Report Computational Intelligence

Student: s280117 - Giuseppe Piombino

Project repo: https://github.com/Girasolo/computational_intelligence_exam

Lab1

README.md

s280117 Worked in group with 302294 and 291871 however code is different

N=5, A* [list([2]) list([0, 1]) list([3]) list([4])] Found a solution in 5 steps; visited 61 states The weight is 5 N=5, breadth first [list([0]) list([1, 3]) list([2, 4])] Found a solution in 4 steps; visited 3,454 states The weight is 5

N=10,A* [list([2, 5]) list([8, 1, 6]) list([9, 3]) list([0, 4, 7])] Found a solution in 5 steps; visited 2,558 states The weight is 10 N=10, breadth first [list([1, 2, 3]) list([0, 3, 4, 7, 9]) list([3, 4, 5, 6, 8])] Found a solution in 4 steps; visited 197,080 states The weight is 13

N=20,A* [list([0, 5, 11, 16, 17]) list([8, 4, 7]) list([2, 6, 8, 10, 12, 15, 18]) list([16, 9, 19, 6]) list([1, 3, 13, 14])] Found a solution in 6 steps; visited 467,479 states The weight is 23 N=20,breadth first [list([0, 1, 2, 3, 6, 13, 17, 18]) list([0, 6, 16, 17, 19]) list([4, 7, 11, 12, 15, 16, 18]) list([0, 3, 5, 8, 9, 10, 13, 14, 17])] Found a solution in 5 steps; visited 2,172,545 states The weight is 29

N=100,A* execution time>5 min N=100,breadth first execution time>5 min

lab1.ipynb

```
{
import random
import numpy as np
from typing import Callable
from gx_utils import *
import logging

N=20
SEED=42
def problem(N, seed=None):
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N // 5, N // 2))))
        for n in range(random.randint(N, N * 5))
    ]
class State:
    def __init__(self, data: np.ndarray):
        self._data = data.copy()
        self._data.flags.writeable = False

    def __hash__(self):
        return hash(bytes(self._data))
        '''str_=''
        for list_ in self._data:
            #for el in list_:
            #    str_+=str(el)
            #str_+='|'
            str_+=str(list_)
        print(str_)
        if(str_==''):
            str_='[]'
        return hash(str_)'''


    def __eq__(self, other):
        return bytes(self._data) == bytes(other._data)
```

```
    def __lt__(self, other):
        return bytes(self._data) < bytes(other._data)

    def __str__(self):
        return str(self._data)

    def __repr__(self):
        return repr(self._data)

    @property
    def data(self):
        return self._data

    def copy_data(self):
        return self._data.copy()

P=np.array(problem(N,SEED),dtype=object)

INITIAL_STATE= State(np.array([],dtype=object))
def is_valid(frozenset_: np.array, action: list):
    #print(action)
    for list_ in frozenset_:
        #print(list_)
        if set(list_)==set(action):
            return False

    return True

#setEx=np.array([[0],[1,2],[3,4,5],[6]],dtype=object)
#actionEx=[2,1]

#print(is_valid(setEx,actionEx))


def possible_actions(state: State):
    return [list_ for list_ in P if is_valid(state._data, list_)]

#for a in possible_actions(State(setEx)):
#    print(a)
def result(state,action):
    temp=state.copy_data().tolist()
    #print(temp+[action])
    return State(np.array(temp+[action],dtype=object))

#setEx=np.array([[0],[1,2],[3,4,5],[6]],dtype=object)
#actionEx=[1,3]

#print(result(State(setEx),actionEx))
#GOAL = State(np.array(list(range(1, SIZE**2)) + [0]).reshape((SIZE, SIZE)))
#logging.info(f"Goal:\n{GOAL}")

'''
def goal_test(state):
    goal=np.zeros(N)
    for list_ in P:
        for el in list_:
            goal[el]=1
            if np.sum(goal)==N:
                return True
    if np.sum(goal)==N:
        return True
    else:
        return False
'''


def goal_test(state):
    goal=set()
```

```
    for list_ in state._data:
        goal.update(list_)
    return len(goal)== N

#goal_test(P)







def search(
    initial_state: State,
    goal_test: Callable,
    parent_state: dict,
    state_cost: dict,
    priority_function: Callable,
    unit_cost: Callable,
):
    frontier = PriorityQueue()
    parent_state.clear()
    state_cost.clear()

    state = initial_state
    parent_state[state] = None
    state_cost[state] = 0

    while state is not None and not goal_test(state):
        for a in possible_actions(state):
            new_state = result(state, a)
            cost = unit_cost(a)
            if new_state not in state_cost and new_state not in frontier:
                parent_state[new_state] = state
                state_cost[new_state] = state_cost[state] + cost
                frontier.push(new_state, p=priority_function(new_state))
                logging.debug(f"Added new node to frontier (cost={state_cost[new_state]})")
            elif new_state in frontier and state_cost[new_state] > state_cost[state] + cost:
                old_cost = state_cost[new_state]
                parent_state[new_state] = state
                state_cost[new_state] = state_cost[state] + cost
                logging.debug(f"Updated node cost in frontier: {old_cost} -> {state_cost[new_state]}")
        if frontier:
            state = frontier.pop()
        else:
            state = None

    path = list()
    s = state
    print(s)
    while s:
        path.append(s.copy_data())
        s = parent_state[s]

    print(f"Found a solution in {len(path):,} steps; visited {len(state_cost):,} states")
    print(f"The weight is {state_cost[state]}")
    logging.info(f"Found a solution in {len(path):,} steps; visited {len(state_cost):,} states")
    return list(reversed(path))
BREADTH-FIRST

parent_state=dict()
state_cost=dict()
'''
final=search(
    INITIAL_STATE,
    goal_test=goal_test,
    parent_state=parent_state,
    state_cost=state_cost,
    priority_function=lambda s: len(state_cost),
```

```
    unit_cost=lambda a: len(a),
)
'''
'\nfinal=search(\n    INITIAL_STATE,\n    goal_test=goal_test,\n    parent_state=parent_state,\n
state_cost=state_cost,\n    priority_function=lambda s: len(state_cost),\n    unit_cost=lambda a:
len(a),\n)\n'
A*

parent_state = dict()
state_cost = dict()


def h(state):
    goal=set()
    for list_ in state._data:
        goal.update(list_)
    #g = set([i for i in range(N)])
    #difference = g.difference(goal)
    #return sum(difference)
    return N-len(goal)


#setEx=np.array([[0],[1,2],[3,4,5],[6]],dtype=object)
#actionEx=[2,1]

#print(set(setEx))
final = search(
    INITIAL_STATE,
    goal_test=goal_test,
    parent_state=parent_state,
    state_cost=state_cost,
    priority_function=lambda s: state_cost[s] + h(s),
    unit_cost=lambda a: len(a),
)
[list([1, 3, 13, 14]) list([2, 6, 8, 10, 12, 15, 18]) list([8, 4, 7])
 list([16, 9, 19, 6]) list([0, 5, 11, 16, 17])]
Found a solution in 6 steps; visited 441,019 states
The weight is 23

}
```

Review lab1

REVIEW LAB1 of MatteMagnaldi

https://github.com/MatteMagnaldi/Computational_intelligence2022

REVIEW BY GIUSEPPE PIOMBINO S280117

- pycache not needed ignore it with gitignored
- in possible_actions you don't need to check if the state is a subset of set(p) (function check_subset). If you add to the state a list which has all the already added elemts plus one it's anyway an improvement of the solution
- nice that you cut the list of possible action by using the index of the last_el
- really like the use of the density in solution2

Code

```
{
import random
import logging
import itertools
from queue import PriorityQueue
from gx_utils import *
logging.basicConfig(format="%(message)s", level=logging.INFO)
def problem(N, seed=None):
```

```
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N // 5, N // 2))))
        for n in range(random.randint(N, N * 5))
    ]

def remove_duplicates(P):
    set_list = []
    for p in P:
        if p not in set_list and len(p) != 0:
            set_list.append(p)
    return set_list


def goal_test(state):
    return set(itertools.chain(*state)) == set(range(0,N))

def check_subset(a: set, b: set):
    return  not set(a).issubset(b) and not b.issubset(set(a))

def possible_actions(state, P):
    last = -1
    state_set = set(itertools.chain(*state))
    if len(state) != 0:
        last_el = list(state[-1])
        last = P.index(last_el)
    return (tuple(p) for i,p in enumerate(P) if not state_set or (i > last and check_subset(state_set,
set(p))))

def path_cost(state):
    return sum(len(p) for p in state)

def search(P : list,
           goal_test,
           path_cost,
           priority_function ):

    frontier = PriorityQueue()
    explored_nodes = 0
    state = list()
    cost = dict()
    while state is not None and not goal_test(state):
        explored_nodes+=1

        for action in possible_actions(state, P):

            new_state = tuple([*state, action])

            #logging.info(f"found state: {new_state}:")
            if new_state not in cost and new_state not in frontier:
                #logging.info(f"\t\t-- Added to the frontier")
                cost[new_state] = path_cost(new_state)
                #logging.info(f"\t\t-- with cost: {cost[new_state]}")
                frontier.push(new_state, p=priority_function(new_state) )
        if frontier:
            state = frontier.pop()
        else:
            logging.info("Unsolvable")
            return
    logging.info(f"visited {explored_nodes} nodes.")
    return state, cost[state]
Solution 1:
Priority function = path cost
Already examineted nodes are not considered, using pruning into the possible action mothod
Find optimal solution
Computational limit N = 20

N = 20
```

```
P = problem(N, seed=42)

P = remove_duplicates(P)
#logging.info(f"P set: {P}.")
search(P, goal_test, path_cost, path_cost)
Solution 2:
A* search
Priority function = len of the state - density of the state
Shortest and most denes state preferred
Already examineted nodes are not considered, using pruning into the possible action mothod
Find optimal solution
Computational limit N = 20
Faster then first solution, visited less nodes

def priority_function(state):
    state_len = sum(len(p) for p in state)
    return state_len - len(set(itertools.chain(*state))) / state_len

N = 20
P = problem(N, seed=42)
P = remove_duplicates(P)
#logging.info(f"P set: {P}.")
search(P, goal_test, path_cost, priority_function)
Solution 3:
Priority function = longest and most dense state first
Already examineted nodes are not considered, using pruning into the possible action mothod
Not optimal solution
Much faster than other solutions

def priority_function(state):
    state_len = sum(len(p) for p in state)
    return  -(len(set(itertools.chain(*state))) / state_len + state_len/N)

N =1000
P = problem(N,seed=42)
P=remove_duplicates(P)
search(P,goal_test,path_cost,priority_function)

}
```

REVIEW LAB1 of Andred00

https://github.com/Andred00/computationalintelligence2022-s303339

REVIEW BY GIUSEPPE PIOMBINO S280117

- compared to other codes yours is incredibly fast and allows to find a solution for bigger N. Choosing to start with the longest list allows you to cut the time of computing but doesn't assure you to obtain the optimal value, in fact other implementations have find a better solution for N=20
- it was possible to avoid the use of flat_solution and use inside the h() function directly the function flatten() of numpy.ndarray
- the comments weren't necessary since you explained the algorithm in the README file and the variables names were appropriate

Code

```
{
Copyright (c) 2022 Giovanni Squillero <squillero@polito.it>
https://github.com/squillero/computational-intelligence
Free for personal or classroom use; see LICENSE.md for details.

Lab 1: Set Covering
First lab + peer review. List this activity in your final report, it will be part of your exam.

Task
Given a number
 and some lists of integers
```

```
, determine, if possible,
 such that each number between
 and
 appears in at least one list

and that the total numbers of elements in all
 is minimum.

Instructions
Create the directory lab1 inside the course repo (the one you registered with Andrea)
Put a README.md and your solution (all the files, code and auxiliary data if needed)
Use problem to generate the problems with different
In the README.md, report the the total numbers of elements in
 for problem with
 and the total number on
 visited during the search. Use seed=42.
Use GitHub Issues to peer review others' lab
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.
Deadline

Sunday, October 16th 23:59:59 for the working solution
Sunday, October 23rd 23:59:59 for the peer reviews
import random
def problem(N, seed=None):
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N // 5, N // 2))))
        for n in range(random.randint(N, N * 5))
    ]
import logging
import copy

def h(sol, current_x):
    common_elements = len(set(sol) & set(current_x))
    new_elements = len(current_x) - common_elements
    if (new_elements == 0):
        return float('inf')
    return common_elements/new_elements

def greedy(N):
    goal = set(range(N))

    lists = sorted(problem(N, seed=42), key=lambda l: -len(l))

    starting_x = lists.pop(0)

    sol = list()
    sol.append(starting_x)

    flat_sol = list(starting_x)
    nodes = 1

    covered = set(starting_x)
    while goal != covered:
        most_promising_x = min(lists, key = lambda x: h(flat_sol, x))
        lists.remove(most_promising_x)

        flat_sol.extend(most_promising_x)
        sol.append(most_promising_x)
        nodes = nodes + 1

        covered |= set(most_promising_x)

    w = len(flat_sol)
```

```
    logging.info(
        f"Greedy solution for N={N}: w={w} (bloat={(w-N)/N*100:.0f}%) - visited {nodes} nodes"
    )
    logging.debug(f"{sol}")
    return sol
logging.getLogger().setLevel(logging.INFO)

for N in [5, 10, 20, 100, 500, 1000]:
   greedy(N)
INFO:root:Greedy solution for N=5: w=5 (bloat=0%) - visited 3 nodes
INFO:root:Greedy solution for N=10: w=10 (bloat=0%) - visited 3 nodes
INFO:root:Greedy solution for N=20: w=24 (bloat=20%) - visited 4 nodes
INFO:root:Greedy solution for N=100: w=182 (bloat=82%) - visited 8 nodes
INFO:root:Greedy solution for N=500: w=1262 (bloat=152%) - visited 11 nodes
INFO:root:Greedy solution for N=1000: w=2878 (bloat=188%) - visited 13 nodes
```

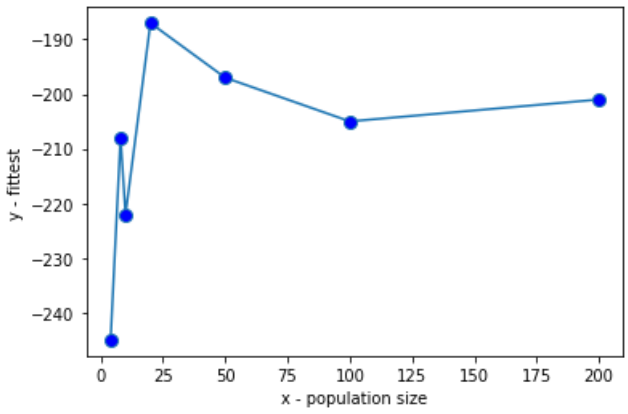
}

Lab2

s280117 Worked in group with 291871 and 302294 consulted with 292113

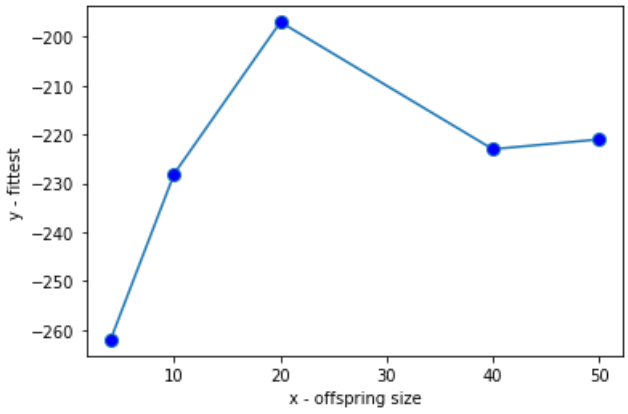
Vary population size (fitness call=20000)

time: 0.96 fittest: (100, -245) time: 0.94 fittest: (100, -208) time: 0.93 fittest: (100, -222) time: 0.96 fittest: (100, -187) time: 1.03 fittest: (100, -197) time: 1.16 fittest: (100, -205) time: 1.40 fittest: (100, -201)



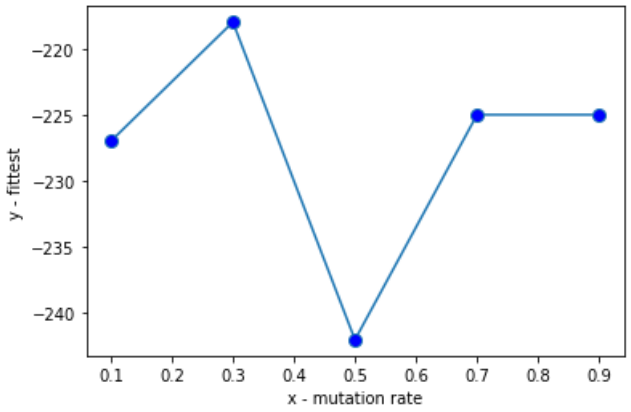
Vary the offspring size

time: 0.25 fittest: (100, -262) time: 0.52 fittest: (100, -228) time: 0.98 fittest: (100, -197) time: 1.88 fittest: (100, -223) time: 2.37 fittest: (100, -221)



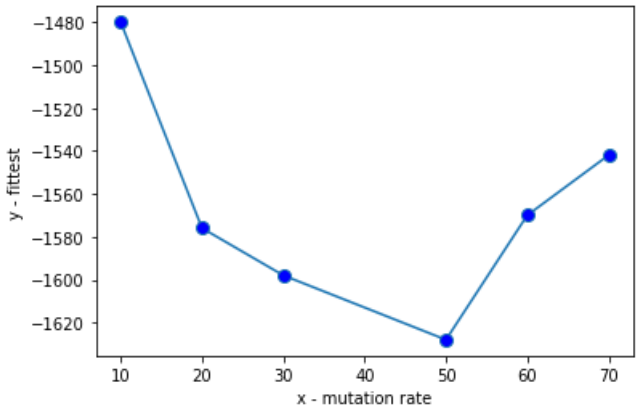
Vary the mutation rate

time: 0.99 fittest: (100, -227) time: 0.97 fittest: (100, -218) time: 0.96 fittest: (100, -242) time: 0.96 fittest: (100, -225) time: 0.96 fittest: (100, -225)



Vary the cleaner rate

time: 47.74 fittest: (500, -1480) time: 43.75 fittest: (500, -1576) time: 40.68 fittest: (500, -1598) time: 41.44 fittest: (500, -1628) time: 40.22 fittest: (500, -1570)



Final trial

To let the computation be feasible have been chosen small but decent value of population and offspring size instead of the best ones time: 0.36 peak memory: 120.03 MiB fittest: (5, -5) time: 0.39 peak memory: 120.15 MiB fittest: (10, -10) time: 0.44 peak memory: 120.15 MiB fittest: (20, -23) time: 2.38 peak memory: 120.48 MiB fittest: (100, -229) time: 9.50 peak memory: 127.90 MiB fittest: (500, -1460) time: 20.38 peak memory: 157.10 MiB fittest: (1000, -3728) time: 54.80 peak memory: 275.57 MiB fittest: (2000, -9209) time: 107.20 peak memory: 599.98 MiB fittest: (5000, -24806) time: 444.04 peak memory: 2074.37 MiBfittest: (10000, -55340)

Strategy2

time: 0.25 peak memory: 119.12 MiB fittest: (5, -5) time: 0.27 peak memory: 119.24 MiB fittest: (10, -14) time: 0.40 peak memory: 119.26 MiB fittest: (20, -35) time: 1.93 peak memory: 119.45 MiB fittest: (100, -257) time: 7.78 peak memory: 126.46 MiB fittest: (500, -1908) time: 20.20 peak memory: 154.77 MiB fittest: (1000, -4354) time: 45.22 peak memory: 272.02 MiB fittest: (2000, -9560) time: 112.67 peak memory: 604.85 MiB fittest: (5000, -26718) time: 271.91 peak memory: 2073.34 MiB fittest: (10000, -56751)

lab2.ipynb

```
{
import random
from collections import namedtuple
import numpy as np
import matplotlib.pyplot as plt
from copy import copy
import time
#!pip3 install memory_profiler
%load_ext memory_profiler
def problem(N, seed=None):
    random.seed(seed)
    return list(sorted({
        tuple(set(random.randint(0, N - 1) for n in range(random.randint(N // 5, N // 2))))
```

```
        for n in range(random.randint(N, N * 5))})
    )
Individual= namedtuple('Individual',['genome','fitTuple'])

def fitness_function(genome: list):
    '''Returns a tuple (number_of_covered_elements, -weigth)'''
    #this fitness function allows the precence of not optimal solution. The hierarchy of them shold be
    # based at first of the #ofCoverdEl and then on the lighter
    num_covered_elements = set()
    weigth = 0
    for (i, list_) in enumerate(ALL_LISTS):
        if genome[i]:
            num_covered_elements.update(list_)
            weigth += len(list_)

    return len(num_covered_elements), -weigth #the reason why weight is negative is we want to minimize
it
def tournament(population, tournament_size=2):
    return max(random.choices(population, k=tournament_size), key=lambda i: i.fitTuple)


def cross_over(g1, g2):
    cut = random.randint(0, len(ALL_LISTS)-1)
    choice=random.randint(0,2)
    if choice==0:
        return g1[:cut] + g2[cut:]
    else:
        return g2[:cut] + g1[cut:]


def mutation(g):
    point = random.randint(0, len(ALL_LISTS) - 1)
    return g[:point] + (1 - g[point],) + g[point + 1 :]

def multiple_mutation(g):
    '''using multiple it's worse, because mutation should be small'''
    nflips=random.randint(1,100)
    for _ in range(nflips):
        g=mutation(g)
    return g

def vacuum_cleaner(g):
    '''takes some weight off by switching off a list'''
    temp_g=g
    for _ in range (random.randint(0,len(ALL_LISTS))):
        point=random.randint(0, len(ALL_LISTS) - 1)
        if temp_g[point] == 1:
            temp_g= temp_g[:point] + (0,) + temp_g[point + 1 :]
    return temp_g


def deClonizator(population):
    '''returns a population without duplicates'''
    unique_population=list()
    for individual in population:
        if individual not in unique_population:
            unique_population.append(individual)
    return unique_population
def GA(population, NUM_GENERATIONS, OFFSPRING_SIZE, mutation_rate,POPULATION_SIZE,cleaner_rate):
    start = time.time()                 #to take note of the computation time
    fittestIndividual=copy(population[0])
    for g in range(NUM_GENERATIONS):
        offspring = list()
        for i in range(OFFSPRING_SIZE): #generation of the offSprings
            if random.random() < mutation_rate:
                p = tournament(population)
                if g<NUM_GENERATIONS/cleaner_rate:
                    o= vacuum_cleaner(p.genome)
```

```
                else:
                    o = mutation(p.genome)
                #o = mutation(p.genome)

            else:
                p1 = tournament(population)
                p2 = tournament(population)
                o = cross_over(p1.genome, p2.genome)
            offspring.append(Individual(o, fitness_function(o)))
        population += offspring  #offspring added to the population

        #population=deClonizator(population)
        population=list(dict.fromkeys(population))  #deletion of duplicates

        population = sorted(population, key=lambda i: i.fitTuple, reverse=True)[:POPULATION_SIZE]  #cut
the population to the original size
        if(population[0].fitTuple>fittestIndividual.fitTuple):  #copy the fittest individual
            fittestIndividual=copy(population[0])
    end = time.time()
    delta=format(end-start, '.2f')
    #fitCall=NUM_GENERATIONS*OFFSPRING_SIZE #take note of #ofFitnessFunctionCall
    #print( "time: " + str(delta)+ " fit call " + str(fitCall))
    print( "time: " + str(delta) + " fittest: " + str(fittestIndividual.fitTuple))
    return fittestIndividual
Comparison between GA with cleaner and without it
def GAclean(population, NUM_GENERATIONS, OFFSPRING_SIZE, mutation_rate,POPULATION_SIZE,cleaner_rate):
    start = time.time()                #to take note of the computation time
    fittestIndividual=copy(population[0])
    fitArray_=list()
    for g in range(NUM_GENERATIONS):
        offspring = list()
        for i in range(OFFSPRING_SIZE): #generation of the offSprings
            if random.random() < mutation_rate:
                p = tournament(population)
                if g<NUM_GENERATIONS/cleaner_rate:
                    o= vacuum_cleaner(p.genome)
                else:
                    o = mutation(p.genome)
                #o = mutation(p.genome)

            else:
                p1 = tournament(population)
                p2 = tournament(population)
                o = cross_over(p1.genome, p2.genome)
            offspring.append(Individual(o, fitness_function(o)))
        population += offspring  #offspring added to the population

        #population=deClonizator(population)
        population=list(dict.fromkeys(population))  #deletion of duplicates

        population = sorted(population, key=lambda i: i.fitTuple, reverse=True)[:POPULATION_SIZE]  #cut
the population to the original size
        if g%100==0:
            fitArray_.append(copy(fittestIndividual.fitTuple[1]))
        if(population[0].fitTuple>fittestIndividual.fitTuple):  #copy the fittest individual
            fittestIndividual=copy(population[0])
    end = time.time()
    delta=format(end-start, '.2f')
    #fitCall=NUM_GENERATIONS*OFFSPRING_SIZE #take note of #ofFitnessFunctionCall
    #print( "time: " + str(delta)+ " fit call " + str(fitCall))
    print( "time: " + str(delta) + " fittest: " + str(fittestIndividual.fitTuple))
    return fittestIndividual,fitArray_
def GAnoClean(population, NUM_GENERATIONS, OFFSPRING_SIZE, mutation_rate,POPULATION_SIZE):
    start = time.time()                #to take note of the computation time
    fittestIndividual=copy(population[0])
    fitArray_=list()
    for g in range(NUM_GENERATIONS):
        offspring = list()
```

```
        for i in range(OFFSPRING_SIZE): #generation of the offSprings
            if random.random() < mutation_rate:
                p = tournament(population)
                o = mutation(p.genome)
                #o = mutation(p.genome)

            else:
                p1 = tournament(population)
                p2 = tournament(population)
                o = cross_over(p1.genome, p2.genome)
            offspring.append(Individual(o, fitness_function(o)))
        population += offspring  #offspring added to the population

        #population=deClonizator(population)
        population=list(dict.fromkeys(population))  #deletion of duplicates

        population = sorted(population, key=lambda i: i.fitTuple, reverse=True)[:POPULATION_SIZE]  #cut
the population to the original size
        if g%100==0:
            fitArray_.append(copy(fittestIndividual.fitTuple[1]))
        if(population[0].fitTuple>fittestIndividual.fitTuple):  #copy the fittest individual
            fittestIndividual=copy(population[0])
    end = time.time()
    delta=format(end-start, '.2f')
    fitCall=NUM_GENERATIONS*OFFSPRING_SIZE #take note of #ofFitnessFunctionCall
    #print( "time: " + str(delta)+ " fit call " + str(fitCall))
    print( "time: " + str(delta) + " fittest: " + str(fittestIndividual.fitTuple))
    return fittestIndividual,fitArray_
population_SIZE=20
PROBLEM_SIZE=500
OFFSPRING_SIZE=20
mutation_rate=0.5
cleaner_rate=10
NUM_GENERATIONS=1000
i=0
ALL_LISTS=problem(PROBLEM_SIZE,42)
fittest=list()

population = list()

for genome in [tuple([random.choice([1, 0]) for _ in range(len(ALL_LISTS))]) for _ in
range(population_SIZE)]:
    population.append(Individual(genome,fitness_function(genome)))

population = sorted(population, key=lambda i: i.fitTuple, reverse=True)
fittest_ind,cleanFitArray=GAclean(population,NUM_GENERATIONS,OFFSPRING_SIZE,mutation_rate,population_SIZE
,cleaner_rate)
fittest_ind,fitArray=GAnoClean(population,NUM_GENERATIONS,OFFSPRING_SIZE,mutation_rate,population_SIZE)
print(cleanFitArray)
print(fitArray)
range_=range(0,1000,100)
plt.plot(range_,cleanFitArray, fitArray,marker='o', markerfacecolor='blue', markersize=3)
plt.xlabel('x - problem size')
plt.ylabel('y - fittest')
time: 4.91 fittest: (500, -1460)
time: 9.22 fittest: (500, -2323)
[-129941, -1480, -1460, -1460, -1460, -1460, -1460, -1460, -1460, -1460]
[-129941, -41641, -34870, -27636, -21250, -16377, -12214, -8871, -5626, -3747]
Text(0, 0.5, 'y - fittest')

GA strategy2
def GAstrategy2(population, NUM_GENERATIONS, OFFSPRING_SIZE, mutation_rate,POPULATION_SIZE,cleaner_rate):
    start = time.time()                 #to take note of the computation time
    fittestIndividual=copy(population[0])
    for g in range(NUM_GENERATIONS):
        for i in range(OFFSPRING_SIZE):
            o=0
            if random.random() < mutation_rate:
```

```
                p = tournament(population)
                if g<NUM_GENERATIONS/cleaner_rate:
                    o= vacuum_cleaner(p.genome)
                else:
                    o = mutation(p.genome)
            if random.random() < 0.5:
                p1 = tournament(population)
                p2 = tournament(population)
                o = cross_over(p1.genome, p2.genome)
            if o:
                population.append(Individual(o, fitness_function(o)))
                population = sorted(population, key=lambda i: i.fitTuple, reverse=True)
                del population[-1]
            #in the strategy2 the population is cut every time we add a new individual

        #population=deClonizator(population)
        population=list(dict.fromkeys(population))  #deletion of duplicates

        if(population[0].fitTuple>fittestIndividual.fitTuple):
            fittestIndividual=copy(population[0])
    end = time.time()
    delta=format(end-start, '.2f')
    #fitCall=NUM_GENERATIONS*OFFSPRING_SIZE #take note of #ofFitnessFunctionCall
    #print( "time: " + str(delta)+ " fit call " + str(fitCall))
    print( "time: " + str(delta) + " fittest: " + str(fittestIndividual.fitTuple))
    return fittestIndividual
Vary the population size
population_SIZE=[4,8,10,20,50,100,200]
PROBLEM_SIZE=500
OFFSPRING_SIZE=20
mutation_rate=0.3
cleaner_rate=20
NUM_GENERATIONS=1000
N=3
ALL_LISTS=problem(PROBLEM_SIZE,42)
fittest=list()
i=0
for pop_size in population_SIZE:
    population = list()

    for genome in [tuple([random.choice([1, 0]) for _ in range(len(ALL_LISTS))]) for _ in
range(pop_size)]:
        population.append(Individual(genome,fitness_function(genome)))

    population = sorted(population, key=lambda i: i.fitTuple, reverse=True)
    fittest_ind=GA(population,NUM_GENERATIONS,OFFSPRING_SIZE,mutation_rate,pop_size,cleaner_rate)
    fittest.append(fittest_ind.fitTuple[1])
plt.plot(population_SIZE,fittest, marker='o', markerfacecolor='blue', markersize=8)
plt.xlabel('x - population size')
# naming the y axis
plt.ylabel('y - fittest')
time: 4.08 fittest: (500, -1951)
time: 3.98 fittest: (500, -1688)
time: 3.92 fittest: (500, -1668)
time: 4.06 fittest: (500, -1644)
time: 4.49 fittest: (500, -1655)
time: 5.20 fittest: (500, -1591)
time: 6.43 fittest: (500, -1572)
Text(0, 0.5, 'y - fittest')

Vary the offspring size
population_SIZE=20
PROBLEM_SIZE=500
OFFSPRING_SIZE=[4,10,20,40,50]
mutation_rate=0.3
cleaner_rate=20
NUM_GENERATIONS=1000
N=3
```

```
ALL_LISTS=problem(PROBLEM_SIZE,42)
fittest=list()
i=0
for off_size in OFFSPRING_SIZE:
    population = list()

    for genome in [tuple([random.choice([1, 0]) for _ in range(len(ALL_LISTS))]) for _ in
range(population_SIZE)]:
        population.append(Individual(genome,fitness_function(genome)))

    population = sorted(population, key=lambda i: i.fitTuple, reverse=True)
    fittest_ind=GA(population,NUM_GENERATIONS,off_size,mutation_rate,population_SIZE,cleaner_rate)
    fittest.append(fittest_ind.fitTuple[1])
plt.plot(OFFSPRING_SIZE,fittest, marker='o', markerfacecolor='blue', markersize=8)
plt.xlabel('x - offspring size')
# naming the y axis
plt.ylabel('y - fittest')
time: 1.06 fittest: (500, -1799)
time: 2.17 fittest: (500, -1663)
time: 4.15 fittest: (500, -1632)
time: 8.17 fittest: (500, -1769)
time: 9.83 fittest: (500, -1685)
Text(0, 0.5, 'y - fittest')

Vary the mutation rate
population_SIZE=20
PROBLEM_SIZE=500
OFFSPRING_SIZE=20
mutation_rate=[0.1,0.3,0.5,0.7,0.9]
cleaner_rate=20
NUM_GENERATIONS=1000
ALL_LISTS=problem(PROBLEM_SIZE,42)
fittest=list()
i=0
for mut in mutation_rate:
    population = list()

    for genome in [tuple([random.choice([1, 0]) for _ in range(len(ALL_LISTS))]) for _ in
range(population_SIZE)]:
        population.append(Individual(genome,fitness_function(genome)))

    population = sorted(population, key=lambda i: i.fitTuple, reverse=True)
    fittest_ind=GA(population,NUM_GENERATIONS,OFFSPRING_SIZE,mut,population_SIZE,cleaner_rate)
    fittest.append(fittest_ind.fitTuple[1])
plt.plot(mutation_rate,fittest, marker='o', markerfacecolor='blue', markersize=8)
plt.xlabel('x - mutation rate')
# naming the y axis
plt.ylabel('y - fittest')
time: 4.56 fittest: (500, -1642)
time: 5.09 fittest: (500, -1704)
time: 4.60 fittest: (500, -1629)
time: 4.61 fittest: (500, -1509)
time: 4.79 fittest: (500, -1609)
Text(0, 0.5, 'y - fittest')

Vary the cleaner rate
population_SIZE=20
PROBLEM_SIZE=500
OFFSPRING_SIZE=20
mutation_rate=0.7
cleaner_rate=[10,20,30,50,60,70]
NUM_GENERATIONS=1000
ALL_LISTS=problem(PROBLEM_SIZE,42)
fittest=list()
i=0
for clean_ in cleaner_rate:
    population = list()
```

```
    for genome in [tuple([random.choice([1, 0]) for _ in range(len(ALL_LISTS))]) for _ in
range(population_SIZE)]:
        population.append(Individual(genome,fitness_function(genome)))

    population = sorted(population, key=lambda i: i.fitTuple, reverse=True)
    fittest_ind=GA(population,NUM_GENERATIONS,OFFSPRING_SIZE,mutation_rate,population_SIZE,clean_)
    fittest.append(fittest_ind.fitTuple[1])
plt.plot(cleaner_rate,fittest, marker='o', markerfacecolor='blue', markersize=8)
plt.xlabel('x - mutation rate')
# naming the y axis
plt.ylabel('y - fittest')
time: 5.07 fittest: (500, -1595)
time: 4.51 fittest: (500, -1607)
time: 4.26 fittest: (500, -1736)
time: 4.20 fittest: (500, -1795)
time: 4.09 fittest: (500, -1777)
time: 4.07 fittest: (500, -1724)
Text(0, 0.5, 'y - fittest')

Final trial
To let the computation be feasible have been chosen small but decent value of population and offspring
size instead of the best ones

population_SIZE=20
PROBLEM_SIZE=[5,10,20,100,500,1000,2000]
PROBLEM_SIZE_=[5000,10000]
PROBLEM_SIZE_=[20000,50000] #unfeasible
OFFSPRING_SIZE=20
mutation_rate=0.5
cleaner_rate=10
NUM_GENERATIONS=1000
for prbl_size in PROBLEM_SIZE:
    ALL_LISTS=problem(prbl_size,42)
    fittest=list()

    population = list()

    for genome in [tuple([random.choice([1, 0]) for _ in range(len(ALL_LISTS))]) for _ in
range(population_SIZE)]:
        population.append(Individual(genome,fitness_function(genome)))

    population = sorted(population, key=lambda i: i.fitTuple, reverse=True)
    %memit
fittest_ind=GA(population,NUM_GENERATIONS,OFFSPRING_SIZE,mutation_rate,population_SIZE,cleaner_rate)
time: 0.15 fittest: (5, -5)
time: 0.15 fittest: (5, -5)
peak memory: 130.32 MiB, increment: 0.19 MiB
time: 0.21 fittest: (10, -10)
time: 0.21 fittest: (10, -10)
peak memory: 130.09 MiB, increment: -0.23 MiB
time: 0.21 fittest: (20, -23)
time: 0.21 fittest: (20, -28)
peak memory: 130.11 MiB, increment: 0.02 MiB
time: 1.04 fittest: (100, -229)
peak memory: 130.44 MiB, increment: 0.29 MiB
time: 4.65 fittest: (500, -1460)
peak memory: 136.55 MiB, increment: 1.08 MiB
time: 10.59 fittest: (1000, -3728)
peak memory: 166.24 MiB, increment: 0.80 MiB
time: 29.00 fittest: (2000, -9209)
peak memory: 284.47 MiB, increment: 4.73 MiB
Strategy2
population_SIZE=20
PROBLEM_SIZE=[5,10,20,100,500,1000,2000]
PROBLEM_SIZE_=[5000,10000]
OFFSPRING_SIZE=20
mutation_rate=0.3
cleaner_rate=10
```

```
NUM_GENERATIONS=1000
i=0
for prbl_size in PROBLEM_SIZE:
    ALL_LISTS=problem(prbl_size,42)
    fittest=list()

    population = list()

    for genome in [tuple([random.choice([1, 0]) for _ in range(len(ALL_LISTS))]) for _ in
range(population_SIZE)]:
        population.append(Individual(genome,fitness_function(genome)))

    population = sorted(population, key=lambda i: i.fitTuple, reverse=True)
    %memit
fittest_ind=GAstrategy2(population,NUM_GENERATIONS,OFFSPRING_SIZE,mutation_rate,population_SIZE,cleaner_r
ate)
    fittest.append(fittest_ind.fitTuple[1])
time: 0.11 fittest: (5, -5)
time: 0.11 fittest: (5, -5)
peak memory: 130.55 MiB, increment: -4.09 MiB
time: 0.17 fittest: (10, -13)
time: 0.16 fittest: (10, -14)
peak memory: 130.46 MiB, increment: -0.09 MiB
time: 0.18 fittest: (20, -31)
time: 0.16 fittest: (20, -33)
peak memory: 130.48 MiB, increment: 0.02 MiB
time: 0.69 fittest: (100, -263)
peak memory: 130.81 MiB, increment: 0.20 MiB
time: 3.11 fittest: (500, -1939)
peak memory: 136.32 MiB, increment: 0.73 MiB
time: 7.53 fittest: (1000, -4114)
peak memory: 166.02 MiB, increment: 0.00 MiB
time: 18.02 fittest: (2000, -9438)
peak memory: 281.43 MiB, increment: 1.66 MiB
}
```

