

# Computational Intelligence

Thomas Osorio - s307107

January 2023

# Contents

<b>1 Lab 1 - Set covering Problem using Single-state Methods</b>	<b>3</b>
1.1 Task . . . . .	4
1.2 Solutions - Before Review . . . . .	4
1.3 Results - Before Review . . . . .	7
1.3.1 SOLUTION 1: THE BEST . . . . .	7
1.3.2 SOLUTION 2: . . . . .	8
1.3.3 SOLUTION 3: . . . . .	9
1.3.4 Solution 4 . . . . .	9
1.4 Reviews . . . . .	10
1.4.1 Review 1 by GiuseppeEsposito98 . . . . .	10
1.4.2 Review 2 by Riccardo Musumarra . . . . .	10
1.5 Results - After Review . . . . .	15
1.6 My Reviews . . . . .	15
1.6.1 Laio95 . . . . .	15
1.6.2 SalvatoreAdalberto . . . . .	16
<b>2 Lab 2 - EA for Set Covering problem</b>	<b>17</b>
2.1 Task . . . . .	18
2.2 Solutions - before Review . . . . .	18
2.2.1 Fitness . . . . .	18
2.2.2 Tournament . . . . .	18
2.2.3 Cross-OVER . . . . .	18
2.2.4 Mutation . . . . .	18
2.2.5 Infeasible Solutions . . . . .	19
2.2.6 Population initialization . . . . .	19
2.2.7 Evolution . . . . .	20
2.3 Results - Before Review . . . . .	20
2.3.1 N 5 to 100 . . . . .	20
2.3.2 N 500 and 1000 . . . . .	21
2.4 Reviews . . . . .	22
2.4.1 Review 1 by bred91 . . . . .	22
2.4.2 Review 2 by Gabriele Greco s303435 . . . . .	22
2.4.3 Review 3 Francesco Sattolo . . . . .	23
2.5 Results - After Review . . . . .	24

2.5.1	Results . . . . .	24
2.6	My Reviews . . . . .	26
2.6.1	mistr97 . . . . .	26
<b>3</b>	<b>Lab 3 - Playing Nim</b>	<b>27</b>
3.1	Task . . . . .	28
3.2	Solution - Before Review . . . . .	28
3.2.1	The game . . . . .	28
3.2.2	Rule Based Agent . . . . .	28
3.2.3	Evolved Agent . . . . .	34
3.2.4	Minimax . . . . .	37
3.2.5	Reinforcement Learning . . . . .	38
3.3	Reviews . . . . .	39
3.3.1	Review 1 by Francesco Sattolo . . . . .	39
3.4	Results - After Review . . . . .	39
3.4.1	MinMax Improved . . . . .	39
3.5	My Reviews . . . . .	41
3.5.1	bred91 . . . . .	41
3.5.2	antoniodecinque99 . . . . .	41
<b>4</b>	<b>Final Project - The Quarto Game</b>	<b>43</b>
4.1	The game . . . . .	44
4.2	Agent - Rule Based System . . . . .	45
4.2.1	Placing Action . . . . .	45
4.2.2	Choosing Action . . . . .	51
4.2.3	First Results . . . . .	56
4.3	Evolving the Rules . . . . .	57
4.3.1	Individuals . . . . .	58
4.3.2	Fitness . . . . .	58
4.3.3	Cross-Over . . . . .	59
4.3.4	Mutation . . . . .	59
4.3.5	Tournament . . . . .	59
4.3.6	Evolution . . . . .	59
4.3.7	Results . . . . .	60

## 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-2022-01URR0V-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  $P = (L_0, L_1, L_2, \dots, L_n)$ , determine, if possible,  $S = (L_{s_0}, L_{s_1}, L_{s_2}, \dots, L_{s_n})$  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

```
1 class State:
2     """
3     The state at a moment of the search
4     defined by an array
5     """
6     def __init__(self, arr=[]):
7         self._array = arr.copy()
8
9     def __hash__(self):
10        return hash(np.array(self._array).tobytes())
11
12    def __eq__(self, other):
13        return sum(len(_) for _ in self._array) == sum(len(_) for _
14        _ in other._array)
15
16    def __lt__(self, other):
17        return sum(len(_) for _ in self._array) < sum(len(_) for _
18        _ in other._array)
19
20    @property
21    def array(self):
22        return self._array
23    @array.setter
24    def array(self, value):
25        self._array = value
```

### Search function

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

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

```

5 def result(state, a):
6     array = state.array.copy()
7     array.append(a)
8     return State(array)
9
10 from gx_utils import PriorityQueue
11
12 def possible_actionsV2(all_lists, covering):
13     return [array for array in all_lists if len(set(array)).
14         difference(covering)) > 0]
15
16 logging.basicConfig(format="%(message)s", level=logging.INFO)
17 def searchV2(N, initial_state, goal_test, parent_state, state_cost,
18             priority_function, unit_cost):
19     """
20     The search function
21     """
22     state = initial_state
23     parent_state[state] = None
24     state_cost[state] = 0
25     solutions = []
26     covering = set()
27     step = 0
28     all_lists = sorted(problem(N, seed=42), key=lambda l: len(l))
29     while state is not None and covering != goal_test:
30         frontier = PriorityQueue()
31         for a in possible_actionsV2(all_lists, covering):
32             new_state = result(state, a)
33             cost = unit_cost(new_state)
34             if new_state not in state_cost and new_state not in
35                 frontier:
36                 parent_state[new_state] = state
37                 state_cost[new_state] = state_cost[state] +
38                 cost
39                 frontier.push(new_state, p=priority_function(
40                     new_state))
41                 elif new_state in frontier and state_cost[new_state]
42                     ] > state_cost[state] + cost:
43                     old_cost = state_cost[new_state]
44                     parent_state[new_state] = state
45                     state_cost[new_state] = state_cost[state] +
46                     cost
47                     if frontier:
48                         state = frontier.pop()
49                         step = step + 1
50                         solutions.append(state.array)
51                         covering |= set([el for a in state.array for el in a])
52                         else:
53                             state = None
54                         path = list()
55                         s = state
56                         i = 0
57                         while s:
58                             path.append(s)
59                             s = parent_state[s]
60                         logging.info(f"NEW SOL: Found a solution in {step} nodes; ")
61                         return state.array, list(reversed(path))

```

## Algoirthms

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

```
1 def greedy_first(N):
2     def h(state:State):
3         goal = set(range(N))
4         return len(goal.difference(set([el for array in state.array
5             for el in array])))
6         final = search(
7             N,
8             State(),
9             goal_test=goal_test,
10            parent_state=dict(),
11            state_cost=dict(),
12            priority_function=lambda s: h(s) ,
13            unit_cost=lambda a: 1
14        )
15        logging.info(
16            f"greedy_best_first first solution for N={N}: w={sum(
17                len(_) for _ in final[0])} (bloat={(sum(len(_)) for _ in final
18                [0])-N)/N*100:.0f}%)"
19        )
20    logging.getLogger().setLevel(logging.INFO)
21    for N in [5,10,20,100]:
22        greedy_first(N)
23
24    def a_star(N):
25        def h(state:State):
26            goal = set(range(N))
27            return len(goal.difference(set([el for array in state.array
28                for el in array])))
29            state_cost = dict()
30            final = search(
31                N,
32                State(),
33                goal_test=goal_test ,
34                parent_state=dict(),
35                state_cost=state_cost ,
36                priority_function=lambda s: h(s) + state_cost[s] ,
37                unit_cost=lambda a: len(a)
38        )
39        logging.info(
40            f"greedy_best_first first solution for N={N}: w={sum(
41                len(_) for _ in final[0])} (bloat={(sum(len(_)) for _ in final
42                [0])-N)/N*100:.0f}%)"
43        )
44    logging.getLogger().setLevel(logging.INFO)
45    for N in [5,10,20]:
46        a_star(N)
47
48    def breadth_first(N):
49        def h(state:State):
50            goal = set(range(N))
51            return len(goal.difference(set([el for array in state.array
52                for el in array])))
53            state_cost = dict()
```

```

47     final = search(
48         N,
49         State(),
50         goal_test=goal_test,
51         parent_state=dict(),
52         state_cost=state_cost,
53         priority_function=lambda s: len(state_cost),
54         unit_cost=lambda a: 1
55     )
56     logging.info(
57         f"greedy_best_first first solution for N={N}: w={sum(
58             len(_) for _ in final[0])} (bloat={(sum(len(_)) for _ in final
59             [0])-N)/N*100:.0f}%)"
60     )
61
62     logging.getLogger().setLevel(logging.INFO)
63     for N in [5,10,20]:
64         breadth_first(N)
65
66     def depth_first(N):
67         INITIAL_STATE = State()
68         parent_state = dict()
69         state_cost = dict()
70         covering = set()
71         goal = set(range(N))
72
73         final = searchV2(
74             N,
75             INITIAL_STATE,
76             goal_test=goal,
77             parent_state=parent_state,
78             state_cost=state_cost,
79             priority_function=lambda s: -len(state_cost) ,
80             unit_cost=lambda a: 1
81         )
82         logging.info(
83             f"breadth_first first solution for N={N}: w={sum(len(_)
84                 for _ in final[0])} (bloat={(sum(len(_)) for _ in final[0])-N)/
85                 N*100:.0f}%)"
86     )

