Computational Intelligence 2023/2024 - Final Report

The aim of this report is to create a log-file to explain the work I did during the Computational Intelligence course.

LAB 1 - A*

- The goal of this lab is to solve the “Set Covering Problem” using the A* algorithm.
- Given a template of the problem, we need to build a new H function
- In the code, three different versions of the H function, have been developed
- You can see useful comments in the code

Solution

```
from random import random
from functools import reduce
from collections import namedtuple
from queue import PriorityQueue
from math import ceil
import numpy as np


PROBLEM_SIZE = 15
NUM_SETS = 25
SETS = tuple(
np.array([random() < 0.3 for _ in range(PROBLEM_SIZE)])
for _ in range(NUM_SETS)
)
State = namedtuple('State', ['taken', 'not_taken'])

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

```python
    def g(state):
    return len(state.taken)

# number of positions to cover to reach the goal
def h1(state):
    return PROBLEM_SIZE - sum(
        covered(state))


def h2(state):
    covered_tiles = sum(covered(state))
    if covered_tiles == PROBLEM_SIZE:
        return 0
    return 1 / covered_tiles if covered_tiles != 0 else 1


# We only considered the sets not taken,
# so as not to be influenced by the existence of large sets which have already
been taken
def h3(state):
    not_taken_sets = [s for i, s in enumerate(SETS) if i not in state.taken]
    largest_set_size = max(sum(s) for s in not_taken_sets) # select the larget
tiles (more number of true)
    missing_size = PROBLEM_SIZE - sum(covered(state)) # evaluates the number of
tiles that are not covered
    optimistic_estimate = ceil(missing_size / largest_set_size) # estimate the
number of set that are missing for the solution in a optimistic way
    # if the largest set is 5 and the missing size is 10 --> "maybe" 2 sets are
missing (optimistic assumption)
    return optimistic_estimate


def f1(state):
    cost_1 = g(state)
    cost_2 = h1(state)

    return cost_1 + cost_2

# since h2 is a value between 0 and 1, we multiply it by 0.1 to make it more
significant
def f2(state):
    cost_1 = 0.1*g(state)
    cost_2 = h2(state)

    return cost_1 + cost_2

def f3(state):
    cost_1 = g(state)
    cost_2 = h3(state)
```

```python
    return cost_1 + cost_2

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

# SOLUTION WITH H1
frontier = PriorityQueue()
state = State(set(), set(range(NUM_SETS)))
frontier.put((f1(state), state))

counter = 0
_, current_state = frontier.get()
while not goal_check(current_state):
    counter += 1
    for action in current_state[1]:
        new_state = State(
            current_state.taken ^ {action},
            current_state.not_taken ^ {action},
        )
        frontier.put((f1(new_state), new_state))
    _, current_state = frontier.get()

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

Result: “Solved in 3 steps (3 tiles)”

```python
#SOLUTION WITH H2
frontier = PriorityQueue()
state = State(set(), set(range(NUM_SETS)))
frontier.put((f2(state), state))

counter = 0
_, current_state = frontier.get()
while not goal_check(current_state):
    counter += 1
    for action in current_state[1]:
        new_state = State(
            current_state.taken ^ {action},
            current_state.not_taken ^ {action},
        )
        frontier.put((f2(new_state), new_state))
    _, current_state = frontier.get()

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

Result: "Solved in 47 steps (3 tiles)"

```
#SOLUTION WITH H3
frontier = PriorityQueue()
state = State(set(), set(range(NUM_SETS)))
frontier.put((f3(state), state))

counter = 0
_, current_state = frontier.get()
while not goal_check(current_state):
    counter += 1
    for action in current_state[1]:
        new_state = State(
            current_state.taken ^ {action},
            current_state.not_taken ^ {action},
        )
        frontier.put((f3(new_state), new_state))
    _, current_state = frontier.get()

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

Result: "Solved in 412 steps (3 tiles)"

- *Collaborations*: Worked with Angelo Iannielli - s317887
- Peer Reviews: Not requested for this Lab

Halloween Challenge - Set Covering

The aim of this challenge is to obtain the best results on a set covering problem with differrent algorithm, minimizing the number of calls to the fitness function

Two proposed solutions:

- Simulated Annealing: it is just an attempt but not properly working
- Tabu Search: results shown later

Solution

```python
from itertools import product
import numpy as np
from scipy import sparse
from random import random, choice, randint, seed
from functools import reduce
from copy import copy
import math
import matplotlib.pyplot as plt

num_points = [100, 1_000, 5_000]
num_sets = num_points
density = [0.3, 0.7]

points = num_points[1]
sets = num_sets[1]
den = density[1]
iterations = 4000

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

SETS = make_set_covering_problem(points, sets, den)
```

Hill Climbing

```
# Taken from Giovanni Squillero's notebook on Github
def evaluate(state):
    cost = sum(state)
    valid = np.all(
        reduce(
            np.logical_or,
            [SETS.getrow(i).toarray().flatten() for i, t in enumerate(state) if
t],
            np.array([False for _ in range(points)]),
        )
    )
    return valid, -cost if valid else 0

def tweak(state):
    new_state = copy(state)
    index = randint(0, sets - 1)
    new_state[index] = not new_state[index]

    return new_state

#current_state = [choice([True, False]) for _ in range(sets)]
current_state = [choice([False]) for _ in range(sets)]
taken_sets = []

iteration_sets = []
for step in range(iterations):
    new_state = tweak(current_state)
    if evaluate(new_state) >= evaluate(current_state):
        current_state = new_state
        # print(current_state, evaluate(current_state))
        taken_sets.append(-evaluate(current_state)[1])
        iteration_sets.append(step)
        print("Step: " + str(step) + " Current state: " +
str(evaluate(current_state)))

print("Final state:", evaluate(current_state))

plt.plot(iteration_sets, taken_sets)
plt.xlabel("Iterations")
plt.ylabel("Taken Sets")
plt.show()
```

Simulated Annealing

```
def acceptance_probability(current_solution, tweaked_solution, temp):
    x = -abs(current_solution[1] - tweaked_solution[1]) / temp
    return math.exp(x)

#current_state = [choice([True, False]) for _ in range(sets)]
current_state = [choice([False]) for _ in range(sets)]
```

```
temp_array = []
probability_array = []
taken_sets = []
iteration_sets = []

for step in range(iterations):
    new_state = tweak(current_state)
    temp = iterations / (5 * step + 1)
    temp_array.append(temp)
    p = acceptance_probability(evaluate(current_state), evaluate(new_state), temp)
    probability_array.append(p)

    if evaluate(new_state) >= evaluate(current_state) or random() < p:
        current_state = new_state
        # print(current_state, evaluate(current_state))
        taken_sets.append(-evaluate(current_state)[1])
        iteration_sets.append(step)
        print("Step: " + str(step) + " Current state: " +
str(evaluate(current_state)))

print("Final state:", evaluate(current_state))

plt.plot(iteration_sets, taken_sets)
plt.xlabel("Iterations")
plt.ylabel("Taken Sets")
plt.show()

plt.plot(range(iterations), temp_array)
plt.xlabel("Iterazioni")
plt.ylabel("Temperature")
plt.show()

plt.plot(range(iterations), probability_array)
plt.xlabel("Iterations")
plt.ylabel("Acceptance Probability")
plt.show()
```

Tabu Search

```
temperature = 1000
cooling_rate = 0.8
taboo_list = []
temp_array = []
iteration_sets = []
probability_array = []

def find_greatest_set(x):
    return x.sum(axis=1).argmax()


def evaluate_2(state):
    cost = sum(state)
```

```python
    elem_covered = reduce(
        np.logical_or,
        [SETS.getrow(i).toarray() for i, t in enumerate(state) if t],
        np.array([False for _ in range(points)]),
    )

    valid = np.all(elem_covered)

    num_elem_covered = np.count_nonzero(elem_covered)

    return valid, num_elem_covered, -cost


def tweak_2(state):
    new_state = copy(state)

    while new_state in taboo_list:
        index = randint(0, sets - 1)
        new_state[index] = not new_state[index]

    taboo_list.append(new_state)
    return new_state


## Initialize the taboo list
taboo_list.clear()

## Find the set that cover the most num of elements and use it as starting point
current_solution = [False] * sets
current_solution[find_greatest_set(SETS)] = True
current_cost = evaluate_2(current_solution)

# Memorize that as the best solution for the moment
best_solution = [True] * sets
best_cost = (True, points, -sets)

# Insert the starting point into taboo list
taboo_list.append(current_solution)

for step in range(iterations):
    # Find a new possible solution
    new_state = tweak_2(current_solution)
    # print(new_state)

    # Evaluate the cost
    new_cost = evaluate_2(new_state)
    print(new_cost)

    # Calculate deltaE using the number of taken elements
```

```python
    deltaE = - ( new_cost[1] - current_cost[1] )
    print(deltaE)

    if deltaE == 0:
        # Calculate deltaE using the number of taken sets
        deltaE = - ( new_cost[2] - current_cost[2] )

    # The solution is better
    if deltaE < 0:
        current_solution = new_state
        current_cost = new_cost

        if current_cost[2] > best_cost[2] and current_cost[0] == True:
            best_solution = current_solution
            best_cost = current_cost
    else:
        probability = math.exp(-deltaE / temperature)
        probability_array.append(probability)

        if random() < probability:
                current_solution = new_state
                current_cost = new_cost

    temperature *= cooling_rate
    temp_array.append(temperature)
    iteration_sets.append(step)
```

Results: (only for Tabu Search since it was the best)

size	density	Best result	Calls to solution	Total calls
100	0.3	-7	504	1000
1000	0.3	-15	991	1000
5000	0.3	-21	545	1000
100	0.7	-3	460	1000
1000	0.7	-6	4	1000
5000	0.7	-7	5	1000

- *Collaborations*: Worked with Angelo Iannielli - s317887
- Peer Reviews: Not requested for the challenge.