Presentation of lab2

presentationLab2.pdf

GA FOR SET COVERING PROBLEM

WORK GROUP

- 280117 Giuseppe Piombino
  https://github.com/Girasolo/computational_intelligence
- 291871 Alessia Leclercq
  https://github.com/AlessiaLeclercq/ComputationalIntelligence_S291871
- 302294 Leonardo Tredese
  https://github.com/LeonardoTredese/s302294-computational-intelligence-2022-2023
- 292113 Filippo Cardano
  https://github.com/Frititati/CI_2022_292113

- Idea from the onemax problem:

  each flag correspond to one of the lists generated by the well known function problem

  Genome

```
for genome in [tuple([random.choice([1, 0]) for _ in range(len(ALL_LISTS))]) for _ in range(population_SIZE)]:
    population.append(Individual(genome,fitness_function(genome)))
```

REPRESENTATION

As in the onemax we have used a namedtuple

The fitness function returns a tuple (#coveredEl,-weight), which is assigned to the fitTuple

```
Individual= namedtuple('Individual',['genome','fitTuple'])

def fitness_function(genome: list):
    '''Returns a tuple (number_of_covered_elements, -weigth)'''
    #this fitness function allows the precence of not optimal solution. The hierarchy of them shold be
    # based at first of the #ofCoverdEl and then on the lighter
    num_covered_elements = set()
    weigth = 0
    for (i, list_) in enumerate(ALL_LISTS):
        if genome[i]:
            num_covered_elements.update(list_)
            weigth += len(list_)

    return len(num_covered_elements), -weigth #the reason why weight is negative is we want to minimize it
```

REPRESENTATION

```python
def GA(population, NUM_GENERATIONS, OFFSPRING_SIZE, mutation_rate,POPULATION_SIZE,cleaner_rate):
    start = time.time()                 #to take note of the computation time
    fittestIndividual=copy(population[0])
    for g in range(NUM_GENERATIONS):
        offspring = list()
        for i in range(OFFSPRING_SIZE): #generation of the offSprings
            if random.random() < mutation_rate:
                p = tournament(population)
                if g<NUM_GENERATIONS/cleaner_rate:
                    o= vacuum_cleaner(p.genome)
                else:
                    o = mutation(p.genome)
                #o = mutation(p.genome)

            else:
                p1 = tournament(population)
                p2 = tournament(population)
                o = cross_over(p1.genome, p2.genome)
            offspring.append(Individual(o, fitness_function(o)))
        population += offspring  #offspring added to the population

        #population=deClonizator(population)
        population=list(dict.fromkeys(population))  #deletion of duplicates

        population = sorted(population, key=lambda i: i.fitTuple, reverse=True)[:POPULATION_SIZE]  #cut the population
        if(population[0].fitTuple>fittestIndividual.fitTuple):  #copy the fittest individual
            fittestIndividual=copy(population[0])
```

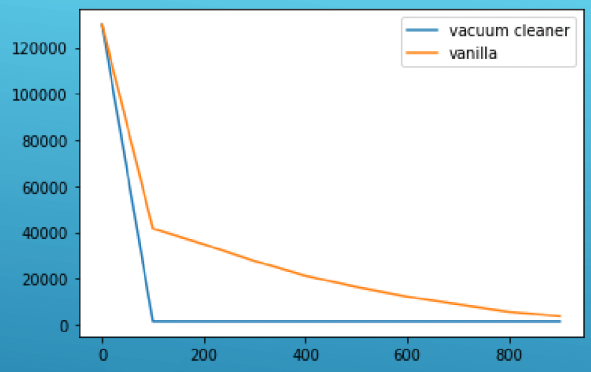
IMPROVEMENTS

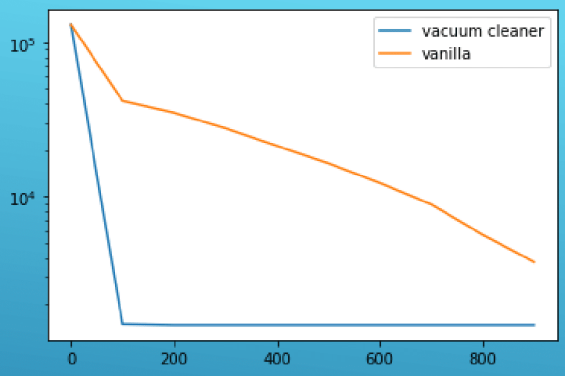
```python
def cross_over(g1, g2):
    cut = random.randint(0, len(ALL_LISTS)-1)
    choice=random.randint(0,2)
    if choice==0:
        return g1[:cut] + g2[cut:]
    else:
        return g2[:cut] + g1[cut:]
```

CROSS_OVER

```python
def vacuum_cleaner(g):
    '''takes some weight off by switching off a list'''
    temp_g=g
    for _ in range (random.randint(0,len(ALL_LISTS))):
        point=random.randint(0, len(ALL_LISTS) - 1)
        if temp_g[point] == 1:
            temp_g= temp_g[:point] + (0,) + temp_g[point + 1 :]
    return temp_g
```

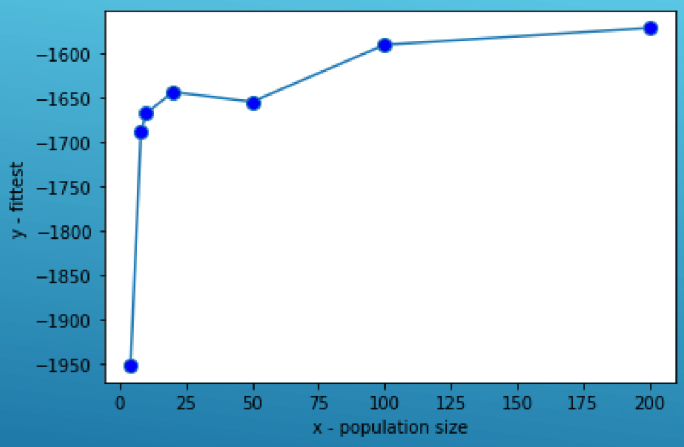
VACUUM CLEANER






```
[-129941, -1480, -1460, -1460, -1460, -1460, -1460, -1460, -1460, -1460]
[-129941, -41641, -34870, -27636, -21250, -16377, -12214, -8871, -5626, -3747]
```

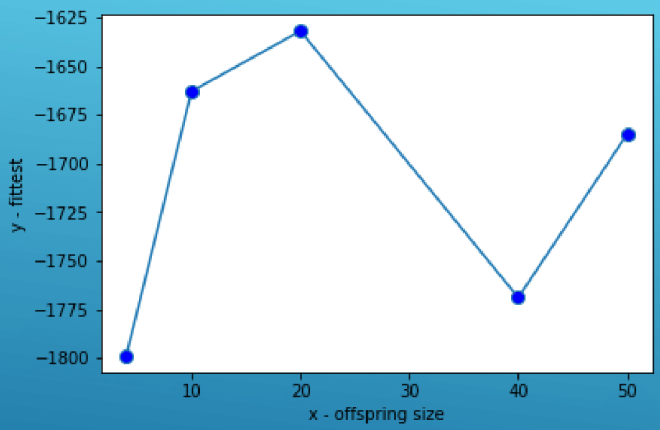
VACUUM CLEANER




```
population_SIZE=[4,8,10,20,50,100,200]
PROBLEM_SIZE=500
OFFSPRING_SIZE=20
mutation_rate=0.3
cleaner_rate=20
NUM_GENERATIONS=1000
```

```
time: 4.03 fittest: (500, -1951)
time: 4.02 fittest: (500, -1688)
time: 3.94 fittest: (500, -1668)
time: 4.08 fittest: (500, -1644)
time: 4.55 fittest: (500, -1655)
time: 5.26 fittest: (500, -1591)
time: 7.01 fittest: (500, -1572)
```

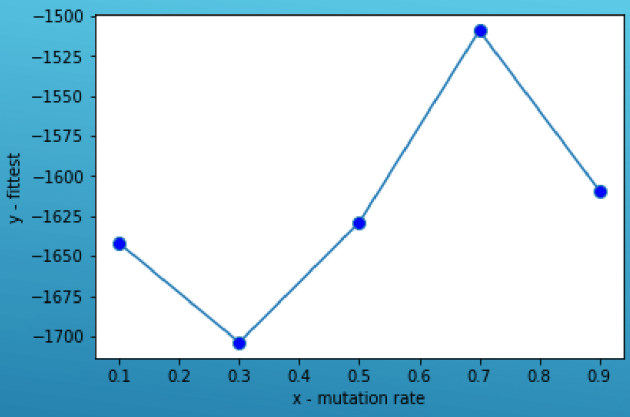
PARAMETER STUDY:POPULATION SIZE




```
population_SIZE=20
PROBLEM_SIZE=500
OFFSPRING_SIZE=[4,10,20,40,50]
mutation_rate=0.3
cleaner_rate=20
NUM_GENERATIONS=1000
```

```
time: 1.41 fittest: (500, -1799)
time: 2.33 fittest: (500, -1663)
time: 4.54 fittest: (500, -1632)
time: 9.63 fittest: (500, -1769)
time: 10.69 fittest: (500, -1685)
```

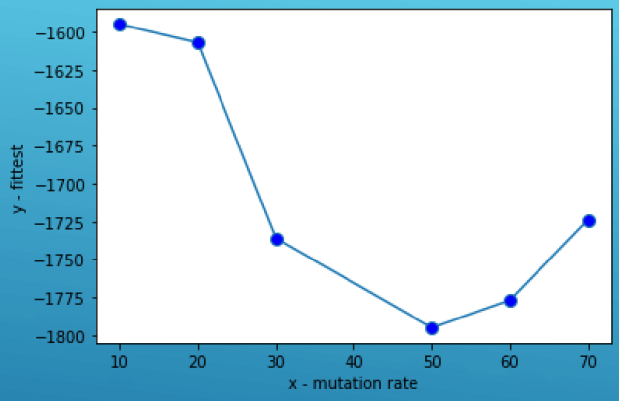
PARAMETER STUDY: OFFSPRING SIZE




```
population_SIZE=20
PROBLEM_SIZE=500
OFFSPRING_SIZE=20
mutation_rate=[0.1,0.3,0.5,0.7,0.9]
cleaner_rate=20
NUM_GENERATIONS=1000
```

```
time: 4.65 fittest: (500, -1642)
time: 4.40 fittest: (500, -1704)
time: 4.67 fittest: (500, -1629)
time: 4.99 fittest: (500, -1509)
time: 5.12 fittest: (500, -1609)
```

PARAMETER STUDY: MUTATION RATE




```
population_SIZE=20
PROBLEM_SIZE=500
OFFSPRING_SIZE=20
mutation_rate=0.7
cleaner_rate=[10,20,30,50,60,70]
NUM_GENERATIONS=1000
```

```
time: 6.10 fittest: (500, -1595)
time: 5.01 fittest: (500, -1607)
time: 4.94 fittest: (500, -1736)
time: 4.39 fittest: (500, -1795)
time: 4.31 fittest: (500, -1777)
time: 4.27 fittest: (500, -1724)
```

PARAMETER STUDY: CLEANER RATE

```
population_SIZE=20
PROBLEM_SIZE=[5,10,20,100,500,1000,2000]
PROBLEM_SIZE_=[5000,10000]
OFFSPRING_SIZE=20
mutation_rate=0.7
cleaner_rate=10
NUM_GENERATIONS=1000
```

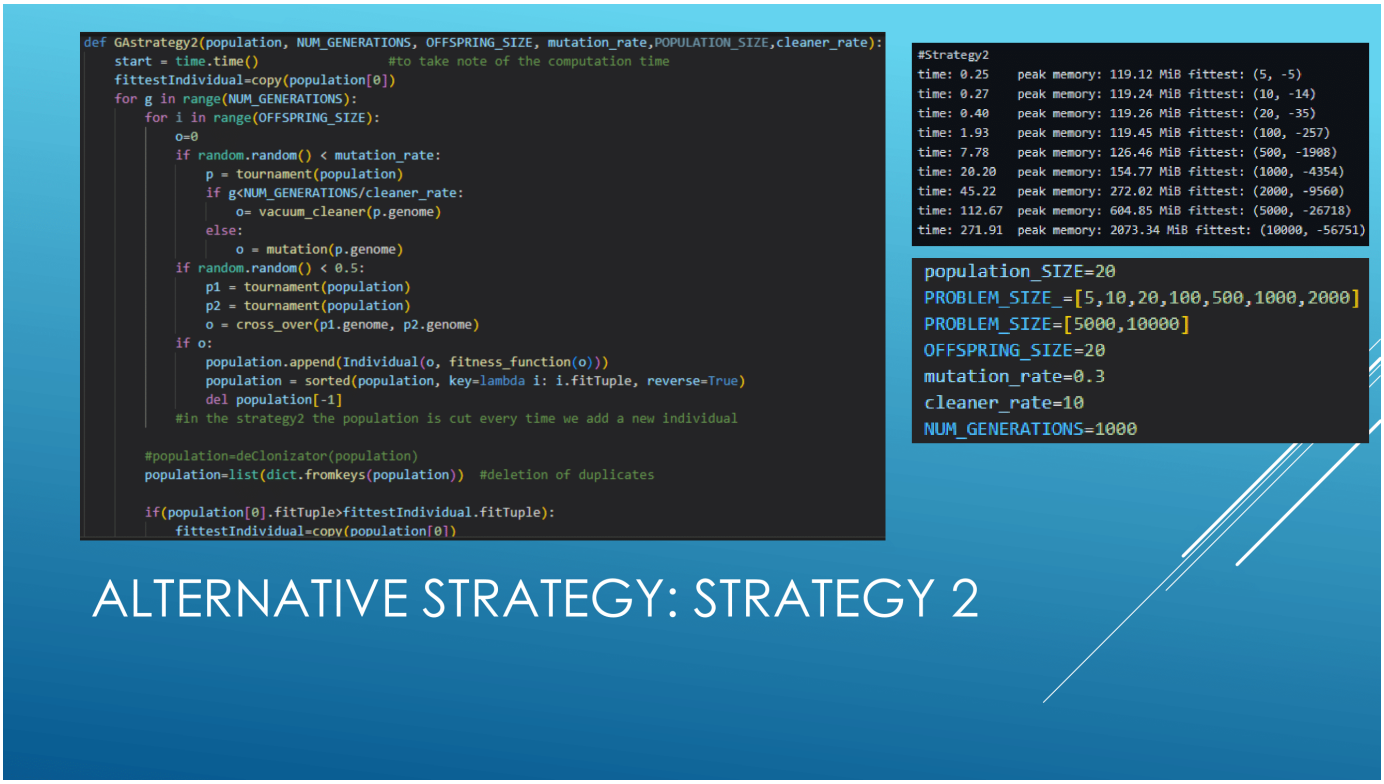
```
time: 112.67 fittest: (5000, -26718)
peak memory: 604.85 MiB, increment: 2.73 MiB
time: 271.91 fittest: (10000, -56751)
peak memory: 2073.34 MiB, increment: 3.61 MiB
```

FINAL TRIAL

```
time: 0.26 fittest: (5, -5)
peak memory: 159.83 MiB, increment: -2.07 MiB
time: 0.22 fittest: (10, -11)
peak memory: 159.84 MiB, increment: 0.01 MiB
time: 0.24 fittest: (20, -24)
peak memory: 159.85 MiB, increment: 0.00 MiB
time: 1.14 fittest: (100, -198)
peak memory: 160.15 MiB, increment: 0.25 MiB
time: 5.27 fittest: (500, -1595)
peak memory: 161.92 MiB, increment: 1.06 MiB
time: 12.28 fittest: (1000, -3810)
peak memory: 195.11 MiB, increment: 1.64 MiB
time: 29.81 fittest: (2000, -8789)
peak memory: 309.75 MiB, increment: 4.27 MiB
```

- Use of binary numbers instead of an array of ones and zeros
- …

FURTHER POSSIBLE OPTIMIZATIONS



Review of lab2

REVIEW LAB2 of Andred00

https://github.com/Andred00/computationalintelligence2022-s303339

REVIEW by Giuseppe Piombino

- In order to return a more complete result it is better to provide also the numer of call to the fitness function. In your case it is a fixed value, in some other cases it could vary. For example if you store the result of the fitness function in a database, pulling the result in case of a duplicate would avoid the call of the function.
- Another way to compute the efficience of the code is measuring the memory usage. It is very simple to use: you just have to install the package memory_profiler, import it and then in correspondence to the function you want to measure use the command %memit; if you would like to see an example take a look to my code (in the first codeSection and in the last one you can find how to install, import and finally how to use it)
- To let the code be more adaptable to perform a study on the parameters maybe it is better to use variables instead of fixed numbers, so that it becomes easy to change them

further optimization

- retrieve the already computed fitness function by a database
- divide the mutation in flipToZero and flipToOne, so that you can decide when it is better to use one or the other. For example at the beginning it's more usefull to set a lot of bit of the genome to zero, since it is more probable that it is filled with useless list

Code

```
{
Copyright (c) 2022 Giovanni Squillero <squillero@polito.it>
https://github.com/squillero/computational-intelligence
Free for personal or classroom use; see LICENSE.md for details.

Lab 2 : Set Covering using GA
Task
Given a number
 and some lists of integers
, determine, if possible,
 such that each number between
 and
```

```
 appears in at least one list

and that the total numbers of elements in all
 is minimum.

Try to use a GA approach.

import logging
from collections import namedtuple
import random
POPULATION_SIZE = 1500
OFFSPRING_SIZE = 1000

NUM_GENERATIONS = 200
def distinct(list):
   result = []
   for l in list:
      if l not in result:
         result.append(l)
   return result
Individual = namedtuple("Individual", ["genome", "fitness"])

def coverage(genome, lists):                                    # How many numbers are covered
    values = set()
    for i in range(len(genome)):
        if genome[i] == 1:
            values |= set(lists[i])
    return len(values)

def weight(genome, lists):                                      # Weight of the solution
    return sum(genome[i]*len(lists[i]) for i in range(len(lists)))

def compute_fitness(genome, lists):
    return (coverage(genome, lists), -weight(genome, lists))

def tournament(population, tournament_size=2):
    return max(random.choices(population, k=tournament_size), key=lambda i: i.fitness)

def cross_over(g1, g2):
    cut = random.randint(0, len(g1))
    return g1[:cut] + g2[cut:]

def mutation(g):
    point = random.randint(0, len(g) - 1)
    return g[:point] + (1 - g[point],) + g[point + 1 :]
def problem(N, seed=None):
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N // 5, N // 2))))
        for n in range(random.randint(N, N * 5))
    ]
Evolution
def my_genetic_algorithm(N, lists, population):
    for generation in range(NUM_GENERATIONS):
        offspring = list()
        for i in range(OFFSPRING_SIZE):
            m_rate = 0.3
            if random.random() < m_rate:                # MUTATION
                p = tournament(population)                  # promising genome
                o = mutation(p.genome)                      # promising genome mutated
            else:                                       # CROSSOVER
                p1 = tournament(population)                 # promising genome 1
                p2 = tournament(population)                 # promising genome 2
                o = cross_over(p1.genome, p2.genome)

            f = compute_fitness(o, lists)
            offspring.append(Individual(o, f))
```

```
        population += offspring
        population = sorted(population, key=lambda i: i.fitness, reverse=True)[:POPULATION_SIZE]

        best_so_far = population[0]
        if best_so_far.fitness[0] == -best_so_far.fitness[1] and best_so_far.fitness[0] == N:
            break

    print(f"#{N} : weight: {-best_so_far.fitness[1]}, bloat: {(-best_so_far.fitness[1]-N)/N*100},
#generations: {generation+1}")
logging.getLogger().setLevel(logging.INFO)

for N in [5, 10, 20, 100, 500, 1000]:
    population = list()
    lists = problem(N, 42)
    lists = distinct(lists)

    # try to create randomic population, this leads to worse results
    # for genome in [tuple([random.choice([1, 0]) for _ in range(len(lists))]) for _ in
range(POPULATION_SIZE)]:
      #  population.append(Individual(genome, compute_fitness(genome, lists)))
    for genome in [tuple([0 for _ in range(len(lists))]) for _ in range(POPULATION_SIZE)]:
        population.append(Individual(genome, compute_fitness(genome, lists)))

    my_genetic_algorithm(N, lists, population)
    print("------------")
#5 : weight: 5, bloat: 0.0, #generations: 3
------------
#10 : weight: 10, bloat: 0.0, #generations: 7
------------
#20 : weight: 24, bloat: 20.0, #generations: 200
------------
#100 : weight: 202, bloat: 102.0, #generations: 200
------------
#500 : weight: 1610, bloat: 222.00000000000003, #generations: 200
------------
#1000 : weight: 3600, bloat: 260.0, #generations: 200
------------

}
```

Review lab3

REVIEW LAB3 of marcopra

https://github.com/marcopra/computational_intellligence_22_23_294815

REVIEW LAB3 by Giuseppe Piombino

Advices

- to improve the performances of the agent it's possible to let him learn also by the moves of the opponent by storing them in a parallel state history array. To further improve the performances can be usefull switch the starting player each match.
- an improvement in minmax could be saving the already visited state with the cache method, as shown by the prof in the previous lab.

Doubts

- In evolvable_based_on_fixed_rules I don't understand why and in which case elements should be zero and if so, why you should set the variable to one.
- in the quality array I have doubt about the use of the state as only index instead of a cpuple (state, ply).

