Computational Intelligence Report

Gregorio Nicora, s310820
February 19, 2024

LAB 1 - A*

Last commit: 26/10/2023

Comment:

The objective was to write A* algorithm implementing g() (the actual cost) and a new h() (heuristic cost): The actual cost g() is computed as the len of sets taken in the current state while the heuristic cost h() is the ratio between uncovered tiles and total number of sets.

In a first analysis we thought it was a good solution, but we ran into some inconvenience:

The first set taken will influence future choices, even leading the algorithm to using more sets than necessary. In fact, the solution identified (also not correctly) as best, will always be the one on top of the Priority Queue.

Unfortunately, we thought this was a parallel solution to what Professor Squillero showed us in class (h3 function) but it turned out not to be so.

Contributing

Made with the contribution of Lorenzo Ugoccioni s315734

Code:

```
from random import random
from math import ceil
from functools import reduce
from collections import namedtuple, deque
from queue import PriorityQueue

import numpy as np
from tqdm.auto import tqdm

State = namedtuple('State', ['taken', 'not_taken'])

def goal_check(state):
    return np.all(covered(state))

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

PROBLEM_SIZE = 500
NUM_SETS = 1000
SETS = tuple(np.array([random()<.3 for _ in range(PROBLEM_SIZE)])for _ in range(NUM_SET
assert goal_check(State(set(range(NUM_SETS)), set())), "Probelm not solvable"

def f(state):
    return g(state) + h(state)

def g(state):
    return PROBLEM_SIZE - sum(covered(state))

def h(state):
    tiles_presi = covered(state)
    free_tiles = min(PROBLEM_SIZE - sum(np.logical_or(tiles_presi, SETS[s])) for s in s
    return free_tiles

frontier = PriorityQueue()
state = State(set(), set(range(NUM_SETS)))      # ({taken}, {not_taken})
frontier.put((f(state), state))       # [ ( {f=g+h}, { {taken}, {not_taken} } ), ]

counter = 0
_ , current_state = frontier.get()
with tqdm(total=None) as pbar:
    while not goal_check(current_state):
```

```
        counter += 1
        for tiles in current_state[1]:
            new_state = State(current_state.taken ^ {tiles}, current_state.not_taken ^
            frontier.put((f(new_state), new_state))
        _ ,current_state = frontier.get()
        pbar.update(1)

print(f"Solved in {counter:,} steps ({len(current_state.taken)} tiles)")
from random import random
from math import ceil
from functools import reduce
from collections import namedtuple, deque
from queue import PriorityQueue

import numpy as np
from tqdm.auto import tqdm
#%%
State = namedtuple('State', ['taken', 'not_taken'])

def goal_check(state):
    return np.all(covered(state))

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

PROBLEM_SIZE = 500
NUM_SETS = 1000
SETS = tuple(np.array([random()<.3 for _ in range(PROBLEM_SIZE)])for _ in range(NUM_SET
assert goal_check(State(set(range(NUM_SETS)), set())), "Probelm not solvable"

def f(state):
    return g(state) + h(state)

def g(state):
    return PROBLEM_SIZE - sum(covered(state))

def h(state):
    tiles_presi = covered(state)
    free_tiles = min(PROBLEM_SIZE - sum(np.logical_or(tiles_presi, SETS[s])) for s in s
    return free_tiles

frontier = PriorityQueue()
state = State(set(), set(range(NUM_SETS)))      # ({taken}, {not_taken})
```

```
frontier.put((f(state), state))       # [ ( {f=g+h}, { {taken}, {not_taken} } ), ]

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

print(f"Solved in {counter:,} steps ({len(current_state.taken)} tiles)")
from random import random
from math import ceil
from functools import reduce
from collections import namedtuple, deque
from queue import PriorityQueue

import numpy as np
from tqdm.auto import tqdm

State = namedtuple('State', ['taken', 'not_taken'])

def goal_check(state):
    return np.all(covered(state))

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

PROBLEM_SIZE = 500
NUM_SETS = 1000
SETS = tuple(np.array([random()<.3 for _ in range(PROBLEM_SIZE)])for _ in range(NUM_SET
assert goal_check(State(set(range(NUM_SETS)), set())), "Probelm not solvable"
#%%
def f(state):
    return g(state) + h(state)

def g(state):
    return PROBLEM_SIZE - sum(covered(state))

def h(state):
```

```
    tiles_presi = covered(state)
    free_tiles = min(PROBLEM_SIZE - sum(np.logical_or(tiles_presi, SETS[s])) for s in s
    return free_tiles

frontier = PriorityQueue()
state = State(set(), set(range(NUM_SETS)))      # ({taken}, {not_taken})
frontier.put((f(state), state))       # [ ( {f=g+h}, { {taken}, {not_taken} } ), ]

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

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

LAB 2 - NIM GAME

Last commit: 17/11/2023

Comment:

This repository contains a Nim game implementation where the agent learns how to play the game using a genetic algorithm. The agent's strategy is defined by a genotype, and its performance is evaluated against a specified opponent.

Nim Game

Nim is a two-player mathematical game where players take turns removing objects from distinct heaps or piles. The game ends when a player removes the last object, and the last player to make a move wins.

Agent

The best agent is selected using an evolutionary strategy. The genotype of the agent defines its strategy, and the evolutionary strategy evolves a population of agents over multiple generations in

order to improve their performance in playing Nim.

Evolutionary strategy Components

- **Initialization:** A population of agents having as a genotype: first an array of probabilities to pick that specific row; second, a matrix representing for each row the probability of picking a certain number of element from that specific row.
- **Selection:** Agents are selected from the population based on their fitness: their success in playing against a specified opponent.
- **Evolution:** Some agents undergo mutation, where the genotype is modified summming random variable obtained by a gaussian mutation with a mutation step (σ=2), re-normalizing them each time to accomplish sum(all probability)=1.
- **Evaluation:** The fitness of each agent is evaluated by playing multiple games (100) of Nim against the opponent (pure_random).

Agent Strategy

The agent strategy consists in taking the most probable number of element from the most probable row.
The behaviour for each agent is encoded in its genotype, which consists of two arrays: first an array of probabilities to pick that specific row; second, a matrix representing for each row the probability of picking a certain number of element from that specific row.

The strategy also includes considerations for edge cases. In fact when a row is empty the probabilities of the other rows are re-normalized. Instead, if some element are already missing from a row (not empty) the probabilities of taking a certain number of elements are re-normalized.

Using a Different Evolution Strategy

We implemented (1+lambda), (1,lambda) and (mu+lambda) strategy.
In the conclusion part, we are taking into consideration only the (mu+lambda) strategy.

Users can also tune our different strategies by adjusting population size (mu), offspring size (lamba), number of generations, mutation step (sigma) and number of row of the game.

Conclusion

The agent with genotype has a winning rate on avarage of 32% against the optimal algorithm.

Contributing

Made with the contribution of Lorenzo Ugoccioni s315734

Code:

```
import logging
from pprint import pprint, pformat
from collections import namedtuple
import random
from copy import deepcopy, copy
from tqdm import tqdm
from matplotlib import pyplot as plt
from dataclasses import dataclass
import numpy as np


Nimply = namedtuple("Nimply", "row, num_objects")
N_ROWS = 5
GAME_DIMENSION = N_ROWS
ELEMENT_PER_ROW = 2*N_ROWS - 1
POPULATION_SIZE = 5             # μ
OFFSPRING_SIZE = 20             # λ
MUTATION_STEP = 2               # σ
NUM_GENERATIONS = 1000


class Nim:
    def __init__(self, num_rows: int, k: int = None) -> None:
        self._rows = [i * 2 + 1 for i in range(num_rows)]
        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 pure_random(state: Nim) -> Nimply:
    """A completely random move"""
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    num_objects = random.randint(1, state.rows[row])
    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])))


def adaptive(state: Nim) -> Nimply:
    """A strategy that can adapt its parameters"""
    genome = {"love_small": 0.5}


import numpy as np


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


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


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

strategy = (optimal, pure_random)

nim = Nim(5)
logging.info(f"init : {nim}")
player = 0
while nim:
    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!")


# ES (1 + λ)
@dataclass
class Individual:
    genotype: list[list[float], list[list[float]]]
    fitness: float

def player_move(ind: Individual, state: Nim) -> Nimply:
    agent = deepcopy(ind)
    rows = [i for i in range(N_ROWS)]
    element = [i for i in range(ELEMENT_PER_ROW)]

    for r, n in enumerate(state.rows):
        if n == 0:
            agent.genotype[0][r] = 0.0

        for j in range(ELEMENT_PER_ROW):
            if j >= n:
                agent.genotype[1][r][j] = 0.0

    agent.genotype[0] = agent.genotype[0]/(np.sum(agent.genotype[0]) + 1e-9)
    agent.genotype[1] = agent.genotype[1]/(np.reshape(np.sum(agent.genotype[1], axis=1)
```

```
    row = random.choices(rows, weights=agent.genotype[0])[0]
    elem = random.choices(element, weights=np.squeeze(agent.genotype[1][row, :]))[0]
    return Nimply(row, elem+1)

def evolve_Individual(ind: Individual) -> list:
    new_population = [deepcopy(ind)]
    for _ in range(OFFSPRING_SIZE):
        offspring = deepcopy(ind)
        mutation_gene_1 = np.random.normal(loc=0, scale=2, size=(N_ROWS))
        new_gene_1 = np.abs(offspring.genotype[0] + mutation_gene_1)
        tot = np.sum(new_gene_1)
        offspring.genotype[0]=np.array(new_gene_1/tot)
        # - - - - - - #
        mutation_gene_2 = np.random.normal(loc=0, scale=2, size=(N_ROWS, ELEMENT_PER_RO
        for r in range(1, N_ROWS+1):
            for i in range(ELEMENT_PER_ROW):
                if i>=2*r-1:
                    mutation_gene_2[r-1, i]=0
        new_gene_2 = np.abs(mutation_gene_2 + ind.genotype[1])
        tot = np.reshape(np.sum(new_gene_2, axis=1), (N_ROWS, 1))
        offspring.genotype[1]=np.array(new_gene_2/tot)

        new_population.append(offspring)

    return new_population

def Evaluation(players) -> float:
    win = 0

    for _ in range(100):
        nim = Nim(GAME_DIMENSION)
        # logging.info(f"init : {nim}")
        player = random.choice([0, 1])
        while nim:
            if player == 0:
                ply = player_move(players[0], nim)
                # logging.info(f"ply: player agent plays {ply}")

            if player == 1:
                ply = players[player](nim)
                # logging.info(f"ply: opponent plays {ply}")
```

```
            nim.nimming(ply)
            # logging.info(f"status: {nim}")
            player = 1 - player
        # logging.info(f"status: { 'Agent'if player==0 else 'Opponent'} won!")
        if player == 0 : win += 1

    return win/100

first_player = Individual(
    genotype=[
        np.array([1/N_ROWS for _ in range(N_ROWS)]),
        np.array([[1/(2*r-1) if i<2*r-1 else 0 for i in range(ELEMENT_PER_ROW) ] for r
        ],
    fitness = 0.0)

logging.getLogger().setLevel(logging.INFO)
solution = first_player
history = list()
best_agent = deepcopy(solution)
# history.append(solution)

opponent = pure_random

for n in tqdm(range(1_000 // OFFSPRING_SIZE)):
    evals = []
    # offspring <- select λ random points mutating the current solution
    offsprings = evolve_Individual(solution)
    # evaluate and select best
    for player in offsprings:
        player.fitness = Evaluation((player, opponent))
        # logging.info(f"Agent {n} won {player.fitness*100} over 100 matches")
        evals.append(player.fitness)

    # max(population, key=lambda i:i.fitness, reverse=True)
    solution = offsprings[np.argmax(evals)]
    logging.info(f"\nBest Agent of {n+1}° generation won {round(solution.fitness*100, 2
    if best_agent.fitness < solution.fitness:
        best_agent = deepcopy(solution)
    history.append(best_agent.fitness)
    logging.info(f"\nBest Agent util now have won {round(best_agent.fitness*100, 2)} ov


logging.info(f"\nBest solution: {best_agent.fitness}\nRows choice: {best_agent.genotype
```

```
history = np.array(history)
generations = list(range(1, 50 + 1))
plt.plot(generations, history, marker='o', linestyle='-', color='r')
plt.title('Fitness Over Generations')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.grid(True)
plt.show()

