.
- The quality of a solution (the chosen state) is given by the smallest number of taken sets to get all the elements

Objectives

1 - Implementation of Search Algorithms The goal of the project is to find an algorithm that can efficiently solve the Set Covering Problem.

2 - Build a Heuristic Function H for A star The focus of the laboratory is on the A* search algorithm and therefore on finding a Heuristic Function H. To be sure to find the optimal solution, the heuristic function must be:

- Admissible: It never overestimates the cost of reaching the goal, so that $H(n) \leq H^*(n)$
- Consistent: It satisfies the triangle inequality, so that $H(n) \leq c(n, a, n') + H(n')$

Code

Imported Libraries

```
from random import random, choice, randint
from functools import reduce
from math import ceil
from collections import namedtuple
from queue import PriorityQueue, SimpleQueue, LifoQueue
import numpy as np
from tqdm.auto import tqdm
from copy import copy
```

Problem instance

We implement the sets as an array of arrays where one element has a 20% chance of being true. A set indicates which item is inside the set and which is not, if an element of the set has a value of true it means that it is present otherwise not

```
PROBLEM_SIZE = 50
NUM_SETS = 100
```

```
SETS = tuple(np.array([random() < .2 for _ in range(PROBLEM_SIZE)]) for _ in
range(NUM_SETS))
State = namedtuple('State', ['taken', 'not_taken'])
```

Functions for algorithms

- `goal_check` is the function that checks whether the state given in input is a solution, that is if all the elements are present within the states taken. To do that, we need to do an or between all the elements of all the sets of the state to see if a certain element is present among all the sets or not (true is present, false is not). With `all`, we check if all the elements are present, that is, if the `reduce` gives me an array of True which means that all the elements are present among the sets taken
- `distance` is the function that calculate the distance from the input state to the final goal (for a greedy search), in short, it returns how many elements that need to be taken are missing
- `covered` is the function that returns what are the covered elements in the taken sets

```
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)])))


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

```
assert goal_check(State(set(range(NUM_SETS)), set())), "Problem not solvable"
```

Hill Climbing

The `current_state` is an array of boolean where true indicates if the state contains that particular set, false if it doesn't contain it. We have to initialize it to a random possible solution.

- In `tweak` function we swap one of the set randomly, if it was taken we change it into not taken and vice versa.
- The `fintess1` function returns a boolean that indicates if the state given as input is a solution and the negative cost that is the number of taken sets as negative. That's because when we check if we want to swap the `current_state` we first check if the state is a solution (False < True) and then we want to take the solution with the smallest number of taken sets. The problem with this function is that if we start with a `current_state` that is not a solution, the algorithm will go to another invalid solution with just less taken sets. The more he takes away the set, the more difficult it is to move towards the solution.

- the `fitness2` function solves the problem of the previous function using the number of covered elements as the first object of the tuple instead of the boolean that indicate if the `current_state` is a solution.

Speaking in a more general way, the `fitness` function is as if it gives a rank to the current state and the tweak function is the one that allows us to move between different solutions.

```
def fitness1(state):
    cost = sum(state)
    valid = np.all(
        reduce(
            np.logical_or,
            [SETS[i] for i, t in enumerate(state) if t],
            np.array([False for _ in range(PROBLEM_SIZE)]),
        )
    )
    return valid, -cost


def fitness2(state):
    cost = sum(state)
    valid = np.sum(
        reduce(
            np.logical_or,
            [SETS[i] for i, t in enumerate(state) if t],
            np.array([False for _ in range(PROBLEM_SIZE)]),
        )
    )
    return valid, -cost

def tweak(state):
    new_state = copy(state)
    index = randint(0, PROBLEM_SIZE - 1) # pick a random index
    new_state[index] = not new_state[index] # swap
    return new_state

fitness = fitness2
```

```
current_state = [choice([True, False]) for _ in range(NUM_SETS)]
print(fitness(current_state))

for step in range(100):
    new_state = tweak(current_state)
    if fitness(new_state) > fitness(current_state): # with fitness2 we have to use
>, with fitness1 >=
        current_state = new_state
        print(fitness(current_state))
```

Breadth-first search algorithm

Implementation using breadth-first search algorithm using a Simple Queue which would be a FIFO If we used a LifoQueue it would become a depth-first search

```
frontier = SimpleQueue()
frontier.put(State(set(), set(range(NUM_SETS))))

counter = 0
current_state = frontier.get()
with tqdm(total=None) as pbar:
    while not goal_check(current_state):
        counter += 1
        for action in current_state[1]:
            new_state = State(current_state.taken ^ {action},
                              current_state.not_taken ^ {action})  # {1,2,3} ^ {2}
= {1,3}  {1,3} ^ {2} = {1,2,3}
            frontier.put(new_state)
        current_state = frontier.get()
        pbar.update(1)

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

Dijkstra search algorithm

The cost used for the PriorQueue is the number of elements in a set i.e., those taken

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

counter = 0
_, current_state = frontier.get()
with tqdm(total=None) as pbar:
    while not goal_check(current_state):
        counter += 1
        for action in current_state[1]:
            new_state = State(current_state.taken ^ {action},
                              current_state.not_taken ^ {action})  # {1,2,3} ^ {2}
= {1,3}  {1,3} ^ {2} = {1,2,3}
            frontier.put((len(new_state.taken), new_state))
        _, current_state = frontier.get()
        pbar.update(1)

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

Greedy search algorithm

```
frontier = PriorityQueue()
state = State(set(), set(range(NUM_SETS)))
```

```
frontier.put((distance(state), state))

counter = 0
_, current_state = frontier.get()
with tqdm(total=None) as pbar:
    while not goal_check(current_state):
        counter += 1
        for action in current_state[1]:
            new_state = State(current_state.taken ^ {action},
                              current_state.not_taken ^ {action})  # {1,2,3} ^ {2}
= {1,3}  {1,3} ^ {2} = {1,2,3}
            frontier.put((distance(new_state), new_state))
        _, current_state = frontier.get()
        pbar.update(1)

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

A* search algorithm

The `distance` function is not usable as heuristic for A* because it is pessimistic. If it returns that the distance from the goal is three, I'm sure that I can cover them with three sets, but it is the most pessimistic solution and the heuristic for A* has to be optimistic. These are the functions that I tried ordered from the least performing to the most performing:

- `h1`: if the largest set (the one with most elements) covers two elements and if `missing_size` is three (the number of elements there are missing), then I will need at least two sets (3/2).
- `h2`: the same as h1 but for the largest set I take into account only the items that have not yet been covered. `largest_remaining_set_size` it will therefore be the maximum number of uncovered elements present in a set.
- `h3`: instead of considering only the set with the maximum size, we use the size of all uncovered sets and sort it taking into account, for the dimension of them, only the items that have not yet been covered. So for example, if I am missing four elements and in the list I have five sets with ordered dimensions (`candidates` = [2, 1, 1, 1, 1]), it means that I will need at least three sets (2+1+1 = 4)

```
def h1(state):
    largest_set_size = max(sum(s) for s in SETS)
    missing_size = PROBLEM_SIZE - sum(covered(state))
    optimistic_estimate = ceil(missing_size / largest_set_size)
    return optimistic_estimate


def h2(state):
    already_covered = covered(state)
    if np.all(already_covered):
        return 0
    largest_remaining_set_size = max(sum(np.logical_and(s,
np.logical_not(already_covered))) for s in SETS)
    missing_size = PROBLEM_SIZE - sum(already_covered)
    optimistic_estimate = ceil(missing_size / largest_remaining_set_size)
    return optimistic_estimate
```

```
def h3(state):
    already_covered = covered(state)
    if np.all(already_covered):
        return 0
    missing_size = PROBLEM_SIZE - sum(already_covered)
    candidates = sorted((sum(np.logical_and(s, np.logical_not(already_covered)))
for s in SETS), reverse=True)
    taken = 1
    while sum(candidates[:taken]) < missing_size:
        taken += 1
    return taken


def A_star(state):
    return len(state.taken) + h1(state)
```

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

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

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

2. Lab 02 — Nim Game

Nim is a game in which two players take turns removing objects from distinct rows. On each turn, a player must remove at least one object, and may remove any number of objects provided they all come from the same row.

In our type of Nim, the player who wins is the one that has to take an object and only one is left. In this way, a player needs to play to force the opponent to take the last object.

Evolved strategy

The main idea of our solution is to use some kind of probability to choose the best move.

- An individual is composed of N probabilities, where N is the number of rows in the nim game.
- The fitness function is out `evaluate_population` function. It returns the number of wins of the current population against the optimal strategy.
- With the `make_move` function we compute for each row the possible move that is (row,n object), where n object will be a number between one and the actual number of objects in that row. Once we have the moves, we choose the one which row corresponds to the highest probability of the individual.

Example: n = 5, individual = [0.1, 0.5, 0.4, 0.6, 0.9], moves = [(0,1), (1,1), (2,1), (3,4), (4,6)] The individual tells us that the move (4,6) is the one with the highest prob.

Collaboration

For this laboratory, I collaborate with Nicholas Berardo s319441

Code

Write agents able to play *Nim*, with an arbitrary number of rows and an upper bound k on the number of objects that can be removed in a turn (a.k.a., *subtraction game*).

The goal of the game is to **avoid** taking the last object.

- Task2.1: An agent using fixed rules based on *nim-sum* (i.e., an *expert system*)
- Task2.2: An agent using evolved rules using ES

```
import logging
from pprint import pprint, pformat
from collections import namedtuple
import random
from copy import deepcopy
```

The *Nim* and *Nimply* classes

```
Nimply = namedtuple("Nimply", "row, num_objects")
```

```
class Nim:
    def __init__(self, num_rows: int, k: int = None) -> None:
        self.initial_rows = [i * 2 + 1 for i in range(num_rows)]
        self._rows = self.initial_rows.copy()
        self._k = k

    def __bool__(self):
        return sum(self._rows) > 0

    def __str__(self):
        return "<" + " ".join(str(_) for _ in self._rows) + ">"

    @property
    def rows(self) -> tuple:
        return tuple(self._rows)

    def nimming(self, ply: Nimply) -> None:
        row, num_objects = ply
        assert self._rows[row] >= num_objects
        assert self._k is None or num_objects <= self._k
        self._rows[row] -= num_objects

    def reset(self):
        self._rows = self.initial_rows.copy()
```

Sample (and silly) startegies

```
def pure_random(state: Nim) -> Nimply:
    """A completely random move"""
    #takes a row r if the value of that row c is > 0
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    #take a random value from 1 to the value of the row
    num_objects = random.randint(1, state.rows[row])
    #subtract it
    return Nimply(row, num_objects)
```

```
def gabriele(state: Nim) -> Nimply:
    """Pick always the maximum possible number of the lowest row"""
    possible_moves = [(r, o) for r, c in enumerate(state.rows) for o in range(1, c
+ 1)]
    return Nimply(*max(possible_moves, key=lambda m: (-m[0], m[1])))
```

```
import numpy as np

#nim_sum mi restituisce XOR in binario delle varie righe che ci sono.
#Se ho 1 e 3, quindi 2 righe:
# 1 -> 01
# 3 -> 11
# 1 XOR 3 = 10 -> 2
def nim_sum(state: Nim) -> int:
    # {c:032b} mi trasforma il valore di state.rows in binario su 32 bit
    tmp = np.array([tuple(int(x) for x in f"{c:032b}") for c in state.rows])
    #Qua è come se facessi XOR, la posso fare con una somma
    xor = tmp.sum(axis=0) % 2
    return int("".join(str(_) for _ in xor), base=2)


def analize(raw: Nim) -> dict:
    cooked = dict()
    cooked["possible_moves"] = dict()
    for ply in (Nimply(r, o) for r, c in enumerate(raw.rows) for o in range(1, c +
1)):
        tmp = deepcopy(raw)
        tmp.nimming(ply)
        cooked["possible_moves"][ply] = nim_sum(tmp)
    return cooked


def optimal(state: Nim) -> Nimply:
    analysis = analize(state)
    logging.debug(f"analysis:\n{pformat(analysis)}")
    spicy_moves = [ply for ply, ns in analysis["possible_moves"].items() if ns !=
0]
    if not spicy_moves:
        spicy_moves = list(analysis["possible_moves"].keys())
    ply = random.choice(spicy_moves)
    return ply
```

Adaptive

- `NimAgent` is the class that represent an individual. In our problem, the individual is an array of number between 0 and 1 where each number indicates the probability of making a move in the nim row equivalent to the index of the single number in the array.
- `make_move`: is the function that, given the `state` of the game, first create a possible move for each row of the game board and then take the best one using the individual's odds.

```
class NimAgent:
    def __init__(self, individual):
        self.individual = individual

    def __str__(self) -> str:
```

```
        return str(self.individual)

    def make_move(self, state) -> Nimply:
        #(row,obj)
        moves = [
            (row, min(state.rows[row], max(1, int(self.individual[row] *
state.rows[row]))))
            for row in range(len(state.rows)) if state.rows[row] > 0]

        # Select the move with the highest preference, the highest preference is
the one with the highest genome value.
        # Self.individual[x[0]] indicates the probability associated with the row
of the game board x[0] that we want to modify
        chosen_move = max(moves, key=lambda x: self.individual[x[0]])

        # Return the move
        return Nimply(*chosen_move)
```

Evolutionary algorithm functions

- `reproduce`: function that creates new individuals with `mutation` and `crossover` based on the `mutation_probability`.
- `mutation`: function that modifies every probability in the individual selected by the `reproduce` function. It modifies the values with a Gaussian distribution making sure that the genome is within 0 and 1 after the mutation.
- `crossover`: It creates a child genome by randomly selecting each gene from either parent with equal probability.
- `replacement`: This function receives as parameters the population, the new individuals crated by the `reproduce` function and the `fitness_score` that indicates the times one individual wins against the optimal algorithm. In this function we sort the population based on the `fitness_score` and replace the worst individuals. In the end, half of the population will be changed and the other half will be preserved.