LAB 2 - NIM with ES

The goal of this lab is to 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

Solution

```python
import logging
from pprint import pprint, pformat
from typing import Callable
from collections import namedtuple
import random
from copy import deepcopy
import matplotlib.pyplot as plt
import random
import numpy as np

NUM_ROWS = 5
K = None
NUM_MATCHES = 200
λ = 20
σ = 0.1
GENERATION_SIZE = 500 // λ
random.seed(42)

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

class Nim:
    def __init__(self, num_rows: int, k: int = None) -> None:
        # Initialize the Nim object with given number of rows and an optional
maximum object limit
        self._rows = [
            i * 2 + 1 for i in range(num_rows)
        ]  # Create a list of odd numbers as row sizes
        self._k = k  # Store the maximum object limit

    def __bool__(self):
        # Return True if there are objects remaining in the game, False otherwise
        return sum(self._rows) > 0

    def __str__(self):
        # Return a string representation of the object
        return "<" + " ".join(str(_) for _ in self._rows) + ">"

    @property
```

```
    def rows(self) -> tuple:
        # Return the rows as a tuple
        return tuple(self._rows)

    def nimming(self, ply: Nimply) -> None:
        # Perform a nimming move by removing objects from a specified row
        row, num_objects = ply  # Unpack the tuple
        assert (
            self._rows[row] >= num_objects
        )  # Check if the specified row has enough objects
        assert (
            self._k is None or num_objects <= self._k
        )  # Check if the number of objects is within the maximum limit
        self._rows[
            row
        ] -= num_objects  # Subtract the number of objects from the specified row

def pure_random(state: Nim) -> Nimply:
    """A completely random move"""
    # Select a row that has at least one object remaining
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    # Randomly choose a number of objects to remove from the selected row
    num_objects = random.randint(1, state.rows[row])
    # Create and return a Nimply object representing the chosen move
    return Nimply(row, num_objects)


def gabriele(state: Nim) -> Nimply:
    """Pick always the maximum possible number of the lowest row"""
    # Generate a list of possible moves
    possible_moves = [(r, o) for r, c in enumerate(state.rows) for o in range(1, c
+ 1)]
    # Select the move with the maximum number of objects from the lowest row
    return Nimply(*max(possible_moves, key=lambda m: (-m[0], m[1])))

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
```

```python
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


def state_info(state: Nim) -> dict:
    info = dict()
    info["possible_moves"] = [
        (r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1)
    ]
    info["shortest_row"] = min(
        (x for x in enumerate(state.rows) if x[1] > 0), key=lambda y: y[1]
    )[0]
    info["longest_row"] = max((x for x in enumerate(state.rows)), key=lambda y: y[1])[0]
    info["random_row"] = random.choice([r for r, c in enumerate(state.rows) if c > 0])


    return info
```

Evolved Strategy

In the evolved strategy we introduce 5 different kind of strategies (i.e. way to choose an action in the game)

- “shortest”: chooses the shortest row and removes a random number of objects from it.
- “longest”: chooses the longest row and removes a random number of objects from it.
- “random”: chooses a random row and removes a random number of objects from it.
- “half_random”: chooses a random row and removes half of the objects from it, rounded up.
- “one_random”: chooses a random row and removes only one object from it.

An individual is a set of probabilities to choose one of the previous strategy. The algorithm selects the best individuals adapting the probabilities.

```python
def evolved_strategy(genome) -> Callable:
    strategy_dict = {0: "shortest", 1: "longest", 2: "random", 3: "half_random", 4: "one_random"}

    def adaptive(state: Nim) -> Nimply:
        data = state_info(state)

        # select a strategy in a random way wheighed by the genome
```

```python
        selected_strategy = random.choices(range(len(genome)), weights=genome)[0]
        selected_strategy = strategy_dict[selected_strategy]
        if selected_strategy == "shortest":
            ply = Nimply(
                data["shortest_row"],
                random.randint(1, state.rows[data["shortest_row"]]),
            )
        elif selected_strategy == "longest":
            ply = Nimply(
                data["longest_row"], random.randint(1,
state.rows[data["longest_row"]])
            )
        elif selected_strategy == "random":
            ply = Nimply(
                data["random_row"], random.randint(1,
state.rows[data["random_row"]])
            )
        elif selected_strategy == "half_random":
            ply = Nimply(data["random_row"], (state.rows[data["random_row"]] // 2
+ 1))

        elif selected_strategy == "one_random":
            ply = Nimply(data["random_row"], 1)
        # else:
        #     ply = optimal(state)
        return ply

    return adaptive
```

The fitness function is evaluated as the percentage of victories of the individual against a player that plays with the optimal strategy

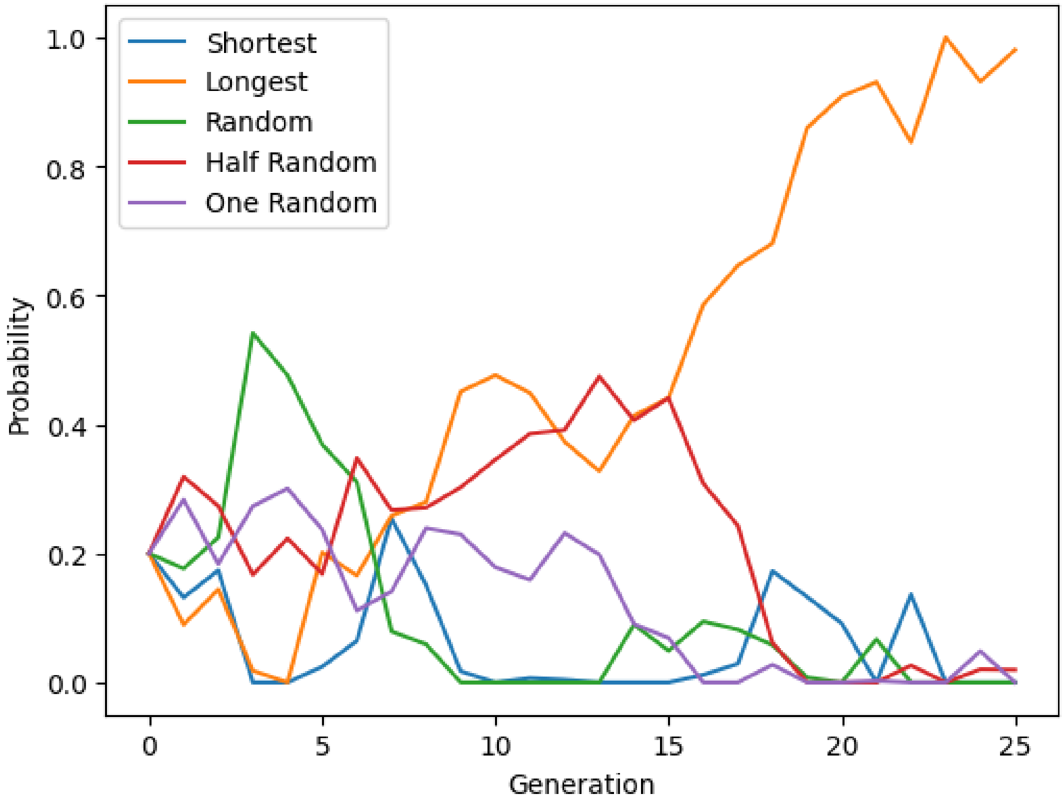
```python
# In the fitness function we play Nim for NUM_MATCHES times where the player is:
# adaptive: for each move, choose a rule in a random way wheighed by the genome
# optimal: choose the optimal move


# Since "optimal" strategy is our upper bound, we can find the best individual
among population by comparing it with an individual that plays always with the
optimal solution

def fitness(adaptive: Callable) -> int:
    won = 0
    opponent = (adaptive, optimal)

    for _ in range(NUM_MATCHES):
        nim = Nim(NUM_ROWS)
        player = 0
        while nim:
            ply = opponent[player](nim)
            nim.nimming(ply)  # perform the move
```

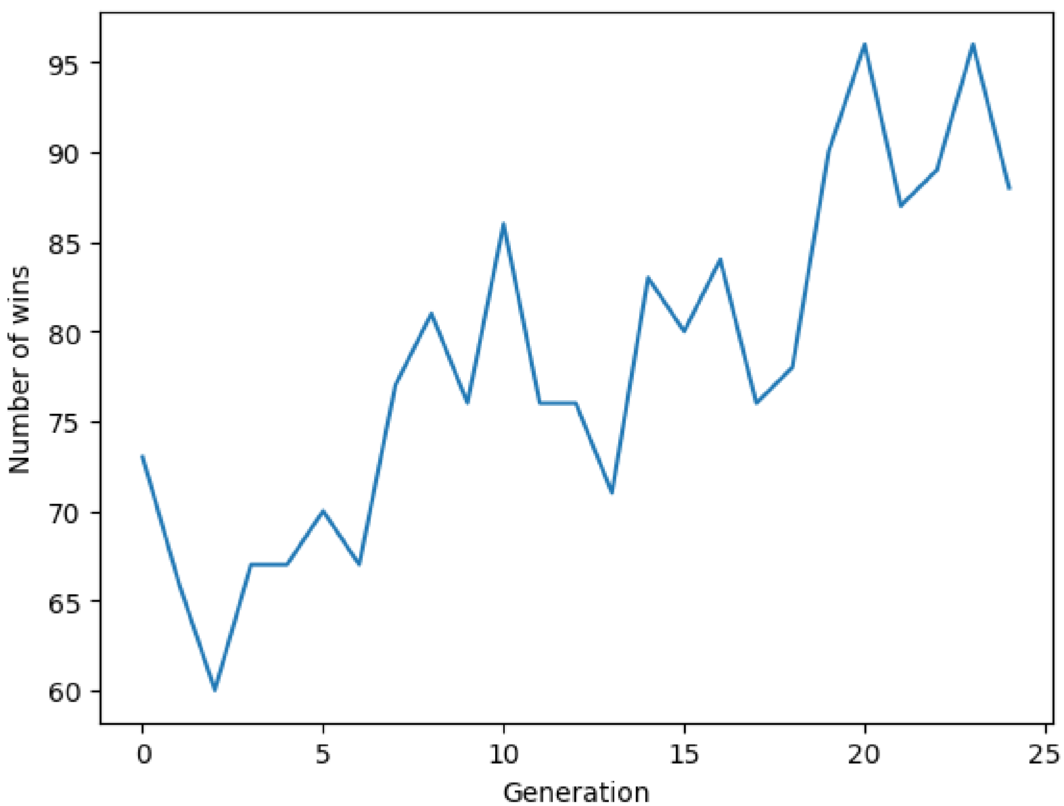
```
            player ^= 1

        if player == 0:
            won += 1

    return won  # return the number of matches won

def generate_offsprings(offspring) -> list:
    output = []

    for _ in range(λ):
        new_offspring = [
            np.clip(val + np.random.normal(0, σ), 0, 1) for val in offspring
        ]

        current_sum = sum(new_offspring)

# Normalize the sum to 1 if it is not already
        if current_sum != 1:
            scale_factor = 1 / current_sum
            # Apply scale factor to each value
            values = [val * scale_factor for val in new_offspring]
        else:
            values = new_offspring

        output.append(values)

    return output
```

