Dimitri Masetta 306130

COMPUTATIONAL INTELLIGENCE

2023-2024

Work report

Dimitri Masetta 306130

Minimum coverage (breadth, depth, dijkstra)

23/10/23

minimumCoverage.ipynb

```
from random import random
from functools import reduce
from collections import namedtuple, deque
from queue import  PriorityQueue, SimpleQueue, LifoQueue
import numpy as np
```

```
PROBLEM_DIMENSION = 10
NUM_SAMPLES = 20
samples = tuple(np.array([random() < .3 for _ in range(PROBLEM_DIMENSION)]) for _ in range(NUM_SAMPLES))
State = namedtuple('State', ['taken', 'not_taken'])
print(samples)
```

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

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

Implementation with python varius queue

```
#Breadth first
frontier = SimpleQueue()
#Depth first
#frontier = LifoQueue()
#Dijkstra
# frontier = PriorityQueue()

frontier.put(State(set(), set(range(NUM_SAMPLES))))

counter = 0
current_state: State = frontier.get()
while not goal_check(current_state):
    counter += 1
    for action in current_state.not_taken:
        new_state = State(current_state.taken | {action}, current_state.not_taken - {action})
        frontier.put(new_state)
        # print("Frontier:", frontier.queue)
    # print("Frontier:", frontier.queue)
    current_state = frontier.get()
```

```
    # print("CurrentState:", current_state)
print(f"Solved in {counter} steps")
print(f"State: {current_state}")
```

Implementation with python DEQUE

```
#Breadth first
frontier = deque()
starting_state = State(set(), set(range(NUM_SAMPLES)))
frontier.append(starting_state)

counter = 0
current_state: State = frontier.popleft()   #FIFO
while not goal_check(current_state):
    counter += 1
    for action in current_state.not_taken:
        new_state = State(current_state.taken | {action}, current_state.not_taken - {action})
        frontier.append(new_state)
        # print("Frontier:", frontier.queue)
    # print("Frontier:", frontier.queue)
    current_state = frontier.popleft()
    # print("CurrentState:", current_state)
print(f"Solved in {counter} steps")
print(f"State: {current_state}")
```

```
#Depth first
frontier = deque()
starting_state = State(set(), set(range(NUM_SAMPLES)))
frontier.append(starting_state)

counter = 0
current_state: State = frontier.pop()   #LIFO
while not goal_check(current_state):
    counter += 1
    for action in current_state.not_taken:
        new_state = State(current_state.taken | {action}, current_state.not_taken - {action})
        frontier.append(new_state)
        # print("Frontier:", frontier.queue)
    # print("Frontier:", frontier.queue)
    current_state = frontier.pop()
    # print("CurrentState:", current_state)
print(f"Solved in {counter} steps")
print(f"State: {current_state}")
```

Dimitri Masetta 306130

Minimum coverage (Greedy)

23/10/23

minimumCoverageGreedy.ipynb

```
from random import random
from functools import reduce
from collections import namedtuple
from queue import  PriorityQueue, SimpleQueue, LifoQueue
import numpy as np

PROBLEM_DIMENSION = 10
NUM_SAMPLES = 20
samples = tuple(np.array([random() < .3 for _ in range(PROBLEM_DIMENSION)]) f
or _ in range(NUM_SAMPLES))
State = namedtuple('State', ['taken', 'not_taken'])
print(samples)

def goal_check(state: State):
    return np.all(reduce(np.logical_or, [samples[i] for i in state.taken], np
.array([False for _ in range(PROBLEM_DIMENSION)])))

def distance(state: State):
    return PROBLEM_DIMENSION - sum(reduce(np.logical_or, [samples[i] for i in
state.taken], np.array([False for _ in range(PROBLEM_DIMENSION)])))

assert goal_check(State(set(range(NUM_SAMPLES)), set())), "Problem not solvab
le"

frontier = PriorityQueue()
state: State = State(set(), set(range(NUM_SAMPLES)))
frontier.put((distance(state), state))

counter = 0
_, current_state = frontier.get()
while not goal_check(current_state):
    counter += 1
    for action in current_state.not_taken:
        new_state = State(current_state.taken | {action}, current_state.not_t
aken - {action})
        frontier.put((distance(new_state), new_state))
        # print("Frontier:", frontier.queue)
    _, current_state = frontier.get()
    # print("CurrentState:", current_state)
print(f"Solved in {counter} steps")
print(f"State: {current_state}")
```

LAB 1 (Minimum coverage AStar)

23/10/23

minimumCoverageAStar.ipynb

```
from random import random
from functools import reduce
from collections import namedtuple
from queue import  PriorityQueue, SimpleQueue, LifoQueue
import numpy as np
```

```
PROBLEM_DIMENSION = 3
NUM_SAMPLES = 5
samples = tuple(np.array([random() < .3 for _ in range(PROBLEM_DIMENSION)]) for _ in range(NUM_SAMPLES))
State = namedtuple('State', ['taken', 'not_taken'])
print(samples)
```

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

def distance_traveled(state: State):
    return len(state.taken)

def distance_from_goal(state: State):
    return PROBLEM_DIMENSION - sum(reduce(np.logical_or, [samples[i] for i in state.taken], np.array([False for _ in range(PROBLEM_DIMENSION)])))

def f_score(state: State):
    return  distance_traveled(state) + distance_from_goal(state)
```

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

```
frontier = PriorityQueue()
state: State = State(set(), set(range(NUM_SAMPLES)))
frontier.put((f_score(state), state))

counter = 0
_, current_state = frontier.get()
while not goal_check(current_state):
    counter += 1
    for action in current_state.not_taken:
        new_state = State(current_state.taken | {action}, current_state.not_taken - {action})
        frontier.put((f_score(new_state), new_state))
        # print("Frontier:", frontier.queue)
```

```
    # print("Frontier:", frontier.queue)
    _, current_state = frontier.get()
    #print("CurrentState:", current_state)
print(f"Solved in {counter} steps")
print(f"State: {current_state}")
```

Proposed personal project (Minimum coverage EA)

Discussed with the professor, not exposed because it was too early in the course.

23/10/23

minimumCoverageEA.ipynb

```
import numpy as np
import torch as t
import matplotlib.pyplot as plt

device = t.device("cuda") if t.cuda.is_available() else t.device("cpu")
print(device)

PROBLEM_DIMENSION = 30
NUM_SAMPLE = 50

def check_sets_feasibility(samples):
    return t.sum(samples, dim=0).all()

THRESHOLD = 0.3
samples = t.rand(NUM_SAMPLE, PROBLEM_DIMENSION, device=device) < THRESHOLD
# while not check_sets_feasibility(samples):
#     samples = t.rand(NUM_SAMPLE, PROBLEM_DIMENSION, device=device) < THRESH
OLD

assert check_sets_feasibility(samples), "Problem not solvable"

samples

class Population:
    def __init__(self, population_len, genome_len, samples, is_highest_best=T
rue, genomes=None, crossover=None, mutation_rate = 0.03):
        self.population_len = population_len
        self.genome_len = genome_len
        self.generation = -1
        self.is_highest_best = is_highest_best
        self.genomes = genomes if genomes is not None else t.rand(population_
len, genome_len, device=device) >= 0.5
        self.fitness = None
        self.probability = None
        self.mutation_rate = mutation_rate
        self.crossover_function = None
        if crossover == None or crossover == "uniform":
            self.crossover_function = self.uniform_crossover
        elif crossover == "one_point":
```

```
            self.crossover_function = self.one_point_crossover
        #
        self.samples = samples.expand(self.population_len, -1, -1)

        self.updatePopulation()

    def __str__(self):
        strs = list()
        strs.append(f'Generation: {self.generation}')
        strs.append(f'Genomes: {self.genomes}')
        strs.append(f'Best fitness: {self.get_best_fitness()}, of id: {self.g
et_best_id()}')
        return '\n'.join(strs)

    def updatePopulation(self):
        self.generation += 1
        self.set_fitness()
        self.set_probability()

    def get_phenotype(self):
        return t.mul(self.samples, self.genomes.unsqueeze(-1)).sum(dim=1)


    def set_fitness(self):
        # print("SAMPLES:", self.samples)
        res = self.get_phenotype()
        # print("RES:", res)
        # print("Genomes:", self.genomes)
        used_samples = self.genomes.sum(dim=1)
        # print("Used samples:", used_samples)
        self.fitness = (res == 0).sum(dim=1) * (self.samples.size()[2] + 1) +
used_samples
        # print("Fit:", self.fitness)

    def set_probability(self):
        if self.is_highest_best:
            self.probability = self.fitness / t.sum(self.fitness)
        else:
            self.probability = 1 / self.fitness
            self.probability.div_(t.sum(self.probability))
        # print("Prob:", self.probability)

    def get_best_id(self):
        return t.argmax(self.probability)

    def get_best_fitness(self):
        return self.fitness[self.get_best_id()]

    def get_best_genome(self):
        return self.genomes[self.get_best_id(), :]
```

```
    def evolve(self):
        self.crossover_function()
        self.mutation()
        self.updatePopulation()
        # print(self)
        # print("Best fitness:", self.fitness[self.get_best_id()])

    def evolve_for_generations(self, generations):
        for _ in range(generations):
            self.evolve()
        print(self)
        # print("Best fitness:", self.get_best_fitness())

    def get_parents(self):
        parents = self.probability.expand(self.population_len, self.populatio
n_len).multinomial(2)
        # print("Parents:", parents)
        p1 = self.genomes[parents[:,0],:]
        p2 = self.genomes[parents[:,1],:]
        return p1, p2

    def one_point_crossover(self):
        raise("To be implemented")
        # p1, p2 = self.get_parents()
        # u = t.rand(self.population_len, device=device) * self.genome_len

    def uniform_crossover(self):
        p1, p2 = self.get_parents()
        mask = t.rand(self.population_len, self.genome_len, device=device) >=
0.5
        self.genomes = p1 * mask + p2 * ~mask

    def mutation(self):
        mutation = t.rand(self.population_len, self.genome_len, device=device
) < self.mutation_rate
        # print("Mutation:", mutation)
        self.genomes = t.where(mutation, ~self.genomes, self.genomes)
        # print("Genomes:", self.genomes)

population_len = 10
population = Population(population_len, NUM_SAMPLE, samples, is_highest_best=
False)
print(samples)
print(population)

population.evolve_for_generations(500)

population.get_phenotype()[254]
```

Dimitri Masetta 306130

Hallowen Challenge

01/11/23

Hallowen.ipynb

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

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
```

Halloween Challenge

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

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

```
x = make_set_covering_problem(100, 100, .3)
print("Element at row=42 and column=42:", x[42, 42])

PROBLEM_SIZE = 1000
NUM_SETS = 1000
SETS_TRUE_PROBABILITY = 0.3

SETS = make_set_covering_problem(PROBLEM_SIZE, NUM_SETS, SETS_TRUE_PROBABILIT
Y)

SETS.A

def fitness(state):
    result_coverage = coverage(state)
```

```
    return -(state.sum()+ (result_coverage == False).sum() *  PROBLEM_SIZE)

def coverage(state):
    return (SETS * state).sum(axis=0)

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

def tweak_and_feat(state, state_fitness):
    index = randint(0, NUM_SETS - 1)
    state[index] = not state[index]
    new_fitness = fitness(state)
    if new_fitness > state_fitness:
        return True, state, new_fitness
    else:
        state[index] = not state[index]
        return False, state, state_fitness
```

Hill climber

```
# random() < 0.005
current_state = [False for _ in range(NUM_SETS)]
current_state = np.expand_dims(current_state, 0).transpose()
fitness_current = fitness(current_state)
print(f"Start fitness {fitness_current}")

max_step = 5_000
count_steady = max_step/10
count_equal = 0

for step in range(max_step):
    count_equal += 1
    res, st, fit = tweak_and_feat(current_state, fitness_current)
    if res:
        count_equal = 0
        current_state, fitness_current = st, fit
        print(f"Fit {fitness_current} Step {step}")
    if count_equal > count_steady:
        print("steady state reached")
        break

print(f"Resolved {(coverage(current_state) == False).sum() == 0} with {curren
t_state.sum()} in {step} step")
coverage(current_state)
```

Hill climber best of three

```
current_state = [False for _ in range(NUM_SETS)]
current_state = np.expand_dims(current_state, 0).transpose()
fitness_current = fitness(current_state)
print(f"Start fitness {fitness_current}")

max_step = 5_000
count_steady = max_step/10
count_equal = 0

for step in range(max_step):
    count_equal += 1
    state_new = tweak(current_state)
    fitness_new = fitness(state_new)
    for _ in range(4):
        s_new = tweak(current_state)
        f_new = fitness(state_new)
        if f_new > fitness_new:
            state_new, fitness_new = s_new, f_new

    if fitness_new > fitness_current:
        count_equal = 0
        current_state, fitness_current = state_new, fitness_new
        print(f"Fit {fitness_current} Step {step}")
    if count_equal > count_steady:
        print("steady state reached")
        break

print(f"Resolved {(coverage(current_state) == False).sum() == 0} with {curren
t_state.sum()} in {step * 3} step")
coverage(current_state)
```

Simulated annealing

```
def probability(current_fitness, new_fitness, t):
    if new_fitness > current_fitness:
        return 1
    return math.exp(-(current_fitness - new_fitness) / t) / 2

current_state = [False for _ in range(NUM_SETS)]
current_state = np.expand_dims(current_state, 0).transpose()
fitness_current = fitness(current_state)
print(f"Start fitness {fitness_current}")

max_step = 5_000
count_steady = max_step/10
count_equal = 0
```

```
max_temp = max_step / 10
temperature1 = [1-(i+1)/max_step for i in range(max_step)]
temperature2 = [max_temp/(i+1) for i in range(max_step)]

temperature = temperature2

for step in range(max_step):
    count_equal += 1
    state_new = tweak(current_state)
    fitness_new = fitness(state_new)
    prob = probability(fitness_current, fitness_new, temperature[step])
    if random() < prob:
        count_equal = 0
        current_state, fitness_current = state_new, fitness_new
        print(f"Fit {fitness_current} Step {step} Prob {prob}")
    if count_equal > count_steady:
        print("steady state reached")
        break

print(f"Resolved {(coverage(current_state) == False).sum() == 0} with {curren
t_state.sum()} in {step} step")
coverage(current_state)
```

Dimitri Masetta 306130

LAB 2(nim)

17/11/23

lab2-nim.ipynb

Task

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

Instructions

- Create the directory `lab2` inside your personal course repository for the course
- Put a `README.md` and your solution (all the files, code and auxiliary data if needed)

Notes

- Working in group is not only allowed, but recommended (see: *Ubuntu* and *Cooperative Learning*). Collaborations must be explicitly declared in the `README.md`.
- *Yanking* from the internet is allowed, but sources must be explicitly declared in the `README.md`.