```
def reproduce(selected_agents, mutation_rate, mutation_probability):
    new_population = []

    while len(new_population) < len(selected_agents):
        if random.random() < mutation_probability:
            parent1 = random.sample(selected_agents, 1)[0]
            child_genome = mutation(parent1.individual, mutation_rate)
        else:
            parent1, parent2 = random.sample(selected_agents, 2)
            child_genome = crossover(parent1.individual, parent2.individual)
        new_population.append(NimAgent(child_genome))
    return new_population

def mutation(genome, mutation_rate, mu=0, sigma=1):
    for prob in range(len(genome)):
        if random.random() < mutation_rate:
            mutation = random.gauss(mu,sigma)
```

```
            genome[prob] = min(max(genome[prob] + mutation, 0), 1)
    return genome

def crossover(genome1, genome2):
    child_genome = [g1 if random.random() > 0.5 else g2 for g1, g2 in zip(genome1,
genome2)]
    return child_genome

def replacement(population, new_population, fitness_scores):
    sorted_population = sorted(zip(population, fitness_scores), key=lambda x:
x[1], reverse=True)
    survivors = sorted_population[:len(population) - len(new_population)]
    return [agent for agent, score in survivors] + new_population
```

Populations functions

- `initialize_population`: this function creates a population of size `population_size` composed of elements of type `NimAgent`. This element is composed of an array named `individual` with size `genome_length`. Each value of the array is a random number between 0 and 1 that represents the probability of making a move in the row equivalent to the index of the element in the array.
- `evaluate_population`: this function evaluates the population with the number of wins they can achieve against the optimal algorithm. In a game therefore our algorithm will be based on making a move on the line with greater probability written in an `individual`. We pass as input the `population` that is the array of `NimAgent`, `nim` is the status of the game and `num_games` represent the number of matches that the population has to do against the optimal algorithm.
- `select_agents`: this function implements the logic to select the fittest agents and to do that we use a simple tournament selection. it takes 2 random participants from the population and selects the winner based on how many times this individual wins against the optimal algorithm i.e., the `fitness_score`. We repeat this process until we have half of the `population_size`.

```
def initialize_population(pop_size, genome_length):
    population = [NimAgent([random.random() for _ in range(genome_length)]) for _
in range(pop_size)]
    return population


def evaluate_population(population: [NimAgent], nim: Nim, num_games: int):
    wins = []
    for individual in population:
        strategy = (optimal, individual.__move)
        win = 0
        for _ in range(num_games):
            nim.reset()
            player = 0
            while nim:
                if player == 1:
                    ply = strategy[player](nim)
                else:
                    ply = strategy[player](nim)
```

```
                nim.nimming(ply)
                player = 1 - player
            if player == 1:
                win += 1
        wins.append(win)
    return wins


def select_agents(population, fitness_scores):
    selected = []
    while len(selected) < len(population) // 2:
        participant = random.sample(list(zip(population, fitness_scores)), 2)
        winner = max(participant, key=lambda x: x[1])
        selected.append(winner[0])
    return selected
```

Evolution Strategy

Problem parameters:

- `nim`: Object of type Nim with the status of the game
- `generations`: number of generations of population to create
- `population_size`: size of the population
- `initial_mutation_rate`: probability to do mutation
- `wins_goal`: number of victories to be achieved to finish the creation of new generations first
- `num_games`: number of games against the optimal algorithm to adjust the fitness scores
- `mutation_probability`: probability to do mutation instead of crossover

In the `evolutionary_strategy` function, we implement the logic of the Evolutionary Algorithm. It returns the population with the best individual found.

```
GENERATIONS = 50
POPULATION_SIZE = 10
INITIAL_MUTATION_RATE = 0.5
WINS_GOAL = 90
NUM_GAMES = 100
MUTATION_PROBABILITY = 0.3
```

```
def evolutionary_strategy(nim, generations, population_size,
initial_mutation_rate, wins_goal, num_games, mutation_probability):
    population = initialize_population(population_size, len(nim.rows))
    best_individual = None
    best_fitness = -1
    mutation_rate = initial_mutation_rate

    for generation in range(generations):
        fitness_scores = evaluate_population(population, nim, num_games)
```

```
        # The best score is halved to report the number of wins, since each win is
worth double points in the scoring system.
        print(f"Generation {generation}: Best score {max(fitness_scores)} wins")

        # Check for termination condition (e.g., a perfect score)
        if max(fitness_scores) >= wins_goal:
            print("Stopping early, reached perfect score!")
            break

        # Selection
        selected_agents = select_agents(population, fitness_scores)

        # Reproduction
        new_population = reproduce(selected_agents, mutation_rate,
mutation_probability)

        # Replacement
        population = replacement(population, new_population, fitness_scores)

        # Check if the new best individual is found
        max_fitness = max(fitness_scores)
        if max_fitness > best_fitness:
            best_fitness = max_fitness
            best_individual_index = fitness_scores.index(max_fitness)
            best_individual = population[best_individual_index]
        # Optionally, adapt the mutation rate
        # This can be a function of the progress stagnation, diversity in
population, etc.

    return population, best_individual
```

Code test

```
nim = Nim(5)
pop, best_ind =
evolutionary_strategy(nim,GENERATIONS,POPULATION_SIZE,INITIAL_MUTATION_RATE,WINS_G
OAL,NUM_GAMES,MUTATION_PROBABILITY)
```

```
Generation 0: Best score 44 wins
Generation 1: Best score 47 wins
Generation 2: Best score 45 wins
Generation 3: Best score 48 wins
Generation 4: Best score 47 wins
Generation 5: Best score 47 wins
Generation 6: Best score 48 wins
Generation 7: Best score 53 wins
Generation 8: Best score 53 wins
Generation 9: Best score 56 wins
```

```
Generation 10: Best score 51 wins
Generation 11: Best score 56 wins
Generation 12: Best score 52 wins
Generation 13: Best score 53 wins
Generation 14: Best score 55 wins
Generation 15: Best score 56 wins
Generation 16: Best score 53 wins
Generation 17: Best score 53 wins
Generation 18: Best score 63 wins
Generation 19: Best score 57 wins
Generation 20: Best score 53 wins
Generation 21: Best score 58 wins
Generation 22: Best score 52 wins
Generation 23: Best score 58 wins
Generation 24: Best score 57 wins
Generation 25: Best score 52 wins
Generation 26: Best score 58 wins
Generation 27: Best score 60 wins
Generation 28: Best score 63 wins
Generation 29: Best score 58 wins
Generation 30: Best score 55 wins
Generation 31: Best score 67 wins
Generation 32: Best score 60 wins
Generation 33: Best score 63 wins
Generation 34: Best score 56 wins
Generation 35: Best score 52 wins
Generation 36: Best score 59 wins
Generation 37: Best score 56 wins
Generation 38: Best score 58 wins
Generation 39: Best score 54 wins
Generation 40: Best score 56 wins
Generation 41: Best score 49 wins
Generation 42: Best score 59 wins
Generation 43: Best score 56 wins
Generation 44: Best score 58 wins
Generation 45: Best score 59 wins
Generation 46: Best score 55 wins
Generation 47: Best score 57 wins
Generation 48: Best score 54 wins
Generation 49: Best score 59 wins
```

Oversimplified match

```
logging.getLogger().setLevel(logging.INFO)

strategy = (optimal, best_ind.__move)

nim = Nim(5)
logging.info(f"init : {nim}")
```

```
player = 0
while nim:
    if player == 1:
        ply = strategy[player](nim)
    else:
        ply = strategy[player](nim)
    logging.info(f"ply: player {player} plays {ply}")
    nim.nimming(ply)
    logging.info(f"status: {nim}")
    player = 1 - player
logging.info(f"status: Player {player} won!")
```

```
INFO:root:init : <1 3 5 7 9>
INFO:root:ply: player 0 plays Nimply(row=4, num_objects=6)
INFO:root:status: <1 3 5 7 3>
INFO:root:ply: player 1 plays Nimply(row=0, num_objects=1)
INFO:root:status: <0 3 5 7 3>
INFO:root:ply: player 0 plays Nimply(row=3, num_objects=5)
INFO:root:status: <0 3 5 2 3>
INFO:root:ply: player 1 plays Nimply(row=1, num_objects=3)
INFO:root:status: <0 0 5 2 3>
INFO:root:ply: player 0 plays Nimply(row=3, num_objects=2)
INFO:root:status: <0 0 5 0 3>
INFO:root:ply: player 1 plays Nimply(row=4, num_objects=3)
INFO:root:status: <0 0 5 0 0>
INFO:root:ply: player 0 plays Nimply(row=2, num_objects=1)
INFO:root:status: <0 0 4 0 0>
INFO:root:ply: player 1 plays Nimply(row=2, num_objects=3)
INFO:root:status: <0 0 1 0 0>
INFO:root:ply: player 0 plays Nimply(row=2, num_objects=1)
INFO:root:status: <0 0 0 0 0>
INFO:root:status: Player 1 won!
```

3. Lab 03 — Black Box Problem

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

Collaboration

For this laboratory, I collaborate with Nicholas Berardo s319441

Professor's lib file

```
# Copyright © 2023 Giovanni Squillero <giovanni.squillero@polito.it>
# https://github.com/squillero/computational-intelligence
# Free for personal or classroom use; see 'LICENSE.md' for details.

from abc import abstractmethod


class AbstractProblem:
    def __init__(self):
        self._calls = 0

    @property
    @abstractmethod
    def x(self):
        pass

    @property
    def calls(self):
        return self._calls

    @staticmethod
    def onemax(genome): #returns the number of 1 in the genome
        return sum(bool(g) for g in genome)

    def __call__(self, genome):
        self._calls += 1
        #genome[s::self.x] -> from s to end skipping self.x
        #onemax(...) -> count number of 1
        #so finsesses will be an array of self.x elements sorted
        fitnesses = sorted((AbstractProblem.onemax(genome[s :: self.x]) for s in
range(self.x)), reverse=True)
        #sum(f for f in fitnesses if f == fitnesses[0]) count the number of
elements in fintsesses that
        # has value equal to the maximum that is in the 0 position
        val = sum(f for f in fitnesses if f == fitnesses[0]) - sum(
            f * (0.1 ** (k + 1)) for k, f in enumerate(f for f in fitnesses if f <
fitnesses[0])
        )
        return val / len(genome)
```

```
def make_problem(a):
    class Problem(AbstractProblem):
        @property
        @abstractmethod
        def x(self):
            return a
    return Problem()
```

Code

```
from random import choices,randint, choice
import random

import lab3_lib
```

```
fitness = lab3_lib.make_problem(10)
for n in range(10):
    ind = choices([0, 1], k=50)
    print(f"{''.join(str(g) for g in ind)}: {fitness(ind):.2%}")

print(fitness.calls)
```

```
10100111010010000111001110111010000110010101110111: 15.33%
11000110011001101011001101110010110100110110001101: 15.33%
01001101001110001000110111110001011110000001100011: 15.33%
01100010110101101011000110001101011000110101100101: 9.11%
01100110110110001000100101100100000000110101110110: 29.56%
11100010000001110001100101001111110001111100110100: 7.33%
00101101110011000100100101110100101011110011110010: 15.33%
11110101001101001001100110100000101010011011110111: 15.33%
11000000101010011110101111110001000110111111000110: 15.33%
01101001011111110001001011000010101010100011010011: 9.13%
10
```

EA

- `mutation`: function that creates a new individual. In this case, we simply change the random value of the genotype. If the selected number of the genome is 0, it will become 1 and viceversa.
- `one_cut_xover`: takes two individuals and produces one. The one-cut technique was used, i.e., a random value is chosen which will be the index of the cut of the parent genotypes.

- `xover`: it creates a child genome by randomly selecting each gene from either parent with equal probability.