Code

```
{

Copyright (c) 2022 Giovanni Squillero <squillero@polito.it>
https://github.com/squillero/computational-intelligence
Free for personal or classroom use; see LICENSE.md for details.

Lab 3: Policy Search
Task
Write agents able to play Nim, with an arbitrary number of rows and an upper bound
 on the number of objects that can be removed in a turn (a.k.a., subtraction game).

The player taking the last object wins.

Task3.1: An agent using fixed rules based on nim-sum (i.e., an expert system)
Task3.2: An agent using evolved rules
Task3.3: An agent using minmax
Task3.4: An agent using reinforcement learning
Instructions
Create the directory lab3 inside the course repo
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.
Deadline

T.b.d.

import logging
import random
import numpy as np
import functools
from typing import Callable
from itertools import accumulate
from copy import deepcopy
from operator import xor
from collections import namedtuple
import matplotlib.pyplot as plt
from tqdm import tqdm
import os
import pickle


TUNING = False
CODE_FOR_TABLE = True
RL_TRAINING = False
random.seed(42)
Nimply = namedtuple("Nimply", "row, num_objects")
Move = namedtuple("Move", "row num_objects fitness")
NIM Game
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 __str__(self):
        return f"{self._rows}"

    def nimming(self, row: int, num_objects: int) -> None:
        assert self._rows[row] >= num_objects
        assert self._k is None or num_objects <= self._k
        assert num_objects > 0, f"You have to pick at least one"
        self._rows[row] -= num_objects
        if sum(self._rows) == 0:
            logging.debug("Yeuch")

    def nimming2(self, ply: Nimply) -> None:
        row, num_objects = ply
```

```
        assert self._rows[row] >= num_objects
        assert self._k is None or num_objects <= self._k
        assert num_objects > 0, f"You have to pick at least one"
        self._rows[row] -= num_objects

    @property
    def rows(self):
        return self._rows

    @property
    def k(self) -> int:
        return self._k
def nim_sum(rows: list) -> int:
    # List XOR
    # Using reduce() + lambda + "^" operator
    res = functools.reduce(lambda x, y: x ^ y, rows)
    return res
def tournament(population, tournament_size=2):
    return min(random.choices(population, k=tournament_size), key=lambda i: i.fitness)

def mutation(p: Move, nim: Nim):
    if nim.k is None:
        elements = random.randrange(1, nim.rows[p.row] + 1)
        temp_rows = nim.rows.copy()
        temp_rows[p.row] -=elements
        offspring = Move(p.row, elements , nim_sum(temp_rows))
    else:
        elements = min(nim.k, random.randrange(1, nim.rows[p.row] + 1))
        temp_rows = nim.rows.copy()
        temp_rows[p.row] -=elements
        offspring = Move(p.row, elements , nim_sum(temp_rows))

    return offspring

def cross_over(p1: Move, p2: Move, nim: Nim):

    n_random = random.randint(0, 1)

    if n_random == 0:

        temp_rows = nim.rows.copy()
        temp_rows[p1.row] -= p2.num_objects

        if temp_rows[p1.row] < 0:
            return None

        offspring = Move(p1.row, p2.num_objects , nim_sum(temp_rows))
    else:
        temp_rows = nim.rows.copy()
        temp_rows[p2.row] -= p1.num_objects

        if temp_rows[p2.row] < 0:
            return None

        offspring = Move(p2.row, p1.num_objects , nim_sum(temp_rows))

    if nim.k is not None and offspring.num_objects > nim.k:
        return None

    return offspring
Creating the NIM Table
N_ROWS = 3
GAMEOVER = [0 for _ in range(N_ROWS)]
POPULATION_SIZE = 10
OFFSPRING_SIZE = 10
N_GENERATIONS = 20
def cook_status(state: Nim) -> dict:
    cooked = dict()
```

```
    cooked["possible_moves"] = [
        (r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1) if state.k is None or o <=
state.k
    ]
    cooked["active_rows_number"] = sum(o > 0 for o in state.rows)
    cooked["shortest_row"] = min((x for x in enumerate(state.rows) if x[1] > 0), key=lambda y: y[1])[0]
    cooked["longest_row"] = max((x for x in enumerate(state.rows)), key=lambda y: y[1])[0]
    cooked["nim_sum"] = nim_sum(state.rows)

    brute_force = list()
    for m in cooked["possible_moves"]:
        tmp = deepcopy(state)
        tmp.nimming2(m)
        brute_force.append((m, nim_sum(tmp.rows)))
    cooked["brute_force"] = brute_force

    possible_new_states = list()
    for m in cooked["possible_moves"]:
            tmp = deepcopy(state)
            tmp.nimming2(m)
            # (state, move to reach the state)
            possible_new_states.append((tmp, m))
    cooked["possible_new_states"] = possible_new_states

    return cooked
Player:
Strategies:
Task 3.1 (Expert System) : Best Possibile Strategy implemented in best_strategy and nominated as best
(see below). To understand the algorithm of the winning strategy, look at Nim!
Task 3.1 Bis (Expert System) : Fixed Rules I would play implemented in evolvable_based_on_fixed_rules but
using
 and
 and nominated as evolvable.
Task 3.2 (Evolvable Strategies): Evolvable Strategies based on GA implemented in evolvable_based_on_GA
and nominated as ga.
Task 3.2 bis (Evolvable Strategies): Evolvable Strategies based on Fixed Rules implemented in
evolvable_based_on_fixed_rules but using
 and
 (best parameters found when playing against a pure random opponent and using a random
) and nominated as evolvable.
Task 3.3: Minimax Strategy implemented in min_max_best_move and nominated as min_max, code based on
MinMax.
Task 3.4: RL agent implemented in rl_agent and nominated as rl.
Other strategies:
Best strategy By Prof. Squillero implemented in best_strategy_by_prof and nominated as best_prof.
Evolvable strategy By Prof. Squillero implemented in evolvable_by_prof and nominated as evolvable_prof.
To use this strategies, use their nomination (e.g. best, evolvable, ... ) in the evaulate function below.

class Player:
    def __init__(self, strategy = 'best') -> None:
        # Two parts for the best strategy:
        # 0 -> before all rows have one element
        # 1 -> after all rows have one element
        self._best_strategy = 0

        assert strategy in ['best', 'best_prof', 'pure_random', 'ga', 'evolvable', 'evolvable_prof',
'evolvable_tuned', 'min_max', 'rl'], f"Strategy non-available"
        self._strategy = strategy
        if strategy == 'rl':
            self.agent = Agent(pretrained=True)

    def moves(self, Nim, alpha = 0.5, beta = 0.5):
        if self._strategy == 'best':
            return self.best_strategy(Nim)
        elif self._strategy == 'best_prof':
            return self.best_strategy_by_prof(Nim)
        elif self._strategy == 'pure_random':
            return self.pure_random(Nim)
```

```
        elif self._strategy == 'ga':
            return self.evolvable_based_on_GA(Nim)
        elif self._strategy == 'evolvable':
            return self.evolvable_based_on_fixed_rules(Nim, cook_status(Nim), alpha, beta)
        elif self._strategy == 'evolvable_tuned':
            return self.evolvable_based_on_fixed_rules(Nim, cook_status(Nim), 0.4, 0.1)
        elif self._strategy == 'evolvable_prof':
            return self.evolvable_by_prof(Nim, alpha)
        elif self._strategy == 'min_max':
            return self.min_max_best_move(Nim)
        elif self._strategy == 'rl':
            return self.rl_agent(Nim)
        else:
            assert f"Can't use a strategy"
        return


    def pure_random(self, Nim):

        # The opponent choose randomly a non-empty row
        nonzeroind = np.nonzero(Nim.rows)[0]
        random_row = random.choice(nonzeroind)

        # The opponen choose to remove a random number of elements
        if Nim._k == None:
            random_elements = random.randint(1,Nim.rows[random_row])
        else:
            random_elements = random.randint(1,min(Nim._k,Nim.rows[random_row]))

        return Nimply(random_row, random_elements)


    def best_strategy(self, Nim):

        # If all the elements are equal or less then k, we can play the 'normal' nim game
        if Nim._k != None and all(v <= Nim._k for v in Nim.rows):
            temp_k = None
        else:
            temp_k = Nim._k

        if temp_k != None:

            # Try brute force:
            for ind, row in enumerate(Nim.rows):

                for el in range(1, min(row + 1, Nim._k + 1)):
                    # Reset temp_rows
                    temp_rows = Nim.rows.copy()

                    # See if nim_sum == 0
                    temp_rows[ind] -= el
                    if nim_sum(temp_rows) == 0:
                        # Update table
                        # Nim.nimming(ind, el)
                        return Nimply(ind, el)

            equal_grater_than_k_ind = [i for i,v in enumerate(Nim.rows) if v >= Nim._k + 1]

            random_row = random.choice(equal_grater_than_k_ind)
            elements = Nim.rows[random_row]%(Nim._k+1)

            if elements == 0:
                elements = 1

            return Nimply(random_row, elements)

        # If there is only one element greater to one, the agent picks a number of object to make
        # all the rows of the table equal to 1.
```

```
        # He can choose to remove all the objects or all the objects but one from the rows with n>1
        if sum(x >= 2 for x in Nim.rows) == 1:
            # Row with more than one element
            equal_grater_than_two_ind = [i for i,v in enumerate(Nim.rows) if v >= 2][0]

            # Change of strategy
            self._best_strategy = 1


            # To win, the remaing number of objects has to be even
            if (sum(x for x in Nim.rows) - Nim.rows[equal_grater_than_two_ind]) % 2 == 0 :

                return Nimply(equal_grater_than_two_ind, Nim.rows[equal_grater_than_two_ind])

            else:

                return Nimply(equal_grater_than_two_ind, Nim.rows[equal_grater_than_two_ind]-1)

        # Strategy before all rows have one element
        if self._best_strategy == 0:

            res = nim_sum(Nim.rows)

            for ind, row in enumerate(Nim.rows):

                if row == 0:
                    continue

                if row ^ res < row:

                    elements = row - (row ^ res)

                    return Nimply(ind, elements)

        # Strategy after all rows have one element
        else:

            nonzeroind = [i for i, e in enumerate(Nim.rows) if e != 0]
            random_row = random.choice(nonzeroind)

            return Nimply(random_row, 1)

        # Default move -> Random
        nonzeroind = [i for i, e in enumerate(Nim.rows) if e != 0]
        random_row = random.choice(nonzeroind)

        if Nim._k == None:
            random_elements = random.randrange(1,Nim.rows[random_row] + 1)
        else:
            random_elements = random.randrange(1,min(Nim._k,Nim.rows[random_row])+1)


        return Nimply(random_row, random_elements)

    def best_strategy_by_prof(self, state: Nim):
        data = cook_status(state)
        move  = next((bf for bf in data["brute_force"] if bf[1] == 0),
random.choice(data["brute_force"]))[0]
        return Nimply(move[0], move[1])

    def evolvable_by_prof(self, state: Nim, p = 0.5):
        data = cook_status(state)

        if random.random() < p:
            if state.k is not None:
                ply = Nimply(data["shortest_row"], min(state.k, random.randint(1,
state.rows[data["shortest_row"]])))
            else:
```

```
                ply = Nimply(data["shortest_row"], random.randint(1, state.rows[data["shortest_row"]]))

        else:
            if state.k is not None:
                ply = Nimply(data["longest_row"], min(state.k,random.randint(1,
state.rows[data["longest_row"]])))
            else:
                ply = Nimply(data["longest_row"], random.randint(1, state.rows[data["longest_row"]]))

        return ply

    def evolvable_based_on_fixed_rules(self, state: Nim, cook_status: dict, alpha: float = 0.5, beta:
float = 0.5):
        initial_numbers = sum([i*2 + 1 for i in range(N_ROWS)])
        actual_numbers = sum(state.rows)

        # Early game strategy
        if actual_numbers > alpha * initial_numbers:

            if cook_status['active_rows_number'] >= beta*N_ROWS:

                row = cook_status["longest_row"]
                if state.k is not None:
                    elements = min(state.k, state.rows[row])

                else:
                    elements = state.rows[row]

                # state.nimming(row, elements)
                return Nimply(row, elements)

        row = cook_status["longest_row"]

        if cook_status["active_rows_number"]%2 == 0 and state.rows[row]!=1 :
            if state.k is not None:
                #Leave at least one element
                elements = min( (state.k - 1), state.rows[row] - 1)

            else:
                #Leave at least one element
                elements = state.rows[row] - 1

        else:
            if state.k is not None:
                #Try to remove the maximum number of elements
                elements = min(state.k, state.rows[row])
            else:
                #Try to remove the maximum number of elements
                elements = state.rows[row]

        if elements == 0:
            elements = 1
        # state.nimming(row, elements)
        return Nimply(row, elements)

    def evolvable_based_on_GA(self, state: Nim):

        # Population = possible moves
        population = []
        for _ in range(POPULATION_SIZE):
            # temp_rows needed to evaluate nim_sum = fitness (low nim_sum is better!)
            temp_rows = state.rows.copy()

            # Choosing a random_row
            nonzeroind = [i for i, e in enumerate(state.rows) if e != 0]
            random_row = random.choice(nonzeroind)

            #Choosing a random_move
```

```
            if state.k is None:
                random_elements = random.randrange(1, state.rows[random_row] + 1)

            else:
                random_elements =  min(random.randrange(1, state.rows[random_row] + 1), state.k)

            temp_rows[random_row] -= random_elements
            fitness = nim_sum(temp_rows)
            pop = Move(random_row, random_elements, fitness)
            population.append(pop)


        for g in range(N_GENERATIONS):
            offspring = list()
            for i in range(OFFSPRING_SIZE):

                if random.random() < 0.5:
                    # Selection of parents
                    p = tournament(population.copy())

                    # Offspring generation
                    o = mutation(p, state)

                else:

                    p1 = tournament(population)
                    p2 = tournament(population)
                    o = cross_over(p1, p2, state)

                # Check if cross-over returned a valid solution.
                # In this code, only valid solutions has been considered.
                # Possible Improvement: Acceptance with penalties of non-valid solutions
                if o == None:
                    continue

                offspring.append(o)

            # Adding new Offspings generated to Population list
            population+=offspring

            # Sorting the Population, according to their fitness and selecting the firsts n_elements =
POPULATION_SIZE
            population = sorted(population, key=lambda i: i.fitness, reverse=False)[:POPULATION_SIZE]
            logging.debug(f"actual best {population[0]}")

        return Nimply(population[0].row, population[0].num_objects)

    # Internal function
    def _minimax(self, state: Nim, maximizing: int = True, alpha=-1, beta=1, depth = 0):

        if depth == 100:
            return False
        # Check the result of the previous move
        if all(item == 0 for item in state.rows):
            # The player who made the previous move has already won
            return -1 if maximizing else 1

        cooked = cook_status(state)

        scores = []
        for new_state, move in cooked["possible_new_states"]:
            score = self._minimax(new_state, maximizing = not maximizing, alpha = alpha, beta = beta,
depth= depth + 1)
            if score != False:
                scores.append(score)
            if maximizing:
                alpha = max(alpha, score)
            else:
```

```
                beta = min(beta, score)
            if beta <= alpha:
                break

        return (max if maximizing else min)(scores)

    def min_max_best_move(self, state: Nim):
        cooked = cook_status(state)
        ply = None
        for new_state, move in cooked["possible_new_states"]:

            score = self._minimax(new_state, maximizing = False)

            if score > 0:
                ply = Nimply(move[0], move[1])
                break

        if ply is None:
            logging.debug(" No winning moves :(")
            nonzeroind = [i for i, e in enumerate(state.rows) if e != 0]
            random_row = random.choice(nonzeroind)

            ply =  Nimply(random_row, 1)

        return ply

    def rl_agent(self, state: Nim):
        ply = self.agent.choose_action(state)
        return ply
Reinforcement Learning Agent
class Agent(object):
    def __init__(self, state: Nim = None, alpha=0.15, random_factor=0.2, pretrained = False):  # 80%
explore, 20% exploit
        self.state_history = []  # state, reward
        self.alpha = alpha
        self.G = {}

        if pretrained:
            with open('RL_agent.pkl', 'rb') as f:
                self.G = pickle.load(f)
            self.random_factor = 0
            self.k = True
        else:
            assert state is not None, f'Please insert the starting state to inizialize rewards, if you
want to train the agent. Otherwise, choose the `pretrained` option'
            self.G = {}
            self.init_reward(state)
            self.random_factor = random_factor


    def init_reward(self, state: Nim):
        cooked = cook_status(state)
        for state, move in cooked["possible_new_states"]:
            self.G[tuple(state.rows)] =  np.random.uniform(low=-1.0, high=1.0)

    def choose_action(self, state: Nim):

        maxG = -10e15
        next_move = None
        cooked = cook_status(state)

        randomN = np.random.random()
        if randomN < self.random_factor:
            # if random number below random factor, choose random action
            moves = [move for new_state, move in cooked["possible_new_states"]]
            next_move = random.choice(moves)

        else:
```

```
            # if exploiting, gather all possible actions and choose one with the highest G (reward)
            for new_state, move in cooked["possible_new_states"]:
                if tuple(new_state.rows) not in self.G:
                    self.G[tuple(new_state.rows)] = np.random.uniform(low=-1.0, high=1.0)

                if self.G[tuple(new_state.rows)] > maxG:

                    next_move = move
                    next_move = Nimply(next_move[0], next_move[1])
                    maxG = self.G[tuple(new_state.rows)]

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

    def update_state_history(self, state, reward):
        self.state_history.append((state, reward))

    def get_reward_from_state(self, state: Nim ):
        return self.G[tuple(state.rows)]

    def learn(self):
        prev = 0

        for curr, reward in reversed(self.state_history):
            if tuple(curr.rows) not in self.G:
                self.G[tuple(curr.rows)] =  np.random.uniform(low=-1.0, high=1.0)
            prev = reward + self.alpha * (prev)
            self.G[tuple(curr.rows)] += prev

        self.state_history = []
        with open('RL_agent.pkl', 'wb') as f:
            pickle.dump(self.G, f)
        # np.save(os.path.join(os.getcwd(), 'lab3/RL_agent.npy'), self.G)

        self.random_factor -= 10e-5  # decrease random factor each episode of play

Games
def evaluate(agent_strategy = 'best', opponent_strategy = 'pure_random', parameter_dict: dict = {"alpha":
None, "beta": None}, NUM_MATCHES = 10, random_board = True) -> float:

    won = 0
    start = 0

    for m in tqdm(range(NUM_MATCHES)):

        agent = Player(agent_strategy)
        opponent = Player(opponent_strategy)

        # 0 -> Agent's turn
        # 1 -> Opponent's turn
        turn = start

        # the first move is equally distributed within matches
        start = 1 - start

        if random_board:
            # Random board
            rows = random.randint(1, 10)
            K = random.randint(0, 2*rows)
            if K == 0:
                K = None
        else:
            rows = 3
            K = 3

        # Nim Table Creation
        nim = Nim(rows, K)

        game_over = [0 for _ in range(rows)]
```

```
        logging.debug(f"\n\n\n--------NEW GAME---------")

        # Game
        while nim.rows != game_over:

            if turn == 0:
                logging.debug(f" Actual turn: Agent")
            else:
                logging.debug(f" Actual turn: Opponent")

            logging.debug(f" \tTable before move: {nim} and Nim_sum: {nim_sum(nim._rows)}")

            if turn == 0:
                if agent_strategy == 'evolvable':
                    assert parameter_dict['alpha'] is not None, f"Please choose a value for alfa"
                    assert parameter_dict['beta'] is not None, f"Please choose a value for beta"
                    ply = agent.moves(nim, parameter_dict['alpha'], parameter_dict['beta'] )


                elif agent_strategy == 'evolvable_by_prof':
                    assert parameter_dict['alpha'] is not None, f"Please choose a value for alfa"
                    ply = agent.moves(nim, parameter_dict['alpha'])
                else:
                    ply = agent.moves(nim)

                logging.debug(f" \tAgent:   <Row: {ply.row}- Elements: {ply.num_objects}>")

            else:

                if opponent_strategy == 'evolvable':
                    assert parameter_dict['alpha_opp'] is not None, f"Please choose a value for alfa used
by the opponent -> 'alpha_opp'"
                    assert parameter_dict['beta_opp'] is not None, f"Please choose a value for beta used
by the opponent -> 'beta_opp'"
                    ply = opponent.moves(nim, parameter_dict['alpha_opp'], parameter_dict['beta_opp'] )

                elif opponent_strategy == 'evolvable_by_prof':
                    assert parameter_dict['alpha_opp'] is not None, f"Please choose a value for alfa used
by the opponent -> 'alpha_opp'"
                    ply = opponent.moves(nim, parameter_dict['alpha_opp'])

                else:
                    ply = opponent.moves(nim)

                logging.debug(f" \tOpponent:   <Row: {ply.row}- Elements: {ply.num_objects}>")
            if ply.num_objects == 0:
                print(f"turn = {turn} ")
            nim.nimming2(ply)
            logging.debug(f" \tTable after move: {nim} and Nim_sum: {nim_sum(nim._rows)}\n")


            turn = 1 - turn

        logging.debug(f"--------GAME OVER---------")
        # Game Over
        if turn == 1:
            won +=1
        else:
            logging.debug(f"Game Lost by the agent is the n°{m}")


    return won / NUM_MATCHES
Tuning of
 and
 for taks 3.2
logging.getLogger().setLevel(logging.INFO)
if TUNING:
```

```
    max_win_rate = 0

    # Value to test
    values = list(x/10 for x in range(1, 10, 1))

    parameter_dict= {}

    for alpha in values:
        for beta in values:

            parameter_dict["alpha"] = alpha
            parameter_dict["beta"] = beta

            act_won_rate = evaluate(agent_strategy='evolvable', opponent_strategy='pure_random',
parameter_dict = parameter_dict)

            if act_won_rate > max_win_rate:
                alpha_best = alpha
                beta_best = beta
                max_win_rate = act_won_rate

                logging.debug(f" Found a new configuration:\n\talpha = {alpha_best}\n\tbeta =
{beta_best}\n\twon rate = {max_win_rate}")


    logging.info(f"\nBest configuration:\n\talpha = {alpha_best}\n\tbeta = {beta_best}\n\twon rate =
{max_win_rate}")
Evaluation part
logging.getLogger().setLevel(logging.INFO)
parameter_dict= {}

# Insert here your own parameters
# Best values of alpha and beta against a pure random opponent are respectively 0.4 and 0.1 generally

parameter_dict["alpha"] = 0.4
parameter_dict["beta"] = 0.1
parameter_dict["alpha_opp"] = 0.99
parameter_dict["beta_opp"] = 0.1

# Evaluation Section:

# print(f"Agent Won: {evaluate()*100}% of the games")
# print(f"Agent Won: {evaluate(agent_strategy='best_prof')*100}% of the games")
# print(f"Agent Won: {evaluate(agent_strategy='best', opponent_strategy='best_prof')*100}% of the games")
# print(f"Agent Won: {evaluate(agent_strategy='ga', opponent_strategy='best_prof')*100}% of the games")
# print(f"Agent Won: {evaluate(agent_strategy='evolvable', opponent_strategy='evolvable', parameter_dict
= parameter_dict)*100}% of the games")
# print(f"Agent Won: {evaluate(agent_strategy='evolvable', opponent_strategy='pure_random',
parameter_dict = parameter_dict)*100}% of the games")
print(f"Agent Won: {evaluate(agent_strategy='rl', opponent_strategy='pure_random', parameter_dict =
parameter_dict)*100}% of the games")
100%|██████████| 10/10 [00:03<00:00,  2.87it/s]
Agent Won: 70.0% of the games
RL Agent Training
def trainingRL():

    # Nim Table Creation
    # This is the biggest table the RL agent can find and it is needed to inizialize the possible states
    nim = Nim(10, None)

    agent = Agent(nim, alpha=0.8, random_factor=0.5)

    win_history_to_plot = []
    indices = []

    parameter_dict= {}
    parameter_dict["alpha_opp"] = 0.5
    parameter_dict["beta_opp"] = 0.5
```

```
start = 0

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

for i in tqdm(range(5000)):

    turn = 1 - start
    start = turn

    opponent_strategy = 'best'
    opponent = Player(opponent_strategy)

    current_state = None
    step = 0

    # Training the agent using DR
    rows = random.randint(1, 10)
    K = random.randint(0, 2*rows)
    if K == 0:
        K = None
    # Nim Table Creation
    nim = Nim(rows, K)


    game_over = [0 for _ in range(rows)]

    while nim.rows != game_over:
        if turn == 0:
            logging.debug(f" Actual turn: Agent")
        else:
            logging.debug(f" Actual turn: Opponent")

        logging.debug(f" \tTable before move: {nim} ")
        step +=1
        if turn == 0:
            # Current state
            current_state = nim
            old_state =  nim
            # Choose an action (explore or exploit)
            ply = agent.choose_action(current_state)
            logging.debug(f" \tAgent:   <Row: {ply.row}- Elements: {ply.num_objects}>")
            nim.nimming2(ply)

            # New current state
            current_state = deepcopy(nim)

            # update the robot memory with state and reward
            if current_state.rows == GAMEOVER:
                agent.update_state_history(current_state, 1)

        else:

            ply = opponent.moves(nim, parameter_dict['alpha_opp'], parameter_dict['beta_opp'] )
            nim.nimming2(ply)

            if current_state is not None:
                # update the robot memory with state and reward
                if nim.rows == GAMEOVER and turn == 1:
                    agent.update_state_history(current_state, -1)
                else:
                    agent.update_state_history(current_state, 0)

        if step >= 1000:
            break

        logging.debug(f" \tTable after move: {nim} ")
        turn = 1 - turn
```

```
        agent.learn()  # robot should learn after every episode

        # if i%5001 == 0:
        #     # Game Over
        #     if turn == 1:
        #         # win_history_to_plot.append(0)
        #         # indices.append(i)
        #     else:
        #         # win_history_to_plot.append(1)
        #         # indices.append(i)

    # plt.plot(indices, win_history_to_plot, "b", linewidth=1)
    # plt.show()
    return
if RL_TRAINING:
    trainingRL()
Results For Table
if CODE_FOR_TABLE:
    strategies = ['pure_random', 'best', 'best_prof', 'evolvable', 'evolvable_tuned',
'ga','evolvable_prof', 'rl' ]
    parameter_dict= {}
    parameter_dict["alpha"] = 0.5
    parameter_dict["beta"] = 0.5
    parameter_dict["alpha_opp"] = 0.5
    parameter_dict["beta_opp"] = 0.5
    for agent in strategies:
        for opponent in strategies:
            print(f"Agent Strategy: {agent} Opponent Strategy: {opponent} -> Agent Won:
{evaluate(agent_strategy= agent, opponent_strategy=opponent, parameter_dict = parameter_dict,
NUM_MATCHES=100, random_board=False)*100}% of the games ")
            # print(f"{agent} - {opponent} - {evaluate(agent_strategy= agent, opponent_strategy=opponent,
parameter_dict = parameter_dict)*100}% of the games ")

100%|██████████| 100/100 [00:00<00:00, 3382.83it/s]
Agent Strategy: pure_random Opponent Strategy: pure_random -> Agent Won: 63.0% of the games
100%|██████████| 100/100 [00:00<00:00, 3169.96it/s]
Agent Strategy: pure_random Opponent Strategy: best -> Agent Won: 3.0% of the games
100%|██████████| 100/100 [00:00<00:00, 818.12it/s]
Agent Strategy: pure_random Opponent Strategy: best_prof -> Agent Won: 5.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1101.83it/s]
Agent Strategy: pure_random Opponent Strategy: evolvable -> Agent Won: 12.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1170.57it/s]
Agent Strategy: pure_random Opponent Strategy: evolvable_tuned -> Agent Won: 26.0% of the games
100%|██████████| 100/100 [00:01<00:00, 81.70it/s]
Agent Strategy: pure_random Opponent Strategy: ga -> Agent Won: 5.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1080.36it/s]
Agent Strategy: pure_random Opponent Strategy: evolvable_prof -> Agent Won: 52.0% of the games
100%|██████████| 100/100 [00:22<00:00,  4.46it/s]
Agent Strategy: pure_random Opponent Strategy: rl -> Agent Won: 5.0% of the games
100%|██████████| 100/100 [00:00<00:00, 4184.85it/s]
Agent Strategy: best Opponent Strategy: pure_random -> Agent Won: 95.0% of the games
100%|██████████| 100/100 [00:00<00:00, 5557.73it/s]
Agent Strategy: best Opponent Strategy: best -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1177.58it/s]
Agent Strategy: best Opponent Strategy: best_prof -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1364.96it/s]
Agent Strategy: best Opponent Strategy: evolvable -> Agent Won: 73.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1240.19it/s]
Agent Strategy: best Opponent Strategy: evolvable_tuned -> Agent Won: 82.0% of the games
100%|██████████| 100/100 [00:00<00:00, 112.27it/s]
Agent Strategy: best Opponent Strategy: ga -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1190.39it/s]
Agent Strategy: best Opponent Strategy: evolvable_prof -> Agent Won: 96.0% of the games
100%|██████████| 100/100 [00:19<00:00,  5.02it/s]
Agent Strategy: best Opponent Strategy: rl -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1019.74it/s]
Agent Strategy: best_prof Opponent Strategy: pure_random -> Agent Won: 96.0% of the games
```

```
100%|██████████| 100/100 [00:00<00:00, 1265.90it/s]
Agent Strategy: best_prof Opponent Strategy: best -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 688.34it/s]
Agent Strategy: best_prof Opponent Strategy: best_prof -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 811.68it/s]
Agent Strategy: best_prof Opponent Strategy: evolvable -> Agent Won: 67.0% of the games
100%|██████████| 100/100 [00:00<00:00, 800.44it/s]
Agent Strategy: best_prof Opponent Strategy: evolvable_tuned -> Agent Won: 76.0% of the games
100%|██████████| 100/100 [00:00<00:00, 106.17it/s]
Agent Strategy: best_prof Opponent Strategy: ga -> Agent Won: 51.0% of the games
100%|██████████| 100/100 [00:00<00:00, 670.88it/s]
Agent Strategy: best_prof Opponent Strategy: evolvable_prof -> Agent Won: 97.0% of the games
100%|██████████| 100/100 [00:19<00:00,  5.08it/s]
Agent Strategy: best_prof Opponent Strategy: rl -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1303.71it/s]
Agent Strategy: evolvable Opponent Strategy: pure_random -> Agent Won: 86.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1451.47it/s]
Agent Strategy: evolvable Opponent Strategy: best -> Agent Won: 30.0% of the games
100%|██████████| 100/100 [00:00<00:00, 816.01it/s]
Agent Strategy: evolvable Opponent Strategy: best_prof -> Agent Won: 34.0% of the games
100%|██████████| 100/100 [00:00<00:00, 810.33it/s]
Agent Strategy: evolvable Opponent Strategy: evolvable -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 851.61it/s]
Agent Strategy: evolvable Opponent Strategy: evolvable_tuned -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 105.99it/s]
Agent Strategy: evolvable Opponent Strategy: ga -> Agent Won: 1.0% of the games
100%|██████████| 100/100 [00:00<00:00, 773.11it/s]
Agent Strategy: evolvable Opponent Strategy: evolvable_prof -> Agent Won: 77.0% of the games
100%|██████████| 100/100 [00:20<00:00,  4.96it/s]
Agent Strategy: evolvable Opponent Strategy: rl -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1039.50it/s]
Agent Strategy: evolvable_tuned Opponent Strategy: pure_random -> Agent Won: 69.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1246.23it/s]
Agent Strategy: evolvable_tuned Opponent Strategy: best -> Agent Won: 18.0% of the games
100%|██████████| 100/100 [00:00<00:00, 730.67it/s]
Agent Strategy: evolvable_tuned Opponent Strategy: best_prof -> Agent Won: 28.000000000000004% of the
games
100%|██████████| 100/100 [00:00<00:00, 764.83it/s]
Agent Strategy: evolvable_tuned Opponent Strategy: evolvable -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 809.30it/s]
Agent Strategy: evolvable_tuned Opponent Strategy: evolvable_tuned -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 106.87it/s]
Agent Strategy: evolvable_tuned Opponent Strategy: ga -> Agent Won: 0.0% of the games
100%|██████████| 100/100 [00:00<00:00, 741.78it/s]
Agent Strategy: evolvable_tuned Opponent Strategy: evolvable_prof -> Agent Won: 70.0% of the games
100%|██████████| 100/100 [00:19<00:00,  5.14it/s]
Agent Strategy: evolvable_tuned Opponent Strategy: rl -> Agent Won: 0.0% of the games
100%|██████████| 100/100 [00:00<00:00, 101.47it/s]
Agent Strategy: ga Opponent Strategy: pure_random -> Agent Won: 91.0% of the games
100%|██████████| 100/100 [00:00<00:00, 113.42it/s]
Agent Strategy: ga Opponent Strategy: best -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 108.12it/s]
Agent Strategy: ga Opponent Strategy: best_prof -> Agent Won: 48.0% of the games
100%|██████████| 100/100 [00:00<00:00, 109.10it/s]
Agent Strategy: ga Opponent Strategy: evolvable -> Agent Won: 100.0% of the games
100%|██████████| 100/100 [00:00<00:00, 108.53it/s]
Agent Strategy: ga Opponent Strategy: evolvable_tuned -> Agent Won: 100.0% of the games
100%|██████████| 100/100 [00:01<00:00, 59.91it/s]
Agent Strategy: ga Opponent Strategy: ga -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:01<00:00, 91.80it/s]
Agent Strategy: ga Opponent Strategy: evolvable_prof -> Agent Won: 97.0% of the games
100%|██████████| 100/100 [00:20<00:00,  4.93it/s]
Agent Strategy: ga Opponent Strategy: rl -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1132.25it/s]
Agent Strategy: evolvable_prof Opponent Strategy: pure_random -> Agent Won: 38.0% of the games
100%|██████████| 100/100 [00:00<00:00, 1134.30it/s]
Agent Strategy: evolvable_prof Opponent Strategy: best -> Agent Won: 3.0% of the games
100%|██████████| 100/100 [00:00<00:00, 646.54it/s]
```

```
Agent Strategy: evolvable_prof Opponent Strategy: best_prof -> Agent Won: 1.0% of the games
100%|██████████| 100/100 [00:00<00:00, 696.22it/s]
Agent Strategy: evolvable_prof Opponent Strategy: evolvable -> Agent Won: 12.0% of the games
100%|██████████| 100/100 [00:00<00:00, 511.77it/s]
Agent Strategy: evolvable_prof Opponent Strategy: evolvable_tuned -> Agent Won: 28.000000000000004% of
the games
100%|██████████| 100/100 [00:01<00:00, 72.36it/s]
Agent Strategy: evolvable_prof Opponent Strategy: ga -> Agent Won: 2.0% of the games
100%|██████████| 100/100 [00:00<00:00, 546.67it/s]
Agent Strategy: evolvable_prof Opponent Strategy: evolvable_prof -> Agent Won: 48.0% of the games
100%|██████████| 100/100 [00:20<00:00,  4.89it/s]
Agent Strategy: evolvable_prof Opponent Strategy: rl -> Agent Won: 1.0% of the games
100%|██████████| 100/100 [00:19<00:00,  5.17it/s]
Agent Strategy: rl Opponent Strategy: pure_random -> Agent Won: 93.0% of the games
100%|██████████| 100/100 [00:22<00:00,  4.38it/s]
Agent Strategy: rl Opponent Strategy: best -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:22<00:00,  4.54it/s]
Agent Strategy: rl Opponent Strategy: best_prof -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:22<00:00,  4.54it/s]
Agent Strategy: rl Opponent Strategy: evolvable -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:21<00:00,  4.56it/s]
Agent Strategy: rl Opponent Strategy: evolvable_tuned -> Agent Won: 100.0% of the games
100%|██████████| 100/100 [00:22<00:00,  4.38it/s]
Agent Strategy: rl Opponent Strategy: ga -> Agent Won: 50.0% of the games
100%|██████████| 100/100 [00:22<00:00,  4.54it/s]
Agent Strategy: rl Opponent Strategy: evolvable_prof -> Agent Won: 97.0% of the games
100%|██████████| 100/100 [00:42<00:00,  2.35it/s]
Agent Strategy: rl Opponent Strategy: rl -> Agent Won: 50.0% of the games

}
```