```

## 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 heterogeneous and non-contiguous in memory, on the other hand numpy.ndarrays are homogeneous and contiguous so each access is easier. Moreover numpy 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

```
1 class State:
2     """
3         The state at a moment of the search
4         defined by an array
5     """
6
7     def __init__(self, arr=[]):
8         self._array = arr
9
10    def __hash__(self):
11        return hash(bytes(np.array(self._array)))
12
13    def __eq__(self, other):
14        return (bytes(np.array(self._array)) == bytes(np.array(
15            other._array))) & (([sum(_) for _ in self._array]) == ([sum(_)
16                for _ in other._array]))
17
18    def __lt__(self, other):
19        return ([sum(_) for _ in self._array]) < ([sum(_) for _ in
20            other._array])
21
22    def __str__(self):
```

```

19         return str(self._array)
20
21     def __repr__(self):
22         return repr(self._array)
23
24     @property
25     def array(self):
26         return self._array
27     @array.setter
28     def array(self,value):
29         self._array = value
30
31
32     def possible_actions(all_lists: list, covering: set):
33         """
34             Description: return all the possibles actions sorted according
35             each array
36
37             @args:
38                 all_lists [list]: the search space
39                 covered [set]: the numbers already covered
40
41             return sorted([array for array in all_lists if len(set(array)).
42                         difference(covering)) > 0], key=lambda l: len(l) )
43
44     def goal_test(state: State(),goal: set):
45         """
46             Description: testing if we already found every number
47
48             @args:
49                 covered [set]: the numbers already covered
50                 goal [set]: the numbers to found
51
52             return set([el for array in state.array for el in array]) ==
53                     goal
54
55     def result(state: State, a:list):
56         array = copy.deepcopy(state)._array
57         array.append(a)
58         return State(array)
59
60     def search(
61         N: int,
62         initial_state: State,
63         goal_test: Callable,
64         parent_state: dict,
65         state_cost: dict,
66         priority_function: Callable,
67         unit_cost: Callable,
68     ):
69         frontier = PriorityQueue()
70         parent_state.clear()
71         state_cost.clear()
72
73         state = initial_state
74         parent_state[state] = None
75         state_cost[state] = 0

```

```

73     covered = set()
74     all_lists = sorted(problem(N, seed=42), key=lambda l: len(l))
75     step = 0
76     #print(f"SEARCH SPACE {search_space}")
77     while state is not None and not goal_test(state, set(range(N))):
78     :
79         for a in possible_actions(all_lists, covered):
80             new_state = result(state, a)
81             cost = unit_cost(a)
82             if new_state not in state_cost and new_state not in
83             frontier:
84                 parent_state[new_state] = state
85                 state_cost[new_state] = state_cost[state] + cost
86                 frontier.push(new_state, p=priority_function(
87                     new_state))
88                 elif new_state in frontier and state_cost[new_state] >
89                     state_cost[state] + cost:
90                     old_cost = state_cost[new_state]
91                     parent_state[new_state] = state
92                     state_cost[new_state] = state_cost[state] + cost
93                 if frontier:
94                     state = frontier.pop()
95                     step = step + 1
96                 else:
97                     state = None
98                     return "No solution"
99
100                path = list()
101                s = state
102                i = 0
103                while s:
104                    path.append(s)
105                    s = parent_state[s]
106                if N <=20:
107                    logging.info(f"NEW SOL: Found a solution in {step} nodes; {state.__str__()}")
108                else:
109                    logging.info(f"NEW SOL: Found a solution in {step} nodes; ")
110                return state.array, list(reversed(path))
111
112    def greedy_first(N):
113        def h(state:State):
114            goal = set(range(N))
115            return len(goal.difference(set([el for array in state.array
116                for el in array])))
117        final = search(
118            N,
119            State(),
120            goal_test=goal_test,
121            parent_state=dict(),
122            state_cost=dict(),
123            priority_function=lambda s: h(s) ,
124            unit_cost=lambda a: 1
125        )
126        logging.info(
127            f"greedy_best_first solution for N={N}: w={sum(

```

```

123     len(_) for _ in final[0])} (bloat={{sum(len(_)
124         for _ in final[0])-N)/N*100:.0f}}%)"
125     )
126 logging.getLogger().setLevel(logging.INFO)
127 for N in [5,10,20,100]:
128     greedy_first(N)
129
130 def a_star(N):
131     def h(state:State):
132         goal = set(range(N))
133         return len(goal.difference(set([el for array in state.array
134             for el in array])))
135         state_cost = dict()
136         final = search(
137             N,
138             State(),
139             goal_test=goal_test,
140             parent_state=dict(),
141             state_cost=state_cost,
142             priority_function=lambda s: h(s) + state_cost[s] ,
143             unit_cost=lambda a: len(a)
144         )
145         logging.info(
146             f"greedy_best_first first solution for N={N}: w={sum(
147                 len(_) for _ in final[0])} (bloat={{sum(len(_)
148                     for _ in final[0])-N)/N*100:.0f}}%)"
149             )
150 logging.getLogger().setLevel(logging.INFO)
151 for N in [5,10,20]:
152     a_star(N)
153
154 def breadth_first(N):
155     def h(state:State):
156         goal = set(range(N))
157         return len(goal.difference(set([el for array in state.array
158             for el in array])))
159         state_cost = dict()
160         final = search(
161             N,
162             State(),
163             goal_test=goal_test,
164             parent_state=dict(),
165             state_cost=state_cost,
166             priority_function=lambda s: len(state_cost),
167             unit_cost=lambda a: 1
168         )
169         logging.info(
170             f"greedy_best_first first solution for N={N}: w={sum(
171                 len(_) for _ in final[0])} (bloat={{sum(len(_)
172                     for _ in final[0])-N)/N*100:.0f}}%)"
173             )
174 logging.getLogger().setLevel(logging.INFO)
175 for N in [5,10,20]:
176     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(_)
192         for _ in final[0])} (float={(sum(len(_)) for _ in final[0])-N)/
193         N*100:.0f})"
194     )
194 logging.getLogger().setLevel(logging.INFO)
195 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 first	best	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.
- Moreover 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 necessarily the best solution. For instance, if you have the biggest one as  $L_{max} = [1, \dots, N]$  instead of each array in  $[[1], [2], \dots, [N]]$  (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 interesting to have a look also on different test you did. It is always interesting to compare algorithms or different 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 solution 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  $P = (L_0, L_1, L_2, \dots, L_n)$ , determine, if possible,  $S = (L_{s_0}, L_{s_1}, L_{s_2}, \dots, L_{s_n})$  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( $N$ ).