```
def mutation(genome):
    index = randint(0,len(genome[0])-1)
    genome[0][index] = 1-genome[0][index]
    return genome[0]

def one_cut_xover(ind1, ind2):
    cut_point = randint(0, len(ind1[0]))
    offspring = ind1[0][:cut_point]+ind2[0][cut_point:]
    return offspring

def xover(genome1, genome2):
    child_genome = [g1 if random.random() > 0.5 else g2 for g1, g2 in
zip(genome1[0], genome2[0])]
    return child_genome
```

Population functions

- `init_population`: function that creates the population. An individual is a tuple with a `length` elemento long array of zero or one and the fitness of this array.
- `gen_new_population`: function that, given the population, returns a new population with the addition of some individual created with `mutation` or `xover`. When we add the individual to the population, we calculate its fitness
- `select_parent`: function that returns the champion out of the population. We first take the better half of the population, then we randomly choose half of these individuals, and then we take the best.
- `replacement`: function that joins the initial population and the new population to select the best `POPULATION_SIZE` individuals

```
def init_population(n_individual,length,fitness):
    pop = []
    for _ in range(n_individual):
        ind = (choices([0, 1], k=length))
        pop.append((ind,fitness(ind)))
    return pop

def gen_new_population(offspring_size,mutation_prob,old_population,fitness):
    new_individual = []
    for _ in range(offspring_size):
        if random.random() < mutation_prob:
            old_ind = select_parent(old_population)
            tmp = mutation(old_ind)
        else:
            old_ind = select_parent(old_population)
            old_ind_2 = select_parent(old_population)
            tmp = one_cut_xover(old_ind,old_ind_2)
        new_individual.append((tmp,fitness(tmp)))
    return new_individual
```

```
def select_parent(population):
    best_parents = sorted(population,key= lambda i:i[1],reverse=True)
[:int(len(population)/2)]
    pool = [choice(best_parents) for _ in range(int(len(population)/4))]
    champion = max(pool, key=lambda i: i[1])
    return champion

def replacement(new_pop,old_pop):
    tmp_pop = new_pop + old_pop
    sorted_pop = sorted(tmp_pop,key= lambda i:i[1],reverse=True)
    return sorted_pop[:len(old_pop)]
```

Problem parameters

- `POPULATION_SIZE`: number of individuals in the population
- `OFFSPRING_SIZE`: number of new individuals created by the `gen_new_population` function
- `LENGTH_INDV`: how many numbers does an individual have
- `GENERATION`: number of generations of population to create
- `MUTATION_PROBABILITY`: probability to do mutation instead of crossover

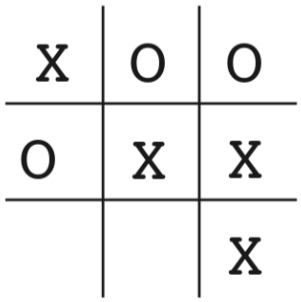
```
POPULATION_SIZE = 500
OFFSPRING_SIZE = 500
LENGTH_INDV = 1000
GENERATION = 500
MUTATION_PROBABILITY = 0.1

problem_size = [1,2,5,10]
```

```
for ps in problem_size:
    fit = lab3_lib.make_problem(ps)
    pop = init_population(POPULATION_SIZE,LENGTH_INDV,fit)
    best = 0
    n_calls = 0
    gen = 0
    for g in range(GENERATION):
        new_pop = gen_new_population(OFFSPRING_SIZE,0.1,pop,fit)
        pop = replacement(new_pop,pop)
        if pop[0][1] > best:
            best = pop[0][1]
            n_calls = fit.calls
            gen = g
    print(f"Problem size: {ps}")
    print(f"Best fitness: {best}")
    print(f"Fitness calls: {n_calls}")
    print(f"Generation: {gen}")
    print(f"Population size: {POPULATION_SIZE}")
```

```
Problem size: 1
Best fitness: 1.0
Fitness calls: 160500
Generation: 319
Population size: 500
Problem size: 2
Best fitness: 0.896
Fitness calls: 238000
Generation: 474
Population size: 500
Problem size: 5
Best fitness: 0.465
Fitness calls: 179000
Generation: 356
Population size: 500
Problem size: 10
Best fitness: 0.2719
Fitness calls: 248500
Generation: 495
Population size: 500
```

4. Lab 04 — Reinforcement Learning

Given the Tic Tac Toe game, we create two players: one with Reinforcement Learning and another with Q-learning.



Collaboration

For this laboratory, I collaborate with Nicholas Berardo s319441

Reinforcement Learning Player

Board State

The `TicTacToe` class reflects the state of the board. We use 1 to indicate player1 and -1 for player 2.

Parameter description

- `board`: numpy array of dimension 3x3 that represents the game board.
- `p1`: player1 class.
- `p2`: player2 class.
- `current_player`: indicates who has to take the turn.
- `isEnd`: boolean that indicate if the game has finished.
- `boardHash`: it is the board status but as a string.

Methods description

- `available_positions`: it returns an array with the list of possible moves (each element is a tuple of two integers that indicates where to play).
- `make_move`: takes as input the location of where a player played and puts the value of the `current player` in that place on the board, i.e. who is playing at that moment. It also gives the other player the turn by changing the `current_player` parameter.
- `get_hash`: it returns the board state but in a string format.
- `check_winner`: It checks if there is a winner.
- `reward`: It calls the function of the players that update the value estimation of states giving them the reward (1 if a player won, 0 if he loses).
- `reset`: It resets the state of the board by emptying the boxes with also all the other parameters.
- `show_board`: It prints the board status. `player1` is the X and `player2` is the O.

- `train`: We used two agents that use Reinforcement Learning to play against each other. During training the process of each player is: look for available positions, choose action, update board state and add the action to player's states, judge if reach the end of the game and give reward accordingly.
- `test`: We test our trained policy with a random player.

```
import numpy as np
import pickle
from tqdm.auto import tqdm
```

```
class TicTacToe:
    def __init__(self, p1, p2):
        self.board = np.zeros((3, 3))
        self.p1 = p1
        self.p2 = p2
        self.isEnd = False
        self.boardHash = None
        self.current_player = 1  # 1 is p1, -1 is p2

    def available_positions(self):
        pos = []
        for i in range(3):
            for j in range(3):
                if self.board[i, j] == 0:
                    pos.append((i, j))
        return pos

    def make_move(self, position):
        if position not in self.available_positions():
            return None
        self.board[position] = self.current_player
        self.current_player = self.current_player * -1

    def get_hash(self):
        self.boardHash = str(self.board.reshape(3 * 3))
        return self.boardHash

    def check_winner(self):
        # check if rows contains 3 or -3 (someone win)
        for i in range(3):
            if sum(self.board[i, :]) == 3:
                self.isEnd = True
                return 1  # player 1 won
        for i in range(3):  # loop on the rows
            if sum(self.board[i, :]) == -3:
                self.isEnd = True
                return -1  # player 2 won

        # check if col contains 3 or -3
        for i in range(3):
            if sum(self.board[:, i]) == 3:
```

```
                self.isEnd = True
                return 1
            for i in range(3):
                if sum(self.board[:, i]) == -3:
                    self.isEnd = True
                    return -1

            # check diagonal win
            diag_sum = sum([self.board[i, i] for i in range(3)])
            if diag_sum == 3:
                self.isEnd = True
                return 1
            if diag_sum == -3:
                self.isEnd = True
                return -1

            diag_sum = sum([self.board[i, 3 - i - 1] for i in range(3)])
            if diag_sum == 3:
                self.isEnd = True
                return 1
            if diag_sum == -3:
                self.isEnd = True
                return -1

            # here no one won..
            if len(self.available_positions()) == 0:
                self.isEnd = True
                return 0  # no one won

            return None  # Here there are still moves, so keep playing !!!

        def reward(self, result):
            if result == 1:
                self.p1.give_rew(1)  # player 1 won, so give 1 reward
                self.p2.give_rew(0)
            elif result == -1:
                self.p1.give_rew(0)
                self.p2.give_rew(1)
            else:
                self.p1.give_rew(0.1)  # give a less reward because we don't want ties
                self.p2.give_rew(0.5)

        def reset(self):
            self.board = np.zeros((3, 3))
            self.boardHash = None
            self.isEnd = False
            self.current_player = 1

        def show_board(self):
            # p1: x  p2: o
            for i in range(0, 3):
                print('-------------')
                out = '| '
                for j in range(0, 3):
```

```
            if self.board[i, j] == 1:
                token = 'x'
            if self.board[i, j] == -1:
                token = 'o'
            if self.board[i, j] == 0:
                token = ' '
            out += token + ' | '
        print(out)
    print('-------------')

def train(self, rounds=10000):
    for epochs in tqdm(range(rounds)):
        while not self.isEnd:

            # Player 1
            positions = self.available_positions()
            p1_action = self.p1.make_move(positions, self.board,
self.current_player)
            # take action and update board state
            self.make_move(p1_action)
            board_hash = self.get_hash()
            self.p1.add_state(board_hash)
            # check the board status if it is ended
            win = self.check_winner()

            if win is not None:  # It returns None only when no one finished or
tied.
                # self.showBoard()
                # ended with p1 either win or draw
                self.reward(win)  # send rewards to the players, the game has ended
                self.p1.reset()
                self.p2.reset()
                self.reset()
                break

            else:
                # Player 2
                positions = self.available_positions()
                p2_action = self.p2.make_move(positions, self.board,
self.current_player)
                self.make_move(p2_action)
                board_hash = self.get_hash()
                self.p2.add_state(board_hash)

                win = self.check_winner()
                if win is not None:
                    # self.showBoard()
                    # ended with p2 either win or draw
                    self.reward(win)
                    self.p1.reset()
                    self.p2.reset()
                    self.reset()
                    break
```

```
    def test(self):
        while not self.isEnd:
            # Player 1
            positions = self.available_positions()
            p1_action = self.p1.make_move(positions, self.board, self.current_player)
            # take action and update board state
            self.make_move(p1_action)
            # check board status if it is ended
            win = self.check_winner()
            if win is not None:  # if win is not None means someone win or tie
                return win

            else:
                # Player 2
                positions = self.available_positions()
                p2_action = self.p2.make_move(positions, self.board,
self.current_player)

                self.make_move(p2_action)
                win = self.check_winner()
                if win is not None:
                    return win
```

RL Player

This class represents a player that uses Reinforcement Learning to make decisions in Quixo. More precisely, our player uses Temporal difference (TD) learning. TD in reinforcement learning is an unsupervised learning technique very commonly used in it for the purpose of predicting the total reward expected over the future. Essentially, TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods perform state value function updates based on current estimates.

Parameters description

- `states_value`: Dictionary that has as key the states that a player has seen during all the matches and as value the parameter that we want to train.
- `exp_rate`: Probability of doing a random move instead of the action with max Q-value.
- `decay_gamma`: The exploration decay rate used during the training
- `states`: All state-action pairs a player has seen during a single match. It is used at the end of each match to update the `states_value`.

Methods description

- `get_hash`: it returns the board state but in a string format.
- `add_state`: It adds to the `states` array the state that a player has seen during a game.
- `reset`: It reset the `states` array to be able to start a new game.
- `choose_action`: It receives as input all the possible `positions` to play, the `current_board` that is the status of the board and the `simbol` that indicate the current player (1 for player1 and -1 for player2).

This function has the job to decide the move of a player that can be random or based on the value of the dictionary. It takes from the dictionary, for each possible move, the value associated with the state of the board with the move performed. The maximum value will be the move to execute. We use the following recursive (bellman equation) formula to compute the state-value table:

$$V(S_t) \leftarrow V(S_t) + \alpha * (\gamma * V(S_t + 1) - V(S_t))$$

The formula simply tells us that the updated value of state t equals the current value of state t adding the difference between the value of the next state , which is multiplied by the discount factor of the Bellman Equation, and the value of the current state, which is multiplied by a learning rate α. The logic is that we update the current value slowly based on our latest observation.

- `give_rew`: This function is called at every end of each game. It updates the values of the `states_value` dictionary based on the states that the player has seen during the game and the reward that they have provided.
- `save_policy`: It saves the `states_value` dictionary that we have trained to a file.
- `load_policy`: It loads the `states_value` dictionary from a file.

```
class RLPlayer:
    def __init__(self, name, lr=0.2, decay_gamma=0.9, exp_rate = 0.2):
        self.name = name
        self.states = []  # record all positions taken
        self.lr = lr
        self.exp_rate = exp_rate
        self.decay_gamma = decay_gamma
        self.states_value = {}  # state -> value

    def get_hash(self, board):
        boardHash = str(board.reshape(3*3))
        return boardHash

    def add_state(self, state):
        self.states.append(state)

    def choose_action(self, positions, current_board, symbol):
        if np.random.uniform(0, 1) <= self.exp_rate: # Do exploration, take random
            # take random action
            idx = np.random.choice(len(positions))
            action = positions[idx]
        else: #Here do exploitation, take the action that has the highest value
            value_max = -999
            for p in positions:
                next_board = current_board.copy() #create a tmp board
                next_board[p] = symbol #do the action
                next_board_hash = self.get_hash(next_board) #get the hash
                value = 0 if self.states_value.get(next_board_hash) is None else
self.states_value.get(next_board_hash)
                # print("value", value)
                if value >= value_max: #find the action that has max value.
                    value_max = value
                    action = p
        return action
```

```
    def reset(self):
        self.states = []

    def give_rew(self, reward):
        #At the end of the game, I'll get a reward. The iterating on the states in
reverse.
        # Set the value of the state to 0 if not existing, otherwise update it
with the reward.
        for st in reversed(self.states):
            if self.states_value.get(st) is None: #if the state doesn't have a
value, set it to 0
                self.states_value[st] = 0
            #this is V(t) = V(t) + lr * (gamma*V(t+1) - V(t))
            self.states_value[st] += self.lr * (self.decay_gamma * reward -
self.states_value[st])
            reward = self.states_value[st]

    def save_policy(self):
        fw = open('policy_' + str(self.name), 'wb')
        pickle.dump(self.states_value, fw)
        fw.close()

    def load_policy(self, file):
        fr = open(file,'rb')
        self.states_value = pickle.load(fr)
        fr.close()
```

Random Player

```
class RandomPlayer:
    def __init__(self, name):
        self.name = "random"

    def choose_action(self, positions, board, current_player):
        x = np.random.randint(0,len(positions)-1)
        return positions[x]

    def add_state(self, state):
        pass

    def give_rew(self, reward):
        pass

    def reset(self):
        pass
```

Hyperparameters

- epochs: training epochs

- `alpha`: learning rate
- `epsilon`: probability of doing a random move instead of the action with max value
- `discount_factor`: the discount rate of the Bellman equation
- `num_games`: number of games for testing