Review of lab3

REVIEW LAB3 of FabioSofer

https://github.com/FabioSofer/Computational_Intelligence2022

REVIEW LAB3 by Giuseppe Piombino

Major

- In [lab3] make_evolutionary_strategy when you are using the min function, it works as constraints only for positive numbers. To fastly fix it it is possible to check the absolute value of the direction.
- In [lab3] minmax val is saved as -val in evaluations, but it is still 1 or 0 in the for cycle, so the condittion is never satisfied and you can not enter the if. ####Minor
- In [lab3] minmax instead of using the wrapper you could have returned che ply by putting [0] after the max function
- In [lab3-4] an alternative to the initial setting of the q matrix is to add a new state when encountered instead of at the beginning
- In [lab3-4] check the comments of the get_reward functions (dont match with the function)
- In [lab3-4] checking the number of step can be usefull to teach the agent to finish as soon as possible it's interesting, I'm going to try it for the exam project. However if the aim was avoind stalls in this case is useless because the game is zerosum
- to improve the performances of the agent it's possible to let him to learn also by the moves of the oppent by storing them in a parallel history. To further improve the performances can be usefull switching the starting player each match. However, to do so it's important to give a negative rewared to the lost matches in order to penalize the entire bad path. Indeed I'm not so sure that the strategy of considering a lost match as non influent (reward=0) is otpimal. This could also be the reason why you had problem with the rewards and had to increase the positive one up to 10

Code

lab3.ipynb

```
{
Copyright (c) 2022 Giovanni Squillero <squillero@polito.it>
https://github.com/squillero/computational-intelligence
```

```
Free for personal or classroom use; see LICENSE.md for details.

Lab 3: Policy Search
Task
Write agents able to play Nim, with an arbitrary number of rows and an upper bound
 on the number of objects that can be removed in a turn (a.k.a., subtraction game).

The player taking the last object wins.

Task3.1: An agent using fixed rules based on nim-sum (i.e., an expert system)
Task3.2: An agent using evolved rules
Task3.3: An agent using minmax
Task3.4: An agent using reinforcement learning
Instructions
Create the directory lab3 inside the course repo
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.
Deadlines (AoE)
Sunday, December 4th for Task3.1 and Task3.2
Sunday, December 11th for Task3.3 and Task3.4
Sunday, December 18th for all reviews
import logging
from collections import namedtuple
import random
from typing import Callable
from copy import deepcopy
from itertools import accumulate
from operator import xor
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) + ">"

    def __hash__(self) -> int:
        rowList=list(self._rows)
        rowList.sort()
        return hash(" ".join(str(_) for _ in self._rows))

    def __eq__(self, __o: object) -> bool:
        return (self.__hash__()==__o.__hash__())

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

    @property
    def k(self) -> int:
        return self._k

    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

Sample Strategies
def pure_random(state: Nim) -> Nimply:
```

```
    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])))
P1: Expert Player (same as Professor's)
def nim_sum(state: Nim) -> int:
    *_, result = accumulate(state.rows, xor)
    return result


def cook_status(state: Nim,nimSum=False) -> dict:
    cooked = dict()
    cooked["possible_moves"] = [
        (r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1) if state.k is None or o <=
state.k
    ]
    cooked["active_rows_number"] = sum(o > 0 for o in state.rows)
    cooked["shortest_row"] = min((x for x in enumerate(state.rows) if x[1] > 0), key=lambda y: y[1])[0]
    cooked["longest_row"] = max((x for x in enumerate(state.rows)), key=lambda y: y[1])[0]
    if nimSum:
        cooked["nim_sum"] = nim_sum(state)

        brute_force = list()
        for m in cooked["possible_moves"]:
            tmp = deepcopy(state)
            tmp.nimming(m)
            brute_force.append((m, nim_sum(tmp)))
        cooked["brute_force"] = brute_force
    return cooked
def optimal_startegy(state: Nim) -> Nimply:
    data = cook_status(state,True)
    return next((bf for bf in data["brute_force"] if bf[1] == 0), random.choice(data["brute_force"]))[0]
NUM_MATCHES = 100
NIM_SIZE = 20


def evaluate(strategy: Callable,opponent) -> float:
    opponent = (strategy, opponent)
    won = 0

    for m in range(NUM_MATCHES):
        nim = Nim(NIM_SIZE)
        player = 0
        while nim:
            ply = opponent[player](nim)
            nim.nimming(ply)
            player = 1 - player
        if player == 1:
            won += 1
    return won / NUM_MATCHES
P2 -> Evolutionary Alg
def make_evolutionary_strategy(genome: dict) -> Callable:
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)

        if data["active_rows_number"]==1:
            ply=Nimply(data["shortest_row"],state.rows[data["shortest_row"]])
            return ply

        if random.random() < genome["p"]:
            #Take Everything
            ply=Nimply(data["shortest_row"],state.rows[data["shortest_row"]])
        else:
            #Take Everything but 1
            rowChoice=random.choice([x[0] for x in enumerate(state.rows) if x[1]!=0])
```

```
            ply=Nimply(rowChoice,state.rows[rowChoice] if state.rows[rowChoice]==1 else
state.rows[rowChoice]-1)
        return ply

    return evolvable
NUM_GEN=100


def evolve(strat:Callable,opponent):
    genome={"p":0.5}
    lostMatchesCounter=0
    prevWR=evaluate(strat(genome),opponent)
    direction=+0.05
    for i in range(NUM_GEN):
        sign=(direction/abs(direction)) #current sign of direction + or -

        if lostMatchesCounter==0: #if you just made a mistake, double check
            genome["p"]+=direction
        newStrat=strat(genome)
        wr=evaluate(newStrat,opponent)
        if(prevWR<=wr):
            lostMatchesCounter=0
            direction=min(direction*1.2,0.1*sign) #we slowly gain confidence in our direction,
            # up to a maximum accelleration of .1
        else :
            lostMatchesCounter+=1
            direction=0.05*sign#reset step to 0.05
            if lostMatchesCounter>=2:
                direction=-direction #set to flipped sign
                lostMatchesCounter=0

        prevWR=wr
        print(f"Round {i+1}: WR:{wr} with P:{genome['p']}")
    return genome
Example match
logging.getLogger().setLevel(logging.DEBUG)
#Evolve him for the matchup
strategy = (make_evolutionary_strategy(
    evolve(make_evolutionary_strategy,pure_random)
),pure_random)
#play the "real" game
nim = Nim(7)
logging.debug(f"status: Initial board  -> {nim}")
player = 0
while nim:
    ply = strategy[player](nim)
    nim.nimming(ply)
    logging.debug(f"status: After player {player} -> {nim}")
    player = 1 - player
winner = 1 - player
logging.info(f"status: Player {winner} won!")
Round 1: WR:0.89 with P:0.55
Round 2: WR:0.9 with P:0.6100000000000001
Round 3: WR:0.93 with P:0.682
Round 4: WR:0.94 with P:0.7684000000000001
Round 5: WR:0.93 with P:0.8684000000000001
Round 6: WR:0.99 with P:0.8684000000000001
Round 7: WR:0.97 with P:0.9284000000000001
Round 8: WR:1.0 with P:0.9284000000000001
Round 9: WR:0.97 with P:0.9884000000000002
Round 10: WR:0.99 with P:0.9884000000000002
Round 11: WR:0.91 with P:1.0484000000000002
Round 12: WR:0.97 with P:1.0484000000000002
Round 13: WR:0.98 with P:1.1084000000000003
Round 14: WR:0.98 with P:1.1804000000000003
Round 15: WR:0.97 with P:1.2668000000000004
Round 16: WR:0.97 with P:1.2668000000000004
Round 17: WR:0.95 with P:1.3268000000000004
```

```
Round 18: WR:0.94 with P:1.3268000000000004
Round 19: WR:0.98 with P:1.2768000000000004
Round 20: WR:0.97 with P:1.1768000000000003
Round 21: WR:0.99 with P:1.1768000000000003
Round 22: WR:0.99 with P:1.0768000000000002
Round 23: WR:0.94 with P:0.9568000000000002
Round 24: WR:0.97 with P:0.9568000000000002
Round 25: WR:0.98 with P:0.8568000000000002
Round 26: WR:0.96 with P:0.7368000000000002
Round 27: WR:0.97 with P:0.7368000000000002
Round 28: WR:0.95 with P:0.6368000000000003
Round 29: WR:0.93 with P:0.6368000000000003
Round 30: WR:0.93 with P:0.6868000000000003
Round 31: WR:0.95 with P:0.7468000000000004
Round 32: WR:0.93 with P:0.8188000000000003
Round 33: WR:0.93 with P:0.8188000000000003
Round 34: WR:0.94 with P:0.8788000000000002
Round 35: WR:0.97 with P:0.9508000000000002
Round 36: WR:0.96 with P:1.0372000000000001
Round 37: WR:0.98 with P:1.0372000000000001
Round 38: WR:0.97 with P:1.0972000000000002
Round 39: WR:0.98 with P:1.0972000000000002
Round 40: WR:0.98 with P:1.1572000000000002
Round 41: WR:0.98 with P:1.2292000000000003
Round 42: WR:0.96 with P:1.3156000000000003
Round 43: WR:0.91 with P:1.3156000000000003
Round 44: WR:0.97 with P:1.2656000000000003
Round 45: WR:0.97 with P:1.1656000000000002
Round 46: WR:0.94 with P:1.0456000000000003
Round 47: WR:0.99 with P:1.0456000000000003
Round 48: WR:0.95 with P:0.9456000000000003
Round 49: WR:0.92 with P:0.9456000000000003
Round 50: WR:0.96 with P:0.9956000000000004
Round 51: WR:0.98 with P:1.0556000000000003
Round 52: WR:0.96 with P:1.1276000000000004
Round 53: WR:0.98 with P:1.1276000000000004
Round 54: WR:0.97 with P:1.1876000000000004
Round 55: WR:0.97 with P:1.1876000000000004
Round 56: WR:0.96 with P:1.2476000000000005
Round 57: WR:0.97 with P:1.2476000000000005
Round 58: WR:0.98 with P:1.3076000000000005
Round 59: WR:0.95 with P:1.3796000000000006
Round 60: WR:0.96 with P:1.3796000000000006
Round 61: WR:0.99 with P:1.4396000000000007
Round 62: WR:0.95 with P:1.5116000000000007
Round 63: WR:0.91 with P:1.5116000000000007
Round 64: WR:0.98 with P:1.4616000000000007
Round 65: WR:0.99 with P:1.3616000000000006
Round 66: WR:0.95 with P:1.2416000000000005
Round 67: WR:0.98 with P:1.2416000000000005
Round 68: WR:0.94 with P:1.1416000000000004
Round 69: WR:0.92 with P:1.1416000000000004
Round 70: WR:0.97 with P:1.1916000000000004
Round 71: WR:0.99 with P:1.2516000000000005
Round 72: WR:0.99 with P:1.3236000000000006
Round 73: WR:0.96 with P:1.4100000000000006
Round 74: WR:0.99 with P:1.4100000000000006
Round 75: WR:0.98 with P:1.4700000000000006
Round 76: WR:0.95 with P:1.4700000000000006
Round 77: WR:0.94 with P:1.4200000000000006
Round 78: WR:0.93 with P:1.4200000000000006
Round 79: WR:1.0 with P:1.4700000000000006
Round 80: WR:0.98 with P:1.5300000000000007
Round 81: WR:0.96 with P:1.5300000000000007
Round 82: WR:0.97 with P:1.4800000000000006
Round 83: WR:0.99 with P:1.3800000000000006
Round 84: WR:0.99 with P:1.2600000000000007
Round 85: WR:0.98 with P:1.1160000000000008
```

```
Round 86: WR:0.98 with P:1.1160000000000008
Round 87: WR:0.95 with P:1.0160000000000007
Round 88: WR:0.93 with P:1.0160000000000007
Round 89: WR:0.97 with P:1.0660000000000007
Round 90: WR:0.98 with P:1.1260000000000008
Round 91: WR:0.94 with P:1.1980000000000008
Round 92: WR:0.92 with P:1.1980000000000008
Round 93: WR:0.96 with P:1.1480000000000008
Round 94: WR:0.96 with P:1.0480000000000007
Round 95: WR:0.98 with P:0.9280000000000007
Round 96: WR:0.98 with P:0.7840000000000007
Round 97: WR:0.85 with P:0.6112000000000007
Round 98: WR:0.93 with P:0.6112000000000007
Round 99: WR:0.88 with P:0.5112000000000008
DEBUG:root:status: Initial board  -> <1 3 5 7 9 11 13>
DEBUG:root:status: After player 0 -> <0 3 5 7 9 11 13>
DEBUG:root:status: After player 1 -> <0 3 5 7 2 11 13>
DEBUG:root:status: After player 0 -> <0 3 5 7 0 11 13>
DEBUG:root:status: After player 1 -> <0 3 5 7 0 11 11>
DEBUG:root:status: After player 0 -> <0 3 1 7 0 11 11>
DEBUG:root:status: After player 1 -> <0 3 0 7 0 11 11>
DEBUG:root:status: After player 0 -> <0 1 0 7 0 11 11>
DEBUG:root:status: After player 1 -> <0 1 0 7 0 3 11>
DEBUG:root:status: After player 0 -> <0 0 0 7 0 3 11>
DEBUG:root:status: After player 1 -> <0 0 0 2 0 3 11>
DEBUG:root:status: After player 0 -> <0 0 0 0 0 3 11>
DEBUG:root:status: After player 1 -> <0 0 0 0 0 2 11>
DEBUG:root:status: After player 0 -> <0 0 0 0 0 0 11>
DEBUG:root:status: After player 1 -> <0 0 0 0 0 0 0>
INFO:root:status: Player 1 won!
Round 100: WR:0.88 with P:0.5112000000000008
P3 -> MinMax Approach
def get_val(board:Nim):
    #if there is only one row left you have won
    if sum(o > 0 for o in board.rows)==1:
        return 1
    else:
        return 0

boardCache=dict()

#wrapper for the recursive minmax_rec, used to discard val and only return ply
def minmax(board:Nim):
    return minmax_rec(board)[0]

def minmax_rec(board:Nim):
    #check if the state is already evaluated to avoid recomputation
    if board in boardCache.keys() :
        return boardCache[board]
    val = get_val(board)
    possible = [(r, o) for r, c in enumerate(board.rows) for o in range(1,c+1)]
    #we sort the possible moves in decreasing order based on the number of elements removed
    possible.sort(reverse=True,key=lambda k: k[1])
    if val != 0:
        #if someone has won return the move and its val
        return possible[0],val
    evaluations = list()
    for ply in possible:
        newBoard=deepcopy(board) #return new board
        newBoard.nimming(ply)
        _, val = minmax_rec(newBoard)
        evaluations.append((ply, -val))
        if(val==-1):
            #as we are returning only the max we stop as soon as we get -1 (because it is saved as -val)
            break
    #we cache the evaluation of the current state
    boardCache[board]=max(evaluations, key=lambda k: k[1])
    return max(evaluations, key=lambda k: k[1])
```

```
Example match
logging.getLogger().setLevel(logging.DEBUG)

NUM_ROWS = 6

strategy = ()
nim = Nim(NUM_ROWS)

#We check which player should start in order not to lose automatically against the optimal strategy
minmax_player = int(nim_sum(nim) == 0)
logging.debug(f"We are player {minmax_player}")
if minmax_player == 1:
    strategy = (optimal_startegy,minmax)
else:
    strategy = (minmax,optimal_startegy)

logging.debug(f"status: Initial board  -> {nim}")
player = 0
while nim:
    ply = strategy[player](nim)
    nim.nimming(ply)
    logging.debug(f"status: After player {player} -> {nim}")
    player = 1 - player
winner = 1 - player

logging.info(f"status: Player {winner} won!")
DEBUG:root:We are player 0
DEBUG:root:status: Initial board  -> <1 3 5 7 9 11>
DEBUG:root:status: After player 0 -> <1 1 5 7 9 11>
DEBUG:root:status: After player 1 -> <1 1 5 7 9 8>
DEBUG:root:status: After player 0 -> <1 1 5 4 9 8>
DEBUG:root:status: After player 1 -> <1 1 5 4 5 8>
DEBUG:root:status: After player 0 -> <1 1 5 4 5 4>
DEBUG:root:status: After player 1 -> <1 1 5 3 5 4>
DEBUG:root:status: After player 0 -> <1 1 2 3 5 4>
DEBUG:root:status: After player 1 -> <1 1 2 3 5 2>
DEBUG:root:status: After player 0 -> <1 1 2 3 3 2>
DEBUG:root:status: After player 1 -> <1 1 2 3 3 0>
DEBUG:root:status: After player 0 -> <1 1 0 3 3 0>
DEBUG:root:status: After player 1 -> <1 1 0 2 3 0>
DEBUG:root:status: After player 0 -> <0 1 0 2 3 0>
DEBUG:root:status: After player 1 -> <0 0 0 2 3 0>
DEBUG:root:status: After player 0 -> <0 0 0 2 2 0>
DEBUG:root:status: After player 1 -> <0 0 0 2 1 0>
DEBUG:root:status: After player 0 -> <0 0 0 1 1 0>
DEBUG:root:status: After player 1 -> <0 0 0 1 0 0>
DEBUG:root:status: After player 0 -> <0 0 0 0 0 0>
INFO:root:status: Player 0 won!

}
```

lab3-4.ipynb

```
{

Lab 3: Policy Search
Task
Write agents able to play Nim, with an arbitrary number of rows and an upper bound
 on the number of objects that can be removed in a turn (a.k.a., subtraction game).

The player taking the last object wins.

Task3.1: An agent using fixed rules based on nim-sum (i.e., an expert system)
Task3.2: An agent using evolved rules
Task3.3: An agent using minmax
Task3.4: An agent using reinforcement learning
```

```
Instructions
Create the directory lab3 inside the course repo
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.
Deadlines (AoE)
Sunday, December 4th for Task3.1 and Task3.2
Sunday, December 11th for Task3.3 and Task3.4
Sunday, December 18th for all reviews
import logging
from collections import namedtuple
import random
from typing import Callable
from copy import deepcopy
from itertools import accumulate, product
from operator import xor
The Nim and Agent 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) + ">"

    def __hash__(self) -> int:
        rowList=list(self._rows)
        rowList.sort()
        return hash(" ".join(str(_) for _ in self._rows))

    def __eq__(self, __o: object) -> bool:
        return (self.__hash__()==__o.__hash__())

    def assign_rows(self, rows):
        self._rows = list(rows)
        return self

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

    @property
    def k(self) -> int:
        return self._k

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

    def is_game_over(self):
        if sum(o > 0 for o in self._rows)==0:
            return True
        else:
            return False

    def get_reward(self):
        # if at end give 0 reward
        # if not at end give -1 reward
        #return -1 * int(not self.is_game_over())
        return int(self.is_game_over()) * 10
```

```
class Agent:
    def __init__(self, num_rows: int,  alpha=0.15, random_factor=0.2) -> None:
        self.G = {}
        self.alpha = alpha
        self.random_factor = random_factor
        self.state_history = []

        tmp = []
        for i in range(num_rows):
            tmp.append(range(0, i * 2 + 2))
        for i in list(product(*tmp)):
            n = Nim(num_rows)
            self.G[n.assign_rows(i)] = random.random()

    def choose_action(self, board: Nim):
        """ chooses action according to action policy """
        r = random.random()
        possible = [(r, o) for r, c in enumerate(board.rows) for o in range(1,c+1)]

        if r < self.random_factor: # for epsilon-greedy policy
            next_move = random.choice(possible)
        else:       # choose best possible action in this state
            evaluations = list()
            for ply in possible:
                newBoard=deepcopy(board) #return new board
                newBoard.nimming(ply)
                evaluations.append((ply, self.G[newBoard]))

            # we choose the action with the higher G
            next_move = max(evaluations, key=lambda k: k[1])[0]

        return next_move

    def update_state_history(self, state, reward):
        self.state_history.append((state, reward))

    def learn(self):
        target = 0

        for prev, reward in reversed(self.state_history):
            self.G[prev] = self.G[prev] + self.alpha * (target - self.G[prev])
            target += reward

        self.state_history = []

        self.random_factor -= 10e-5  # decrease random factor each episode of play

Sample Strategies
def pure_random(state: Nim) -> Nimply:
    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),0
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]))),0
def franchino(state: Nim) -> Nimply:
    """Pick always the one from the longest row"""
    return Nimply(max(enumerate(state.rows), key=lambda a: a[1])[0], 1),0
P1: Expert Player (same as Professor's)
def nim_sum(state: Nim) -> int:
    *_, result = accumulate(state.rows, xor)
    return result


def cook_status(state: Nim,nimSum=False) -> dict:
    cooked = dict()
    cooked["possible_moves"] = [
```

```
        (r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1) if state.k is None or o <=
state.k
    ]
    cooked["active_rows_number"] = sum(o > 0 for o in state.rows)
    cooked["shortest_row"] = min((x for x in enumerate(state.rows) if x[1] > 0), key=lambda y: y[1])[0]
    cooked["longest_row"] = max((x for x in enumerate(state.rows)), key=lambda y: y[1])[0]
    if nimSum:
        cooked["nim_sum"] = nim_sum(state)

        brute_force = list()
        for m in cooked["possible_moves"]:
            tmp = deepcopy(state)
            tmp.nimming(m)
            brute_force.append((m, nim_sum(tmp)))
        cooked["brute_force"] = brute_force
    return cooked
def optimal_startegy(state: Nim) -> Nimply:
    data = cook_status(state,True)
    return next((bf for bf in data["brute_force"] if bf[1] == 0), random.choice(data["brute_force"]))
[0],0
P4: Reinforcement Learning
logging.getLogger().setLevel(logging.DEBUG)


NUM_ROWS = 6
opponent = franchino

board = Nim(NUM_ROWS)
logging.debug(f"status: Initial board  -> {board}")

jack = Agent(NUM_ROWS, alpha=0.3, random_factor=0.1)
win = 0

for i in range(5000):
    steps = 0
    player = 0
    while not board.is_game_over():
        steps += 1
        if player == 0:
            # choose an action (explore or exploit)
            action = jack.choose_action(board)
            board.nimming(action)
            reward = board.get_reward()  # get the new reward
            # update the agent memory with state and reward
            jack.update_state_history(board, reward)
        else:
            ply = opponent(board)[0]
            board.nimming(ply)

        if steps > 100:
            # end the agent if it takes too long to find the goal
            break

        player = 1 - player

        #if i % 50 == 0:
        #    logging.debug(f"status: After player {player} -> {board}")

    winner = 1 - player
    if winner == 0:
        win += 1

    jack.learn()  # jack should learn after every episode

    if i % 50 == 0 and i > 0:
        win_ratio = win/50
        win = 0
        logging.info(f"Win ration {win_ratio} won!")
```

```
        #if we are keeping on losing reset the agent
        if win_ratio < 0.1 :
            jack = Agent(NUM_ROWS, alpha=0.3, random_factor=0.1)

    board = Nim(NUM_ROWS)

}
```

REVIEW RECEIVED

REVIEW BY SQUILLERO

MAJOR

- Is `h` really *admissible*? I don't think so. It should **never** overestimates the actual cost to the solution. You found the solution in 467,479, I found it in 14,095 with Dijkstra... but Dijkstra should be worse tha A* 😉

MINOR

- Use markdown in README if you use `.md` extension (Check out: GitHub Flavored Markdown Spec)
- If you need to comment out cells, use something like `if VAR_ENABLED:`
- Don't commit `__pycache__`, use `.gitignore`
- Why name a variable `frozenset_` if it actually a numpy ndarray?
- `is_valid` looks like an `in` operator, why not using the real `in` (and perhaps sets)

REVIEW BY Pietro Noto

Overview

The algorith(s) you used are A* and BFS search and you made a comparison between the outcomes of the two. The problem has been solved for N = 5, 10, 20 and you tried also N = 100. The code, provided by the prof, isn't very understable (not your fault) but I'd have appreciated some comments if I want to be picky.

Performance analysis

I ran the script with several values of N and got the following values: A*: N = 5 -> 5 elements in 61 visited nodes (optimal :)) N = 10 -> 10 elements in 3149 visited nodes (optimal :)) N = 20 -> 23 elements in 479,873 visited nodes (must be the best possible, I got the same :)) N >= 30 -> eternity and RAM maxed out (:( )

```
BFS: N = 5 -> 5 elements and 3,454 visited nodes ( optimal but much slower  :| )
     N = 10 -> 13 elements and 197,080 visited nodes ( not optimal and very slow :( )
     N = 20 -> 29 elements and 2,172,545 visited nodes ( not optimal and very slow :( )
     N >= 30 -> eternity and RAM maxed out ( :( )

It's clear that A* is a better choice, but I think it's not ideal for this problem because of the extreme
amount of memory needed, for N > 20. Anyway I appreciate you tried more than one solution!
```

What you can improve

I noticed the algorith uses a massive amount of RAM (maxing out my 16GB machine and even using virtual memory of hard drive) with N >= 30. I also had that issue that was caused by listing of all the combinations possible instead of leaving them as an optimized iterator. In your case I think it's caused by keeping in memory all the possibile states the algorithm generates! In the search function I found annoying the log of frontier node addition (who cares!), it is potentially capable of flooding the output, and it did! So I just commented that line. I also have preferred if you had made some sort of functions called (A* and BFS) instead of commenting some parts of the code in order to

switch between the former or the latter. Besides, given the amount of time needed to solve the problem (with N > 20), I would have preferred you to implement some sort of for loop with N values in order to run it and potentially go afk instead to continously check it and change the N value after a test is done.

Comparison with my solution

My solution uses an approach based on full exploration of the solution space and computes all the possible combinations, instead of using tree-search algorithims like you did. In terms of performance, I ran both tests and mine turned out to be better and capable of solving the problem till N = 40 in a few minutes, without maxing out system memory. I think for very small values of N, like 5, however, A* performed better. Honestly, I expected mine to be worse because it explores the whole solution space, but perhaps it is better because of the pruning.

Conclusions

I think yours isn't the fittest algorithm for this kind of problem because these algorithms are known to be computing time "friendly" (not that much) but memory unfriendly unless you have a workstation. For next assignment I recommend you to make the code more modular and put some more comments, I'm not judging you for the memory issues, I almost randomly fell into windows task manager and noticed it :)

REVIEW BY LEONARDO TREDESE

Hardcoded player

- You do not need function closures to create the benchmarking strategies, you could have used directly the name of the defined functions.

Evolved strategies

- You could have used an additional parameter p3 for the 2 params evolutive strategies to make the two inner ifs independent.
- `cooked["completion"] = sum(o for o in state.rows) / state.total_elements` should always be 1 because `total_elements` is defined as:

```
@property
    def total_elements(self) -> int:
        return sum(self._rows)
```

so using the `data["completion"]==1` is useless.

