Computational Intelligence

Thomas Osorio - s307107

January 2023

Contents

Chapter 1

Lab 1 - Set covering Problem using Single-state Methods

In the first weeks of the Computational Intelligence course, we learned about the tree search and some algorithms to solve problems using this methodology. The first lab was a practical exercise to implement different algorithms in python to solve the set covering problem.

- link: https://github.com/tonatiu92/Computational-Intelligence-CI-2022-01URROV-s307107/tree/main/lab1
- No group to declare
- source: Squillero github gxutils file and 8 puzzles

1.1 Task

Given a number N and some lists of integers $P = (L_0, L_1, L_2, ..., L_n)$, determine, if possible,$S = (L_{s_0}, L_{s_1}, L_{s_2}, ..., L_{s_n})$ such that each number between 0 and $N - 1$ appears in at least one list

$$\forall n \in [0, N-1] \exists i : n \in L_{s_i}$$

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

1.2 Solutions - Before Review

General Class

This code shows the definition of a state object to use in the tree search for set covering problem

```
class State:
    """
    The state at a moment of the search
    defined by an array
    """
    def __init__(self, arr=[]):
       self._array = arr.copy()

    def __hash__(self):
        return hash(np.array(self._array).tobytes())

    def __eq__(self, other):
        return sum(len(_) for _ in self._array) ==  sum(len(_) for
    _ in other._array)

    def __lt__(self, other):
        return sum(len(_) for _ in self._array) < sum(len(_) for _
    in other._array)
    def __str__(self):
        return str(self)

    @property
    def array(self):
        return self._array
    @array.setter
    def array(self,value):
        self._array = value
```

Search function

The search function is the main function of the Tree search.

```
def goal_test(state,goal):
    return state == goal
def possible_actions(all_lists,covering):
    return sorted([array for array in all_lists if len(set(array).
    difference(covering)) > 0], key=lambda l: len(l) )
```

```
def result(state, a):
    array = state.array.copy()
    array.append(a)
    return(State(array))

from gx_utils  import PriorityQueue

def possible_actionsV2(all_lists,covering):
    return [array for array in all_lists if len(set(array).
    difference(covering)) > 0]

logging.basicConfig(format="%(message)s", level=logging.INFO)
def searchV2(N,initial_state,goal_test,parent_state,state_cost,
    priority_function,unit_cost):
    """
    The search function
    """
    state = initial_state
    parent_state[state] = None
    state_cost[state] = 0
    solutions = []
    covering = set()
    step = 0
    all_lists = sorted(problem(N,seed=42), key=lambda l: len(l))
    while state is not None and  covering != goal_test:
        frontier = PriorityQueue()
        for a in possible_actionsV2(all_lists,covering):
                new_state = result(state, a)
                cost = unit_cost(new_state)
                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))
                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
        if frontier:
            state = frontier.pop()
            step = step + 1
            solutions.append(state.array)
            covering |= set([el for a in state.array for el in a])
        else:
            state = None
    path = list()
    s = state
    i = 0
    while s:
        path.append(s)
        s = parent_state[s]
    logging.info(f"NEW SOL: Found a solution in {step} nodes; ")
    return state.array,list(reversed(path))
```

Algortihms

I choose to look the results for the four main algorithms that we learn in class: greedy best first, A*, breath first and depth first

```
def greedy_first(N):
    def h(state:State):
        goal = set(range(N))
        return len(goal.difference(set([el for array in state.array
     for el in array])))
    final = search(
        N,
        State(),
        goal_test=goal_test,
        parent_state=dict(),
        state_cost=dict(),
        priority_function=lambda s: h(s) ,
        unit_cost=lambda a: 1
    )
    logging.info(
             f"gready_best_first first solution for N={N}: w={sum(
    len(_) for _  in final[0])} (bloat={(sum(len(_) for _ in final
    [0])-N)/N*100:.0f}%)"
        )
logging.getLogger().setLevel(logging.INFO)
for N in [5,10,20,100]:
    greedy_first(N)

def a_star(N):
    def h(state:State):
        goal = set(range(N))
        return len(goal.difference(set([el for array in state.array
     for el in array])))
    state_cost = dict()
    final = search(
        N,
        State(),
        goal_test=goal_test,
        parent_state=dict(),
        state_cost=state_cost,
        priority_function=lambda s: h(s) + state_cost[s] ,
        unit_cost=lambda a: len(a)
    )
    logging.info(
             f"gready_best_first first solution for N={N}: w={sum(
    len(_) for _  in final[0])} (bloat={(sum(len(_) for _ in final
    [0])-N)/N*100:.0f}%)"
        )
logging.getLogger().setLevel(logging.INFO)
for N in [5,10,20]:
    a_star(N)

def breadth_first(N):
    def h(state:State):
        goal = set(range(N))
        return len(goal.difference(set([el for array in state.array
     for el in array])))
    state_cost = dict()
```

```
    final = search(
        N,
        State(),
        goal_test=goal_test,
        parent_state=dict(),
        state_cost=state_cost,
        priority_function=lambda s: len(state_cost),
        unit_cost=lambda a: 1
    )
    logging.info(
            f"gready_best_first first solution for N={N}: w={sum(
    len(_) for _  in final[0])} (bloat={(sum(len(_) for _ in final
    [0])-N)/N*100:.0f}%)"
        )

logging.getLogger().setLevel(logging.INFO)
for N in [5,10,20]:
    breadth_first(N)

def depth_first(N):
    INITIAL_STATE = State()
    parent_state = dict()
    state_cost = dict()
    covering = set()
    goal  =  set(range(N))


    final = searchV2(
        N,
        INITIAL_STATE,
        goal_test=goal,
        parent_state=parent_state,
        state_cost=state_cost,
        priority_function=lambda s: -len(state_cost) ,
        unit_cost=lambda a: 1
    )
    logging.info(
            f"breath_first first solution for N={N}: w={sum(len(_)
    for _  in final[0])} (bloat={(sum(len(_) for _ in final[0])-N)/
    N*100:.0f}%)"
        )
logging.getLogger().setLevel(logging.INFO)
for N in [5,10,20]:
    depth_first(N)
```

1.3 Results - Before Review

1.3.1 SOLUTION 1: THE BEST

Greedy_first

We get this solution

- NEW SOL: Found a solution in 3 nodes;

  - gready_best_first first solution for N=5: w=5 (bloat=0%)
- NEW SOL: Found a solution in 3 nodes;
  - gready_best_first first solution for N=10: w=10 (bloat=0%)
- NEW SOL: Found a solution in 4 nodes;
  - gready_best_first first solution for N=20: w=28 (bloat=40%)
- NEW SOL: Found a solution in 5 nodes;
  - gready_best_first first solution for N=100: w=192 (bloat=92%)
- NEW SOL: Found a solution in 7 nodes;
  - gready_best_first first solution for N=500: w=1320 (bloat=164%)
- NEW SOL: Found a solution in 8 nodes;
  - gready_best_first first solution for N=1000: w=2893 (bloat=189%)

1.3.2 SOLUTION 2:

A*

- NEW SOL: Found a solution in 5 nodes;
  - A* first solution for N=5: w=5 (bloat=0%)
- NEW SOL: Found a solution in 6 nodes;
  - A* first solution for N=10: w=11 (bloat=10%)
- NEW SOL: Found a solution in 7 nodes;
  - A* first solution for N=20: w=28 (bloat=40%)
- NEW SOL: Found a solution in 13 nodes;
  - A* first solution for N=100: w=230 (bloat=130%)
- NEW SOL: Found a solution in 20 nodes;
  - A* first solution for N=500: w=1828 (bloat=266%)
- NEW SOL: Found a solution in 23 nodes;
  - A* first solution for N=1000: w=4130 (bloat=313%)

1.3.3 SOLUTION 3:

Breath first

- NEW SOL: Found a solution in 6 steps; visited 40 nodes
  - breath_first first solution for N=5: w=5 (bloat=0%)
- NEW SOL: Found a solution in 8 steps; visited 267 nodes
  - breath_first first solution for N=10: w=13 (bloat=30%)
- NEW SOL: Found a solution in 13 steps; visited 304 nodes
  - breath_first first solution for N=20: w=46 (bloat=130%)
- NEW SOL: Found a solution in 20 steps; visited 7,270 nodes
  - breath_first first solution for N=100: w=332 (bloat=232%)
- NEW SOL: Found a solution in 25 steps; visited 40,366 nodes
  - breath_first first solution for N=500: w=2162 (bloat=332%)
- NEW SOL: Found a solution in 27 steps; visited 87,907 nodes
  - breath_first first solution for N=1000: w=4652 (bloat=365%)

1.3.4 Solution 4

Depth first

- NEW SOL: Found a solution in 5 steps; visited 47 nodes
  - breath_first first solution for N=5: w=8 (bloat=60%)
- NEW SOL: Found a solution in 5 steps; visited 119 nodes
  - breath_first first solution for N=10: w=19 (bloat=90%)
- NEW SOL: Found a solution in 8 steps; visited 159 nodes
  - breath_first first solution for N=20: w=57 (bloat=185%)
- NEW SOL: Found a solution in 10 steps; visited 3,123 nodes
  - breath_first first solution for N=100: w=379 (bloat=279%)
- NEW SOL: Found a solution in 11 steps; visited 16,751 nodes
  - breath_first first solution for N=500: w=2044 (bloat=309%)
- NEW SOL: Found a solution in 14 steps; visited 40,987 nodes
  - breath_first first solution for N=1000: w=5242 (bloat=424%)

1.4 Reviews

1.4.1 Review 1 by GiuseppeEsposito98

- instead of using lists and set, numpy.ndarrays are better for efficiency purposes: python lists are hetereogeneous and non-contiguous in memory, on the other hand numpy.ndarrays are homogeneous and contiguous so each access is easier. Moreover lumpy package can parallelize computations. In the end the unpack of the different lists would be better computed with reshape and you can also use numpy.unique() in order to compute the unique values of an numpy.ndarray.
- more comments and also the specifications of the type in the function definition would be appreciated to make the code more human readable.
- The readme should contain also the applied methodology and the changes you did on the general searching algorithm
- In the end we were supposed to find the optimal solution of the problem for the given values of N but from N = 20 on we got a lower w. This could be due by the fact that the frontier is initialised at each cycle of the while which means that it will always find a local solution instead of the global one, nevertheless it scales well for high N but probably it does not find the optimal solution.

1.4.2 Review 2 by Riccardo Musumarra

- I appreciate the effort in showing the comparison between different search algorithms. Yet, the results are inconsistent with the algorithm said to have been used. For instance, Greedy performs better than A* and Breadth-first for N=100, which would not be possible since Breadth-first and A* always find the optimal solution, if exists. You should check your cost and heuristic functions.

After Review

The main problem of my solution, was that my frontier was created at each iteration, therefore I wasn't searching for an optimal solution but a local one. It leads to good results but not answer exactly the task. I wrote my frontier at this position in the code because it was taking too much time for the big Ns. Thanks to the professor correction, the reviews received and the reviews I gave, I understood the main goal of this Lab and I improved my code.

My first action was to put my frontier outside the loop, unfortunately it leads to chaotic results because the algorithms was returning wrong solutions with always only one array inside the solution list. After looking at the frontier content, I observed that I made a big mistake.