```
epochs = 50000
alpha = 0.2
epsilon = 0.2
discount_factor = 0.9
num_games = 1000
```

Let`s do some computation

```
p1 = RLPlayer("p1_RL", lr=alpha, decay_gamma=discount_factor, exp_rate=epsilon)
p2 = RLPlayer("p2_RL", lr=alpha, decay_gamma=discount_factor, exp_rate=epsilon)
st = TicTacToe(p1, p2)

print("training...")
st.train(rounds=epochs)
```

Test Reinforcement Learning

```
p2 = RandomPlayer("Random")
st = TicTacToe(p1,p2)
win_comp = 0
num_draws = 0

for epoch in range(num_games):
    win = st.test()
    if win == 1:
        win_comp+=1
    if win == 0:
        num_draws+=1
    st.reset()

print(f"Over 1000 matches: {win_comp} wins, {1000 - win_comp - num_draws} losses,
{num_draws} draws")
print(f"Wins + Draws percentage: {(win_comp + num_draws) / epochs * 100}")
```

```
Over 1000 matches: 910 wins, 51 losses, 39 draws
Wins + Draws percentage: 94.89999999999999
```

Q-Learning Player

Board State

The `TicTacToe` class reflects the state of the board. We use 1 to indicate player1 and -1 for player 2.

Parameter description

- `board`: numpy array of dimension 3x3 that represents the game board.
- `players`: contains the numbers that identify our players (1 is `player1` and -1 is `player2`).
- `current_player`: indicates who has to take the turn.
- `winner`: indicates the winner of the game (1 if `player1` won, -1 if `player2` won).
- `game_over`: boolean that indicate if the game has finished.

Methods description

- `available_moves`: it returns an array with the list of possible moves (each element is a tuple of two integers that indicates where to play).
- `make_move`: takes as input the location of where a player played and puts the value of the `current player` in that place on the board, i.e. who is playing at that moment. It also calls the functions `check_winner` and `switch_player` to control if that move makes a player win and to and gives the other player the turn. It returns the new board but not as a matrix but as a tuple of tuples using the `convert_matrix_to_tuple` function, just to be more comfortable with the QAgent dictionary implementation.
- `switch_player`: It gives the other player the turn. If there is, it set the `winner` param with the player who won (1 or -1) and also set `game_over` to `True`.
- `check_winner`: It checks if there is a winner.
- `convert_matrix_to_tuple`: It converts a matrix (that will always be the board status) to a tuple of tuples. For example, a matrix [[1,0,0],[0,1,0],[0,0,1]] will become ((1,0,0), (0,1,0),(0,0,1)).
- `reset`: It resets the state of the board by emptying the boxes with also all the other parameters.
- `show_board`: It prints the board status. `player1` is the X and `player2` is the O.

```
import numpy as np
import random
import pickle
from tqdm.auto import tqdm
```

```
class TicTacToe:
    def __init__(self):
        self.board = np.zeros((3, 3))
        self.players = [1, -1]
        self.current_player = 1
        self.winner = None
        self.game_over = False

    def available_moves(self):
        moves = []
        for i in range(3):
```

```
                for j in range(3):
                    if self.board[i][j] == 0:
                        moves.append((i, j))
        return moves

    def make_move(self, move):
        if self.board[move[0]][move[1]] != 0:
            return False
        self.board[move[0]][move[1]] = self.current_player
        self.check_winner()
        self.switch_player()
        return self.convert_matrix_to_tuple(self.board)

    def switch_player(self):
        if self.current_player == self.players[0]:
            self.current_player = self.players[1]
        else:
            self.current_player = self.players[0]

    def check_winner(self):
        # Check rows
        for i in range(3):
            if self.board[i][0] == self.board[i][1] == self.board[i][2] != 0:
                self.winner = self.board[i][0]
                self.game_over = True
        # Check columns
        for j in range(3):
            if self.board[0][j] == self.board[1][j] == self.board[2][j] != 0:
                self.winner = self.board[0][j]
                self.game_over = True
        # Check diagonals
        if self.board[0][0] == self.board[1][1] == self.board[2][2] != 0:
            self.winner = self.board[0][0]
            self.game_over = True
        if self.board[0][2] == self.board[1][1] == self.board[2][0] != 0:
            self.winner = self.board[0][2]
            self.game_over = True
        # Check tie
        if len(self.available_moves())==0:
            self.winner = 0
            self.game_over = True

    def convert_matrix_to_tuple(self, board):
        current_board = tuple(tuple(riga) for riga in board)
        return current_board

    def reset(self):
        self.board = np.zeros((3, 3))
        self.current_player = 1
        self.winner = 0
        self.game_over = False

    def show_board(self):
        # p1: x  p2: o
```

```
        for i in range(0, 3):
            print('-------------')
            out = '| '
            for j in range(0, 3):
                if self.board[i, j] == 1:
                    token = 'x'
                if self.board[i, j] == -1:
                    token = 'o'
                if self.board[i, j] == 0:
                    token = ' '
                out += token + ' | '
            print(out)
        print('-------------')
        print()
```

Q Player

The QLearningAgent represents the player that will be trained with Q-Learning. Q-learning is a reinforcement learning technique which is based on updating the action-value based on the difference between the current estimate and the actual rewards received.

We will represent the Q-values as a dictionary of state-action pairs, where each state is a tuple representing the current state of the board, and each action is a tuple representing the coordinates of the move. This state-action pair will be the key of our dictionary and the Q-values are the values. The initial Q-values will be set to zero. We update the action-value $Q(s_t, a_t)$ according to this formula:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * (\gamma * R_{t+1} - Q(s_t, a_t))$$

Parameters description

- `Q`: Dictionary with state-action pairs as key the Q-values as values.
- `alpha`: Learning rate.
- `epsilon`: Probability of doing a random move instead of the action with max Q-value.
- `discount_factor`: The exploration decay rate used during the training
- `states`: All state-action pairs a player has seen during a single match. It is used at the end of each match to update the Q-values.

Methods description

- `get_Q_value`: function that returns the Q-value given a state-action pair. If the key is not present in the dictionary, it creates it with Q-value equal to 0.
- `add_state`: It adds to the `states` array a state-action pair.
- `reset`: It reset the `states` array to be able to start a new game.
- `choose_action`: It firstly adds to the dictionary every new state-action pair based on the new possible state of the board. It then chooses the action that can be random or based on the `Q` dictionary.
- `update_Q_value`: This function is called at every end of each game. It updates the Q-values of the `Q` dictionary based on the states that the player has seen during the game and the reward that they have provided.
- `save_policy`: It saves the `Q` dictionary that we have trained to a file.

- `load_policy`: It loads the `Q` dictionary from a file.

```
class QLearningAgent:
    def __init__(self, alpha, epsilon, discount_factor):
        self.Q = {}
        self.alpha = alpha
        self.epsilon = epsilon
        self.discount_factor = discount_factor
        self.states = [] # record all positions taken + action

    def get_Q_value(self, state, action):
        if (state, action) not in self.Q:
            self.Q[(state, action)] = 0.0
        return self.Q[(state, action)]

    def add_state(self, state, action):
        self.states.append((state,action))

    def reset(self):
        self.states = []

    def choose_action(self, state, available_moves):
        Q_values = [self.get_Q_value(state, action) for action in available_moves]
        if random.uniform(0, 1) < self.epsilon:
            return random.choice(available_moves)
        else:
            max_Q = max(Q_values)
            if Q_values.count(max_Q) > 1:
                best_moves = [i for i in range(len(available_moves)) if
Q_values[i] == max_Q]
                i = random.choice(best_moves)
            else:
                i = Q_values.index(max_Q)
            return available_moves[i]

    def update_Q_value(self, reward):
        for st in reversed(self.states):
            current_q_value = self.Q[(st[0], st[1])] # st[0] = board state st[1] =
action
            reward = current_q_value + self.alpha * (self.discount_factor * reward
- current_q_value)
            self.Q[(st[0], st[1])] = reward

    def save_policy(self):
        fw = open('policy_QL', 'wb')
        pickle.dump(self.Q, fw)
        fw.close()

    def load_policy(self, file):
        fr = open(file,'rb')
        self.Q = pickle.load(fr)
        fr.close()
```

Train and Test functions

We used two agents that use Q-learning. They play against each other to train.

```
def train(player1, player2, num_episodes):
    state = TicTacToe()
    for epoch in tqdm(range(num_episodes)):
        state.reset()
        player1.reset()
        player2.reset()
        state_board = state.convert_matrix_to_tuple(state.board)
        while not state.game_over:
            # Player 1
            action = player1.make_move(state_board, state.available_moves())
            player1.add_state(state_board, action)
            state_board = state.__move(action)

            if state.winner is not None:
                if state.winner == 1:
                    player1.update_Q_value(1)  # player 1 won, so give 1 reward
                    player2.update_Q_value(0)
                elif state.winner == -1:
                    player1.update_Q_value(0)
                    player2.update_Q_value(1)
                else:
                    player1.update_Q_value(0.1)  # give a less reward because we don't
want ties
                    player2.update_Q_value(0.5)

            else:
                # Player 2
                action = player2.make_move(state_board, state.available_moves())
                player2.add_state(state_board, action)
                state_board = state.__move(action)

                if state.winner is not None:
                    if state.winner == 1:
                        player1.update_Q_value(1)  # player 1 won, so give 1 reward
                        player2.update_Q_value(0)
                    elif state.winner == -1:
                        player1.update_Q_value(0)
                        player2.update_Q_value(1)
                    else:
                        player1.update_Q_value(0.1)  # give a less reward because we
don't want ties
                        player2.update_Q_value(0.5)
    return player1, player2


def test(agent, num_games, print_board=False):
    num_wins = 0
```

```
    num_draws = 0
    for i in range(num_games):
        state = TicTacToe()
        state_board = state.convert_matrix_to_tuple(state.board)
        while not state.game_over:
            if state.current_player == 1:
                action = agent.make_move(state_board, state.available_moves())
            else:
                action = random.choice(state.available_moves())
            state_board = state.__move(action)
            if print_board:
                state.show_board()
        if state.winner == 1:
            num_wins += 1
        if state.winner == 0:
            num_draws += 1
    return num_wins, num_draws
```

Hyperparameters

- `epochs`: training epochs
- `alpha`: learning rate
- `epsilon`: probability of doing a random move instead of the action with max value
- `discount_factor`: the exploration decay rate used during the training
- `num_games`: number of games for testing

```
epochs = 2000000
alpha = 0.2
epsilon = 0.2
discount_factor = 0.9
num_games = 1000
```

Let's do some computation

```
player1 = QLearningAgent(alpha, epsilon, discount_factor)
player2 = QLearningAgent(alpha, epsilon, discount_factor)

Trained_player1, Trained_player2 = train(player1, player2, epochs)
```

```
  0%|          | 0/2000000 [00:00<?, ?it/s]
```

```
# Trainer_player1 is the X
_ = test(agent=Trained_player1, num_games=1, print_board=True)
```

```
-------------
|   |   |   |
-------------
|   | x |   |
-------------
|   |   |   |
-------------

-------------
|   | o |   |
-------------
|   | x |   |
-------------
|   |   |   |
-------------

-------------
|   | o | x |
-------------
|   | x |   |
-------------
|   |   |   |
-------------

-------------
| o | o | x |
-------------
|   | x |   |
-------------
|   |   |   |
-------------

-------------
| o | o | x |
-------------
|   | x |   |
-------------
| x |   |   |
-------------
```

```
num_wins, num_draws = test(agent=Trained_player1, num_games=num_games)
print(f"Over 1000 matches: {num_wins} wins, {1000 - num_wins - num_draws} losses,
{num_draws} draws")
print(f"Wins + Draws percentage: {(num_wins + num_draws) / num_games * 100}")
```

```
Over 1000 matches: 911 wins, 47 losses, 42 draws
Wins + Draws percentage: 95.3
```

5. Halloween challenge

Find the best solution with the fewest calls to the fitness functions for:

- `num_points = [100, 1_000, 5_000]`
- `num_sets = num_points`
- `density = [.3, .7]`

Code

Imported Libraries

```
from itertools import product
from random import random, randint, shuffle, seed, choice
import numpy as np
from scipy import sparse
from copy import copy
```

Problem instance

This function, called `make_set_covering_problem`, returns a sparse matrix of size `num_sets` (number of sets) * `num_points` (number of elements) with Boolean values (True or False) in which the rows represent sets, and the columns represent the elements covered by the sets. The density of the matrix is controlled by a parameter called `density`. For each pair, if a random number generated with random() is less than the specified `density`, sets the corresponding element to True in the set.

The `current_state` is an array of boolean where true indicates if the state contains that particular set, false if it doesn't contain it. We have to initialize it to a random possible solution.

```
def make_set_covering_problem(num_points, num_sets, density):
    """Returns a sparse array where rows are sets and columns are the covered
items"""
    seed(num_points*2654435761+num_sets+density)
    sets = sparse.lil_array((num_sets, num_points), dtype=bool)
    for s, p in product(range(num_sets), range(num_points)):
        if random() < density:
            sets[s, p] = True
    for p in range(num_points):
        sets[randint(0, num_sets-1), p] = True
    return sets
```

```
num_points = 1000
num_sets = num_points
```

```
density = .3
matrix = make_set_covering_problem(num_points, num_sets, density)

starting_state = [choice([True, False]) for _ in range(num_sets)]
```

Fitness function

The `fitness` function return the number of covered elements and the negative cost that is the number of taken sets as negative. That's because we want to take the solution with the smallest number of taken sets.