# ES (μ + λ)

first_population = []

for m in range(5):
    gene_1 = np.array([random.random() for _ in range(N_ROWS)])
    gene_2 = np.array([[random.random() if i<2*r-1 else 0 for i in range(ELEMENT_PER_RO

    first_population.append(Individual(
        genotype=[
            gene_1/np.sum(gene_1),
            gene_2/np.reshape(np.sum(gene_2, axis=1), (N_ROWS, 1))
            ],
        fitness = 0.0))

def mutate_Individual(ind: Individual) -> list:
    offspring = deepcopy(ind)
    mutation_gene_1 = np.random.normal(loc=0, scale=2, size=(N_ROWS))
    new_gene_1 = np.abs(offspring.genotype[0] + mutation_gene_1)
    tot = np.sum(new_gene_1)
    offspring.genotype[0]=np.array(new_gene_1/tot)
    # - - - - - - #
    mutation_gene_2 = np.random.normal(loc=0, scale=2, size=(N_ROWS, ELEMENT_PER_ROW))
    for r in range(1, N_ROWS+1):
        for i in range(ELEMENT_PER_ROW):
            if i>=2*r-1:
                mutation_gene_2[r-1, i]=0
    new_gene_2 = np.abs(mutation_gene_2 + ind.genotype[1])
    tot = np.reshape(np.sum(new_gene_2, axis=1), (N_ROWS, 1))
    offspring.genotype[1]=np.array(new_gene_2/tot)

    return offspring
```

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

current_population = first_population
history = list()
best_so_far = deepcopy(solution)

opponent = pure_random

for n in tqdm(range(50)):
    offsprings = []

    for _ in range(OFFSPRING_SIZE):
        offsprings.append(mutate_Individual(current_population[random.choice(range(POPU

    for player in offsprings:
        player.fitness = Evaluation((player, opponent))

    current_population.extend(offsprings)
    current_population.sort(key=lambda i:i.fitness, reverse=True)       # ORDERING FROM
    current_population = current_population[:POPULATION_SIZE]       # SURVIVAL SELECTIO

    logging.info(f"\nBest Agent of {n+1}° generation won {round(current_population[0].f
    history.append(current_population[0].fitness)

best_agent = copy(current_population[0])
logging.info(f"\nBest solution: {best_agent.fitness}\nRows choice: {best_agent.genotype

history = np.array(history)
generations = list(range(1, 50 + 1))
plt.plot(generations, history, marker='o', linestyle='-', color='r')
plt.title('Fitness Over Generations')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.grid(True)
plt.show()

best_agent = current_population[0]

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

strategy = (best_agent, optimal)
win = 0
```

```
for _ in range(100):
    nim = Nim(GAME_DIMENSION)
    # logging.info(f"init : {nim}")
    player = random.choice([0, 1])
    while nim:
        if player == 0:
            ply = player_move(players[0], nim)
            # logging.info(f"ply: player agent plays {ply}")
        if player == 1:
            ply = players[player](nim)
            # logging.info(f"ply: opponent plays {ply}")
        nim.nimming(ply)
        # logging.info(f"status: {nim}")
        player = 1 - player
    # logging.info(f"status: { 'Agent'if player==0 else 'Opponent'} won!")
    if player == 0 : win += 1

print("\nWin rate: ", win/100)
```

Peer reviews:

To Matteo Martini - s314786:

Hi Matteo!
First of all I wanted to give you a hint for the labs to come: try to write a simple (but effective) readme file in which you explain how your code works. It doesn't have to be very detailed but a high level description would be very useful for those who will read it in the future to do the other peer reviews. Don't get me wrong, you're code was actually pretty easy to understand because it is well-written but that would make it even easier 😃

Getting to your implementation of the evolution strategies: I think that your adaptive strategy is very interesting but I think I didn't understand the fitness function behaviour. You compute the fitness as (tot_games-wins)/tot_games but when you evaluate the best one you take the one that has the maximum fitness value.
The thing that I didn't get is that the best one isn't the one that has the highest fitness value but the one with the lowest one e.g. if an individual wins 100 times its fitness value would be (100-100)/100 = 0 while if it wins 0 games it would be (100-0)/100 = 1.

Let me know if I misunderstood something.
In any case you did a good job, good luck for the next labs!

To Lorenzo Tozzi - s317330:

Hi Lorenzo!
First of all I wanted to give you credit about how easy it was to read your code: both the "in-code" comments and the readme file made it really easy to read.
The only thing that I wanted to understand better is how you evaluate the goodness of an individual exploiting the fitness function.

For what I was able to understand you evaluate the goodness of an individual based on the number of wins it gets against the random strategy using as first move the one encoded in the individual and then playing randomly right? But doesn't this represent a sort of very random selection of what the best individual solution is?
Let me know if I understood it correctly and whether my doubts make sense ahahah.

At the end of the day you managed to achieve great results against the optimal strategy so I guess that you got the job amazingly done!
Good luck for the lab_03 (or 09, I guess 😃)!

LAB 9 - ES

Last commit: 03/12/2023

Comment:

Problem description

The problem consisted in writing 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.
In order to do so, we had to obtain an individual in our population with fitness = 1

Our approach

We started thinking about how we could mantain some diversity among our population.
We decided to implement an algorithm following the island idea.

In order to do so, we generated 10 different population of 15 individuals each and made them evolve (recombination using the crossover function and mutation) separately, exploiting *migration* every 5 "separated" generations.

After each era (= num of generations on separate islands), we selected the 2 best individuals of each island and put them on another island, randomly. After this operation, the separate evolution would take again place and so on.

Initialization

We decided to initialize the population randomly and distributing the individuals on the different islands.

Our results

Unfortunately, we didn't manage to achieve good results. We tried different combinations of NUM_ERAS, NUM_GENERATIONS_PER_ERA and population dimensions but our results were still saturate to a non-optimal fitness value.
We thought the islands approach was a good idea but, looking at our results, we probably got something working not properly.