At each iteration, I was creating new states with a new cost but after the first iteration, my frontier wasn't empty any more and take in account the precedent state I looked for. Regarding my search methodology, I wasn't going through every path. At each iteration, I was trying to look at a state and starting from this state I was searching for the optimal local solution. On the other hand, my frontier was filling with all the states analyzed. As my goal test function was looking for the state already covered, it crash very quickly because the covered array analyzed by the search algorithm reach the goal very quickly.

ITERATION 1
FRONTIER [(2, [[0]]), (3, [[1]]), (4, [[4]]), (5, [[2]]), (6, [[3]]), (7, [[1, 3]]), (8, [[0, 1]]), (9, [[0, 2]]), (10, [[2, 4]]), (11, [[2, 3]])]
STATE AFTER ITERATION 1: [[0]] covered 0

ITERATION 2
FRONTIER [(3, [[1]]), (5, [[2]]), (4, [[4]]), (9, [[0, 2]]), (6, [[3]]), (7, [[1, 3]]), (8, [[0, 1]]), (11, [[2, 3]]), (10, [[2, 4]]), (12, [[0], [1]]), (13, [[0], [4]]), (14, [[0], [2]]), (15, [[0], [3]]), (16, [[0], [1, 3]]), (17, [[0], [0, 1]]), (18, [[0], [0, 2]]), (19, [[0], [2, 4]]), (20, [[0], [2, 4]]), (21, [[0], [0, 1]]), (22, [[0], [2, 3]])]
STATE AFTER ITERATION 2: [[1]] covered 0, 1

To correct my algorithm, I am now comparing my goal with the actual state and not the covered states values. The code below is the new one. Moreover I had also some errors with the State class with the eq() and lt() functions. It wasn't comparing well the states. That is why I made some modification. Finally, I also take in account what my colleagues wrote about numpy array and try to use it for operation across my algorithm

```
class State:
    """
    The state at a moment of the search
    defined by an array
    """
    def __init__(self, arr=[]):
       self._array = arr

    def __hash__(self):
        return hash(bytes(np.array(self._array)))

    def __eq__(self, other):
        return (bytes(np.array(self._array)) == bytes(np.array(
    other._array))) & (([sum(_) for _ in self._array]) ==([sum(_)
    for _ in other._array]))

    def __lt__(self, other):
        return ([sum(_) for _ in self._array]) < ([sum(_) for _ in
    other._array])

    def __str__(self):
```

```
        return str(self._array)

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

    @property
    def array(self):
        return self._array
    @array.setter
    def array(self,value):
        self._array = value


def possible_actions(all_lists: list, covering: set):
    """
    Description: return all the possibles actions sorted according
    each array

    @args:
    all_lists [list]: the search space
    covered [set]: the numbers already covered
    """
    return sorted([array for array in all_lists if len(set(array).
    difference(covering)) > 0], key=lambda l: len(l) )

def goal_test(state: State(),goal: set):
    """
    Description: testing if we already found every number

    @args;
    covered [set]: the numbers already covered
    goal [set]: the numbers to found
    """
    return set([el for array in state.array for el in array]) ==
    goal

def result(state: State, a:list):
    array = copy.deepcopy(state)._array
    array.append(a)
    return State(array)

def search(
    N: int,
    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
```

```
    covered = set()
    all_lists = sorted(problem(N,seed=42), key=lambda l: len(l))
    step = 0
    #print(f"SEARCH SPACE {search_space}")
    while state is not None and not goal_test(state, set(range(N)))
    :
        for a in possible_actions(all_lists,covered):
            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))
            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
        if frontier:
            state = frontier.pop()
            step = step + 1
        else:
            state = None
            return "No solution"

    path = list()
    s = state
    i = 0
    while s:
        path.append(s)
        s = parent_state[s]
    if N <=20:
        logging.info(f"NEW SOL: Found a solution in {step} nodes; {
    state.__str__()}")
    else:
        logging.info(f"NEW SOL: Found a solution in {step} nodes; "
    )
    return state.array,list(reversed(path))

def greedy_first(N):
    def h(state:State):
        goal = set(range(N))
        return len(goal.difference(set([el for array in state.array
     for el in array])))
    final = search(
        N,
        State(),
        goal_test=goal_test,
        parent_state=dict(),
        state_cost=dict(),
        priority_function=lambda s: h(s) ,
        unit_cost=lambda a: 1
    )
    logging.info(
            f"gready_best_first first solution for N={N}: w={sum(
```

```
    len(_) for _  in final[0])} (bloat={(sum(len(_) for _ in final
    [0])-N)/N*100:.0f}%)"
        )
logging.getLogger().setLevel(logging.INFO)
for N in [5,10,20,100]:
    greedy_first(N)

def a_star(N):
    def h(state:State):
        goal = set(range(N))
        return len(goal.difference(set([el for array in state.array
     for el in array])))
    state_cost = dict()
    final = search(
        N,
        State(),
        goal_test=goal_test,
        parent_state=dict(),
        state_cost=state_cost,
        priority_function=lambda s: h(s) + state_cost[s] ,
        unit_cost=lambda a: len(a)
    )
    logging.info(
            f"gready_best_first first solution for N={N}: w={sum(
    len(_) for _  in final[0])} (bloat={(sum(len(_) for _ in final
    [0])-N)/N*100:.0f}%)"
        )
logging.getLogger().setLevel(logging.INFO)
for N in [5,10,20]:
    a_star(N)


def breadth_first(N):
    def h(state:State):
        goal = set(range(N))
        return len(goal.difference(set([el for array in state.array
     for el in array])))
    state_cost = dict()
    final = search(
        N,
        State(),
        goal_test=goal_test,
        parent_state=dict(),
        state_cost=state_cost,
        priority_function=lambda s: len(state_cost),
        unit_cost=lambda a: 1
    )
    logging.info(
            f"gready_best_first first solution for N={N}: w={sum(
    len(_) for _  in final[0])} (bloat={(sum(len(_) for _ in final
    [0])-N)/N*100:.0f}%)"
        )

logging.getLogger().setLevel(logging.INFO)
for N in [5,10,20]:
    breadth_first(N)

```

```
172
173 def depth_first(N):
174     INITIAL_STATE = State()
175     parent_state = dict()
176     state_cost = dict()
177     covering = set()
178     goal  =  set(range(N))
179
180
181     final = searchV2(
182         N,
183         INITIAL_STATE,
184         goal_test=goal,
185         parent_state=parent_state,
186         state_cost=state_cost,
187         priority_function=lambda s: -len(state_cost) ,
188         unit_cost=lambda a: 1
189     )
190     logging.info(
191             f"breath_first first solution for N={N}: w={sum(len(_)
    for _  in final[0])} (bloat={(sum(len(_) for _ in final[0])-N)/
    N*100:.0f}%)"
192         )
193 logging.getLogger().setLevel(logging.INFO)
194 for N in [5,10,20]:
195     depth_first(N)
```

1.5 Results - After Review

The processing time increased a lot so it was hard to go over N bigger than 20 but this result make more sense after looking corrections and reviews.

Algorithm List				
N	greedy best first	A*	Breath First	Depth First
5	0	0	0	60
10	10	0	0	90
20	40	15	20	185

1.6 My Reviews

Here are some of my reviews for this Lab

1.6.1 Laio95

Major

- Great job! The results are good and the code well-written with comments, so it is pleasant to read. About the algorithm, I see some improvement

- You put all the initial list in the frontier, therefore you are not respecting the real definition of a frontier. It is the path from a starting node, or the different solution from each node.
- Using a true frontier, you should be able to determine a bound easier and maybe accelerating the algorithm.
- Morevover you are iterating the whole initial lists in the while loop, some possible combination are not studied. Sorting by ascending order in the initial list is not necessarly the best solution. For instance, if you have the biggest one as Lmax = [1,....,N] instead of each array in [[1],[2]....[N]] (little lists) , you would be able solve it in one iteration instead of N iteration and you will get a 0

Minor

- You could put some part of your code and more detailed results on the Readme file to read it faster
- I don't know if you tried different algorithms, but it could be interessant to have a look also on different test you did. It is always interesting to compare algorithms or differents admissible heuristic
- In my review, someone advised me to use np.array instead of lists, it improves the performance. You use 2 loops for that could be united in one.

1.6.2 SalvatoreAdalberto

Major

- Globally, The code is well-written and you respects the principal goal of this homework.
- May be you could try other algorithm and heuristics. You could find better results mainly for the lowest N.
- You found worst results than greedy algorithm made by the teacher for N = 10. The actual heuristic might be not so good as you want.
- For the possible actions you are always trying to find among the whole list data set. Maybe you should look again to find the best solution according to the solution you already found.

Minor

- The next time try to comment more your code
- In the read me we were asking to put all our results.

Chapter 2

Lab 2 - EA for Set Covering problem

Evolutionary algorithm are inspired from the biologic world. It is a genetic mimicking method. Using big population of individual that represents a solutionn we can compute a fitness, the metric that allow us to 'score' our solutions. Throughout crossover and mutations on the individuals we can evolve our initial population. Our goal is to get the best solution as we can.

- link: https://github.com/tonatiu92/Computational-Intelligence-CI-2022-01URROV-s307107/tree/main/lab2
- No group to declare
- source: Squillero github minsum

2.1 Task

Given a number N and some lists of integers $P = (L_0, L_1, L_2, ..., L_n)$, determine, if possible,$S = (L_{s_0}, L_{s_1}, L_{s_2}, ..., L_{s_n})$ such that each number between 0 and $N - 1$ appears in at least one list

$$\forall n \in [0, N-1] \exists i : n \in L_{s_i}$$

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

2.2 Solutions - before Review

2.2.1 Fitness

The fitness evaluate how fit a given solution is in solving the set covering problem. Set covering problem looks to minimize the size of the final solution to get all number in range(N).

```
def fitness(genome: list,N: int):
    """ Definition of the fitness"""
    return sum(len(_) for _ in genome)
```

2.2.2 Tournament

The set covering problem is a minimisation problem, therefore we will try to minimize it. As we are looking to the values closest to zero, we can use the absolute value of the fitness.

```
def tournament(population: list, tournament_size=2):
    """Tournament function"""
    return min(random.choices(population, k=tournament_size), key=
    lambda i: i.fitness)
```

2.2.3 Cross-Over

As seen in cass, I take a random point of my genome and then I cut at this point. The cut must be maximum the length of the smallest genome.

```
def cross_over(g1: list, g2: list):
    cut = random.randint(1, min([len(g1),len(g2)]))
    return g1[:cut] + g2[cut:]
```

2.2.4 Mutation

Three types of mutation:

- if len(genome) < Len(goal) ==> insertion of random array from the initial set
- if len(genome) > len(goal) ==> deletion of a random array

- if len(genome) == len(goal) ==> change random value with one from the initial set

```
def mutation(g: list,goal: set, initial_set:list):
    if sum(len(_) for _ in g) > len(goal):
        point = random.randint(0, len(g)-1)
        del g[point]
    elif sum(len(_) for _ in g) < len(goal):
        point = random.randint(0, len(initial_set)-1)
        g.append(initial_set[point])
    else:
        point1 = random.randint(0, len(g)-1)
        point2 = random.randint(0, len(initial_set)-1)
        del g[point1]
        g.append(initial_set[point2])
    return g.copy()
```