```
def fitness(state):
    global fitness_counter
    fitness_counter+=1
    cost = sum(state)
    valid = len(list(set(np.concatenate(matrix.rows[state]))))
    return valid, -cost
```

Hill Climbing

The `tweak` function swap one of the set randomly, if it was taken we change it into not taken and vice versa.

Results:

num points	num sets	density	number of evaluation	best fitness
100	100	.3	200	(100, -14)
100	100	.7	200	(100, -21)
1000	1000	.3	2.000	(1000, -183)
1000	1000	.7	2.000	(1000, -192)
5000	5000	.3	10.000	(5000, -929)
5000	5000	.7	10.000	(5000, -)

```
def tweak(state):
    new_state = copy(state)
    index = randint(0, num_points - 1) # pick a random index
    new_state[index] = not new_state[index] # swap
    return new_state
```

```
current_state = starting_state
global fitness_counter
fitness_counter=0
for step in range(num_points):
    new_state = tweak(current_state)
```

```
    if fitness(new_state) > fitness(current_state):
        current_state = new_state
print(fitness(current_state), fitness_counter)
```

Simulated Annealing

This algorithm is a Hill Climbing but with probability p≠0 of accepting a worsening solution s' where

$$f(s) > f(s)' : p = e^{-} \frac{f(s) - f(s')}{t}$$

where s is the current solution, s' is the tweaked one and t is the temperature. The idea is that the further we go with the exploration, the more T decreases and with it also the probability of accepting worse solutions.

Results:

num points	num sets	density	number of evaluation	best fitness
100	100	.3	200	(100, -38)
100	100	.7	200	(100, -45)
1000	1000	.3	2.000	(1000, -488)
1000	1000	.7	2.000	(1000, -492)
5000	5000	.3	10.000	(5000, -2461)
5000	5000	.7	10.000	(5000, -2511)

```
current_state = starting_state
global fitness_counter
fitness_counter=0
t = num_points
for step in range(num_points):
    new_state = tweak(current_state)
    _, newCost = fitness(new_state)
    _, oldCost = fitness(current_state)
    if newCost <= oldCost and random() < np.exp(-((oldCost-newCost)/t)):
        current_state = new_state
    elif newCost > oldCost:
        current_state = new_state
    t=t-1
print((fitness(current_state), fitness_counter))
```

Tabu Search

This algorithm keeps track of the states we have visited and avoids going back to them.

Results:

num points	num sets	density	number of evaluation	best fitness
100	100	.3	191	(100, -14)
100	100	.7	197	(100, -21)
1000	1000	.3	1.993	(1000, -183)
1000	1000	.7	1.991	(1000, -190)
5000	5000	.3	9.997	(5000, -929)
5000	5000	.7	9.991	(5000, -941)

```
current_state = starting_state
already_visited = []
global fitness_counter
fitness_counter=0
for step in range(num_points):
    new_state = tweak(current_state)
    if new_state not in already_visited:
        already_visited.append(new_state)
        if fitness(new_state) > fitness(current_state):
            current_state = new_state
print(fitness(current_state), fitness_counter)
```

```
(1000, -183) 1993
```

Iterated Local Search

Is a version of Hill Climbing. The Hill Climbing algorithm is put into a loop and restarted in a new position which can be the global optimum or the last optimum.

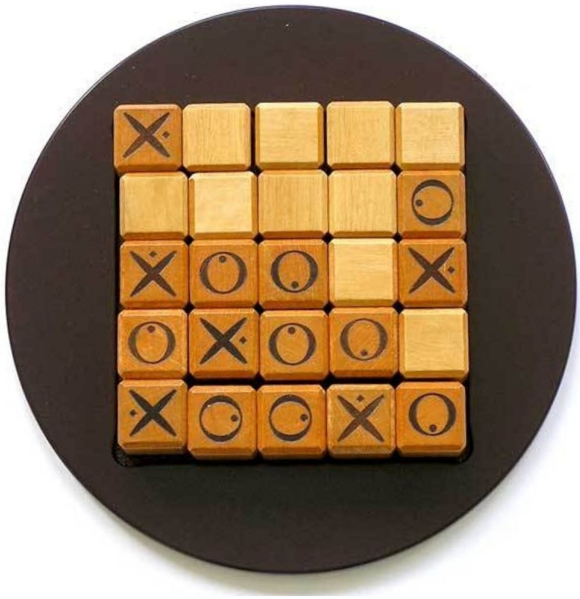
Results:

num points	num sets	density	number of evaluation	best fitness
100	100	.3	200	(100, -9)
100	100	.7	200	(100, -4)
1000	1000	.3	2.000	(1000, -14)
1000	1000	.7	5.000	(1000, -5)
5000	5000	.3	100.000	(5000, -21)
5000	5000	.7	100.000	(5000, -11)

```
current_state = starting_state
global fitness_counter
fitness_counter=0
N_Restart = 5
solution = current_state
for i in range (N_Restart):
    for step in range(num_points):
        new_state = tweak(solution)
        if fitness(new_state) > fitness(solution):
            solution = new_state
print((fitness(solution), fitness_counter))
```

6. Final Project - Quixo

This project is a game of Quixo, a board game similar to Tic-Tac-Toe. The game is played on a 5x5 board, and the goal is to get five of your pieces in a row. The catch is that you can only move the pieces on the outside of the board, and you can only move them in a straight line. The game is played by two players, and each player has five pieces. The game is played by moving a piece to an empty space, and then pushing the row or column of the piece in the direction of the move.



Files

- `game.py`: This file is used to play the game using the different players.
- `CustomGameClass.py`: This file contains the custom class that extend `game.py`.
- `Quixo.ipynb`: This notebook contains the players with their testing.
- `Players.rar`: This file contains the trained players (RL_Player_1, RL_Player_2, RL_Player_3).

Players

We develop two players:

- Reinforcement Learning player trained against a random player.
- Minimax player

We tried a lot of different parameters for the Reinforcement Learning player, but we saved the best three that are saved in the `Players.rar` file. The hyperparameters of all the players are reported in the notebook.

Resources used

- Reinforcement Learning: an Easy Introduction to Value Iteration
- Temporal difference learning

Collaboration

For this project, I collaborate with Nicholas Berardo s319441

```
import pickle
import math
import matplotlib.pyplot as plt
from random import randint, random, choice
from game import Move, Player
from CustomGameClass import Quixo as Game
from tqdm import trange
from typing import Literal
import numpy as np
import json
```

Reinforcement Learning Player

For the implementation of our Reinforcement Learning player, we used the same strategy as for Laboratory 4 i.e. using a Temporal difference method. However, I want to write the entire description of the method for completeness.

This class represents a player that uses Reinforcement Learning to make decisions in Quixo. More precisely, our player uses Temporal difference (TD) learning. TD in reinforcement learning is an unsupervised learning technique very commonly used in it for the purpose of predicting the total reward expected over the future. Essentially, TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods perform state value function updates based on current estimates.

Attributes:

- `epochs` (int): The number of training epochs.
- `alpha` (float): The learning rate.
- `discount_factor` (float): The discount factor of the Bellman equation.
- `min_exploration_rate` (float): The minimum exploration rate during training.
- `exploration_decay_rate` (float): The rate at which the exploration rate decays during training.
- `opponent` (Player): The opponent player.
- `states` (list): A list to store the states visited during a game.
- `state_value` (dict): A dictionary to store the value of each state.
- `training_phase` (bool): A boolean to indicate if the player is training or not. It basically enables the exploration if it is true, otherwise it uses only the `state_value` dictionary to make decisions.

I want to underline that I saw the idea of `exploration_decay_rate` from Davide Sferrazza in his Lab 4 implementation thanks to the peer review.

Methods:

- `give_rew(reward)`: Placeholder method for giving reward to the player.
- `add_state(state)`: Adds to the `states` array the state that a player has seen during a game.
- `reset()`: Reset the `states` array to be able to start a new game.

- `make_move(game)`: Chooses an action to take based on the current game state that can be random or based on the value of the dictionary. It takes from the dictionary, for each possible move, the value associated with the state of the board with the move performed. The maximum value will be the move to execute. We use the following recursive (bellman equation) formula to compute the state-value table:

$$V(S_t) \leftarrow V(S_t) + \alpha * (\gamma * V(S_t + 1) - V(S_t))$$

  The formula simply tells us that the updated value of state t equals the current value of state t adding the difference between the value of the next state , which is multiplied by the discount factor of the Bellman Equation, and the value of the current state, which is multiplied by a learning rate α. The logic is that we update the current value slowly based on our latest observation.
- `update_state_value_table(reward)`: Updates the values of the `states_value` dictionary based on the states that the player has seen during the game and the reward that they have provided.
- `game_reward(player)`: Calculates the reward for the player in the current game.
- `train()`: Trains the player using reinforcement learning.
- `save_policy(name)`: Saves the state value table to a file.
- `load_policy(file)`: Loads the state value table from a file.

```
class RLPlayer(Player):
    def __init__(self, epochs: int,
                 alpha: float,
                 discount_factor: float,
                 min_exploration_rate: float,
                 exploration_decay_rate: float,
                 opponent: 'Player',
                 training_phase: bool) -> None:

        super().__init__()
        self.epochs = epochs
        self.alpha = alpha
        self.discount_factor = discount_factor
        self.exploration_rate = 1
        self.min_exploration_rate = min_exploration_rate
        self.exploration_decay_rate = exploration_decay_rate
        self.opponent = opponent
        self.training_phase = training_phase
        self.states=[]
        self.state_value = {}

    def add_state(self, state):
        self.states.append(state)

    def reset(self):
        self.states = []

    def make_move(self, game: Game) -> tuple[tuple[int, int], Move]:
        available_moves = get_possible_moves(game, game.get_current_player())
        if self.training_phase and (random() < self.exploration_rate):  # do
exploration
            return choice(available_moves)
```

```
        else:  # do exploitation
            value_max = -math.inf
            for move in available_moves:
                tmp = game.get_board()
                game._Game__move(move[0], move[1], game.get_current_player())
                next_status = convert_matrix_board_to_tuple(game.get_board())
                game.set_board(tmp)
                value = 0 if self.state_value.get(next_status) is None else
self.state_value.get(next_status)
                if value > value_max:
                    value_max = value
                    action = move
        return action

    def update_state_value_table(self, reward):
        for st in reversed(self.states):
            if self.state_value.get(st) is None:
                self.state_value[st] = 0
            current_value = self.state_value[st]
            reward = current_value + self.alpha * (self.discount_factor * reward -
current_value)
            self.state_value[st] = reward

    def game_reward(self, player: 'Player')-> Literal[-10, 10]:
        if self == player:
            return 10
        else:
            return -10

    def train(self, player_name='') -> None:
        game = Game()
        all_rewards = []
        # define how many episodes to run
        pbar = trange(self.epochs)
        # define the players
        players = (self, self.opponent)

        for epochs in pbar:
            rewards = 0
            winner = -1
            players = (players[1], players[0])
            player_idx = 1

            while winner < 0:
                # change player
                player_idx = (player_idx + 1) % 2
                player = players[player_idx]
                game.switch_player()

                ok = False
                if self == player:
                    while not ok:
                        from_pos, slide = self.make_move(game)
                        ok = game._Game__move(from_pos, slide,
```

```
game.get_current_player())
                        state_after_move =
convert_matrix_board_to_tuple(game.get_board())
                        self.add_state(state_after_move)

                else:
                    while not ok:
                        from_pos, slide = player.make_move(game)
                        ok = game._Game__move(from_pos, slide,
game.get_current_player())

                winner = game.check_winner()

            # update the exploration rate
            self.exploration_rate = np.clip(
                np.exp(-self.exploration_decay_rate * epochs),
self.min_exploration_rate, 1
            )

            reward = self.game_reward(player)
            self.update_state_value_table(reward)
            rewards += reward
            all_rewards.append(rewards)

            self.reset()
            game.reset()

            pbar.set_description(f'rewards value: {rewards}, current exploration
rate: {self.exploration_rate:2f}')

        plot_training_trends(all_rewards, filename=f"{player_name} trained against
{self.opponent.__class__.__name__}")

        print(f'** Last 50_000 episodes - Mean rewards value:
{sum(all_rewards[-50_000:]) / 50_000:.2f} **')

    def save_policy(self, name):
        fw = open(name, 'wb')
        pickle.dump(self.state_value, fw, protocol=4)
        fw.close()

    def load_policy(self, file):
        fr = open(file, 'rb')
        self.state_value = pickle.load(fr)
        fr.close()
```

MinMax Player

This class represents a player who uses the MinMax algorithm to make decisions in the game. The MinMax algorithm is a search algorithm that is used in two-player games to make optimal decisions.

Attributes:

- `playerPlaying` (int): The player who is currently playing.
- `levels_depth` (list): A list of tuples, where each tuple contains the depth level and the maximum number of possible moves for that depth level.

Methods:

- `game_evaluation`: Evaluate the game state based on the current player and depth. If player X wins, the reward is 1, otherwise it is -1. If player O plays instead, the rewards are reversed.
- `min_max`: Perform the Minimax algorithm to determine the best move for the current player. This method takes as input the current game, the alpha and beta values (used for alpha-beta pruning, a technique for reducing the number of nodes evaluated by the MinMax algorithm), and the current depth of the search tree. If the depth is zero or if there is a winner in the game, the method returns the game rating and no moves. Otherwise, for each possible move, it creates a copy of the game, makes the move, and recursively calls the min_max method on the copy of the game. If the returned rating is greater than alpha, alpha is updated and the move is considered the best move. If beta is less than or equal to alpha, the cycle stops for alpha-beta pruning. The process is similar for the case where the current player is non-zero, with the difference that we try to minimize the rating instead of maximizing it.
- `choose_action`: Choose the best action (move) for the current player using the Minimax algorithm. The method starts by getting all possible moves for the current player using the `get_possible_moves` function. It then calculates the search depth for the MinMax algorithm based on the number of possible moves. This is done through a for loop that passes through the `levels_Depth` list. If the number of possible moves is greater than a certain value, the depth is set to a certain level. The cycle stops as soon as a level is found that does not exceed the number of possible moves. The method then calls the `min_max` method to determine the best move for the current player. Finally, the method returns the best move.