Problem(1) -> fitness(best individual) = 1.0 with 65950 fitness function calls
Problem(2) -> fitness(best individual) = 0.884 with 150100 fitness function calls (stopped because reached 200 eras)
Problem(5) -> fitness(best individual) = 0.436 with 150100 fitness function calls (stopped because reached 200 eras) (stalled from 39º era)
Problem(10) -> fitness(best individual) = 0.33 with 150100 fitness function calls (stopped because reached 200 eras) (stalled from 69° era)

Contributing

Made with the contribution of Lorenzo Ugoccioni s315734

Code:

lab9.ipynb

```
from random import choices, randint, choice, random
from copy import copy
import lab9_lib
import numpy as np

fitness = lab9_lib.make_problem(2)

# GLOBAL PARAMETER #
NUM_LOCI = 1000
POPULATION_SIZE = 15
NEW_OFFSPRING = 10
TOURNAMENT_SIZE = 2
NUM_ISLANDS = 10
NUM_GENERATIONS = 5
NUM_ERA = 100

MUTATION_PROBABILITY = .25
NUM_MUTATION = 10

def inizialization():
    dim = NUM_LOCI//NUM_ISLANDS
    starting_population = [
        [
            [0 for _ in range(i*dim)] + choices([0, 1], k=dim) + [0 for _ in range((NUM

    return starting_population


def distribute_individuals(populations):
    galapagos = []
    individual_tag = [i for i in range(POPULATION_SIZE*NUM_ISLANDS)]
    for _ in range(NUM_ISLANDS):
        island_population = []
        for _ in range(POPULATION_SIZE):
            tag = choice(individual_tag)
            island_population.append(populations[tag % NUM_ISLANDS][tag % POPULATION_SI
            individual_tag.remove(tag)
        galapagos.append(island_population)
```

```
    return galapagos


def mutation_one(ind):  # 1 loci(gene) mutated
    offspring = copy(ind)
    pos = randint(0, NUM_LOCI - 1)
    offspring[pos] = 1 - offspring[pos]  # Se ho T/F-> offspring[pos] = not offspring[p

    assert len(offspring) == NUM_LOCI
    return offspring


def mutation_n_random(ind):  # 2 loci(gene) mutated
    offspring = copy(ind)
    n = randint(1, NUM_MUTATION+1)
    for _ in range(n):
        pos = randint(0, NUM_LOCI - 1)
        offspring[pos] = 1 - offspring[pos]  # Se ho T/F-> offspring[pos] = not offspri

    assert len(offspring) == NUM_LOCI
    return offspring

def mutation_n(ind):  # Reset randomly a random number of loci
    poss = (randint(0, NUM_LOCI - 1), randint(0, NUM_LOCI - 1))
    offspring = ind[:min(poss)] + [choice([0, 1]) for _ in range(max(poss) - min(poss))

    assert len(offspring) == NUM_LOCI
    return offspring


def mutation_reverse_n(ind):  # Invert a random number of loci
    offspring = copy(ind)
    poss = (randint(0, NUM_LOCI - 1), randint(0, NUM_LOCI - 1))
    offspring = ind[:min(poss)] + [1 - offspring[pos] for pos in range(max(poss) - min(

    assert len(offspring) == NUM_LOCI
    return offspring


def one_cut_crossover(ind1, ind2):
    cut_point = randint(0, NUM_LOCI - 1)
    offspring = ind1[:cut_point] + ind2[cut_point:]
```

```
    assert len(offspring) == NUM_LOCI
    return offspring


def two_cut_crossover(ind1, ind2):
    cut_points = (randint(0, NUM_LOCI - 1), randint(0, NUM_LOCI - 1))
    offspring = ind1[:min(cut_points)] + ind2[min(cut_points): max(cut_points)] + ind1[

    assert len(offspring) == NUM_LOCI
    return offspring


def uniform_crossover(ind1, ind2):
    offspring = [ind1[i] if i % 2 else ind2[i] for i in range(NUM_LOCI)]

    assert len(offspring) == NUM_LOCI
    return offspring


def uniform_crossover_double(ind1, ind2):
    o1, o2 = (
    [ind1[i] if i % 2 else ind2[i] for i in range(NUM_LOCI)], [ind2[i] if i % 2 else in

    assert len(o1) == NUM_LOCI and len(o2) == NUM_LOCI
    return o1, o2

def roulette_wheel(population):  # Roulette wheel with same probability
    return choice(population)[0]


def roulette_wheel_fitness_weight(population):  # Roulette wheel with fitness as weight
    w = [ind[1] for ind in population]
    return choices(population, weights=w, k=1)[0][0]


def static_tournament(population):
    pool = [choice(population) for _ in range(TOURNAMENT_SIZE)]
    champ = max(pool, key=lambda ind: ind[1])
    return champ[0]


def best_parent_ever(population):
    champ = max(population, key=lambda ind: ind[1])
```

```
    return champ[0]


def genotype_distance(population):  # Select the 2 element with most different genotype
    distance = np.array(
        [[sum(np.bitwise_xor(np.array(population[i][0]), np.array(population[j][0]))) f
         for i in range(len(population))]])
    max_position = np.unravel_index(np.argmax(distance, axis=None), distance.shape)

    return population[max_position[0]][0], population[max_position[1]][0]

def survival_selection(population):
    population.sort(key=lambda ind: ind[1], reverse=True)  # ORDERING FROM BEST TO WORS
    return population[:POPULATION_SIZE]  # SURVIVAL SELECTION


def remove_twin(population):  # Remove TWIN from the population because I belive that t
    twins = {j for i in range(len(population)) for j in range(i + 1, len(population)) i
    new_p = [ind for i, ind in enumerate(population) if i not in twins]
    return new_p


def survive_only_the_best(population):
    population.sort(key=lambda ind: ind[1], reverse=True)  # ORDERING FROM BEST TO WORS
    return population[:1]  # SURVIVAL SELECTION

# STANDARD ISLAND MODEL #
recombination_strategy = (one_cut_crossover, two_cut_crossover, uniform_crossover)
parent_selection_strategy = (roulette_wheel, static_tournament, roulette_wheel_fitness_
survival_selection_strategy = (remove_twin, survival_selection)
mutation_strategy = (mutation_one, mutation_n, mutation_n_random, mutation_reverse_n)



def island(population):
    offspring = list()
    for counter in range(NEW_OFFSPRING):
        p1 = static_tournament(population)
        p2 = static_tournament(population)
        o = one_cut_crossover(p1, p2)

        if random() < MUTATION_PROBABILITY:
            # MUTATION
```

```
            o = mutation_one(o)

        offspring.append((o, fitness(o)))

    population.extend(offspring)
    population = remove_twin(population)
    population = survival_selection(population)

    return population


# MIGRATION STRATEGY #
def migration(galapagos):
    migrants = 2
    individual_tag = [i for i in range(migrants*NUM_ISLANDS)]
    galapagos_best = [population[0:migrants] for population in galapagos]

    for ni in range(NUM_ISLANDS):
        chosen = []
        for _ in range(2):
            tag = choice(individual_tag)
            chosen.append(galapagos_best[tag % NUM_ISLANDS][tag % migrants])
            individual_tag.remove(tag)

        galapagos[ni].extend(chosen)

    return galapagos


def check_early_end(ind):
    if ind[1] == 1:
        return True
    else:
        return False

galapagos_population = inizialization()
galapagos_population = distribute_individuals(galapagos_population)
# galapagos_population = [choices(starting_population, k=POPULATION_SIZE) for _ in rang
galapagos_population = [[(ind, fitness(ind)) for ind in population] for population in g

best_ind = []

for era in range(NUM_ERA):
```

```
    # print("New era !")
    if len(best_ind) != 0:
        if check_early_end(best_ind[0]):
            break

    for g in range((era+1)*3):
        # print("New generation !")
        i = 0
        if len(best_ind) != 0:
            if check_early_end(best_ind[0]):
                break

        for p in range(len(galapagos_population)):
            i += 1
            if len(best_ind) != 0:
                if check_early_end(best_ind[0]):
                    break

            # print(f'Isola numero {i}, generazione {g+1}, era {era+1} !')
            galapagos_population[p] = island(galapagos_population[p])

        best_ind = [p[0] for p in galapagos_population]
        best_ind.sort(key=lambda ind: ind[1], reverse=True)

    galapagos_population = migration(galapagos_population)
    print(era, best_ind[0][1], best_ind[0][0])

print(fitness.calls, "\n", best_ind[0][1])
```