- I apreciate that you tried a population method as well, but `make_strategy_evol` is not defined so it crashes, nice that you did a tweak of the parameters, it would have been nice if you added some kind of simple crossover.
- In tournament selection is just a random the choice of an individual, you should look also at their fitness when selecting them

MinMax

- 👌
- min max with ab pruning does not beat optimal strategy like nomal minmax (which already does some pruning `if(val==-1 and player==0):`)

RL

- 👌
- Nice that you itialize `self._q` just in time
- Nice that the agent is trained to play first and last
- Nice that you give a negative reward to the opponent's mistakes

3 Lab3 - Solving NIM game with different strategies

3.1 Problem statement

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

1. An agent using *fixed rules based on nim-sum* (i.e., an expert system)
2. An agent using *evolved rules*
3. An agent using *min-max*
4. An agent using *reinforcement learning*

3.2 Nim game

Nim is a mathematical game of strategy in which two players take turns removing (or "nimming") objects from distinct heaps or piles. On each turn, a player must remove at least one object, and may remove any number of objects provided they all come from the same heap or pile. The player taking the last object wins [2]. The interface to run the game is presented in 17

```
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
        self._total_elements = num_rows*num_rows

    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)

    @property
    def k(self) -> int:
        return self._k

    def __hash__(self):
        return hash(bytes(self._rows))

    @property
    def total_elements(self) -> int:
        return sum(self._rows)

```

```
30      def nimming(self, ply: Nimply) -> None:
31          row, num_objects = ply
32          assert self._rows[row] >= num_objects
33          assert self._k is None or num_objects <= self._k
34          self._rows[row] -= num_objects
35
36      def get_state_and_reward(self):
37          return self.total_elements, self.give_reward()
38
39      def give_reward(self):
40          # if at end give 0 reward
41          # if not at end give -1 reward
42          return -1 * int(not self.total_elements == 0)
43
44      def possible_actions(self):
45          actions=[]
46          for r,c in enumerate(self.rows):
47              for o in range(1,c+1):
48                  if self.k is None or o<=self.k:
49                      actions.append((r,o))
50          return actions
51
52  # SUPPORT FUNCTIONS
53  def nim_sum(state: Nim) -> int:
54      *_, result = accumulate(state.rows, xor)
55      return result
56
57  def cook_status(state: Nim) -> dict:
58      cooked = dict()
59      cooked["possible_moves"] = [
60          (r, o) for r, c in enumerate(state.rows) for o in range(1,
        c + 1) if state.k is None or o <= state.k
61      ]
62      cooked["active_rows_number"] = sum(o > 0 for o in state.rows)
63      cooked["shortest_row"] = min((x for x in enumerate(state.rows)
        if x[1] > 0), key=lambda y: y[1])[0]
64      cooked["longest_row"] = max((x for x in enumerate(state.rows)),
         key=lambda y: y[1])[0]
65      cooked["random"]=random.choice([r for r, c in enumerate(state.
        rows) if c > 0])
66      cooked["nim_sum"] = nim_sum(state)
67      cooked["completion"] = sum(o for o in state.rows) / state.
        total_elements
68
69      brute_force = list()
70      for m in cooked["possible_moves"]:
71          tmp = deepcopy(state)
72          tmp.nimming(m)
73          brute_force.append((m, nim_sum(tmp)))
74      cooked["brute_force"] = brute_force
75      return cooked
```

Listing 17: interface to run Nim

In addition to the original one, we added some utility functions as the __hash__() that applies a hash function to the board passed as an argument.

3.3 Rule-based strategies

We started with some rule-based strategies that the professor provided us during the lectures:

- *randomNim*: Random row, random number of elements
- *gabrielleNim*: Pick always the maximum possible number of the lowest row

Therefore, we developed some strategies to benchmark our more advanced strategies, this usually includes lots of randomnesses, but some have a little logic to improve the finishing moves for example.

- *randomAllNim*: Random row, but pick the maximum number of elements
- *longestAllNim*: Pick always the longest row
- *randomSmartNim*: Here the strategy improves on the pure random, where it improves the last move

```
1
2  # strategies
3  def randomNim() -> Callable:
4      def pure_random(state: Nim) -> Nimply:
5          """Random row, random number of elements"""
6          row = random.choice([r for r, c in enumerate(state.rows) if
       c > 0])
7          num_objects = random.randint(1, state.rows[row])
8          return Nimply(row, num_objects)
9      return pure_random
10
11 def gabrieleNim() -> Callable:
12     def gabriele(state: Nim) -> Nimply:
13         """Pick always the maximum possible number of the lowest
       row"""
14         possible_moves = [(r, o) for r, c in enumerate(state.rows)
       for o in range(1, c + 1)]
15         return Nimply(*max(possible_moves, key=lambda m: (-m[0], m
       [1])))
16     return gabriele
17
18 def optimalStrategy() -> Callable:
19     def optimal_strategy(state: Nim) -> Nimply:
20         data = cook_status(state)
21         return next((bf for bf in data["brute_force"] if bf[1] ==
       0), random.choice(data["brute_force"]))[0]
22     return optimal_strategy
23
24 def randomAllNim() -> Callable:
25     def randomAll(state: Nim) -> Nimply:
26         row = random.choice([r for r, c in enumerate(state.rows) if
       c > 0])
27         num_objects = random.randint(1, state.rows[row])
28         return Nimply(row, num_objects)
```

```
    return randomAll

def longestAllNim() -> Callable:
    def longestAll(state: Nim) -> Nimply:
        row =max((x for x in enumerate(state.rows)), key=lambda y:
    y[1])[0]
        num_objects = state.rows[row]
        return Nimply(row, num_objects)
    return longestAll

# Here the strategy improves on the pure random, where it improves
    the last move
def randomSmartNim() -> Callable:
    def randomSmart(state: Nim) -> Nimply:
        data = cook_status(state)
        if data["active_rows_number"]==1:
           return Nimply(data["random"], state.rows[data["random"
    ]])
        else:
            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)
    return randomSmart
```

Listing 18: Rule-based agents that we used to test our strategies and some support functions

3.4 Evolutionary Approach: 2-parameters

These strategies are designed to evolve when competing against the benchmarks. We first tried to create some hard-coded algorithms that evolve only one parameter:

- *Elongest_allVS1*: From the longest row, take all the elements VS take only one
- *Eshortest_allVS1*: From the shortest row, take all the elements VS take only one
- *Eall_shortestVSlongest*: Take all the elements, from the shortest row VS from the longest
- *Elongest_1allVSALL*: From the longest row, if the board is full take all the elements VS to take only one. If the board is not full, take all the elements
- *Eshortest_1allVSALL*: From the shortest row, if the board is full take all the elements VS to take only one. If the board is not full, take all the elements
- *Erandom_allVS1*: From a random row, take only one element VS take a random number

On the other hand, we also developed strategies that use 2 parameters to evolve, for simplicity they are called "p1" and "p2". The evolution consists of evaluating the game and accordingly updating the parameters:

- *E2longestVSshortest_allVS1smart*: Shortest row vs Longest Row and pick one element vs take the maximum number of elements
- *E2randomVSshortest_allVS1smart*: Random row vs Shortest Row and pick one element vs take the maximum number of elements
- *E2longestVSshortest_allVS1allsmart*: Shortest row vs Longest Row and a complex selection of whether to take one element or a subset
- *E2longestVSrandom_allVS1allsmart*: Random row vs Longest Row and a complex selection of whether to take one element or a subset
- *E2shortestVSrandom_allVS1allsmart*: Random row vs Shortest Row and a complex selection of whether to take one element or a subset
- *EsafetySmart*: Safety strategy creates based on p1 "safety" nets to fall back on, as the game progresses. The idea is that creating rows with 2 elements can help this strategy win against more complex benchmarks as it uses the safety nets created at the beginning of the game to outsmart the opponent towards the end of the match

The corresponding code is shown in alg. 19.

```
# ONE PARAMETER EVOLUTION

def Elongest_allVS1(genome: dict) -> Callable:
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)

        if random.random() < genome["p"]:
            ply = Nimply(data["longest_row"], state.rows[data["longest_row"]])
        else:
            ply = Nimply(data["longest_row"], 1)

        return ply
    return evolvable

def Eshortest_allVS1(genome: dict) -> Callable:
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)

        if random.random() < genome["p"]:
            ply = Nimply(data["shortest_row"], state.rows[data["shortest_row"]])
        else:
            ply = Nimply(data["shortest_row"], 1)

        return ply
    return evolvable

```

```
def Eall_shortestVSlongest(genome: dict) -> Callable:
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)

        if random.random() < genome["p"]:
            ply = Nimply(data["shortest_row"], state.rows[data["shortest_row"]])
        else:
            ply = Nimply(data["longest_row"],  state.rows[data["longest_row"]])

        return ply
    return evolvable

def Elongest_1allVSALL(genome: dict) -> Callable:
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)

        if random.random() < genome["p"]:
            if(data["completion"]==1):
                ply= Nimply(data["longest_row"], 1)
            else:
                ply = Nimply(data["longest_row"], state.rows[data["longest_row"]])
        else:
            ply = Nimply(data["longest_row"], state.rows[data["longest_row"]])

        return ply
    return evolvable

def Eshortest_1allVSALL(genome: dict) -> Callable:
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)

        if random.random() < genome["p"]:
            if(data["completion"]==1):
                ply= Nimply(data["shortest_row"], 1)
            else:
                ply = Nimply(data["shortest_row"], state.rows[data["shortest_row"]])
        else:
            ply = Nimply(data["shortest_row"], state.rows[data["shortest_row"]])

        return ply
    return evolvable

def Erandom_allVS1(genome: dict) -> Callable:
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)

        if random.random() < genome["p"]:
            ply = Nimply(data["random"], state.rows[data["random"]])
        else:
            ply = Nimply(data["random"], 1)
```

```
77
78          return ply
79      return evolvable
80
81  def Erandom_j(genome: dict) -> Callable:
82      def evolvable(state: Nim) -> Nimply:
83          data = cook_status(state)
84
85          if random.random() < genome["p"]:
86              ply = Nimply(data["random"], state.rows[data["random"
    ]])
87          else:
88              ply = Nimply(data["random"], 1)
89
90          return ply
91      return evolvable
92
93  # 2 PARAMETERS EVOLUTION
94  def E2longestVSshortest_allVS1smart(genome: dict) -> Callable:
95      '''Shortest row vs Longest Row and pick one element vs take the
     maximum number of elements'''
96      def evolvable(state: Nim) -> Nimply:
97          data = cook_status(state)
98          if data["active_rows_number"]==1:
99             return Nimply(data["random"], state.rows[data["random"
    ]])
100
101         if random.random() < genome["p1"]:
102             if random.random() < genome["p2"]:
103                 ply = Nimply(data["longest_row"], state.rows[data["
    longest_row"]])
104             else:
105                 ply = Nimply(data["longest_row"], 1)
106         else:
107             if random.random() < genome["p2"]:
108                 ply = Nimply(data["shortest_row"], state.rows[data[
    "shortest_row"]])
109             else:
110                 ply = Nimply(data["shortest_row"], 1)
111
112         return ply
113     return evolvable
114
115 def E2randomVSshortest_allVS1smart(genome: dict) -> Callable:
116     '''Random row vs Shortest Row and pick one elements vs take the
     maximum number of elements'''
117     def evolvable(state: Nim) -> Nimply:
118         data = cook_status(state)
119         if data["active_rows_number"]==1:
120            return Nimply(data["random"], state.rows[data["random"
    ]])
121
122         if random.random() < genome["p1"]:
123             if random.random() < genome["p2"]:
124                 ply = Nimply(data["random"], state.rows[data["
    random"]])
125             else:
```

```
                ply = Nimply(data["random"], 1)
        else:
            if random.random() < genome["p2"]:
                ply = Nimply(data["shortest_row"], state.rows[data[
    "shortest_row"]])
            else:
                ply = Nimply(data["shortest_row"], 1)

        return ply
    return evolvable

def E2longestVSshortest_allVS1allsmart(genome: dict) -> Callable:
    '''Shortest row vs Longest Row and a complex selection of
    whether to take one element or a subset'''
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)
        if data["active_rows_number"]==1:
           return Nimply(data["random"], state.rows[data["random"
    ]])
        if random.random() < genome["p1"]:
            if random.random() < genome["p2"]:
                ply = Nimply(data["longest_row"], state.rows[data["
    longest_row"]])
            else:
                if(data["completion"]==1):
                    ply= Nimply(data["longest_row"], 1)
                else:
                    ply = Nimply(data["longest_row"], state.rows[
    data["longest_row"]])
        else:
            if random.random() < genome["p2"]:
                ply = Nimply(data["shortest_row"], state.rows[data[
    "shortest_row"]])
            else:
                if(data["completion"]==1):
                    ply= Nimply(data["shortest_row"], 1)
                else:
                    ply = Nimply(data["shortest_row"], state.rows[
    data["shortest_row"]])
        return ply
    return evolvable

def E2longestVSrandom_allVS1allsmart(genome: dict) -> Callable:
    '''Random row vs Longest Row and a complex selection of whether
     to take one element or a subset'''
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)
        if data["active_rows_number"]==1:
           return Nimply(data["random"], state.rows[data["random"
    ]])
        if random.random() < genome["p1"]:
            if random.random() < genome["p2"]:
                ply = Nimply(data["longest_row"], state.rows[data["
    longest_row"]])
            else:
                if(data["completion"]==1):
                    ply= Nimply(data["longest_row"], 1)
```

```
                else:
                    ply = Nimply(data["longest_row"], state.rows[
    data["longest_row"]])
        else:
            if random.random() < genome["p2"]:
                ply = Nimply(data["random"], state.rows[data["
    random"]])
            else:
                if(data["completion"]==1):
                    ply= Nimply(data["random"], 1)
                else:
                    ply = Nimply(data["random"], state.rows[data["
    random"]])
        return ply
    return evolvable

def E2shortestVSrandom_allVS1allsmart(genome: dict) -> Callable:
    '''Random row vs Shortest Row and a complex selection of
    whether to take one element or a subset'''
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)
        if data["active_rows_number"]==1:
           return Nimply(data["random"], state.rows[data["random"
    ]])
        if random.random() < genome["p1"]:
            if random.random() < genome["p2"]:
                ply = Nimply(data["shortest_row"], state.rows[data[
    "shortest_row"]])
            else:
                if(data["completion"]==1):
                    ply= Nimply(data["shortest_row"], 1)
                else:
                    ply = Nimply(data["shortest_row"], state.rows[
    data["shortest_row"]])
        else:
            if random.random() < genome["p2"]:
                ply = Nimply(data["random"], state.rows[data["
    random"]])
            else:
                if(data["completion"]==1):
                    ply= Nimply(data["random"], 1)
                else:
                    ply = Nimply(data["random"], state.rows[data["
    random"]])
        return ply
    return evolvable

def EsafetySmart(genome: dict) -> Callable:
    '''
        Safety strategy creates based on p1 "safety" nets to fall
    back on, as the game progresses.
        The idea is that creating rows with 2 elements can help
    this strategy win against more complex benchmarks as
        it uses the safety nets created at the beginning of the
    game to outsmart the opponent towards the end of the match
    '''
    #DICTIONARY OF PARAMETERS: {"p1", "p2"}
```

```
    # p1: makes safeties
    # p2: uses safeties
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)

        safety = []
        can_be_safety = []
        row = 0
        for row_val in state.rows:
            if row_val > 2:
                can_be_safety.append(row)
            if row_val < 2 and row_val == 1:
                safety.append(row)
            row += 1

        if data["active_rows_number"] == 1:
            # take the whole last row, last move to win
            ply = Nimply(data["longest_row"], state.rows[data["
    longest_row"]])
        elif len(safety) < genome["p1"] and len(can_be_safety) > 0:
            # need safety, make a safety
            row_choice = random.choice(can_be_safety)
            ply = Nimply(row_choice, state.rows[row_choice] - 2)
        elif data["completion"] < genome["p2"] and len(safety) > 0:
            # use safety
            row_choice = random.choice(safety)
            ply = Nimply(row_choice, 1)
        else:
            # do random move
            ply = Nimply(data["random"], state.rows[data["random"
    ]])
        return ply

    return evolvable

def E3shortestVSlongest_percentage(genome: dict) -> Callable:
    # DICTIONARY OF PARAMETERS: {"%_taken_longest", "%
    _taken_shortest", "binary_chance"}
    # binary chance: chance to taking shortest or longest
    # %_taken_shortest, %_taken_longest : percentage of object to
    take
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)

        if random.random() < genome['binary_chance']:
            x = max(1, int(state.rows[data['shortest_row']]*genome[
    'perc_taken']/100))
            ply = Nimply(data['shortest_row'],random.randint(1, x))
        else:
            x = max(1, int(state.rows[data['longest_row']]*genome['
    perc_taken']/100))
            ply = Nimply(data['longest_row'], random.randint(1, x))

        return ply
    return evolvable
```

Listing 19: Evolvable strategies

3.4.1 Experiments

A tournament has been set in order to evaluate the best strategy among the ones based on the evaluation. The algorithms have been ranked based on the result obtained against randomSmartNim(), which is the player that plays random moves but is able to recognize if he is in a winning situation. The results are shown in 2

opponent strategy	result
E2longestVSshortest_allVS1smart	0.45
E2longestVSshortest_allVS1allsmart	0.1
E2randomVSshortest_allVS1smart	0.14
EsafetySmart	0.47
E3shortestVSlongest_percentage	0.45

Table 2: Nim-size = 11 , games = 100

3.5 Genetic algorithm

3.5.1 Strategy

E3shortestVSlongest_percentage is the only strategy that uses the genetic approach with 3 paraments. Each parament represents the chance of taking the shortest or longest and how many elements in the row to pick.

3.5.2 Initial population

The individuals in the initial population are represented as a dictionary whose keys are *%_taken_longest*, *%_taken_shortest*, *binary chance*. The implementation of its initialization is reported in 20

```
1 def init_population() -> list:
2     population = list()
3     for _ in range(POPULATION_SIZE):
4         param = {'%_taken_longest': random.randint(0, 100), '%
    _taken_shortest': random.randint(0, 100), 'binary_chance' :
    random.random()}
5         if param not in population:
6             population.append((param, evaluate(make_strategy_evol(
    param),randomSmartNim(),11)))
7     return population
```

Listing 20: Initialization of the population

3.5.3 Fitness Metric

Our fitness metric is based on a reaped test (function evaluate) against a different strategy, usually a random strategy with a few tweaks to improve it.

```python
def evaluate(strategy1: Callable, strategy2: Callable, NIM_SIZE:
    int) -> float:
    opponent = (strategy1, strategy2)
    won = 0
    for m in range(NUM_MATCHES):
        nim = Nim(NIM_SIZE)
        player = 0
        while nim:
            ply = opponent[player](nim)
            #logging.debug(f"player: {player} ply: {ply}")
            nim.nimming(ply)
            player = 1 - player
        if player == 1:
            won += 1
    return won / NUM_MATCHES
```

Listing 21: Fitness evaluation

3.5.4 Genetic operators

The parent on which we apply the genetic operator is chosen with a tournament selection which basically consists in selecting two individuals randomly and selecting who has the best fitness. Moreover, we apply a random gaussian noise to the parameters of the parent and add it to the offspring list.

```python
def tweak(parameters) -> dict:
    new_param = dict()
    new_param["binary_chance"] = parameters["binary_chance"] +
    random.gauss(0, 0.1)
    new_param["%_taken_shortest"] = parameters["%_taken_shortest"]
    + random.gauss(0, 0.1)
    new_param["%_taken_longest"] = parameters["%_taken_longest"] +
    random.gauss(0, 0.1)
    return new_param
```

Listing 22: Genetic operators

Eventually, the whole algorithm works as follows:

```python
population = init_population()

for _ in range(1000):
    for __ in range(OFFSPRING_SIZE):
        offspring_pool = list()
        parameters = tournament_selection(population)
        offspring = tweak(parameters)
        o = (offspring, evaluate(E3shortestVSlongest_percentage(
    offspring), randomSmartNim(), NIM_SIZE))
        if o not in offspring_pool:
            offspring_pool.append(o)

    population += offspring_pool
    unique_population = list()
    for p in population:
        if p not in unique_population:
            unique_population.append(p)
    population = unique_population
```

```
    population.sort(key=lambda x: x[1], reverse=True)
    population = population[:POPULATION_SIZE]
    logging.debug(f"Iteration {_} : best {population[0][0]} nWin {
    population[0][1]}")
```

Listing 23: Complete genetic algorithm

3.6 Min-max

For what concerns the idea behind the min-max it is very close to the one that will be explained in 4.2.3 and the code is reported in 24

```
def evaluate1match(pl0,pl1,nim,activeRows=0) -> float:
    if not nim:
        if len(pl0)>len(pl1):
            return 1
        else:
            return -1
    else:
        return 0

def minmax(pl0,pl1,nim,player):
    possible= list()
    if nim:
        data=cook_status(nim)
        possible= data["possible_moves"]
        val=evaluate1match(pl0,pl1,nim,data["active_rows_number"])
    else:
        val=evaluate1match(pl0,pl1,nim)
    if(val or not possible):
        return None,val
    evaluations=list()
    for ply in possible:
        nimR=deepcopy(nim)
        nimR.nimming(ply)
        plr0=deepcopy(pl0)
        plr0.append(ply)
        _,val=minmax(pl1,plr0,nimR,1-player)
        evaluations.append((ply,-val))
        if(val==-1 and player==0):
            break
    return max(evaluations,key=lambda k: k[1])

def minmaxNim() -> Callable:
    def wrapperMinMax(state: Nim) -> Nimply:
        pl0=list()
        pl1=list()
        ply=minmax(pl0, pl1, state,0)[0]
        logging.debug(f"___________{ply}")
        return (ply[0],ply[1])
    return wrapperMinMax

def minimax_a_b_pruning(state: Nim, is_maximizing = False, alpha=1,
     beta=-1):
    possible = []
    if state:
```

```
        data = cook_status(state)
        possible = data["possible_moves"]

    val = ending_game(state, is_maximizing) #1 if player 0 win, -1 if player 1 win, 0 otherwise

    if val!=0 or not possible:
        return None, val

    evaluations = list()

    for ply in possible:
        state_copy = deepcopy(state)
        state_copy.nimming((ply))
        _,val = minimax_a_b_pruning(state_copy, is_maximizing= not is_maximizing, alpha = alpha, beta = beta)
        evaluations.append((ply,val))
        if is_maximizing:
            alpha = max(alpha, val)
        else:
            beta = min(beta, val)
        if beta <= alpha:
            break
    return max(evaluations, key=lambda x: x[1]) if is_maximizing else min(evaluations, key=lambda x: x[1])

def ending_game(player_1: list, player_2: list, state: Nim) -> int:
    if not state:
        if len(player_1) > len(player_2):
            return 1
        else:
            return -1
    else:
        return 0
```

Listing 24: minmax algorithm either with and without the $\alpha - \beta$ pruning

3.7 Reinforcement Learning

RL algorithm solves a continuously changing optimization problem with the following setup:

- The *state space* is defined by the vectors s_t which represents the board status, which means: how many pieces are available at each row of the board.
- The *action* is represented by the combination of the selected row from which the player wants to remove the pieces and the number of pieces to remove.
- The *reward function* maps the combination of state and possible action to a reward according to the Eq. 3 if it already exists, otherwise, a "fake" previous reward is initialized to a random number between -0.2 and 0.2 such that Eq. 3 can be used to obtain the new one.

$$q' = q + \alpha \cdot (r - q) \tag{1}$$

Figure 3: state/action → reward function

3.7.1 Agent design

Where q' is the updated reward, q is the reward of the current state, α is a discount factor and r is the gained outcome (either positive or negative) from the win/loss of the game. The corresponding code is reported in fig 25

```
    def learn(self, state, ply, reward):
        self.q[state][ply] = self.q[state][ply] + self.alpha * (
    reward - self.q[state][ply])
```

Listing 25: state/action → reward function

3.7.2 Agent

Each turn the trainable agent performs a ply that can be either random or the one that, in the q-table, maximizes the reward with a variable probability (*random_factor*) with respect to the current state. This functioning is shown in 26.

```
def choose_action(self, nim):
    action_chosen = None
    randomN = random.random()
    poss_actions = nim.possible_actions()

    if not hash(nim) in self.q:
        self._q.update({hash(nim) : {key : random.uniform(-0.2,
    0.2) for key in poss_actions}})

    if randomN < self.random_factor:
        # if random number below random factor, choose random
    action
        action_chosen = random.choice(poss_actions)
    else:
        # if exploiting, gather all possible actions and choose one
     with the highest G (reward)
        potential_actions = self.q[hash(nim)]
        action_chosen = max(potential_actions, key=
    potential_actions.get)

    self.update_state_history(nim, action_chosen)

    return action_chosen
```

Listing 26: Choose ply

3.7.3 Training

As we can see in 27, the training process, a game of nim is run, and if the agent wins, then a reward of $+1$ is saved in a list otherwise, a reward of -1 is stored.

```
nim = Nim(NIM_SIZE)
while nim:
    reward = 0
    ply = agent.choose_action(nim)
    nim.nimming(ply)
    if nim:
        ply = opponent(nim)
        agent.save_opponent_move(nim, ply)
        nim.nimming(ply)
        if nim.total_elements == 0:
            reward = -1
            counter_negative_rewards += 1
    else:
        reward = 1
        counter_positive_rewards += 1
agent.learn_all(reward)
```

Listing 27: Training process of reinforcement learning agent

As Nim can be solved by simple bit-wise operations, it can be defined as a deterministic game where if a player always applies the Nim sum and he always plays first, he can always win. Therefore, the whole training process is performed by alternating the player who has to play as first. So that the whole training process becomes 28.

```
for _ in range(number_of_trials):

    # if _ % 100 == 0:
    #     opponent = opponents[opponent_switch_counter]
    #     opponent_switch_counter += 1
    #     if opponent_switch_counter == 4:
    #         opponent_switch_counter = 0

    nim = Nim(NIM_SIZE)
    while nim:
        reward = 0
        ply = agent.choose_action(nim)
        nim.nimming(ply)
        if nim:
            ply = opponent(nim)
            agent.save_opponent_move(nim, ply)
            nim.nimming(ply)
            if nim.total_elements == 0:
                reward = -1
                counter_negative_rewards += 1
        else:
            reward = 1
            counter_positive_rewards += 1
    agent.learn_all(reward)

    nim = Nim(NIM_SIZE)
    while nim:
```

```
        reward = 0
        ply = opponent(nim)
        agent.save_opponent_move(nim, ply)
        nim.nimming(ply)
        if nim:
            ply = agent.choose_action(nim)
            nim.nimming(ply)
            if nim.total_elements == 0:
                reward = 1
                counter_positive_rewards += 1
        else:
            reward = -1
            counter_negative_rewards += 1
    agent.learn_all(reward)

    if(_==0.6*number_of_trials):
        agent.nowExploit(0.2)
```

Listing 28: *Complete* training process of reinforcement learning agent

Eventually, it has been proven that another useful implementation consists in storing, in the same q-table, also the states that the opponent has discovered and assign them a reward too, 29.

```
def save_opponent_move(self, state, ply):
    if not hash(state) in self.q:
        # self.test.append(hash(state))
        self._q.update({hash(state) : {key : random.uniform(-0.2,
    0.2) for key in state.possible_actions()}})

    self.history_opponent.append((hash(state), ply))
```

Listing 29: Storing and exploiting also the states discovered by the environment

3.7.4 Experiments

Agent trained with optimal strategy (nim-sum)

opponent strategy	result
optimal	0.91
randomSmartNim	1.0
random_all	1.0
E2longestVSshortest_allVS1smart	1.0

Table 3: Nim-size = 7 , games = 100000

Of course, the higher the Nim_size, the higher number of games the agent needs to be properly trained.

opponent strategy	result
optimal	0.06
randomSmartNim	1.0
random_all	1.0
E2longestVSshortest_allVS1smart	1.0

Table 4: Nim-size = 9 , games = 100000

opponent strategy	result
optimal	0.32
randomSmartNim	0.99
random_all	1.0
E2longestVSshortest_allVS1smart	1.0

Table 5: Nim-size = 9 , games = 200000

4 Quarto!

4.1 Introduction

Quarto is a board game invented by the Swiss mathematician Blaise Müller. It consists in placing 16 different pieces on a board of shape 4×4. All the available pieces are shared between players and each one has 4 binary features: white or black, tall or short, round or angular, and hollow or solid. It is usually played by 2 players who, in turn, have to perform two actions:

- *Choose piece*: choose the piece that in the next turn, the opponent has to place
- *Place piece*: place the piece provided by the opponent on an empty cell on the board.

Since the properties of pieces are binary, each piece can be mapped to its corresponding binary vector with fixed order and number of features: [high, colored, solid, squared]. Specifically, given the index of the piece (from 0 to 15), the binary vector is the binary encoding of the index.

The players' goal is to do *Quarto!* which consists of filling a row either horizontally or vertically or diagonally with 4 pieces that share at least one feature. For example, given the board a winning condition is the following:

10	x	x	3
x	x	15	x
0	**2**	**4**	**6**
x	5	x	x

Table 6: All the empty cells are represented with "x", the other values correspond to the index of the already placed piece

Notice if we compute the binary encoding of the pieces placed on the third horizontal row, the pieces share the same feature (the fourth) i.e. they are all *round* (or not squared).

4.1.1 Classification of the game

Quarto! is a *zero-sum* game with *perfect information* for the two players. *zero-sum* means that the outcomes for the two players sum to zero, in fact, if the player wins its outcome is 1, -1 if the player loses, and 0 if a draw occurs. Eventually, the *perfect information* feature is given by the fact that both players know the state of the game and all possible actions at any point in time.

The work that I'm going to present has been carried out by Cardano Filippo, Esposito Giuseppe, Piombino Giuseppe, and Scalera Francesco.

4.1.2 Strategy design

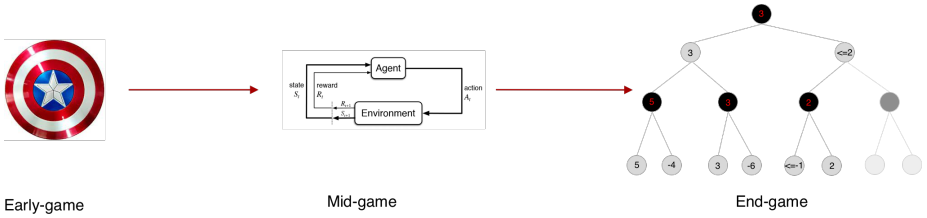


Figure 4: Our first strategy prototype

Given the afore described setup, it is easily deducible that, since *Quarto!* has 16 cells on the board and 16 pieces, the total number of states is $16!^2 \approx 4.4 \cdot 10^{26}$ and of course it is not possible, given our computational power, to visit all the possible states in the search space. In order to reduce it, thanks to the presentation [3], we notice that some states can be mapped to other equivalent states (symmetry) such that, the already visited ones output an outcome without any further computation. In [3], symmetries can focus on 2 main factors of the game: the board and the pieces, which reduce the search space to another one with the same order of magnitude. This still is not enough to meet the limits of our computational power.

The state-of-the-art [4] proposed minimax-based approach which has a feasible computation until a depth of 10. Such a technique is based on the assumption that the enemy is playing optimally. The min-max player can foresee a ¡max_depth¿ of plies.

Nevertheless, reinforcement learning is about achieving the long-term goal without dividing the problem into sub-tasks, thereby maximizing the overall rewards that the agent gets from an action related to a state. Furthermore, it is built on an adaptive framework that learns with experience as the agent continues to interact with the game, and with changing environmental constraints.

On the other hand, the large amount of states automatically discards the possibility to use an end-to-end model-free reinforcement learning algorithm with the Q-tables, as only the $10^{-18}\%$ of the total number of states would have been explored successfully within computational limits (taking the symmetries into account). Given this analysis, our intentions were to further reduce the number of states. To do so, at the early stages of the project, we introduced a rule-based logic consisting of designing a strategy with the pipeline described in fig. 4, because we thought that a 'defensive' strategy would have cut a lot of states from the search space, if it was used at the beginning of the game ($\approx 25\%$ of the game is played). Furthermore, the RL algorithm would train against a random player until the final plies ($\approx 80\%$ of the game is played). Eventually, the agent would finish with a min-max implementation. The main drawbacks that we faced concerned the effectiveness of the rule-based strategy and of the min-max player as first and last decision-makers. At each turn the rule-based strategy tries to end the game, by placing a piece in a winning *Quarto!* If this winning moves was not available then it tries to block a possible *Quarto!* for the other play. It searches for 3 aligned pieces that share a common feature, if so, place the piece at hand such that it blocks the 3 aligned pieces. The use of the rule-based player was minimal as it never meet such a situation. Despite the different optimization features we applied during the development of the min-max algorithm, described in sec. 4.2.3, during an ablation study we experienced that the performances of the algorithm were not affected by our design choice, even though we reached a tree max_depth of 4.