2.2.5 Infeasible Solutions

- Our problem provide us a lot of an infeasible solution therefore it is better to take this values.
- As the heuristic repair could be dangerous, I will not use it
- We will give a penalty to them. As set covering problem is a minmization problem, we will try to maximize this values
- According, we will provide a factor of 100 * N.

```
def feasible(solution, GOAL):
    return set([el for array in solution for el in array]) == GOAL
def penalty(value,N):
    return value*N*100 + N*100
```

2.2.6 Population initialization

```
Individual = namedtuple("Individual", ["genome", "fitness"])
def initial_population(N: int,POP_SIZE: int,seed = 42):
    """ Initialisation of the population according to feasibility
    """
    population = list()
    initial_set = problem(N,seed)
    GOAL = set(list(range(N)))
    for values in range(POP_SIZE):
       genome = random.sample(initial_set,random.randint(1,N)).copy
    ()# The ideal solution is N array of length 1 with all value
       if feasible(genome,GOAL):
           population.append(Individual(genome,fitness(genome,N)))
       else:
           population.append(Individual(genome, penalty(fitness(
    genome,N),N)))
```

```
    logging.info(f"init: N={N} pop_size={len(population)};
    min_bloat_init={min(population, key=lambda i: abs(i.fitness))
    [1]}")
    return population,initial_set
```

2.2.7 Evolution

```
fitness_log = []
def evolution(N: int,population: list,POPULATION_SIZE: int,
    NUM_GENERATIONS:int, OFFSPRING_SIZE:int,initial_set:list):
    GOAL = set(list(range(N)))
    for g in range(NUM_GENERATIONS):
        offspring = list()
        for i in range(OFFSPRING_SIZE):
            if random.random() < 0.3:
                p = tournament(population)
                o = mutation(p.genome,GOAL,initial_set)
            else:
                p1 = tournament(population)
                p2 = tournament(population)
                o = cross_over(p1.genome, p2.genome)
            if feasible(o,GOAL):
                f = fitness(o,N)
            else:
                f = penalty(fitness(o,N),N)
            offspring.append(Individual(o, f))
        population+=offspring
        population = sorted(population, key=lambda indi: abs(indi.
    fitness))[:POPULATION_SIZE]
    #plot_result(fitness_log,10*N)
    return population

def algorithm(N,pop_size,num_gen,offsize):
    POPULATION_SIZE = pop_size
    OFFSPRING_SIZE = offsize
    NUM_GENERATIONS = num_gen
    ### Initialisation of the problem
    population,initial_set = initial_population(N,POPULATION_SIZE)
    ### Evolution
    return evolution(N,population,POPULATION_SIZE,NUM_GENERATIONS,
    OFFSPRING_SIZE,initial_set)
```

2.3 Results - Before Review

To get the performance of my algorithm, I will use different parameter value according N. Now, we are not trying to find an optimal solution, but the best one using genetical algorithm. Min bloat show the best solution in the initial population. Every results shows some improvment

2.3.1 N 5 to 100

- POPULATION SIZE = N*N
- NUMBER OF GENERATION = 100*N

- NUMBER OF OFFSPRING = N*10

Results

IINFO:root:init: N=5 $\text{pop}_size = 25; min_bloat_init = 5$
INFO:root:Top 3 bests results for 5: [5, 5, 5]
INFO:root:Found in 0.1624 s

INFO:root:init: N=10 $\text{pop}_size = 100; min_bloat_init = 19$
INFO:root:Top 3 bests results for 10: [10, 10, 10]
INFO:root:Found in 0.7574 s

INFO:root:init: N=20 $\text{pop}_size = 400; min_bloat_init = 35$
INFO:root:Top 3 bests results for 20: [24, 24, 24]
INFO:root:Found in 3.2346 s

INFO:root:init: N=50 $\text{pop}_size = 2500; min_bloat_init = 121$
INFO:root:Top 3 bests results for 50: [72, 72, 72]
INFO:root:Found in 23.8752 s

INFO:root:init: N=100 $\text{pop}_size = 10000; min_bloat_init = 260$
INFO:root:Top 3 bests results for 100: [180, 188, 188]
INFO:root:Found in 130.8220 s

2.3.2 N 500 and 1000

After all test, I observe that for a initial population > 5000 there is no more improvement. And After a offsize > 1500 my performances were decreasing

- POPULATION SIZE = 5000
- NUMBER OF GENERATION = 1000
- NUMBER OF OFFSPRING = 1200

Results

INFO:root:init: N=500 $\text{pop}_size = 5000; min_bloat_init = 2094$
INFO:root:Top 3 bests results for 500: [1430, 1430, 1430]
INFO:root:Found in 49.5678 s

INFO:root:init: N=1000 $\text{pop}_size = 5000; min_bloat_init = 5307$
INFO:root:Top 3 bests results for 1000: [3399, 3399, 3399]
INFO:root:Found in 194.2127 s

2.4 Reviews

2.4.1 Review 1 by bred91

Hi! I am Raffaele Pane and this is my peer review (my choice) for lab 2.

SUMMARY

Taking a closer look at the code, I have a general good impression. The algorithm is well structured and the code is clear, organized and well commented. Suddenly, testing it, I found a major issue that leads to a wrong final result.

MAJIOR

In fact, as you can see from the image below, despite the fact that the fitness value shows optimal values for the corresponding N, looking at the best 3 genomes, one can see that there are some duplicates (and in some cases the genome length is even shorter than the expected N). The results for N= [5,10] are as follows: image

As you can see, for example in this run for N = 5, we have:

a vector of length 4 a vector of length 5 with the integers 2 and 4 duplicates a vector of length 5 with the integers 0 and 4 duplicates. The problem is probably caused by your chosen fitness function, which only takes into account the genome length and not, also, the coverage of the genome. image

My suggestion is to use a function that takes both of the above into account, and then, inside the GA, sort the population first on the coverage values and then on the length, as below: image image

SUGGESTION

One suggestion I could give you is to implement a variable mutation rate. Ideally, it would be better to do more exploration (crossover) in the early stages, and then do more exploitation (mutation) in the later ones.

2.4.2 Review 2 by Gabriele Greco s303435

Hello, I'm Gabriele and this is my peer review for your lab2

About your Solution

- mutation: the three different kind of mutations is a really good idea for length of solution
- good use of different values from N equal from 5 to 100 and from 500 to 1000. I tried dynamic values too since I notices that it's better to have different values for popsize, offpring and number of generations for small N and big N

- I didn't quite understood what the feasible function is used for. What kind of solutions are you trying to remove by assigning them a high fitness value?
- for the fitness function you can also count the distinct elements of the solution and not only the number of total elements. Like this you will reduce the number of duplicates elements
- I'm impressed by the optimal results for low value of N and even if for bigger N the solutions tend to be less optimal, the computational time is not expensive (only 222 sec for N = 1000). I obtained better results but mine algorithm runs for 20/30 minutes for these values of N ahhah. So, good job to keep the computational time low

  About code and readme very detailed readme. For the results you could add the bloat ((w-N)/ N*100), fitness calls and computational time really good use of .ipynb format with all the detail and explanation for every part of code you have the perfomance evaluation code, but no plots were show in readme or in the code.

Conclusion

Very good solution and presentation of informations, good job ;)

2.4.3 Review 3 Francesco Sattolo

Design considerations

- As a rule of thumb, genetic operators that take into consideration the individual fitness, like your mutation function does, often lead to local maxima instead of global maxima
- The crossover is inherently asymmetric since the offspring will often obtain more genetic material from the longest parent. This is not a bad thing per-se, but maybe adding another more balanced crossover function and randomly choosing which one to execute could help improving exploration.
- Good idea to change the parameters according to the problem size
- The penalty idea is interesting and could probably work in theory but I don't think in this case it is particularly effective since for example with N=10, from the fifth generation no individuals with the penalty remains. Maybe tuning the value of the penalty so that penalized individuals keep contributing to the diversity of the population for a longer time would result in an higher exploration.

Implementation considerations

- def fitness(genome: list,N: int): can be replaced by 'def fitness(genome: list):
- In the tournament function there is no reason to consider the abs(i.fitness) since the fitness is always a positive value (sum of positive lengths)
- In the mutation function you could use fitness(g) instead of repeating sum(len(el) for el in g) since you defined the function earlier
- After calling the problem() function it is necessary to reset the seed to a random value using random.seed() otherways all runs will always use 42 as seed value, so they won't be truly random

2.5 Results - After Review

Thanks to bred91 that try my algorithm in a different way, I could be able to see where my error was. My genome during mutation and crossover wasn't creating new genome but just modifying some of it so I had to copiate the entire object using genome.copy() Testing it it gave me correct results and the performance results were very similar to the previous one.

```
fitness_log = []
def evolution(N: int,population: list,POPULATION_SIZE: int,
    NUM_GENERATIONS:int, OFFSPRING_SIZE:int,initial_set:list):
    GOAL = set(list(range(N)))
    for g in range(NUM_GENERATIONS):
        offspring = list()
        for i in range(OFFSPRING_SIZE):
            if random.random() < 0.3:
                p = tournament(population)
                o = mutation(p.genome.copy(),GOAL,initial_set)
            else:
                p1 = tournament(population)
                p2 = tournament(population)
                o = cross_over(p1.genome.copy(), p2.genome.copy())
            if feasible(o,GOAL):
                f = fitness(o,N)
            else:
                f = penalty(fitness(o,N),N)
            offspring.append(Individual(o, f))
        population+=offspring
        population = sorted(population, key=lambda indi: abs(indi.
    fitness))[:POPULATION_SIZE]
    #plot_result(fitness_log,10*N)
    return population
```

2.5.1 Results

N 5 to 100

- POPULATION SIZE = N*N

- NUMBER OF GENERATION = 100*N
- NUMBER OF OFFSPRING = N*10

Results

INFO:root:init: N=5 pop$_size = 25; min_bloat_init = 5$
INFO:root:Top 3 bests results for 5: [5, 5, 5]
INFO:root:Found in 0.1746 s

INFO:root:init: N=10 pop$_size = 100; min_bloat_init = 19$
INFO:root:Top 3 bests results for 10: [10, 10, 10]
INFO:root:Found in 0.9137 s

INFO:root:init: N=20 pop$_size = 400; min_bloat_init = 35$
INFO:root:Top 3 bests results for 20: [24, 24, 24]
INFO:root:Found in 3.6648 s

INFO:root:init: N=50 pop$_size = 2500; min_bloat_init = 121$
INFO:root:Top 3 bests results for 50: [71, 71, 76]
INFO:root:Found in 25.4041 s

INFO:root:init: N=100 pop$_size = 10000; min_bloat_init = 260$
INFO:root:Top 3 bests results for 100: [162, 162, 162]
INFO:root:Found in 154.3124

N 500 and 1000

- POPULATION SIZE = 5000
- NUMBER OF GENERATION = 1000
- NUMBER OF OFFSPRING = 1200

Results

INFO:root:init: N=500 pop$_size = 5000; min_bloat_init = 2094$
INFO:root:Top 3 bests results for 500: [1426, 1426, 1426]
INFO:root:Found in 76.3377 s

INFO:root:init: N=1000 pop$_size = 5000; min_bloat_init = 5307$
INFO:root:Top 3 bests results for 1000: [3429, 3431, 3431]
INFO:root:Found in 288.7160 s

2.6 My Reviews

Here are some of my reviews for this Lab

2.6.1 mistru97

Major

Initialisation of the population

- I think it is not the most efficient way to represent the solution as you pick only one array inside the alllists solutions at the initialisation, may be you could pick a sample. It will improve the velocity but also the randomness and the diversification of your initial solution.
- I like the idea of giving penalty according to the repetition inside thefinal solution.