(1,λ)-ES

```
current_solution = (0.20, 0.20, 0.20, 0.20, 0.20)


choosen_probability = list()
solutions_list = list()
for n in range(GENERATION_SIZE):
    # offspring <- select λ random points mutating the current solution
    # print("Starting probability for generation", n+1, "is:", current_solution)
    offsprings = generate_offsprings(current_solution)
    offsprings.append(current_solution)
    # evaluate and select best

    evals = [
        (offspring, fitness(evolved_strategy(offspring))) for offspring in 
offsprings
    ]

    evals.sort(key=lambda x: x[1], reverse=True)
    # pprint(evals)
```

```python
    current_solution = evals[0][0]
    choosen_probability.append(current_solution)
    solutions_list.append(evals[0][1])

    print(f"Best result for generation {n+1} is:", evals[0])


val = np.array([[0.20], [0.20], [0.20], [0.20], [0.20]])
curve_names = ["Shortest", "Longest", "Random", "Half Random", "One Random"]
choosen_probability = np.array(choosen_probability)

for i in range(GENERATION_SIZE):
    val = np.hstack((val, choosen_probability[i].reshape(-1, 1)))

for i in range(5):
    plt.plot(range(GENERATION_SIZE + 1), val[i], label=curve_names[i])

plt.xlabel("Generation")
plt.ylabel("Probability")
plt.legend()
plt.show()


plt.plot(range(GENERATION_SIZE), solutions_list)
plt.xlabel("Generation")
plt.ylabel("Number of wins")
plt.show()
```

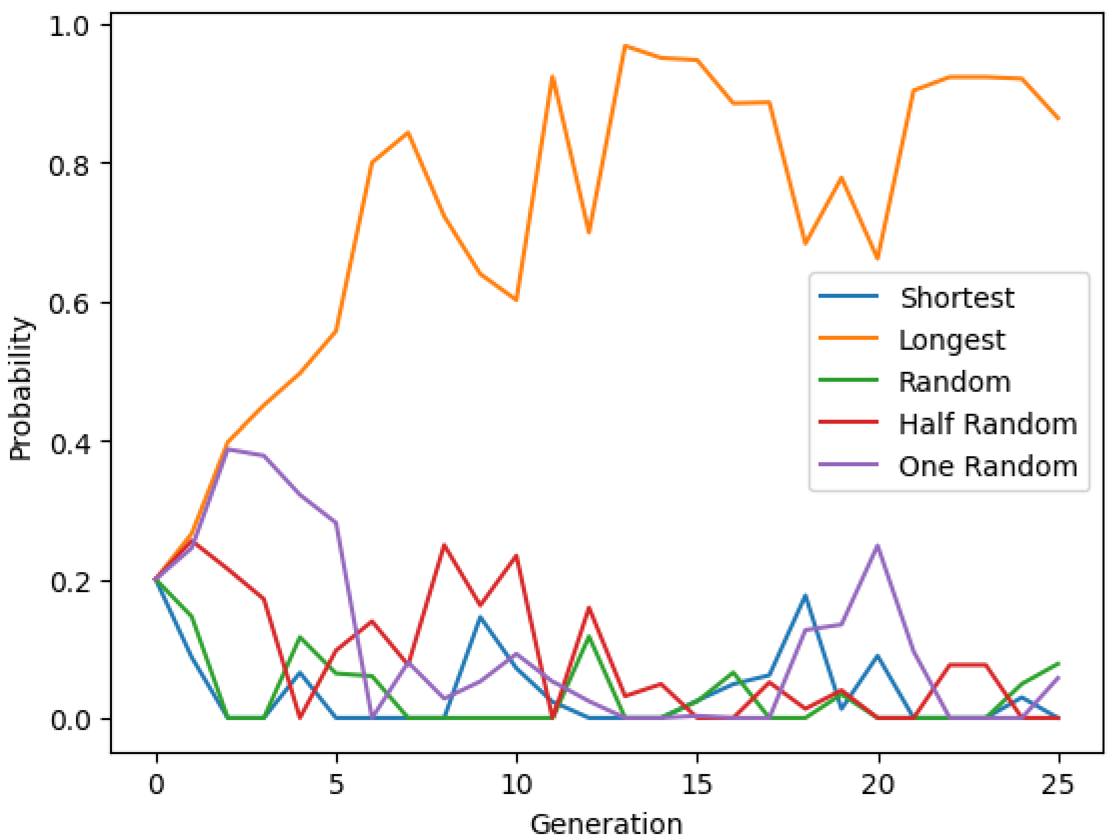

Shortest
Longest
Random
Half Random
One Random
Probability
Generation
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20
25

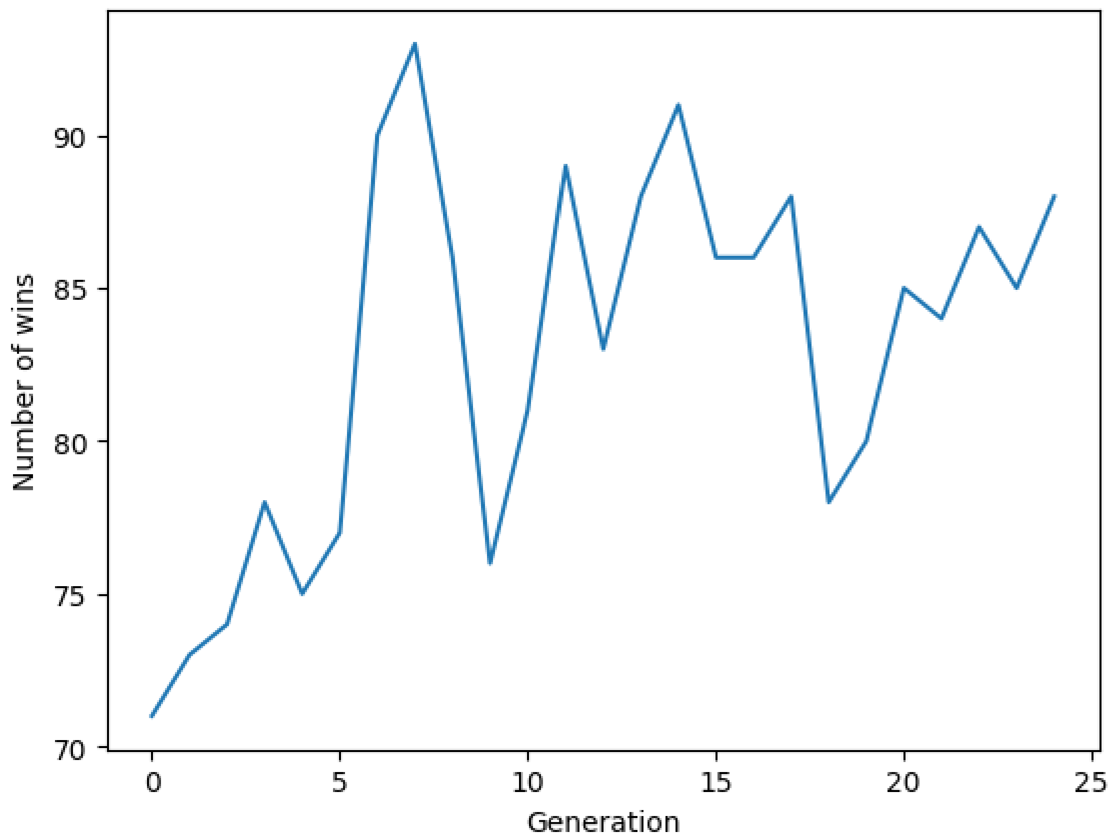

Number of wins
Generation
60
65
70
75
80
85
90
95
0
5
10
15
20
25

Adaptive (1,λ)-ES

```
current_solution = (0.20, 0.20, 0.20, 0.20, 0.20)
choosen_probability = list()
solutions_list = list()
stats = [0, 0]
counter = 0
for n in range(GENERATION_SIZE):
    print("Sigma for generation", n + 1, "is:", σ)
    offsprings = generate_offsprings(current_solution)
    offsprings.append(current_solution)

    evals = [
        (offspring, fitness(evolved_strategy(offspring))) for offspring in
offsprings
    ]
    previous_solution = evals[λ]
    for i in range(λ):
        if evals[i][1] > previous_solution[1]:
            counter += 1

    stats[1] += counter
    stats[0] += λ

    if (n + 1) % 5 == 0:
        if stats[1] / stats[0] < 1 / 5:
            σ /= 1.1
        elif stats[1] / stats[0] > 1 / 5:
            σ *= 1.1


    evals.sort(key=lambda x: x[1], reverse=True)

    # pprint(evals)

    current_solution = evals[0][0]
    choosen_probability.append(current_solution)
    solutions_list.append(evals[0][1])

    print(f"Best result for generation {n+1} is:", evals[0])


val = np.array([[0.20], [0.20], [0.20], [0.20], [0.20]])
curve_names = ["Shortest", "Longest", "Random", "Half Random", "One Random"]
choosen_probability = np.array(choosen_probability)

for i in range(GENERATION_SIZE):
    val = np.hstack((val, choosen_probability[i].reshape(-1, 1)))

for i in range(5):
    plt.plot(range(GENERATION_SIZE + 1), val[i], label=curve_names[i])
```

```
plt.xlabel("Generation")
plt.ylabel("Probability")
plt.legend()
plt.show()


plt.plot(range(GENERATION_SIZE), solutions_list)
plt.xlabel("Generation")
plt.ylabel("Number of wins")
plt.show()
```





- *Collaborations*: Worked with Angelo Iannielli - s317887
- *Sources*: Functions “fitness”, “state_info” and “evolved_strategy” have been inspired from Giovanni Squillero’s repository on github nevertheless they have been modified and readapted.