Given the analysis we have just presented, in the final version of the project, we decided to develop an agent which mixes up *Reinforcement Learning (with Q-table)* and a rule-based strategy i.e.*Defensive strategy*. In such a way we are drastically reducing the search space while making the agent able to perform exploration in the early stages of the game. Therefore we have decided to split each game into two phases such that the pipeline in fig 5 is performed.

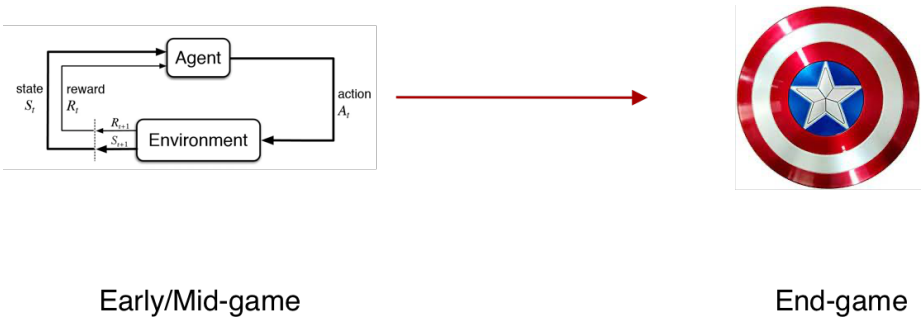


Figure 5: Our agent strategy

In agreement with this idea, we set the random factor fixed to 0.80 and eventually the agent is forced to play the rest of the game in a strategic manner according to the defensive strategy we are going to explain in 4.2.2. This empir-

ically proved that the best rule-based approach to the game is the defensive one and it is confirmed by some documentation [5] regarding the zero-sum game. The only drawback of this strategy lies in its definition because it is a *rule-based strategy* which means that it is a sequence of hard-coded instructions without any adaptive component.

4.2 Method

In this section, we explored the implementation detail of both the aforementioned approaches.

The applied algorithms for both prototypes are:

- *Reinforcement learning*: given a state it tries to choose the ply which maximizes the reward that the agent gets, with a variable probability.
- *Defensive strategy*: it tries to keep the game as far as possible from the end by checking winning conditions with some bit-wise operations. If it finds a winning ply then it is played.
- *min-max*: assuming that the opponent will always play the best plies within the next n number of turns (where n corresponds to the turns of both players which is the max_depth that the tree is able to reach), the algorithm tries to maximize its own reward, while the opponent's reward is minimized by simulating the next n turns.

Before going in-depth about each strategy, let us give some notation: from now on we will refer to the term *ply* as the sequence of the actions *choose piece* and *place piece*.

4.2.1 Reinforcement Learning

Algorithm design The data structure we decided to use to store the q-table is a python dictionary with the hash encoding of the board as key and a further dictionary of possible actions as value. Taking in consideration that the ply is composed by 2 actions, we split the table in two: one for the moves quality and another for the piece choices quality. In order to reduce the number of states, we performed a xor operation between the current state of the board and the piece at hand, the board is then hashed and the result is used as key in the dictionary of the quality of the moves.

To keep track of the plies of a whole game a list of moves and pieces has been used. Its elements are tuples of type (hash_board, ply).

RL algorithm solves a continuously changing optimization problem with the following setup:

- The *state space* is defined by the vectors s_x of 16 elements which represent the board and each element is a binary encoding of a piece placed in the corresponding position on the boards.

- The *action*, as stated before, is represented by the coordinates of the cell where the agent is intended to place the piece at hand (place_piece) and by the index of the piece to give the opponent for its next turn.
- The *reward function* updates the outcome of the quality table during the training, according to Eq. 6.

$$q' = q + \alpha \cdot (r - q) \tag{2}$$

Figure 6: state/action → reward function

Where q' is the updated quality, q is the quality of the current state, α is a discount factor and r is the gained reward (either positive or negative) from the win/loss of the game. The corresponding code is reported in fig 30

```
def __learn_moves(self, state, ply, reward):
    self.q_moves[state][ply] = self.q_moves[state][ply] + self.
  alpha * (reward - self.q_moves[state][ply])

def __learn_pieces(self, state, ply, reward):
    self.q_pieces[state][ply] = self.q_pieces[state][ply] + self.
  alpha * (reward - self.q_pieces[state][ply])
```

Listing 30: state/action → reward

We looked at a possible *canonical representation of the board*, where a set of boards are always represented in the same manner using symmetries. However, we found this not useful as it would only increment computational time and it would be more efficient to simply take the *canonical version of the board* as the first one that the RL agent encounters.

To get more into details about a ply:

- *Choose piece*: The board is checked against the existing q-table using symmetries or direct match, to see whether it is already exists (on the q-table). In both cases, the returned piece can be chosen randomly or from the action that maximizes the reward of the corresponding state with a given probability α (i.e. *random factor*). In absence of symmetry, if the game state is totally new (i.e. no matching board with applied symmetries is found), the q-table is updated according to the newly discovered state, where a new entry is created, composed of a dictionary with available pieces as key and an initialized value of 0. Finally, the history_state_pieces is updated. As stated before the last 6 pieces are managed with a rule-based policy (*Defensive strategy*).
- *Place piece*: it is a function used to choose the coordinates to place the piece. The agent takes note of the opponent ply and then looks for an action. In order to reduce the number of states to be explored, the board

has been xored with the place_to_piece. As in choose_piece the agent needs to check for symmetries and in case none is found, the q-table is updated with the new explored state. If a symmetry has been found, since the action has been chosen from a list of possible move for that specific symmetric board, it is transposed to a correspondent move that fit the current board (de_symm_move).

- *Save opponent*: this function is in charge of storing and updating the history of the state visited by the opponent. In this way the agent is able to learn from the actions of the opponent. Since the run was not allowed to be modified, it was necessary to gain all the necessary information by analyzing the board and keeping track of its states. The function is called when the agent is in the initial phase of his ply (place piece). At this stage of the ply, the agent knows that the piece chosen by the opponent is the piece_to_place and that the state of the board when it made this choice is the current one. Differently, for the history of the moves, it is necessary to know the state of the board before the place_piece of the opponent and the piece he had at hand. To do so, it is necessary to store the state of the board of the agent when he finishes his piece_place. Having the previous state of the board (preGame), is very simple to recover the move chosen by the opponent and the piece he had at hand. Finally, the history of moves is updated, in the same way of the agent, using as index the state of the board *xored* with the piece at hand. In both cases to avoid increasing the size of the q-tables, the board is checked for symmetries. If a symmetry has been found, to use it as a state, the move of the opponent has to be translated with the correspondent symmetric move. It is done by means of the de_symm_move, originally used to undo a symmetry on a move.

Eventually, at training time, as this is a zero-sum game, the returned reward from a win is 1, from a draw is 0 and from a loss is -1 so that, after a certain number of games, the q-table is updated accordingly to the reward function defined in subsec. 4.2.1. In alg. 31 is presented the evaluation function at training time of the algorithm.

```
def trainReinforcement(player0):

    version = 0
    game = Quarto()

    player0_withgame = player0(game)
    player1_withgame = RandomPlayer(game)

    for _ in range(0, 10_000_000):

        if _ % 50_000 == 0 and _ != 0:
            player0_withgame.save_to_db(version)
            version += 1

        if _ % 1000 == 0:
            print(_)

```

```
        game.reset()

        game.set_players((player0_withgame, player1_withgame))
        winner = game.run()

        if winner == 0:
            player0_withgame.learn_all(1)
        if winner == 1:
            player0_withgame.learn_all(-1)
        if winner == -1:
            player0_withgame.learn_all(1)

        game.reset()

        game.set_players((player1_withgame, player0_withgame))
        winner = game.run()

        if winner == 1:
            player0_withgame.learn_all(1)
        if winner == 0:
            player0_withgame.learn_all(-1)
        if winner == -1:
            player0_withgame.learn_all(1)

        # print(_)
```

Listing 31: Evaluation function for the reinforcement learning algorithm at training time.

Now let us recall that the total number of states is of the order of magnitude 10^{26}, then the q-tables would be very huge, this is why our first challenge consists of optimizing space and computational cost. **Seeking symmetries** In mathematics, symmetry is an operation that moves or transforms an object while leaving its appearance unchanged. The main scope of their implementation is to map the board hashes with the corresponding symmetric ones, such that every time a reward in the q-table must be updated, only one item in the dictionary has changed. The function used to seek the symmetries is reported in alg. 32

3 Guided us toward the choice of the possible symmetries to implement. In fig. 7 are reported all the checked symmetries between the different states.

```

def seek_symm_moves(self, board):
    hash_board = 0
    found_symmetry = False
    sym_type = ''
    sym_count = 0
    if not found_symmetry:
        for r in range(1,4):
            copyBoard = symmRotation(board, r)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_moves:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type = 'rot'
```

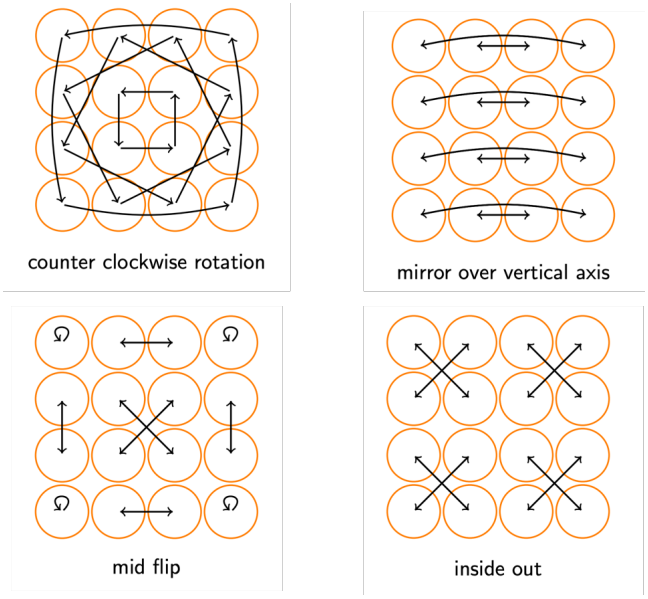


Figure 7: Applied symmetries from [3]

```
                sym_count = r
                break

    if not found_symmetry:
        copyBoard = symmFlipOrizontal(board)
        hash_board_Symm = boardHash(copyBoard)
        if hash_board_Symm in self.q_moves:
            hash_board = hash_board_Symm
            found_symmetry = True
            sym_type='flip_h'

    if not found_symmetry:
        copyBoard = symmFlipVertical(board)
        hash_board_Symm = boardHash(copyBoard)
        if hash_board_Symm in self.q_moves:
            hash_board = hash_board_Symm
            found_symmetry = True
            sym_type='flip_v'

    if not found_symmetry:
        copyBoard = symmFlipMid(board)
        hash_board_Symm = boardHash(copyBoard)
        if hash_board_Symm in self.q_moves:
            hash_board = hash_board_Symm
            found_symmetry = True
            sym_type='flip_m'

    if not found_symmetry:
        copyBoard = symmFlipInside(board)
```

```
        hash_board_Symm = boardHash(copyBoard)
        if hash_board_Symm in self.q_moves:
            hash_board = hash_board_Symm
            found_symmetry = True
            sym_type='flip_i'

    return hash_board, sym_type, sym_count
```

Listing 32: Checks whether, applying a symmetry, an already discovered board and map the hash of the current board to the one of the already saved board

For the sake of efficiency and space optimization, we have noticed that, when a place piece action has to be performed, instead of trying to place the piece in all the possible cells and then apply the hash function to the resulting board, we apply a xor logic operator between the boolean encoding of the piece to place and all the pieces already placed on the board. The intuition that led us to this choice is the fact that the winning condition occurs when, given a piece at hand (a) with an arbitrary boolean encoding, the following equation is satisfied.

$$a \oplus b \vee a \oplus c \vee a \oplus d = e \tag{3}$$

Figure 8: Winning condition in bit-wise operations where a, b, c, and d are 4 aligned pieces and e is a vector which contains at least one 0

4.2.2 Defensive Strategy

The defensive strategy is characterized by a simple ply function:

- *Choose piece*: Avoid giving the piece which lets the opponent win. If there is are no remaining pieces, randomly sample a piece from the list of available pieces.
- *Place piece*: If there is any winning move, play it, otherwise avoid choosing a cell that aligns three pieces with at least a common feature.

We decided to keep the number of decisions made by this strategy low (6) such that it is able to play the last 10% of each game. The source code is shown in 33

```
class DefensivePlayer(Player):
    """DefensivePlayer player"""

    def __init__(self, quarto: Quarto) -> None:
        super().__init__(quarto)
        self.all = [*range(0, 16, 1)]
        self.cells_row = [*range(0,4)]

    def choose_piece(self) -> int:
        board = self.get_game().get_board_status()
        _possible_pieces = possible_pieces(board)
```

```
        _possible_moves = possible_moves(board)
        for piece in _possible_pieces:
            for move in _possible_moves:
                new_board = deepcopy(board)
                new_board[move] = piece
                if check_current_player_winner(new_board):
                    # we know this piece is bad
                    _possible_pieces.remove(piece)
                    break

        if len(_possible_pieces) > 0:
            return _possible_pieces[0]
        else:
            return random.choice(possible_pieces(board))

    def place_piece(self) -> tuple[int, int]:
        board = self.get_game().get_board_status()
        piece_to_place = self.get_game().get_selected_piece()
        _possible_moves = possible_moves(board)
        for move in _possible_moves:
            # here we can test for the piece to place
            new_board = deepcopy(board)
            new_board[move] = piece_to_place
            if check_current_player_winner(new_board):
                return move[1], move[0]

        # 2 check if there is a 3 piece row or column or diagnonal
        x, y = check_tris(board)
        if x != -1:
            return y, x
        else:
            return random.randint(0, 3), random.randint(0, 3)


    def possible_moves(board) -> list():
        magic = np.where(board == -1)
        return [mov for mov in zip(magic[0], magic[1])]

    def possible_pieces(board) -> list():
        flattened_board = sum(board.tolist(), [])
        return list(set(all) - set(flattened_board))


    def check_current_player_winner(in_board) -> bool:
        board = number_to_binary(in_board)

        # horizontal
        hsum = np.sum(board, axis=1)
        if BOARD_SIDE in hsum or 0 in hsum:
            return True

        # vertical
        vsum = np.sum(board, axis=0)
        if BOARD_SIDE in vsum or 0 in vsum:
            return True

        # diagonal
```

```
        dsum1 = np.trace(board, axis1=0, axis2=1)
        dsum2 = np.trace(np.fliplr(board), axis1=0, axis2=1)
        if BOARD_SIDE in dsum1 or BOARD_SIDE in dsum2 or 0 in dsum1
    or 0 in dsum2:
            return True

        return False
```

Listing 33: Defensive strategy

4.2.3 Min-max

As presented in [4], min-max is based on the assumption that the opponent for the next n moves, will play optimally, where n is the depth of the generated tree which means, how many plies the min-max algorithm is able to forecast.

```
def minmax(self, board, isMax, depth, alpha, beta):

        # does this player win?
        if self.check_current_player_winner(board):
            if isMax:
                return 10000
            else:
                return -10000
        else:
            if self.check_if_board_full(board):
                return 0

        if depth == 0:
            return self.utility(board, isMax)

        if isMax:
            best_value = -float("inf")
            possible_ply = self.possible_ply(board)
            for ply in possible_ply:
                new_board = deepcopy(board)
                new_board[ply[0]] = ply[1]
                value = self.minmax(board = new_board,
                                    isMax = False,
                                    depth = depth - 1,
                                    alpha = alpha,
                                    beta = beta)
                best_value = max(best_value, value)
                alpha = max(alpha, best_value)
                if beta <= alpha:
                    break
            return best_value
        else:
            best_value = float("inf")
            possible_ply = self.possible_ply(board)
            for ply in possible_ply:
                new_board = deepcopy(board)
                new_board[ply[0]] = ply[1]
                value = self.minmax(board = new_board,
                                    isMax = True,
                                    depth = depth - 1,
```

```
                                    alpha = alpha,
                                    beta = beta)
                best_value = min(best_value, value)
                beta = min(beta, best_value)
                if beta <= alpha:
                    break
            return best_value

    def possible_ply(board) -> list():
        _possible_moves = possible_moves(board)
        _possible_pieces = possible_pieces(board)
        possible_plies = list(product(_possible_moves,
    _possible_pieces))
        random.shuffle(possible_plies)
        return possible_plies
```

Listing 34: recursive function with limitation and optimization for the tree design

As we can see in alg. 34, the ply is defined as a tuple (move, chosen_piece), and for all the legal plies a recursive call is performed, where the flag isMax is continuously switched such that we can simulate the behavior of maximizing the reward of the agent and minimizing the reward of the opponent. The distinction lies in the definition of the functions that compose the plies, 35. *Choose_piece* is set such that it has to Minimize the list of evaluations which means minimizing the reward that the opponent would receive in playing with the corresponding selected piece. On the other hand *Place_piece* is set such that it has to Maximize the list of evaluations which means Maximizing the reward the agent would receive in placing the piece at hand in the corresponding cell.

```
class MinMaxPlayer3(Player):
    def __init__(self, quarto: Quarto):
        super().__init__(quarto)
        self.max_depth = 2

    def choose_piece(self):
        # check = possible_pieces(self.get_game().get_board_status
    ())
        # if len(check) > 14:
        #     return random.randint(0, 15)

        board = self.get_game().get_board_status()
        possible_plies = possible_ply(board)

        evaluations = list()
        for ply in possible_plies:
            b = board.copy()
            b[ply[0]] = ply[1]
            eval = self.minmax(board = b, depth = self.max_depth,
    isMax = True, alpha = -float("inf"), beta = float("inf"))
            evaluations.append((ply[1], eval))

        all = dict()
        for k, v in evaluations:
            if k in all:
                if all[k] < v:
```

```
                    all[k] = v
                # all[k] = all[k] + v
            else:
                all.update({k: v})

        min_ply = min(all, key=all.get)

        print(f"min_ply piece: {min_ply}")

        return min_ply

    def place_piece(self):
        # check = self.possible_pieces(self.get_game().
    get_board_status())
        # if len(check) > 14:
        #     return random.randint(0, 3), random.randint(0, 3)

        piece_to_place = self.get_game().get_selected_piece()
        board = self.get_game().get_board_status()

        possible_ply = list()
        boardSquares = list(product([*range(0,4)],repeat=2))
        for (x, y) in boardSquares:
            if board[x, y] == -1:
                possible_ply.append(((x, y), piece_to_place))

        evaluations = list()
        for ply in possible_ply:
            b = board.copy()
            b[ply[0]] = ply[1]
            eval = self.minmax(board = b, depth = self.max_depth,
    isMax = True, alpha = -float("inf"), beta = float("inf"))
            evaluations.append((ply, eval))

        max_ply = max(evaluations,key=lambda k: k[1])

        print(f"max_ply move: {max_ply}")

        return max_ply[0][0][1], max_ply[0][0][0]

    def minmax(self, board, isMax, depth, alpha, beta):

        # does this player win?
        if check_current_player_winner(board):
            if isMax:
                return 10000
            else:
                return -10000
        else:
            if check_if_board_full(board):
                return 0

        if depth == 0:
            return self.utility(board, isMax)

        if isMax:
            best_value = -float("inf")
```

```
            possible_plies = possible_ply(board)
            for ply in possible_plies:
                new_board = board.copy()
                new_board[ply[0]] = ply[1]
                value = self.minmax(board = new_board, isMax =
    False, depth = depth - 1, alpha = alpha, beta = beta)
                best_value = max(best_value, value)
                alpha = max(alpha, best_value)
                if beta <= alpha:
                    break
            return best_value
        else:
            best_value = float("inf")
            possible_plies = possible_ply(board)
            for ply in possible_plies:
                new_board = board.copy()
                new_board[ply[0]] = ply[1]
                value = self.minmax(board = new_board, isMax = True
    , depth = depth - 1, alpha = alpha, beta = beta)
                best_value = min(best_value, value)
                beta = min(beta, best_value)
                if beta <= alpha:
                    break
            return best_value
```

Listing 35: minmax main class

Now are missing two things: an early stopping condition (max_depth) which depends on the computational cost of the whole algorithm and an alpha-beta pruning which lighten the computational cost. In the implementation of our early stopping condition, it is interesting to notice as soon as the stopping condition occurs, the returned evaluation is designed by a heuristic function that basically evaluated how close the game is to the end checking the number of available pieces and how many pieces are aligned along the rows, the columns or the diagonals.

```
    def utility(self, int_board, isMax):
        score = 0

        # pieces_remaining = possible_pieces(int_board)

        binary_board = number_to_binary(int_board)

        # Number of completed rows/columns
        row = np.all(int_board != -1, axis = 0)
        col = np.all(int_board != -1, axis = 1)
        score += row.sum() * 10
        score += col.sum() * 10

        # Number of pieces with matching attributes
        magic = np.where(int_board != -1)
        for i, j in zip(magic[0], magic[1]):
            piece = binary_board[i][j]

            # check row
            for k in range(4):
                if k != j:
```

```
                    c = (piece == binary_board[i][k])
                    score += c.sum() * 3

            # check col
            for k in range(4):
                if k != i:
                    c = (piece == binary_board[k][j])
                    score += c.sum() * 3

            # check if piece is on diagonal
            if (i == j):
                for k in range(4):
                    if k != j:
                        c = (piece == binary_board[k][k])
                        score += c.sum() * 3

            if ((i + j) == 3):
                for k in range(4):
                    if k != i and (3 - k) != j:
                        c = (piece == binary_board[k][3 - k])
                        score += c.sum() * 3

        if isMax:
            return score
        else:
            return -score
```

4.3 Experiments

In order to test our agent, we developed some other expert strategies which we retained to be some valid players and trainers for RL algorithm. We developed 3 strategies that are:

- *RandomPlayer*: each ply is simply chosen at random. It has been useful specifically for the first steps of the train.
- *DefensivePlayer*: It is basically the algorithm used at the beginning of our final agent.
- *AggressivePlayer*: It always tries to finish the game as soon as possible but first checks if his next ply would let the opponent win, in such a case, it gives a random piece among the remaining valid ones. The corresponding code is shown in alg. 36
- *CornerFirst*: It places the first piece on the corner of the board and then it starts playing randomly until the end of the game.
- *CenterFirst*: It places the first piece on one of the central cells of the board and then it starts playing randomly until the end of the game.

Eventually, we mixed up some of these in order to obtain strategies that in our mind resulted more compliant with the idea behind it. For example, we have *FirstCornerDefensivePlayer* (notice that, if the first piece is placed on the

corner there are fewer pieces to place around that piece and this is why we retained it a more conservative ply) and *FirstCenterAggressivePlayer* (on the other hand, if the piece is placed in the center of the board, there are more empty cells around that piece so that more possibility to get closer to the end).

```
class AggressivePlayer(Player):
    """AggressivePlayer player"""

    def __init__(self, quarto: Quarto) -> None:
        super().__init__(quarto)
        self.cells_row = [*range(0,4)]

    def choose_piece(self) -> int:
        board = self.get_game().get_board_status()
        _possible_pieces = possible_pieces(board)
        _possible_moves = possible_moves(board)
        for piece in _possible_pieces:
            for move in _possible_moves:
                new_board = deepcopy(board)
                new_board[move] = piece
                if check_current_player_winner(new_board):
                    # we know this piece is bad
                    _possible_pieces.remove(piece)
                    break

        if len(_possible_pieces) > 0:
            return _possible_pieces[0]
        else:
            return random.choice(possible_pieces(board))

    def place_piece(self) -> tuple[int, int]:
        piece_to_place = self.get_game().get_selected_piece()
        board = self.get_game().get_board_status()

        list_of_best = []

        poss_moves = possible_moves(board)

        for ply in poss_moves:
            if board[ply] == -1:
                # here we can test for the piece to place
                b = board.copy()
                b[ply] = piece_to_place
                for j in range(4, 1, -1):
                    res = check_horizontal(b, j)
                    if res:
                        list_of_best.append((j, ply))
                    res = check_vertical(b, j)
                    if res:
                        list_of_best.append((j, ply))
                    res = check_diagonal(b, j)
                    if res:
                        list_of_best.append((j, ply))

        if len(list_of_best) > 0:
            max_ply = max(list_of_best, key=lambda k: k[0])
            return max_ply[1][1], max_ply[1][0]
```

```
53          else:
54              return random.randint(0, 3), random.randint(0, 3)
```

Listing 36: Aggressive strategy

4.4 Results

Here is a table of results for which we compare the various Algorithms against each other. Mainly we are splitting the comparison from three different points of view:

- *Rule based Agents*
- *Reinforcement based Agent*
- *MinMax based Agent*

4.4.1 Rule based Agent

win	all	tie	win/all
9182	9958	42	0.9220

Table 7: Defensive VS Random Algorithms

win	all	tie	win/all
8351	10000	0	0.8351

Table 8: Defensive VS Aggressive Algorithms

4.4.2 Reinforcement-based Agent: Trained against Defensive Player

win	all	tie	win/all	q moves hit	q moves miss	q piece hit	q piece miss
5271	9932	68	0.5307	25042	13703	27244	10526

Table 9: Reinforcement VS Defensive Algorithms

win	all	tie	win/all	q moves hit	q moves miss	q piece hit	q piece miss
8201	9983	17	0.8214	20785	5570	24583	4557

Table 10: Reinforcement VS Aggressive Algorithms

win	all	tie	win/all	q moves hit	q moves miss	q piece hit	q piece miss
9139	9939	61	0.9195	20443	26023	15449	26550

Table 11: Reinforcement VS Random Algorithms

4.4.3 Reinforcement-based Agent: Trained against Random Player

win	all	tie	win/all	q moves hit	q moves miss	q piece hit	q piece miss
4838	9911	89	0.4881	19432	24131	20721	19195

Table 12: Reinforcement VS Defensive Algorithms

win	all	tie	win/all	q moves hit	q moves miss	q piece hit	q piece miss
8384	9988	12	0.839	19420	6396	20442	8358

Table 13: Reinforcement VS Aggressive Algorithms

win	all	tie	win/all	q moves hit	q moves miss	q piece hit	q piece miss
9143	9939	61	0.920	19460	26437	19502	22386

Table 14: Reinforcement VS Random Algorithms

4.4.4 Reinforcement-based Agent: Trained against Random and Defensive

win	all	tie	win/all	q moves hit	q moves miss	q piece hit	q piece miss
4714	9933	67	0.474	24531	17628	26539	13379

Table 15: Reinforcement VS Defensive Algorithms

win	all	tie	win/all	q moves hit	q moves miss	q piece hit	q piece miss
9100	9952	48	0.914	20197	26648	20003	22245

Table 16: Reinforcement VS Random Algorithms

win	all	tie	win/all	q moves hit	q moves miss	q piece hit	q piece miss
7448	9996	12	0.745	21448	5276	24439	5479

Table 17: Reinforcement VS Aggressive Algorithms

4.4.5 Min-max tested against Rule-based players

win	all	tie	win/all
10	10	0	1.0

Table 18: Min-max VS Random Players

win	all	tie	win/all
7	10	0	0.85

Table 19: Min-max VS Defensive Players

win	all	tie	win/all
10	10	0	1.0

Table 20: Min-max VS Aggressive Players

4.5 Conclusion

Our experiments has shown that play Quarto! with Defensive Strategy produced excellent results in terms of win ratio (about 91%), that is the metric that we used to assess the goodness of our method. We notice also that the mixed strategy who involves RL + Defensive Strategy is able to train an agent capable to produce comparable results with respect expert system. In fact, with this approach the strategy produces 92% of win ratio, the best results achieved so far. Since the improvement gained with the RL are very small, our conclusion is that, among the competence learnt during this course, the best choice was a rule-based agent and more effort should have been dedicated to it instead of RL.

4.6 Future work

Further improvement can be applied on our implementation. We would suggest:

- The defensive strategy could be turned into an evolutionary strategy with some self-adaptive components.
- RL can be turned into deep RL.
- Implementation of other strategies like the mirror one
- train the RL model against itself

Since the RL seems to be not very influential on the results further strategies can be tried. Instead of correcting the behavior of the RL with a rule-based strategy, it is possible to improve the behavior of a rule-based strategy using the RL. To do so, instead of exploring the states by means of a random choice, the agent should always choose the rule-based strategy. In this way, it should be able to learn when the rule-based fails and when the choice is good. However, our predictions on this trial are not positive and the improvement is not expected influent enough.