Evolution

- I think probability of 0.5 for mutation can be dangerous, in general we use value between 0.1 0.3 (What I read in some research).
- The cross over is good
- For the mutation you can think about you could orientate the randomness to the final goal. May be taking in account the len of the genome.

Results

- They are good but I think that working again on the initialisation part to get better results

Minor

- Good use of global values
- You could comment more your code to explain how you structure the problem it will be easier to understand

Chapter 3

Lab 3 - Playing Nim

Play a game involves rules and strategy. Most of the time, we can adopt some specific algorithm to try to win a game. In this lab we tried to understand different possible techniques that are rule-based system, evolved rules, Minimax and Reinforcment Learning.

(Observation: As I was hill the weeks of this lab I only manage to solve until the Minimax

- link: https://github.com/tonatiu92/Computational-Intelligence-CI-2022-01URROV-s307107/tree/main/lab3
- No group to declare
- source: Squillero github and https://realpython.com/python-minimax-nim/

3.1 Task

3.2 Solution - Before Review

3.2.1 The game

- Deterministic
- Perfect Information
- Turn-based
- Zero Sum

3.2.2 Rule Based Agent

```
    def __init__(self,id, params = [0.2,0.16,0.16,0.16,0.16,0.16]):
        super().__init__(id)
        self._params = params

    @property
    def params(self):
        return self._params
    @params.setter
    def name(self, value):
        self._params = value

    def cooked_rules(self, state: Nim, moves: list):
        cooked = dict()
        moves_row = [el for el in state.rows if el != 0]

        # SIMPLE RULES
        simple = dict()

        shortest_row = min((x for x in enumerate(state.rows) if x
    [1] > 0), key=lambda y: y[1])[0] #index shortest row (In class)
        longest_row =  max((x for x in enumerate(state.rows)), key=
    lambda y: y[1])[0]
        if state.k:
            simple["shortest_row"] = Nimply(shortest_row,self.
    choose_nim_quantity(state,min(state.rows[shortest_row],state.k)
    ))
            simple["longest_row"] = Nimply(longest_row,self.
    choose_nim_quantity(state,min(state.rows[shortest_row],state.k)
    )) #index longest row (In class)
        else:
            simple["shortest_row"] = Nimply(shortest_row,self.
    choose_nim_quantity(state,state.rows[shortest_row]))
            simple["longest_row"] = Nimply(longest_row,self.
    choose_nim_quantity(state,state.rows[shortest_row])) #index
    longest row (In class)

        simple["min_k"] = Nimply(state.rows.index(moves_row[random.
    randint(0,len(moves_row)-1)]),1) #nimming always 1
```

```
29          #simple["same_action"] = state.last_action #try to play the
     same action as the opponent
30          cooked["simple"] = simple
31
32          #GENERALIZED RULES
33          cooked["remaining_parity_row"] = self.action_strategy_row(
    state,cooked,moves_row)
34          cooked["remaining_quantity_row"] = self.
    action_strategy_quantity(state,moves_row)
35          cooked["reaction_longest_shortest"] = self.
    reaction_strategy_longest_vs_shortest(state,cooked)
36          cooked["reaction_quantity"] = self.
    reaction_strategy_quantity(state,moves_row)
37          cooked["random"] = self.pure_random(state)
38          return cooked
39
40      def choose_nim_quantity(self,state: Nim,row: int):
41          if (state.k):
42              if (row > state.k):
43                  return random.randint(1,state.k)
44          return random.randint(1,row)
45
46      def clear_row(self,state: Nim, row_value: int):
47          """action of clear a row"""
48          return Nimply(state.rows.index(row_value),row_value)
49
50      def  keep_one(self,state: Nim, row_value: int):
51          """Try to keep the last one for the player"""
52          return Nimply(state.rows.index(row_value),min(int(row_value
    - (state.k + 1)),state.k))
53
54      def action_strategy_row(self, state: Nim, rules: dict,
    possible_rows: list):
55          """
56          According to the parity of the remaning rows
57          """
58          ply = None
59          non_empty_row = possible_rows.copy()
60
61          if (len(non_empty_row)%2 == 0):
62              """If the number of rows remaining is pair we don't
    want to empty a row"""
63              non_empty_row = sorted(non_empty_row, reverse = True) #
     It is harder to empty the longest one.
64              if state.k:
65                  if non_empty_row[0] <= state.k:
66                      """If the row size is less than the state.k, we
     only remove 1"""
67                      ply = Nimply(state.rows.index(non_empty_row[0])
    ,1)
68                  else:
69                      ply = self.keep_one(state, non_empty_row[0])
70              else:
71                  """If k is not defined we will try to force the
    opponent to take the last one"""
72                  longest = rules["simple"]["longest_row"][0]
73                  ply = Nimply(longest,state.rows[longest]-1)
```

```
74
75          elif (len(non_empty_row)%2 != 0):
76              """If the number of rows remaining is unpair we want to
     empty a row"""
77              non_empty_row = sorted(non_empty_row) # It is easier to
     remove the shortest rows
78              if state.k:
79                  if non_empty_row[0] <= state._k :
80                      """The player can nim all pieces"""
81                      ply = self.clear_row(state,non_empty_row[0])
82                  else:
83                      """Try to keep the last one for the player and
    not the opponent"""
84                      ply = self.keep_one(state,non_empty_row[0]) #
    Try to go to a quantity of state.k + 1 in the last row if not
    remove state.k
85              else:
86                  ply = Nimply(rules["simple"]["shortest_row"][0],
    non_empty_row[0])
87
88          return ply
89
90      def action_strategy_quantity(self, state: Nim,possible_rows:
    list):
91          """
92          Strategy according to the reamining quantity of nim
93          """
94          ply = None
95          non_empty_row = possible_rows.copy()
96          if(sum(state.rows)%2 ==0):
97              """If the quantity remaining is paired we want to keep
    a paired quantity"""
98              valid_rows  = [el for el in non_empty_row if el > 1] #>
     1 because we want to nim a paired quantity
99              if valid_rows:
100                 row = valid_rows[random.randint(0,len(valid_rows)
    -1)] # Select a random row where we can remove a pair number
101                 if state.k:
102                     if state.k == 1:
103                         """This strategy focus on the quantity to
    nim so if state.k == 1 is the only case were we have only
    unpaired quantity to remove"""
104                         nimming =[1]
105                     else:
106                         nimming = [el for el in range(1,state.k+1)
    if (el%2==0) & (el <= row)] #Nim only unpaired quantity
    possible in the selected row
107                 else:
108                     if row == 1:
109                         nimming = [1]
110                     else:
111                         nimming = [el for el in range(1,row+1) if (
    el%2==0) & (el <= row)] #Nim only unpaired quantity possible in
     the selected row
112                 ply = Nimply(state.rows.index(row), sorted(nimming,
    reverse=True)[0])
113             else:
```

```
                row  = non_empty_row[random.randint(0,len(non_empty_row)-1)]#Select a random non empty row
                if state.k:
                    if state.k == 1:
                        """This strategy focus on the quantity to nim so if state.k == 1 is the only case were we have only unpaired quantity to remove"""
                        nimming =[1]
                    else:
                        nimming = [el for el in range(1,state.k+1) if (el%2!=0) & (el <= row)] #Nim only paired quantity possible in the selected row
                else:
                    if row == 1:
                        nimming = [1]
                    else:
                        nimming = [el for el in range(1,row+1) if (el%2!=0) & (el <= row)] #Nim only paired quantity possible in the selected row
                ply = Nimply(state.rows.index(row), sorted(nimming)[0])
            else:
                """If the quantity remainig is unpaired we want to get a paired quantity"""
                row  = non_empty_row[random.randint(0,len(non_empty_row)-1)]#Select a random non empty row
                if state.k:
                    if state.k == 1:
                        """This strategy focus on the quantity to nim so if state.k == 1 is the only case were we have only unpaired quantity to remove"""
                        nimming =[1]
                    else:
                        nimming = [el for el in range(1,state.k+1) if (el%2!=0) & (el <= row)] #Nim only paired quantity possible in the selected row
                else:
                    if row == 1:
                        nimming = [1]
                    else:
                        nimming = [el for el in range(1,row+1) if (el%2!=0) & (el <= row)] #Nim only paired quantity possible in the selected row
                ply = Nimply(state.rows.index(row), sorted(nimming)[0])
            return ply

        def reaction_strategy_longest_vs_shortest(self, state: Nim, rules:dict):
            """
            The player will play in a different domain than the opponent
            """
            if state.last_action:
                if state.row[state.last_action[0]] <= sum(state.rows)//2:
                    """The oppponent nimmed shortest rows so I want to play on the longest"""
```

```
                row  = state.rows[rules["longest_row"]]#Select the longest row
                if state.k:
                    if state.k == 1:
                         nimming =[1]
                    else:
                        nimming = [el for el in range(1,state.k+1) if el <= row] #Nim  quantity possible in the selected row
                else:
                    nimming = [el for el in range(1,row+1) if el <= row] #Nim quantity possible in the selected row
                ply = Nimply(state.rows.index(row), nimming[0])
            else:
                """The oppponent nimmed longest row so I want to play on the shortest"""
                row  = state.rows[rules["shortest_row"]]#Select the shortest row
                if state.k:
                    if state.k == 1:
                         nimming =[1]
                    else:
                        nimming = [el for el in range(1,state.k+1) if el <= row] #Nim  quantity possible in the selected row
                else:
                    if row == 1:
                        nimming = [1]
                    else:
                        nimming = [el for el in range(1,row+1) if el <= row] #No  quantity possible in the selected row
                ply = Nimply(state.rows.index(row), nimming[0])
            return ply
        else:
            return self.pure_random(state)

    def reaction_strategy_quantity(self, state: Nim, possible_rows):
        """
        The player will play different quantity than the oppponent
        """
        non_empty_row = possible_rows.copy()
        if state.last_action:
            row  = non_empty_row[random.randint(0,len(non_empty_row)-1)]#Select a random row
            if state.k == 1:
                """This strategy focus on the quantity to nim so if state.k == 1 is the only case were we have only unpaired quantity to remove"""
                ply = Nimply(state.rows.index(row), 1)
            elif state.last_action[1]%2==0:
                """The oppponent nimmed shortest a pair quantity of nim"""
                if state.k:
                    nimming = [el for el in range(1,state.k+1) if (el%2!=0) & (el <= row)] #Nim quantity possible in the selected row
                else:
                    if row == 1:
```

```
                        nimming = [1]
                    else:
                        nimming = [el for el in range(1,row+1) if (
    el%2!=0) & (el <= row)] #Nim quantity possible in the selected
    row
                ply = Nimply(state.rows.index(row), nimming[0])
            else:
                """The oppponent nimmed shortest an unpair quantity
     of nim"""""
                if state.k:
                    nimming = [el for el in range(1,state.k+1) if (
    el%2==0) & (el <= row)] #Nim  quantity possible in the selected
     row
                else:
                    if row == 1:
                        nimming = [1]
                    else:
                        nimming = [el for el in range(1,row+1) if (
    el%2==0) & (el <= row)] #No  quantity possible in the selected
     row
                ply = Nimply(state.rows.index(row), nimming[0])
            return ply

        else:
            return self.pure_random(state)

    def pure_random(self,state: Nim):
        if not state.k:
            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)
        else:
            if self.possible_moves(state):
                row = random.choice([r for r, c in self.
    possible_moves(state) if (c > 0) & (c <=state.k)])
                num_objects = random.randint(1, min(random.randint
    (1,state.k),state.rows[row]))
                return Nimply(row, num_objects)
            else:
                None

    def strategy(self, state:Nim): #DEFINE PARAMS AS ATTRIBUTE
        """
        strategies used to react to opponent actions, I xhoose to
    wait different kind of rules
        @state: the state of the nim game
        @params: probability of each strategy the simple strategy
    will be always the same. sum of probability all simple = 0.2
        """
        moves = self.possible_moves(state)
        rules = self.cooked_rules(state,moves)

        strategies_name = list(rules.keys())
        strategy_choosed = np.random.choice(strategies_name, p=self
    .params)
        if strategy_choosed == "simple":
```

```
238             strategy_choosed = np.random.choice(list(rules[
    strategy_choosed].keys()))
239             return rules["simple"][strategy_choosed]
240         return rules[strategy_choosed]
```