Submittetd Reviews for LAB 2

R1:

USER: Donato Lanzillotti

Hi Donato, you’ve been randomly chosen on random.org for my review. I hope you find my comments helpful.

Your methodologies are similar to mine; it seems we’re on the right way. I appreciate the idea of creating a new optimal function even though it wasn’t required.

I noticed that you evaluate fitness by having an individual play against an opponent whose strategy changes with each move. While this makes results more robust, considering there’s a proven optimal strategy for Nim, you could have had them play directly against the “optimal” strategy. This way, the algorithm learns to play against the best opponent and indirectly against all others.

Either way, it seems like you've done a good job. The code comments are helpful; perhaps you could have created a function that directly returns the move based on weights, avoiding if-else conditions in fitness. Anyway, the code is still clear.

Lastly, as a best practice, remember to always include labels in graphs for better readability.

Overall, great work! Feel free to comment on my code if you'd like.

R2:

USER: Michelangelo Caretto

Hi Michelangelo, I enjoyed your work and want to share some ideas. I'd like to congratulate with you for the README, it was very clear making it easy to understand the code. The game rules you designed are interesting and unconventional, good job. The variation of opponents and shifts when calculating fitness is a clever touch for more robust results. The range of weights is correct, but consider using real numbers from 0 onward to more easily evaluate the convergence of suboptimal strategies. Your implementation is good, but you could introduce a random choice based on the weights for strategy selection and maybe increment the number of games for each individual in order to avoid same fitness results for different individuals. I suggest adding intermediate outputs or a graph to visualize the evolution of the weights. Finally, label the graphs as a best practice for immediate understanding. I hope you find these suggestions useful. Good work!

Received Reviews

R1:

USER: Caretto Michelangelo s310178

Hello Nico, i'm going to review your work , hope you'll enjoy. I like a lot the way you wrote the code, because is so clean and readable and due to the graph i can easily understand how your "weight" structure works and which strategy is more strong in term of fitness. I would suggest you , when trying to create an agent playing a game to switch starting player every match too,otherwise your training will be only by one starting side. Talkink about the Es algortihm , you managed to find a solution in 25 Generation ,due to the fact of "Optimal function" is not optimal , but in general we can not call an algortihm "ES" with this low number of Generation . Next time maybe with a little curiosity you could study the Nim problem more to make the optimal function stronger…. (less homework done and more curiosity) I would like to say that despite everything, I appreciated the work and the cleanliness. Thank you for your effort and attention to detail. Bye Bye Nico, Michelangelo.

R2:

USER: Samuele Vanini s318684

Overview The code is well-written and easy to follow. The strategies implemented seem reasonable and are easy to understand.

Areas for Improvement I have just a couple of considerations about what I feel is not completely right:

All the evolution strategies shown are in the realm of the single-state methods; from what I understood, we also had to implement evolution strategies with a population mu bigger than one. In "Adaptive (1,λ)-ES" you are forcing the step size of sigma (dividing or multiplying it by 1.5); this is, in principle, wrong. We should not force a certain evolution using fixed parameters but let the algorithm find its way. If you want to balance exploration and exploitation, you can work on the selective pressure (in comma strategy, increase the number of lambda with respect to mu). Nim is an impartial game with a balanced state if and only if the nim-sum is 0, so you should alternate the player that does the first move during training. Take, for example, the match between 2 expert systems. The first player will always win if the game has an unbalanced initial state, even if the second player has the best strategy. This bias propagates in the training, introducing uncertainty during the fitness evaluation. Suggestions The number of generations, 25, is pretty low. The graph shows a fast convergence to 1 for the longest strategy. I would try to find new strategies to balance it. In "Adaptive (1,λ)-ES" there is a sigma for all the probability; would have been good to see the effect of an adaptive sigma for each weight. It would have been interesting to see other evolution strategies like (μ + λ) or (μ/ρ + λ)

R3:

USER: Donato Lanzillotti

PEER REVIEW Nim-Game POINT 1 The first point consisted of trying the different strategies and understand their performance. Running different games would have allowed you to realizing that the optimal strategy proposed was not totally correct (no 100% winning rate). Althought, changing it was not required.

POINT 2 About the ES, your idea was using the ES in order to find the most appropriate probabilities in choosing the shortest or the longest row. I noticed that you evaluate fitness by counting the number of winning games against a player that uses the optimal strategy. It is reasonable, but playing at the beginning just against a optimal player would not give you useful insight to move, since the winning rate would be always 0. This not the case since the opitmal strategy proposed has not a winning rate equals to 100%. Thnaks to the graphs it is possible to appreciate the rate at which the probability goes to 0, it means choosing always the longest row.

Overall, it seems you have done a good job. The code is quite clear to understand but as a suggestions more commments will help into the comprehension.

LAB 9

The aim of this lab is to Write a local-search algorithm (eg. an EA) able to solve the *Problem* instances 1, 2, 5, and 10 on a 1000-loci genomes, using a minimum number of fitness calls.

```
from random import choices
from random import random, randint, sample, uniform, seed
from copy import copy
from dataclasses import dataclass
import matplotlib.pyplot as plt

from tabulate import tabulate

import lab9_lib
```

BLACK-BOX PROBLEM with EA

- The goal of this implementation is to solve a problem with EA
- The goal is to maximize the fitness of an individual, how the fitness is evaluated is not known since it is a black-box problem
- An individual is has a genome of 1000 LOCI where a gene could be 0 or 1
- As additional information we know that an individual with all ones will have fitness equal 1, this information is additional and must not be used in the implementation since it can be considered cheating.
- We cannot use any method creating individuals, that gives an higher probability to have 1 as a gene but the algorithm must favor the survival of individuals with more ones by itself.

```
OFFSPRING_SIZE = 80
POPULATION_SIZE = 40
MUTATION_PROBABILITY = .10
NUM_LOCI = 1000
PROBLEM = [1, 2, 5, 10]
seed(20)
```

Individual

- The individual is organized as a class where fitness is the fitness value of the individual and the genotype is a list of 1000 integers (0/1)
- The individual also has a function to perform the mutation and a functon to perform the xover
- The individual also show the roulette wheel selection that is a tecnique to choose a parent in the population

```
@dataclass
class Individual:
    fitness: tuple
    genotype: list[int]

# NOT USED
def tournament_selection(population, tournament_size):
```

```python
    # Randomly select individuals for the tournament
    tournament = sample(population, tournament_size)
    # Return the individual with the highest fitness
    return max(tournament, key=lambda ind: ind.fitness)


# Select a parent in a random way, giving more probability to individuals with
higher fitness
def roulette_wheel_selection(population):
    # Calculate the total fitness of the population (total numbers of the roulette
wheel)
    total_fitness = sum(ind.fitness for ind in population)
    # Select a random value between 0 and the total fitness (select a random point
in the roulette wheel, like throwing the ball in a real roulette wheel)
    selection_point = uniform(0, total_fitness)
    # Go through the population and sum the fitness from 0, stop when the sum is
greater than the selection point
    # An individual with a higher fitness will have a higher probability that sum
will be greater than the selection point
    current_sum = 0
    for ind in population:
        current_sum += ind.fitness
        if current_sum > selection_point:
            return ind


# Given the genome, select randomly a gene and switch is value
def mutate(ind: Individual) -> Individual:
    offspring = copy(ind)
    pos = randint(0, NUM_LOCI - 1)

    if offspring.genotype[pos] == 1:
        offspring.genotype[pos] = 0
    else:
        offspring.genotype[pos] = 1
    offspring.fitness = None
    return offspring


# Give the genome of two individual, create a new offspring by removing a portion
of the genome from Ind1 and substituting it with the corresponding portion of Ind2
def n_cut_xover(ind1: Individual, ind2: Individual, n: int) -> Individual:
    # Generate n random cut points within the genotype range
    cut_points = sorted([randint(0, NUM_LOCI - 1) for _ in range(n)])

    # Initialize an empty offspring genotype
    offspring_genotype = []

    # Alternate between parents for each segment
    for i in range(n + 1):
```

```python
        # Determine the start and end of the segment
        start = cut_points[i - 1] if i > 0 else 0
        end = cut_points[i] if i < n else NUM_LOCI

        # Add the segment from the appropriate parent to the offspring genotype
        if i % 2 == 0:
            offspring_genotype += ind1.genotype[start:end]
        else:
            offspring_genotype += ind2.genotype[start:end]

    # Create the offspring
    offspring = Individual(fitness=None, genotype=offspring_genotype)

    assert len(offspring.genotype) == NUM_LOCI

    return offspring
```

Initial Population

- The initial population of size *POPULATON_SIZE* is created randomly with more probability to have a 0 then 1 in a locus
- For each individual in the initial population, we evaluate the fitness
- DISCLAIMER: creating an initial population with more zeros can be considered cheating because we know that these individuals will have a low value of fitness, by the way the intent of this choice is to appreciate how the algorithm increases the value of the fitness
- This choice also affects the number of fitness calls because the "plateau" will be reached with more generations
- Try the verision with uniform weights

```python
def generate_init_population(fitness):
    weights = [0.9,0.1]
    # weights = [0.5,0.5]
    population = [
        Individual(
            genotype = choices([0,1], weights=weights, k=NUM_LOCI),
            fitness=None,
        )
        for _ in range(POPULATION_SIZE)
    ]

    for i in population:
        i.fitness = fitness(i.genotype)
    return population
```