```
class MinMaxPlayer(Player):
    def __init__(self, playerPlaying, levels_depth):
        super().__init__()
        self.moves_value = []
        self.playerPlaying = playerPlaying
        self.levels_depth = levels_depth

    def game_evaluation(self, game: Game, depth):
        win = game.check_winner()
        ret = 0 + depth
        if win == 0 and self.playerPlaying == 0:
            ret = 100 + depth
        elif win == 0 and self.playerPlaying == 1:
            ret = -100 - depth
        elif win == 1 and self.playerPlaying == 1:
            ret = 100 + depth
        elif win == 1 and self.playerPlaying == 0:
            ret = - 100 - depth
        return ret

    def min_max(self, game: 'Game', alpha, beta, depth):
```

```
        if depth <= 0 or game.check_winner() != -1:
            return self.game_evaluation(game, depth), None
        best_move = None
        if game.current_player == self.playerPlaying:
            for move in get_possible_moves(game, game.get_current_player()):
                tmp = game.get_board()
                g = Game()
                g.set_board(tmp)
                g.current_player = self.playerPlaying
                g._Game__move(move[0], move[1], g.get_current_player())
                g.current_player = 1 - self.playerPlaying
                eval, _ = self.min_max(g, alpha, beta, depth - 1)
                if eval > alpha:
                    alpha = eval
                    best_move = move
                if beta <= alpha:
                    break
            return alpha, best_move
        else:
            for move in get_possible_moves(game, game.get_current_player()):
                tmp = game.get_board()
                g = Game()
                g.set_board(tmp)
                g.current_player = 1 - self.playerPlaying
                g._Game__move(move[0], move[1], g.get_current_player())
                g.current_player = self.playerPlaying
                eval, _ = self.min_max(g, alpha, beta, depth - 1)
                if eval < beta:
                    beta = eval
                    best_move = move
                if beta <= alpha:
                    break
            return beta, best_move

    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        possibleMoves = get_possible_moves(game, game.get_current_player())
        possibleMoveCount = len(possibleMoves)

        depth = 0
        for depthLvl in self.levels_depth:
            if possibleMoveCount > depthLvl[1]:
                depth = depthLvl[0]
            else:
                break

        _, move = self.min_max(game, -math.inf, math.inf, depth)

        return move
```

Utility functions

- `convert_matrix_to_tuple`: Convert the matrix board representation to a tuple representation.
- `get_possible_moves`: Returns a list of possible moves for the player. It returns a list of tuples, where each tuple contains the move coordinates and the move direction. Each move is a tuple of the form (row, column, direction). The direction is a string that can be either 'up', 'down', 'left' or 'right'.
- `test_player`: Test the performance of a player against another player. The method takes as input the two players, the number of games to play, and the name of the two players just to specify them in the plot. The method prints the number of wins for the first player and the number of wins for the second player.
- `plot_total_win_rate`: Creates a bar graph to display the total number of wins of two players in a game. The generated image is saved in the `images` folder.
- `plot_training_trends`: Creates a graph to visualize the trend of total rewards while training a reinforcement learning agent. Within the function, the average of the rewards is calculated every 500 training steps. The generated image is saved in the `images` folder.

```
def convert_matrix_board_to_tuple(board):
    """
    Converts a matrix board to a tuple representation.

    Args:
        board (list): The matrix board to be converted.

    Returns:
        tuple: The converted tuple representation of the board.
    """
    current_board = tuple(tuple(riga) for riga in board)
    return current_board

def get_possible_moves(game: 'Game', player: int) -> list[tuple[tuple[int, int],
Move]]:
    """
    Get a list of possible moves for a given player in the game.

    Args:
        game (Game): The game object representing the current state of the game.
        player (int): The player for whom to find the possible moves.

    Returns:
        list[tuple[tuple[int, int], Move]]: A list of tuples, where each tuple
contains the coordinates of a possible move
        and the corresponding move direction.

    """
    # possible moves:
    # - take border empty and fill the hole by moving in the 3 directions
    # - take one of your blocks on the border and fill the hole by moving in the 3
directions
    # 44 at start possible moves
    pos = set()
    for r in [0, 4]:
        for c in range(5):
            if game.get_board()[r, c] == -1 or game.get_board()[r, c] == player:
```

```
                if r == 0 and c == 0:  # OK
                    pos.add(((c, r), Move.BOTTOM))
                    pos.add(((c, r), Move.RIGHT))
                elif r == 0 and c == 4:  # OK
                    pos.add(((c, r), Move.BOTTOM))
                    pos.add(((c, r), Move.LEFT))
                elif r == 4 and c == 0:  # OK
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.RIGHT))
                elif r == 4 and c == 4:  # OK
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.LEFT))
                elif r == 0:  # OK
                    pos.add(((c, r), Move.BOTTOM))
                    pos.add(((c, r), Move.LEFT))
                    pos.add(((c, r), Move.RIGHT))
                elif r == 4:  # OK
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.LEFT))
                    pos.add(((c, r), Move.RIGHT))
    for c in [0, 4]:
        for r in range(5):
            if game.get_board()[r, c] == -1 or game.get_board()[r, c] == player:
                if r == 0 and c == 0:  # OK
                    pos.add(((c, r), Move.BOTTOM))
                    pos.add(((c, r), Move.RIGHT))
                elif r == 0 and c == 4:  # OK
                    pos.add(((c, r), Move.BOTTOM))
                    pos.add(((c, r), Move.LEFT))
                elif r == 4 and c == 0:  # OK
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.RIGHT))
                elif r == 4 and c == 4:  # OK
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.LEFT))
                elif c == 0:
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.RIGHT))
                    pos.add(((c, r), Move.BOTTOM))
                elif c == 4:
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.LEFT))
                    pos.add(((c, r), Move.BOTTOM))
    return list(pos)

def test_player(player1, player2, num_games, name_player1, name_player2):
    """
    Test the performance of two players in a series of games.

    Parameters:
    player1 (object): The first player object.
    player2 (object): The second player object.
    num_games (int): The number of games to be played.
    name_player1 (str): The name of the first player.
```

```
    name_player2 (str): The name of the second player.

    Returns:
    None
    """
    g = Game()
    player1_wins = 0
    player2_wins = 0
    draws = 0
    games = 0
    for _ in range(num_games):
        winner = g.play(player1, player2)
        games += 1
        g.reset()
        if winner == 0:
            player1_wins += 1
        if winner == 1:
            player2_wins += 1
        if winner == -1:
            draws += 1

    #plot_total_win_rate(player1_wins, player2_wins, draws, name_player1,
name_player2)
    print(f"{name_player1} won {player1_wins / num_games * 100}%")
    print(f"{name_player2} won {player2_wins / num_games * 100}%")
    print(f"Draws: {draws / num_games * 100}%")

def plot_total_win_rate(wins_player1, wins_player2, draws, name_player1,
name_player2):
    """
    Plots the total win rate of two players.

    Parameters:
    - wins_player1 (int): Number of wins for player 1.
    - wins_player2 (int): Number of wins for player 2.
    - draws (int): Number of draws.
    - name_player1 (str): Name of player 1.
    - name_player2 (str): Name of player 2.

    Returns:
    None
    """
    plt.bar([name_player1, name_player2, 'draws'], [wins_player1, wins_player2,
draws], color=['blue', 'orange', 'green'])
    plt.ylabel('Number of games')
    plt.savefig(f"images/{name_player1} wins vs {name_player2} wins.png")
    plt.show()

def plot_training_trends(total_rewards: [int], filename=''):
    """
    Plots the training trends of the total rewards.

    Args:
        total_rewards (list[int]): List of total rewards obtained during training.
```

```
        filename (str, optional): Name of the file to save the plot. Defaults to
''.
    """
    mean_array = np.mean(np.array(total_rewards).reshape(-1, 500), axis=1)
    index = np.arange(0, len(total_rewards), 500)
    plt.plot(index, mean_array, label='Mean rewards value')
    plt.ylabel('Mean rewards value')
    plt.xlabel('Epochs')
    plt.savefig(f"images/{filename} training_trends.png")
    plt.show()
```

Random Player Definition

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

    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        from_pos = (randint(0, 4), randint(0, 4))
        move = choice([Move.TOP, Move.BOTTOM, Move.LEFT, Move.RIGHT])
        return from_pos, move
```

Human Player Definition

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

    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
            available_moves = get_possible_moves(game, game.get_current_player())
            while True:
                row = int(input("Input your action row:"))
                col = int(input("Input your action column:"))
                from_pos = (col, row)
                move = int(input("Input your action move: (1 for top, 2 for
bottom, 3 for left, 4 for right):"))
                if move == 1:
                    move = Move.TOP
                elif move == 2:
                    move = Move.BOTTOM
                elif move == 3:
                    move = Move.LEFT
                elif move == 4:
                    move = Move.RIGHT
                else:
                    print("Invalid move, please input again")
                    continue
```

```
                if (from_pos, move) in available_moves:
                    return from_pos, move
                else:
                    print("Invalid move, please input again")
                    continue
```

Hyperparameters

- `epochs`: training epochs
- `alpha`: learning rate
- `discount_factor`: the discount rate of the Bellman equation
- `min_exploration_rate`: the minimum rate for exploration during the training phase
- `exploration_decay_rate`: the exploration decay rate used during the training
- `training_phase`: a boolean value that indicates if the player is in training phase or not. It basically enables the exploration if it is true, otherwise it uses only the `state_value` dictionary to make decisions.
- `RandomP`: the opponent to play against that use a Random strategy for the RL training
- `MinMaxP`: the opponent to play against that use a MinMax strategy for the RL testing. It takes as input the `level_depth`, a list of tuples, where each tuple contains the depth level and the maximum number of possible moves for that depth level. For example, if there are five possible moves, the depth level is 4, if there are 40 possible moves, the depth level is 1.
- `num_games`: number of games for testing

```
epochs = 500000
alpha = 0.1
discount_factor = 0.95
min_exploration_rate=0.01
exploration_decay_rate=5e-6
training_phase=True
RandomP = RandomPlayer()
MinMaxP = MinMaxPlayer(0, [(4,0),(3,23),(2,28),(1,32)])
num_games = 100
```

Let's do some computation: RL Player trained against Random Player

```
# create the RL player
rl_agent_RandomOpponent = RLPlayer(
    epochs=epochs,
    alpha=alpha,
    discount_factor=discount_factor,
    min_exploration_rate=min_exploration_rate,
    exploration_decay_rate=exploration_decay_rate,
    opponent=RandomP,
    training_phase=training_phase
)
```

```
# train the RL player
rl_agent_RandomOpponent.train(player_name='RL_Player_1')
```

Test Reinforcement Learning Player vs Random Player

```
rl_player = RLPlayer(
    epochs=epochs,
    alpha=alpha,
    discount_factor=discount_factor,
    min_exploration_rate=min_exploration_rate,
    exploration_decay_rate=exploration_decay_rate,
    opponent=RandomP,
    training_phase=False
)

rl_player.load_policy('RL_player_1')
```

```
test_player(rl_player, RandomP, num_games, 'RL_player_1(first_move)', 'Random
Player')
test_player(RandomP, rl_player, num_games, 'Random Player',
'RL_player_1(second_move)')
```

RL Player results

- **RL Player 1**
    - `epochs` = 500_000,
    - `alpha` = 0.1,
    - `discount_factor` = 0.95,
    - `min_exploration_rate` = 0.01,
    - `exploration_decay_rate` = 1e-5,

  Results

    - length of `states_value` dictionary: 3_988_351
    - Last 50000 episodes - Mean rewards value: 6.44
    - win rate vs `RandomPlayer` in 1000 games (RL always first move) - 89%
    - win rate vs `RandomPlayer` in 1000 games (Random always first move) - 72%

- **RL Player 2**
    - `epochs` = 750_000,
    - `alpha` = 0.1,
    - `discount_factor` = 0.95,
    - `min_exploration_rate` = 0.01,

- `exploration_decay_rate` = 5e-6,

Results

- length of `states_value` dictionary: 6_610_522
- Last 50000 episodes - Mean rewards value: 5.78
- win rate vs `RandomPlayer` in 1000 games (RL always first move) - 88%
- win rate vs `RandomPlayer` in 1000 games (Random always first move) - 63%

- **RL Player 3**

  - `epochs` = 850_000,
  - `alpha` = 0.1,
  - `discount_factor` = 0.95,
  - `min_exploration_rate` = 0.01,
  - `exploration_decay_rate` = 5e-6,

  Results

  - length of `states_value` dictionary: 8_727_589
  - Last 50000 episodes - Mean rewards value: 5.39
  - win rate vs `RandomPlayer` in 1000 games (RL always first move) - 83%
  - win rate vs `RandomPlayer` in 1000 games (Random always first move) - 71%

RL Player 1 is the best policy we obtained, and we saved it in `RL_player_1`. We will use it for the next tests.

Test MinMax Player vs Random Player