```
1 def fitness(genome: list, N: int):
2     """ Definition of the fitness """
3     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.

```
1 def tournament(population: list, tournament_size=2):
2     """Tournament function"""
3     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.

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

### 2.2.4 Mutation

Three types of mutation:

- if  $\text{len}(\text{genome}) < \text{Len}(\text{goal}) \implies$  insertion of random array from the initial set
- if  $\text{len}(\text{genome}) > \text{len}(\text{goal}) \implies$  deletion of a random array

- if  $\text{len}(\text{genome}) == \text{len}(\text{goal}) \Rightarrow$  change random value with one from the initial set

```

1 def mutation(g: list, goal: set, initial_set:list):
2     if sum(len(_) for _ in g) > len(goal):
3         point = random.randint(0, len(g)-1)
4         del g[point]
5     elif sum(len(_) for _ in g) < len(goal):
6         point = random.randint(0, len(initial_set)-1)
7         g.append(initial_set[point])
8     else:
9         point1 = random.randint(0, len(g)-1)
10        point2 = random.randint(0, len(initial_set)-1)
11        del g[point1]
12        g.append(initial_set[point2])
13
14    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 minimization problem, we will try to maximize this values
- According, we will provide a factor of  $100 * N$ .

```

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

```

### 2.2.6 Population initialization

```

1 Individual = namedtuple("Individual", ["genome", "fitness"])
2 def initial_population(N: int, POP_SIZE: int, seed = 42):
3     """ Initialisation of the population according to feasibility
4     """
5     population = list()
6     initial_set = problem(N,seed)
7     GOAL = set(list(range(N)))
8     for values in range(POP_SIZE):
9         genome = random.sample(initial_set,random.randint(1,N)).copy()
10        # The ideal solution is N array of length 1 with all value
11        if feasible(genome,GOAL):
12            population.append(Individual(genome,fitness(genome,N)))
13        else:
14            population.append(Individual(genome, penalty(fitness(
15                genome,N),N)))

```

```

13     logging.info(f"init: N={N} pop_size={len(population)};\n"
14     min_bloat_init={min(population, key=lambda i: abs(i.fitness))
15     [1]}")
16     return population,initial_set

```

### 2.2.7 Evolution

```

1 fitness_log = []
2 def evolution(N: int, population: list, POPULATION_SIZE: int,
3     NUM_GENERATIONS: int, OFFSPRING_SIZE: int, initial_set: list):
4     GOAL = set(list(range(N)))
5     for g in range(NUM_GENERATIONS):
6         offspring = list()
7         for i in range(OFFSPRING_SIZE):
8             if random.random() < 0.3:
9                 p = tournament(population)
10                o = mutation(p.genome, GOAL, initial_set)
11            else:
12                p1 = tournament(population)
13                p2 = tournament(population)
14                o = cross_over(p1.genome, p2.genome)
15            if feasible(o, GOAL):
16                f = fitness(o, N)
17            else:
18                f = penalty(fitness(o, N), N)
19            offspring.append(Individual(o, f))
20        population+=offspring
21        population = sorted(population, key=lambda indi: abs(indi.
22        fitness))[:POPULATION_SIZE]
23        #plot_result(fitness_log,10*N)
24    return population
25
26 def algorithm(N, pop_size, num_gen, offsize):
27     POPULATION_SIZE = pop_size
28     OFFSPRING_SIZE = offsize
29     NUM_GENERATIONS = num_gen
30     ### Initialisation of the problem
31     population, initial_set = initial_population(N, POPULATION_SIZE)
32     ### Evolution
33     return evolution(N, population, POPULATION_SIZE, NUM_GENERATIONS,
34     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 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 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 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 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 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 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 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.

#### MAJOR

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, offspring and number of generations for small N and big N

- I didn't quite understand 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 ( $(w-N)/N^2$ ), 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 performance evaluation code, but no plots were shown 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.

```
1 fitness_log = []
2 def evolution(N: int, population: list, POPULATION_SIZE: int,
3     NUM_GENERATIONS: int, OFFSPRING_SIZE: int, initial_set: list):
4     GOAL = set(list(range(N)))
5     for g in range(NUM_GENERATIONS):
6         offspring = list()
7         for i in range(OFFSPRING_SIZE):
8             if random.random() < 0.3:
9                 p = tournament(population)
10                o = mutation(p.genome.copy(), GOAL, initial_set)
11            else:
12                p1 = tournament(population)
13                p2 = tournament(population)
14                o = cross_over(p1.genome.copy(), p2.genome.copy())
15                if feasible(o, GOAL):
16                    f = fitness(o, N)
17                else:
18                    f = penalty(fitness(o, N), N)
19                offspring.append(Individual(o, f))
20            population += offspring
21            population = sorted(population, key=lambda indi: abs(indi.
22            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; minloatinit = 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; minloatinit = 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; minloatinit = 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; minloatinit = 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; minloatinit = 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; minloatinit = 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; minloatinit = 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 Reinforcement 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

```
1  def __init__(self, id, params = [0.2, 0.16, 0.16, 0.16, 0.16, 0.16]):  
2      super().__init__(id)  
3      self._params = params  
4  
5      @property  
6      def params(self):  
7          return self._params  
8      @params.setter  
9      def name(self, value):  
10         self._params = value  
11  
12     def cooked_rules(self, state: Nim, moves: list):  
13         cooked = dict()  
14         moves_row = [el for el in state.rows if el != 0]  
15  
16         # SIMPLE RULES  
17         simple = dict()  
18  
19         shortest_row = min((x for x in enumerate(state.rows) if x  
20             [1] > 0), key=lambda y: y[1])[0] #index shortest row (In class)  
21         longest_row = max((x for x in enumerate(state.rows)), key=  
22             lambda y: y[1])[0]  
23         if state.k:  
24             simple["shortest_row"] = Nimply(shortest_row, self.  
choose_nim_quantity(state, min(state.rows[shortest_row], state.k))  
))  
25             simple["longest_row"] = Nimply(longest_row, self.  
choose_nim_quantity(state, min(state.rows[shortest_row], state.k))  
)) #index longest row (In class)  
26             else:  
27                 simple["shortest_row"] = Nimply(shortest_row, self.  
choose_nim_quantity(state, state.rows[shortest_row]))  
28                 simple["longest_row"] = Nimply(longest_row, self.  
choose_nim_quantity(state, state.rows[shortest_row])) #index  
longest row (In class)  
29  
30         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
30         same action as the opponent
31         cooked["simple"] = simple
32
33         #GENERALIZED RULES
34         cooked["remaining_parity_row"] = self.action_strategy_row(
35             state,cooked,moves_row)
36         cooked["remaining_quantity_row"] = self.
37             action_strategy_quantity(state,moves_row)
38         cooked["reaction_longest_shortest"] = self.
39             reaction_strategy_longest_vs_shortest(state,cooked)
40         cooked["reaction_quantity"] = self.
41             reaction_strategy_quantity(state,moves_row)
42         cooked["random"] = self.pure_random(state)
43
44         return cooked
45
46     def choose_nim_quantity(self,state: Nim, row: int):
47         if (state.k):
48             if (row > state.k):
49                 return random.randint(1,state.k)
50             return random.randint(1,row)
51
52     def clear_row(self,state: Nim, row_value: int):
53         """action of clear a row"""
54         return Nimply(state.rows.index(row_value),row_value)
55
56     def keep_one(self,state: Nim, row_value: int):
57         """Try to keep the last one for the player"""
58         return Nimply(state.rows.index(row_value),min(int(row_value
59             - (state.k + 1)),state.k))
60
61     def action_strategy_row(self, state: Nim, rules: dict,
62         possible_rows: list):
63         """
64             According to the parity of the remaining rows
65         """
66
67         ply = None
68         non_empty_row = possible_rows.copy()
69
70         if (len(non_empty_row)%2 == 0):
71             """If the number of rows remaining is pair we don't
72             want to empty a row"""
73             non_empty_row = sorted(non_empty_row, reverse = True) #
74             It is harder to empty the longest one.
75             if state.k:
76                 if non_empty_row[0] <= state.k:
77                     """If the row size is less than the state.k, we
78                     only remove 1"""
79                     ply = Nimply(state.rows.index(non_empty_row[0])
80 ,1)
81                 else:
82                     ply = self.keep_one(state, non_empty_row[0])
83             else:
84                 """If k is not defined we will try to force the
85                 opponent to take the last one"""
86                 longest = rules["simple"]["longest_row"][0]
87                 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
77         empty a row"""
78         non_empty_row = sorted(non_empty_row) # It is easier to
79         remove the shortest rows
80         if state.k:
81             if non_empty_row[0] <= state._k :
82                 """The player can nim all pieces"""
83                 ply = self.clear_row(state,non_empty_row[0])
84             else:
85                 """Try to keep the last one for the player and
86                 not the opponent"""
87                 ply = self.keep_one(state,non_empty_row[0]) #
88                 Try to go to a quantity of state.k + 1 in the last row if not
89                 remove state.k
90             else:
91                 ply = Nimply(rules["simple"]["shortest_row"][0],
92                 non_empty_row[0])
93
94     return ply
95
96
97 def action_strategy_quantity(self, state: Nim,possible_rows:
98     list):
99     """
100     Strategy according to the reamining quantity of nim
101     """
102
103     ply = None
104     non_empty_row = possible_rows.copy()
105     if(sum(state.rows)%2 ==0):
106         """If the quantity remaining is paired we want to keep
107         a paired quantity"""
108         valid_rows = [el for el in non_empty_row if el > 1] #>
109         1 because we want to nim a paired quantity
110         if valid_rows:
111             row = valid_rows[random.randint(0,len(valid_rows)
112             -1)] # Select a random row where we can remove a pair number
113             if state.k:
114                 if state.k == 1:
115                     """This strategy focus on the quantity to
116                     nim so if state.k == 1 is the only case were we have only
117                     unpaired quantity to remove"""
118                     nimming =[1]
119                 else:
120                     nimming = [el for el in range(1,state.k+1)
121                     if (el%2==0) & (el <= row)] #Nim only unpaired quantity
122                     possible in the selected row
123                     else:
124                         if row == 1:
125                             nimming = [1]
126                         else:
127                             nimming = [el for el in range(1,row+1) if (
128                             el%2==0) & (el <= row)] #Nim only unpaired quantity possible in
129                             the selected row
130                             ply = Nimply(state.rows.index(row), sorted(nimming,
131                             reverse=True)[0])
132                         else:

```

```

114         row = non_empty_row[random.randint(0, len(
115             non_empty_row)-1)] #Select a random non empty row
116         if state.k:
117             if state.k == 1:
118                 """This strategy focus on the quantity to
119                 nim so if state.k == 1 is the only case were we have only
120                 unpaired quantity to remove"""
121                 nimming = [1]
122             else:
123                 nimming = [el for el in range(1, state.k+1)
124                         if (el%2!=0) & (el <= row)] #Nim only paired quantity possible
125                         in the selected row
126             else:
127                 if row == 1:
128                     nimming = [1]
129                 else:
130                     nimming = [el for el in range(1, row+1) if (
131                         el%2!=0) & (el <= row)] #Nim only paired quantity possible in
132                         the selected row
133             ply = Nimply(state.rows.index(row), sorted(nimming)[0])
134         else:
135             """If the quantity remainig is unpaired we want to get
136             a paired quantity"""
137             row = non_empty_row[random.randint(0, len(non_empty_row)
138                 -1)] #Select a random non empty row
139             if state.k:
140                 if state.k == 1:
141                     """This strategy focus on the quantity to nim
142                     so if state.k == 1 is the only case were we have only unpaired
143                     quantity to remove"""
144                     nimming = [1]
145                 else:
146                     nimming = [el for el in range(1, state.k+1) if (
147                         el%2!=0) & (el <= row)] #Nim only paired quantity possible in
148                         the selected row
149             else:
150                 if row == 1:
151                     nimming = [1]
152                 else:
153                     nimming = [el for el in range(1, row+1) if (el
154                         %2!=0) & (el <= row)] #Nim only paired quantity possible in the
155                         selected row
156             ply = Nimply(state.rows.index(row), sorted(nimming)[0])
157         return ply
158
159     def reaction_strategy_longest_vs_shortest(self, state: Nim,
160                                                 rules: dict):
161         """
162             The player will play in a different domain than the
163             opponent
164         """
165         if state.last_action:
166             if state.row[state.last_action[0]] <= sum(state.rows)
167             //2:
168                 """The oppponent nimmmed shortest rows so I want to
169                 play on the longest"""

```

```

151             row = state.rows[rules["longest_row"]]]#Select the
152             longest row
153             if state.k:
154                 if state.k == 1:
155                     nimming = [1]
156                 else:
157                     nimming = [el for el in range(1,state.k+1)
158 if el <= row] #Nim quantity possible in the selected row
159                     else:
160                         nimming = [el for el in range(1,row+1) if el <=
161 row] #Nim quantity possible in the selected row
162                         ply = Nimply(state.rows.index(row), nimming[0])
163                         else:
164                             """The oppponent nimmed longest row so I want to
165 play on the shortest"""
166                             row = state.rows[rules["shortest_row"]]]#Select the
167                             shortest row
168                             if state.k:
169                                 if state.k == 1:
170                                     nimming = [1]
171                                 else:
172                                     nimming = [el for el in range(1,state.k+1)
173 if el <= row] #Nim quantity possible in the selected row
174                                     else:
175                                         if row == 1:
176                                             nimming = [1]
177                                         else:
178                                             nimming = [el for el in range(1,row+1) if
179 el <= row] #No quantity possible in the selected row
180                                             ply = Nimply(state.rows.index(row), nimming[0])
181                                             return ply
182                                         else:
183                                             return self.pure_random(state)
184
185     def reaction_strategy_quantity(self, state: Nim, possible_rows):
186         """
187             The player will play different quantity than the oppponent
188         """
189         non_empty_row = possible_rows.copy()
190         if state.last_action:
191             row = non_empty_row[random.randint(0, len(non_empty_row)
192 )-1]]#Select a random row
193             if state.k == 1:
194                 """This strategy focus on the quantity to nim so if
195 state.k == 1 is the only case were we have only unpaired
196 quantity to remove"""
197                 ply = Nimply(state.rows.index(row), 1)
198             elif state.last_action[1]%2==0:
199                 """The oppponent nimmed shortest a pair quantity of
200 nim"""
201                 if state.k:
202                     nimming = [el for el in range(1,state.k+1) if (
203 el%2!=0) & (el <= row)] #Nim quantity possible in the selected
204 row
205                     else:
206                         if row == 1:

```

```

194             nimming = [1]
195         else:
196             nimming = [el for el in range(1, row+1) if (
197                 el%2!=0) & (el <= row)] #Nim quantity possible in the selected
198             row
199             ply = Nimply(state.rows.index(row), nimming[0])
200         else:
201             """The opponent nimmed shortest an unpair quantity
202             of nim"""
203             if state.k:
204                 nimming = [el for el in range(1, state.k+1) if (
205                     el%2==0) & (el <= row)] #Nim quantity possible in the selected
206                     row
207             else:
208                 if row == 1:
209                     nimming = [1]
210                 else:
211                     nimming = [el for el in range(1, row+1) if (
212                         el%2==0) & (el <= row)] #No quantity possible in the selected
213                     row
214                     ply = Nimply(state.rows.index(row), nimming[0])
215             return ply
216
217     else:
218         return self.pure_random(state)
219
220     def pure_random(self, state: Nim):
221         if not state.k:
222             row = random.choice([r for r, c in enumerate(state.rows)
223             ) if c > 0])
224             num_objects = random.randint(1, state.rows[row])
225             return Nimply(row, num_objects)
226         else:
227             if self.possible_moves(state):
228                 row = random.choice([r for r, c in self.
229             possible_moves(state) if (c > 0) & (c <=state.k)])
230                 num_objects = random.randint(1, min(random.randint
231                 (1, state.k), state.rows[row]))
232                 return Nimply(row, num_objects)
233             else:
234                 None
235
236     def strategy(self, state:Nim): #DEFINE PARAMS AS ATTRIBUTE
237         """
238             strategies used to react to opponent actions, I choose to
239             wait different kind of rules
240             @state: the state of the nim game
241             @params: probability of each strategy the simple strategy
242             will be always the same. sum of probability all simple = 0.2
243             """
244             moves = self.possible_moves(state)
245             rules = self.cooked_rules(state, moves)
246
247             strategies_name = list(rules.keys())
248             strategy_choosed = np.random.choice(strategies_name, p=self
249             .params)
250             if strategy_choosed == "simple":