EA-Algorithm

- Given the initial population, the algorithm select with a certain probability, to create a given number of offsprings using two different techniques (mutation,xover)
- The population is extended with the offsprings and than it's trimmed, letting survive only the best individuals

- The algorithm is stopped when there are no improvements in the fitness (0.5% of variations in fitness in the last 600 generations) or if the value of the fitenss reachs 1

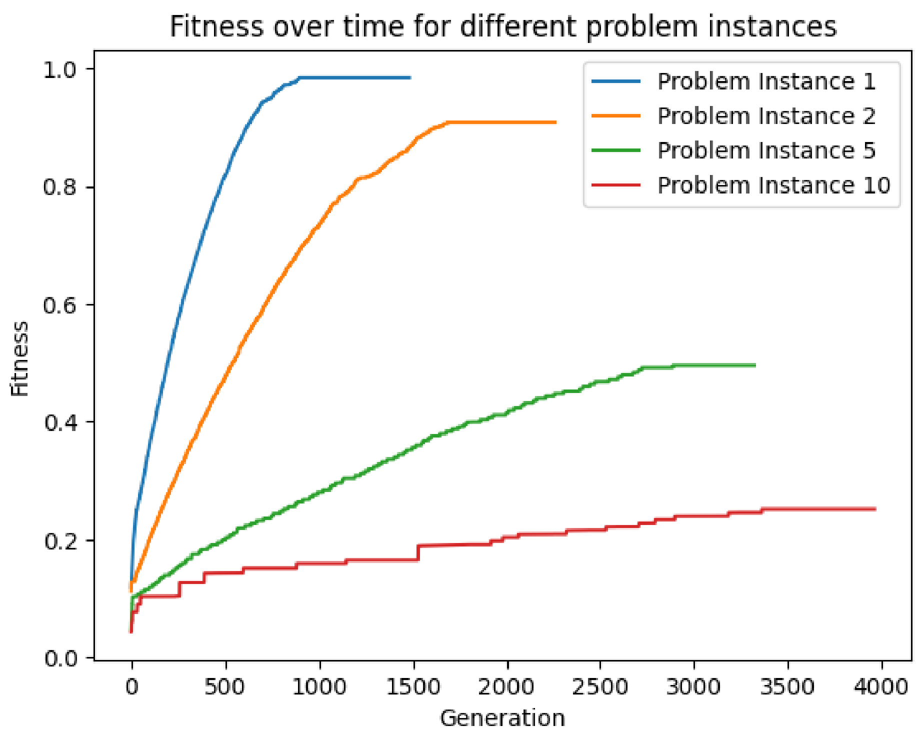
```python
def evolutionary_algorithm(fitness, population):
    generation = 0
    fitness_history = []
    while True:
        offspring = []

        for _ in range(OFFSPRING_SIZE):
            if random() < MUTATION_PROBABILITY:
                # mutation
                parent = roulette_wheel_selection(population)
                child = mutate(parent)
            else:
                # xover
                parent1 = roulette_wheel_selection(population)
                parent2 = roulette_wheel_selection(population)
                child = n_cut_xover(parent1, parent2, 6)
            offspring.append(child)

        for ind in offspring:
            ind.fitness = fitness(ind.genotype)

        population.extend(offspring) # add generated offspring to the population
        population.sort(key=lambda ind: ind.fitness, reverse=True) # sort the population by fitness

        ind = population[0] # get the best individual

        population = population[:POPULATION_SIZE] # keep only the best POPULATION_SIZE individuals

        current_fitness = ind.fitness

        # Append the fitness to the history
        fitness_history.append(current_fitness)

        # Check termination condition
        if generation >= 600:
            recent_fitness_variation = max(fitness_history[-600:]) - min(
                fitness_history[-600:]
            )
            if (
                recent_fitness_variation < 0.005 or current_fitness == 1
            ):  # 0.5% variation or current_fitness is 1
                print(
                    f"Terminating at generation {generation} due to low fitness variation or current_fitness reached 1."
                )
                break
```

```
        generation += 1
    return fitness_history
```

Problem Results

- In this section the EA algorithm produces results for the different istances of the problem we have, showing results on a plot

```
problem_instances = PROBLEM
fitness_histories = []
call_history = []

for instance in problem_instances:
    # Initialize your problem here with the current instance
    fitness = lab9_lib.make_problem(instance)

    init_population = generate_init_population(fitness)

    fitness_history = evolutionary_algorithm(fitness, init_population)

    # Append the fitness history of this run to the fitness_histories list
    fitness_histories.append(fitness_history)

    call_history.append(fitness.calls)

# Plot the results
for i, fitness_history in enumerate(fitness_histories):
    plt.plot(fitness_history, label=f"Problem Instance {problem_instances[i]}")

plt.title("Fitness over time for different problem instances")
plt.xlabel("Generation")
plt.ylabel("Fitness")
plt.legend()
plt.show()


table_data = []
for i, instance in enumerate(problem_instances):
    table_data.append([f"Problem Instance {instance}", f"{fitness_histories[i][-
1]:.2%}", call_history[i]])

table_headers = ["Problem Instance", "Final Fitness", "Fitness Calls"]
table = tabulate(table_data, headers=table_headers, tablefmt="pretty")

print(table)
```



Problem Instance	Final Fitness	Fitness Calls
Problem Instance 1	98.40%	118680
Problem Instance 2	90.80%	180840
Problem Instance 5	49.54%	265800
Problem Instance 10	25.13%	316920

- *Collaborations*: No collaborations

Submittetd Reviews

R1:

USER: Lorenzo Calosso - s306041

Hi Lorenzo I am sending you my review, I hope you will appreciate it.

In general your work seems to me really well done and structured, I had no difficulties in reading your code thanks to the short texts you inserted before each section. I really appreciated the various comparisons you made using various techniques which led you to choose the best configuration.

Improvements You used the same number of individuals for both the offsprings and the initial population, this is not a mistake but usually there are more offsprings than population size. You can try changing these parameters to note any reduction in the number of fitness calls while maintaining the same performance. I recommend you to add a stop condition in case your fitness reaches 1 and also avoid entering a fixed number of generations, the stop criterion you used instead seems very correct to me. For comparison between the various configurations you used it would also be interesting to have graphs also to show the learning process of your algorithm Running my code the results obtained in the various instances of the problem were sometimes quite different due to the presence of random elements, since your code contains many random elements I suggest you reevaluate the choices you made by perhaps setting a fixed seed for the random functions.

R2:

USER: Michelangelo Caretto s310178

Hi Michelangelo I am writing you this review hoping you will enjoy it.

First of all I thank you for the readme you wrote which allowed me to understand your idea. Next time I suggest you to put markdown comments also before the various code sections so that it will be more understandable.

Your idea of using metrics other than fitness seems interesting and certainly from the results you have shown it allows you to reduce the number of fitness calls while still getting good results. I am not entirely sure that using other metrics besides fitness is “standard” procedure for an EA but after all this seems to work. The results obtained for the vanilla verison seem a bit low to me, I think the reason is that you imposed a fixed number of generations rather than letting the algorithm go. You could set an infinite loop that interrupts if the fitness value reaches 1 or if you don’t notice any substantial improvement in the last x generations, this certainly increases the number of fitness calls but may also increase the result. Finally, I suggest you introduce some graphs to show the learning of the algorithm.

Received Reviews

R1:

USER: Lorenzo Calosso - s306041

Some considerations:

The code is well written and organized, the markdowns and the comments help to understand what you are doing The graph and the table at the end are a smart and nice way to present your results You obtained some good results on the first three instances with respect to the number of fitness calls done Some advice:

Try to combine also other techniques and see if the fitness improves; personally, I found out that, on this problem, techniques like Elitism, local-search mutation, inversion mutation or also the normal xover, combined together, give an improvement on the results. For what regard the final fitness of the instance 10, you could try to implement the self adaptive mutation rate instead of using a fixed one: in my case it has given a significant improvement

R2:

USER: Federico Buccellato s309075

Hello,

I have noticed that your solution faithfully follows the literature provided by the course. I find your code to be extremely well-organized and readable, and the comments significantly contribute to the understanding of the code. The evolutionary algorithm is structured effectively and appears correct. I particularly appreciated how you handle the possible termination in case of reaching the maximum fitness or constant fitness across generations.

Overall, your work is of high quality. The only suggestion I can give is to experiment with some different algorithms to assess how fitness behaves in alternative contexts.

Nevertheless, it is a job very well done!

LAB 10 - TicTacToe with RL

The aim of this lab is to solve a very simple game, using Reinforcement Learning Strategy

Tic-Tac-Toe with RL

- The game is played on a grid that's 3 squares by 3 squares.
- Players are "X" and "O".
- Players take turns putting their marks in empty squares.
- The first player to get 3 of her marks in a row (up, down, across, or diagonally) is the winner.
- When all 9 squares are full, the game is over. If no player has 3 marks in a row, the game ends in a tie.