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

The *Nim* and *Nimply* classes

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

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
```

Match

The match class is used to define one or more match between different strategies

```
class Match:
    def __init__(self, num_rows, player0, player1, verbose=False):
        self.nim = Nim(num_rows)
        self.strategies = (player0, player1)
        if verbose:
            logging.getLogger().setLevel(logging.DEBUG)
        else:
            logging.getLogger().setLevel(logging.INFO)

    def play(self, no_mark=False, starting_player=0):
        if no_mark:
            nim = deepcopy(self.nim)
        else:
             nim = self.nim
        player = starting_player
        logging.debug(f"init : {nim}")
        while nim:
            ply = self.strategies[player](nim)
            logging.debug(f"ply: player {player} plays {ply}")
            nim.nimming(ply)
            logging.debug(f"status: {nim}")
            player = 1 - player
        logging.debug(f"status: Player {player} won!")
        return player

    def play_n(self, n_matches, change_starting_player=True):
        player0_win = 0
        player1_win = 0
        for i in range(n_matches):
            w = self.play(no_mark=True, starting_player=i%2 if change_startin
g_player else 0)
```

```python
            if w == 0:
                player0_win += 1
            else:
                player1_win += 1
        logging.debug(f"Player 0 won {player0_win} times")
        logging.debug(f"Player 1 won {player1_win} times")
        return player0_win, player1_win
```

Sample (and silly) startegies

```python
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}

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
```

Expert agent (Task2.1)

a.k.a. Real Optimal

An agent using fixed rules based on *nim-sum* that is the real optimal and always win

```
def real_optimal(state: Nim) -> Nimply:
    rows_with_more_than_two = [id for id, dim in enumerate(state.rows) if dim
>=2]
    if len(rows_with_more_than_two) == 1:
        rows_with_one = len([None for dim in state.rows if dim==1])
        row_id = rows_with_more_than_two[0]
        if rows_with_one%2 == 0:
            return Nimply(row_id, state.rows[row_id] - 1)
        else:
            return Nimply(row_id, state.rows[row_id])
    else:
        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
```

ES (Task2.2)

Rules

List of 6 rules that the Evolutionary Strategy will try to select based on an evolving probability.

```
#######Util###########
def min_id_of_list(my_list, non_zero=False):
    if non_zero:
        lower = 0
        id = 0
        for i, val in enumerate(my_list):
            if (val < lower and val != 0) or lower == 0:
                lower = val
                id = i
        return  id
    else:
        return min(range(len(my_list)), key=my_list.__getitem__)

def max_id_of_list(my_list):
```

```
    return max(range(len(my_list)), key=my_list.__getitem__)
#######################


def empty_lower(state: Nim) ->Nimply:
    index = min_id_of_list(state.rows, non_zero=True)
    return Nimply(index, state.rows[index])

def empty_greater(state: Nim) ->Nimply:
    index = max_id_of_list(state.rows)
    return Nimply(index, state.rows[index])

def leave_one_to_lower(state: Nim) ->Nimply | None:
    index = min_id_of_list(state.rows, non_zero=True)
    if state.rows[index] == 1:
        return None
    return Nimply(index, state.rows[index] - 1)
def leave_one_to_greater(state: Nim) ->Nimply | None:
    index = max_id_of_list(state.rows)
    if state.rows[index] == 1:
        return None
    return Nimply(index, state.rows[index] - 1)

def remove_one_to_lower(state: Nim) ->Nimply:
    index = min_id_of_list(state.rows, non_zero=True)
    return Nimply(index, 1)

def remove_one_to_greater(state: Nim) ->Nimply:
    index = max_id_of_list(state.rows)
    return Nimply(index, 1)


rules = [empty_lower, empty_greater, leave_one_to_lower, leave_one_to_greater
, remove_one_to_lower, remove_one_to_greater]
```

Agent that play the game, based on a list of probability

```
class Agent:
    def __init__(self, rules_probability, rules):
        self.rules_probability = rules_probability / sum(rules_probability)
        self.rules = rules

    def move(self, state: Nim) -> Nimply:
        #Every move a list of rules, ordered based on the evolute probability
, are extracted and are applied until one of them give a valid result.
        rules_ordered = np.random.choice(self.rules, len(self.rules), replace
=False, p=self.rules_probability)
        for rule in rules_ordered:
            out = rule(state)
```

```
            if out is not None:
                return out
```

Fitness function to evaluate the agent against a list of opponents

```
opponents = [gabriele, pure_random, optimal]
match_per_opponents = 15
row_count = 5

def fitness(agent):
    agent_win = 0
    for opponent in opponents:
        match = Match(row_count, agent.move, opponent)
        win_0, _ = match.play_n(match_per_opponents)
        agent_win += win_0
    return agent_win / (match_per_opponents * len(opponents))
```

Adaptive (μ,λ)-ES

```
population_len = 15 #u
offspring_len = 100 #lambda
mu = 0.5
dimensions = len(rules) + 1
generations = 200 #1_000_000 // offspring_len

population = np.random.random((population_len, dimensions))
best_fitness = None
best_offspring = None
history = list()

for gen in tqdm(range(generations)):
    offspring = population[np.random.randint(0, population_len, size=(offspri
ng_len, ))]

    #mutate mu
    offspring[:, -1] = np.random.normal(
            loc=offspring[:, -1], scale=0.2
        )
    #clamp the value of mu at 1e-5 (value of mu to low are not useful because
lead to no mutation)
    offspring[offspring[:, -1] < 1e-5, -1] = 1e-5

    #mutate values
    offspring[:, 0:-1] = np.random.normal(
            loc=offspring[:, 0:-1], scale=offspring[:, -1].reshape(-1, 1)
        )
    #clamp the value of the values at 1e-5 (values lower than 0 are not valid
probability)
    offspring[offspring < 1e-5] = 1e-5
```

```
    fit = np.array([fitness(Agent(offspring[i, 0:-1], rules)) for i in range(
population_len)])
    offspring = offspring[fit.argsort()]

    if best_fitness is None or best_fitness < np.max(fit):
            best_fitness = np.max(fit)
            best_offspring = offspring[0,:-1]
            history.append((gen, best_fitness))
    population = np.copy(offspring[-population_len:])

logging.info(
    f"Best solution: {history[len(history) - 1][1]}"
)

history = np.array(history[:-1])
plt.figure(figsize=(14, 4))
plt.plot(history[:, 0], history[:, 1], marker=".")
```

Result

The result of the agent based on the evolutionary strategy is a 82.2% of win rate against the three standard opponent (gabriele, pure_random, optimal)

Dimitri Masetta 306130

PEER REVIEW LAB 2

24/11/23

github.com/gregorio-nic

Peer review Lab 2

Preface

The code is generally well written and well documented thanks to the useful readme, that easly allow to understand the structure of the proposed solution. Be careful with the use of deepcopy that in some cases is unnecessary and really slow the ES algorithm.
Let's now analyze more in details the two tasks.

Task 2.1 - Expert System

There is little to say, an expert system has not been implemented.

Task 2.2 - Evolution Strategy

Starting from the genotype, I appreciate your original idea to use a list and a matrix for the probabilities respectively of the row to choose and the number of matchsticks to pick. The only draw back can be that the dimension of the genotype grow linearly with the number of matchsticks. As further advice, save the two part of the genotype as numpy arrays and not lists, so as to avoid the implicit conversion that appen each time you call a numpy function and speed up your algorithm(for the matrix, given your implementation, you can also try a sparse array to reduce the memory footprint).
With regard to the mutation/evolution I have some concerns:

- You use a static sigma of 2 as mutation step, I don't know from where you derived this value, but in general a far more performant approach than a static sigma is to use a self-adaptation strategy, that allow to balance exploration and exploitation.
- Your mutation can be large because of the static sigma, listed above, and the fact that you use the abs function that lead to huge change when the value is near zero(consider a gene with value 0.01 a mutation gene of -0.5, after the abs you obtain a value of 0.49). A better approach would be to clamp the min value slightly larger than zero, like at 1e-5 (from the example above instead of 0.49 you will obtain 1e-5, that is much more similar to 0.01)
- As small advice, you can avoid the normalization during this phase, since you do it again in the evaluation.

For the evaluation phase, good job, work as expected. Maybe you can add gabriele and the optimal(as it is not a real optimal) as adversary, so your ES agent can learn against different

playstyle.
In general you have made a good job in the implementation of the 1, lambda(be careful you implemented a comma not a plus) and mu+lambda strategies, with good results as support.

github.com/beatrice-occhiena

Peer review Lab2

Preface

Where to start from, your code is very well documented and really easy to read, allowing anyone to easily understand your intentions. You have made an amazing job in that regards, well done!! Let's now analyze more in details the two tasks.

Task 2.1 - Expert System

There is little to say about this, you nailed its implementation; using the optimal move in every case except for the situation that you have defined of scarcity, in which there is only one row with more than one matchstick.
I also really liked the implementation of other companion agent, which can improve learning in the task 2.2 by comparing the ES to agents with different playstyles.

Task 2.2 - Evolution Strategy

Starting from the ruleset, I appreciate your idea for the definition of the ruleset using also a condition to enable the activation of the rule, nice work. Given your innovative approach towards the ruleset and the new knowledge about GP, I can imagine an improved version of your agent able to define more intricate rules.
For the Evolution Strategy itself, it is generally correct and well implemented, but there are some points where improvements are possible:

- In the parent selection(5.1) you come with a sorted population from the survival selection(5.3) and then you perform a top slice of the populations, in this way you obtain purely deterministic parent selection in contrast with the standard random parent selection that is usually exploited in ES. This does not affect the functioning of the agent but can lead to premature convergence, as you lose that part of randomness that give at every memeber of the population a chance to be choose.

Dimitri Masetta 306130

- The section in which you do the self-adaptation(5.4) must be moved before the mutation, because otherwise you lose the correlation between sigma and the fitness, as the fitness will be calculated on the values based on the old sigma.
- You have described your code as a comma strategy but in the survival selection(5.3) you have implemented a plus strategy.

Dimitri Masetta 306130

LAB 9

03/12/23

lab9.ipynb

LAB9

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

Deadlines:

- Submission: Sunday, December 3 (*CET*)
- Reviews: Sunday, December 10 (*CET*)

Notes:

- Reviews will be assigned on Monday, December 4
- You need to commit in order to be selected as a reviewer (ie. better to commit an empty work than not to commit)

```
from random import choices
from collections.abc import Callable, Sequence
import numpy as np

import lab9_lib
from population import Population
from population_builder import PopulationBuilder
from island import Island

fitness = lab9_lib.make_problem(1)
for n in range(10):
    ind = choices([0, 1], k=50)
    print(f"{''.join(str(g) for g in ind)}: {fitness(ind):.2%}")

print(fitness.calls)

def run_population(problem_fitness, generations, builder):
    population = (builder
                  .add_fitness_function(fitness_function=problem_fitness)
                  .build())
    population.run_for_generations_or_until_no_upgrades(generations, log_data
=False, n_generations_without_upgrade=100)

    print(f"Generations ran {population.generations_ran}")
    print(f"Max fitness {population.max_fitness}")
```

```
    print(f"Fitness call {problem_fitness.calls / 1000}k")
    population.log_history_fitness()
```

Population

Thanks, to the population builder, we can easily build a population, in this case after many test we develop this population that perform discretely in all problem(we the problem of premature convergence in the problems 5 and 10). Be aware that there are not Individual class but only a Population, so that every function applied to it is made with numpy function and not even a single for loop is present(this dramatically increase speed but reduce code readability).

```
#Population builder
builder = (PopulationBuilder()
           .initialize_random(population_size=100, genome_len=1000)
           .add_parents_selector_tournament(tournament_size=10, offspring_siz
e=150)
           .add_survivals_selector_generational()
           .set_recombination_and_mutation_mutualexclusive(probability_recomb
ination_over_mutation=.2)
           .add_mutation_single_flip()
           .add_recombination_uniform_xover())
```

```
run_population(lab9_lib.make_problem(1), 1500, builder)
```

```
run_population(lab9_lib.make_problem(2), 1500, builder)
```

```
run_population(lab9_lib.make_problem(5), 1500, builder)
```

```
run_population(lab9_lib.make_problem(10), 1500, builder)
```

Islands

We have developed a solution base on island to try to mitigate the previous problem.

```
def run_island(population_size, genome_len, epoch, problem_fitness, builders
):
    builders = [b.add_fitness_function(fitness_function=problem_fitness) for
b in builders]
    island = Island(population_size, genome_len, builders)

    island.run(epochs=epoch)
    print(f"Max fitness {island.max_fitness}")
    print(f"Fitness call {problem_fitness.calls / 1000}k")
    island.log_history_fitness()
```

```
population_size = 50
genome_len = 1000
```

```
problem_fitness = lab9_lib.make_problem(2)

builders =[ (PopulationBuilder()
            .add_parents_selector_tournament(tournament_size=10, offspring_si
ze=70)
            .add_survivals_selector_generational()
            .add_recombination_uniform_xover()
            .add_mutation_single_flip()
            .set_recombination_and_mutation_mutualexclusive(probability_recom
bination_over_mutation=0.2)),
            (PopulationBuilder()
            .add_parents_selector_tournament(tournament_size=2, offspring_siz
e=70)
            .add_survivals_selector_generational()
            .add_recombination_one_point_xover()
            .add_mutation_single_flip()
            .set_recombination_and_mutation_mutualexclusive(probability_recom
bination_over_mutation=0.7)),
            (PopulationBuilder()
            .add_parents_selector_tournament(tournament_size=10, offspring_si
ze=30)
            .add_survivals_selector_steady_state()
            .add_recombination_one_point_xover()
            .add_mutation_single_flip()
            .set_mutation_sequential_to_recombination(probability_mutation=0.
3))]

run_island(population_size, genome_len, 6, lab9_lib.make_problem(1), builders
)

run_island(population_size, genome_len, 20, lab9_lib.make_problem(2), builder
s)

run_island(population_size, genome_len, 10, lab9_lib.make_problem(5), builder
s)

run_island(population_size, genome_len, 10, lab9_lib.make_problem(10), builde
rs)
```

lab9_lib.py