```

```

238     strategy_choosed = np.random.choice(list(rules[
239         strategy_choosed].keys()))
240     return rules["simple"][strategy_choosed]
241     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 likely 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 similar 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 actions

**According the region opponent action (longest or shortest rows) L144-176** Here, I want to play in a different region than 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     """
```

```

3     strategies used to react to opponent actions, I choose to
4     wait different kind of rules
5     @state: the state of the nim game
6     @params: probability of each strategy the simple strategy
7     will be always the same. sum of probability all simple = 0.2
8     """
9
10    moves = self.possible_moves(state)
11    rules = self.cooked_rules(state,moves)
12
13    strategies_name = list(rules.keys())
14    strategy_choosed = np.random.choice(strategies_name, p=self.
15    params)
16    if strategy_choosed == "simple":
17        strategy_choosed = np.random.choice(list(rules[
18        strategy_choosed].keys()))
19    return rules["simple"][strategy_choosed]
20    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]\*\*

```

1 Individual = namedtuple("Individual", ["genome", "fitness"])
2 def generate_weights(weights: list):
3     """Function that make the sum of weight == 1"""
4     p = weights
5     p = np.array(p)
6     p = p/p.sum() # normalize
7     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.

```

1 NUM_MATCHES = 50
2 NIM_SIZE = 20
3 K = 5
4
5 def fitness(genome: list):
6     evolution = []
7     for i in range(NIM_SIZE):
8         winner = []
9         for m in range(NUM_MATCHES):
10             nim = Nim(i)
11             p1 = RuleBased("classic_rules", genome)
12             p2 = RuleBased("pure_random", [0,0,0,0,0,1])
13             game = Game(p1,p2, i)
14             winner.append(game.run())
15         evolution.append(op.countOf(winner, "classic_rules")/len(
16         winner))
17         #print(f'Percentage of victory against pure random:{op.
18         countOf(winner, "classic_rules")/len(winner)} with nim size {i
19         }')
20     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.

```
1 def cross_over(g1, g2, g3):
2     cut = random.randint(0, 3)
3     cut2 = random.randint(cut,5)
4     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

```
1 def mutation(g:list):
2     point1 = random.randint(0, len(g) - 1)
3     point2 = None
4     while not point2:
5         test = random.randint(0, len(g) - 1)
6         if test != point1:
7             point2 = test
8     tmp = g[point1]
9     g[point1] = g[point2]
10    g[point2] = tmp
11    return g
```

## Tournament

We will get the maximum fitness between two Individuals

```
1 def tournament(population: list, tournament_size=2):
2     """Tournament function"""
3     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.

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

```

13         for i in range(OFFSPRING_SIZE):
14             if random.random() < 0.3:
15                 p = tournament(population)
16                 o = mutation(p.genome.copy())
17             else:
18                 p1 = tournament(population)
19                 p2 = tournament(population)
20                 p3 = tournament(population)
21                 o = cross_over(p1.genome.copy(), p2.genome.copy(),
22                                 p3.genome.copy())
23                 f = fitness(o)
24                 offspring.append(Individual(o, f))
25             population+=offspring
26             population = sorted(population, key=lambda indi: indi.
27                                 fitness, reverse=True)[:POPULATION_SIZE]
28             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

```

1 class MinMax(Player):
2     def __init__(self,id):
3         super().__init__(id)
4
5
6     def evaluate(self, state, is_maximizing):
7         if all(el == 0 for el in state.rows):
8             return -1 if is_maximizing else 1
9
10    def possible_states(self,state:Nim):
11        states = []
12        for el in self.possible_moves(state):
13            cop = copy.deepcopy(state)
14            cop.nimming(Nimply(el[0],el[1]))
15            states.append(cop)
16        return states
17
18    def minimax(self, state: Nim, is_maximizing, alpha=-1, beta=1):
19        if (score := self.evaluate(state, is_maximizing)) is not
None:
20            """if we are in final leaf, return the score"""
21            return score
22
23        scores = []

```

```

24         for moves in self.possible_moves(state):
25             cop = copy.deepcopy(state)
26             cop.nimming(Nimply(moves[0], moves[1]))
27             scores.append(
28                 score := self.minimax(cop, not is_maximizing, alpha
29             , beta)
30             )
31             if is_maximizing:
32                 alpha = max(alpha, score)
33             else:
34                 beta = min(beta, score)
35             if beta <= alpha:
36                 break
37         return (max if is_maximizing else min)(scores)
38
39     def best_move(self, state:Nim):
40         scores = {}
41         for s in self.possible_states(state):
42             scores[s] = self.minimax(s, False)
43         return sorted(scores.items(), key=lambda x: x[1], reverse=
44             True)[0][0]
45
46     def diff(self, stateA: Nim, stateB: Nim):
47         for i in range(len(stateA.rows)):
48             if stateA.rows[i] != stateB.rows[i]:
49                 return (i, stateA.rows[i]-stateB.rows[i])
50
51     def strategy(self, state:Nim):
52         # Minimax move
53         cop = copy.deepcopy(state)
54         move = self.best_move(state)
55         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 assignment. 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 puerandom, 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.

```

1  class MinMaxImproved(Player):
2      def __init__(self, id):
3          super().__init__(id)
4          self._possible_new_states = None
5
6      def diff(self, stateA: Nim, stateB: Nim):
7          for i in range(len(stateA)):
8              if stateA[i] != stateB[i]:
9                  return (i, stateA[i]-stateB[i])
10
11     def possible_new_states(self, state):
12         for pile, counters in enumerate(state):
13             for remain in range(counters):
14                 yield state[:pile] + (remain,) + state[pile + 1 :]
15
16     def evaluate(self, state, is_maximizing):
17         if all(counters == 0 for counters in state):
18             return -1 if is_maximizing else 1
19
20     @functools.cache
21     def minimax(self, state, is_maximizing, alpha=-1, beta=1):
22         """Evaluate a game state using the minimax algorithm"""
23         if (score := self.evaluate(state, is_maximizing)) is not
None:
24             return score
25
26         scores = []
27         for new_state in self.possible_new_states(state):
28             scores.append(
29                 score := self.minimax(new_state, not is_maximizing,
alpha, beta)
29             )
30         if is_maximizing:
31             alpha = max(alpha, score)
32         else:
33             beta = min(beta, score)
34         if beta <= alpha:
35             break
36         return (max if is_maximizing else min)(scores)
37
38     def best_move(self, state):
39         """Use minimax() to find the best move"""
40         evaluate = functools.partial(self.minimax, is_maximizing=
False)
41         return max(self.possible_new_states(state), key=evaluate,
default=None)
42
43     def strategy(self, state:Nim):
44         cop = state.rows
45         # self._possible_new_states = self.possible_new_states(state
46         )
47         move = self.best_move(tuple(cop))
48         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 are many possible states of the game that can be hardcoded according to 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