Environment

- The grid is composed by a magic square [2, 7, 6, 9, 5, 1, 4, 3, 8]
- The idea of the magic square is that each row/column/diagonals sum up to 15

State

- The state is the set of positions taken by player "X" and player "O"

Action

- The action is the choice of a number inside the magic square

Reward

- The reward given to the agent is: 1 if the player wins the game, -1 if it loses and 0.5 if the game is a tie

Agent

- The agent is a player that uses the Q-Learning logic

```python
import random
from collections import namedtuple
from itertools import combinations
from random import seed
import matplotlib.pyplot as plt
from tqdm import tqdm

Position = namedtuple("Position", ["x", "o"])

seed(40)
```

Game Class

- play_game(): create the logic of the game where "X" and "O" players take turns
- win(): checks if a players has completed a row/column/diagonal (sum to 15)
- board_full(): checks if all the positions have been setted in the board (the game is tie)
- print_board() && print_board_info(): pretty print of the game board

```
class TicTacToe:
    def __init__(self, playerX, playerO, human_game=False):
        self.board = [2, 7, 6, 9, 5, 1, 4, 3, 8]
        self.current_board = Position(set(), set())
        self.playerX, self.playerO = playerX, playerO
        self.playerX_turn = random.choice([True, False]) #randomly choose who goes
first
        self.winner = None
        self.human_game = human_game

    def play_game(self):
        self.playerX.start_game("X")
        self.playerO.start_game("O")
        while True:
            player, char, other_player = (
                (self.playerX, "X", self.playerO)
                if self.playerX_turn
                else (self.playerO, "O", self.playerX)
            )

            if self.human_game:
                print(f"Player {char} move")
                self.print_board_info()

            move = player.move(self.current_board)

            moves = self.current_board.x if self.playerX_turn else
self.current_board.o
            moves.add(move)
            if self.human_game:
                self.print_board()

            if self.win(moves):
                player.reward(1, self.current_board)
                other_player.reward(-1, self.current_board)
                self.winner = char
                break

            if self.board_full():  # tie game
                player.reward(0.5, self.current_board)
                other_player.reward(0.5, self.current_board)
                self.winner = None
                break

            other_player.reward(0, self.current_board)
            self.playerX_turn = not self.playerX_turn

    def win(self, state):
        return any(sum(c) == 15 for c in combinations(state, 3))
```

```
    def board_full(self):
        player = self.playerX if self.playerX_turn else self.playerO
        return player.available_moves(self.current_board) == set()

    def print_board(self):
        for r in range(3):
            print("-------------")
            for c in range(3):
                i = r * 3 + c
                char = " "
                if self.board[i] in self.current_board.x:
                    char = "X"
                elif self.board[i] in self.current_board.o:
                    char = "O"
                print(f"| {char}", end=" ")
            print("|")
        print("-------------")

    def print_board_info(self):
        for r in range(3):
            print("-------------")
            for c in range(3):
                i = r * 3 + c

                print(f"| {self.board[i]}", end=" ")
            print("|")
        print("-------------")
```

Player Class

- The player class is a generic class with some methods to implement the choosing action logic, available moves and also a reward functon if needed
- Random Player class overrides the player class implementing a player that takes action randomly
- Q-Learining Player class overrides the player implementing a RL agent

```
class Player(object):
    def __init__(self):
        self.name = "human"

    def start_game(self, char):
        print("\nNew game!")

    def move(self, current_board):

        move = int(input("Your move? "))
        if move not in self.available_moves(current_board):
            print("Illegal move.")
            move = self.move(current_board)
```

```
        return move

    def reward(self, value, current_board):
        print("{} rewarded: {}".format(self.name, value))

    def available_moves(self, current_board):
        available = set(range(1, 9 + 1)) - current_board.x - current_board.o

        return available


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

    def reward(self, value, board):
        pass

    def start_game(self, char):
        pass

    def move(self, current_board):
        available = self.available_moves(current_board)
        return random.choice(list(available))

class QLearningPlayer(Player):
    def __init__(self, epsilon=0.2, alpha=0.2, gamma=0.9):
        self.name = "Qlearner"
        self.q = {}  # (state, action) keys: Q values
        self.epsilon = epsilon  # e-greedy chance of random exploration
        self.alpha = alpha  # learning rate
        self.gamma = gamma  # discount factor for future rewards

    def start_game(self, char):
        self.last_state = (set(), set())
        self.last_action = None

    def getQ(self, state, action):
        # encourage exploration; "optimistic" 1.0 initial values
        if self.q.get((state, action)) is None:
            self.q[(state, action)] = 1.0
        return self.q.get((state, action))

    def move(self, current_board):
        self.last_state = (
            tuple(current_board.x),
            tuple(current_board.o),
        )  # Convert Position to tuple
        possible_actions = list(self.available_moves(self.last_state))

        if random.random() < self.epsilon:
```

```
            self.last_action = random.choice(list(possible_actions))
            return self.last_action

        qs = [self.getQ(self.last_state, a) for a in possible_actions]
        maxQ = max(qs)

        if qs.count(maxQ) > 1:
            # more than 1 best option; choose among them randomly
            best_options = [i for i in range(len(possible_actions)) if qs[i] ==
maxQ]
            i = random.choice(best_options)
        else:
            i = qs.index(maxQ)

        self.last_action = possible_actions[i]
        return possible_actions[i]

    def reward(self, value, current_board):
        new_state = (tuple(current_board[0]), tuple(current_board[1]))
        if self.last_action:
            self.learn(
                self.last_state,
                self.last_action,
                value,
                new_state
            )

    def learn(self, state, action, reward, result_state):
        prev = self.getQ(state, action)
        if self.available_moves(result_state) == set():
            self.q[(state, action)] = prev
        else:
            maxqnew = max([self.getQ(result_state, a) for a in
self.available_moves(result_state)])
            self.q[(state, action)] = (1-self.alpha)*prev + self.alpha * (
                (reward + self.gamma * maxqnew)
            )

    def available_moves(self, current_board):
        available = set(range(1, 9 + 1)) - set(current_board[0]) -
set(current_board[1])
        return available
```

Train

- Train the RL agent against a player

```
def trained_agent(p1, agent, num_games=200000):
    for _ in tqdm(range(0, num_games)):
        t = TicTacToe(p1, agent)
```

```
        t.play_game()
    return agent
```

Test

- Test the trained agent against a player

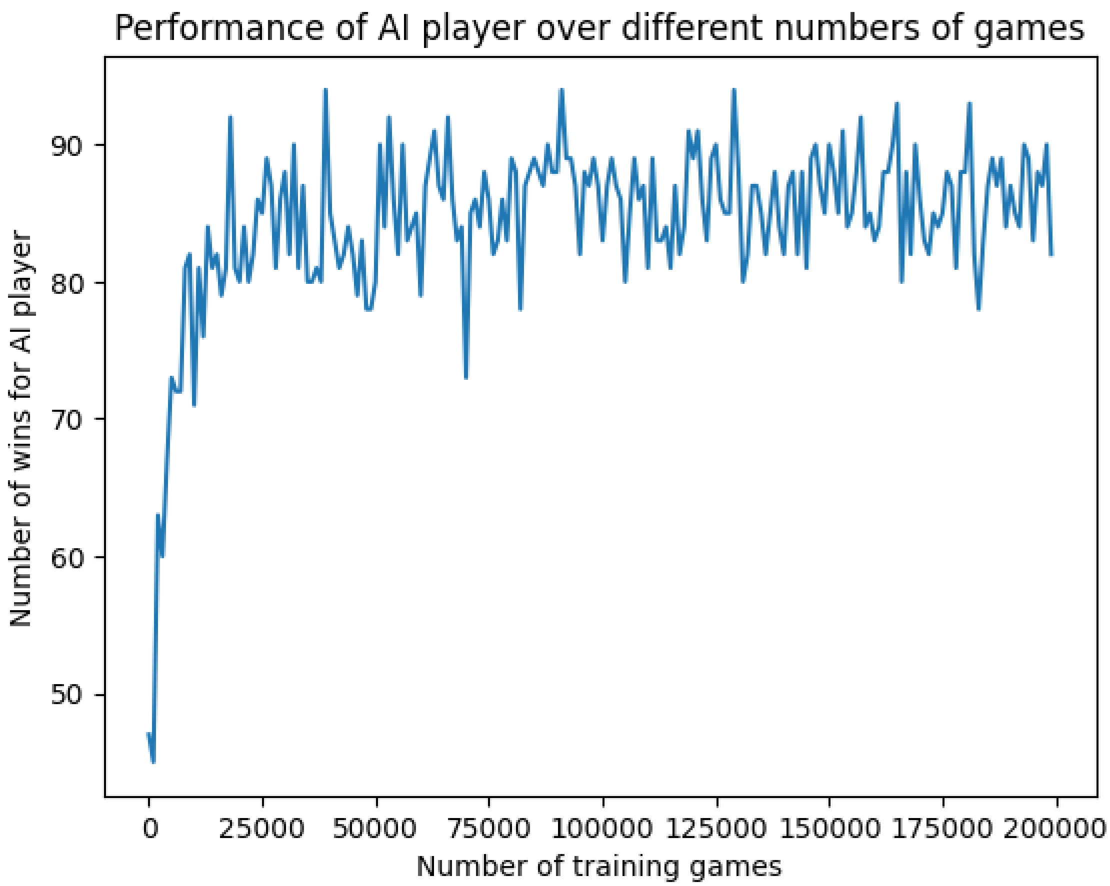
```
agent = trained_agent(RandomPlayer(), QLearningPlayer())
p1 = RandomPlayer()
agent.epsilon = 0 # remove randomness from the trained agent

num_X = 0
num_O = 0
num_ties = 0
for _ in range(100):
    t = TicTacToe(p1, agent)
    t.play_game()

    if t.winner == "X":
        num_X += 1
    elif t.winner == "O":
        num_O += 1
    else:
        num_ties += 1

print("X wins: " + str(num_X))
print("O wins: " + str(num_O))
print("Ties: " + str(num_ties))
```

Learning process

- The agent is trained for different number of games values
- Higher the number of training games, higher the performance
- Run this part is time consuming (20min on my laptop)

```
num_train_games_values = list(range(1, 200001, 1000))


results = []
for num in tqdm(num_train_games_values):
    agent = trained_agent(RandomPlayer(), QLearningPlayer(), num)
    p1 = RandomPlayer()
    num_win = 0

    for _ in range(100):

        t = TicTacToe(p1, agent)
        t.play_game()

        if t.winner == "O":
```

```
            num_win += 1
    results.append(num_win)



plt.plot(num_train_games_values, results)
plt.xlabel('Number of training games')
plt.ylabel('Number of wins for AI player')
plt.title('Performance of AI player over different numbers of games')
plt.show()
```



Plays against AI

- Try to defeat the AI playing a game

```
human = Player()
t = TicTacToe(human, agent, human_game=True)
print("TIC TAC TOE")
t.print_board_info()
print("------------------------------")

t.play_game()
print("Winner is: " + str(t.winner))
t.print_board()
```

- *Collaborations*: No collaborations

Submittetd Reviews

R1:

USER: Federico Buccellato s309075

Hi Federico here is my review for your work, I hope you will appreciate it.