```
from abc import abstractmethod


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

```
    @property
    @abstractmethod
    def x(self):
        pass

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

    @staticmethod
    def onemax(genome):
        return sum(bool(g) for g in genome)

    def __call__(self, genome):
        self._calls += 1
        fitnesses = sorted((AbstractProblem.onemax(genome[s :: self.x]) for s
in range(self.x)), reverse=True)
        val = sum(f for f in fitnesses if f == fitnesses[0]) - sum(
            f * (0.1 ** (k + 1)) for k, f in enumerate(f for f in fitnesses
if f < fitnesses[0])
        )
        return val / len(genome)


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

    return Problem()
```

population.py

```
from collections.abc import Callable
import numpy as np
from matplotlib import pyplot as plt


class Population:
    def __init__(self,
                 population_genome: np.ndarray,
                 parents_selector: Callable[[np.ndarray, np.ndarray],
(np.ndarray, np.ndarray)],
                 survivals_selector: Callable[
                     [np.ndarray, np.ndarray, np.ndarray, np.ndarray],
(np.ndarray, np.ndarray)],
```

```
                 offspring_generator: Callable[[np.ndarray], np.ndarray],
                 fitness_calculator: Callable[[np.ndarray], np.ndarray]):

        self._population_genome = population_genome
        self._parents_selector = parents_selector
        self._survivals_selector = survivals_selector
        self._offspring_generator = offspring_generator
        self._fitness_calculator = fitness_calculator
        self._population_fitness =
self._fitness_calculator(population_genome)
        sorting_indices = self._population_fitness.argsort()[::-1]
        self._population_genome = self._population_genome[sorting_indices]
        self._population_fitness = self._population_fitness[sorting_indices]
        self._generations_ran = 0
        self._history_fitness = [(0, self.max_fitness), ]

    def __str__(self):
        return f"""Generation: {self._generations_ran}
Max fitness: {self.max_fitness}"""

    def log_data(self):
        print(self)
        print(f"Genome\n{self._population_genome}")
        print(f"Fitness\n{self._population_fitness}")

    @property
    def genome(self):
        return self._population_genome

    @property
    def fitness(self):
        return self._population_fitness

    @property
    def generations_ran(self):
        return self._generations_ran

    @property
    def max_fitness(self):
        return self._population_fitness[0]

    @property
    def history_fitness(self):
        return self._history_fitness

    def log_history_fitness(self):
        history = np.array(self._history_fitness)
        print(history)
        plt.figure(figsize=(14, 4))
        plt.plot(history[:, 0], history[:, 1], marker=".")
```

```
    def run_generation(self):
        self._generations_ran += 1
        parents_genome, parents_fitness =
self._parents_selector(self._population_genome, self._population_fitness)
        offspring_genome = self._offspring_generator(parents_genome)
        offspring_fitness = self._fitness_calculator(offspring_genome)
        self._population_genome, self._population_fitness =
self._survivals_selector(self._population_genome,

self._population_fitness,

offspring_genome,

offspring_fitness)
        self._history_fitness.append((self._generations_ran,
self.max_fitness))

    def run_for_generations(self, n_generations, log_data=False):
        for _ in range(n_generations):
            self.run_generation()
            if log_data:
                self.log_data()

    def run_for_generations_or_until_no_upgrades(self, n_generations,
n_generations_without_upgrade= None, log_data=False):
        n_generations_without_upgrade = n_generations_without_upgrade if
n_generations_without_upgrade is not None else int(n_generations/4)
        max_fit = 0
        gen_without_upgrade = 0
        for i in range(n_generations):
            if gen_without_upgrade > n_generations_without_upgrade:
                break
            self.run_generation()
            if max_fit < self.max_fitness:
                max_fit = self.max_fitness
                gen_without_upgrade = 0
            else:
                gen_without_upgrade += 1
            if log_data:
                self.log_data()
```

island.py

```
from collections.abc import Callable, Sequence
import numpy as np
from population_builder import PopulationBuilder
from population import Population
```

```
from matplotlib import pyplot as plt



class Island:
    # Builders must be without initializations
    def __init__(self, population_size, genome_len, builders:
Sequence[PopulationBuilder], repeat_builder=1, ):
        self._builders: Sequence[PopulationBuilder] = [b for _ in
range(repeat_builder) for b in builders]
        self._populations = [b.initialize_random(population_size,
genome_len).build() for b in self._builders]
        self._population_size = population_size
        self._rng = np.random.default_rng()
        self._history_fitness = []

    @property
    def max_population(self):
        return max(self._populations, key=lambda p: p.max_fitness)

    @property
    def max_fitness(self):
        return self.max_population.max_fitness

    def log_history_fitness(self):
        history = np.array(self._history_fitness)
        plt.figure(figsize=(14, 4))
        plt.plot(history[:, 0], history[:, 1], marker=".")

    def run(self, epochs, generation_per_epoch=100, migrants=None,
log_data=False):
        migrants = migrants if migrants is not None else
int(self._population_size / 10)
        assert migrants != 0, "There are no migrants"
        assert migrants < self._population_size / 2, "Invalid configuration,
is not possible a number of migrant greater than half of the population size"
        for epoch in range(epochs):
            self._run_epoch(generation_per_epoch, log_data)
            if log_data:
                self._log_epoch(epoch)
            self._history_fitness.append((epoch, self.max_fitness))
            self._migrate(migrants)

    def _run_epoch(self, generations, log_data):
        for population in self._populations:
            population.run_for_generations_or_until_no_upgrades(generations,
log_data=log_data)

    def _migrate(self, migrants):
        permutation = self._rng.permutation(len(self._populations))
```

```
        new_populations_genomes = []
        for i in range(len(self._populations)):
            new_genome = self._populations[i].genome
            new_genome[-migrants:, :] =
self._populations[permutation[i]].genome[:migrants, ]
            new_populations_genomes.append(new_genome)
        self._populations =
[b.initialize_with_genome(new_populations_genomes[i]).build() for i, b in
                                 enumerate(self._builders)]

    def _log_epoch(self, epoch):
        print(f"####EPOCH {epoch}####")
        for i, population in enumerate(self._populations):
            print(f"Population #{i}\n{population}")
```

mutator.py

```
from collections.abc import Callable
import numpy as np


class Mutator:
    _rng = np.random.default_rng()

    @staticmethod
    def single_flip_mutation(parents_genome: np.ndarray) -> np.ndarray:
        col_id_to_change = Mutator._rng.integers(0, parents_genome.shape[1],
(parents_genome.shape[0]))
        row_id_range = np.arange(parents_genome.shape[0])
        parents_genome[row_id_range, col_id_to_change] =
~parents_genome[row_id_range, col_id_to_change]
        return parents_genome
```

offspring_generator.py

```
from collections.abc import Callable
import numpy as np


class OffspringGenerator:
    @staticmethod
    def sequential_generator(probability_mutation: float, parents_genome:
np.ndarray,
                             recombinator: Callable[[np.ndarray], np.ndarray]
| None,
                             mutator: Callable[[np.ndarray], np.ndarray] |
```

```
None) -> np.ndarray:
        _ = recombinator(parents_genome)
        mutate_until_index = int(parents_genome.shape[0] *
probability_mutation)
        _ = mutator(parents_genome[:mutate_until_index, :])
        return parents_genome

    @staticmethod
    def mutualexclusive_generator(probability_recombination_over_mutation:
float, parents_genome: np.ndarray,
                                  recombinator: Callable[[np.ndarray],
np.ndarray] | None,
                                  mutator: Callable[[np.ndarray], np.ndarray]
| None) -> np.ndarray:
        recombinate_until_index = int(parents_genome.shape[0] *
probability_recombination_over_mutation)
        recombinate_until_index += 0 if recombinate_until_index % 2 == 0 else
1
        _ = recombinator(parents_genome[:recombinate_until_index, :])
        _ = mutator(parents_genome[recombinate_until_index:, :])
        return parents_genome
```

parents_selector.py

```
from collections.abc import Callable
import numpy as np


class ParentSelector:
    _rng = np.random.default_rng()

    @staticmethod
    def tournament_selector(tournament_size: int, offspring_size: int,
population_genome: np.ndarray,
                            population_fitness: np.ndarray) -> (np.ndarray,
np.ndarray):
        pool = ParentSelector._rng.integers(0, population_fitness.shape[0],
(offspring_size, tournament_size))
        fitpool = population_fitness[pool]
        fitpool_sorted_indices = np.argsort(fitpool, axis=1)
        parents_indices = np.take_along_axis(pool, fitpool_sorted_indices,
axis=1)[:, -1]
        return population_genome[parents_indices, :],
population_fitness[parents_indices]

    @staticmethod
    def roulette_wheel(offspring_size: int, population_genome: np.ndarray,
population_fitness: np.ndarray) -> (
```

```
    np.ndarray, np.ndarray):
        # TODO
        pass
```

population_builder.py

```
from collections.abc import Callable
import numpy as np
from population import Population
from parents_selector import ParentSelector
from survivals_selector import SurvivalSelector
from offspring_generator import OffspringGenerator
from recombinator import Recombinator
from mutator import Mutator


class PopulationBuilder:
    def __init__(self):
        # Initialization
        self.initialization_function: Callable[[], np.ndarray] | None = None
        self.population_size = 0
        self.offspring_size = 0
        # Selectors
        self.parents_selector: Callable[[np.ndarray, np.ndarray],
(np.ndarray, np.ndarray)] | None = None
        self.survivals_selector: Callable[[np.ndarray, np.ndarray,
np.ndarray, np.ndarray], (
        np.ndarray, np.ndarray)] | None = None
        self.is_survivals_selector_generational = False
        # Offspring generator
        self.recombinator: Callable[[np.ndarray], np.ndarray] | None = None
        self.mutator: Callable[[np.ndarray], np.ndarray] | None = None
        self.partial_offspring_generator: Callable[[np.ndarray,
Callable[[np.ndarray], np.ndarray] | None, Callable[
            [np.ndarray], np.ndarray] | None], np.ndarray] | None = None
        # Fitness
        self.fitness_calculator: Callable[[np.ndarray], np.ndarray] | None =
None
        # Util
        self.rng = np.random.default_rng()

    ####################
    ## INITIALIZATION ##
    ####################
    def initialize_with_genome(self, population_genome):
        def initialize_by_genome() -> np.ndarray:
            return population_genome
```

```
        self.initialization_function = initialize_by_genome
        self.population_size = population_genome.shape[0]
        return self

    def initialize_random(self, population_size, genome_len,
zero_probability=0.5):
        def initialize_random_genome() -> np.ndarray:
            return self.rng.choice((True, False), (population_size,
genome_len),
                                             p=(1 - zero_probability,
zero_probability))

        self.initialization_function = initialize_random_genome
        self.population_size = population_size
        return self

    ######################
    ## PARENT SELECTORS ##
    ######################
    def add_parents_selector_tournament(self, tournament_size,
offspring_size):
        def function(population_genome: np.ndarray, population_fitness:
np.ndarray) -> (np.ndarray, np.ndarray):
            return ParentSelector.tournament_selector(tournament_size,
offspring_size, population_genome,
                                                                    population_fitness)

        self.parents_selector = function
        self.offspring_size = offspring_size
        return self

    def add_parents_selector_roulette_wheel(self, offspring_size):
        def function(population_genome: np.ndarray, population_fitness:
np.ndarray) -> (np.ndarray, np.ndarray):
            return ParentSelector.roulette_wheel(offspring_size,
population_genome, population_fitness)

        self.parents_selector = function
        self.offspring_size = offspring_size
        return self

    ########################
    ## SURVIVAL SELECTORS ##
    ########################
    def add_survivals_selector_steady_state(self):
        self.survivals_selector = SurvivalSelector.steady_state_selector
        return self

    def add_survivals_selector_generational(self):
        self.survivals_selector = SurvivalSelector.generational_selector
```

```
        self.is_survivals_selector_generational = True
        return self

    ##########################
    ## OFFSPRING GENERATOR ##
    ##########################
    def add_mutation_single_flip(self):
        self.mutator = Mutator.single_flip_mutation
        return self

    def add_recombination_one_point_xover(self):
        self.recombinator = Recombinator.one_point_xover
        return self

    def add_recombination_uniform_xover(self):
        self.recombinator = Recombinator.uniform_xover
        return self

    def set_mutation_sequential_to_recombination(self, probability_mutation):
        def function(parents_genome: np.ndarray, recombinator:
Callable[[np.ndarray], np.ndarray] | None,
                     mutator: Callable[[np.ndarray], np.ndarray] | None) ->
np.ndarray:
            return
OffspringGenerator.sequential_generator(probability_mutation, parents_genome,
recombinator, mutator)

        self.partial_offspring_generator = function
        return self

    def set_recombination_and_mutation_mutualexclusive(self,
probability_recombination_over_mutation):
        def function(parents_genome: np.ndarray, recombinator:
Callable[[np.ndarray], np.ndarray] | None,
                     mutator: Callable[[np.ndarray], np.ndarray] | None) ->
np.ndarray:
            return
OffspringGenerator.mutualexclusive_generator(probability_recombination_over_m
utation, parents_genome,
                                                                 recombinator,
mutator)

        self.partial_offspring_generator = function
        return self

    #############
    ## FITNESS ##
    #############
    def add_fitness_function(self, fitness_function):
        def fitness(population_genome: np.ndarray) -> np.ndarray:
```

```
            return np.apply_along_axis(fitness_function, axis=1,
arr=population_genome)

        self.fitness_calculator = fitness
        return self

    ###########
    ## BUILD ##
    ###########
    def build(self):
        assert not self.is_survivals_selector_generational or
self.offspring_size >= self.population_size, "Invalid configuration, if the
survival selector is generational the offspring size must be greater or equal
than the population size"

        def offspring_generator(parents_genome: np.ndarray) -> np.ndarray:
            return self.partial_offspring_generator(parents_genome,
self.recombinator, self.mutator)

        return Population(population_genome=self.initialization_function(),
parents_selector=self.parents_selector,
                          survivals_selector=self.survivals_selector,
offspring_generator=offspring_generator,
                          fitness_calculator=self.fitness_calculator)
```

recombination.py

```
from collections.abc import Callable
import numpy as np


class Recombinator:
    _rng = np.random.default_rng()

    @staticmethod
    def one_point_xover(parents_genome: np.ndarray) -> np.ndarray:
        parent_count = parents_genome.shape[0]
        assert parent_count % 2 == 0, "The number of parents for the
crossover must be even"
        genome_len = parents_genome.shape[1]
        couple_count = int(parent_count / 2)
        points_of_xover = Recombinator._rng.integers(0, genome_len - 1,
couple_count).reshape(couple_count, 1)
        mask_change_per_couple = np.arange(genome_len) <= points_of_xover
        return Recombinator._mask_couple_to_recombination(parents_genome,
mask_change_per_couple)

    @staticmethod
```

```
    def uniform_xover(parents_genome: np.ndarray) -> np.ndarray:
        parent_count = parents_genome.shape[0]
        assert parent_count % 2 == 0, "The number of parents for the
crossover must be even"
        genome_len = parents_genome.shape[1]
        couple_count = int(parent_count / 2)
        mask_change_per_couple = Recombinator._rng.choice((True, False),
(couple_count, genome_len))
        return Recombinator._mask_couple_to_recombination(parents_genome,
mask_change_per_couple)

    @staticmethod
    def _mask_couple_to_recombination(parents_genome,
mask_change_per_couple):
        parent_count = parents_genome.shape[0]
        genome_len = parents_genome.shape[1]
        mask_change_per_parents = np.hstack((mask_change_per_couple,
~mask_change_per_couple)).reshape(-1, genome_len)
        mask = mask_change_per_parents +
np.broadcast_to(np.repeat(np.arange(0, parent_count, 2), 2),
                                                         (genome_len,
parent_count)).T
        return np.take_along_axis(parents_genome, mask, axis=0)
```

survival_selector.py

```
from collections.abc import Callable
import numpy as np


class SurvivalSelector:
    @staticmethod
    def steady_state_selector(oldgen_genome: np.ndarray, oldgen_fitness:
np.ndarray, offspring_genome: np.ndarray,
                              offspring_fitness: np.ndarray) -> (np.ndarray,
np.ndarray):
        population_size = oldgen_genome.shape[0]
        population_genome = np.vstack((oldgen_genome, offspring_genome))
        population_fitness = np.concatenate((oldgen_fitness,
offspring_fitness))
        sorting_indices = population_fitness.argsort()[::-1]
        population_genome = population_genome[sorting_indices]
        population_fitness = population_fitness[sorting_indices]
        return population_genome[:population_size, :],
population_fitness[:population_size]

    @staticmethod
    def generational_selector(oldgen_genome: np.ndarray, oldgen_fitness:
```

```
np.ndarray, offspring_genome: np.ndarray,
                              offspring_fitness: np.ndarray) -> (np.ndarray,
np.ndarray):
        population_size = oldgen_genome.shape[0]
        sorting_indices = offspring_fitness.argsort()[::-1]
        population_genome = offspring_genome[sorting_indices]
        population_fitness = offspring_fitness[sorting_indices]
        return population_genome[:population_size, :],
population_fitness[:population_size]
```

Dimitri Masetta 306130

PEER REVIEW LAB 9

09/12/23

github.com/Matteo-Celia

Preface

I would have preferred the use of some numpy structure to speed up the execution, but excluding that the code is generally well written and well documented, thanks to the useful readme.