Simple Rules

First of all, I tried to understand how to play nim. I used some simple rules implemented in class as always take a nim in the shortest row or the longest rows, always take only 1 nim. Line 17 to 30

Generalized Rules

I implemented 4 rules to play nim.

According parity of the remaining rows L54-88 My first observation was that when it remains an unpaired number of rows in the game, we should try to remove this row. Otherwise, if it remains a paired quantity of rows, we should force the opponent to remove one row. If the quantity of row remaining is paired, we will nim in order to get only one nim in the row. Therefore if k is None, we remained the quantity - 1, or if k is not None we will nim only one nim. Moreover we will play on the longest row because it is the less likeky to be empty. If the remained quantity is unpaired, we will try to nim everything in the row if is None, or will try to get K+1 nim in the row because we don't want the opponent to empty the row. As we want to remove a row, we will work on the shortest one that are easier to remove.

According the quantity of row remaining L92-142 The second rule is smilar to the first one. Instead of looking at the parity of the row quantity, we will look after the nim quantity. We will do the same acions

According the region opponent action (longest or shortest rows) L144-176 Here, I want to play in a different region that the opponent. The goal is to play in different rows to split the game in "two sides"

According the quantity of nim removed by the opponent L178-211 If the opponent played paired quantity, the agent will play unpaired quantity and vice-versa

Results

3.2.3 Evolved Agent

The first agent was created to define a set of rules. Now we need to parameterize this rules to evolve them. That is why we give a weight for our different strategies. Choosing a specific strategy at each turn

```
1 def strategy(self, state:Nim): #DEFINE PARAMS AS ATTRIBUTE
2       """
```

```
      strategies used to react to opponent actions, I xhoose to wait different kind of rules
      @state: the state of the nim game
      @params: probability of each strategy the simple strategy will be always the same. sum of probability all simple = 0.2
      """
      moves = self.possible_moves(state)
      rules = self.cooked_rules(state,moves)

      strategies_name = list(rules.keys())
      strategy_choosed = np.random.choice(strategies_name, p=self.params)
      if strategy_choosed == "simple":
          strategy_choosed = np.random.choice(list(rules[strategy_choosed].keys()))
          return rules["simple"][strategy_choosed]
      return rules[strategy_choosed]
```

Individuals

In this case an individual or genome is a list of all parameters/ all the wheights for each rules. Example of parameters: **[0.2,0.16,0.16,0.16,0.16,0.16]**

```
Individual = namedtuple("Individual", ["genome", "fitness"])
def generate_weights(weights: list):
    """Function that make the sum of weight == 1"""
    p = weights
    p   = np.array(p)
    p = p/p.sum()  # normalize
    return p
```

Fitness

The fitness is the mean of victory for the player with the genome. We also play with a different nim size to get the most accurate value.

```
NUM_MATCHES = 50
NIM_SIZE = 20
K = 5

def fitness(genome: list):
    evolution = []
    for i in range(NIM_SIZE):
        winner = []
        for m in range(NUM_MATCHES):
            nim = Nim(i)
            p1 = RuleBased("classic_rules",genome)
            p2 = RuleBased("pure_random",[0,0,0,0,0,1])
            game = Game(p1,p2, i)
            winner.append(game.run())
        evolution.append(op.countOf(winner,"classic_rules")/len(winner))
        #print(f'Percentage of victory against pure random:{op.countOf(winner,"classic_rules")/len(winner)} with nim size {i}')
    return np.mean(evolution)
```

Cross-Over

We will perform a cross-over between the parameters of three parents because the len of genome is quite big and it will offer better diversity.

```
def cross_over(g1, g2, g3):
    cut = random.randint(0, 3)
    cut2 = random.randint(cut,5)
    return generate_weights(g1.tolist()[:cut] + g2.tolist()[cut:
    cut2] + g3.tolist()[cut2:])
```

Mutation

We will perform a mutation on one of the two parameter, applying a random mutation on it. It will be the permutation of two parameters inside the genome

```
def mutation(g:list):
    point1 = random.randint(0, len(g) - 1)
    point2 = None
    while not point2:
        test = random.randint(0, len(g) - 1)
        if test != point1:
            point2 = test
    tmp = g[point1]
    g[point1] = g[point2]
    g[point2] = tmp
    return g
```

Tournament

We will get the maximum fitness between two Individuals

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

Evolution

Now, we can evolve our rules to get the best value.

```
def create_rand_pop(count):
    pop = []
    for i in range(count):
        genome = generate_weights([random.randint(1,100) for i in
    range(6)])
        logging.debug(f"status: Created {i} genome")
        fitness_value = fitness(genome)
        pop.append(Individual(genome,fitness_value))
    return pop
def evolution(population: list,POPULATION_SIZE: int,
    NUM_GENERATIONS:int, OFFSPRING_SIZE:int):
    for g in range(NUM_GENERATIONS):
        logging.debug(f"generation_number {g}")
        offspring = list()
```

```
        for i in range(OFFSPRING_SIZE):
            if random.random() < 0.3:
                p = tournament(population)
                o = mutation(p.genome.copy())
            else:
                p1 = tournament(population)
                p2 = tournament(population)
                p3 = tournament(population)
                o = cross_over(p1.genome.copy(), p2.genome.copy(),
    p3.genome.copy())
            f = fitness(o)
            offspring.append(Individual(o, f))
        population+=offspring
        population = sorted(population, key=lambda indi: indi.
    fitness,reverse=True)[:POPULATION_SIZE]
        logging.debug(f"top fitness {population[0].fitness}")
    #plot_result(fitness_log,10*N)
    return population[0]
```

Results

The evolution show a significant improvement when I play without k constraints I am winning more than 90 percent of the games against the random player, It shows a little improvement when we have a variation of k. But the results are still not good, 70 percents of winning more or less I assume it must be because my rules are not focusing on the k value. I didn't find any interesting idea to implement.

3.2.4 Minimax

```
class MinMax(Player):
    def __init__(self,id):
        super().__init__(id)


    def evaluate(self,  state, is_maximizing):
        if all(el == 0 for el in state.rows):
            return -1 if is_maximizing else 1

    def possible_states(self,state:Nim):
        states = []
        for el in self.possible_moves(state):
            cop = copy.deepcopy(state)
            cop.nimming(Nimply(el[0],el[1]))
            states.append(cop)
        return  states

    def minimax(self, state: Nim, is_maximizing, alpha=-1, beta=1):
        if (score := self.evaluate(state, is_maximizing)) is not
    None:
            """if we are in final leaf, return the score"""
            return score

        scores = []
```

```
        for moves in self.possible_moves(state):
            cop = copy.deepcopy(state)
            cop.nimming(Nimply(moves[0],moves[1]))
            scores.append(
                score := self.minimax(cop, not is_maximizing, alpha
    , beta)
            )
            if is_maximizing:
                alpha = max(alpha, score)
            else:
                beta = min(beta, score)
            if beta <= alpha:
                break
        return (max if is_maximizing else min)(scores)

    def best_move(self, state:Nim):
        scores = {}
        for s in self.possible_states(state):
            scores[s] = self.minimax(s,False)
        return sorted(scores.items(),key=lambda x: x[1], reverse=
    True)[0][0]

    def diff(self,stateA: Nim,stateB: Nim):
        for i in range(len(stateA.rows)):
            if stateA.rows[i] != stateB.rows[i]:
                return (i,stateA.rows[i]-stateB.rows[i])

    def strategy(self,state:Nim):
        # Minimax move
        cop = copy.deepcopy(state)
        move = self.best_move(state)
        return self.diff(state,move)
```

Results

source:https://realpython.com/python-minimax-nim/ The basic minmax works well until 4 rows. It always win as expected but the processing time is very big.

3.2.5 Reinforcement Learning

I didn't have time to code it for the final assignement. But we can setup the environment as the 'nim board' and the agent will be my previous Rule Based agent.

Results

No results

3.3 Reviews

3.3.1 Review 1 by Francesco Sattolo

Design considerations

- Good amount of variety in the fixed rules of part 1.
- Maybe more detailed statistics can provide better insights, like distinguishing winnable and unwinnable games to see how many of each type were won.
- In part 2 (evolved strategies) you used a similar strategy to mine to find the best combination of strategies. In the fitness function maybe you could also make it compete with different strategies and not only with purerandom, so that it can improve more. The next step would be to evolve which rule is better to use depending on the current game situation but I couldn't do it either.
- Minmax algorithm can be improved by 1. storing previously computed states and 2. avoiding deepcopies as much as possible. For example, since we use recursion and backtracking, I just try a nim, and then after the recursion I do the "opposite" nimming to reset the state to how it was before the recursion.