Peer reviews:

To Yoan Boute - s321389:

Hello there!
First of all I would like to say that you're README.md file was pretty useful in order to understand your code. What I would suggest is also to write more comments in key parts of it so that it is clearer to realize how it works.
I also implemented the islands approach and I also found that the results are generally worse than the ones that you can get from an "ordinary" EA (if the goal is also to maintain low the number of fitness calls). If you want you can try to improve your idea of the islands by using a variable migration rate i.e. as generations proceed it would be better to do more mutation and recombination on each island

rather than migrating individuals from one island to another. I found this approach to make the island idea slightly better.
Good luck for the next lab and the final project!!

To Michelangelo Caretto - s310178:

Hello there!
I've just finished looking at your code and I have some observations to make to you.
Firstly, the README.md file was really helpful to make the code easier and faster to read, so good job.
Secondly, I found really interesting the different metrics (Hamming distance and entropy) you tried to use in order to maintain some diversity in your population: if I had to suggest other techniques to employ I would say islands and also something else to measure the distance between two different individuals (like a XOR between genomes).
Finally, I would suggest you to let the algorithm run more (ok the goal was to do few fitness calls but I think the primary objective was to find the highest possible fitness value), unless you encountered a stall during runtime.
In any case you did a very good job!
Good luck for the next lab and the final project!!

LAB 10 - TIC TAC TOE

Last commit: 24/12/2023

Comment:

Problem description

The problem consisted in writing an agent (**X player**) which is able to play tic-tac-toe exploiting reinforcement learning tecniques.

Our approach

We adapted the initial solution of Professor Squillero to a solution we found on the internet.
The article can be found at this link.

State

We use as state *photograph* of each player's move at every time a player makes a move. After ending each game we rate the list of moves that our agent has done with a positive rate if it is a winning game.

Action

The action is the actual choice of the move, which is done randomly and evaluated at the end of the game.

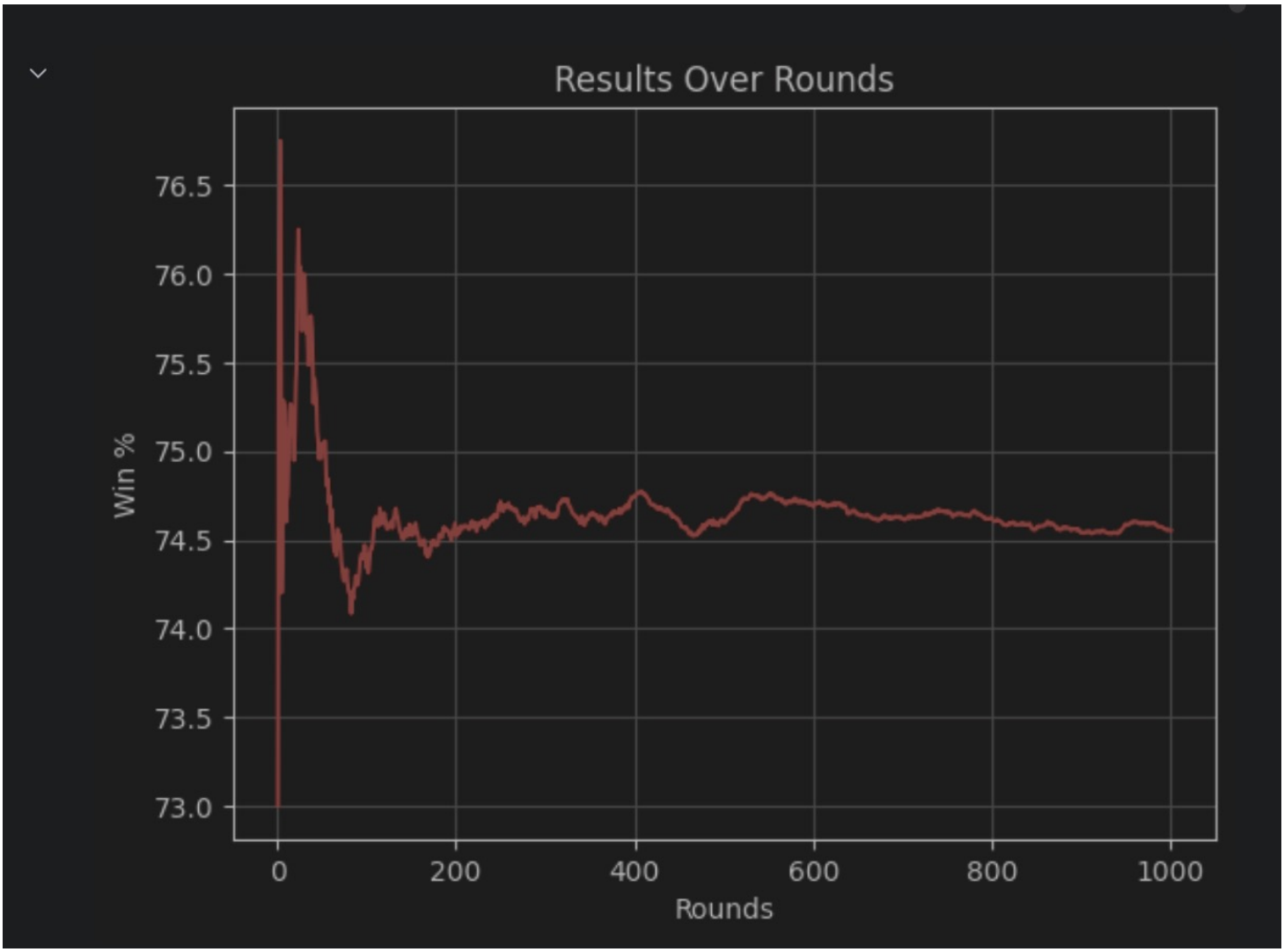
Testing

We decided, in order to have some exploration also at test time, to have a certain probability to make a random move instead of picking the highest rated in the dictionary built at training time. This is done by using the EXP_RATE variable.