4.7 Appendix

4.7.1 Reinforcement learning algorithms

```
import quarto
from quarto import Quarto, Player
import random
import sqlite3
import sys
import gc
import numpy as np
from utils import *

class ReinforcementV1(Player):
    """ReinforcementV1 player"""

    def __init__(self, quarto: Quarto, alpha=0.15, random_factor
    =0.8) -> None:
        super().__init__(quarto)
        self.cells_row = [*range(0,4)]
        self.all = [*range(0, 16, 1)]
        self.q_moves = dict()
        self.q_pieces = dict()
        self.alpha = alpha
        self.random_factor = random_factor
        self.history_state_moves = list()
        self.history_state_pieces = list()
        self.opp_history_state_moves = list()
        self.opp_history_state_pieces = list()
        self.hitratio = 0
        self.preGame = np.array([])

        self.read_from_db(5)

    def seek_symm_pieces(self, board):
        hash_board = 0
        found_symmetry = False
        sym_type = ''
        sym_count = 0
        if not found_symmetry:
            for r in range(1,4):
                copyBoard = symmRotation(board, r)
                hash_board_Symm = boardHash(copyBoard)
                if hash_board_Symm in self.q_pieces:
                    hash_board = hash_board_Symm
                    found_symmetry = True
                    sym_type = 'rot'
                    sym_count = r
                    break

        if not found_symmetry:
            copyBoard = symmFlipOrizontal(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_pieces:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_h'
```

```
53
54          if not found_symmetry:
55              copyBoard = symmFlipVertical(board)
56              hash_board_Symm = boardHash(copyBoard)
57              if hash_board_Symm in self.q_pieces:
58                  hash_board = hash_board_Symm
59                  found_symmetry = True
60                  sym_type='flip_v'
61
62          if not found_symmetry:
63              copyBoard = symmFlipMid(board)
64              hash_board_Symm = boardHash(copyBoard)
65              if hash_board_Symm in self.q_pieces:
66                  hash_board = hash_board_Symm
67                  found_symmetry = True
68                  sym_type='flip_m'
69
70          if not found_symmetry:
71              copyBoard = symmFlipInside(board)
72              hash_board_Symm = boardHash(copyBoard)
73              if hash_board_Symm in self.q_pieces:
74                  hash_board = hash_board_Symm
75                  found_symmetry = True
76                  sym_type='flip_i'
77
78          return hash_board, sym_type, sym_count
79
80      def seek_symm_moves(self, board):
81          hash_board = 0
82          found_symmetry = False
83          sym_type = ''
84          sym_count = 0
85          if not found_symmetry:
86              for r in range(1,4):
87                  copyBoard = symmRotation(board, r)
88                  hash_board_Symm = boardHash(copyBoard)
89                  if hash_board_Symm in self.q_moves:
90                      hash_board = hash_board_Symm
91                      found_symmetry = True
92                      sym_type = 'rot'
93                      sym_count = r
94                      break
95
96          if not found_symmetry:
97              copyBoard = symmFlipOrizontal(board)
98              hash_board_Symm = boardHash(copyBoard)
99              if hash_board_Symm in self.q_moves:
100                 hash_board = hash_board_Symm
101                 found_symmetry = True
102                 sym_type='flip_h'
103
104         if not found_symmetry:
105             copyBoard = symmFlipVertical(board)
106             hash_board_Symm = boardHash(copyBoard)
107             if hash_board_Symm in self.q_moves:
108                 hash_board = hash_board_Symm
109                 found_symmetry = True
```

```
                sym_type='flip_v'

        if not found_symmetry:
            copyBoard = symmFlipMid(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_moves:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_m'

        if not found_symmetry:
            copyBoard = symmFlipInside(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_moves:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_i'

        return hash_board, sym_type, sym_count


    def choose_piece(self) -> int:
        board = self.get_game().get_board_status()

        piece_choosen = None
        randomN = random.random()
        hash_board = boardHash(board)

        if hash_board not in self.q_pieces:
            #look for symmetries
            hash_board_temp, _, _ = self.seek_symm_pieces(board)
            if hash_board_temp != 0:
                hash_board=hash_board_temp

        poss_pieces = possible_pieces(board)

        if not hash_board in self.q_pieces:
            self.q_pieces.update({hash_board : {one_piece : 0 for
    one_piece in poss_pieces}})
        else:
            self.hitratio += 1

        if randomN < self.random_factor:
            piece_choosen = random.choice(poss_pieces)
        else:
            potential_pieces = self.q_pieces[hash_board]
            piece_choosen = max(potential_pieces, key=
    potential_pieces.get)

        self.update_history_state_pieces(hash_board, piece_choosen)
        return piece_choosen

    def update_history_state_moves(self, hash_board, ply):
        self.history_state_moves.append((hash_board, ply))

    def update_history_state_pieces(self, hash_board, ply):
        self.history_state_pieces.append((hash_board, ply))
```

```
165
166     def update_opp_history_state_moves(self, hash_board, ply):
167         self.opp_history_state_moves.append((hash_board, ply))
168
169     def update_opp_history_state_pieces(self, hash_board, ply):
170         self.opp_history_state_pieces.append((hash_board, ply))
171
172     def saveOpponentMoveAndPiece(self, piece_to_place):
173         if len(self.preGame) == 0:
174             return
175
176         board=self.get_game().get_board_status()
177         opponentMoveDirt = np.where(( board - self.preGame) != 0)
178         opponentMove = (opponentMoveDirt[0][0], opponentMoveDirt
    [1][0])
179         hash_board = boardHash(board)
180
181         if hash_board not in self.q_pieces:
182             hash_board_temp, _, _ = self.seek_symm_pieces(board)
183             if hash_board_temp != 0:
184                 hash_board=hash_board_temp
185
186         if not hash_board in self.q_pieces:
187             poss_pieces = possible_pieces(board)
188             self.q_pieces.update({hash_board : {one_piece : 0 for
    one_piece in poss_pieces}})
189
190         self.update_opp_history_state_pieces(hash_board,
    piece_to_place)
191
192         prePiece= board[opponentMove[0]][opponentMove[1]]
193
194         xored_board = xorer(self.preGame, prePiece)
195         found_symm = False
196         hash_board = boardHash(xored_board)
197         if hash_board not in self.q_moves:
198             #look for symmetries
199             hash_board_temp, sym_type, sym_count = self.
    seek_symm_moves(xored_board)
200             if hash_board_temp != 0:
201                 found_symm = True
202                 hash_board = hash_board_temp
203
204         if hash_board not in self.q_moves:
205             poss_actions = possible_moves(self.preGame)
206             self.q_moves.update({hash_board : {one_action : 0 for
    one_action in poss_actions}})
207
208         if found_symm:
209             opponentMoveSymm = de_symm_move(opponentMove, sym_type,
     -sym_count-4)
210             self.update_opp_history_state_moves(hash_board,
    opponentMoveSymm)
211         else:
212             self.update_opp_history_state_moves(hash_board,
    opponentMove)
213
```

```
214
215
216
217     def place_piece(self) -> tuple[int, int]:
218         board = self.get_game().get_board_status()
219         piece_to_place = self.get_game().get_selected_piece()
220
221         action_chosen = None
222         randomN = random.random()
223
224         self.saveOpponentMoveAndPiece(piece_to_place)
225
226         # board xor with the piece
227         # seek sym on board xored
228         xored_board = xorer(board, piece_to_place)
229         found_symm = False
230         hash_board = boardHash(xored_board)
231         if hash_board not in self.q_moves:
232             #look for symmetries
233             hash_board_temp, sym_type, sym_count = self.
    seek_symm_moves(xored_board)
234             if hash_board_temp != 0:
235                 found_symm = True
236                 hash_board = hash_board_temp
237                 # print("sym")
238
239         if found_symm:
240             # print(f'found symm: {found_symm}')
241             if randomN < self.random_factor:
242                 # print(f'random: {randomN}')
243                 potential_actions_sym = list(self.q_moves[
    hash_board].keys())
244                 action_chosen_sym = random.choice(
    potential_actions_sym)
245             else:
246                 # print(f'not random: {randomN}')
247                 potential_actions_sym = self.q_moves[hash_board]
248                 action_chosen_sym = max(potential_actions_sym, key=
     potential_actions_sym.get)
249
250             self.update_history_state_moves(hash_board,
    action_chosen_sym)
251             action_chosen = de_symm_move(action_chosen_sym,
    sym_type, sym_count)
252         else:
253             initial_poss_actions = possible_moves(board)
254             if not hash_board in self.q_moves:
255                 self.q_moves.update({hash_board : {one_action : 0
    for one_action in initial_poss_actions}})
256
257             # print(f'not found symm: {found_symm}')
258             if randomN < self.random_factor:
259                 # print(f'random: {randomN}')
260                 action_chosen = random.choice(initial_poss_actions)
261             else:
262                 # print(f'not random: {randomN}')
263                 potential_actions = self.q_moves[hash_board]
```

```
264                 action_chosen = max(potential_actions, key=
    potential_actions.get)
265 
266             self.update_history_state_moves(hash_board,
    action_chosen)
267 
268 
269         copy_board = board
270         copy_board[action_chosen[0], action_chosen[1]] =
    piece_to_place
271         self.preGame = copy_board
272 
273         return action_chosen[1], action_chosen[0]
274 
275     def learn_all(self, reward):
276 
277         for h in self.history_state_moves:
278             self.__learn_moves(h[0], h[1], reward = reward)
279 
280         for h in self.history_state_pieces:
281             self.__learn_pieces(h[0], h[1], reward = reward)
282 
283         for h in self.opp_history_state_moves:
284             self.__learn_moves(h[0], h[1], reward = -reward)
285 
286         for h in self.opp_history_state_pieces:
287             self.__learn_pieces(h[0], h[1], reward = -reward)
288 
289         self.history_state_moves = list()
290         self.history_state_pieces = list()
291         self.opp_history_state_moves = list()
292         self.opp_history_state_pieces = list()
293         self.preGame = np.array([])
294 
295     def nowExploit(self,random_factor):
296         # print("hi there")
297         self.random_factor = random_factor
298 
299     def __learn_moves(self, state, ply, reward):
300         self.q_moves[state][ply] = self.q_moves[state][ply] + self.
    alpha * (reward - self.q_moves[state][ply])
301 
302     def __learn_pieces(self, state, ply, reward):
303         self.q_pieces[state][ply] = self.q_pieces[state][ply] +
    self.alpha * (reward - self.q_pieces[state][ply])
304 
305     def __row_to_q_moves(self, row):
306         temp_dict = dict()
307         check_list = [(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1,
    1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3), (3, 0), (3,
     1), (3, 2), (3, 3)]
308         for a, b in zip(check_list, row[3:19]):
309             if b != 91:
310                 temp_dict.update({a : b})
311         self.q_moves.update({row[1]: temp_dict})
312 
313     def __row_to_q_pieces(self, row):
```

```
314         temp_dict = dict()
315         check_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
     14, 15]
316         for a, b in zip(check_list, row[3:19]):
317             if b != 91:
318                 temp_dict.update({a : b})
319         self.q_pieces.update({row[1]: temp_dict})
320 
321     def save_to_db(self, version):
322         conn = sqlite3.connect('actors/reinforcement.db')
323         cursor = conn.cursor()
324 
325         print("Saving data to DB")
326 
327         for key, value in self.q_moves.items():
328             insertion = q_moves_to_insertion(hash_value = key,
    in_dict = value, version = version)
329             cursor.execute(insertion)
330 
331         for key, value in self.q_pieces.items():
332             insertion = q_pieces_to_insertion(hash_value = key,
    in_dict = value, version = version)
333             cursor.execute(insertion)
334 
335         print("Finished Saving data to DB")
336 
337         conn.commit()
338         conn.close()
339 
340     def read_from_db(self, version):
341         conn = sqlite3.connect('actors/reinforcement.db')
342         cursor = conn.cursor()
343 
344         print("Reading data from DB")
345 
346         statement = f"SELECT * FROM moves WHERE version = {version}
    "
347         cursor.execute(statement)
348         while True:
349             rows = cursor.fetchmany(10000)
350             if len(rows) == 0:
351                 break
352             for row in rows:
353                 self.__row_to_q_moves(row)
354 
355         print("first part")
356         cursor.close()
357         conn.close()
358         gc.collect()
359 
360 
361         conn = sqlite3.connect('actors/reinforcement.db')
362         cursor = conn.cursor()
363 
364         statement = f"SELECT * FROM pieces WHERE version = {version
    }"
365         cursor.execute(statement)
```

```
366         while True:
367             rows = cursor.fetchmany(10000)
368             if len(rows) == 0:
369                 break
370             for row in rows:
371                 self.__row_to_q_pieces(row)
372 
373         print("Finished Reading data from DB")
374 
375         conn.close()
376 
377 class ReinforcementPlay(Player):
378     """ReinforcementPlay player"""
379 
380     def __init__(self, quarto: Quarto) -> None:
381         super().__init__(quarto)
382         self.cells_row = [*range(0,4)]
383         self.all = [*range(0, 16, 1)]
384 
385         self.q_moves = dict()
386         self.q_pieces = dict()
387         self.hit_move = 0
388         self.no_hit_move = 0
389         self.hit_piece = 0
390         self.no_hit_piece = 0
391         self.place_call = 0
392         self.choose_call = 0
393         self.place_special = 0
394         self.choose_special = 0
395 
396         self.read_from_db(11)
397 
398 
399     def seek_symm_pieces(self, board):
400         hash_board = 0
401         found_symmetry = False
402         sym_type = ''
403         sym_count = 0
404         if not found_symmetry:
405             for r in range(1,4):
406                 copyBoard = symmRotation(board, r)
407                 hash_board_Symm = boardHash(copyBoard)
408                 if hash_board_Symm in self.q_pieces:
409                     hash_board = hash_board_Symm
410                     found_symmetry = True
411                     sym_type = 'rot'
412                     sym_count = r
413                     break
414 
415         if not found_symmetry:
416             copyBoard = symmFlipOrizontal(board)
417             hash_board_Symm = boardHash(copyBoard)
418             if hash_board_Symm in self.q_pieces:
419                 hash_board = hash_board_Symm
420                 found_symmetry = True
421                 sym_type='flip_h'
422 
```

```
423         if not found_symmetry:
424             copyBoard = symmFlipVertical(board)
425             hash_board_Symm = boardHash(copyBoard)
426             if hash_board_Symm in self.q_pieces:
427                 hash_board = hash_board_Symm
428                 found_symmetry = True
429                 sym_type='flip_v'
430 
431         if not found_symmetry:
432             copyBoard = symmFlipMid(board)
433             hash_board_Symm = boardHash(copyBoard)
434             if hash_board_Symm in self.q_pieces:
435                 hash_board = hash_board_Symm
436                 found_symmetry = True
437                 sym_type='flip_m'
438 
439         if not found_symmetry:
440             copyBoard = symmFlipInside(board)
441             hash_board_Symm = boardHash(copyBoard)
442             if hash_board_Symm in self.q_pieces:
443                 hash_board = hash_board_Symm
444                 found_symmetry = True
445                 sym_type='flip_i'
446 
447         return hash_board, sym_type, sym_count
448 
449     def seek_symm_moves(self, board):
450         hash_board = 0
451         found_symmetry = False
452         sym_type = ''
453         sym_count = 0
454         if not found_symmetry:
455             for r in range(1,4):
456                 copyBoard = symmRotation(board, r)
457                 hash_board_Symm = boardHash(copyBoard)
458                 if hash_board_Symm in self.q_moves:
459                     hash_board = hash_board_Symm
460                     found_symmetry = True
461                     sym_type = 'rot'
462                     sym_count = r
463                     break
464 
465         if not found_symmetry:
466             copyBoard = symmFlipOrizontal(board)
467             hash_board_Symm = boardHash(copyBoard)
468             if hash_board_Symm in self.q_moves:
469                 hash_board = hash_board_Symm
470                 found_symmetry = True
471                 sym_type='flip_h'
472 
473         if not found_symmetry:
474             copyBoard = symmFlipVertical(board)
475             hash_board_Symm = boardHash(copyBoard)
476             if hash_board_Symm in self.q_moves:
477                 hash_board = hash_board_Symm
478                 found_symmetry = True
479                 sym_type='flip_v'
```

```
480
481         if not found_symmetry:
482             copyBoard = symmFlipMid(board)
483             hash_board_Symm = boardHash(copyBoard)
484             if hash_board_Symm in self.q_moves:
485                 hash_board = hash_board_Symm
486                 found_symmetry = True
487                 sym_type='flip_m'
488
489         if not found_symmetry:
490             copyBoard = symmFlipInside(board)
491             hash_board_Symm = boardHash(copyBoard)
492             if hash_board_Symm in self.q_moves:
493                 hash_board = hash_board_Symm
494                 found_symmetry = True
495                 sym_type='flip_i'
496
497         return hash_board, sym_type, sym_count
498
499
500     def choose_piece(self) -> int:
501         self.choose_call += 1
502         board = self.get_game().get_board_status()
503
504         piece_choosen = None
505         hash_board = boardHash(board)
506
507         if hash_board not in self.q_pieces:
508             #look for symmetries
509             hash_board_temp, _, _ = self.seek_symm_pieces(board)
510             if hash_board_temp != 0:
511                 hash_board=hash_board_temp
512
513         poss_pieces = possible_pieces(board)
514         if len(poss_pieces) < 6:
515             self.choose_special += 1
516             # last 6 pieces
517             # this uses defensive logic
518             poss_moves = possible_moves(board)
519             for piece in poss_pieces:
520                 for move in poss_moves:
521                     new_board = deepcopy(board)
522                     new_board[move] = piece
523                     if check_current_player_winner(new_board):
524                         poss_pieces.remove(piece)
525                         break
526             # check if list is empty, otherwise reset the list
527             if len(poss_pieces) == 0:
528                 poss_pieces = possible_pieces(board)
529             return random.choice(poss_pieces)
530         else:
531             # first 10 pieces
532             # this uses the q_pieces hashmap
533             if hash_board not in self.q_pieces:
534                 piece_choosen = random.choice(poss_pieces)
535                 self.no_hit_piece += 1
536             else:
```

```
537                 potential_pieces = self.q_pieces[hash_board]
538                 piece_choosen = max(potential_pieces, key=
    potential_pieces.get)
539                 self.hit_piece += 1
540 
541         return piece_choosen
542 
543     def place_piece(self) -> tuple[int, int]:
544         self.place_call += 1
545         board = self.get_game().get_board_status()
546         piece_to_place = self.get_game().get_selected_piece()
547 
548         _possible_moves = possible_moves(board)
549         for move in _possible_moves:
550             new_board = deepcopy(board)
551             new_board[move] = piece_to_place
552             if check_current_player_winner(new_board):
553                 return move[1], move[0]
554 
555         check = possible_pieces(board)
556         if len(check) < 6:
557             self.place_special += 1
558             # last 6 moves
559             x, y = check_tris(board)
560             if x != -1:
561                 return y, x
562             else:
563                 return random.randint(0, 3), random.randint(0, 3)
564         else:
565             # first 10 moves
566             action_chosen = None
567 
568             xored_board = xorer(board, piece_to_place)
569             found_symm = False
570             hash_board = boardHash(xored_board)
571             if hash_board not in self.q_moves:
572                 #look for symmetries
573                 hash_board_temp, sym_type, sym_count = self.
    seek_symm_moves(xored_board)
574                 if hash_board_temp != 0:
575                     found_symm = True
576                     hash_board = hash_board_temp
577                     # print("sym")
578 
579             if found_symm:
580                 potential_actions_sym = self.q_moves[hash_board]
581                 action_chosen_sym = max(potential_actions_sym, key=
     potential_actions_sym.get)
582                 action_chosen = de_symm_move(action_chosen_sym,
    sym_type, sym_count)
583                 self.hit_move += 1
584                 return action_chosen[1], action_chosen[0]
585             else:
586                 # no sym
587                 if not hash_board in self.q_moves:
588                     # no hit
589                     # use defensive logic
```

```
590                     self.no_hit_move += 1
591                     x, y = check_tris(board)
592                     if x != -1:
593                         return y, x
594                     else:
595                         return random.randint(0, 3), random.randint
    (0, 3)
596                 else:
597                     # hit on hash not sym
598                     self.hit_move += 1
599                     potential_actions = self.q_moves[hash_board]
600                     action_chosen = max(potential_actions, key=
    potential_actions.get)
601                     return action_chosen[1], action_chosen[0]
602 
603     def printHits(self):
604         print("we have these nuts for moves:")
605         print(f"hits: {self.hit_move}")
606         print(f"NOhits: {self.no_hit_move}")
607         print(f"calls: {self.place_call}")
608         print(f"special: {self.place_special}")
609         print(f"q_moves length {len(self.q_moves)}")
610         print("we have these nuts for pieces:")
611         print(f"hits: {self.hit_piece}")
612         print(f"NOhits: {self.no_hit_piece}")
613         print(f"calls: {self.choose_call}")
614         print(f"special: {self.choose_special}")
615         print(f"q_moves length {len(self.q_pieces)}")
616 
617     def __row_to_q_moves(self, row):
618         temp_dict = dict()
619         check_list = [(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1,
    1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3), (3, 0), (3,
     1), (3, 2), (3, 3)]
620         for a, b in zip(check_list, row[3:19]):
621             if b != 91:
622                 temp_dict.update({a : b})
623         self.q_moves.update({row[1]: temp_dict})
624 
625     def __row_to_q_pieces(self, row):
626         temp_dict = dict()
627         check_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
     14, 15]
628         for a, b in zip(check_list, row[3:19]):
629             if b != 91:
630                 temp_dict.update({a : b})
631         self.q_pieces.update({row[1]: temp_dict})
632 
633     def read_from_db(self, version):
634         conn = sqlite3.connect('actors/reinforcement.db')
635         cursor = conn.cursor()
636 
637         print("Reading data from DB")
638 
639         statement = f"SELECT * FROM moves WHERE version = {version}
    "
640         cursor.execute(statement)
```

```
641         while True:
642             rows = cursor.fetchmany(10000)
643             if len(rows) == 0:
644                 break
645             for row in rows:
646                 self.__row_to_q_moves(row)
647 
648         print("first part")
649         cursor.close()
650         conn.close()
651         gc.collect()
652 
653         conn = sqlite3.connect('actors/reinforcement.db')
654         cursor = conn.cursor()
655 
656         statement = f"SELECT * FROM pieces WHERE version = {version
    }"
657         cursor.execute(statement)
658         while True:
659             rows = cursor.fetchmany(10000)
660             if len(rows) == 0:
661                 break
662             for row in rows:
663                 self.__row_to_q_pieces(row)
664 
665         print("Finished Reading data from DB")
666 
667         conn.close()
668 
669 class ReinforcementV2(Player):
670     """ReinforcementV2 player"""
671 
672     def __init__(self, quarto: Quarto, alpha=0.15, random_factor
    =0.8) -> None:
673         super().__init__(quarto)
674         self.cells_row = [*range(0,4)]
675         self.all = [*range(0, 16, 1)]
676         self.q_moves = dict()
677         self.q_pieces = dict()
678         self.alpha = alpha
679         self.random_factor = random_factor
680         self.history_state_moves = list()
681         self.history_state_pieces = list()
682         self.opp_history_state_moves = list()
683         self.opp_history_state_pieces = list()
684         self.hitratio = 0
685         self.preGame = np.array([])
686 
687         # self.read_from_db(5)
688 
689     def seek_symm_pieces(self, board):
690         hash_board = 0
691         found_symmetry = False
692         sym_type = ''
693         sym_count = 0
694         if not found_symmetry:
695             for r in range(1,4):
```

```
                copyBoard = symmRotation(board, r)
                hash_board_Symm = boardHash(copyBoard)
                if hash_board_Symm in self.q_pieces:
                    hash_board = hash_board_Symm
                    found_symmetry = True
                    sym_type = 'rot'
                    sym_count = r
                    break

        if not found_symmetry:
            copyBoard = symmFlipOrizontal(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_pieces:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_h'

        if not found_symmetry:
            copyBoard = symmFlipVertical(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_pieces:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_v'

        if not found_symmetry:
            copyBoard = symmFlipMid(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_pieces:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_m'

        if not found_symmetry:
            copyBoard = symmFlipInside(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_pieces:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_i'

        return hash_board, sym_type, sym_count

    def seek_symm_moves(self, board):
        hash_board = 0
        found_symmetry = False
        sym_type = ''
        sym_count = 0
        if not found_symmetry:
            for r in range(1,4):
                copyBoard = symmRotation(board, r)
                hash_board_Symm = boardHash(copyBoard)
                if hash_board_Symm in self.q_moves:
                    hash_board = hash_board_Symm
                    found_symmetry = True
                    sym_type = 'rot'
                    sym_count = r
```

```
                    break

        if not found_symmetry:
            copyBoard = symmFlipOrizontal(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_moves:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_h'

        if not found_symmetry:
            copyBoard = symmFlipVertical(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_moves:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_v'

        if not found_symmetry:
            copyBoard = symmFlipMid(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_moves:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_m'

        if not found_symmetry:
            copyBoard = symmFlipInside(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_moves:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_i'

        return hash_board, sym_type, sym_count

    def choose_piece(self) -> int:
        board = self.get_game().get_board_status()

        piece_choosen = None
        randomN = random.random()
        hash_board = boardHash(board)

        if hash_board not in self.q_pieces:
            #look for symmetries
            hash_board_temp, _, _ = self.seek_symm_pieces(board)
            if hash_board_temp != 0:
                hash_board=hash_board_temp

        poss_pieces = possible_pieces(board)

        if not hash_board in self.q_pieces:
            self.q_pieces.update({hash_board : {one_piece : 0 for
    one_piece in poss_pieces}})

        if len(poss_pieces) < 6:
            # last 6 pieces
```

```
            poss_moves = possible_moves(board)
            for piece in poss_pieces:
                for move in poss_moves:
                    new_board = deepcopy(board)
                    new_board[move] = piece
                    if check_current_player_winner(new_board):
                        poss_pieces.remove(piece)
                        break
            # check if list is empty, otherwise reset the list
            if len(poss_pieces) == 0:
                poss_pieces = possible_pieces(board)
            return random.choice(poss_pieces)
        else:
            # first 10 pieces
            if randomN < self.random_factor:
                piece_choosen = random.choice(poss_pieces)
            else:
                potential_pieces = self.q_pieces[hash_board]
                piece_choosen = max(potential_pieces, key=
potential_pieces.get)

        self.update_history_state_pieces(hash_board, piece_choosen)
        return piece_choosen

    def update_history_state_moves(self, hash_board, ply):
        self.history_state_moves.append((hash_board, ply))

    def update_history_state_pieces(self, hash_board, ply):
        self.history_state_pieces.append((hash_board, ply))

    def update_opp_history_state_moves(self, hash_board, ply):
        self.opp_history_state_moves.append((hash_board, ply))

    def update_opp_history_state_pieces(self, hash_board, ply):
        self.opp_history_state_pieces.append((hash_board, ply))

    def saveOpponentMoveAndPiece(self, piece_to_place):
        if len(self.preGame) == 0:
            return

        board=self.get_game().get_board_status()
        opponentMoveDirt = np.where(( board - self.preGame) != 0)
        opponentMove = (opponentMoveDirt[0][0], opponentMoveDirt
[1][0])
        hash_board = boardHash(board)

        if hash_board not in self.q_pieces:
            hash_board_temp, _, _ = self.seek_symm_pieces(board)
            if hash_board_temp != 0:
                hash_board=hash_board_temp

        if not hash_board in self.q_pieces:
            poss_pieces = possible_pieces(board)
            self.q_pieces.update({hash_board : {one_piece : 0 for
one_piece in poss_pieces}})

        self.update_opp_history_state_pieces(hash_board,
```

```
    piece_to_place)

        prePiece= board[opponentMove[0]][opponentMove[1]]

        xored_board = xorer(self.preGame, prePiece)
        found_symm = False
        hash_board = boardHash(xored_board)
        if hash_board not in self.q_moves:
            #look for symmetries
            hash_board_temp, sym_type, sym_count = self.
    seek_symm_moves(xored_board)
            if hash_board_temp != 0:
                found_symm = True
                hash_board = hash_board_temp

        if hash_board not in self.q_moves:
            poss_actions = possible_moves(self.preGame)
            self.q_moves.update({hash_board : {one_action : 0 for
    one_action in poss_actions}})

        if found_symm:
            opponentMoveSymm = de_symm_move(opponentMove, sym_type,
     -sym_count-4)
            self.update_opp_history_state_moves(hash_board,
    opponentMoveSymm)
        else:
            self.update_opp_history_state_moves(hash_board,
    opponentMove)

    def place_piece(self) -> tuple[int, int]:
        board = self.get_game().get_board_status()
        piece_to_place = self.get_game().get_selected_piece()

        check = possible_pieces(board)
        if len(check) < 6:
            # last 6 moves
            _possible_moves = possible_moves(board)
            for move in _possible_moves:
                new_board = deepcopy(board)
                new_board[move] = piece_to_place
                if check_current_player_winner(new_board):
                    return move[1], move[0]
            # print("here")
            x, y = check_tris(board)
            if x != -1:
                return y, x
            else:
                return random.randint(0, 3), random.randint(0, 3)

        action_chosen = None
        randomN = random.random()

        self.saveOpponentMoveAndPiece(piece_to_place)

        # board xor with the piece
        # seek sym on board xored
        xored_board = xorer(board, piece_to_place)
```

```
914         found_symm = False
915         hash_board = boardHash(xored_board)
916         if hash_board not in self.q_moves:
917             #look for symmetries
918             hash_board_temp, sym_type, sym_count = self.
    seek_symm_moves(xored_board)
919             if hash_board_temp != 0:
920                 found_symm = True
921                 hash_board = hash_board_temp
922                 # print("sym")
923 
924         if found_symm:
925             # print(f'found symm: {found_symm}')
926             if randomN < self.random_factor:
927                 # print(f'random: {randomN}')
928                 potential_actions_sym = list(self.q_moves[
    hash_board].keys())
929                 action_chosen_sym = random.choice(
    potential_actions_sym)
930             else:
931                 # print(f'not random: {randomN}')
932                 potential_actions_sym = self.q_moves[hash_board]
933                 action_chosen_sym = max(potential_actions_sym, key=
     potential_actions_sym.get)
934 
935             self.update_history_state_moves(hash_board,
    action_chosen_sym)
936             action_chosen = de_symm_move(action_chosen_sym,
    sym_type, sym_count)
937         else:
938             initial_poss_actions = possible_moves(board)
939             if not hash_board in self.q_moves:
940                 self.q_moves.update({hash_board : {one_action : 0
    for one_action in initial_poss_actions}})
941 
942             # print(f'not found symm: {found_symm}')
943             if randomN < self.random_factor:
944                 # print(f'random: {randomN}')
945                 action_chosen = random.choice(initial_poss_actions)
946             else:
947                 # print(f'not random: {randomN}')
948                 potential_actions = self.q_moves[hash_board]
949                 action_chosen = max(potential_actions, key=
    potential_actions.get)
950 
951             self.update_history_state_moves(hash_board,
    action_chosen)
952 
953 
954         copy_board = board
955         copy_board[action_chosen[0], action_chosen[1]] =
    piece_to_place
956         self.preGame = copy_board
957 
958         return action_chosen[1], action_chosen[0]
959 
960     def learn_all(self, reward):
```

```
961
962         for h in self.history_state_moves:
963             self.__learn_moves(h[0], h[1], reward = reward)
964
965         for h in self.history_state_pieces:
966             self.__learn_pieces(h[0], h[1], reward = reward)
967
968         for h in self.opp_history_state_moves:
969             self.__learn_moves(h[0], h[1], reward = -reward)
970
971         for h in self.opp_history_state_pieces:
972             self.__learn_pieces(h[0], h[1], reward = -reward)
973
974         self.history_state_moves = list()
975         self.history_state_pieces = list()
976         self.opp_history_state_moves = list()
977         self.opp_history_state_pieces = list()
978         self.preGame = np.array([])
979
980     def nowExploit(self,random_factor):
981         # print("hi there")
982         self.random_factor = random_factor
983
984     def __learn_moves(self, state, ply, reward):
985         self.q_moves[state][ply] = self.q_moves[state][ply] + self.
    alpha * (reward - self.q_moves[state][ply])
986
987     def __learn_pieces(self, state, ply, reward):
988         self.q_pieces[state][ply] = self.q_pieces[state][ply] +
    self.alpha * (reward - self.q_pieces[state][ply])
989
990     def __row_to_q_moves(self, row):
991         temp_dict = dict()
992         check_list = [(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1,
    1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3), (3, 0), (3,
     1), (3, 2), (3, 3)]
993         for a, b in zip(check_list, row[3:19]):
994             if b != 91:
995                 temp_dict.update({a : b})
996         self.q_moves.update({row[1]: temp_dict})
997
998     def __row_to_q_pieces(self, row):
999         temp_dict = dict()
1000        check_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
     14, 15]
1001        for a, b in zip(check_list, row[3:19]):
1002            if b != 91:
1003                temp_dict.update({a : b})
1004        self.q_pieces.update({row[1]: temp_dict})
1005
1006    def save_to_db(self, version):
1007        conn = sqlite3.connect('actors/reinforcement.db')
1008        cursor = conn.cursor()
1009
1010        print("Saving data to DB")
1011
1012        for key, value in self.q_moves.items():
```

```
            insertion = q_moves_to_insertion(hash_value = key, in_dict = value, version = version)
            cursor.execute(insertion)

        for key, value in self.q_pieces.items():
            insertion = q_pieces_to_insertion(hash_value = key, in_dict = value, version = version)
            cursor.execute(insertion)

        print("Finished Saving data to DB")

        conn.commit()
        conn.close()

    def read_from_db(self, version):
        conn = sqlite3.connect('actors/reinforcement.db')
        cursor = conn.cursor()

        print("Reading data from DB")

        statement = f"SELECT * FROM moves WHERE version = {version}"
        cursor.execute(statement)
        while True:
            rows = cursor.fetchmany(10000)
            if len(rows) == 0:
                break
            for row in rows:
                self.__row_to_q_moves(row)

        print("first part")
        cursor.close()
        conn.close()
        gc.collect()


        conn = sqlite3.connect('actors/reinforcement.db')
        cursor = conn.cursor()

        statement = f"SELECT * FROM pieces WHERE version = {version}"
        cursor.execute(statement)
        while True:
            rows = cursor.fetchmany(10000)
            if len(rows) == 0:
                break
            for row in rows:
                self.__row_to_q_pieces(row)

        print("Finished Reading data from DB")

        conn.close()


class ReinforcementPlay2(Player):
    """ReinforcementPlay player"""