Implementation considerations

- Trying to execute the task 3.1 results system resources exhaust quickly and I can't complete the execution. There is a way to compute the best move (the one that brings the nim sum=0) without bruteforcing it, which will improve performance. You can find it in my repository.
- *, result = accumulate(state.rows, xor) can be replaced by result = reduce(state.rows, xor)
- I like the idea of using classes for the Game, Player, ExpertPlayer
- I appreciate the comments explaining the code's working

3.4 Results - After Review

3.4.1 MinMax Improved

Now the MinMax got good results until 8 nim.

```python
class MinMaxImproved(Player):
    def __init__(self,id):
        super().__init__(id)
        self._possible_new_states = None

    def diff(self,stateA: Nim,stateB: Nim):
        for i in range(len(stateA)):
            if stateA[i] != stateB[i]:
                return (i,stateA[i]-stateB[i])

    def possible_new_states(self,state):
        for pile, counters in enumerate(state):
            for remain in range(counters):
                yield state[:pile] + (remain,) + state[pile + 1 :]

    def evaluate(self,state, is_maximizing):
        if all(counters == 0 for counters in state):
            return -1 if is_maximizing else 1

    @functools.cache
    def minimax(self,state, is_maximizing, alpha=-1, beta=1):
        """Evaluate a game state using the minimax algorithm"""
        if (score := self.evaluate(state, is_maximizing)) is not None:
            return score

        scores = []
        for new_state in self.possible_new_states(state):
            scores.append(
                score := self.minimax(new_state, not is_maximizing, alpha, beta)
            )
            if is_maximizing:
                alpha = max(alpha, score)
            else:
                beta = min(beta, score)
            if beta <= alpha:
                break
        return (max if is_maximizing else min)(scores)

    def best_move(self, state):
        """Use minimax() to find the best move"""
        evaluate = functools.partial(self.minimax, is_maximizing=False)
        return max(self.possible_new_states(state), key=evaluate, default=None)

    def strategy(self,state:Nim):
        cop = state.rows
       # self._possible_new_states = self.possible_new_states(state)
        move = self.best_move(tuple(cop))
        return self.diff(cop,move)
```

3.5 My Reviews

Here are some of my reviews for this Lab

3.5.1 bred91

Major

- The rule based AI uses a great strategy, but I think you could deeper in some other strategy. There is many possible states of the game that can be hardcoded according the number of row missing (as you do), the last action of the player, the number of nim missing and also the value of k.
- The Second task, you use the same methodology as I do. In my reviews they said that we could also train the GA with another algorithm than pure random to be more efficient
- Your Minmax seems to work well

Minor

- May be using class, you should make a more generic code
- You should try to show the results against a serie of matches (100) and do some average to have a quantitative results vision. The code is well presented and commented

3.5.2 antoniodecinque99

Major

- Your rules are interesting, you could also train your rules changing the k value, you are taking in account only the case when k is None
- My rules are similar to yours, maybe you could also compare it with other strategies like gabriele or some other basic rules
- I think you could improve task 2 weighting different rules, It will evolve to get the best probabilities of each rules. For me I see both picj function as the same IF ELSE rules, maybe adding some randomness rules, some Other Simple rules and weighting them you will get better results for your fitness. Your evolved rules are not evolving very well compared to your results in the task 1

Minor

- Well coded and well commented code

- Sorry for the 2 other tasks but I didn't had great results, your minmax is quite efficient and the Q-learning is very impressive, I would have a deeper look at your code to understand what I was not able to do.

Chapter 4

Final Project - The Quarto Game

This part aims to explain the work did on the final project.

- link: https://github.com/tonatiu92/FinalExam_CI2223_S307107
- No group to declare
- source: Squillero github

4.1 The game

The game is played on a 4x4 grid, with 16 pieces that each have four attributes: shape (square or circle), height (tall or short), color (red or blue), an solidity (solid or hollow).

At the start of the game, all 16 pieces are placed in a bag or box. Players take turns drawing a piece from the bag or box and placing it on the board. On their turn, player 1 choose the piece that their opponent will place on their next turn. Player 2 places the chosen piece on the board The game continues until one player creates a row of four pieces that share a common attributes (such as all being the same color or shape), at which point that player wins the game.

4.2 Agent - Rule Based System

I choose to create a rule based System to play QUarto. First, we have two main actions: choose a piece for the next player and place a piece on the board.

4.2.1 Placing Action

As many strategy game, we got two type of actions for placing pieces on the board. It is possible to make an attack move or a defensive one.

Analysis

As a human, before any move, the agent need to analyze the board. That is why I implemented some functions to score the free positions on the board.

To do it, I created two functions that score the alignment present in the board. Therefore, instead of scoring a specific case, I will score each row, column and diagonal where it is possible to make a quarto. In other word, we have four possible alignment horizontally and vertically and 2 possible diagonals. I implemented the function **boardanalysis**. Furthermore, I decided to score the game by common characteristics shared between the aligned pieces. The **scorealignement** calculate for each characteristics the number of piece of the alignment which share the same. If all pieces aligned share the same characteristics with the piece chosen by the opponent, we can give a score to the piece. Otherwise, the characteristic is discarded.

```
def score_alignement(self, alignement: list, selected_piece_char:
    quarto.Piece) -> list:
    """
    Description: calculate the score for aligned pieces

    @args:
    alignement [list]: the list of pieces aligned
    selected_piece_char [quarto.Piece]: the piece to compare in
    order to get the score

    @returns:
    the list of score for each characteristics
    """
    score_char = [0,0,0,0] #HIGH, COLOURED, SOLID, SQUARE
    features = [[piece.HIGH, piece.COLOURED, piece.SOLID, piece.
    SQUARE] for piece in alignement] #Get the features of each
    piece

    selected_value  = [selected_piece_char.HIGH,
    selected_piece_char.COLOURED, selected_piece_char.SOLID,
    selected_piece_char.SQUARE]
    for i in range(4):
        count_feature = set([piece[i] for piece in features]) #get
    the feature n   i
        if ((len(count_feature) == 1) and (list(count_feature)[0]
    == selected_value[i])) :
            score_char[i] = len(alignement)
    return score_char
```

```
21
22 def board_analysis(self,piece_selected: int) -> dict:
23     """
24     Description: Calculate the score for each possible alignment
25
26     @args
27     piece_selected [int]: index of the piece selected to be insert
       on the board
28
29     @returns
30     The score for each possible alignement
31     """
32     board_copy = self.get_game().get_board_status()
33     piece_selected = self.get_game().get_piece_charachteristics(
       piece_selected)
34     horizontal = {"0":[],"1":[],"2":[],"3":[]}
35     vertical = {"0":[],"1":[],"2":[],"3":[]}
36     diag = {"0":[],"1":[]}
37     for i in range(self.get_game().BOARD_SIDE):
38         for j in range(self.get_game().BOARD_SIDE):
39             if board_copy[i,j] != -1:
40                 piece = self.get_game().get_piece_charachteristics(
       board_copy[i,j])
41                 horizontal[str(i)].append(piece)
42                 vertical[str(j)].append(piece)
43                 if i == j:
44                     diag["0"].append(piece)
45                 elif i + j == 3:
46                     diag["1"].append(piece)
47     scores = {"scoreH": {"0":[],"1":[],"2":[],"3":[]},"scoreV":{"0"
       :[],"1":[],"2":[],"3":[]},"scoreD":{"0":[],"1":[]}}
48     for k in range(self.get_game().BOARD_SIDE):
49         scores["scoreH"][str(k)] = self.score_alignement(horizontal
       [str(k)],piece_selected)
50         scores["scoreV"][str(k)] = self.score_alignement(vertical[
       str(k)],piece_selected)
51         if k <= 1:
52             scores["scoreD"][str(k)] = self.score_alignement(diag[
       str(k)],piece_selected)
53     return scores
```

Beginning of the game

At the beginning of the game, we don't know where to play, so we can force the player to play in the diagonal. Therefore, I created a function that play randomly on diagonal a the beginning of the game.

```
1 def oriented_random(self) -> tuple[int, int]:
2       """
3       TYPE OF STRATEGY: ATTACK
4       DESCRIPTION: At the beginning of the game, it could be
      interesting to play on the diagonal where the possibility of
      quarto is 3 lines
5
6       @returns: tuple[int, int]
7       """
```

```
        l=[]
        length = self.get_game().BOARD_SIDE
        for i in range(length):
            if self.get_game().get_board_status()[i,i] == -1:
                l.append((i,i))
            if self.get_game().get_board_status()[length-1-i,i] ==
    -1:
                l.append((i,length-1-i))
        if len(l) > 0:
            choose = random.randint(0,len(l)-1)
            return l[choose]
        else:
            return random.randint(0,3),random.randint(0,3)
```

Attack

Attack consist of making a move in order to win. It involves to take a risk. Using an offensive strategy, the player will try to make a quarto finding the best alignment.

Critical Attack:

First of all, I take in account a critical case that will be the priority number one for the agent. If there is any possibility to make a quarto while he is playing, he must place the piece on the board. That is why I created a method **completingraw** that will make a quarto according the result of the method method **possiblequarto** that look at the score results seen previously. If there is a score of 3 in any alignement, the agent can place the piece on the free position.

```
def completing_raw(self, board_copy:np.ndarray) -> tuple[int, int]:
        """
        TYPE OF STRATEGY:  ATTACK (critical action => Priority 1)
        DESCRIPTION: If there is an open space on the board where
    placing a piece would create a row of four pieces that all have
     a similarity,
        then place a piece of that similarity in that space.
        @returns: tuple[int, int]
        """
       # print("COMP")
        scores = self.board_analysis(self.get_game().
    get_selected_piece())
        return self.possible_quarto(board_copy,scores)

def possible_quarto(self, board_copy: np.ndarray, scores: dict) ->
    tuple[int,int]:
    for i in range(self.get_game().BOARD_SIDE):
        for j in range(self.get_game().BOARD_SIDE):
            if board_copy[i,j] == -1:
                scoreH = scores["scoreH"][str(i)]
                scoreV = scores["scoreV"][str(j)]
                if i==j:
                    scoreD = scores["scoreD"]["0"]
                elif i +j == 3:
                    scoreD = scores["scoreD"]["1"]
                else:
```

```
                    scoreD = [0]
                if 3 in scoreH or 3 in scoreV or 3 in scoreD:
                    return [j,i]
    return [-1,-1]
```

Attack Scoring

The attack strategy in a general case, is to place the piece in the best position to win the game. To begin with, we sum all the score for each possible alignement in the board (4 horizontal, 4 vertical, 2 diagonal) as seen before. Knowing all those alignement, we can know score the free positions.

For each aligned piece, the agent sum previous calculated score. Moreover, some pieces are present in many alignements. In order, to know the region/alignement where it will be more probable to make a quarto, we can upgrade the score according the number of occurences of a piece in an alignement. Finally, we will extract the best horizontal, diagonal and vertical alignement. (cf **alignement** function)