It can be seen from the graph in the notebook that we're able to get 74% wins on average.

Our results

We can see that, at the end, we can get decent results when playing against random player.



Contributing

Made with the contribution of Lorenzo Ugoccioni s315734

Code:

```
from itertools import combinations
from collections import namedtuple, defaultdict
from random import choice
from copy import deepcopy

from tqdm.auto import tqdm
import numpy as np
import logging
from matplotlib import pyplot as plt


EXP_RATE = 0.3
NUM_ROUNDS = 1_000
NUM_MATCHES = 100

State = namedtuple('State', ['x', 'o'])

MAGIC = [2, 7, 6, 9, 5, 1, 4, 3, 8]

def print_board(pos):
    """Nicely prints the board"""
    for r in range(3):
        for c in range(3):
            i = r * 3 + c
            if MAGIC[i] in pos.x:
                print('X', end='')
            elif MAGIC[i] in pos.o:
                print('O', end='')
            else:
                print('.', end='')
        print()
    print()

def win(elements):
    """Checks is elements is winning"""
    return any(sum(c) == 15 for c in combinations(elements, 3))

def state_value(pos: State):
    """Evaluate state: +1 first player wins"""
    if win(pos.x):
        return 1
```

```
    elif win(pos.o):
        return -1
    else:
        return 0


def random_game(turn):
    trajectory = list()
    state = State(set(), set())
    available = set(range(1, 9+1))
    while available:
        if turn == 0:
            x = choice(list(available))
            state.x.add(x)
            trajectory.append(deepcopy(state))
            available.remove(x)
            if win(state.x) or not available:
                break
            turn = 1
        else:
            o = choice(list(available))
            state.o.add(o)
            trajectory.append(deepcopy(state))
            available.remove(o)
            if win(state.o):
                break
            turn = 0
    return trajectory

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

for steps in tqdm(range(500_000)):
    turn = choice([0,1])
    trajectory = random_game(turn)
    final_reward = state_value(trajectory[-1])
    for state in trajectory:
        hashable_state = (frozenset(state.x), frozenset(state.o))
        hit_state[hashable_state] += 1
        value_dictionary[hashable_state] = value_dictionary[
            hashable_state
        ] + epsilon * (final_reward - value_dictionary[hashable_state])
```

```
sorted(value_dictionary.items(), key=lambda e: e[1], reverse=True)[:10]

def chooseAction(available_positions, actual_state, value_dictionary):
        if np.random.uniform(0, 1) <= EXP_RATE:
            # take random action
            action = choice(list(available_positions))
        else:
            value_max = -999
            for p in available_positions:
                # next_board = current_board.copy()
                # next_board[p] = symbol
                next_state = deepcopy(actual_state)
                next_state.x.add(p)
                hashable_state = (frozenset(next_state.x), frozenset(next_state.o))

                value = 0 if value_dictionary.get(hashable_state) is None else value_di
                # print("value", value)
                if value >= value_max:
                    value_max = value
                    action = p
        # print("{} takes action {}".format(self.name, action))
        return action

def play(value_dictionary, turn) :
    trajectory = list()
    state = State(set(), set())
    available = set(range(1, 9+1))
    while available:
        if turn == 0:
            x = chooseAction(available, state, value_dictionary)
            state.x.add(x)
            trajectory.append(deepcopy(state))
            available.remove(x)
            if win(state.x) or not available:
                break
            turn = 1
        else:
            o = choice(list(available))
            state.o.add(o)
            trajectory.append(deepcopy(state))
            available.remove(o)
            if win(state.o):
```

```
                break
            turn = 0
    return trajectory

    # print_board(state)


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

history = []
for round in range(NUM_ROUNDS):
    result = [0, 0, 0]          # --> [WINNING, STALL, LOSING]
    n_win=0
    for match in range(NUM_MATCHES):
        turn = choice([0,1])
        winner = state_value(play(value_dictionary, turn)[-1])

        if winner==1:
            result[0] += 1
            n_win += 1
        elif winner==0:
            result[1] += 1
        elif winner==-1:
            result[2] += 1

    history.append(n_win)
    print(f"{n_win} %")
    print(f"Win:{result[0]}\nStall:{result[1]}\nLose:{result[2]}\nNot lose %: {(result[

history = np.array(history)
avg_at_every_round = np.cumsum(history) / (np.arange(len(history)) + 1)
generations = range(1, 1000 + 1)

plt.plot(generations, avg_at_every_round, linestyle='-', color='r')
plt.title('Results Over Rounds')
plt.xlabel('Rounds')
plt.ylabel('Win %')
plt.grid(True)
plt.show()
```

Peer reviews:

To Samaneh Gharehdagh Sani - s309100:

HI!
First of all I wanted to point out that a markdown file could be helpful in order to understand faster what your code is doing. Overall, your code is well-written and I found the strategy pretty straight-forward so GOOD JOB!
Also, adding some results would have been nice 😃
Keep it going!!

To Luca Barbato - s320213:

The code is well written. I found the Markdown file really useful in order top understand the key points of your strategy and it can be seen that you put a good amount of effort into this lab!!
I don't really have much to say except the fact that you did a pretty good job guys!!
Good luck for the final project!!

EXAM PROJECT - QUIXO

Comment:

My player exploits the minimax algorithm as its strategy. In particular i implemented an alpha-beta pruned version of it, to limit the tree exploration width if I'm able to decide before that a particular branch is not gonna be worth exploring as I know I'm going anyway to opt for another, better, decision.
In fact, every time my player has to make a move it evaluates the possible future consequences of its possible moves, given the actual state of the game board, in order to decide which move gives it the highest probability to win.
Minimax, in fact is a strategy in which the player that implements it wants to maximize its probability of winning while assuming that the other player is going to make itself the best decision for it.