- Maybe using class, you should make a more generic code
- You should try to show the results against a series 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 into 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](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), and 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 chooses 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 attribute (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 **scorealignment** 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.

```
1 def score_alignment(self, alignment: list, selected_piece_char: quarto.Piece) -> list:
2     """
3         Description: calculate the score for aligned pieces
4
5         @args:
6             alignment [list]: the list of pieces aligned
7             selected_piece_char [quarto.Piece]: the piece to compare in
8                 order to get the score
9
10        @returns:
11            the list of score for each characteristics
12        """
13
14        score_char = [0,0,0,0] #HIGH, COLOURED, SOLID, SQUARE
15        features = [[piece.HIGH, piece.COLOURED, piece.SOLID, piece.
16        SQUARE] for piece in alignment] #Get the features of each
17        piece
18
19        selected_value = [selected_piece_char.HIGH,
20                          selected_piece_char.COLOURED, selected_piece_char.SOLID,
21                          selected_piece_char.SQUARE]
22        for i in range(4):
23            count_feature = set([piece[i] for piece in features]) #get
24            the feature n
25            if ((len(count_feature) == 1) and (list(count_feature)[0]
26            == selected_value[i])):
27                score_char[i] = len(alignment)
28
29        return score_char
```

```

21 def board_analysis(self, piece_selected: int) -> dict:
22     """
23     Description: Calculate the score for each possible alignment
24
25     @args
26     piece_selected [int]: index of the piece selected to be insert
27     on the board
28
29     @returns
30     The score for each possible alignment
31     """
32     board_copy = self.get_game().get_board_status()
33     piece_selected = self.get_game().get_piece_charachteristics(
34         piece_selected)
35     horizontal = {"0": [], "1": [], "2": [], "3": []}
36     vertical = {"0": [], "1": [], "2": [], "3": []}
37     diag = {"0": [], "1": []}
38     for i in range(self.get_game().BOARD_SIDE):
39         for j in range(self.get_game().BOARD_SIDE):
40             if board_copy[i, j] != -1:
41                 piece = self.get_game().get_piece_charachteristics(
42                     board_copy[i, j])
43                 horizontal[str(i)].append(piece)
44                 vertical[str(j)].append(piece)
45                 if i == j:
46                     diag["0"].append(piece)
47                 elif i + j == 3:
48                     diag["1"].append(piece)
49     scores = {"scoreH": {"0": [], "1": [], "2": [], "3": []}, "scoreV": {"0": [],
50     "1": [], "2": [], "3": []}, "scoreD": {"0": [], "1": []}}
51     for k in range(self.get_game().BOARD_SIDE):
52         scores["scoreH"][str(k)] = self.score_alignement(horizontal[
53             str(k)], piece_selected)
54         scores["scoreV"][str(k)] = self.score_alignement(vertical[
55             str(k)], piece_selected)
56         if k <= 1:
57             scores["scoreD"][str(k)] = self.score_alignement(diag[
58                 str(k)], piece_selected)
59     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 at 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
5     interesting to play on the diagonal where the possibility of
6     quarto is 3 lines
7     @returns: tuple[int, int]
8     """

```

```

8     l=[]
9     length = self.get_game().BOARD_SIDE
10    for i in range(length):
11        if self.get_game().get_board_status()[i,i] == -1:
12            l.append((i,i))
13        if self.get_game().get_board_status()[length-1-i,i] == -1:
14            l.append((i,length-1-i))
15    if len(l) > 0:
16        choose = random.randint(0,len(l)-1)
17        return l[choose]
18    else:
19        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 alignment, the agent can place the piece on the free position.

```

1 def completing_raw(self, board_copy:np.ndarray) -> tuple[int, int]:
2     """
3         TYPE OF STRATEGY: ATTACK (critical action => Priority 1)
4         DESCRIPTION: If there is an open space on the board where
5             placing a piece would create a row of four pieces that all have
6             a similarity,
7             then place a piece of that similarity in that space.
8             @returns: tuple[int, int]
9             """
10
11    # print("COMP")
12    scores = self.board_analysis(self.get_game(),
13        get_selected_piece())
14    return self.possible_quarto(board_copy, scores)
15
16
17 def possible_quarto(self, board_copy: np.ndarray, scores: dict) ->
18     tuple[int,int]:
19     for i in range(self.get_game().BOARD_SIDE):
20         for j in range(self.get_game().BOARD_SIDE):
21             if board_copy[i,j] == -1:
22                 scoreH = scores["scoreH"][str(i)]
23                 scoreV = scores["scoreV"][str(j)]
24                 if i==j:
25                     scoreD = scores["scoreD"]["0"]
26                 elif i +j == 3:
27                     scoreD = scores["scoreD"]["1"]
28                 else:
29

```

```

23         scoreD = [0]
24     if 3 in scoreH or 3 in scoreV or 3 in scoreD:
25         return [j,i]
26     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 alignment in the board (4 horizontal, 4 vertical, 2 diagonal) as seen before. Knowing all those alignment, we can know score the free positions.

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

```

1 def alignment(self, board_copy:np.ndarray, selected_piece: int) ->
2     list:
3     """
4     returns: the possible horizontal, vertical, diagonal
5     alignements
6     """
7     board_score = sorted(self.scoring_positions(board_copy,
8         selected_piece), key=lambda x: x[1], reverse = True)
9     horizontal = []
10    vertical = []
11    diagonal = []
12    if len(board_score)<2:
13        return horizontal, vertical, diagonal
14    points_occurrences = {} #dict to identify the points with many
15    alignements type (max 3).
16
17    #Looking for possible alignment
18    for p1 in board_score:
19        for p2 in board_score:
20            #if the piece are not antagonist there are not a
21            #possible alignment
22            if (p1!=p2) & (board_copy[p1[0][0],p1[0][1]] +
23                board_copy[p2[0][0],p2[0][1]] != 15):
24
25                #HorizontalCheck
26                if (p1[0][0] == p2[0][0]) & (((p2[0],p1[0]),p1[1] +
27                    p2[1]) not in horizontal):
28
29                    #adding alignement and penalizing some
30                    horizontal.append(((p1[0],p2[0]),p1[1] + p2[1])
31
32
33                #identifying points & penalizing some
34                if (str(p1[0]) not in points_occurrences):
35                    points_occurrences[str(p1[0])] = 1

```

```

28             else:
29                 points_occurrences[str(p1[0])] += 1
30
31             #VerticalCheck
32             if (p1[0][1] == p2[0][1]) & (((p2[0],p1[0]),p1[1] +
33               p2[1]) not in vertical):
34                 vertical.append(((p1[0],p2[0]),p1[1] + p2[1]))
35
36             #identifying points
37             if str(p1[0]) not in points_occurrences:
38                 points_occurrences[str(p1[0])] = 1
39             else:
40                 points_occurrences[str(p1[0])] += 1
41
42             #DiagonalCheck
43             if (p1[0][0] == p1[0][1]) & (p2[0][0] == p2[0][1]):
44               & (((p2[0],p1[0]),p1[1] + p2[1]) not in diagonal):
45                 diagonal.append(((p1[0],p2[0]),p1[1] + p2[1]))
46             #identifying points
47             if str(p1[0]) not in points_occurrences:
48                 points_occurrences[str(p1[0])] = 1
49             else:
50                 points_occurrences[str(p1[0])] += 1
51
52             elif (p1[0][0] + p1[0][1] == 3) & (p2[0][0] + p2
53 [0][1]== 3) & (((p2[0],p1[0]),p1[1] + p2[1]) not in diagonal):
54               diagonal.append(((p1[0],p2[0]),p1[1] + p2[1]))
55
56             #identifying points
57             if str(p1[0]) not in points_occurrences:
58                 points_occurrences[str(p1[0])] = 1
59             else:
60                 points_occurrences[str(p1[0])] += 1
61
62             #rewards to points with more than 1 alignment
63             for points in points_occurrences.items():
64               if points[1] >= 3:
65                 for el in horizontal:
66                   if el[0] == ast.literal_eval(points[0]):
67                     el[1] *= 10000
68
69                 for el in vertical:
70                   if el[0] == ast.literal_eval(points[0]):
71                     el[1] *= 10000
72
73                 for el in diagonal:
74                   if el[0] == ast.literal_eval(points[0]):
75                     el[1] *= 10000
76
77               elif points[1] >= 2:
78                 for el in horizontal:
79                   if el[0] == ast.literal_eval(points[0]):
80                     el[1] *= 1000
81
82                 for el in vertical:
83                   if el[0] == ast.literal_eval(points[0]):
84                     el[1] *= 1000
85
86                 for el in diagonal:
87                   if el[0] == ast.literal_eval(points[0]):
88                     el[1] *= 1000