As the two diagonals are the positions where the probability of making a quarto are greater, we will try to prioritize the placement on diagonals. Then using the maxScore we return the best position if some alignement exist.

```
def alignement(self, board_copy:np.ndarray, selected_piece: int) -> list:
    """
    returns: the possible horizontal, vertical, diagonal alignements
    """
    board_score = sorted(self.scoring_positions(board_copy, selected_piece), key=lambda x: x[1], reverse = True)
    horizontal = []
    vertical = []
    diagonal = []
    if len(board_score)<2:
        return horizontal, vertical, diagonal
    points_occurences = {} #dict to identify the points with many alignements type (max 3).

    #Looking for possible alignement
    for p1 in board_score:
        for p2 in board_score:
            #if the piece are not antagonist there are not a possible alignement
            if (p1!=p2) & (board_copy[p1[0][0],p1[0][1]] + board_copy[p2[0][0],p2[0][1]] != 15):

                #HorizontalCheck
                if (p1[0][0] == p2[0][0]) & (((p2[0],p1[0]),p1[1] + p2[1]) not in horizontal):

                    #adding alignement and penalizing some
                    horizontal.append(((p1[0],p2[0]),p1[1] + p2[1]))

                    #identifying points & penalizing some
                    if (str(p1[0]) not in points_occurences):
                        points_occurences[str(p1[0])] = 1
```

```
                    else:
                        points_occurences[str(p1[0])] += 1

                #VerticalCheck
                if (p1[0][1] == p2[0][1]) & (((p2[0],p1[0]),p1[1] +
    p2[1]) not in vertical):
                    vertical.append(((p1[0],p2[0]),p1[1] + p2[1]))

                    #identifying points
                    if str(p1[0]) not in points_occurences:
                        points_occurences[str(p1[0])] = 1
                    else:
                        points_occurences[str(p1[0])] += 1

                #DiagonalCheck
                if (p1[0][0] == p1[0][1]) & (p2[0][0] == p2[0][1])
    & (((p2[0],p1[0]),p1[1] + p2[1]) not in diagonal):
                    diagonal.append(((p1[0],p2[0]),p1[1] + p2[1]))
                    #identifying points
                    if str(p1[0]) not in points_occurences:
                        points_occurences[str(p1[0])] = 1
                    else:
                        points_occurences[str(p1[0])] += 1

                elif (p1[0][0] + p1[0][1] == 3) & (p2[0][0] + p2
    [0][1]== 3) & (((p2[0],p1[0]),p1[1] + p2[1]) not in diagonal):
                    diagonal.append(((p1[0],p2[0]),p1[1] + p2[1]))

                    #identifying points
                    if str(p1[0]) not in points_occurences:
                        points_occurences[str(p1[0])] = 1
                    else:
                        points_occurences[str(p1[0])] += 1

    #rewards to points with more than 1 alignement
    for points in points_occurences.items():
        if points[1] >= 3:
            for el in horizontal:
                if el[0] == ast.literal_eval(points[0]):
                    el[1] *= 10000
            for el in vertical:
                if el[0] == ast.literal_eval(points[0]):
                    el[1] *= 10000
            for el in diagonal:
                if el[0] == ast.literal_eval(points[0]):
                    el[1] *= 10000

        elif points[1] >= 2:
            for el in horizontal:
                if el[0] == ast.literal_eval(points[0]):
                    el[1] *= 1000
            for el in vertical:
                if el[0] == ast.literal_eval(points[0]):
                    el[1] *= 1000
            for el in diagonal:
                if el[0] == ast.literal_eval(points[0]):
                    el[1] *= 1000
```

```
82
83      #Sorting according the sum of scores
84      horizontal = sorted(horizontal, key=lambda x: x[1], reverse =
        True)
85      vertical = sorted(vertical, key=lambda x: x[1], reverse = True)
86      diagonal = sorted(diagonal, key=lambda x: x[1], reverse = True)
87
88      return horizontal, vertical, diagonal
89
90  def best_alignement(self, horizontal: list, vertical: list,
        diagonal: list) -> list:
91      """
92      Here we are trying to find the best alignement, when there is
        negative value of an alignement, we want to check if it is not
        the only one
93      because we don't to consider as good alignement negative one
94      """
95
96      best_horizontal = ((-1,-1),-1)
97      best_vertical = ((-1,-1),-1)
98      best_diagonal = ((-1,-1),-1)
99      if horizontal:
100         if horizontal[-1][1] < 0:
101             for row in horizontal[:-1]:
102                 if (row[0][0] != horizontal[-1][0][0]):
103                     best_horizontal = row
104         else:
105             best_horizontal = horizontal[0]
106     if vertical:
107         if vertical[-1][1] < 0:
108             for col in vertical[:-1]:
109                 if (col[0][0] != vertical[-1][0][0]):
110                     best_vertical = col
111         else:
112             best_vertical = vertical[0]
113
114     if diagonal:
115         if diagonal[-1][1] < 0:
116             for diag in diagonal[:-1]:
117                 if (diag[0][0] != diagonal[-1][0][0]):
118                     best_diagonal = diag
119         else:
120             best_diagonal = diagonal[0]
121     return best_horizontal, best_vertical, best_diagonal
122
123  def scoring_strategy(self, board_copy:np.ndarray, defense = False)
        ->tuple[int, int]:
124         """
125         TYPE OF STRATEGY: ATTACK
126         DESCRIPTION: In this function we are trying to calculate
        the square where we are more likely to do a quarto win
127         We already think the case of three pieces sharing a common
        attribute with the completing method
128         Here we will try to find when 2 pieces are alignated
129
130         @returns: tuple [int, int]
131         """
```

```
132         selected_piece = self.get_game().get_selected_piece()
133         horizontal, vertical, diagonal = self.alignement(board_copy
    , selected_piece, defense)
134         best_horizontal, best_vertical, best_diagonal = self.
    best_alignement(horizontal, vertical, diagonal)
135         return self.choose_position(board_copy,best_horizontal,
    best_vertical, best_diagonal)
```

Defense

Critical Defense:

By defending, we mean trying to prevent the opponent to win. As for attack, the agent need to act critically when the opponent is about to win without calculating many possibilities. If a possible quarto is on the quarto and the piece holded by the agent cannot win, therefore we need to block the opponent. Using the same methodology than before, we will block him.

```
1  def block_opponent(self, board_copy: np.ndarray) -> tuple[int, int
       ]:
2           """
3           TYPE OF STRATEGY: DEFENSE (critical action => Priority 1)
4           DESCRIPTION: if there are no open spaces on the board where
        placing a piece would allow you to create a row of four pieces
        that share a common attribute,
5           then place a piece in a space where it is least likely to
       help your opponent create such a row.
6
7           @returns: tuple[int, int]
8           """
9           for not_selected_pieces in range(self.get_game().NB_PIECES)
       :
10              if (not_selected_pieces != self.get_game().
       get_selected_piece())&(not_selected_pieces not in board_copy):
11                  scores = self.board_analysis(not_selected_pieces)
12                  position = self.possible_quarto(board_copy, scores)
13                  if position != [-1,-1]:
14                      return position
15          return [-1,-1]
```

Defense Scoring The scoring Strategy is the same as before, nevertheless, instead of comparing if the alignment is equal to the selected piece, we can see where is the best alignment where it doesn't share any characteristics to prevent a good alignment

4.2.2 Choosing Action

In the quarto game, the player choose the piece in the bag for the opponent. We can also implement some strategies to decrease the chance of winning of the opponent

Giving a wrong piece

This action has also a critical case. When the opponent can make a quarto, we will try to give him a piece that cannot do this quarto if possible. To that we will try to make the list of the position where it can be a quarto. And if it exists one we will "discard" it from the possible pieces to take inside the bag.

```
def giving_wrong_piece(self,  board_copy: np.ndarray) -> int:
        """
        TYPE OF STRATEGY: DEFENSE (Critical action ==> Priority 1)
        DESCRIPTION: If a possible quarto is coming, give a piece
    that could not make a quarto
        """
        list_possible_quarto = []
        bag = self.get_bag_pieces(board_copy)
        bag_index = []
        for piece in bag:
            index = list(range(self.get_game().NB_PIECES)).index(
    piece)
            scores = self.board_analysis(piece)
            pos = self.possible_quarto_list(board_copy,scores)
            bag_index.append(index)
            for i in range(self.get_game().BOARD_SIDE):
                for j in range(self.get_game().BOARD_SIDE):
                    #self.get_game().set_selected_piece(index)
                    if  [i,j] in pos :
                        list_possible_quarto.append(index)
        wrong_list = list(set(bag_index)-set(list_possible_quarto))
        if (len(wrong_list) != 0)&(len(list_possible_quarto)!=0):
            return wrong_list[random.randint(0, len(wrong_list)-1)]
        return -1
```

For the general case, we will score the position as for the placing action but in an other way. We will try to understand the sharing attributes for each piece and each alignment. Instead of looking the best free position, we will look at the highest pieces on the board. Then, we need to select in the bag, the piece that doesn't match with the best alignment. That is why, instead of scoring free position, the agent will score the piece in the bag according to those already in the boarder. In order to do that, we have 3 main functions.

Throughout we look at each position on the board (**givingwrongalignement()**), we will look the score of the pieces on the board according to their alignment . And with the list of score for this pieces we will compare them to the pieces inside the bag (**addnotquarto()**). Finally we will get the score of the piece that is less likely to make a quarto if selected, or, in other word, the piece in the bag with the lowest score.

```
def giving_wrong_piece_alignement(self,  board_copy: np.ndarray) ->
     int:
        """
        TYPE OF STRATEGY: DEFENSE
        DESCRIPTION: giving a piece that will not be able to be
    place in the best alignement
        """
        score = 0
        index = -1
```

```
        for i in range(self.get_game().BOARD_SIDE):
            for j in range(self.get_game().BOARD_SIDE):
                if board_copy[i,j] != -1:
                    score_ij = 0
                    pieces = {"h":[],"v":[],"d":[]}
                    ##Check the column
                    for a in range(self.get_game().BOARD_SIDE):
                        if (board_copy[i,a] != -1) & ((i,a)!=(i,j))
    :
                            scoring_alignement = self.
    share_attributes_pieces(board_copy,(i,j),(i,a))
                            score_ij += scoring_alignement
                            pieces["h"].append(board_copy[i,a])
                            pieces["h"].append(board_copy[i,j])
                    ##Check the column
                    for a in range(self.get_game().BOARD_SIDE):
                        if (board_copy[a,j] != -1) & ((a,j)!=(i,j))
    :
                            scoring_alignement = self.
    share_attributes_pieces(board_copy,(i,j),(i,a))
                            score_ij += scoring_alignement
                            pieces["v"].append(board_copy[a,j])
                            pieces["v"].append(board_copy[i,j])
                    ##Check the diagonal
                    if i == j:
                        for a in range(self.get_game().BOARD_SIDE):
                            if(board_copy[a,a] != -1)&((a,a) != (i,
    j)):
                                scoring_alignement = self.
    share_attributes_pieces(board_copy,(i,j),(i,a))
                                score_ij += scoring_alignement
                                pieces["d"].append(board_copy[a,a])
                                pieces["d"].append(board_copy[i,j])
                    elif i + j == 3:
                        for a in range(self.get_game().BOARD_SIDE):
                            if(board_copy[a, 3-a] != -1) &((a, 3-a)
    !=(i,j)):
                                scoring_alignement = self.
    share_attributes_pieces(board_copy,(i,j),(i,a))
                                score_ij += scoring_alignement
                                pieces["d"].append(board_copy[a, 3-
    a])
                                pieces["d"].append(board_copy[i,j])
                    if score_ij > score:
                        score = score_ij
                        index = self.add_not_quarto(board_copy,
    pieces)[0]
        return index

def add_not_quarto(self, board_copy:np.ndarray, lines: dict) -> int
    :
        """
        @args:
        lines: dictionary containing the list of index for each
    alignement type h,v,d
        """
        bag = self.get_bag_pieces(board_copy)
```

```
        bag_score = {}

        #initialize the score for each piece in the bag
        for k,v in bag.items():
            bag_score[k] = 0

        #transform each index in a Piece object
        pieces_by_align = {"h":[],"v":[],"d":[]}
        for k,v in lines.items():
            if v:
                for index in v:
                    pieces_by_align[k].append(self.get_game().
    get_piece_charachteristics(index))


        #evaluating which common attributes are most present in the
     alignement
        attributes = {"HIGH":list((None,0)),"COLOURED":list((None
    ,0)),"SOLID":list((None,0)),"SQUARE":list((None,0))}
        for k,v in pieces_by_align.items():
            #Look each pieces in the alignment
            characteristics = {"HIGH":True,"COLOURED":True, "SOLID"
    : True, "SQUARE":True}
            if not pieces_by_align[k]:
                characteristics = {}
            else:
                for piece in pieces_by_align[k]:
                    for piece2 in pieces_by_align[k]:
                        if piece != piece2:
                            if (piece.HIGH != piece2.HIGH) :
                                characteristics.pop("HIGH", None)
                            if (piece.COLOURED != piece2.COLOURED):
                                characteristics.pop("COLOURED",
    None)
                            if (piece.SOLID != piece2.SOLID):
                                characteristics.pop("SOLID", None)
                            if( piece.SQUARE != piece2.SQUARE):
                                characteristics.pop("SQUARE", None)
                for a,b in characteristics.items():
                    if (attributes[a][0] is None)&(pieces_by_align[
    k] != []):
                        attributes[a][0] = getattr(pieces_by_align[
    k][0],a)
                    attributes[a][1] += 1

        scores = dict(sorted(attributes.items(),key=lambda x: x
    [1][1], reverse=True ))

        #scoring each piece in the bag
        for attribute,score in scores.items():
            if score[1] != 0:
                for index,piece in bag.items():
                    if (attribute == "HIGH")&(piece.HIGH == score
    [0]):
                        bag_score[index] += 10 * score[1]
                    elif (attribute == "COLOURED")&(piece.COLOURED
    == score[0]):
```

```
                        bag_score[index] += 10 * score[1]
                    elif (attribute == "SOLID")&(piece.SOLID ==
    score[0]):
                        bag_score[index] += 10 * score[1]
                    elif (attribute == "SQUARE")&(piece.SQUARE ==
    score[0]):
                        bag_score[index] += 10 * score[1]

        bag_score = sorted(bag_score.items(),key=lambda x: x[1])
        return bag_score[0]

def share_attributes_pieces(self, board_copy: np.ndarray, p1: tuple
    [int, int], p2:tuple[int, int]):
        """return a score for a piece in the board compared to the
    chosen piece"""
        score = 0
        piece1 = self.get_game().get_piece_charachteristics(
    board_copy[p1[0],p1[1]])
        piece2 = self.get_game().get_piece_charachteristics(
    board_copy[p2[0],p2[1]])
        if piece1.HIGH == piece2.HIGH:
            score += 1
        if piece1.COLOURED == piece2.COLOURED:
            score += 1
        if piece1.SOLID == piece2.SOLID:
            score += 1
        if piece1.SQUARE == piece2.SQUARE:
            score += 1
        if score == 0:
            """if the score == 0, it means the piece is antagonist
    to the chosen one, so we will try to penalize this positions"""
            score = -1000
        return score
```

