Report by Mohamed Amine Hamdi s301247

All the code for the laboratories and the final project can be found here: https://github.com/moyne/computationalIntelligence_2022_s301247

Projects and Laboratories

Final Project: Quarto

Problem definition

Develop an agent for the game Quarto that can play pretty well against a totally random agent

Proposed solution

For this project I chose to work with the Genetic Programming technique, but developed in my own way, because I wanted to try it out, I started without much hope on the results but came out of it pretty surprised.

My implementation is composed of a genome that has a set of rules for the choosing and some for the placing, a rule is built with two trees, one for the if content and one for the then content, generally GP is used mainly for the if node, and in fact when using a set of predefined functions on the then content, the results may be better, but for "research" purposes, I propose the solution including both trees.

The if tree has a set of nodes that can either be operations, functions or constant values, the operations can require one or two parameters, and are done through a really easy to extend lambda dictionary, the if tree can be as long as the user requires to by the MAX_RECURSION_DEPTH constant.

The if tree is evaluated trying to see if the returned value is True or False.

The then tree is similar but it works differently because it has only operations and functions, the functions tho returns a proposed action, so maybe the function is place_at_diagonal and it returns a value between (0,0),(1,1),(2,2),(3,3) , so these functions are the leafs in our tree, the operations instead works very differently than the if tree counterpart, in fact here we need to return a chosen action, the operation so chooses the rigth node action or the leaf node action.

These actions are taken from the quartolib functions, they are really simple and extendible as well, so the user can try to feed the genetic programming with different actions etc.

The quartolib functions tho are not intelligent ones, they just return a simple value, the functions that I proposed for the choosing then trees in fact are just choose a high piece, or a solid piece etc.

The quality of the results can depend quite a lot on the quality of the functions proposed in the quartolib library, I chose really basic functions for now but as I said this piece of code is pretty extendible and really simple to modify, no functions here do some intelligent work, they do exactly what their name says, and also they don't return a possible solution for sure, in fact one of the operators I chose for now in the then trees is the possible one, this operator returns the possible operation between the two nodes.

The quality of the rule is calculated then in a 3(N) game span, in these games the rule is always chosen first by the genome, so in case I have 5 choosing rules, for each of the rules that needs to be evaluated we play 3 games against the random player and every time the choosing action is required I evaluate the rule to be evaluated as first then the rest, the rule is evaluated in the basis of the times the if tree was true and the times the then tree returned a possible action, after these 3 games I check the results and if the rule is not plausible (the if tree was never true or it was too much true(I want the rules to be correct like 33% of times, to not be overshadowing the rest of the rules, and also to have more specific rules), or the then tree has a possible action a really low amount of times) I try to mutate it until I have a good rule in that sense.

Then after evaluating all the rules that needs to be evaluated I sort them in the basis of a rule quality parameter(that favors rules that win and are true a small amount of time) or sort the rules by the amount of times the if tree was true(favoring less probable rules), and finally playing 100 games against the random player and calculating all the wins and all the times no rule was found(no if tree was true or no then tree returned a possible action), with this data I calculate the fitness of a player.

The mutations of a rule are done randomly through the tree, picking randomly new operations and functions etc, I tried also to work with mutating two rules giving one the then node of the other and vice versa.

The genetic programming then generates initially a population, then through mutations or crossovers(the crossover is done by shuffling the rules of the parents and picking a good amount of them for the child) we generate the offspring, one time every 6(N) generation, a new population arrives and takes by force at least 2(N) places in the population, forcing diversity a little bit and making the algorithm less stagnant.

The best player is pickled(thanks to the dill dependency) in a file and can be used to play against everytime.

I tried also as an approach taking singularly rules as genomes, but that approach didn't work as I wanted so I kept this approach.

The program when called can take as an argument if it needs training through the -t (--training) flag(if set it trains otherwise no), the number of generations of training -g 41 (--generations 41) (the default number is set to 50), and the filename where to save the best player at -f best_p.p (--filename best_p.p) (the default file is best_player.p).