Evolutionary algorithm

You have correctly implemented a GA, with various custom parameters to select the operations to perform. This give great customizability, but you end up with a single solution, that for you is the best performing. Instead you could have exploited this customizability through one of the techniques seen in class to promote diversity like the islands or segregation model, inserting in every island a population that perform different operentions. This could have helped the problem that the algorithm seem to have of premature convergence, higlighted also from the fact that you ended up using a realy strong mutation operator that shift all the genome.
In general you have done a very good work and the performances confirm it.

github.com/Niiikkkk

Preface

The code is well written, but there is no documentation or comments to help the understanding. Moreover I would have preferred the use of some numpy structure to speed up the execution.

Evolutionary algorithm

The GA is correctly written and do is job, but there are some small tweaks noteworthy:

- In the function one_cut_xover, even if not in use, the range of the cut_point should be set to (0, len-1) because if is len you can obtain a xover in which you just invert the two parents genomes
- I don't known why in the parent selection, before the tournament, you sort the population and select only the best, when the aim of the tournament selection is to give to everyone the possibility to compete to become parent, in this way you deterministically reduce the

population to the best half prior to make the tournamet. In each case if you want to mantain this approach I strongly recommend you to move the sorting outside the parent selection because in this case you re-sort the population for each parent.

As said above in general the GA is well implemented, maybe to increase the performance you can try some of the techniques presented in class regarding the promoting of diversity.

Dimitri Masetta 306130

LAB 10

24/12/23

lab10.ipynb

LAB10

Use reinforcement learning to devise a tic-tac-toe player.

Deadlines:

- Submission: [*Dies Natalis Solis Invicti*]
- Reviews: [*Befana*]

Notes:

- Reviews will be assigned on Monday, December 4
- You need to commit in order to be selected as a reviewer (ie. better to commit an empty work than not to commit)

```
from itertools import combinations
from collections import namedtuple, defaultdict
from random import choice
from copy import deepcopy, copy
from random import random, randint
from tqdm.auto import tqdm
import numpy as np
import statistics
import matplotlib.pyplot as plt

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 10
    elif win(pos.o):
        return -100
    else:
        return 0

def possible_actions(state:State):
        return [i for i in range(1, 9+1) if i not in state.x and i not in sta
te.o]

def random_action(state: State):
    actions = possible_actions(state)
    return actions[randint(0, len(actions) - 1)]

class QLearner:
    q = defaultdict(float)
    prev_state = None
    prev_action = None
    def __init__(self, epsilon = 0.80, learning_rate = 0.8, discount_factor =
0.9):
        self.epsilon = epsilon
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor

    def get_action(self, state:State, learn):
        actions = possible_actions(state)
        if random() < self.epsilon or not learn:
            q_values = [self.q[(state, i)] for i in actions]
            return actions[np.argmax(q_values)]
        else:
            return actions[randint(0, len(actions) - 1)]

    def get_maxQ(self, state:State):
        return max(self.q[(state, action)] for action in possible_actions(sta
te))

    @staticmethod
    def invert_state(state: State):
        return State(state.o, state.x)

    def move(self, state:State, learn=False):
        action = self.get_action(state, learn)
        state_moved = State((*state.x, action), (*state.o,))
```

```
        if learn:
            if self.prev_state is not None:
                old_q = self.q[(self.prev_state, self.prev_action)]
                max_q = self.get_maxQ(state)
                reward = state_value(state_moved)
                if reward == 0:
                    reward = -1
                self.q[(self.prev_state, self.prev_action)] = (1 - self.learn
ing_rate) * old_q + self.learning_rate * (reward + self.discount_factor * max
_q)
                if reward > 0:  #terminal win(update also last move q)
                    old_q = self.q[(state, action)]
                    self.q[(state, action)] = (1 - self.learning_rate) * old_
q + self.learning_rate * reward
        self.prev_state = state
        self.prev_action = action
        return state_moved

    def end_play(self):
        self.prev_state = None
        self.prev_action = None

    def loss(self, state):
        old_q = self.q[(self.prev_state, self.prev_action)]
        reward = state_value(state)
        self.q[(self.prev_state, self.prev_action)] =  (1 - self.learning_rat
e) * old_q + self.learning_rate * reward


class RandomPlayer:
    def move(self, state:State):
        action = random_action(state)
        next_state = State(state.x, (*state.o, action))
        return next_state

epochs = 1_000_000
history = []
history_step = 1000
step_win = 0
step_loss = 0
q_agent = QLearner()
rnd_agent = RandomPlayer()

for i in tqdm(range(epochs)):
    q_agent.end_play()
    state = State((), ())
    while True:
        state = q_agent.move(state, learn=True)
        if win(state.x):
            step_win += 1
```

```
                break
            if len(possible_actions(state)) == 0:
                #draw
                break
            state = rnd_agent.move(state)
            if win(state.o):
                q_agent.loss(state)
                step_loss += 1
                break
        if i != 0 and i%history_step == 0:
            history.append(step_win / (step_win+step_loss))
            step_win = 0
            step_loss = 0

print(len(q_agent.q))
plt.plot(history)
```

```
epochs = 100_000
history = []
q_agent = QLearner()
rnd_agent = RandomPlayer()

for i in tqdm(range(epochs)):
    story = []
    state = State((), ())
    while True:
        state = q_agent.move(state)
        story.append(state)
        if win(state.x):
            history.append(1)
            break
        if len(possible_actions(state)) == 0:
            break
        state = rnd_agent.move(state)
        story.append(state)
        if win(state.o):
            history.append(0)
            break
print(f"Win rate: {statistics.mean(history)}")
```

Dimitri Masetta 306130

PEER REVIEW LAB 10

05/01/24

github.com/lucasolaini

Preface

The code is well written, and the use of explanatory markdown sections, allow for an easy comprehension of the code.

Reinforcement Learning algorithm

You use the RL techniques of Q-Learning and the general implementation is correct but there are something I want to draw attention to:

- The update of the Qvalue of the (State, Action) should be done for each (State, Action) during the match, you have choose the approach used by the professor in the MonteCarlo RL technique in which the evaluation is given at the end. Like me you enable the agent to choose only from available move, so you can't act on them, but for example you can penalize the agent each move that don't reach an achivement.

I really liked that you have trained and evaluated the agent by playing as first as well as second, you could have implemented some techniques to update the parameter during the trainig but in general your work is well done and the results speak for themself.

github.com/gabriquaranta

Preface

There is a usefull readme, that combined with the well written code, although a lack of comments, enable an easy understanding of the code!

Reinforcement Learning algorithm

You implemented a Q-Learning RL that in general seems correct, but there is something that don't enable the training process to be effective, and the results prove it (agent vs random with 54 wins, 11 ties, 35 losses is almost as good as random vs random considering that the agent play always as first):

- a dynamic tuning of the parameters during the training is missing, but this is not the main problem, as their influence on the performance is low
- probably something in your "update_q_value" function don't correctly update the Qvalue, but I couldn't point to the source of the problem.

I really liked that you train your agent against different players and really admire your work to develop them but I think that you can improve your training process to greatly improve your agent performance.

Dimitri Masetta 306130

QUIXO

18/02/24

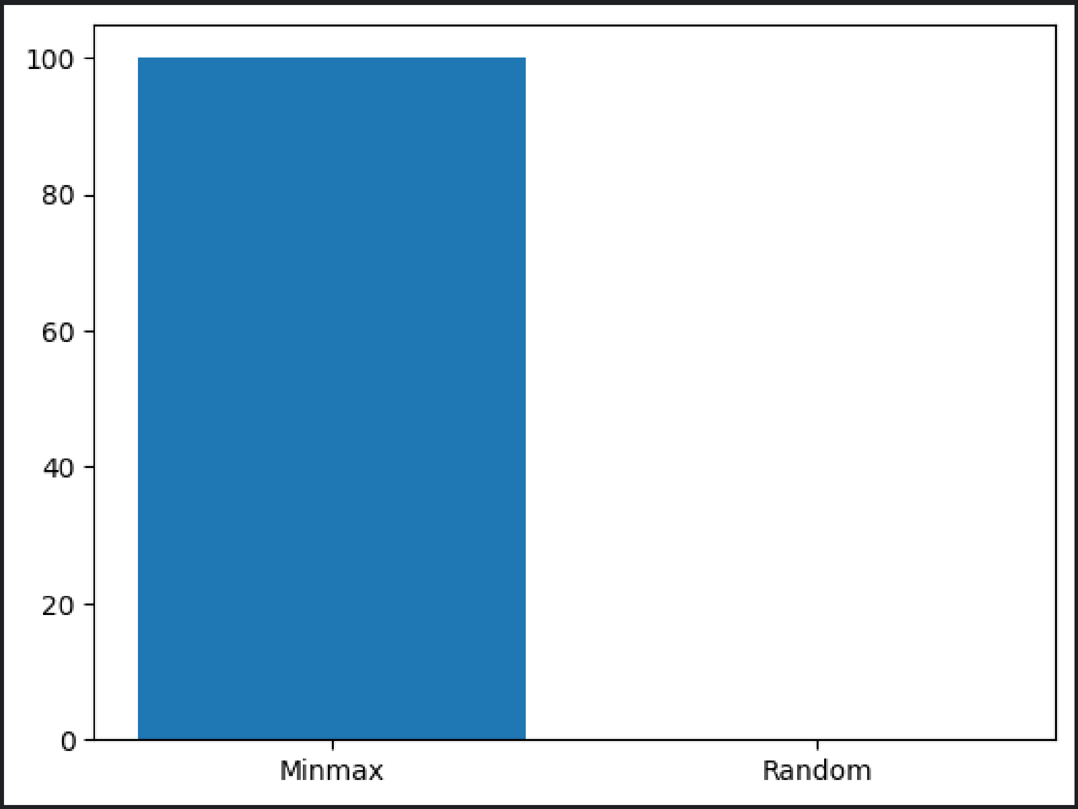
play_the_game.ipynb

```
from minmax import MinMaxPlayer
from my_random_player import MyRandomPlayer
from gp_player import GeneticProgrammingPlayer
from individual import Individual
from quixo_game import QuixoGame
import random
import matplotlib.pyplot as plt
```

Play against minmax with alpha beta pruning

```
player1 = MinMaxPlayer(search_depth=2)
player2 = MyRandomPlayer(random.randint(0, 1000000))
games = 100
w1, w2 = QuixoGame.get_results_over_x_games(player1, player2, games, change_o
rder=True)
print(f"Minmax player win ratio: {w1 / (w1 + w2)}")
fig = plt.figure()
plt.bar(("Minmax", "Random"), (w1, w2))
plt.show()
```

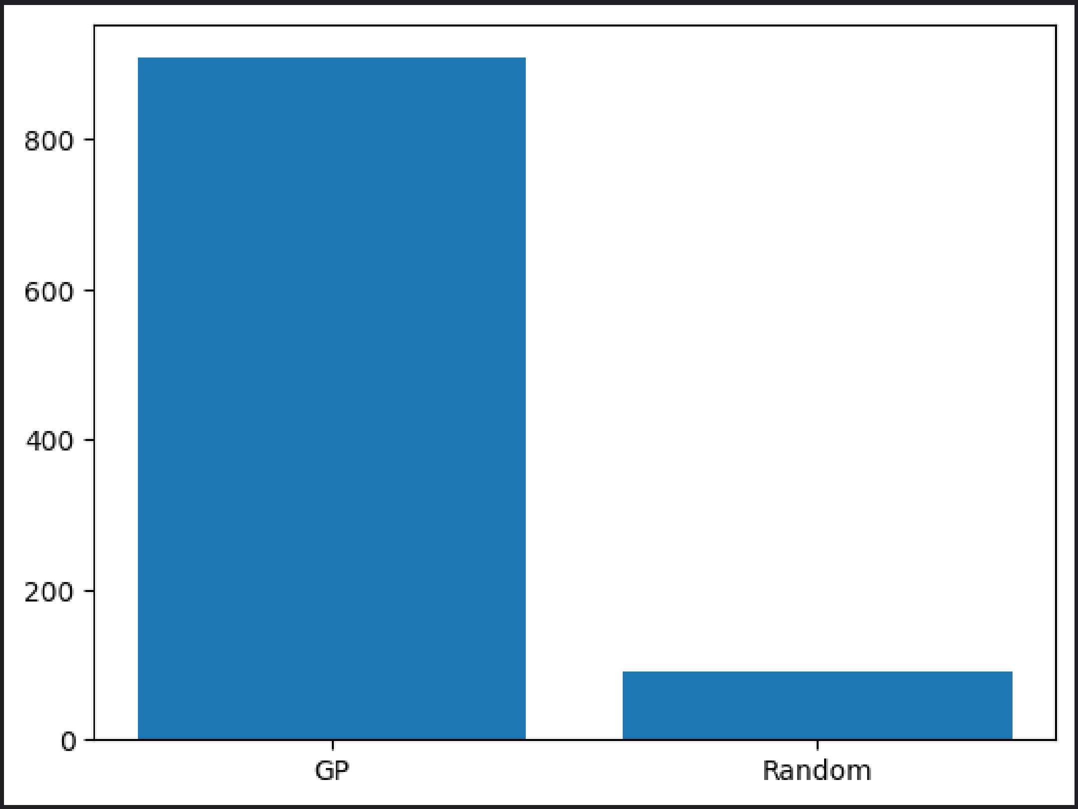
```
Minmax player win ratio: 1.0
```



Play against genetic programming player

```
player1 = GeneticProgrammingPlayer(Individual.generate_from_file("a_931.graph
"))
player2 = MyRandomPlayer(random.randint(0, 1000000))
games = 1000
w1, w2 = QuixoGame.get_results_over_x_games(player1, player2, games, change_o
rder=True)
print(f"GP player win ratio: {w1/ (w1 + w2)}")
fig = plt.figure()
plt.bar(("GP", "Random"), (w1, w2))
plt.show()
```

```
GP player win ratio: 0.908
```