First of all, your work seems well done to me and your extension of what we saw in class using Q-learning seems correct. I have some pointers for you to improve your work and make it better understandable. I would suggest you to divide your code into classes by perhaps creating a "game" class and a "player" class this allows for better organization and also better reading of the code. In the training phase you could add an additional "epsilon" parameter to encourage agent exploration by choosing a random action (greedy approach) The results look good to me, try to increase the number of matches to update the Q-Table you might have better results. To make the agent more robust, you could do training with different types of players, to be varied randomly during the various iterations just for this reason, creating a player class to be extended could be a good idea.

In conclusion your work seems well done, I only suggest you to improve a bit the organization of your code. Good work for the next projects!

R2

USER: Lorenzo Calosso - s306041

Hi Lorenzo here is my review for your work, I hope you will appreciate it.

I really congratulate you as your code is really easy to read and understand due to the use of the classes you created. The implementation of Q-Learning seems to me to be correct from a theoretical and also implementation point of view. I don't have many comments to make to you as the code is well written and very similar to my implementation and also the results obtained seem promising. To further improve your work you could create a "player" class to be extended with different types of players, "random", "minmax", "RL-agent" or other types of players with other strategies, so as to randomly vary the opponent during training making the final agent more robust. It would also be interesting to see the learning process of your agent as the number of iterations of training changes, so you can figure out the right tradeoff between the number of training iterations and the number of wins.

In conclusion, I again congratulate you on your work.

Received Reviews

R1

USER: Paul Raphael

Hello,

Your work is great and very well explained and displayed, the only thing lacking in my opinion is that you don't change the parameters alpha gamma and epsilon during training (maybe you did it on your own or I missed it) It could be interesting. Aside from that I couldn't find any issues good job and good luck for the exam !

R2

USER: Angelo Iannielli

Hello Nicolò,

I've conducted a review of your code and wanted to share my observations.

My 2 cents: Your code is well-structured and organized. I particularly appreciate the clear division into classes, managing both the game logic and player behaviors. This choice significantly enhances the readability and maintainability of the code.

The option to test the trained player in a match against a human player is very interesting. This feature makes your code interactive, providing an immediate test of the achieved results during training.

The integration of graphs at the end of the lab provides a comprehensive view of the learning process. This is a positive touch that offers a visual overview of the agent's performance throughout the training matches.

Recommended Adjustments: The start_game() function might be considered redundant as it merely prints a static string. You may want to evaluate whether it's essential to keep this function. The print statements during training could be shortened to enhance file readability. Consider reducing the length of the prints while retaining essential information. Future Developments: It could be interesting to explore varying the epsilon variable as the algorithm learns to play. This might help reduce randomness in the agent's moves and reduce exploration during the learning process. Additionally, it would be compelling to train the agent against players using different strategies. This could provide insights into how well the agent adapts to diverse playing styles. Overall, you've done an excellent job. Keep it up and consider the suggestions to further enhance your code

Quixo Project

ExtendedGame

Overview

The ExtendedGame class extends the functionality of a basic game represented by the Game class. It introduces additional functions that do not change the logic of the game but are useful to implement a player such as MinMaxPlayer.

Class Overview

ExtendedGame Methods:

- **possible_moves(self, playerId: int) -> tuple[tuple[int, int], Move]:** Returns a tuple of possible moves for a given player in the current state of the game
- **create_new_state(self, from_pos: tuple[int, int], slide: Move, player_id: int) -> "ExtendedGame":** Creates a new game state performing a move
- **_switch_player()** switch the current player after a move

Players

Overview

- RandomPlayer: A player that makes random moves.
- HumanPlayer: A player that allows a human to interactively make moves.
- MinMaxPlayer: An AI player using the Minimax algorithm with Alpha-Beta Pruning to make strategic moves.

Player Classes

- RandomPlayer This class represents a player that randomly selects moves on the game board. It is implemented with the make_move method, where it generates random positions and a random move direction.
- HumanPlayer This class represents a player that allows a human to interactively make moves. The make_move method prompts the user to input the position to move from and the direction to move.
- MinMaxPlayer This class represents an AI player using the Minimax algorithm with Alpha-Beta Pruning to make optimal moves. The make_move method implements the Minimax algorithm to evaluate possible moves and choose the best one. The evaluate method assigns scores to different game states, and the minmax method recursively explores possible moves while considering alpha-beta pruning for optimization.

MinMaxPlayer Configuration

The MinMaxPlayer class takes three parameters during instantiation:

- game: The game object (an instance of ExtendedGame) on which the player will make moves.
- max_depth: The maximum depth to explore in the Minimax algorithm.

The agent is proposed with a **max_depth = 10** even if tests show that the agent as good results also with other lower even numbers.

Useful functions and Testing

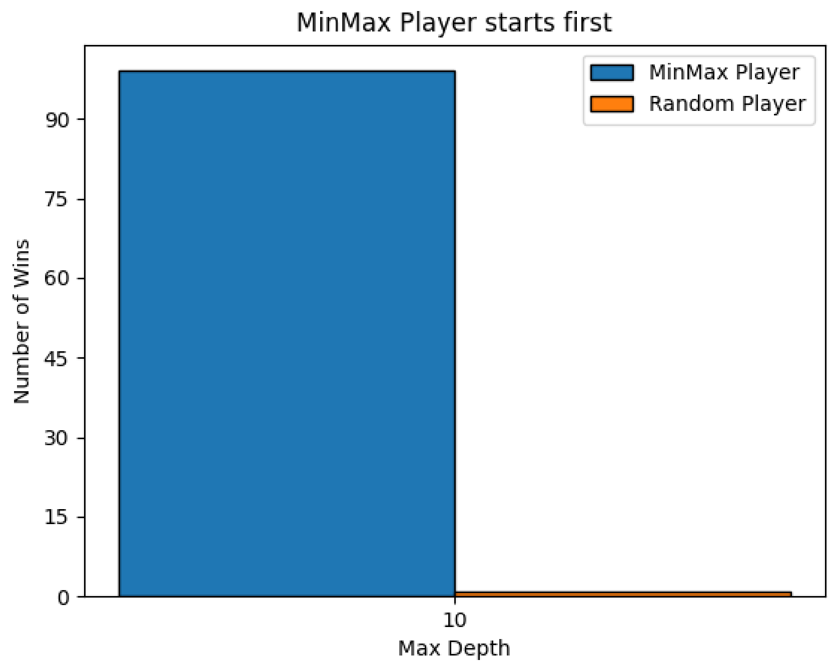
In order to test the performance of the game some useful functions are provided.

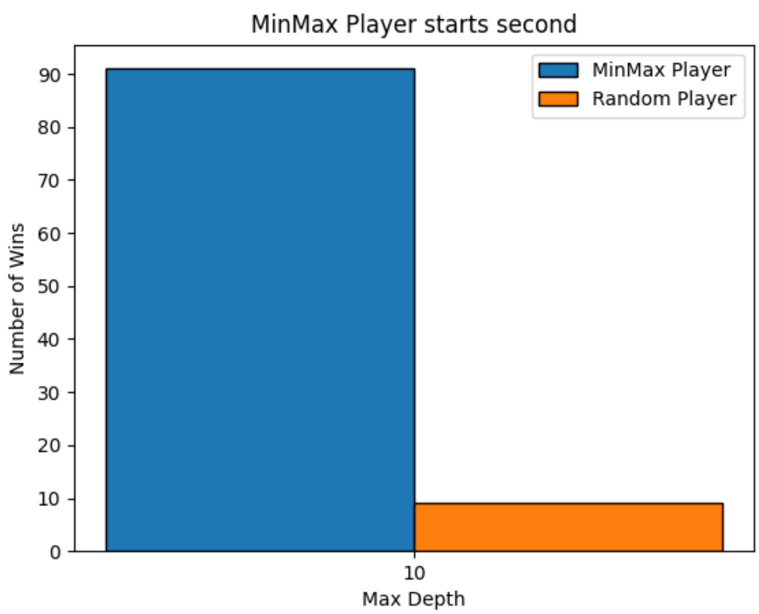
- **test_agent(num_games)** to simply test the Minmax player against a RandomPlayer 100 times

- **test_agent_depths(num_games, max_depths)** to test the MinMaxPlayer performance against a RandomPlayer, playing 100 times as first player (simbol 0) and 100 times as second player (simbol 1)

- **plot_results(results, filename, title)** to print some histograms about results obtained with the previous function.

- **play_against_ai()** to play a real time game against the MinMaxPlayer.