So if I want to train a player for 120 generations and save it into pp.p I will call the program like:

'python main.py --training --generations 120 --filename pp.p'

Else if I just want to play against the random one game with the player saved into pp.p I will call the program like:

'python main.py --filename pp.p'

Or if the filename is best_player.p:

'python main.py'

Example of a rule:

if (2) eq (num_pieces_left) --> then (place_at_diagonal) possible ((place_at_corner) colless (place_at_antidiagonal))

This rule check if the number of pieces left is 2 and in case tries to place a piece in a random spot in the corner and one in the antidiagonal, if the piece on the corner is in a column with less current pieces than the one in the antidiagonal it is chosen, after this pick we confront the first one in the diagonal and this last one and we choose as a placement the possible one, in case both are possible we choose the first one.

All the rules are similar to this and are maybe more deep etc., but they are all generated completely randomly.

One cool thing is about the easiness of possible 'plugins', adding functions or operators is really easy

Results

In general after a couple of iterations the best player usually is around a range of 80% of winrate against the random player, these results are pretty promising, and could be improved significantly with parameter tuning(use of other GA to pick best params maybe) or better library functions.

The best player currently won 82% of the games against the random player, with older architectures I achieved 90, 95% as well but I didn't like the rules because they were too overshadowing of the rest of the rules, in fact is because of that that now rules are forced to have a possibility of trueness of less than 66%.

These percentages are taken from the fitness of the genome, in fact the 82% agent won 82 games but got 79.2 as fitness because during those 100 games 7 moves were random(7 in total between choosing actions and placing ones, fitness is calculated by the number of wins substracted by 0,4*number of random picks).