Giving a piece with the characteristic most present in the bag

Another strategy to give a piece to the opponent is more simple. We can only give a piece whith the characteristics most present in the bag. As if there is more pieces with this characteristics in the bag, it is less likely to make a quarto with this piece because they are not in the board.

```
def most_present_in_bag(self, board_copy: np.ndarray):
        """
        Choose a piece that is the most present in the bag because
    it means it is also the less present in tje board
        """
        bag = self.get_bag_pieces(board_copy)
        target =  dict(self.count_characteristics(bag))
        keys = list(target.keys())
        target[keys[0]] *= 1000
        target[keys[1]] *= 500
        target[keys[2]] *= 100
        target[keys[3]] *= 50
        target[keys[4]] *= 10
        target[keys[5]] *= 5
        target[keys[6]] *= 1
```

```
        target[keys[7]] *= 0.1
    #(attribut, quantit )
        pieces = {}
        for index in bag:
            score = 0
            if (bag[index].HIGH) & (target == "high"):
                score += 1 * target["high"]
            elif (not bag[index].HIGH) & (target == "small"):
                score += 1 * target["small"]
            if (bag[index].COLOURED) & (target == "color"):
                score += 1 * target["color"]
            elif (not bag[index].COLOURED) & (target == "not_color"
    ):
                score += 1 * target["not_color"]
            if (bag[index].SOLID) & (target == "solid"):
                score += 1 * target["solid"]
            elif (not bag[index].SOLID) & (target == "hollow"):
                score += 1 * target["hollow"]
            if (bag[index].SQUARE) & (target == "square"):
                score += 1 * target["square"]
            elif (not bag[index].SQUARE) & (target == "round"):
                score += 1 * target["round"]
            pieces[index] = score
        return sorted(pieces.items(), key=lambda x: x[1],reverse =
    True)[0][0]

def count_characteristics(self,bag:dict):
        characteristics = {"high":0, "small":0, "color": 0, "
    not_color":0, "solid": 0, "hollow": 0, "square":0,"round":0}
        for piece in bag:
            if bag[piece].HIGH:
                characteristics["high"] += 1
            else:
                characteristics["small"] += 1
            if bag[piece].COLOURED:
                characteristics["color"] += 1
            else:
                characteristics["not_color"] += 1
            if bag[piece].SOLID:
                characteristics["solid"] += 1
            else:
                characteristics["hollow"] += 1
            if bag[piece].SQUARE:
                characteristics["square"] += 1
            else:
                characteristics["round"] += 1

        return sorted(characteristics.items(), key=lambda x: x[1],
    reverse = True)
```

4.2.3 First Results

I test this rules against 2 players, the random one provided by the teacher and a "dumb" player that can only block and make quarto when there is a 3 alignment.

The score where great against random as expected. I always got an average of 98 percent more or less of victory, 1 percent of draw and 1 percent of loose.

Against the "dumb" player. There are mostly draws, more or less 45 percents. It is interesting to note that the percentage of victory and loose depends on the order of the turns. If the rule based is the first one, therefore it wins more than it loose. It is a proof of the importance of the final steps of the quarto game to be separated from more complex calculation when there is no possibility to make a quarto.

Setting the parameters to 0.5, 0.5 between the strategies we obtain the following table

Plays between Agents	
Name	Results
RuleBased vs Random	980W(0.98) / 12D(0.12) / 8L(0.008)
Random vs RuleBased	980W(0.98) / 10D(0.12) / 10L(0.01)
RuleBased vs Dumb	318W(0.32) / 409D(0.41) / 273L(0.27)
Dumb vs RuleBased	244W(0.24) / 388D(0.39) / 368L(0.37)

4.3 Evolving the Rules

In the previous section, we discussed about the rules implemented. Both actions own two type of strategy. As we don't know which one to use in a first moment, we can parameterize the use of this rules.

I had give paraemeter in a sequential order to get the best restults: We want to know the best parameter value to decide:

- Placing Strategy: Attack or Defend

```
def placing_strategy(self) -> tuple[int,int]:
    """
    Description: Placing a piece on the board

    @returns
    tuple (i,j): the position where to place the piece
    """
    board_copy = self.get_game().get_board_status()
    completing = self.completing_raw(board_copy)
    if completing != [-1,-1]:
        return completing
    else:
        block = self.block_opponent(board_copy)
        if block != [-1,-1]:
            return block
        elif self.params1 != 0:
            if random.random() > self.params1:
                region = self.scoring_strategy(board_copy)
            else:
                region = self.scoring_strategy(board_copy,
    True)
            if region != [-1,-1]:
                return region
```

```
            return self.oriented_random()
        else:
            """
            Dumb Rule based
            """
            return self.oriented_random()
```

- Choosing Strategy: Most frequent characteristic in the bag or Worst piece to play

```
def choosing_strategy(self):
    board_copy = self.get_game().get_board_status()
    index = self.giving_wrong_piece(board_copy)
    if index != -1:
        return index
    else:
        if self.params2 != 0:
            if random.random() >= self.params2:
                index = self.most_present_in_bag(board_copy)
            else:
                index = self.giving_wrong_piece_alignement(
    board_copy)

        if index != -1:
            return index
        else:
            return random.randint(0,15)
```

As seen for the Lab2, we will use a genetic algorithm to evolve this rules

4.3.1 Individuals

In this case an individual or genome is a list of two float parameters.

4.3.2 Fitness

The fitness is the number of wins of the individual over 2 kind of player as was mentioned by a reviewer in the Lab3 of NIM. As I want to train my algorithm over different level of difficulty, I created two players to train my individuals against. The first one is the one totally random provided by the teacher. The second one, is a naive version of my Rule based system. It try to win when there is a obvious solution, but he plays randomly otherwise.

```
NUM_MATCHES = 100
def fitness(genome: list):
    winners = []
    for i in range(NUM_MATCHES):
        game = main.quarto.Quarto()
        if i % 2 == 0:
            Player1 = ruleBased.RuleBased(game,0,0)
        else:
            Player1 = main.RandomPlayer(game)
        game.set_players((Player1,ruleBased.RuleBased(game,genome
    [0],genome[1])))
```

```
        winner = game.run()
        winners.append(winner)
    return winners.count(1)/NUM_MATCHES
```

4.3.3 Cross-Over

We will perform a cross-over between the parameters of both parent.

```
def cross_over(g1: list, g2:list) -> list:
    g3 = [g1[0],g2[1]]
    g4 = [g1[1],g2[0]]
    return random.choice([g3,g4])
```

4.3.4 Mutation

We will perform a mutation on one of the two parameter, applying a random mutation on it

```
def mutation(g:list):
    params = random.randint(0, 1)
    g[params] = random.random()
    return g
```

4.3.5 Tournament

We will get the maximum fitness between two Individuals

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

4.3.6 Evolution

```
def create_rand_pop(quantity:int) -> list:
    population = []
    for i in range(quantity):
        a = random.random()
        b = random.random()
        genome = [a,b]
        fitness_value = fitness(genome)
        population.append(Individual(genome, fitness_value))
        print(f"{i+1}/{quantity} of the population created")
    return population

def evolution(population: list,POPULATION_SIZE: int,
    NUM_GENERATIONS:int, OFFSPRING_SIZE:int):
    for g in range(NUM_GENERATIONS):
        print(g)
        offspring = list()
        for i in range(OFFSPRING_SIZE):
            if random.random() < 0.3:
```

```
                p = tournament(population)
                o = mutation(p.genome.copy())
            else:
                p1 = tournament(population)
                p2 = tournament(population)
                o = cross_over(p1.genome.copy(),p2.genome.copy())
            f = fitness(o)
            offspring.append(Individual(o, f))
        population+=offspring
        population = sorted(population, key=lambda indi: indi.
    fitness,reverse=True)[:POPULATION_SIZE]
    return population[0]
```

4.3.7 Results

The evolution doesn't work very well and results in high variability in the final parameters. Maybe, it is not training against good agent for the fitness. That is why for the final agent, I decided to implement a risky agent with this parameter that use the following parameters: 0.12077208930524375, 0.47262121794293277. The results against the dumb are random and stay in the same order of size (0.30W-0.30L-0.40D)

Plays between Agents	
Name	Results
RuleBased vs Random	981W(0.98) / 10D(0.10) / 9L(0.009)
Random vs RuleBased	988(0.99) / 10D(0.12) / 6L(0.006)
RuleBased vs Dumb	315(0.32) / 371D(0.37) / 314L(0.31)
Dumb vs RuleBased	308(0.31) / 332D(0.33) / 360L(0.36)

I put the final best parameter as default parameters in the rule based agent class