```

```

82     #Sorting according the sum of scores
83     horizontal = sorted(horizontal, key=lambda x: x[1], reverse =
84     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,
91                     diagonal: list) -> list:
92     """
93     Here we are trying to find the best alignement, when there is
94     negative value of an alignement, we want to check if it is not
95     the only one
96     because we don't to consider as good alignement negative one
97     """
98
99     best_horizontal = ((-1,-1),-1)
100    best_vertical = ((-1,-1),-1)
101    best_diagonal = ((-1,-1),-1)
102
103    if horizontal:
104        if horizontal[-1][1] < 0:
105            for row in horizontal[:-1]:
106                if (row[0][0] != horizontal[-1][0][0]):
107                    best_horizontal = row
108
109    else:
110        best_horizontal = horizontal[0]
111
112    if vertical:
113        if vertical[-1][1] < 0:
114            for col in vertical[:-1]:
115                if (col[0][0] != vertical[-1][0][0]):
116                    best_vertical = col
117
118    else:
119        best_vertical = vertical[0]
120
121
122    if diagonal:
123        if diagonal[-1][1] < 0:
124            for diag in diagonal[:-1]:
125                if (diag[0][0] != diagonal[-1][0][0]):
126                    best_diagonal = diag
127
128    else:
129        best_diagonal = diagonal[0]
130
131    return best_horizontal, best_vertical, best_diagonal
132
133 def scoring_strategy(self, board_copy:np.ndarray, defense = False)
134     ->tuple[int, int]:
135         """
136         TYPE OF STRATEGY: ATTACK
137         DESCRIPTION: In this function we are trying to calculate
138         the square where we are more likely to do a quarto win
139         We already think the case of three pieces sharing a common
140         attribute with the completing method
141         Here we will try to find when 2 pieces are aligned
142
143         @returns: tuple [int, int]
144         """

```

```

132     selected_piece = self.get_game().get_selected_piece()
133     horizontal, vertical, diagonal = self.alignment(board_copy
134     , selected_piece, defense)
135     best_horizontal, best_vertical, best_diagonal = self.
best_alignment(horizontal, vertical, diagonal)
136     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 held 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
5             placing a piece would allow you to create a row of four pieces
6             that share a common attribute,
7             then place a piece in a space where it is least likely to
8             help your opponent create such a row.
9
10    @returns: tuple[int, int]
11    """
12    for not_selected_pieces in range(self.get_game().NB_PIECES):
13        :
14            if (not_selected_pieces != self.get_game().
15                get_selected_piece()) & (not_selected_pieces not in board_copy):
16                scores = self.board_analysis(not_selected_pieces)
17                position = self.possible_quarto(board_copy, scores)
18                if position != [-1, -1]:
19                    return position
20
21    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.

```
1 def giving_wrong_piece(self, board_copy: np.ndarray) -> int:
2     """
3         TYPE OF STRATEGY: DEFENSE (Critical action ==> Priority 1)
4         DESCRIPTION: If a possible quarto is coming, give a piece
5             that could not make a quarto
6             """
7     list_possible_quarto = []
8     bag = self.get_bag_pieces(board_copy)
9     bag_index = []
10    for piece in bag:
11        index = list(range(self.get_game().NB_PIECES)).index(
12            piece)
13        scores = self.board_analysis(piece)
14        pos = self.possible_quarto_list(board_copy, scores)
15        bag_index.append(index)
16        for i in range(self.get_game().BOARD_SIDE):
17            for j in range(self.get_game().BOARD_SIDE):
18                #self.get_game().set_selected_piece(index)
19                if [i,j] in pos:
20                    list_possible_quarto.append(index)
21    wrong_list = list(set(bag_index)-set(list_possible_quarto))
22    if (len(wrong_list) != 0)&(len(list_possible_quarto)!=0):
23        return wrong_list[random.randint(0, len(wrong_list)-1)]
24    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.

```
1 def giving_wrong_piece_alignement(self, board_copy: np.ndarray) ->
2     int:
3     """
4         TYPE OF STRATEGY: DEFENSE
5         DESCRIPTION: giving a piece that will not be able to be
6             place in the best alignement
7             """
8     score = 0
9     index = -1
```

```

8         for i in range(self.get_game().BOARD_SIDE):
9             for j in range(self.get_game().BOARD_SIDE):
10                 if board_copy[i,j] != -1:
11                     score_ij = 0
12                     pieces = {"h":[], "v":[], "d":[]}
13                     ##Check the column
14                     for a in range(self.get_game().BOARD_SIDE):
15                         if (board_copy[i,a] != -1) & ((i,a)!=(i,j))
16                             :
17                             scoring_alignement = self.
18                             share_attributes_pieces(board_copy,(i,j),(i,a))
19                             score_ij += scoring_alignement
20                             pieces["h"].append(board_copy[i,a])
21                             pieces["h"].append(board_copy[i,j])
22                             ##Check the column
23                             for a in range(self.get_game().BOARD_SIDE):
24                                 if (board_copy[a,j] != -1) & ((a,j)!=(i,j))
25                                     :
26                                     scoring_alignement = self.
27                                     share_attributes_pieces(board_copy,(i,j),(i,a))
28                                     score_ij += scoring_alignement
29                                     pieces["v"].append(board_copy[a,j])
30                                     pieces["v"].append(board_copy[i,j])
31                                     ##Check the diagonal
32                                     if i == j:
33                                         for a in range(self.get_game().BOARD_SIDE):
34                                             if(board_copy[a,a] != -1)&((a,a) != (i,
35                                             j)):
36                                             scoring_alignement = self.
37                                             share_attributes_pieces(board_copy,(i,j),(i,a))
38                                             score_ij += scoring_alignement
39                                             pieces["d"].append(board_copy[a,a])
40                                             pieces["d"].append(board_copy[i,j])
41                                             elif i + j == 3:
42                                                 for a in range(self.get_game().BOARD_SIDE):
43                                                     if(board_copy[a, 3-a] != -1) &((a, 3-a)
44                                                     !=(i,j)):
45                                                     scoring_alignement = self.
46                                                     share_attributes_pieces(board_copy,(i,j),(i,a))
47                                                     score_ij += scoring_alignement
48                                                     pieces["d"].append(board_copy[a, 3-
49                                                     a])
50                                                     pieces["d"].append(board_copy[i,j])
51                                                     if score_ij > score:
52                                                         score = score_ij
53                                                         index = self.add_not_quarto(board_copy,
54                                                         pieces)[0]
55             return index
56
57 def add_not_quarto(self, board_copy:np.ndarray, lines: dict) -> int
58     """
59     @args:
60     lines: dictionary containing the list of index for each
61     alignment type h,v,d
62     """
63     bag = self.get_bag_pieces(board_copy)