crossover.py

```
from copy import deepcopy
from random import Random
from typing import Optional

import networkx as nx
from networkx.classes.reportviews import OutEdgeView

from quixo.data_bags import PopulationParameters
from quixo.individual import Individual
from quixo.graph import GraphExtended
from quixo.node import Node


class Crossover:
    @staticmethod
    def one_node_xover(p1: Individual, p2: Individual,
```

```
                              population_param:
PopulationParameters, id: Optional[int] = None) ->
Individual:
        """
        Makes the crossover between two individuals
genome, by replacing a random subgraph of the first with
a random subgraph of the second.
        Return a new individual.
        """
        p1_genome = deepcopy(p1.genome)
        p2_genome = deepcopy(p2.genome)
        p1_edge = GraphExtended.choice_edge(p1_genome,
population_param.rnd)
        p1_parent_edges =
list(p1_genome.out_edges(p1_edge[0]))
        p1_root = list(p1_genome.nodes)[0]
        p2_edge = GraphExtended.choice_edge(p2_genome,
population_param.rnd)
        child_genome: nx.DiGraph = nx.compose(p1_genome,
p2_genome)
        child_genome.remove_edges_from(p1_parent_edges)
        for edge in p1_parent_edges:
            if edge == p1_edge:
                child_genome.add_edge(p1_edge[0],
p2_edge[1])
            else:
                child_genome.add_edge(edge[0], edge[1])
        child_genome.remove_edge(p2_edge[0], p2_edge[1])
        GraphExtended.remove_unconnected(child_genome,
p1_root)
        GraphExtended.reidentify_nodes(child_genome,
p1_root)
        return Individual(id, child_genome,
parents_id=[p1.id, p2.id])
```

data_bags.py

```python
from dataclasses import dataclass, field
from random import Random
from typing import Set, List, Callable

from quixo.function import Function
from quixo.function_set import FunctionSet
from quixo.terminal_set import TerminalSet
from quixo.value_point import ValuePoint


@dataclass
class InitParameters:
    """
    Initialization parameters
    """
    use_grow: bool = field(default=True)
    use_full: bool = field(default=True)
    use_different_depth: bool = field(default=True)
    extend_probability_fun: Callable[[int], float] =
field(default=lambda depth: 1 - (1 / depth))


@dataclass
class AgentParameters:
    """
    Agent specific parameters
    """
    max_depth: int


@dataclass
class PlayerParameters:
    """
    Player specific parameters
    """
    enable_random_move: bool = field(default=True)
    loop_avoidance_limit: int = field(default=5)


@dataclass
class PopulationParameters:
    """
```

```
    Population parameters
    """
    agent_param: AgentParameters
    init_param: InitParameters
    player_param: PlayerParameters
    population_size: int
    tournament_depth: int
    selection_size: int
    rnd: Random = field(default=Random(123456))
    keep_best: bool = field(default=True)
    crossover_probability: float = field(default=.8)
    mutation_probability: float = field(default=.5)
    round_against_random: int = field(default=100)

    @property
    def random_bool(self) -> bool:
        return self.rnd.choice([True, False])
```

decorator.py

```
class classproperty(property):
    """
    Custom decorator that define a class level property
    """
    def __get__(self, owner_self, owner_cls):
        return self.fget(owner_cls)
```

function.py

```
from dataclasses import dataclass, field
from typing import Callable, Sequence
import numpy as np

from quixo.quixo_game import QuixoGame
from quixo.value_point import ValuePoint
```

```
def _fun_default(inputs: Sequence[ValuePoint], game:
QuixoGame) -> ValuePoint:
    """
    Default function, return the first element of the
sequence
    """
    assert len(inputs) == 1, f"Expected 1 inputs got
{len(inputs)}"
    return inputs[0]


@dataclass
class Function:
    """
    Define a function node
    """
    name: str
    inputs: int
    outputs: int = field(default=1)
    op: Callable[[Sequence[ValuePoint], QuixoGame],
ValuePoint] = field(default=_fun_default)
    is_action: bool = field(default=False)

    def __str__(self):
        return self.name.upper()
```

function_set.py

```
from random import Random
from typing import Set, Sequence, List, Optional

import numpy as np

from quixo.decorator import classproperty
from quixo.function import Function
from quixo.quixo_game import QuixoGame
```

```python
from quixo.value_point import ValuePoint


def _fun_and(inputs: Sequence[ValuePoint], game: QuixoGame) -> ValuePoint:
    """
    If both inputs are not NIL returns the second, otherwise return the first
    """
    assert len(inputs) == 2, f"Expected 2 inputs got {len(inputs)}"

    if inputs[0].is_nil() or inputs[1].is_nil():
        return ValuePoint.NIL
    else:
        return inputs[1]


def _fun_or(inputs: Sequence[ValuePoint], game: QuixoGame) -> ValuePoint:
    """
    Returns the first not NIL input, otherwise returns NIL
    """
    assert len(inputs) == 2, f"Expected 2 inputs got {len(inputs)}"

    if not inputs[0].is_nil():
        return inputs[0]
    else:
        return inputs[1]


def _fun_if(inputs: Sequence[ValuePoint], game: QuixoGame) -> ValuePoint:
    """
    If the first input is not NIL returns the second, otherwise the third
    """
    assert len(inputs) == 3, f"Expected 3 inputs got {len(inputs)}"
```

```
    if not inputs[0].is_nil():
        return inputs[1]
    else:
        return inputs[2]


def _fun_mine(inputs: Sequence[ValuePoint], game:
QuixoGame) -> ValuePoint:
    """
    Return NIL if the position is not mine, else returns
the input
    """
    assert len(inputs) == 1, f"Expected 1 inputs got
{len(inputs)}"

    if not inputs[0].is_nil() and
game.is_current_player_pos(inputs[0].point):
        return inputs[0]
    else:
        return ValuePoint.NIL


def _fun_yours(inputs: Sequence[ValuePoint], game:
QuixoGame) -> ValuePoint:
    """
        Return NIL if the position is not yours, else
returns the input
    """
    assert len(inputs) == 1, f"Expected 1 inputs got
{len(inputs)}"

    if not inputs[0].is_nil() and
game.is_other_player_pos(inputs[0].point):
        return inputs[0]
    else:
        return ValuePoint.NIL


def _fun_open(inputs: Sequence[ValuePoint], game:
QuixoGame) -> ValuePoint:
    """
        Return NIL if the position is not open, else
```

```
returns the input
    """
    assert len(inputs) == 1, f"Expected 1 inputs got
{len(inputs)}"

    if not inputs[0].is_nil() and
game.is_void_pos(inputs[0].point):
        return inputs[0]
    else:
        return ValuePoint.NIL


def _get_random_from_set(in_set: List[Function |
ValuePoint], rnd: Random, to_exclude: None | Function |
ValuePoint):
    """
    Get random Function/ValuePoint from set, with the
possibility to exclude one case
    """
    if to_exclude is None:
        return rnd.choice(in_set)
    else:
        return rnd.choice([f for f in in_set if f !=
to_exclude])


class FunctionSet:
    _actions_set: List[Function] = [
        Function(
            name="Left",
            inputs=1,
            is_action=True
        ),
        Function(
            name="Right",
            inputs=1,
            is_action=True
        ),
        Function(
            name="Top",
            inputs=1,
            is_action=True
```

```
        ),
        Function(
            name="Bottom",
            inputs=1,
            is_action=True
        )
    ]
    _set: List[Function] = [
        Function(
            name="And",
            inputs=2,
            op=_fun_and
        ),
        Function(
            name="Or",
            inputs=2,
            op=_fun_or
        ),
        Function(
            name="If",
            inputs=3,
            op=_fun_if
        ),
        Function(
            name="Mine",
            inputs=1,
            op=_fun_mine
        ),
        Function(
            name="Yours",
            inputs=1,
            op=_fun_yours
        ),
        Function(
            name="Open",
            inputs=1,
            op=_fun_open
        ),
        *_actions_set
    ]

    @classproperty
```

```
    def set(cls) -> List[Function]:
        """
        Returns all function set
        """
        return cls._set

    @classproperty
    def actions_set(cls) -> List[Function]:
        """
        Returns only the action set
        """
        return cls._actions_set

    @classmethod
    def get_random_function(cls, rnd: Random, to_exclude:
Optional[Function] = None, inputs: Optional[int] = None,
outputs: Optional[int] = None):
        """
        Returns a random function
        """
        set = cls._set
        if inputs is not None:
            set = [f for f in set if f.inputs == inputs]
        if outputs is not None:
            set = [f for f in set if f.outputs ==
outputs]
        return _get_random_from_set(set, rnd, to_exclude)

    @classmethod
    def get_random_functions(cls, rnd: Random, count) ->
List[Function]:
        """
        Returns some random functions
        """
        return rnd.choices(cls._set, k=count)

    @classmethod
    def get_random_action_function(cls, rnd: Random,
to_exclude: Optional[Function] = None):
        """
        Returns some random action
        """
```

```
        return _get_random_from_set(cls._actions_set,
rnd, to_exclude)
```

game.py

```
from abc import ABC, abstractmethod
from copy import deepcopy
from enum import Enum
import numpy as np


# Rules on PDF


class MoveDirection(Enum):
    '''
    Selects where you want to place the taken piece. The
rest of the pieces are shifted
    '''
    TOP = 0
    BOTTOM = 1
    LEFT = 2
    RIGHT = 3


class Player(ABC):
    def __init__(self) -> None:
        '''You can change this for your player if you
need to handle state/have memory'''
        pass

    @abstractmethod
    def make_move(self, game: 'Game') -> tuple[tuple[int,
int], MoveDirection]:
        '''
        The game accepts coordinates of the type (X, Y).
X goes from left to right, while Y goes from top to
bottom, as in 2D graphics.
```

```
        Thus, the coordinates that this method returns
shall be in the (X, Y) format.

        game: the Quixo game. You can use it to override
the current game with yours, but everything is evaluated
by the main game
        return values: this method shall return a tuple
of X,Y positions and a move among TOP, BOTTOM, LEFT and
RIGHT
        '''
        pass

    @abstractmethod
    def reset(self, also_rnd: bool = False):
        """
        Reset the player
        """
        pass


class Game(object):
    BOARD_DIM = 5

    def __init__(self) -> None:
        self._board = np.ones((self.BOARD_DIM,
self.BOARD_DIM), dtype=np.uint8) * -1
        self.current_player_idx = 1

    def get_board(self) -> np.ndarray:
        '''
        Returns the board
        '''
        return deepcopy(self._board)

    def get_current_player(self) -> int:
        '''
        Returns the current player
        '''
        return deepcopy(self.current_player_idx)

    def print(self):
        '''Prints the board. -1 are neutral pieces, 0 are
```

```
pieces of player 0, 1 pieces of player 1'''
        print(self._board)

    def check_winner(self) -> int:
        '''Check the winner. Returns the player ID of the
winner if any, otherwise returns -1'''
        # for each row
        for x in range(self._board.shape[0]):
            # if a player has completed an entire row
            if self._board[x, 0] != -1 and
all(self._board[x, :] == self._board[x, 0]):
                # return the relative id
                return self._board[x, 0]
        # for each column
        for y in range(self._board.shape[1]):
            # if a player has completed an entire column
            if self._board[0, y] != -1 and
all(self._board[:, y] == self._board[0, y]):
                # return the relative id
                return self._board[0, y]
        # if a player has completed the principal
diagonal
        if self._board[0, 0] != -1 and all(
            [self._board[x, x]
                for x in range(self._board.shape[0])] ==
self._board[0, 0]
        ):
            # return the relative id
            return self._board[0, 0]
        # if a player has completed the secondary
diagonal
        if self._board[0, -1] != -1 and all(
            [self._board[x, -(x + 1)]
             for x in range(self._board.shape[0])] ==
self._board[0, -1]
        ):
            # return the relative id
            return self._board[0, -1]
        return -1

    def play(self, player1: Player, player2: Player) ->
int:
```

```
        '''Play the game. Returns the winning player'''
        players = [player1, player2]
        winner = -1
        while winner < 0:
            self.current_player_idx += 1
            self.current_player_idx %= len(players)
            ok = False
            while not ok:
                from_pos, slide = 
players[self.current_player_idx].make_move(
                    self)
                ok = self.__move(from_pos, slide, 
self.current_player_idx)
                # 
print(f"Player:{self.current_player_idx}, Pos:{from_pos}, 
Move:{slide}, Checked:{ok}")
            winner = self.check_winner()
            # self.print()
        return winner

    def __move(self, from_pos: tuple[int, int], slide: 
MoveDirection, player_id: int) -> bool:
        '''Perform a move'''
        if player_id > 2:
            return False
        # Oh God, Numpy arrays
        prev_value = deepcopy(self._board[(from_pos[1], 
from_pos[0])])
        acceptable = self.__take((from_pos[1], 
from_pos[0]), player_id)
        if acceptable:
            acceptable = self.__slide((from_pos[1], 
from_pos[0]), slide)
            if not acceptable:
                self._board[(from_pos[1], from_pos[0])] = 
deepcopy(prev_value)
        return acceptable

    def __take(self, from_pos: tuple[int, int], 
player_id: int) -> bool:
        '''Take piece'''
        # acceptable only if in border
```

```
        acceptable: bool = (
            # check if it is in the first row
            (from_pos[0] == 0 and from_pos[1] < 5)
            # check if it is in the last row
            or (from_pos[0] == 4 and from_pos[1] < 5)
            # check if it is in the first column
            or (from_pos[1] == 0 and from_pos[0] < 5)
            # check if it is in the last column
            or (from_pos[1] == 4 and from_pos[0] < 5)
            # and check if the piece can be moved by the
current player
        ) and (self._board[from_pos] < 0 or
self._board[from_pos] == player_id)
        if acceptable:
            self._board[from_pos] = player_id
        return acceptable

    def __slide(self, from_pos: tuple[int, int], slide:
MoveDirection) -> bool:
        '''Slide the other pieces'''
        # define the corners
        SIDES = [(0, 0), (0, 4), (4, 0), (4, 4)]
        # if the piece position is not in a corner
        if from_pos not in SIDES:
            # if it is at the TOP, it can be moved down,
left or right
            acceptable_top: bool = from_pos[0] == 0 and (
                    slide == MoveDirection.BOTTOM or
slide == MoveDirection.LEFT or slide ==
MoveDirection.RIGHT
            )
            # if it is at the BOTTOM, it can be moved up,
left or right
            acceptable_bottom: bool = from_pos[0] == 4
and (
                    slide == MoveDirection.TOP or slide
== MoveDirection.LEFT or slide == MoveDirection.RIGHT
            )
            # if it is on the LEFT, it can be moved up,
down or right
            acceptable_left: bool = from_pos[1] == 0 and
(
```

```
                    slide == MoveDirection.BOTTOM or
slide == MoveDirection.TOP or slide ==
MoveDirection.RIGHT
            )
            # if it is on the RIGHT, it can be moved up,
down or left
            acceptable_right: bool = from_pos[1] == 4 and
(
                    slide == MoveDirection.BOTTOM or
slide == MoveDirection.TOP or slide == MoveDirection.LEFT
            )
        # if the piece position is in a corner
        else:
            # if it is in the upper left corner, it can
be moved to the right and down
            acceptable_top: bool = from_pos == (0, 0) and
(
                    slide == MoveDirection.BOTTOM or
slide == MoveDirection.RIGHT)
            # if it is in the lower left corner, it can
be moved to the right and up
            acceptable_left: bool = from_pos == (4, 0)
and (
                    slide == MoveDirection.TOP or slide
== MoveDirection.RIGHT)
            # if it is in the upper right corner, it can
be moved to the left and down
            acceptable_right: bool = from_pos == (0, 4)
and (
                    slide == MoveDirection.BOTTOM or
slide == MoveDirection.LEFT)
            # if it is in the lower right corner, it can
be moved to the left and up
            acceptable_bottom: bool = from_pos == (4, 4)
and (
                    slide == MoveDirection.TOP or slide
== MoveDirection.LEFT)
        # check if the move is acceptable
        acceptable: bool = acceptable_top or
acceptable_bottom or acceptable_left or acceptable_right
        # if it is
        if acceptable:
```

```
            # take the piece
            piece = self._board[from_pos]
            # if the player wants to slide it to the left
            if slide == MoveDirection.LEFT:
                # for each column starting from the
column of the piece and moving to the left
                for i in range(from_pos[1], 0, -1):
                    # copy the value contained in the
same row and the previous column
                    self._board[(from_pos[0], i)] =
self._board[(
                        from_pos[0], i - 1)]
                # move the piece to the left
                self._board[(from_pos[0], 0)] = piece
            # if the player wants to slide it to the
right
            elif slide == MoveDirection.RIGHT:
                # for each column starting from the
column of the piece and moving to the right
                for i in range(from_pos[1],
self._board.shape[1] - 1, 1):
                    # copy the value contained in the
same row and the following column
                    self._board[(from_pos[0], i)] =
self._board[(
                        from_pos[0], i + 1)]
                # move the piece to the right
                self._board[(from_pos[0],
self._board.shape[1] - 1)] = piece
            # if the player wants to slide it upward
            elif slide == MoveDirection.TOP:
                # for each row starting from the row of
the piece and going upward
                for i in range(from_pos[0], 0, -1):
                    # copy the value contained in the
same column and the previous row
                    self._board[(i, from_pos[1])] =
self._board[(
                        i - 1, from_pos[1])]
                # move the piece up
                self._board[(0, from_pos[1])] = piece
            # if the player wants to slide it downward
```

```
            elif slide == MoveDirection.BOTTOM:
                # for each row starting from the row of
the piece and going downward
                for i in range(from_pos[0],
self._board.shape[0] - 1, 1):
                    # copy the value contained in the
same column and the following row
                    self._board[(i, from_pos[1])] =
self._board[(
                        i + 1, from_pos[1])]
                # move the piece down
                self._board[(self._board.shape[0] - 1,
from_pos[1])] = piece
        return acceptable
```