Here are the set of rules of the agent I got:

```
Rule: If not ((characteristic_in_less_used_row) eq
(characteristic_not_in_less_used_column)) --> Then ((not_high_piece) possible
(coloured_piece)) possible (solid_piece)

Rule: If (characteristic_in_antidiagonal) and ((most_used_characteristic) ne
(characteristic_in_most_used_row_not_complete)) --> Then (((high_piece)
similarinmostusedcolumnnotcomplete (not_high_piece)) possible (square_piece))
possible ((not_square_piece) similarinlessusedcolumnnotcomplete (coloured_piece))

Rule: If (((characteristic_not_in_less_used_column) ne (most_used_characteristic))
ne ((((not_square) ne (characteristic_not_in_less_used_row)) and ((not
(characteristic_in_antidiagonal)) ne
((characteristic_not_in_most_used_row_not_complete) and
(characteristic_not_in_most_used_row_not_complete)))) eq (((square) ne
((not_solid) or (not_solid))) ne (characteristic_in_diagonal)))) eq
(((characteristic_not_in_less_used_column) ne
(characteristic_in_less_used_column)) ne (not_high)) --> Then (((coloured_piece)
differentdiag (((not_coloured_piece) similarinmostusedrownotcomplete
(coloured_piece)) diffinlessusedcolnotcomplete (not_solid_piece)))
similarinmostusedrownotcomplete (not_square_piece)) diffinmostusedcolnotcomplete
(square_piece)

Rule: If ((less_used_characteristic) or (not (characteristic_in_diagonal))) eq
```

```
(not_coloured) --> Then ((coloured_piece) lessunique ((square_piece)
similarinlessusedrownotcomplete (((coloured_piece) lessunique (coloured_piece))
lessunique ((coloured_piece) diffinmostusedrownotcomplete (coloured_piece)))))
possible ((coloured_piece) differentdiag (coloured_piece))
Rule: If ((characteristic_not_in_most_used_column_not_complete) eq
(characteristic_in_less_used_row)) and ((False) eq (((not (solid)) ne (False)) and
(characteristic_in_diagonal))) --> Then ((((not_solid_piece)
diffinlessusedrownotcomplete (((not_square_piece) differentantidiag (solid_piece))
falses (coloured_piece))) lessunique (square_piece))
similarinlessusedcolumnnotcomplete (square_piece)) possible (not_coloured_piece)

Rule: If (characteristic_not_in_most_used_column_not_complete) ne
(characteristic_not_in_most_used_row_not_complete) --> Then (not_solid_piece)
possible ((solid_piece) similarinmostusedcolumnnotcomplete (solid_piece))

Rule: If (most_used_characteristic) and (((((solid) ne ((not_solid) and
(less_used_characteristic))) and ((characteristic_in_antidiagonal) or ((coloured)
eq (most_used_characteristic)))) ne (characteristic_in_less_used_column)) eq
(((characteristic_in_antidiagonal) ne (not_coloured)) and (True))) --> Then
(solid_piece) possible (((solid_piece) possible ((((not_solid_piece) differentdiag
(square_piece)) similarinmostusedrownotcomplete (coloured_piece))
diffinmostusedrownotcomplete (((not_high_piece) similarinlessusedrownotcomplete
(square_piece)) lessunique (square_piece)))) moreunique ((((square_piece)
diffinlessusedrownotcomplete ((square_piece) similarinlessusedcolumnnotcomplete
(solid_piece))) differentantidiag (not_coloured_piece)) falses (not_solid_piece)))

Rule: If (((((not_coloured) eq ((False) or (characteristic_in_less_used_row))) ne
(characteristic_in_most_used_column_not_complete)) eq (not
(((characteristic_in_less_used_row) eq (characteristic_not_in_less_used_column))
and (characteristic_not_in_most_used_row_not_complete)))) or
(characteristic_not_in_diagonal)) ne ((((not_high) eq (coloured)) or
(characteristic_in_most_used_column_not_complete)) ne ((False) ne
((most_used_characteristic) and ((characteristic_in_less_used_column) ne
((characteristic_not_in_most_used_row_not_complete) ne (high)))))) --> Then
((((not_square_piece) diffinlessusedrownotcomplete (not_solid_piece))
similarinlessusedrownotcomplete (not_square_piece)) differentantidiag
(high_piece)) possible (coloured_piece)

Rule: If (characteristic_not_in_less_used_row) ne
(characteristic_not_in_less_used_column) --> Then ((not_solid_piece)
diffinmostusedcolnotcomplete ((square_piece) diffinlessusedrownotcomplete
((solid_piece) trues (((coloured_piece) diffinlessusedrownotcomplete
(not_solid_piece)) differentantidiag ((coloured_piece)
diffinmostusedcolnotcomplete (not_solid_piece)))))) diffinlessusedrownotcomplete
((high_piece) diffinlessusedcolnotcomplete (not_solid_piece))

Rule: If (((characteristic_not_in_antidiagonal) ne
(((characteristic_not_in_most_used_column_not_complete) ne (not (not_coloured)))