```
test_player(MinMaxP, RandomP, num_games, 'MinMax Player(first_move)', 'Random
Player')
test_player(RandomP, MinMaxP, num_games, 'Random Player', 'MinMax
Player(second_move)')
```

Results and conclusions

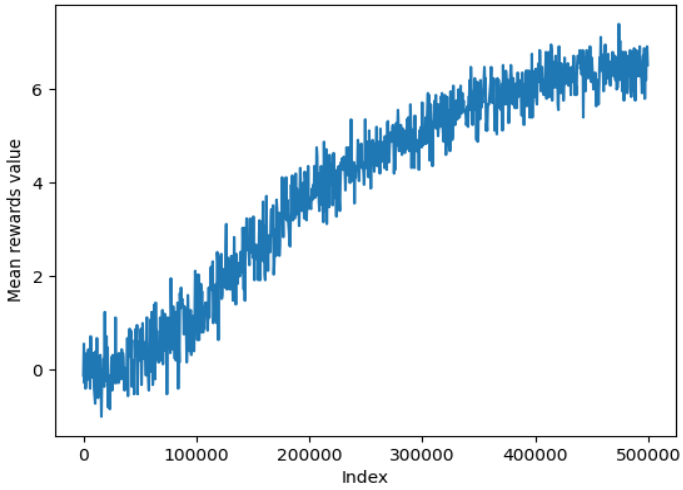
We consider a game as a draw if it lasts more than 200 moves (100 moves for player). This is because the game can last forever. In fact, if the two players play optimally, the game will never end. It happens if we play the MinMax player against the RL player but depends on the train run of the Rl player. In fact, we didn't consider testing the two of them against each other because it's irrelevant, either one player always wins, or the other always wins, or they always draw.

We can see from the results that all our RL players win a decent number of matches against the Random player, but they can't compete against the MinMax player. Our Rl players see only a little part of the state space and to improve them, we should train them for a lot more epochs. We tried to train a player for two million epochs, but the policy obtained was much larger than that of our players and the results were not too distant. This indicates that to obtain a truly high-performance player, we need to reduce the number of keys in the dictionary using symmetries and train it for a long time. This could certainly be the first improvement to be made in the future.

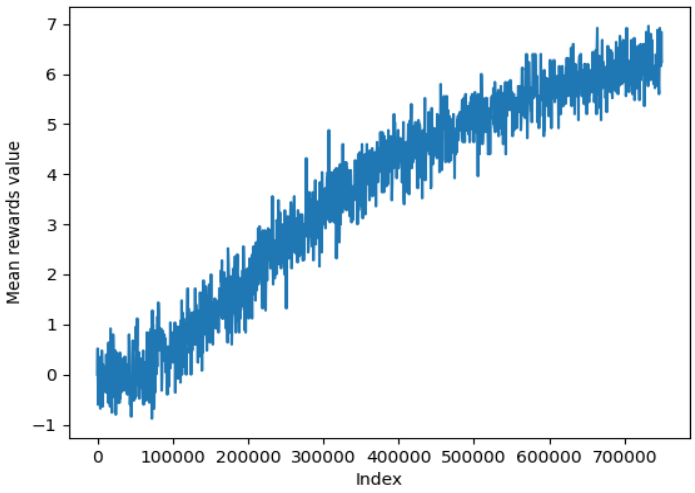
As a conclusion, we can say that the RL player is too much run dependent and the policies that we created are too big, and they do not provide a good result as MinMax.They need also a lot of RAM to play. So, we think that reinforcement Learning is not a good approach for Quixo.

Reinforcement Learning training trends

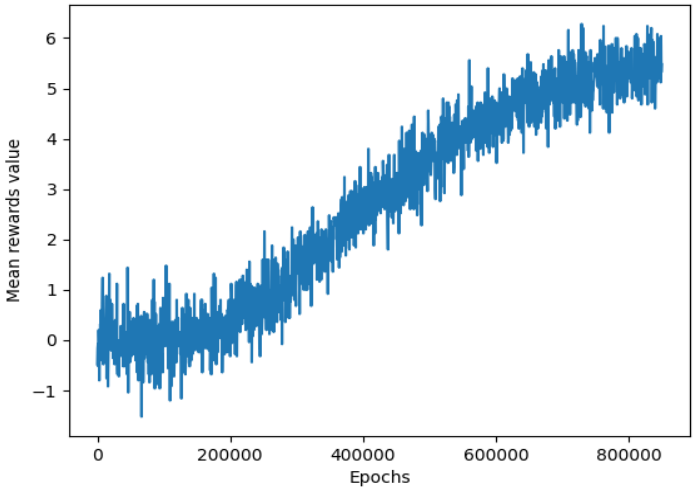
RL Player 1



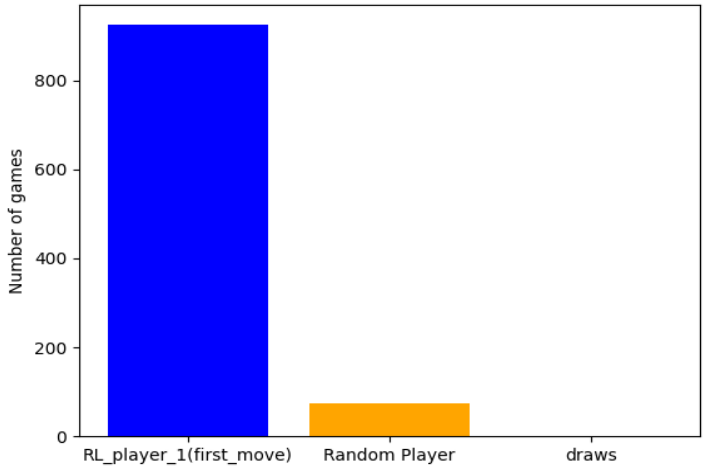
RL Player 2

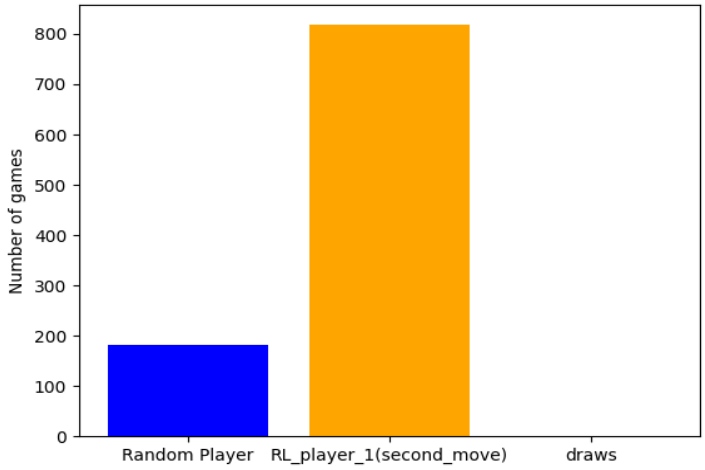


RL Player 3

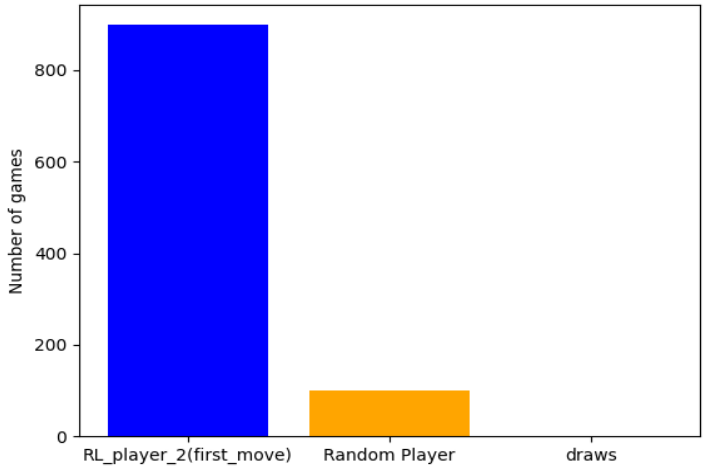


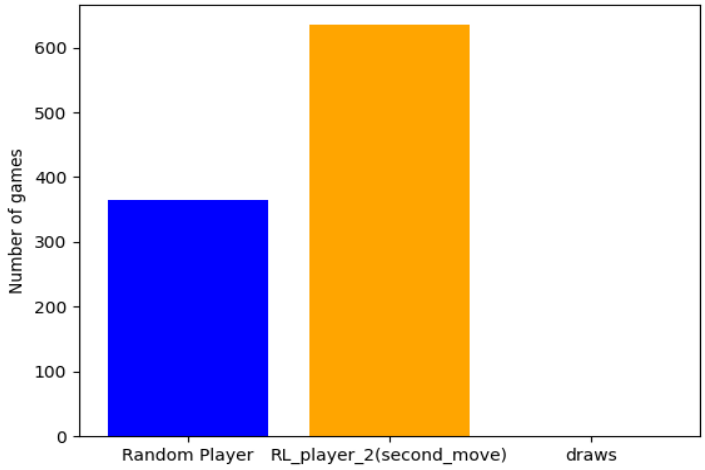
Reinforcement Learning (RL_player_1) vs Random Player



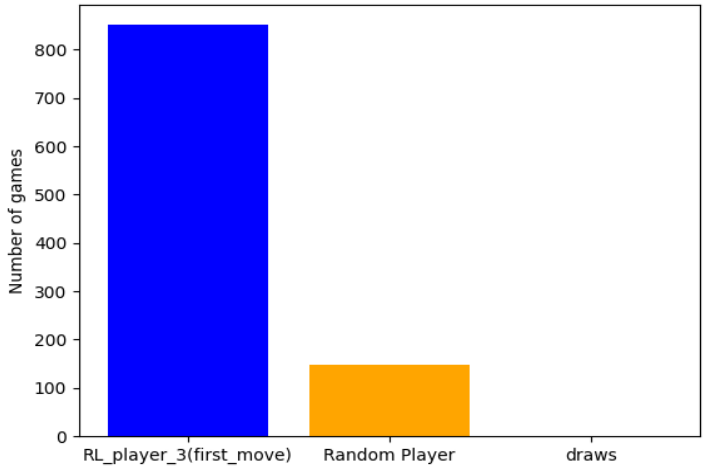


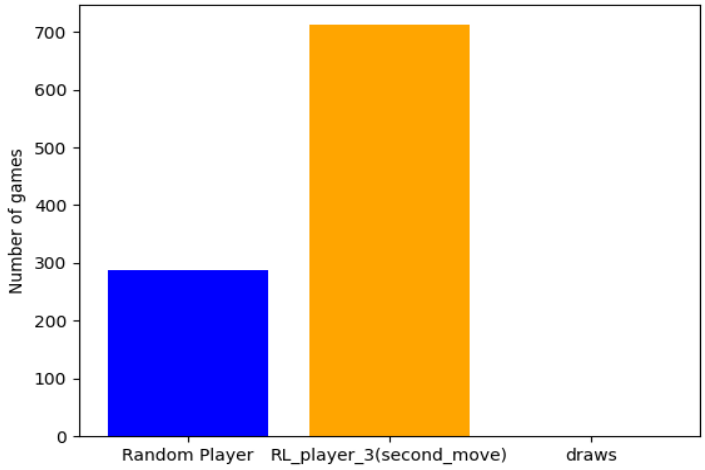
Reinforcement Learning (RL_player_2) vs Random Player



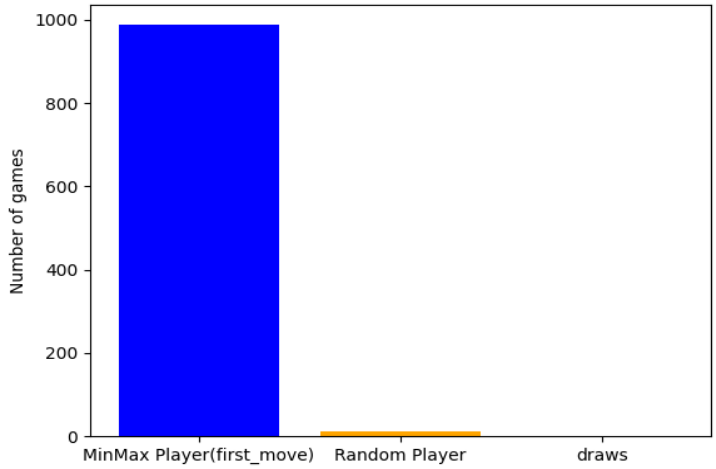


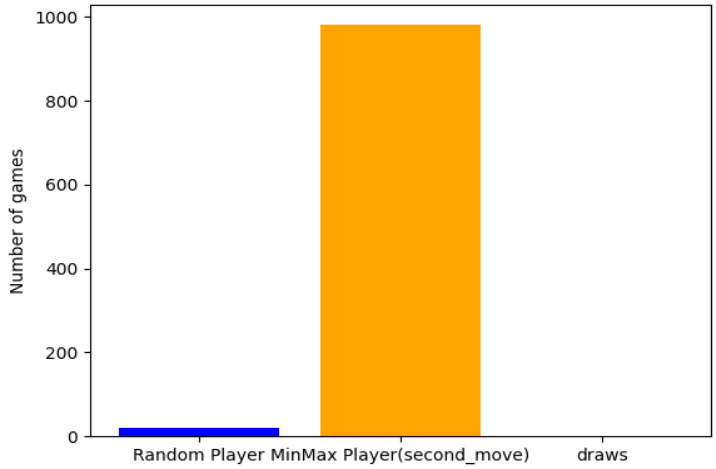
Reinforcement Learning (RL_player_3) vs Random Player





MinMax Player vs Random Player





Let's play!