In order to do so, the minimax function is a recursive function which explore all the possible future states, given an action and and actual state on which it is performed. In order to be 100% accurate and give certain results, it would need to go in depth of the (action -> state) tree until my player or the opponent wins or there is a draw.

The fact is that quixo is a "possible never ending game" and, also, exploring all the possible outcomes for all the possible moves that my player is going to make in order to give them a certain

score is in practice impossible to realize.

So, what I needed was to first define a way in order to limit the width of my tree, and alpha beta pruning came into rescue. Then I needed an evaluation function that, given a certain non-ending state, gives me anyway a score with a certain probability.

I came up with the idea of an evaluation function that checks obviously if one of hte consequences of my action brings me or the opponent to a win and gives it an according score. Another check that i make, in a non-game-ending state, is by checking if my player or the other one took the central position of the board and i also check whether there are four of my blocks or of the other player aligned whether on a line, column or diagonal, giving according scores. Finally i also check the overall number of blocks my player and the other's in order to see which "controls" more board overall.

To reduce the number of actions that my player or the other are trying to make in my minimax function I limit the search to the only margin of the game board and the only moves that can be effectively done i.e. my player cannot choose a block that has the pther player's sign on it or slide from the top when the chosen block comes from the top of the board.

Finally, I cut the depth of the tree to 2 so I'm evaluating the states only two moves ahead of the current move, obiously if in the meantime some branches haven't reached a winning state.

Results:

Overall, I got a winning rate over the RandomPlayer() of nearly 97% by playing 50% of the times as first and 50% as second on 1000 games.

Code:

main.py

```python
import random
from copy import deepcopy

from game import Game, Move, Player


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

    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        # game.print()
        from_pos = (random.randint(0, 4), random.randint(0, 4))
        move = random.choice([Move.TOP, Move.BOTTOM, Move.LEFT, Move.RIGHT])
        # print(f"{game.current_player_idx} trying move {from_pos}, {move}")
        return from_pos, move


class MyPlayer(Player):
    def __init__(self) -> None:
        super().__init__()
        self.my_player_idx = 0

    def get_my_player_idx(self):
        return self.my_player_idx

    def set_my_player_idx(self, idx):
        self.my_player_idx = idx

    def available_pos(self, game: 'Game'):
        available_pos = []
        # percorro solo i bordi della matrice e seleziono le sole posizioni in cui e' p
        for i in range(0, 5):
            for j in range(0, 5) if (i == 0 or i == 4) else (0, 4):
                if game.get_board()[j][i] == -1 or game.get_board()[j][i] == game.curre
                    available_pos.append((i, j))
        return available_pos

    def available_slides(self, position):
```

```
    available_slide = []
    # data la posizione di pick-up scelta mostro solo le slides che posso fare
    if position[0] == 0 and position[1] == 0:
        available_slide = [Move.BOTTOM, Move.RIGHT]
    elif position[0] == 4 and position[1] == 0:
        available_slide = [Move.BOTTOM, Move.LEFT]
    elif position[0] == 0 and position[1] == 4:
        available_slide = [Move.TOP, Move.RIGHT]
    elif position[0] == 4 and position[1] == 4:
        available_slide = [Move.TOP, Move.LEFT]
    elif position[1] == 0:
        available_slide = [Move.BOTTOM, Move.LEFT, Move.RIGHT]
    elif position[1] == 4:
        available_slide = [Move.TOP, Move.LEFT, Move.RIGHT]
    elif position[0] == 0:
        available_slide = [Move.TOP, Move.BOTTOM, Move.RIGHT]
    elif position[0] == 4:
        available_slide = [Move.TOP, Move.BOTTOM, Move.LEFT]
    return available_slide

def actions(self, game: 'Game'):
    actions = [(x,y) for x in self.available_pos(game) for y in self.available_slid
    return actions

# def blocks_player_zero(self, game: 'Game'):
#     count = 0
#     for i in range(5):
#         for j in range(5):
#             if game.get_board()[i][j] == 0:
#                 count += 1
#     return count

def four_blocks_aligned_my_player(self, game: 'Game'):
    four_blocks_aligned_rows = 0
    for i in range(5):
        for j in range(2):
            if (game.get_board()[i][j] == self.my_player_idx) and (
                    game.get_board()[i][j + 1] == self.my_player_idx) and (
                    game.get_board()[i][j + 2] == self.my_player_idx) and (
                    game.get_board()[i][j + 3] == self.my_player_idx):
                four_blocks_aligned_rows += 1
                break
```

```
        # print(four_blocks_aligned_rows)

        four_blocks_aligned_cols = 0
        for i in range(5):
            for j in range(2):
                if (game.get_board()[j][i] == self.my_player_idx) and (
                        game.get_board()[j + 1][i] == self.my_player_idx) and (
                        game.get_board()[j + 2][i] == self.my_player_idx) and (
                        game.get_board()[j + 3][i] == self.my_player_idx):
                    four_blocks_aligned_cols += 1
                    break

        # print(four_blocks_aligned_cols)

        four_blocks_aligned_diag_princ = 0
        for i in range(2):
            if (game.get_board()[i][i] == self.my_player_idx) and (
                    game.get_board()[i + 1][i + 1] == self.my_player_idx) and (
                    game.get_board()[i + 2][i + 2] == self.my_player_idx) and (
                    game.get_board()[i + 3][i + 3] == self.my_player_idx):
                four_blocks_aligned_diag_princ += 1
                break

        # print(four_blocks_aligned_diag_princ)

        four_blocks_aligned_diag_sec = 0
        for i in range(2):
            if (game.get_board()[i][4 - i] == self.my_player_idx) and (
                    game.get_board()[i + 1][4 - (i + 1)] == self.my_player_idx) and (
                    game.get_board()[i + 2][4 - (i + 2)] == self.my_player_idx) and (
                    game.get_board()[i + 3][4 - (i + 3)] == self.my_player_idx):
                four_blocks_aligned_diag_sec += 1
                break

        # print(four_blocks_aligned_diag_sec)

        return four_blocks_aligned_rows + four_blocks_aligned_cols + four_blocks_aligne

    def four_blocks_aligned_other_player(self, game: 'Game'):
        four_blocks_aligned_rows = 0
        for i in range(5):
            for j in range(2):
                if (game.get_board()[i][j] == (self.my_player_idx+1) % 2) and (
```

```
                game.get_board()[i][j + 1] == (self.my_player_idx+1) % 2) and (
                game.get_board()[i][j + 2] == (self.my_player_idx+1) % 2) and (
                game.get_board()[i][j + 3] == (self.my_player_idx+1) % 2):
            four_blocks_aligned_rows += 1
            break

    # print(four_blocks_aligned_rows)

    four_blocks_aligned_cols = 0
    for i in range(5):
        for j in range(2):
            if (game.get_board()[j][i] == (self.my_player_idx+1) % 2) and (
                    game.get_board()[j + 1][i] == (self.my_player_idx+1) % 2) and (
                    game.get_board()[j + 2][i] == (self.my_player_idx+1) % 2) and (
                    game.get_board()[j + 3][i] == (self.my_player_idx+1) % 2):
                four_blocks_aligned_cols += 1
                break

    # print(four_blocks_aligned_cols)

    four_blocks_aligned_diag_princ = 0
    for i in range(2):
        if (game.get_board()[i][i] == (self.my_player_idx+1) % 2) and (
                game.get_board()[i + 1][i + 1] == (self.my_player_idx+1) % 2) and (
                game.get_board()[i + 2][i + 2] == (self.my_player_idx+1) % 2) and (
                game.get_board()[i + 3][i + 3] == (self.my_player_idx+1) % 2):
            four_blocks_aligned_diag_princ += 1
            break

    # print(four_blocks_aligned_diag_princ)

    four_blocks_aligned_diag_sec = 0
    for i in range(2):
        if (game.get_board()[i][4 - i] == (self.my_player_idx+1) % 2) and (
                game.get_board()[i + 1][4 - (i + 1)] == (self.my_player_idx+1) % 2)
                game.get_board()[i + 2][4 - (i + 2)] == (self.my_player_idx+1) % 2)
                game.get_board()[i + 3][4 - (i + 3)] == (self.my_player_idx+1) % 2)
            four_blocks_aligned_diag_sec += 1
            break

    # print(four_blocks_aligned_diag_sec)

    return four_blocks_aligned_rows + four_blocks_aligned_cols + four_blocks_aligne
```

```
def blocks_my_player(self, game: 'Game'):
    count = 0
    for i in range(5):
        for j in range(5):
            if game.get_board()[i][j] == self.my_player_idx:
                count += 1
    return count

def blocks_other_player(self, game: 'Game'):
    count = 0
    for i in range(5):
        for j in range(5):
            if game.get_board()[i][j] == (self.my_player_idx+1) % 2:
                count += 1
    return count

def evaluation_function(self, game: 'Game'):
    score = 0
    board = game.get_board()

    # punteggio positivo in caso di numero maggiore di blocchi in possesso sulla bo
    if self.blocks_my_player(game) >= self.blocks_other_player(game):
        score += 10
    else:
        score -= 10

    # punteggio positivo in caso il player ottenga la posizione centrale, viceversa
    if board[2][2] == self.my_player_idx:
        score += 10
    if board[2][2] == (self.my_player_idx+1) % 2:
        score -= 10

    # punteggio positivo se player 1 ha allineato 4 caselle
    four_aligned = self.four_blocks_aligned_my_player(game)
    if four_aligned != 0:
         score += 8 * four_aligned

     # punteggio positivo se player 2 ha allineato 4 caselle
    four_aligned = self.four_blocks_aligned_other_player(game)
    if four_aligned != 0:
```

```
            score -= 8 * four_aligned

        # game.print()
        # print(score)

        return score

    # TO BE DEFINED
    def minimax(self, game: 'Game', depth, alpha, beta):
        winner = game.check_winner()
        if depth >= 2 or winner != -1:
            if winner == self.my_player_idx:
                # game.print()
                return 100, depth

            elif winner == (self.my_player_idx+1) % 2:
                # game.print()
                return -100, depth

            elif depth >= 2:
                return self.evaluation_function(game), depth

        actions = self.actions(game)

        # maximizing player
        if game.current_player_idx == self.my_player_idx:
            best_score = -10_000
            best_depth = 10
            for a in actions:
                game_copied = deepcopy(game)
                game_copied._Game__move(a[0], a[1], game_copied.current_player_idx)
                game_copied.current_player_idx = (self.my_player_idx+1) % 2
                score, curr_depth = self.minimax(game_copied, depth+1, alpha, beta)
                if score > best_score or (score == best_score and curr_depth < best_dep
                    best_score = score
                    best_action = a
                    best_depth = curr_depth
                alpha = max(alpha, score)
                if beta <= alpha:
                    break

        # minimizing player
        if game.current_player_idx == (self.my_player_idx+1) % 2:
```

```
            best_score = 10_000
            best_depth = 10
            for a in actions:
                game_copied = deepcopy(game)
                game_copied._Game__move(a[0], a[1], game_copied.current_player_idx)
                game_copied.current_player_idx = self.my_player_idx
                score, curr_depth = self.minimax(game_copied, depth+1, alpha, beta)
                if score < best_score or (score == best_score and curr_depth < best_dep
                    best_score = score
                    best_action = a
                    best_depth = curr_depth
                beta = min(beta, score)
                if beta <= alpha:
                    break

        # print(depth)
        if depth == 0:
            return best_score, best_action
        else:
            return best_score, depth

    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        # game.print()
        # actions = self.actions(game)
        # print(actions)
        score, chosen_action = self.minimax(game, 0, -10_000, 10_000)

        # chosen_action = random.choice(actions)
        from_pos, move = chosen_action[0], chosen_action[1]

        print(f"{game.current_player_idx} trying move {from_pos}, {move} with score {sc
        return from_pos, move


if __name__ == '__main__':

    my_player = MyPlayer()
    other_player = RandomPlayer()

    '''
    g = Game()
    winner = g.play(player1, player2)
    print(f"Player {winner} won!")
```

```
'''

count = 0.0
count_starting_first = 0
count_starting_second = 0
num_games = 100
for i in range(num_games):
    g = Game()
    g.print()

    my_player_idx = random.choice([0, 1])
    my_player.set_my_player_idx(my_player_idx)

    if my_player_idx == 0:
        count_starting_first += 1
        winner = g.play(my_player, other_player)
    else:
        count_starting_second += 1
        winner = g.play(other_player, my_player)

    print(f"Player {winner} won!")
    if winner == my_player.get_my_player_idx():
        count += 1.0

print(f"My player winning rate = {count*100.0/num_games} % ! \nStarting {count_star
```