and (characteristic_in_most_used_row_not_complete))) and ((solid) and (coloured)))
ne ((not (((not_high) and ((True) and (not_solid))) ne
(characteristic_not_in_less_used_column))) and (((((less_used_characteristic) and
(not_coloured)) eq (False)) and (((characteristic_not_in_diagonal) or (coloured))
ne ((square) eq (characteristic_in_less_used_row)))) or
(characteristic_in_most_used_row_not_complete))) --> Then (((solid_piece)
lessunique (not_coloured_piece)) similarinmostusedcolumnnotcomplete
((not_high_piece) trues ((high_piece) similarinmostusedcolumnnotcomplete
((not_square_piece) diffinlessusedrownotcomplete ((square_piece) lessunique
(not_coloured_piece)))))) possible (((not_solid_piece)
diffinmostusedcolnotcomplete (square_piece)) similarinlessusedcolumnnotcomplete
((((coloured_piece) diffinlessusedrownotcomplete ((solid_piece)
similarinmostusedcolumnnotcomplete (not_high_piece)))
similarinlessusedrownotcomplete (not_square_piece))
similarinmostusedrownotcomplete ((high_piece) diffinlessusedrownotcomplete
(not_square_piece))))

        ; Place piece rules :

Rule: If (not ((((num_pieces_chosen) or ((most_used_row) lte (9))) eq ((11) mul
((num_pieces_in_less_used_column) gt (8)))) eq ((not
((num_pieces_in_most_used_row_not_complete) gte (5))) ne
(num_pieces_in_less_used_column)))) eq (((2) sub (num_elements_in_antidiagonal))
gt (num_pieces_in_most_used_row_not_complete)) --> Then
(element_in_most_used_column_not_complete) possible
(element_in_most_used_column_not_complete)

Rule: If not (not (num_pieces_in_less_used_row)) --> Then
((element_in_most_used_column_not_complete) colmore (element_in_diagonal))
possible (((element_in_less_used_row) antidiagmore (element_in_diagonal)) possible
((element_in_less_used_row) antidiagmore (((element_in_corner) colmore
(element_in_antidiagonal)) rowmore (((element_in_less_used_column) antidiagless
(element_in_most_used_column_not_complete)) rowless (element_in_corner)))))

Rule: If (1) sub (num_elements_in_antidiagonal) --> Then
(element_in_most_used_row_not_complete) possible (element_in_antidiagonal)

Rule: If not (num_elements_in_diagonal) --> Then
((element_in_most_used_row_not_complete) possible (((((element_in_antidiagonal)
rowmore (element_inside)) antidiagless (element_in_less_used_row)) antidiagmore
(((element_in_diagonal) colmore (element_in_diagonal)) antidiagless
(element_in_corner))) diagmore ((((element_in_antidiagonal) diagless
(element_in_corner)) antidiagmore (element_in_corner)) colmore
(element_in_diagonal)))) possible (element_in_most_used_column_not_complete)

Rule: If ((12) add (((((11) add (num_pieces_in_less_used_row)) sub ((0) add
(False))) gt (16)) lte (7))) and (num_elements_in_diagonal) --> Then
((element_in_diagonal) antidiagmore (element_in_most_used_row_not_complete))
possible ((((element_in_diagonal) antidiagmore
(((element_in_most_used_row_not_complete) antidiagmore (element_in_antidiagonal))
colless (element_in_most_used_column_not_complete))) possible
((element_in_less_used_row) possible ((element_in_less_used_column) colless
(element_in_less_used_row)))) antidiagless
(element_in_most_used_column_not_complete))

Rule: If ((((num_pieces_in_most_used_row_not_complete) gte ((14) lte
(most_used_column))) mul (16)) gt (10)) gte (num_elements_in_antidiagonal) -->
Then (element_in_less_used_row) possible (element_in_less_used_column)

Rule: If ((num_elements_in_diagonal) sub (most_used_row)) eq
```

```
(num_pieces_in_most_used_column_not_complete) --> Then
(element_in_most_used_column_not_complete) diagmore ((element_in_diagonal)
diagmore ((element_in_most_used_column_not_complete) colmore
(element_in_less_used_row)))

Rule: If (not ((not (((4) lte (5)) add (num_pieces_left))) or (9))) add
(((most_used_column) sub (num_elements_in_antidiagonal)) sub
((num_elements_in_diagonal) gte ((num_pieces_in_less_used_row) sub
(((num_elements_in_diagonal) sub (less_used_column)) mul (not
(num_pieces_left)))))) --> Then (element_in_diagonal) possible
(element_in_less_used_column)

Rule: If not ((False) sub (num_elements_in_diagonal)) --> Then
((element_in_diagonal) rowmore (((element_in_less_used_row) diagless
(((element_in_corner) diagless (element_in_most_used_column_not_complete)) rowmore
(element_inside))) diagmore (element_in_corner))) possible
(element_in_most_used_column_not_complete)

with fitness: 79.2 and win% of 82.0%
```