```
''' board indexes

0
1
2
3
4
  0 1 2 3 4

'''

rl_player = RLPlayer(
    epochs=epochs,
    alpha=alpha,
    discount_factor=discount_factor,
    min_exploration_rate=min_exploration_rate,
    exploration_decay_rate=exploration_decay_rate,
    opponent=RandomP,
    training_phase=False
)
rl_player.load_policy('RL_player_1')

human_player = HumanPlayer()

g = Game()
winner = g.play(rl_player, human_player, print_flag=True)
```

7. Reviews

Lab 02

Done

1. **Peer review for Lab 2 of Luca Sturaro s320062**

> Hi Luca ✌️.
>
> First of all, it would have been easier to evaluate the code if the `README.md` file had included some information about your ES. For example, describe the association between the percentage parameter and the choice of row in `evolutionary_strategy_1`. It seems to me that a low percentage corresponds to prefer the first rows and a high percentage corresponds to prefer the last rows.
>
> I'd like to divide my review into two parts corresponding on your two strategies:
>
> - `evolutionary_strategy_1` and `evolve_1`
> - `evolutionary_strategy_2` and `evolve_2`
>
> ### Relative position strategy
>
> Firstly, In `evolutionary_strategy_1` I think that's simply a typo. You never use the parameter that indicates how many elements from that row to take.
>
> ```
> num_objects = ceil(choice_parameters[0] / 100 * state.rows[row])
> ```
>
> In this solution, if I understand correctly, there is only an individual in the population that is `ev_parameters`. This individual plays with the optimal strategy and, depending on the number of wins, it can become the `best_ev_parameters`. After that, you apply a mutation on that individual and the cycle starts again.
>
> I personally implement a similar solution, where an individual is evaluated by testing him against the optimal algorithm. However, the problem with your solution is that that's not an ES. Without a population and a parent selection, you are doing a local search, and so you might get stuck in a local optimum.
>
> ### Ideal Nim-sum strategy
>
> For this implementation, you used the nim-sum as fitness, although I don't think it's a correct evolution strategy (assuming I understood your intentions). The process involves generating an offspring, in the `evolutionary_strategy_2` function, and verify the fitness for every individual creates, one by one. This offspring, composed of moves, determines the best move to execute based on the nim-sum value as the fitness criterion. A smaller nim-sum indicates a superior move. This approach ensures that the only real-valued parameter undergoing training is the variable `ev_parameter`.

> I expected you to develop an agent with the ability to play independently, eliminating the necessity to generate an offspring solely for executing a single move.
>
> ### Ending
>
> I hope you'll find this review useful and good luck for the next labs 😊!

2. **Peer review for Lab 2 of Alessio Cappello s309450**

> Hi Alessio ✌️.
>
> Firstly, good work 👍! The `README.md` is very well written and therefore was beneficial for quickly understanding your work. I also really appreciated your code style since it is very clear.
>
> ### The code
>
> I hadn't thought about the choice of playing the algorithm against `pure_random`, assuming that the best choice was to send it against the optimal. However, I must say that the results look very good even with `pure_random`. I will definitely try to see if in my solution anything changes.
>
> The only thing I recommend changing is the `fitness` function as your algorithm always starts first against pure_random. It would be better if there was fairness between the algorithms in the number of times one of them starts playing. By the way, I also realized this thanks to the reviews 😈.
>
> ### Ending
>
> After all, that's a great job. Congratulations!. I hope you'll find this review useful and good luck for the next labs 😊!

Received

1. **Peer review for Lab 2 received by Luca Sturaro s320062**

> ### 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.

2. **Peer review for Lab 2 received by Lorenzo Ugoccioni s315734**

> Hi Riccardo,
>
> The ReadME is very simple and easy to understand to get an overview of the algorithm right away. Instead, the comments throughout the code help so much in understanding the various cells,: the code is well organized.
>
> For what I was able to understand, your genome is composed of the probabilities of choosing or not choosing a particular row. Wouldn't it be better to normalize the various values so that the sum of all probabilities is 1?
>
> That said, given that throughout the development of the algorithm you also get to win more than 60 % of games against the optimal, I think the definition of the move and its choice are really effective.
>
> Enjoy Lab9.

Lab 03

Done

1. **Peer review for Lab 3 of Lorenzo Bonannella s317985**

> Hi Lorenzo ✌️.
>
> Firstly, good work 👍! The code is clear, easy to read and also well-structured. It's also a complete code because you have tried a lot of solutions for this problem, nice!
>
> It would have been more comfortable anyway to evaluate the code if the `README.md` file had included some information about it or also with some comments, they are always nice to have.
>
> However, the results section is very well written, also describing the choices made while writing the code.
>
> ### Code
>
> The base EA algorithm is well implemented, so nothing to say about it, well done!
>
> Also for the Diversity, you have gone a great work choosing parents based on how different they are, a nice idea!
>
> ### Ending
>
> I hope you'll find this review useful and good luck for the next labs 😉!

2. **Peer review for Lab 3 of Yalda sadat Mobargha s314700**

> Hi Yalda ✌️. Firstly, good job 👍! Very good also to have added the `README.md` file, it is very useful for immediately understanding the work done without going into detail. The code is pretty simple to read, however, you could use some comments, they are always nice to have.
>
> ### Code
>
> The code you committed doesn't run due to a small error in the `mutate` function. From what I understand, you invert a random individual of the given input population. If you change your genome from a tuple to a simple array and change the function mutate like this, it should work:
>
> ```
> def mutate(genome):
>    index = randint(0, NUM_LOCI-1)
>    genome[index] = 1-genome[index]
>    return genome
> ```
>
> ### Possible improvements
>
> I am of the opinion that incorporating various strategies with diversity and expanding the generational scope could significantly improve outcomes. Moreover, the application of diversity promotion is likely

> to resolve problem instance 1 within the existing generation count.
>
> ### Ending
>
> I hope you'll find this review useful and good luck for the next labs 😊!

Received

1. **Peer review for Lab 3 received by Davide Vitabile s330509**

> Hi Riccardo!
>
> First of all I'd like to compliment you on the work you've done, the code is clear and easy to read. It is also compact and with clear comments. Your implementation of a simple genetic algorithm is well done.
>
> Unfortunately, you don't succeed in solving any instance of the problem, to get better results I would give you some advice.
>
> First, you use a larger population than I did, and a very low number of generations. Of course you need to increase the number of generations to get better results, and you could decrease the number of populations to keep the computational cost acceptable.
>
> Second, with a simple genetic algorithm, you can solve problem instance 1, and maybe the second. But to improve your score for problem instances 5 and 10, you need some implementation of promoting diversity. The simplest and easiest I suggest is extinction, but there are other techniques that the professor has explained, you can test what ever you want.
>
> All in all, good job and good luck for the next lab!

2. **Peer review for Lab 3 received by Davide Sferrazza s326619**

> Hello 👋, Riccardo!
>
> Your notebook is well done. Each section is well documented and the code is easy to read.
>
> ### Code analysis
>
> While reading your code I just noticed a small typo in the select_parent and replacement functions. They take the fitness function as an argument, even though it is never used.
>
> One thing that I really want to highlight is the code that you use to keep track of the best individual among all generations (mentioned below).
>
> ```
> if pop[0][1] > best:
>    best = pop[0][1]
>    n_calls = fit.calls
>    gen = g
> ```
>
> In this case you are also keeping track of the fitness calls made up to this moment, so it is for the generation number. I didn't think of this when I implemented my solution. Great insight 💡! I think I'll

> use this idea for my future labs.
>
> ### Possible improvements
>
> I believe that by implementing some diversity promotion techniques and by increasing the number of generations you should be able to get much better results. Also, by using diversity promotion you will probably be able to solve problem instance 1 with the same number of generations.
>
> ### Conclusions
>
> All in all, good work 😉👍!
>
> I hope you'll find this review useful and good luck for the next labs 🍀😉!

3. **Peer review for Lab 3 received by Florentin-Cristian Udrea (S319029)**

> Hi Riccardo,
>
> I really liked the way you structured your code, it is very clean and understandable. The solution works, and that's good, but I feel you could've added some more to it. At the moment you don't deal with local optimums in any way. One simple way > may be by studying population diversity. If you are in a local optima that you can't escape your population will cease to be diverse, and will converge to the same individual. In that case you could stop early your simulation and save some fitness calls.
>
> I hope I was helpful!

Lab 04

Done

1. **Peer review for Lab 4 of Davide Sferrazza s326619**

> Hi Davide ✌️.
>
> Firstly, good job 👍! Each function is well documented and the code is easy to read. Your players also seem unbeatable, nice! Also, nice job on adding to practically every single line in the notebook a little comment to describe what's happening, it gave me a huge help in understanding the code and therefore saved me some time. I also discovered that it's possible to print the board using some emoji for ❌ and ⭕ and I must say that it turns out really well, I will definitely use it in the future 😌.
>
> ## Code
>
> While reading the code, I just noticed a small typo that's also quite irrelevant but anyway, in the `_game_reward` method you write that the player is a `TicTacToe` class instead of a `QLearningRLPlayer`.
>
> Since you use the state in the form of a string as the dictionary key, it took me a while to understand the whole process between wanting to change the -1 to 2 of the board state in `_map_state_to_index` to the various reshapes in `_make_move` and immediately after to get the `action`. I personally find working with strings quite uncomfortable and often not very intuitive, which is why I opted to transform the board matrix into a tuple of tuples and use it as a key. For example, a matrix [[1,0,0], [0,1,0], [0,0,1]] will become ((1,0,0), (0,1,0), (0,0,1)). In this way, in my opinion, it is slightly easier to work with and also more intuitive.
>
> Changing the exploration rate at each epoch is also excellent. I think this step improves the player and not just a little. I hadn't really thought of this, so thank you very much for this addition, I will definitely use it for the project.
>
> ## Ending
>
> I hope you'll find this review useful and good luck for the project 😌!

2. **Peer review for Lab 4 of Arman Behkish s299525**

> Hi Arman ✌️.
>
> Firstly, good work 👍! You have written a nice and working RL algorithm. The code is also well written, you added a small description to each function plus some comments in the code. This is always well appreciated. You also added a `README` file which explains the strategy used very well so excellent! To make it even more complete, you could have added some results that you obtained while testing your algorithm.
>
> ## Code
>
> To improve your algorithm I can offer you a couple of ideas:

1. I don't know how many different learning rates you've tried, but it's definitely a good starting point to start testing your algorithm with different values and see if anything changes.
2. What can give a big boost to the algorithm is to add more state exploration. The simplest thing that could be done is to ensure that during training there is a probability (maybe 20% or 30%) that the algorithm chooses a random move instead of consulting the `value_dictionary`. To improve even more, you could make the exploration rate start from 1 and lower more and more as the epochs increase.
3. Lastly, you could try adding the discount factor to the Bellman equation. The discount factor essentially determines how much the reinforcement learning agents care about rewards in the distant future relative to those in the immediate future. With this, the formula will be:

```
value_dictionary[hashable_state] = value_dictionary[hashable_state] +
epsilon * (discount_factor * final_reward -
value_dictionary[hashable_state])
```

Ending

I hope you'll find this review useful and good luck for the project 😉!

Received

1. **Peer review for Lab 4 received by Alessandro De Marco s317626**

Hi Riccardo!

I've reviewed both of your Tic Tac Toe implementations using reinforcement learning, and I'm impressed by the thoroughness and clarity of your approaches.

In the first implementation, the TicTacToe class and RLPlayer demonstrate a well-structured design, clearly delineating the roles of the game environment and the learning agent. The use of a reward system that differentiates between wins, losses, and ties is a smart way to encourage more strategic play from the agents. The training process, leveraging two RL agents against each other, is a solid approach to ensure robust learning. Testing against a random player is a great way to validate the learned strategies. It would be interesting to see how the agents perform against more sophisticated opponents or even human players.

The second implementation, focusing on Q-learning, offers a different but equally compelling approach. The QLearningAgent class neatly encapsulates the Q-learning algorithm, and the use of a dictionary for state-action pairs is a good choice for this problem. The training and testing functions are well implemented, providing a clear view of how the agents learn over time. The hyperparameter choices, especially the number of epochs, indicate a deep training process, which should ideally lead to a well-trained model. It would be helpful to have some insights into how these hyperparameters were chosen and tuned.

In both cases, your implementations demonstrate a clear understanding of reinforcement learning concepts applied to a classic game. The structure of your code and the use of classes make the projects readable and maintainable. One suggestion for improvement would be to include more in-depth analysis of the agents' performance over time, such as how quickly they learn or their performance

> against different types of opponents. This could provide more insights into the strengths and weaknesses of your approaches.
>
> Overall, excellent work on both approaches