MinMax Player starts first
MinMax Player
Random Player
Number of Wins
0
15
30
45
60
75
90
10
Max Depth


MinMax Player starts second
MinMax Player
Random Player
Number of Wins
0
10
20
30
40
50
60
70
80
90
10
Max Depth

```python
class ExtendedGame(Game):
    def __init__(self):
        super().__init__()

    def possible_moves(self, playerId: int) -> tuple[tuple[int, int], Move]:
        """Return a tuple of possible moves for a given player in a given state of
the game"""

        # Define the edges of the game grid
        perimeter = [0, 4]
        # Initialize an empty list to store possible moves
        possible_moves = []
        # Get the current game board
        board = self.get_board()

        # Iterate over the edges of the game grid
        for index in perimeter:
            # Iterate over the columns of the game grid
            for col in range(5):
                # If the current cell belongs to the current player or is empty
                if board[col][index] in {playerId, -1}:
                    # If we are not on the first column, we can move up
                    if col != 0:
                        possible_moves.append(((index, col), Move.TOP))
                    # If we are not on the last column, we can move down
                    if col != 4:
                        possible_moves.append(((index, col), Move.BOTTOM))
                    # If we are not on the first row, we can move left
                    if index != 0:
                        possible_moves.append(((index, col), Move.LEFT))
                    # If we are not on the last row, we can move right
                    if index != 4:
                        possible_moves.append(((index, col), Move.RIGHT))

            # Iterate over the rows of the game grid
            for row in range(5):
                # If the current cell belongs to the current player or is empty
                if board[index][row] in {playerId, -1}:
                    # If we are not on the first column, we can move up
                    if index != 0:
                        possible_moves.append(((row, index), Move.TOP))
                    # If we are not on the last column, we can move down
                    if index != 4:
                        possible_moves.append(((row, index), Move.BOTTOM))
                    # If we are not on the first row, we can move left
                    if row != 0:
                        possible_moves.append(((row, index), Move.LEFT))
                    # If we are not on the last row, we can move right
                    if row != 4:
                        possible_moves.append(((row, index), Move.RIGHT))
```

```python
        # Return the possible moves as a tuple
        return tuple(possible_moves)

    def create_new_state(
        self, from_pos: tuple[int, int], slide: Move, player_id: int
    ) -> "ExtendedGame":
        """Return a new game state after applying a move"""

        # Swap the position coordinates
        from_pos = (from_pos[1], from_pos[0])
        # Create a new instance of the ExtendedGame
        new_game = ExtendedGame()
        new_game.current_player_idx = player_id
        # Copy the current game board to the new game
        new_game._board = deepcopy(self._board)
        new_game._take(from_pos, player_id)
        new_game._slide(from_pos, slide)
        new_game._switch_player()

        # Return the new game state
        return new_game

    def _switch_player(self):
        self.current_player_idx = 1 - self.current_player_idx


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

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


class HumanPlayer(Player):
    def __init__(self) -> None:
        super().__init__()
        self.name = "HumanPlayer"

    def make_move(self, game: "ExtendedGame") -> tuple[tuple[int, int], Move]:
        # Get the current player
        player = game.get_current_player()

        # Get the list of possible moves
        possible_moves = game.possible_moves(player)
```

```
        print("BOARD:")
        game.print()

        # Print the list of possible moves
        print("Possible moves:")
        for move in possible_moves:
            print(f"From position {move[0]} move {move[1]}")

        # Ask the user for their move
        from_pos = tuple(
            map(int, input("Enter the position to move from (row, col):
").split(","))
        )
        move = Move[
            input("Enter the direction to move (TOP, BOTTOM, LEFT, RIGHT):
").upper()
        ]
        return from_pos, move


class MinMaxPlayer(Player):
    def __init__(self, game: "ExtendedGame", max_depth) -> None:
        super().__init__()
        self.name = "MinMaxPlayer"
        self.game = game
        self.max_depth = max_depth
        self.infinity = float("inf")

    def evaluate(self, game: "ExtendedGame") -> int:
        player = (
            1 - game.get_current_player()
        )  # restore the player of the prevoius state since create_new_state()
switch it.
        score = 0
        board = game.get_board()

        # Check rows
        for row in board:
            score += self.evaluate_line(row, player)

        # Check columns
        for col in board.T:
            score += self.evaluate_line(col, player)

        # Check main diagonal
        main_diag = np.diagonal(board)
        score += self.evaluate_line(main_diag, player)

        # Check secondary diagonal
        secondary_diag = np.diagonal(np.fliplr(board))
```

```
        score += self.evaluate_line(secondary_diag, player)

        return score

    @staticmethod
    def evaluate_line(line: list[int], player_id: int) -> int:
        line_score = 0

        # Count occurrences of player's symbol and opponent's symbol
        player_count = np.sum(line == player_id)
        opponent_count = np.sum(line == 1 - player_id)

        # Assign scores based on counts
        if player_count > 0:
            line_score += 10**player_count
        if opponent_count > 0:
            line_score -= 10**opponent_count

        return line_score

    def minmax(
        self,
        game: "ExtendedGame",
        depth: int,
        alpha: float,
        beta: float,
        isMaximizingPlayer: bool,
    ) -> tuple[int, float, float]:
        # Base case: if we have reached the maximum depth or the game is over,
        # return the evaluation of the game state

        if depth == 0 or game.check_winner() != -1:
            return self.evaluate(game), alpha, beta

        # Decrease the depth
        depth -= 1

        player = game.get_current_player()

        # If we are the maximizing player
        if isMaximizingPlayer:
            # Initialize the maximum evaluation to negative infinity
            bestVal = -self.infinity
            # Iterate over all possible moves
            for move in game.possible_moves(player):
                # Create a new game state by making the move
                new_state = game.create_new_state(move[0], move[1], player)
                # Call minmax recursively on the new state
                value, alpha, beta = self.minmax(new_state, depth, alpha, beta,
False)
```

```
                # Update the maximum evaluation
                bestVal = max(bestVal, value)
                # Update alpha
                alpha = max(alpha, value)
                # If beta is less than or equal to alpha, break the loop (alpha-
beta pruning)
                if alpha >= beta:
                    break
            # The result is the maximum evaluation, alpha and beta
            result = bestVal, alpha, beta

        # If we are the minimizing player
        else:
            # Initialize the minimum evaluation to positive infinity
            bestVal = self.infinity
            # Iterate over all possible moves
            for move in game.possible_moves(player):
                # Create a new game state by making the move
                new_state = game.create_new_state(move[0], move[1], player)
                # Call minmax recursively on the new state
                value, alpha, beta = self.minmax(new_state, depth, alpha, beta,
True)
                # Update the minimum evaluation
                bestVal = min(bestVal, value)
                # Update beta
                beta = min(beta, value)
                # If beta is less than or equal to alpha, break the loop (alpha-
beta pruning)
                if alpha >= beta:
                    break
            # The result is the minimum evaluation, alpha and beta
            result = bestVal, alpha, beta

        return result

    def make_move(self, game: "ExtendedGame") -> tuple[tuple[int, int], Move]:
        # Initialize the best move to None and the best evaluation to negative
infinity
        bestMove = None
        bestVal = -self.infinity

        # Get the current player, it will be the index of the MinmaxPlayer
        player = self.game.get_current_player()

        # Get the list of possible moves for MinmaxPlayer
        possible_moves = list(game.possible_moves(player))

        # Iterate over all possible moves
        for move in possible_moves:
            # Create a new game state by making the move
            new_state = self.game.create_new_state(move[0], move[1], False)
```

```python
            # Early return if a move is a winning move for MinmaxPlayer
            if new_state.check_winner() == player:
                bestMove = move
                break

            # Call the Minimax function on the new state to get the evaluation of the state
            value = self.minmax(
                new_state, self.max_depth, -self.infinity, self.infinity, False
            )[0]

            # If the evaluation of the state is greater than the best evaluation, update the best evaluation and the best move
            if value > bestVal:
                bestVal = value
                bestMove = move

        return bestMove



def test_agent(num_games: int) -> None:
    count_0 = 0
    count_1 = 0

    for _ in range(num_games):
        game = ExtendedGame()
        player1 = MinMaxPlayer(game, 10)
        player2 = RandomPlayer()

        winner = game.play(player1, player2)

        if winner == 0:
            count_0 += 1
        else:
            count_1 += 1
        print(f"{player1.name} win {count_0} matches")
        print(f"{player2.name} win {count_1} matches")


def test_agent_depths(num_games: int, max_depths: list[int]) -> None:
    results_when_first = []
    results_when_second = []

    for max_depth in tqdm(max_depths, desc="Testing depths"):
        wins_when_first = 0
        losses_when_first = 0
        wins_when_second = 0
        losses_when_second = 0
```

```python
        for _ in tqdm(range(num_games), desc="Testing games"):
            # MinMaxPlayer starts first
            game = ExtendedGame()
            player1 = MinMaxPlayer(game, max_depth)
            player2 = RandomPlayer()
            winner = game.play(player1, player2)

            if winner == 0:
                wins_when_first += 1
            else:
                losses_when_first += 1

            # MinMaxPlayer starts second
            game = ExtendedGame()
            player1 = RandomPlayer()
            player2 = MinMaxPlayer(game, max_depth)
            winner = game.play(player1, player2)

            if winner == 1:
                wins_when_second += 1
            else:
                losses_when_second += 1

        results_when_first.append((max_depth, wins_when_first, losses_when_first))
        results_when_second.append((max_depth, wins_when_second,
losses_when_second))

    return results_when_first, results_when_second


def plot_results(
    results: list[tuple[int, int, int]], filename: str, title: str
) -> None:
    depths, wins_0, wins_1 = zip(*results)
    width = 0.35  # the width of the bars

    fig, ax = plt.subplots()

    ax.bar(
        [d - width / 2 for d in depths],
        wins_0,
        width,
        label="MinMax Player",
        edgecolor="black",
    )
    ax.bar(
        [d + width / 2 for d in depths],
        wins_1,
        width,
```

```
        label="Random Player",
        edgecolor="black",
    )

    ax.set_title(title)
    ax.set_xlabel("Max Depth")
    ax.set_ylabel("Number of Wins")
    ax.set_xticks(depths)
    ax.yaxis.set_major_locator(
        MaxNLocator(integer=True)
    )  # Set y-axis to display only integers
    ax.legend()

    plt.savefig(filename)

    plt.show()


def play_against_ai() -> None:
    game = ExtendedGame()
    player1 = MinMaxPlayer(game, 10)
    player2 = HumanPlayer()

    winner = game.play(player1, player2)

    if winner == 0:
        print(f"{player1.name} wins!")
    else:
        print(f"{player2.name} wins!")


if __name__ == "__main__":


    test_agent(100)

    # results_first, results_second = test_agent_depths(
    #     100, [10]
    # )
    # plot_results(results_first, "results_first.png", "MinMax Player starts
first")
    # plot_results(results_second, "results_second.png", "MinMax Player starts
second")

    # play_against_ai()
```