```

```
    def __init__(self, quarto: Quarto) -> None:
        super().__init__(quarto)
        self.cells_row = [*range(0,4)]
        self.all = [*range(0, 16, 1)]

        self.q_moves = dict()
        self.q_pieces = dict()
        self.hit_move = 0
        self.no_hit_move = 0
        self.hit_piece = 0
        self.no_hit_piece = 0
        self.place_call = 0
        self.choose_call = 0
        self.place_special = 0
        self.choose_special = 0

        self.read_from_db(1)


    def seek_symm_pieces(self, board):
        hash_board = 0
        found_symmetry = False
        sym_type = ''
        sym_count = 0
        if not found_symmetry:
            for r in range(1,4):
                copyBoard = symmRotation(board, r)
                hash_board_Symm = boardHash(copyBoard)
                if hash_board_Symm in self.q_pieces:
                    hash_board = hash_board_Symm
                    found_symmetry = True
                    sym_type = 'rot'
                    sym_count = r
                    break

        if not found_symmetry:
            copyBoard = symmFlipOrizontal(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_pieces:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_h'

        if not found_symmetry:
            copyBoard = symmFlipVertical(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_pieces:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_v'

        if not found_symmetry:
            copyBoard = symmFlipMid(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_pieces:
                hash_board = hash_board_Symm
                found_symmetry = True
```

```
                sym_type='flip_m'

        if not found_symmetry:
            copyBoard = symmFlipInside(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_pieces:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_i'

        return hash_board, sym_type, sym_count

    def seek_symm_moves(self, board):
        hash_board = 0
        found_symmetry = False
        sym_type = ''
        sym_count = 0
        if not found_symmetry:
            for r in range(1,4):
                copyBoard = symmRotation(board, r)
                hash_board_Symm = boardHash(copyBoard)
                if hash_board_Symm in self.q_moves:
                    hash_board = hash_board_Symm
                    found_symmetry = True
                    sym_type = 'rot'
                    sym_count = r
                    break

        if not found_symmetry:
            copyBoard = symmFlipOrizontal(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_moves:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_h'

        if not found_symmetry:
            copyBoard = symmFlipVertical(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_moves:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_v'

        if not found_symmetry:
            copyBoard = symmFlipMid(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_moves:
                hash_board = hash_board_Symm
                found_symmetry = True
                sym_type='flip_m'

        if not found_symmetry:
            copyBoard = symmFlipInside(board)
            hash_board_Symm = boardHash(copyBoard)
            if hash_board_Symm in self.q_moves:
                hash_board = hash_board_Symm
```

```
                found_symmetry = True
                sym_type='flip_i'

        return hash_board, sym_type, sym_count


    def choose_piece(self) -> int:
        self.choose_call += 1
        board = self.get_game().get_board_status()

        piece_choosen = None
        hash_board = boardHash(board)

        if hash_board not in self.q_pieces:
            #look for symmetries
            hash_board_temp, _, _ = self.seek_symm_pieces(board)
            if hash_board_temp != 0:
                hash_board=hash_board_temp

        if hash_board not in self.q_pieces:
            poss_pieces = possible_pieces(board)
            piece_choosen = random.choice(poss_pieces)
            self.no_hit_piece += 1
        else:
            potential_pieces = self.q_pieces[hash_board]
            piece_choosen = max(potential_pieces, key=
potential_pieces.get)
            self.hit_piece += 1

        return piece_choosen

    def place_piece(self) -> tuple[int, int]:
        self.place_call += 1
        board = self.get_game().get_board_status()
        piece_to_place = self.get_game().get_selected_piece()

        _possible_moves = possible_moves(board)
        for move in _possible_moves:
            new_board = deepcopy(board)
            new_board[move] = piece_to_place
            if check_current_player_winner(new_board):
                return move[1], move[0]

        action_chosen = None

        xored_board = xorer(board, piece_to_place)
        found_symm = False
        hash_board = boardHash(xored_board)
        if hash_board not in self.q_moves:
            #look for symmetries
            hash_board_temp, sym_type, sym_count = self.
seek_symm_moves(xored_board)
            if hash_board_temp != 0:
                found_symm = True
                hash_board = hash_board_temp
                # print("sym")

```

```
        if found_symm:
            potential_actions_sym = self.q_moves[hash_board]
            action_chosen_sym = max(potential_actions_sym, key=
    potential_actions_sym.get)
            action_chosen = de_symm_move(action_chosen_sym,
    sym_type, sym_count)
            self.hit_move += 1
            return action_chosen[1], action_chosen[0]
        else:
            # no sym
            if not hash_board in self.q_moves:
                # no hit
                # use defensive logic
                self.no_hit_move += 1
                x, y = check_tris(board)
                if x != -1:
                    return y, x
                else:
                    return random.randint(0, 3), random.randint(0,
    3)
            else:
                # hit on hash not sym
                self.hit_move += 1
                potential_actions = self.q_moves[hash_board]
                action_chosen = max(potential_actions, key=
    potential_actions.get)
                return action_chosen[1], action_chosen[0]

    def printHits(self):
        print("we have these nuts for moves:")
        print(f"hits: {self.hit_move}")
        print(f"NOhits: {self.no_hit_move}")
        print(f"calls: {self.place_call}")
        print(f"special: {self.place_special}")
        print(f"q_moves length {len(self.q_moves)}")
        print("we have these nuts for pieces:")
        print(f"hits: {self.hit_piece}")
        print(f"NOhits: {self.no_hit_piece}")
        print(f"calls: {self.choose_call}")
        print(f"special: {self.choose_special}")
        print(f"q_moves length {len(self.q_pieces)}")

    def __row_to_q_moves(self, row):
        temp_dict = dict()
        check_list = [(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1,
    1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3), (3, 0), (3,
     1), (3, 2), (3, 3)]
        for a, b in zip(check_list, row[3:19]):
            if b != 91:
                temp_dict.update({a : b})
        self.q_moves.update({row[1]: temp_dict})

    def __row_to_q_pieces(self, row):
        temp_dict = dict()
        check_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
     14, 15]
        for a, b in zip(check_list, row[3:19]):
```

```
            if b != 91:
                temp_dict.update({a : b})
        self.q_pieces.update({row[1]: temp_dict})

    def read_from_db(self, version):
        conn = sqlite3.connect('actors/reinforcement.db')
        cursor = conn.cursor()

        print("Reading data from DB")

        statement = f"SELECT * FROM moves WHERE version = {version}
    "
        cursor.execute(statement)
        while True:
            rows = cursor.fetchmany(10000)
            if len(rows) == 0:
                break
            for row in rows:
                self.__row_to_q_moves(row)

        print("first part")
        cursor.close()
        conn.close()
        gc.collect()

        conn = sqlite3.connect('actors/reinforcement.db')
        cursor = conn.cursor()

        statement = f"SELECT * FROM pieces WHERE version = {version
    }"
        cursor.execute(statement)
        while True:
            rows = cursor.fetchmany(10000)
            if len(rows) == 0:
                break
            for row in rows:
                self.__row_to_q_pieces(row)

        print("Finished Reading data from DB")

        conn.close()
```

Listing 37: Reinforcement based player and the final agent

4.7.2 Min-max

```
from quarto import Quarto, Player
import numpy as np
from itertools import product
from utils import *

class MinMaxPlayer3(Player):
    def __init__(self, quarto: Quarto):
        super().__init__(quarto)
        self.max_depth = 2

    def choose_piece(self):

```

```
        board = self.get_game().get_board_status()
        possible_plies = possible_ply(board)

        evaluations = list()
        for ply in possible_plies:
            b = board.copy()
            b[ply[0]] = ply[1]
            eval = self.minmax(board = b, depth = self.max_depth, isMax = True, alpha = -float("inf"), beta = float("inf"))
            evaluations.append((ply[1], eval))

        all = dict()
        for k, v in evaluations:
            if k in all:
                if all[k] < v:
                    all[k] = v
                # all[k] = all[k] + v
            else:
                all.update({k: v})

        min_ply = min(all, key=all.get)

        print(f"min_ply piece: {min_ply}")

        return min_ply

    def place_piece(self):

        piece_to_place = self.get_game().get_selected_piece()
        board = self.get_game().get_board_status()

        possible_ply = list()
        boardSquares = list(product([*range(0,4)],repeat=2))
        for (x, y) in boardSquares:
            if board[x, y] == -1:
                possible_ply.append(((x, y), piece_to_place))

        evaluations = list()
        for ply in possible_ply:
            b = board.copy()
            b[ply[0]] = ply[1]
            eval = self.minmax(board = b, depth = self.max_depth, isMax = True, alpha = -float("inf"), beta = float("inf"))
            evaluations.append((ply, eval))

        max_ply = max(evaluations,key=lambda k: k[1])

        print(f"max_ply move: {max_ply}")

        return max_ply[0][0][1], max_ply[0][0][0]

    def minmax(self, board, isMax, depth, alpha, beta):

        # does this player win?
        if check_current_player_winner(board):
            if isMax:
                return 10000
```

```
            else:
                return -10000
        else:
            if check_if_board_full(board):
                return 0

        if depth == 0:
            return self.utility(board, isMax)

        if isMax:
            best_value = -float("inf")
            possible_plies = possible_ply(board)
            for ply in possible_plies:
                new_board = board.copy()
                new_board[ply[0]] = ply[1]
                value = self.minmax(board = new_board, isMax =
    False, depth = depth - 1, alpha = alpha, beta = beta)
                best_value = max(best_value, value)
                alpha = max(alpha, best_value)
                if beta <= alpha:
                    break
            return best_value
        else:
            best_value = float("inf")
            possible_plies = possible_ply(board)
            for ply in possible_plies:
                new_board = board.copy()
                new_board[ply[0]] = ply[1]
                value = self.minmax(board = new_board, isMax = True
    , depth = depth - 1, alpha = alpha, beta = beta)
                best_value = min(best_value, value)
                beta = min(beta, best_value)
                if beta <= alpha:
                    break
            return best_value

    def utility(self, int_board, isMax):
        score = 0

        binary_board = number_to_binary(int_board)

        # Number of completed rows/columns
        row = np.all(int_board != -1, axis = 0)
        col = np.all(int_board != -1, axis = 1)
        score += row.sum() * 10
        score += col.sum() * 10

        # Number of pieces with matching attributes
        magic = np.where(int_board != -1)
        for i, j in zip(magic[0], magic[1]):
            piece = binary_board[i][j]

            # check row
            for k in range(4):
                if k != j:
                    c = (piece == binary_board[i][k])
                    score += c.sum() * 3
```

```
123
124             # check col
125             for k in range(4):
126                 if k != i:
127                     c = (piece == binary_board[k][j])
128                     score += c.sum() * 3
129
130             # check if piece is on diagonal
131             if (i == j):
132                 for k in range(4):
133                     if k != j:
134                         c = (piece == binary_board[k][k])
135                         score += c.sum() * 3
136
137             if ((i + j) == 3):
138                 for k in range(4):
139                     if k != i and (3 - k) != j:
140                         c = (piece == binary_board[k][3 - k])
141                         score += c.sum() * 3
142
143         if isMax:
144             return score
145         else:
146             return -score
```

Listing 38: min-max algorithm

4.7.3 Utility functions

```
1  from collections import Counter
2  from copy import deepcopy
3  from itertools import combinations
4  import random
5  from itertools import product
6  import numpy as np
7  import hashlib
8
9  BOARD_SIDE = 4
10
11 all = [*range(0, 16, 1)]
12
13 piece_dict = {
14     -1: np.nan,
15     0: [0, 0, 0, 0],
16     1: [0, 0, 0, 1],
17     2: [0, 0, 1, 0],
18     3: [0, 0, 1, 1],
19     4: [0, 1, 0, 0],
20     5: [0, 1, 0, 1],
21     6: [0, 1, 1, 0],
22     7: [0, 1, 1, 1],
23     8: [1, 0, 0, 0],
24     9: [1, 0, 0, 1],
25     10: [1, 0, 1, 0],
26     11: [1, 0, 1, 1],
27     12: [1, 1, 0, 0],
28     13: [1, 1, 0, 1],
29     14: [1, 1, 1, 0],
```

```
    15: [1, 1, 1, 1]
}

def three_in_a_row(in_list) -> bool:
    test_list = deepcopy(in_list)
    count_empty = Counter(test_list)

    if count_empty[-1] != 1:
        return False

    test_list.remove(-1)

    possible_combinations = combinations(test_list, 2)

    xor_combinations = [(a^b) for a,b in possible_combinations]

    or_result = xor_combinations[0] | xor_combinations[1] |
    xor_combinations[2]

    if or_result == 15:
        return False
    else:
        return True

def check_tris(in_board) -> tuple[int, int]:
    board_test = in_board.tolist()
    three_pieces=[]
    # horizontal
    for x in range(4):
        if three_in_a_row(board_test[x]):
            three_pieces.append((x, board_test[x].index(-1)))

    right_diag = []
    left_diag = []

    # vertical and generate the 2 diagnonals
    for y in range(4):
        right_diag.append(board_test[y][y])
        left_diag.append(board_test[y][3-y])
        temp = board_test[:][y]
        if three_in_a_row(temp):
            three_pieces.append((temp.index(-1), y))

    # right diag
    if three_in_a_row(right_diag):
        three_pieces.append((right_diag.index(-1), right_diag.index
    (-1)))

    # left diag
    if three_in_a_row(left_diag):
        three_pieces.append((left_diag.index(-1), 3-left_diag.index
    (-1)))

    if len(three_pieces) > 0:
        return random.choice(three_pieces)
    else:
        return -1, -1
```

```
84
85  def possible_ply(board) -> list():
86      _possible_moves = possible_moves(board)
87      _possible_pieces = possible_pieces(board)
88      possible_plies = list(product(_possible_moves, _possible_pieces
        ))
89      random.shuffle(possible_plies)
90      return possible_plies
91
92  def possible_moves(board) -> list():
93      magic = np.where(board == -1)
94      return [mov for mov in zip(magic[0], magic[1])]
95
96  def possible_pieces(board) -> list():
97      flattened_board = sum(board.tolist(), [])
98      return list(set(all) - set(flattened_board))
99
100 def check_current_player_winner(in_board) -> bool:
101     board = number_to_binary(in_board)
102
103     # horizontal
104     hsum = np.sum(board, axis=1)
105     if BOARD_SIDE in hsum or 0 in hsum:
106         return True
107
108     # vertical
109     vsum = np.sum(board, axis=0)
110     if BOARD_SIDE in vsum or 0 in vsum:
111         return True
112
113     # diagonal
114     dsum1 = np.trace(board, axis1=0, axis2=1)
115     dsum2 = np.trace(np.fliplr(board), axis1=0, axis2=1)
116     if BOARD_SIDE in dsum1 or BOARD_SIDE in dsum2 or 0 in dsum1 or
        0 in dsum2:
117         return True
118
119     return False
120
121 def number_to_binary(board) -> np.ndarray:
122     binary_board = np.full(shape=(BOARD_SIDE, BOARD_SIDE, 4),
        fill_value=np.nan)
123     for j in range(0, 4):
124         for i in range(0, 4):
125             binary_board[i, j][:] = piece_dict[board[i, j]]
126     return binary_board
127
128 def check_if_board_full(board) -> bool:
129     for row in board:
130         for elem in row:
131             if elem == -1:
132                 return False
133     #print("tie")
134     return True
135
136 def piece_xorer(piece, xorer):
137     if piece != -1:
```

```
138         return piece^xorer
139     else:
140         return piece
141 
142 def boardHash(board):
143     m1 = hashlib.sha256()
144     m1.update(board.tobytes())
145     return m1.hexdigest()
146 
147 def xorer_hash(board, piece):
148     after_xor = xorer(board, piece)
149     return boardHash(after_xor)
150 
151 def xorer(board, piece_xor):
152     new_board = deepcopy(board)
153     vfunc = np.vectorize(piece_xorer)
154     xored = vfunc(new_board, piece_xor)
155     return xored
156 
157 def symmRotation(board, numRot) -> np.ndarray:
158     board_copy = deepcopy(board)
159     board_copy = np.rot90(board_copy, numRot)
160     return board_copy
161 
162 def symmFlipOrizontal(board) -> np.ndarray:
163     board_copy = deepcopy(board)
164     board_copy = np.fliplr(board_copy)
165     return board_copy
166 
167 def symmFlipVertical(board) -> np.ndarray:
168     board_copy = deepcopy(board)
169     board_copy = np.flipud(board_copy)
170     return board_copy
171 
172 def symmFlipMid(board) -> np.ndarray:
173     board_copy = deepcopy(board)
174     board_copy[0][1] = board[0][2]
175     board_copy[0][2] = board[0][1]
176     board_copy[3][2] = board[3][1]
177     board_copy[3][1] = board[3][2]
178 
179     board_copy[1][0] = board[2][0]
180     board_copy[2][0] = board[1][0]
181     board_copy[1][3] = board[2][3]
182     board_copy[2][3] = board[1][3]
183 
184     board_copy[1][1] = board[2][2]
185     board_copy[2][2] = board[1][1]
186     board_copy[1][2] = board[2][1]
187     board_copy[2][1] = board[1][2]
188     return board_copy
189 
190 def symmFlipInside(board) -> np.ndarray:
191     board_copy = deepcopy(board)
192     board_copy[0][0] = board[1][1]
193     board_copy[1][1] = board[0][0]
194     board_copy[0][1] = board[1][0]
```

```
195     board_copy[1][0] = board[0][1]
196 
197     board_copy[0][2] = board[1][3]
198     board_copy[1][3] = board[0][2]
199     board_copy[0][3] = board[1][2]
200     board_copy[1][2] = board[0][3]
201 
202     board_copy[2][0] = board[3][1]
203     board_copy[3][1] = board[2][0]
204     board_copy[2][1] = board[3][0]
205     board_copy[3][0] = board[2][1]
206 
207     board_copy[2][2] = board[3][3]
208     board_copy[3][3] = board[2][2]
209     board_copy[2][3] = board[3][2]
210     board_copy[3][2] = board[2][3]
211     return board_copy
212 
213 def de_symm_move(move, sym_typ, sym_count):
214 
215     board = np.ones(shape=(BOARD_SIDE, BOARD_SIDE), dtype=bool) *
        False
216     board[move] = True
217 
218     if sym_typ == 'rot':
219         board = symmRotation(board, 4 - sym_count)
220 
221     if sym_typ == 'flip_h':
222         board = symmFlipOrizontal(board)
223 
224     if sym_typ == 'flip_v':
225         board = symmFlipVertical(board)
226 
227     if sym_typ == 'flip_i':
228         board = symmFlipInside(board)
229 
230     if sym_typ == 'flip_m':
231         board = symmFlipMid(board)
232 
233     coordinate = np.where(board==True)
234 
235     return coordinate[0][0], coordinate[1][0]
236 
237 def q_moves_to_list(in_dict):
238     retval = list()
239     check_list = [(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1),
        (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3), (3, 0), (3, 1),
         (3, 2), (3, 3)]
240     for j in check_list:
241         if j in in_dict:
242             retval.append(in_dict[j])
243         else:
244             retval.append(91)
245     return retval
246 
247 def q_pieces_to_list(in_dict):
248     retval = list()
```

```
    check_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
    for j in check_list:
        if j in in_dict:
            retval.append(in_dict[j])
        else:
            retval.append(91)
    return retval

def q_moves_to_insertion(hash_value, in_dict, version) -> str:
    q = q_moves_to_list(in_dict)
    retval = "INSERT INTO moves (hash_value, version, '00', '01', '02', '03', '10', '11', '12', '13', '20', '21', '22', '23', '30', '31', '32', '33') VALUES " \
            f"('{hash_value}', {version}, {q[0]}, {q[1]}, {q[2]}, {q[3]}, {q[4]}, {q[5]}, {q[6]}, {q[7]}, {q[8]}, {q[9]}, {q[10]}, {q[11]}, {q[12]}, {q[13]}, {q[14]}, {q[15]})"
    return retval

def q_pieces_to_insertion(hash_value, in_dict, version) -> str:
    q = q_pieces_to_list(in_dict)
    retval = "INSERT INTO pieces (hash_value, version, '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15') VALUES " \
            f"('{hash_value}', {version}, {q[0]}, {q[1]}, {q[2]}, {q[3]}, {q[4]}, {q[5]}, {q[6]}, {q[7]}, {q[8]}, {q[9]}, {q[10]}, {q[11]}, {q[12]}, {q[13]}, {q[14]}, {q[15]})"
    return retval



def check_horizontal(board, board_len_test) -> bool:
    for i in range(board_len_test):
        high_values = [
            elem for elem in board[i] if elem >= 0 and (elem>>3)&1
        ]
        coloured_values = [
            elem for elem in board[i] if elem >= 0 and (elem>>2)&1
        ]
        solid_values = [
            elem for elem in board[i] if elem >= 0 and (elem>>1)&1
        ]
        square_values = [
            elem for elem in board[i] if elem >= 0 and (elem>>0)&1
        ]
        low_values = [
            elem for elem in board[i] if elem >= 0 and not (elem>>3)&1
        ]
        noncolor_values = [
            elem for elem in board[i] if elem >= 0 and not (elem>>2)&1
        ]
        hollow_values = [
            elem for elem in board[i] if elem >= 0 and not (elem>>1)&1
        ]
```

```
        circle_values = [
            elem for elem in board[i] if elem >= 0 and not (elem
    >>0)&1
        ]
        if len(high_values) == board_len_test or len(
                coloured_values
        ) == board_len_test or len(solid_values) == board_len_test
    or len(
                square_values) == board_len_test or len(low_values)
     == board_len_test or len(
                    noncolor_values) == board_len_test or len(
                        hollow_values) == board_len_test or len(
                            circle_values) == board_len_test:
            return True
    return False

def check_vertical(board, board_len_test)-> bool:
    for i in range(board_len_test):
        high_values = [
            elem for elem in board[:, i] if elem >= 0 and (elem>>3)
    &1
        ]
        coloured_values = [
            elem for elem in board[:, i] if elem >= 0 and (elem>>2)
    &1
        ]
        solid_values = [
            elem for elem in board[:, i] if elem >= 0 and (elem>>1)
    &1
        ]
        square_values = [
            elem for elem in board[:, i] if elem >= 0 and (elem>>0)
    &1
        ]
        low_values = [
            elem for elem in board[:, i] if elem >= 0 and not (elem
    >>3)&1
        ]
        noncolor_values = [
            elem for elem in board[:, i] if elem >= 0 and not (elem
    >>2)&1
        ]
        hollow_values = [
            elem for elem in board[:, i] if elem >= 0 and not (elem
    >>1)&1
        ]
        circle_values = [
            elem for elem in board[:, i] if elem >= 0 and not (elem
    >>0)&1
        ]
        if len(high_values) == board_len_test or len(
                coloured_values
        ) == board_len_test or len(solid_values) == board_len_test
    or len(
                square_values) == board_len_test or len(low_values)
     == board_len_test or len(
                    noncolor_values) == board_len_test or len(
```

```
                        hollow_values) == board_len_test or len(
                            circle_values) == board_len_test:
            return True
    return False

def check_diagonal(board, board_len_test)-> bool:
    high_values = []
    coloured_values = []
    solid_values = []
    square_values = []
    low_values = []
    noncolor_values = []
    hollow_values = []
    circle_values = []
    for i in range(board_len_test):
        if board[i, i] < 0:
            break
        if (board[i, i]>>3)&1:
            high_values.append(board[i, i])
        else:
            low_values.append(board[i, i])
        if (board[i, i]>>2)&1:
            coloured_values.append(board[i, i])
        else:
            noncolor_values.append(board[i, i])
        if (board[i, i]>>1)&1:
            solid_values.append(board[i, i])
        else:
            hollow_values.append(board[i, i])
        if (board[i, i]>>0)&1:
            square_values.append(board[i, i])
        else:
            circle_values.append(board[i, i])
    if len(high_values) == board_len_test or len(coloured_values)
    == board_len_test or len(
            solid_values) == board_len_test or len(square_values)
    == board_len_test or len(
                low_values
            ) == board_len_test or len(noncolor_values) ==
    board_len_test or len(
                hollow_values) == board_len_test or len(
    circle_values) == board_len_test:
        return True
    high_values = []
    coloured_values = []
    solid_values = []
    square_values = []
    low_values = []
    noncolor_values = []
    hollow_values = []
    circle_values = []
    for i in range(board_len_test):
        if board[i, board_len_test - 1 - i] < 0:
            break
        if (board[i, board_len_test - 1 - i]>>3)&1:
            high_values.append(board[i, board_len_test - 1 - i])
        else:
```

```
            low_values.append(board[i, board_len_test - 1 - i])
        if (board[i, board_len_test - 1 - i]>>2)&1:
            coloured_values.append(
                board[i, board_len_test - 1 - i])
        else:
            noncolor_values.append(
                board[i, board_len_test - 1 - i])
        if (board[i, board_len_test - 1 - i]>>1)&1:
            solid_values.append(board[i, board_len_test - 1 - i])
        else:
            hollow_values.append(board[i, board_len_test - 1 - i])
        if (board[i, board_len_test - 1 - i]>>0)&1:
            square_values.append(board[i, board_len_test - 1 - i])
        else:
            circle_values.append(board[i, board_len_test - 1 - i])
    if len(high_values) == board_len_test or len(coloured_values)
    == board_len_test or len(
            solid_values) == board_len_test or len(square_values)
    == board_len_test or len(
                low_values
            ) == board_len_test or len(noncolor_values) ==
    board_len_test or len(
                hollow_values) == board_len_test or len(
    circle_values) == board_len_test:
        return True
    return False
```

Listing 39: Utility functions

4.7.4 main

```
# Free for personal or classroom use; see 'LICENSE.md' for details.
# https://github.com/squillero/computational-intelligence

import logging
import argparse
import random
from quarto import Quarto
from actors.rule_based_players import *
from actors.minmax3 import MinMaxPlayer3
from actors.reinforcement_bases_players  import *
# from actors.final_agent import ReinforcementPlay, ReinforcementV2
import time
from actors import *
from players import SafePlayer

def evaluate(player0, player1):
    win = 0
    games_won = 0
    games_ties = 0
    game = Quarto()

    for _ in range(0, 5):
        if _ % 1000 == 0:
            print(_)

        game.reset()
        game.set_players((player0(game), player1(game)))
```

```
        winner = game.run()

        if winner == 0:
            win += 1
            games_won += 1
        if winner == 1:
            games_won += 1
        if winner == -1:
            games_ties += 1

        game.reset()
        game.set_players((player1(game), player0(game)))
        winner = game.run()

        if winner == 1:
            win += 1
            games_won += 1
        if winner == 0:
            games_won += 1
        if winner == -1:
            games_ties += 1

        print(_)

    print(f"{player0.__name__} won: {win} / {games_won} ({win/
    games_won}); ties {games_ties}")

def trainReinforcement(player0):

    version = 0
    game = Quarto()

    player0_withgame = player0(game)
    player1_withgame = RandomPlayer(game)

    for _ in range(0, 10_000_000):

        if _ % 50_000 == 0 and _ != 0:
            player0_withgame.save_to_db(version)
            version += 1

        if _ % 1000 == 0:
            print(_)

        game.reset()

        game.set_players((player0_withgame, player1_withgame))
        winner = game.run()

        if winner == 0:
            player0_withgame.learn_all(1)
        if winner == 1:
            player0_withgame.learn_all(-1)
        if winner == -1:
            player0_withgame.learn_all(1)

        game.reset()
```

```
84
85          game.set_players((player1_withgame, player0_withgame))
86          winner = game.run()
87
88          if winner == 1:
89              player0_withgame.learn_all(1)
90          if winner == 0:
91              player0_withgame.learn_all(-1)
92          if winner == -1:
93              player0_withgame.learn_all(1)
94
95          # print(_)
96
97  def evaluateReinforcementPlay(player0, player1):
98      win = 0
99      games_won = 0
100     games_ties = 0
101     version = 0
102
103     game = Quarto()
104
105     player0_withgame = player0(game)
106     player1_withgame = player1(game)
107
108     for _ in range(0, 5000):
109
110         if _ % 1000 == 0:
111             print(_)
112
113         game.reset()
114
115         game.set_players((player0_withgame, player1_withgame))
116         winner = game.run()
117
118         if winner == 0:
119             win += 1
120             games_won += 1
121         if winner == 1:
122             games_won += 1
123         if winner == -1:
124             games_ties += 1
125
126         game.reset()
127
128         game.set_players((player1_withgame, player0_withgame))
129         winner = game.run()
130
131         if winner == 1:
132             win += 1
133             games_won += 1
134         if winner == 0:
135             games_won += 1
136         if winner == -1:
137             games_ties += 1
138
139     print(f"{player0.__name__} won: {win} / {games_won} ({win/
    games_won}); ties {games_ties}")
```

```
    player0_withgame.printHits()

def main():
    random.seed(42)
    start = time.time()
    # e = evaluate(MinMaxPlayer3, DefensivePlayer)
    # e = trainReinforcement(ReinforcementV1)
    evaluateReinforcementPlay(ReinforcementPlay, DefensivePlayer)
    end = time.time()

    print(end - start)

    # print(e)
    # game = Quarto()
    # game.set_players((MinMaxPlayer3(game), DefensivePlayer(game))
    )
    # winner = game.run()
    # logging.warning(f"main: Winner: player {winner}")
    # print(game.get_board_status())
    # game.print_play_list()


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-v', '--verbose', action='count', default
    =0, help='increase log verbosity')
    parser.add_argument('-d',
                        '--debug',
                        action='store_const',
                        dest='verbose',
                        const=2,
                        help='log debug messages (same as -vv)')
    args = parser.parse_args()

    if args.verbose == 0:
        logging.getLogger().setLevel(level=logging.WARNING)
    elif args.verbose == 1:
        logging.getLogger().setLevel(level=logging.INFO)
    elif args.verbose == 2:
        logging.getLogger().setLevel(level=logging.DEBUG)

    main()
```

Listing 40: Main

References

[1] Definition of a heuristic function wiki page

[2] Nim explained from wiki page

[3] Wouter M. Koolen from Cakes talk *Quarto!*

[4] Michael Neumann, et al., *An Artificial Intelligence for the Board Game 'Quarto!' in Java*

[5] Zhou, et al., *Analysis of SDN Attack and Defense Strategy Based on Zero-Sum Game* in Advances in Brain Inspired Cognitive Systems

[6] This blog present an intuition for the so-called *Mirror strategy*

[7] Most of the consulted documentation is from the professor's lectures and the laboratories material provided in this repo

[8] Numerous visits to Stack Overflow played a significant contribution during the development phase