gp_player.py

```
from random import Random
from typing import List, Optional

import networkx as nx

from quixo.game import Player, MoveDirection, Game
from quixo.individual import Individual
from quixo.my_move import MyMove
from quixo.node import Node
from quixo.quixo_game import QuixoGame
from quixo.value_point import ValuePoint


class GeneticProgrammingPlayer(Player):

    def __init__(self, brain: Individual,
enable_random_move: Optional[bool] = True,
loop_avoidance_limit: int = 5):
        super().__init__()
        self._brain = brain
        self._enable_random_move = enable_random_move
```

```
        self._loop_avoidance_limit = loop_avoidance_limit
        self._rnd = None
        self._set_rnd()
        self._move_count = 0
        self._rnd_move_count = 0
        self._last_move = None
        self._last_move_occurrences = 0

    @property
    def brain(self):
        """
        Return the individual
        """
        return self._brain

    @property
    def move_count(self) -> int:
        """
        Return the move played
        """
        return self._move_count

    @property
    def rnd_move_count(self) -> int:
        """
        Return the random move played
        """
        return self._rnd_move_count

    @property
    def rnd_move_percentage(self) -> float:
        """
        Return the random move percentage
        """
        return self._rnd_move_count / self._move_count *
100

    def _set_rnd(self):
        """
        Set random seed
        """
        self._rnd = Random(self._brain.random_seed)
```

```
    def reset_counters(self):
        """
        Reset counters
        """
        self._move_count = 0
        self._rnd_move_count = 0
        self._last_move = None
        self._last_move_occurrences = 0

    def reset(self, also_rnd: bool = False):
        """
        Reset player
        """
        self.reset_counters()
        if also_rnd:
            self._set_rnd()

    def _make_move_recursive(self, node: Node, game:
QuixoGame) -> tuple[ValuePoint, Optional[MyMove]]:
        """
        Make move recursively, analyze the genome graph,
depth first, in search of the first doable move.
        Return the tuple[value returned by the node,
final move]
        """
        if node.is_terminal:
            return node.value_point, None

        descendant: List[Node] = [v for u,v in
self._brain.genome.out_edges(node)]
        descendants_results = []
        for d in descendant:
            res, move = self._make_move_recursive(d,
game)
            if move is not None:
                return res, move
            descendants_results.append(res)
        result = node.function.op(descendants_results,
game)
        if node.function.is_action:
            # print(f"Node Res: {result}, Move
```

```
{node.function.name}")
            if not result.is_nil():
                move = MyMove(result.point,
MoveDirection[node.function.name.upper()])
                if game.is_move_doable(move):
                    if self._last_move == move and
self._last_move_occurrences ==
self._loop_avoidance_limit:
                        # print("Loop limit reached")
                        return result, None
                    else:
                        if self._last_move == move:
                            self._last_move_occurrences
+= 1
                        else:
                            self._last_move = move
                            self._last_move_occurrences =
1
                        # print(f"AGENT => Pos:
{move.position}, Dir {move.direction}")
                        return result, move
        return result, None

    def make_move(self, game: QuixoGame) ->
tuple[tuple[int, int], MoveDirection]:
        """
        Make the move, if no move found analyzing the
graph, make a random move.
        """
        if len(self._brain.genome) == 0:
            move = None
        else:
            _, move =
self._make_move_recursive(list(self._brain.genome.nodes)[
0], game)
        if move is None:
            # print("Damn, no doable move found ")
            assert self._enable_random_move, "Random move
not enabled, no move available"
            while True:
                move =
self._rnd.choice(game.available_moves_list)
```

```
                if game.is_move_doable(move):
                    break
            self._last_move = move
            self._last_move_occurrences = 1
            self._rnd_move_count += 1
            # print(f"AGENT RND => Pos: {move.position}, 
Dir: {move.direction}")
        self._move_count += 1
        return move.to_tuple
```

graph.py

```
from random import Random

import networkx as nx

from quixo.node import Node


class GraphExtended:
    @staticmethod
    def remove_unconnected(G: nx.DiGraph, root: Node):
        """
        Remove unconnected nodes and subgraph from the 
root node
        """
        H = nx.Graph()
        H.add_edges_from(G.edges)
        H.add_nodes_from(G.nodes)
        connected = nx.node_connected_component(H, root)
        G.remove_nodes_from(H.nodes - connected)

    @staticmethod
    def reidentify_nodes(G: nx.DiGraph, node: Node):
        """
        Reassign id starting from 0 to the graph breadth 
first(left child id < right child id)
        """
```

```
        node.id = 0
        _ = GraphExtended._reidentify_nodes_recursive(G,
node, 1)

    @staticmethod
    def _reidentify_nodes_recursive(G: nx.DiGraph, node:
Node, count: int) -> int:
        """
        Reassign id starting recursevely
        """
        fan_out = [v for (u, v) in G.out_edges(node)]
        for n in fan_out:
            n.id = count
            count += 1
        for n in fan_out:
            count =
GraphExtended._reidentify_nodes_recursive(G, n, count)
        return count

    @staticmethod
    def choice_edge(p_genome: nx.DiGraph, rnd: Random) ->
(Node, Node):
        """
        Choose a random edge in the graph
        """
        p_node = rnd.choice(list(p_genome.nodes)[1:])
        p_parent_edge = list(p_genome.in_edges(p_node))
        assert len(
            p_parent_edge) == 1, f"Something strange
appened, {len(p_parent_edge)} edges from parent node to
node"
        return p_parent_edge[0]
```

individual.py

```
from functools import cached_property
from random import Random
from typing import Optional, List, Sequence
```

```
import networkx as nx
import matplotlib.pyplot as plt
import pickle

from quixo.decorator import classproperty
from quixo.node import Node


# from networkx.drawing.nx_agraph import write_dot,
graphviz_layout

class Individual:
    def __init__(self, id: Optional[int] = None, genome:
Optional[nx.DiGraph] = None, parents_id:
Optional[Sequence[int]] = None):
        self._genome = genome
        self._id = id
        self._parenst_id = parents_id
        self._fitness = None

    def __str__(self):
        return f"Individual {self._id},
{self._genome.adj}"

    def __repr__(self):
        return f"I_{self._id}"

    def __hash__(self):
        return hash(self.id + hash(self.genome_adj_str))

    @property
    def genome(self) -> nx.DiGraph:
        """
        Return the graph genome
        """
        assert self._genome is not None, "Genome not
defined"
        return self._genome

    @property
    def id(self) -> int:
        """
```

```
        Return the id
        """
        assert self._id is not None, "Id not defined"
        return self._id

    @property
    def fitness(self) -> float:
        """
        Return the fitness
        """
        return self._fitness

    @fitness.setter
    def fitness(self, value: float):
        """
        Set the fitness
        """
        self._fitness = value

    @cached_property
    def genome_adj_str(self):
        """
        Return the cached string representation of the
genome
        """
        return str(self._genome.adj)

    @cached_property
    def traversal_list(self) -> List[Node]:
        """
        Return the cached traversal list of node in the
graph in depth first
        """
        assert self._genome is not None, "Genome not
defined"
        return list(nx.dfs_preorder_nodes(self._genome,
source=list(self._genome.nodes)[0]))

    @cached_property
    def random_seed(self):
        """
        Return a cached random seed
```

```
        """
        return str(self.id) + self.genome_adj_str

    @staticmethod
    def generate_random_individual(id: int) ->
'Individual':
        """
        Generate an individual that do only random move
        """
        return Individual(id, nx.DiGraph())

    @staticmethod
    def generate_from_file(filename: str) ->
'Individual':
        """
        Generate an individual from file
        """
        genome = pickle.load(open(filename, 'rb'))
        return Individual(Random(filename).randint(-
100000000, 0), genome)

    def save_to_file(self, filename: str):
        """
        Save an individual to file
        """
        pickle.dump(self._genome, open(filename, 'wb'))

    def print_graph(self):
        """
        Print the graph representation
        """
        # pos = graphviz_layout(self._genome, prog='dot')
        nx.draw(self._genome,
pos=nx.planar_layout(self._genome), with_labels=True,
arrows=True,
                node_color=['blue' if n.is_terminal else
'red' for n in self._genome], alpha=0.5)
        plt.show()
```

initialize.py

```
import math
from typing import Sequence, List

import networkx as nx

from quixo.data_bags import InitParameters,
AgentParameters, PopulationParameters
from quixo.function import Function
from quixo.function_set import FunctionSet
from quixo.individual import Individual
from quixo.node import Node
from quixo.terminal_set import TerminalSet
from quixo.value_point import ValuePoint


class Initializer:
    def __init__(self, population_param:
PopulationParameters):
        self._population_param = population_param

    def initialize_population(self) ->
Sequence[Individual]:
        """
        Initialize a random population based on the
parameters
        """
        assert self._population_param.init_param.use_grow
or self._population_param.init_param.use_full, "No
initialization method defined(use grow, full or both)"

        usable_depth =
self._population_param.agent_param.max_depth - 1
        if
self._population_param.init_param.use_different_depth:
            individuals_max_per_depth =
math.ceil(self._population_param.population_size /
usable_depth)
            surplus = (usable_depth *
individuals_max_per_depth) -
self._population_param.population_size - 1
```

```
            individuals_per_depth =
[individuals_max_per_depth if i > surplus else
individuals_max_per_depth - 1 for i
                                     in
range(usable_depth)]
            depths = [2 + i for i in range(usable_depth)]
            individuals_and_depth =
zip(individuals_per_depth, depths)
        else:
            individuals_and_depth = [
                (self._population_param.population_size,
self._population_param.agent_param.max_depth)]

        use_both_method =
self._population_param.init_param.use_grow and
self._population_param.init_param.use_full
        is_grow =
self._population_param.init_param.use_grow
        genomes = []
        individual_count = 0
        for individuals, depth in individuals_and_depth:
            for _ in range(individuals):
                if use_both_method:
                    is_grow = not is_grow
                if is_grow:

genomes.append(Individual(individual_count,
self._create_tree(depth, not is_grow)))
                else:

genomes.append(Individual(individual_count,
self._create_tree(depth, not is_grow)))
                individual_count += 1

        assert len(genomes) ==
self._population_param.population_size, f"Something went
wrong, generated {len(genomes)} individuals instead of
{self._population_param.population_size}"
        return genomes

    def _create_tree(self, depth: int, is_full: bool) ->
nx.DiGraph:
```

```
        """
        Create a random individual genome graph tree
        """
        def _add_node_to_graph(graph: nx.DiGraph,
previous_layer_nodes: List[Node], in_layer_nodes:
List[Node], ):
            """
            Add node and edge to the graph
            """
            in_layer_count = 0
            for p_n in previous_layer_nodes:
                for _ in range(p_n.function.inputs):
                    node_to_add =
in_layer_nodes[in_layer_count]
                    graph.add_edge(p_n, node_to_add)
                    in_layer_count += 1

        extend_prob =
self._population_param.init_param.extend_probability_fun(
depth)
        node_count = 0
        graph = nx.DiGraph()
        start_node = Node(node_count,
FunctionSet.get_random_action_function(self._population_p
aram.rnd))
        graph.add_node(start_node)
        previous_layer_nodes = [start_node]
        node_count += 1
        for d in range(1, depth):
            previous_layer_inputs_count =
sum(node.function.inputs for node in
previous_layer_nodes)
            if is_full:
                previous_layer_inputs = [True for _ in
range(previous_layer_inputs_count)]
            else:
                previous_layer_inputs =
self._population_param.rnd.choices([True, False],
[extend_prob, 1 - extend_prob],

k=previous_layer_inputs_count)
```

```
            in_layer_nodes = [Node(node_count + i, n) for
i, n in enumerate(

[FunctionSet.get_random_function(self._population_param.r
nd) if b else
TerminalSet.get_random_terminal(self._population_param.rn
d) for b in previous_layer_inputs]
            )]
            node_count += len(in_layer_nodes)

            _add_node_to_graph(graph,
previous_layer_nodes, in_layer_nodes)
            previous_layer_nodes = [n for n in
in_layer_nodes if n.is_function]

        last_leafs = [Node(node_count + i, l) for i, l in
enumerate(

TerminalSet.get_random_terminals(self._population_param.r
nd, sum(node.function.inputs for node in
previous_layer_nodes))
        )]
        _add_node_to_graph(graph, previous_layer_nodes,
last_leafs)
        return graph
```