Lab 1: Set covering

note: states visited are counted as all the elements added to the frontier, not the ones popped from it

This solution is a mix between a beam search, A* and best first algorithms.

The code alone can be found in ***set_covering_solution.py***

This is because to keep track of states to visit next we have a priority queue where optimal states are favorited because the priority function calculates the amount of elements seen and take the unary minus of it and add it to the size of the state, so if a state is optimal it will have priority function 0 and will get chosen as the first candidate, so we aren't going breadth first because if a state is horrible we may never reach the point where the priority queue pops it.

After this point I chose a metric of where to stop at each level, this is done because otherwise the problem will explode too much space wise, so to contain it I chose the ***metric*** (0,2,5,10,15,...,etc.), this metric is used so if a state gets a priority of 20 and it is on the third level of the tree we will cut it immediately because 20>METRIC[3] so 20>10, knowing this we can guarantee with a really really high confidence that continuing with that state and expanding it wouldn't generate a good solution, so instead we only expand states that are in the range of our metric, this mechanism let us save a lot of precious memory and generate with a good confidence a solution that we can say is optimal in a reasonable amount of time also for big Ns(20 mins for N=50 getting the optimal weight 65) and keeping the memory down without exploding it.

The metric seems to be working not too badly but for sure it could be better, I chose this starting testing other solutions(can be checked on the last part of this notebook or in ***other_solutions.py***) and noticing how the optimal solution was growing, so for example the optimal solution with ***N=20*** had weight ***23*** while for ***N=40*** it had weight ***54***, so I tried to approximize how much loss for each level we have usually before finding the optimal solution, obviously this metric works better in some cases than others but for my testing it can generate the optimal solution without exploding in space too much.

The states are represented as a ***tuple of tuples***, this is done because tuples are immutables like our states and also this gives us theoretically a slight edge on performance with respect to lists, I used tuples also because I wanted to keep track of duplicate states and to do so I needed a hashable data structure without the need to write a hash function for a data structure, the other go to data structure was a set but it lacked this part.

A module used all over my implementations is the bisect one, this is because my states are tuples that are always ordered, so two states that have the same sets can be skipped because just doing a simple ***if in frontier or in state_cost*** will get them because the hash will be equivalent, this module helps me inserting in order because it divides the tuple in two and searches for the position recursevely.

Another piece used was the PriorityQueue version of prof.Squillero.

(In other solutions I used as a state a tuple of numbers representing the sets of integers picked inyo this state, this gave the program a small improvement of memory, but after the use of the metric the space explosion was resolved and so memory was not an issue so I got back to the first implementation of states)

Task

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

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

and that the total numbers of elements in all L_{s_i} is minimum.

Contributions

The code was developed stricly by me but before the final solution I discussed various other solutions with some of my collegues like Diego Gasco, Enrico Magliano, Krzyszstof Kleist, Giovanni Genna, Gabriele Casetti.

Set covering problem data

Results proposed solution (other solutions have interesting data as well, especially with width limitation)

note: states visited are counted as all the elements added to the frontier, not the ones popped from it

Results with N:

- 5:
- - Found a solution in 6 steps; visited 41 states with weight 5 in 0.0002668999950401485 secs
- 10:
- - Found a solution in 5 steps; visited 1,549 states with weigth 10 in 0.006042899971362203 secs
- 20:
- - Found a solution in 6 steps; visited 38,129 states with weight 23 in 0.30784149997634813 secs