```

```

53     bag_score = {}
54
55     #initialize the score for each piece in the bag
56     for k,v in bag.items():
57         bag_score[k] = 0
58
59     #transform each index in a Piece object
60     pieces_by_align = {"h":[], "v":[], "d":[]}
61     for k,v in lines.items():
62         if v:
63             for index in v:
64                 pieces_by_align[k].append(self.get_game().get_piece_characteristics(index))
65
66
67     #evaluating which common attributes are most present in the alignment
68     attributes = {"HIGH":list((None,0)), "COLOURED":list((None,0)), "SOLID":list((None,0)), "SQUARE":list((None,0))}
69     for k,v in pieces_by_align.items():
70         #Look each pieces in the alignment
71         characteristics = {"HIGH":True, "COLOURED":True, "SOLID":True, "SQUARE":True}
72         if not pieces_by_align[k]:
73             characteristics = {}
74         else:
75             for piece in pieces_by_align[k]:
76                 for piece2 in pieces_by_align[k]:
77                     if piece != piece2:
78                         if (piece.HIGH != piece2.HIGH):
79                             characteristics.pop("HIGH", None)
80                         if (piece.COLOURED != piece2.COLOURED):
81                             characteristics.pop("COLOURED", None)
82                         if (piece.SOLID != piece2.SOLID):
83                             characteristics.pop("SOLID", None)
84                         if (piece.SQUARE != piece2.SQUARE):
85                             characteristics.pop("SQUARE", None)
86                     for a,b in characteristics.items():
87                         if (attributes[a][0] is None)&(pieces_by_align[k] != []):
88                             attributes[a][0] = getattr(pieces_by_align[k][0], a)
89                             attributes[a][1] += 1
90
91             scores = dict(sorted(attributes.items(), key=lambda x: x[1][1], reverse=True))
92
93     #scoring each piece in the bag
94     for attribute,score in scores.items():
95         if score[1] != 0:
96             for index,piece in bag.items():
97                 if (attribute == "HIGH")&(piece.HIGH == score[0]):
98                     bag_score[index] += 10 * score[1]
99                 elif (attribute == "COLOURED")&(piece.COLOURED == score[0]):
```

```

100             bag_score[index] += 10 * score[1]
101         elif (attribute == "SOLID") & (piece.SOLID ==
102             score[0]):
103             bag_score[index] += 10 * score[1]
104         elif (attribute == "SQUARE") & (piece.SQUARE ==
105             score[0]):
106             bag_score[index] += 10 * score[1]
107
108     bag_score = sorted(bag_score.items(), key=lambda x: x[1])
109     return bag_score[0]
110
111 def share_attributes_pieces(self, board_copy: np.ndarray, p1: tuple[int, int], p2: tuple[int, int]):
112     """return a score for a piece in the board compared to the
113     chosen piece"""
114     score = 0
115     piece1 = self.get_game().get_piece_charachteristics(
116         board_copy[p1[0], p1[1]])
117     piece2 = self.get_game().get_piece_charachteristics(
118         board_copy[p2[0], p2[1]])
119     if piece1.HIGH == piece2.HIGH:
120         score += 1
121     if piece1.COLOURED == piece2.COLOURED:
122         score += 1
123     if piece1.SOLID == piece2.SOLID:
124         score += 1
125     if piece1.SQUARE == piece2.SQUARE:
126         score += 1
127     if score == 0:
128         """if the score == 0, it means the piece is antagonist
129         to the chosen one, so we will try to penalize this positions"""
130         score = -1000
131     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.

```

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

```

```

15     target[keys[7]] *= 0.1
16     #(attribut, quantit )
17     pieces = {}
18     for index in bag:
19         score = 0
20         if (bag[index].HIGH) & (target == "high"):
21             score += 1 * target["high"]
22         elif (not bag[index].HIGH) & (target == "small"):
23             score += 1 * target["small"]
24         if (bag[index].COLOURED) & (target == "color"):
25             score += 1 * target["color"]
26         elif (not bag[index].COLOURED) & (target == "not_color"
):
27             score += 1 * target["not_color"]
28         if (bag[index].SOLID) & (target == "solid"):
29             score += 1 * target["solid"]
30         elif (not bag[index].SOLID) & (target == "hollow"):
31             score += 1 * target["hollow"]
32         if (bag[index].SQUARE) & (target == "square"):
33             score += 1 * target["square"]
34         elif (not bag[index].SQUARE) & (target == "round"):
35             score += 1 * target["round"]
36         pieces[index] = score
37     return sorted(pieces.items(), key=lambda x: x[1], reverse =
True)[0][0]
38
39 def count_characteristics(self,bag:dict):
40     characteristics = {"high":0, "small":0, "color": 0, "not_color":0, "solid": 0, "hollow": 0, "square":0,"round":0}
41     for piece in bag:
42         if bag[piece].HIGH:
43             characteristics["high"] += 1
44         else:
45             characteristics["small"] += 1
46         if bag[piece].COLOURED:
47             characteristics["color"] += 1
48         else:
49             characteristics["not_color"] += 1
50         if bag[piece].SOLID:
51             characteristics["solid"] += 1
52         else:
53             characteristics["hollow"] += 1
54         if bag[piece].SQUARE:
55             characteristics["square"] += 1
56         else:
57             characteristics["round"] += 1
58
59     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 parameter in a sequential order to get the best restults: We want to know the best parameter value to decide:

- Placing Strategy: Attack or Defend

```

1 def placing_strategy(self) -> tuple[int,int]:
2     """
3         Description: Placing a piece on the board
4
5         @returns
6         tuple (i,j): the position where to place the piece
7         """
8         board_copy = self.get_game().get_board_status()
9         completing = self.completing_raw(board_copy)
10        if completing != [-1,-1]:
11            return completing
12        else:
13            block = self.block_opponent(board_copy)
14            if block != [-1,-1]:
15                return block
16            elif self.params1 != 0:
17                if random.random() > self.params1:
18                    region = self.scoring_strategy(board_copy)
19                else:
20                    region = self.scoring_strategy(board_copy,
21                                         True)
22                if region != [-1,-1]:
23                    return region

```

```

23         return self.oriented_random()
24     else:
25         """
26         Dumb Rule based
27         """
28     return self.oriented_random()

```

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

```

1 def choosing_strategy(self):
2     board_copy = self.get_game().get_board_status()
3     index = self.giving_wrong_piece(board_copy)
4     if index != -1:
5         return index
6     else:
7         if self.params2 != 0:
8             if random.random() >= self.params2:
9                 index = self.most_present_in_bag(board_copy)
10            else:
11                index = self.giving_wrong_piece_alignement(
12                    board_copy)
13
14            if index != -1:
15                return index
16            else:
17                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.

```

1 NUM_MATCHES = 1000
2 def fitness(genome: list):
3     winners = []
4     for i in range(NUM_MATCHES):
5         game = main.quarto.Quarto()
6         if i % 2 == 0:
7             Player1 = ruleBased.RuleBased(game, 0, 0)
8         else:
9             Player1 = main.RandomPlayer(game)
10            if random.random() > 0.5:

```

```

11         game.set_players((Player1,ruleBased.RuleBased(game,
12             genome[0],genome[1])))
13     else:
14         game.set_players((ruleBased.RuleBased(game,genome[0],
15             genome[1]),Player1))
14     winner = game.run()
15     winners.append(winner)
16 return winners.count(1)/NUM_MATCHES

```

#### 4.3.3 Cross-OVER

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

```

1 def cross_over(g1: list, g2:list) -> list:
2     g3 = [g1[0],g2[1]]
3     g4 = [g1[1],g2[0]]
4     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

```

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

```

#### 4.3.5 Tournament

We will get the maximum fitness between two Individuals

```

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

```

#### 4.3.6 Evolution

```

1 def create_rand_pop(quantity:int) -> list:
2     population = []
3     for i in range(quantity):
4         a = random.random()
5         b = random.random()
6         genome = [a,b]
7         fitness_value = fitness(genome)
8         population.append(Individual(genome, fitness_value))
9         print(f"{i+1}/{quantity} of the population created")
10    return population
11
12 def evolution(population: list,POPULATION_SIZE: int,
13 NUM_GENERATIONS:int, OFFSPRING_SIZE:int):

```

```

13     for g in range(NUM_GENERATIONS):
14         print(g)
15         offspring = list()
16         for i in range(OFFSPRING_SIZE):
17             if random.random() < 0.3:
18                 p = tournament(population)
19                 o = mutation(p.genome.copy())
20             else:
21                 p1 = tournament(population)
22                 p2 = tournament(population)
23                 o = cross_over(p1.genome.copy(), p2.genome.copy())
24                 f = fitness(o)
25                 offspring.append(Individual(o, f))
26         population+=offspring
27         population = sorted(population, key=lambda indi: indi.
28                             fitness, reverse=True)[:POPULATION_SIZE]
29     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