minmax.py

```
from copy import deepcopy
from typing import Optional

import numpy as np

from quixo.game import Player, MoveDirection
from quixo.my_move import MyMove
from quixo.quixo_game import QuixoGame
```

```
class MinMaxPlayer(Player):
    MAX_VALUE = 1000

    def __init__(self, search_depth: int = 2):
        super().__init__()
        self._search_depth = search_depth

    def reset(self, also_rnd: bool = False):
        pass

    def make_move(self, game: QuixoGame) ->
tuple[tuple[int, int], MoveDirection]:
        """
        Make the move
        """
        _, move = self._recursive_analysis(game,
self._search_depth, True, - MinMaxPlayer.MAX_VALUE,

MinMaxPlayer.MAX_VALUE)
        return move.to_tuple

    def _game_evaluation(self, game: QuixoGame, winner:
int) -> int:
        """
        Evaluate the current game situation
        """
        if winner == -1:
            return self._player_evaluation(game,
game.player_id) - self._player_evaluation(game,

(game.player_id + 1) % 2)
        elif winner == game.player_id:
            return MinMaxPlayer.MAX_VALUE
        else:
            return - MinMaxPlayer.MAX_VALUE

    def _player_evaluation(self, game: QuixoGame,
player_id: int) -> int:
        """
        Evaluate the player situation
        """
        columns_occurrencies = (game.board ==
```

```
player_id).sum(axis=0).tolist()
        rows_occurrencies = (game.board ==
player_id).sum(axis=1).tolist()
        diagonals_occurrencies = [(game.board.diagonal()
== player_id).sum(),

(np.fliplr(game.board).diagonal() == player_id).sum()]
        all_combinations = [*columns_occurrencies,
*rows_occurrencies, *diagonals_occurrencies]
        result = 0
        for c in all_combinations:
            if c == 1:
                result += c
            if c == 2:
                result += c * 2
            if c == 3:
                result += c * 4
            if c == 4:
                result += c * 8
        return result

    def _recursive_analysis(self, game: QuixoGame,
current_depth: int, is_maximizing: bool, alpha: int,
beta: int) -> tuple[int, Optional[MyMove]]:
        """
        Do the minmax algorithm by recursevly analyze and
evaluating possible solution
        """
        available_moves =
game.current_player_available_move_list
        top_evaluation = - MinMaxPlayer.MAX_VALUE if
is_maximizing else MinMaxPlayer.MAX_VALUE
        top_move = None

        winner = game.check_winner()
        if winner != -1 or current_depth == 0:
            return self._game_evaluation(game, winner),
None

        for move in available_moves:
            pos, dir = move.to_tuple
            test_game = deepcopy(game)
```

```
            test_game.move(pos, dir, game.player_id)

            evaluation, _ =
self._recursive_analysis(test_game, current_depth - 1,
not is_maximizing, alpha, beta)

            if is_maximizing:
                if evaluation > top_evaluation:
                    top_evaluation = evaluation
                    top_move = move
                alpha = max(alpha, top_evaluation)
            else:
                if evaluation < top_evaluation:
                    top_evaluation = evaluation
                    top_move = move
                beta = min(beta, top_evaluation)

            if beta <= alpha:
                break

        return top_evaluation, top_move
```

mutation.py

```
from copy import deepcopy
from typing import Optional

from quixo.data_bags import PopulationParameters
from quixo.function_set import FunctionSet
from quixo.individual import Individual
from quixo.graph import GraphExtended
from quixo.node import Node
from quixo.terminal_set import TerminalSet


class Mutation:
    @staticmethod
    def one_node_mutation(p: Individual,
```

```
                          population_param:
PopulationParameters, id: Optional[int] = None) ->
Individual:
        """
        Mutate a random node of the individual genome,
changing it with a similar one
        """
        p_genome = deepcopy(p.genome)
        p_edge = GraphExtended.choice_edge(p_genome,
population_param.rnd)
        p_root = list(p_genome.nodes)[0]
        node = p_edge[1]
        # print(node)
        p_parent_edges =
list(p_genome.out_edges(p_edge[0]))
        p_node_edges = list(p_genome.out_edges(node))
        if node.is_terminal:
            new_node = Node(node.id,
TerminalSet.get_random_terminal(population_param.rnd,
to_exclude=node))
        else:
            new_node = Node(node.id,

FunctionSet.get_random_function(population_param.rnd,
to_exclude=node,

inputs=node.function.inputs,
outputs=node.function.outputs)
                            if node != p_root else

FunctionSet.get_random_action_function(population_param.r
nd, to_exclude=node)
                            )
        # print(new_node)
        p_genome.remove_node(node)
        p_genome.add_node(new_node)
        for edge in p_parent_edges:
            if edge[1] == node:
                p_genome.add_edge(edge[0], new_node)
            else:
                p_genome.add_edge(edge[0], edge[1])
        for edge in p_node_edges:
```

```
            p_genome.add_edge(new_node, edge[1])
        return Individual(id, p_genome,
parents_id=[p.id])
```

my_move.py

```
from dataclasses import dataclass
from functools import cached_property
from typing import Sequence, List, Tuple

from quixo.game import MoveDirection


@dataclass
class MyMove:
    position: Tuple[int, int]
    direction: MoveDirection

    def __hash__(self):
        return hash((self.position, self.direction))

    def __str__(self):
        return f"Position: {self.position}, Direction:
{self.direction}"

    @property
    def to_tuple(self) -> tuple[tuple[int, int],
MoveDirection]:
        """
        Return the tuple composition of the values
        """
        return (self.position[0], self.position[1]),
self.direction

    @cached_property
    def position_reversed(self) -> Tuple[int, int]:
        """
        Return the position with axis reversed
```

```
        """
        return self.position[1], self.position[0]
```

my_random_player.py

```
from quixo.game import Player, MoveDirection
from random import Random

from quixo.my_move import MyMove
from quixo.quixo_game import QuixoGame


class MyRandomPlayer(Player):

    def __init__(self, seed: int) -> None:
        super().__init__()
        self._seed = seed
        self._rnd = None
        self._set_rnd()

    def _set_rnd(self):
        """
        Set the random generator
        """
        self._rnd = Random(self._seed)

    def reset(self, also_rnd: bool = False):
        """
        Reset the player
        """
        if also_rnd:
            self._set_rnd()

    def make_move(self, game: QuixoGame) ->
tuple[tuple[int, int], MoveDirection]:
        """
        Make a random move
        """
```

```
        while True:
            move =
self._rnd.choice(game.available_moves_list)
            if game.is_move_doable(move):
                break
        # print(f"RANDOM => Pos: {move.position}, Dir:
{move.direction}")
        return move.to_tuple
```

node.py

```
from dataclasses import dataclass

from quixo.function import Function
from quixo.value_point import ValuePoint


class Node:
    def __init__(self, id: int, content: Function |
ValuePoint):
        self._id = id
        self._is_terminal = True if type(content) is
ValuePoint else False
        self._content = content

    def __str__(self):
        return f"Id:{self._id}, {self._content}"

    def __repr__(self):
        return f"<{self._id}, {self._content}>"

    @property
    def id(self) -> int:
        """
        Return id
        """
        return self._id
```

```
    @id.setter
    def id(self, id: int):
        """
        Set id
        """
        self._id = id


    @property
    def is_terminal(self) -> bool:
        """
        Return true if node is a value point(leaf)
        """
        return self._is_terminal

    @property
    def is_function(self) -> bool:
        """
        Return true if node is a function node
        """
        return not self._is_terminal

    @property
    def function(self) -> Function:
        """
        Return the function
        """
        assert type(self._content) is Function,
f"Requested Function, but found {type(self._content)}"
        return self._content

    @property
    def value_point(self) -> ValuePoint:
        """
        Return the value point
        """
        assert type(self._content) is ValuePoint,
f"Requested ValuePoint, but found {type(self._content)}"
        return self._content
```

population.py

```
import concurrent.futures
import math
from concurrent.futures import ThreadPoolExecutor
from dataclasses import dataclass, field
from typing import List, Optional, Callable
from itertools import chain
from tqdm import tqdm
from multiprocessing.pool import ThreadPool

from quixo.crossover import Crossover
from quixo.data_bags import PopulationParameters
from quixo.function_set import FunctionSet
from quixo.game import Player
from quixo.gp_player import GeneticProgrammingPlayer
from quixo.individual import Individual
from quixo.initializer import Initializer
from quixo.mutation import Mutation
from quixo.my_random_player import MyRandomPlayer
from quixo.quixo_game import QuixoGame
from quixo.terminal_set import TerminalSet


class Population:
    def __init__(self, population_param:
PopulationParameters, initial_population:
Optional[List[Individual]] = None):
        self._population_param = population_param
        self._individuals = [] if initial_population is
None else initial_population
        self._selected_parents = []
        self._bests = []
        self._generation = 0

    def __str__(self):
        return f"""Generation: {self._generation}
```

```
Individuals: {len(self._individuals)}"""

    @property
    def individuals(self) -> List[Individual]:
        """
        Return individuals
        """
        return self._individuals

    @property
    def bests(self) -> List[Individual]:
        """
        Return bests across generations
        """
        return self._bests

    @property
    def selected_parents(self) -> List[Individual]:
        """
        Return selected parent of current generation
        """
        return self._selected_parents

    def initialize(self):
        """
        Initialize population
        """
        initializer = Initializer(self._population_param)
        self._individuals =
initializer.initialize_population()

    def _fitness_evaluation_no_coevolution(self,
individuals: List[individuals]) -> List[Individual]:
        """
        Evaluate fitness against random players
        """
        fitnesses = []
        selected = []
        randoms =
[Individual.generate_random_individual(self._population_p
aram.rnd.randint(-100000000, 0)) for _ in
```

```
range(self._population_param.round_against_random)]

        for individual in individuals:
            flag = True
            count = 0
            for r in randoms:
                if flag:
                    w = self._match(individual, r)
                    if w == 0:
                        count += 1
                else:
                    w = self._match(r, individual)
                    if w == 1:
                        count += 1
                flag = not flag
            fitness = count /
self._population_param.round_against_random
            fitnesses.append(fitness)
            individual.fitness = fitness
        print(f"Ind avg: {sum([i.fitness for i in
individuals]) / self._population_param.population_size}")
        selected =
self._fitness_selection_roulette(individuals, fitnesses)
        print(f"Sel avg: {sum([i.fitness for i in
selected]) / self._population_param.selection_size}")
        return selected

    def _fitness_evaluation_no_coevolution_mixed(self,
individuals: List[individuals]) -> List[Individual]:
        """
        Evaluate fitness against random and pre-trained
players
        """
        fitnesses = []
        selected = []
        pre_trained_count = 10
        randoms =
[Individual.generate_random_individual(self._population_p
aram.rnd.randint(-100000000, 0)) for _ in

range(self._population_param.round_against_random)]
        pre_trained =
```

```
[Individual.generate_from_file(f"dev_stuff/bests/run_1/{f
}.graph") for f in range(pre_trained_count)]
        pre_trained = list(chain.from_iterable((x, x) for
x in pre_trained))
        adversaries = [*randoms, *pre_trained]
        for individual in individuals:
            flag = True
            count = 0
            for a in adversaries:
                if flag:
                    w = self._match(individual, a)
                    if w == 0:
                        count += 1
                else:
                    w = self._match(a, individual)
                    if w == 1:
                        count += 1
                flag = not flag
            fitness = count / len(adversaries)
            fitnesses.append(fitness)
            individual.fitness = fitness
        print(f"Ind avg: {sum([i.fitness for i in
individuals]) / self._population_param.population_size}")
        selected =
self._fitness_selection_roulette(individuals, fitnesses)
        print(f"Sel avg: {sum([i.fitness for i in
selected]) / self._population_param.selection_size}")
        return selected

    def _fitness_selection_roulette(self, individuals:
List[individuals], fitnesses: List[float]) ->
List[Individual]:
        """
        Selection based on the roulette wheel approach
        """
        selected =
self._population_param.rnd.choices(individuals,
weights=fitnesses,

k=self._population_param.selection_size)
        return selected
```

```
    def _fitness_selection_tournament(self, individuals:
List[individuals], fitnesses: List[float]) ->
List[Individual]:
        """
        Selection based on the tournament approach
        """
        selected = []
        for i in
range(self._population_param.selection_size):
            tournament_individuals =
self._population_param.rnd.choices(individuals,

k=2 ** self._population_param.tournament_depth)

selected.append(self._tournament(tournament_individuals,
self._tournament_fitness))
        return selected

    def _fitnessless_selection_coevolution(self,
individuals: List[individuals]) -> List[Individual]:
        """
        Coevolution selection, tournament based, no
fitness
        """
        selected = []
        results = {}
        tournament_size = 2 **
self._population_param.tournament_depth
        for i in
range(self._population_param.selection_size):
            if tournament_size <
self._population_param.population_size:
                tournament_individuals =
self._population_param.rnd.sample(individuals,
k=tournament_size)
            else:
                tournament_individuals =
self._population_param.rnd.choices(individuals,
k=tournament_size)
            # print(f"Tournament {i}:
{tournament_individuals}")
            winner =
```

```
self._tournament(tournament_individuals,
self._tournament_match)
            if winner in results:
                results[winner] += 1
            else:
                results[winner] = 1
            selected.append(winner)
            # print(f"Tournament {i} WINNER:
{repr(winner)}")
        best_points = max(results.values())
        best = self._population_param.rnd.choice([k for
k, v in results.items() if v == best_points])
        # print(f"Best: {repr(best)}")
        self._bests.append(best)
        # print(f"Selected: {selected}")
        return selected

    def _interactive_selection_against_random(self,
individuals: List[individuals]):
        """
        Interactive selection against random player
        """
        selected = []
        for ind in individuals:
            tournament_individuals = [
                ind,

Individual.generate_random_individual(self._population_pa
ram.rnd.randint(-100000000, 0)),

Individual.generate_random_individual(self._population_pa
ram.rnd.randint(-100000000, 0)),
                ind
            ]
            winner =
self._tournament(tournament_individuals,
self._tournament_match, override_depth=2)
            # print(f"W{repr(winner)}")
            if winner == ind:
                selected.append(winner)
        print(f"Selected(interactive): {len(selected)}")
        return selected
```

```
    def _tournament(self, individuals: List[Individual],
tournament_fun: Callable[[int, Individual, Individual],
int],
                        override_depth: Optional[int] = None,
) -> Individual:
        """
        TOurnament
        """
        # print(individuals)
        this_level_individuals = individuals
        next_level_individuals = []
        depth = self._population_param.tournament_depth
if override_depth is None else override_depth
        for d in range(depth):
            for i in range(0,
len(this_level_individuals), 2):
                i1 = this_level_individuals[i]
                i2 = this_level_individuals[i + 1]
                if i1 == i2:
                    next_level_individuals.append(i1)
                else:

next_level_individuals.append(this_level_individuals[i +
self._match(i1, i2)])
            this_level_individuals =
next_level_individuals
            next_level_individuals = []
        assert len(this_level_individuals) == 1,
f"Expected only one individual left, got
{len(this_level_individuals)}"
        return this_level_individuals[0]

    def _tournament_match(self, id: int, i1: Individual,
i2: Individual) -> int:
        """
        Tournament based on match
        """
        return id + self._match(i1, i2)

    def _tournament_fitness(self, id: int, i1:
Individual, i2: Individual) -> int:
```

```
        """
        Tournament based on fitness
        """
        if i1.fitness > i2.fitness:
            return id + 0
        else:
            return id + 1

    def _match(self, i1: Individual, i2: Individual) ->
int:
        """
        Match between individuals
        """
        p1 = GeneticProgrammingPlayer(i1,
enable_random_move=self._population_param.player_param.en
able_random_move,

loop_avoidance_limit=self._population_param.player_param.
loop_avoidance_limit)
        p2 = GeneticProgrammingPlayer(i2,
enable_random_move=self._population_param.player_param.en
able_random_move,

loop_avoidance_limit=self._population_param.player_param.
loop_avoidance_limit)
        return Population._match_players(p1, p2)

    @staticmethod
    def _match_players(p1: Player, p2: Player) -> int:
        """
        Match between players
        """
        game = QuixoGame()
        w = game.play(p1, p2)
        # print(f"p1{repr(p1.brain)},
p2{repr(p2.brain)}")
        # print(f"W: {w}")
        return w

    def recombination(self, selected_parents:
List[individuals]) -> List[Individual]:
        """
```

```
        Recombination process, crossover and
reproduction, then mutation
        """
        count = 0
        id_offset = (self._generation + 1) * 1000
        childs = []
        mutations =
self._population_param.rnd.choices([True, False],

weights=[self._population_param.mutation_probability,

self._population_param.mutation_probability],

k=self._population_param.population_size)

        crossover_count =
int(self._population_param.crossover_probability *
self._population_param.population_size)
        crossover_parents =
self._population_param.rnd.choices(selected_parents,
k=crossover_count * 2)
        for i in range(crossover_count):
            child =
Crossover.one_node_xover(crossover_parents[2 * i],
crossover_parents[2 * i + 1],

self._population_param, count + id_offset)
            if mutations[count]:
                child = Mutation.one_node_mutation(child,
self._population_param, child.id)
            childs.append(child)
            count += 1

        reproduction_count =
self._population_param.population_size - crossover_count
        reproduction_parents =
self._population_param.rnd.choices(selected_parents,
k=reproduction_count)
        for i in range(reproduction_count):
            child = Individual(count + id_offset,
reproduction_parents[i].genome,
```

```
parents_id=[reproduction_parents[i].id])
            if mutations[count]:
                child = Mutation.one_node_mutation(child,
self._population_param, child.id)
            childs.append(child)
            count += 1
        return childs

    def _set_best(self, individuals: List[Individual]):
        best = max(individuals, key=lambda ind:
ind.fitness)
        self._bests.append(best)
        print(f"Best fitness: {best.fitness}")

    def proceed_generation(self):
        """
        Make a generation
        """
        print(self)
        # self._selected_parents =
self._fitnessless_selection_coevolution()
        # self._selected_parents =
self._interactive_selection_against_random(self._individu
als)
        # self._selected_parents =
self._fitnessless_selection_coevolution(self._selected_pa
rents)
        self._selected_parents =
self._fitness_evaluation_no_coevolution(self.individuals)
        # self._selected_parents =
self._fitness_evaluation_no_coevolution_mixed(self.indivi
duals)
        self._set_best(self.individuals)
        childs =
self.recombination(self._selected_parents)
        self._individuals = childs
        self._generation += 1
        # for i in self._individuals:
        #     i.print_graph()

    def proceed_x_generation(self, x):
        """
```

```
        Proceed for x generations
        """
        for _ in tqdm(range(x)):
            self.proceed_generation()
```