- 40:
- - Found a solution in 6 steps; visited 520,162 states with weight 54 in 23.404796299990267 secs
- 50:
- - Found a solution in 6 steps; visited 4,877,807 states states with weight 54 in 1282.1018204999855 secs
- - Path [((),), ((), (1, 2, 40, 47, 16, 22, 23, 24, 30)), ((), (1, 2, 40, 47, 16, 22, 23, 24, 30), (32, 34, 3, 36, 38, 6, 8, 41, 10, 43, 45, 48, 17, 18, 20, 26)), ((), (1, 2, 40, 47, 16, 22, 23, 24, 30), (32, 34, 3, 36, 38, 6, 8, 41, 10, 43, 45, 48, 17, 18, 20, 26), (35, 4, 37, 42, 12, 13, 46, 48, 21, 30)), ((), (0, 33, 3, 35, 6, 7, 41, 9, 11, 44, 14, 15, 48, 49, 25, 27, 29, 31), (1, 2, 40, 47, 16, 22, 23, 24, 30), (32, 34, 3, 36, 38, 6, 8, 41, 10, 43, 45, 48, 17, 18, 20, 26), (35, 4, 37, 42, 12, 13, 46, 48, 21, 30)), ((), (0, 33, 3, 35, 6, 7, 41, 9, 11, 44, 14, 15, 48, 49, 25, 27, 29, 31), (1, 2, 40, 47, 16, 22, 23, 24, 30), (2, 3, 4, 5, 38, 39, 41, 18, 19, 23, 28, 29), (32, 34, 3, 36, 38, 6, 8, 41, 10, 43, 45, 48, 17, 18, 20, 26), (35, 4, 37, 42, 12, 13, 46, 48, 21, 30))]
- - Solution ((), (0, 33, 3, 35, 6, 7, 41, 9, 11, 44, 14, 15, 48, 49, 25, 27, 29, 31), (1, 2, 40, 47, 16, 22, 23, 24, 30), (2, 3, 4, 5, 38, 39, 41, 18, 19, 23, 28, 29), (32, 34, 3, 36, 38, 6, 8, 41, 10, 43, 45, 48, 17, 18, 20, 26), (35, 4, 37, 42, 12, 13, 46, 48, 21, 30)) with weight 65

For 100 and successive iterations I didn't have the time to see if the metric helps or if it needs to be tuned in a better way.

Results other solutions

Solutions with other implementations, can see immediately the huge difference in the N=40 example where the proposed solution takes less time and find the solution in 6,85% of the states used by the other solutions.

Anyway we can see by the solutions with 50 that the width limitations can help really really a lot without causing much loss in the optimality of the solution, so we can evince that an implementation that can guarantee limitation in both width and metrics can be really effective.

note: states visited are counted as all the elements added to the frontier, not the ones popped from it

- 5:
- - problem : ((0,), (1,), (0,), (4,), (0,), (1,), (4,), (4,), (4,), (1, 3), (0, 1), (2,), (1,), (0,), (0, 2), (2, 4), (3,), (3,), (4,), (2, 4), (0,), (1,), (0, 1), (3,), (2, 3))
- - Found a solution in 6 steps; visited 41 states in 0.0004899000050500035 secs
- - Path [((),), ((), (0,)), ((), (0,), (1,)), ((), (0,), (1,), (2,)), ((), (0,), (1,), (2,), (3,)), ((), (0,), (1,), (2,), (3,), (4,))] Solution ((), (0,), (1,), (2,), (3,), (4,)) with weight 5

- 10:
- - ((0, 4), (1, 2, 3), (9, 6), (0, 1), (8, 9, 3), (8, 3), (0, 3, 4, 7, 9), (4, 5, 6), (1, 3, 5), (1, 6), (0, 9, 4, 5), (8, 1, 6), (9, 3, 5), (0, 3), (1, 3, 6), (2, 5, 7), (1, 3, 4, 9), (8, 2, 3), (3, 4, 5, 6, 8), (0, 3), (1, 3, 4, 6), (3, 6, 7), (2, 3, 4), (9, 6), (8, 2, 3, 7), (0, 1), (9, 2, 6), (6,), (8, 0, 4, 1), (1, 4, 5, 6), (0, 4, 7), (8, 1, 4), (2, 5), (9, 5), (0, 1, 3, 4, 5), (9, 3), (1, 7), (8, 2), (8, 2, 7), (8, 9, 3, 6), (4, 5, 6), (8, 1, 3, 7), (0, 5), (0, 9, 3), (0, 3), (0, 5), (8, 3), (8, 2, 3, 7), (1, 3, 6, 7), (5, 6))
- - Found a solution in 5 steps; visited 1,549 states in 0.00596939999377355 secs