quixo_game.py

```
from copy import deepcopy
from functools import cached_property
from typing import Sequence, Set, List

import numpy as np

from quixo.game import MoveDirection, Player, Game
from quixo.my_move import MyMove


class QuixoGame(Game):
    def __init__(self) -> None:
        super().__init__()
        self._move_count = 0

    @cached_property
    def available_moves_list(self) -> List[MyMove]:
        """
        Returned the cached list of all the available
moves
        """
        moves = list()
        for i in range(self.BOARD_DIM):
            for j in range(self.BOARD_DIM):
                if i == 0:
                    moves.append(MyMove((i, j),
MoveDirection.RIGHT))
                    if 0 < j < self.BOARD_DIM - 1:
                        moves.append(MyMove((i, j),
MoveDirection.BOTTOM))
                        moves.append(MyMove((i, j),
```

```
MoveDirection.TOP))
                if i == 4:
                    moves.append(MyMove((i, j),
MoveDirection.LEFT))
                    if 0 < j < self.BOARD_DIM - 1:
                        moves.append(MyMove((i, j),
MoveDirection.BOTTOM))
                        moves.append(MyMove((i, j),
MoveDirection.TOP))
                if j == 0:
                    moves.append(MyMove((i, j),
MoveDirection.BOTTOM))
                    if 0 < i < self.BOARD_DIM - 1:
                        moves.append(MyMove((i, j),
MoveDirection.RIGHT))
                        moves.append(MyMove((i, j),
MoveDirection.LEFT))
                if j == 4:
                    moves.append(MyMove((i, j),
MoveDirection.TOP))
                    if 0 < i < self.BOARD_DIM - 1:
                        moves.append(MyMove((i, j),
MoveDirection.RIGHT))
                        moves.append(MyMove((i, j),
MoveDirection.LEFT))
        return moves

    @property
    def current_player_available_move_list(self):
        """
        Return the player effectively available moves
        """
        possible_values = (self.player_id, -1)
        return [m for m in self.available_moves_list if
self.board[m.position] in possible_values]

    @cached_property
    def available_moves_set(self) -> Set[MyMove]:
        """
        Return the cached set of the available moves
        """
        return set(self.available_moves_list)
```

```
    @property
    def move_count(self) -> int:
        """
        Return the move count
        """
        return self._move_count

    @property
    def player_id(self) -> int:
        """
        Return the current player id
        """
        return self.current_player_idx

    @property
    def board(self) -> np.ndarray:
        """
        Return the board
        """
        return self._board

    @property
    def board_clone(self) -> np.ndarray:
        """
        Return the board clone
        """
        return deepcopy(self._board)

    @staticmethod
    def get_results_over_x_games(p1: Player, p2: Player,
games: int, change_order: bool = True, reset_rnd_gen:
bool = False) -> tuple[int, int]:
        """
        Play x games and return the per player scores
        """
        tot = 0
        order = True if change_order else False
        for i in range(games):
            p1.reset(reset_rnd_gen)
            p2.reset(reset_rnd_gen)
            game = QuixoGame()
```

```
            if order == change_order:
                w = game.play(p1, p2)
            else:
                w = game.play(p2, p1)
                w = (w+1) % 2
            # game.print()
            tot += w
            if change_order:
                order = not order
        return games - tot, tot

    def check_winner(self) -> int:
        """
        Check the winner
        """
        # for each row
        for x in range(self._board.shape[0]):
            # if a player has completed an entire row
            if self._board[x, 0] != -1 and
all(self._board[x, :] == self._board[x, 0]):
                # return the relative id
                return self._board[x, 0]
        # for each column
        for y in range(self._board.shape[1]):
            # if a player has completed an entire column
            if self._board[0, y] != -1 and
all(self._board[:, y] == self._board[0, y]):
                # return the relative id
                return self._board[0, y]
        # if a player has completed the principal
diagonal
        if self._board[0, 0] != -1 and all(
                [self._board[x, x]
                 for x in range(self._board.shape[0])] ==
self._board[0, 0]
        ):
            # return the relative id
            return self._board[0, 0]
        # if a player has completed the secondary
diagonal
        if self._board[0, -1] != -1 and all(
                [self._board[x, -(x + 1)]
```

```
                 for x in range(self._board.shape[0])] ==
self._board[0, -1]
        ):
            # return the relative id
            return self._board[0, -1]
        return -1

    def _check_pos(self, pos: Sequence[int]):
        """
        Check the position validity
        """
        assert len(pos) == 2, f"Expected 2 dimensions
instead got {pos}"
        assert pos[0] >= 0 or pos[0] < self.BOARD_DIM,
f"Expected pos between 0-{self.BOARD_DIM} got {pos[0]}"
        assert pos[1] >= 0 or pos[1] < self.BOARD_DIM,
f"Expected pos between 0-{self.BOARD_DIM} got {pos[1]}"

    def is_current_player_pos(self, pos: Sequence[int]) -
> bool:
        """
        Return true if is the player position
        """
        self._check_pos(pos)
        if self._board[pos] == self.current_player_idx:
            return True
        return False

    def is_other_player_pos(self, pos: Sequence[int]) ->
bool:
        """
        Return true if is the opponent position
        """
        self._check_pos(pos)
        if self._board[pos] != self.current_player_idx
and self._board[pos] != -1:
            return True
        return False

    def is_void_pos(self, pos: Sequence[int]) -> bool:
        """
        Return true if is a void position
```

```
        """
        self._check_pos(pos)
        if self._board[pos] == -1:
            return True
        return False

    def is_move_doable(self, move: MyMove) -> bool:
        """
        Return true if the move is really doable
        """
        if move not in self.available_moves_set or
self.is_other_player_pos(move.position_reversed):
            return False
        return True

    def play(self, player1: Player, player2: Player) ->
int:
        """
        Play the game
        """
        players = [player1, player2]
        winner = -1
        while winner < 0:
            self.current_player_idx += 1
            self.current_player_idx %= len(players)

            from_pos, slide =
players[self.current_player_idx].make_move(self)
            self.__move(from_pos, slide,
self.current_player_idx)
            self._move_count += 1

            winner = self.check_winner()
            if self._move_count > 1000:
                return 0
            # self.print()
        return winner

    def move(self, from_pos: tuple[int, int], slide:
MoveDirection, player_id: int):
        """
        Make single move from outside
```

```
        """
        self.__move(from_pos, slide, player_id)
        self.current_player_idx =
(self.current_player_idx + 1) % 2
        self._move_count += 1

    def __move(self, from_pos: tuple[int, int], slide:
MoveDirection, player_id: int):
        """
        Make move
        """
        if player_id > 2:
            return False
        self.__take((from_pos[1], from_pos[0]),
player_id)
        self.__slide((from_pos[1], from_pos[0]), slide)

    def __take(self, from_pos: tuple[int, int],
player_id: int):
        """
        Take piece
        """
        self._board[from_pos] = player_id

    def __slide(self, from_pos: tuple[int, int], slide:
MoveDirection):
        """
        Slide piece
        """
        piece = self._board[from_pos]
        # if the player wants to slide it to the left
        if slide == MoveDirection.LEFT:
            # for each column starting from the column of
the piece and moving to the left
            for i in range(from_pos[1], 0, -1):
                # copy the value contained in the same
row and the previous column
                self._board[(from_pos[0], i)] =
self._board[(
                    from_pos[0], i - 1)]
            # move the piece to the left
            self._board[(from_pos[0], 0)] = piece
```

```
        # if the player wants to slide it to the right
        elif slide == MoveDirection.RIGHT:
            # for each column starting from the column of
the piece and moving to the right
            for i in range(from_pos[1],
self._board.shape[1] - 1, 1):
                # copy the value contained in the same
row and the following column
                self._board[(from_pos[0], i)] =
self._board[(
                    from_pos[0], i + 1)]
            # move the piece to the right
            self._board[(from_pos[0],
self._board.shape[1] - 1)] = piece
        # if the player wants to slide it upward
        elif slide == MoveDirection.TOP:
            # for each row starting from the row of the
piece and going upward
            for i in range(from_pos[0], 0, -1):
                # copy the value contained in the same
column and the previous row
                self._board[(i, from_pos[1])] =
self._board[(
                    i - 1, from_pos[1])]
            # move the piece up
            self._board[(0, from_pos[1])] = piece
        # if the player wants to slide it downward
        elif slide == MoveDirection.BOTTOM:
            # for each row starting from the row of the
piece and going downward
            for i in range(from_pos[0],
self._board.shape[0] - 1, 1):
                # copy the value contained in the same
column and the following row
                self._board[(i, from_pos[1])] =
self._board[(
                    i + 1, from_pos[1])]
            # move the piece down
            self._board[(self._board.shape[0] - 1,
from_pos[1])] = piece
```

terminal_set.py

```
from random import Random
from typing import Set, List, Optional

from quixo.decorator import classproperty
from quixo.function_set import _get_random_from_set
from quixo.quixo_game import QuixoGame
from quixo.value_point import ValuePoint


class TerminalSet:
    _set: List[ValuePoint] = [
        *[ValuePoint((i, j)) for i in
range(QuixoGame.BOARD_DIM) for j in
range(QuixoGame.BOARD_DIM)],
        ValuePoint.NIL
    ]

    @classproperty
    def set(cls) -> List[ValuePoint]:
        """
        Return the terminal set
        """
        return cls._set

    @classmethod
    def get_random_terminal(cls, rnd: Random, to_exclude:
Optional[ValuePoint] = None) -> ValuePoint:
        """
        Get random value point from terminal set
        """
        return _get_random_from_set(cls._set, rnd,
to_exclude)

    @classmethod
    def get_random_terminals(cls, rnd: Random, count) ->
List[ValuePoint]:
        """
```

```
        Get randoms value points from terminal set
        """
        return rnd.choices(cls._set, k=count)
```

value_point.py

```
from typing import Optional, Sequence, Tuple

from quixo.decorator import classproperty


class ValuePoint:
    def __init__(self, value: Optional[Sequence[int]] =
None):
        self._value = value
        if not self.is_nil():
            assert len(self._value) == 2, f"Value has len
{len(self._value)} instead of 2"
            self._value = tuple(self._value)

    def __str__(self):
        if self.is_nil():
            return "NIL"
        else:
            return f"P{self._value[0]}{self._value[1]}"

    def is_nil(self) -> bool:
        """
        Return true if value point is NIL
        """
        return self._value is None

    @property
    def point(self) -> Tuple[int, int]:
        """
        Return the value point position tuple
        """
        assert not self.is_nil(), "ValuePoint is None"
```

```
        return self._value

    @classproperty
    def NIL(cls) -> 'ValuePoint':
        """
        Generate a NIL value point
        """
        return ValuePoint(None)
```

Dimitri Masetta 306130

Quixo Report

Notable features:

- Space of states = 3^{25} as an upper limit, some states are unreachable.
- Space of actions = 44, max limit reachable only on the first move.
- Symmetry = 4 flips (horizontal, vertical, diagonal left, diagonal right) and 4 rotations.

Minmax agent:

The defined minmax agent explores all the solution tree by recursively evaluating each possible move until a defined depth. Alpha beta pruning was implemented to improve performance in terms of time.

As expected, the results are great for this kind of game with almost 100% of won matches against a random player. The only drawback is the execution time ~ 3s per games.

Genetic programming agent:

A genetic programming algorithm was developed, capable of evolve individuals that have a graph as their genome. The nodes of the graph consist of a function set and a terminal set inspired by [**Competitive Environments Evolve Better Solutions for Complex Tasks,** Peter J. Angeline and Jordan B. Pollack].

The recombination function can be:

- One node crossover
- One node mutation
- Reproduction.

Various mechanisms for selection and evaluation of individuals in the population have been developed:

Fitnessless coevolution, inspired by [**Fitnessless Coevolution,** Wojciech Ja«skowski, Krzysztof Krawiec, Bartosz Wieloch]

- Fitness evaluation against a random players or pre-trained ones followed by roulette fitness selection or tournament fitness selection.

After many tests on all the major configuration, the one leading to the best individuals is as follows:

- Population size = 50
- Selection size = 25
- Crossover probability = 0.2 (despite the usual 0.8, no particular bloating detected)
- Reproduction probability = 1 - Crossover probability
- Mutation probability = 0.4 (despite the usual 0.1)
- Round against random = 30
- Fitness evaluation without coevolution against random
- Fitness selection roulette wheel

With the above parameters with 100 generations, we arrive at agents that win on average 91% of the time against a random agent, with a match time of only ~2ms.

Try it for yourself:

To try the agents and see the results described above try the notebook *play_the_game.ipynb*