- - Path [((),), ((), (0, 1)), ((), (0, 1), (4, 5, 6)), ((), (0, 1), (4, 5, 6), (8, 2, 7)), ((), (0, 1), (4, 5, 6), (8, 2, 7), (9, 3))] Solution ((), (0, 1), (4, 5, 6), (8, 2, 7), (9, 3)) with weight 10

- 20:
- - ((8, 4, 7), (0, 1, 2, 3, 6, 13, 17, 18), (0, 6, 16, 17, 19), (0, 5, 7, 8, 13, 14, 17, 18), (2, 3, 4, 6, 8, 10), (1, 3, 8, 11, 14, 19), (2, 3, 9, 11, 12, 17, 18, 19), (1, 2, 9, 7), (3, 5, 6, 7, 8, 11, 12, 14), (2, 5, 7, 8, 12, 14, 17, 19), (17, 10, 1, 7), (2, 6, 8, 10, 12, 15, 18), (4, 7, 8, 14, 17, 18), (4, 7, 11, 12, 15, 16, 18), (1, 3, 4, 5), (2, 8, 12, 13, 14, 16, 17, 19), (0, 3, 5, 8, 9, 10, 13, 14, 17), (8, 16, 5), (16, 9, 19, 6), (0, 5, 11, 16, 17), (0, 1, 3, 7, 9, 10, 11, 15), (18, 2, 15), (4, 5, 8, 13, 15, 16, 17, 19), (6, 9, 11, 12, 17), (2, 3, 7, 10, 14, 16), (17, 18, 7), (0, 1, 2, 7), (16, 10, 2, 7), (4, 6, 15, 17, 18), (3, 6, 7, 13, 15), (1, 3, 13, 14), (3, 6, 7, 10, 14, 17), (5, 7, 8, 13, 14), (0, 1, 2, 3, 5, 7, 14, 17))
- - Found a solution in 6 steps; visited 40,109 states in 0.1827651999774389 secs
- - Path [((),), ((), (0, 5, 11, 16, 17)), ((), (0, 5, 11, 16, 17), (1, 3, 13, 14)), ((), (0, 5, 11, 16, 17), (1, 3, 13, 14), (2, 6, 8, 10, 12, 15, 18)), ((), (0, 5, 11, 16, 17), (1, 3, 13, 14), (2, 6, 8, 10, 12, 15, 18), (8, 4, 7)), ((), (0, 5, 11, 16, 17), (1, 3, 13, 14), (2, 6, 8, 10, 12, 15, 18), (8, 4, 7), (16, 9, 19, 6))] Solution ((), (0, 5, 11, 16, 17), (1, 3, 13, 14), (2, 6, 8, 10, 12, 15, 18), (8, 4, 7), (16, 9, 19, 6)) with weight 23

- 40:
- - ((34, 5, 6, 37, 8, 14, 15, 17), (32, 1, 2, 35, 34, 5, 38, 12, 13, 14, 26, 28), (0, 5, 6, 38, 9, 10, 13, 16, 17, 21, 22, 24, 27), ... , (32, 35, 4, 38, 8, 9, 12, 15, 17, 24, 26, 29, 30), (32, 0, 34, 37, 9, 17, 18, 19, 22, 28, 30, 31))
- - Found a solution in 6 steps; visited 7,586,233 states in 65.18534940003883 secs
- - Solution ((), (0, 32, 34, 33, 6, 38, 9, 10, 11, 12, 16, 19, 23), (1, 3, 35, 39, 8, 7, 13, 17, 18, 28, 31), (2, 4, 5, 38, 7, 36, 37, 39, 11, 15, 25, 26), (2, 35, 3, 14, 17, 20, 24, 29), (33, 35, 37, 17, 21, 22, 23, 27, 29, 30)) with weight 54

- 50 WITH LIMITATED ALGORITHM WITH 25% AS WIDTH PARAMETER:
- - ((1, 34, 5, 6, 37, 8, 43, 14, 15, 47, 17), (0, 1, 2, 5, 10, 12, 13, 14, 17, 26, 28, 32, 34, 35, 37, 38, 41, 44, 45, 48), (2, 34, 35, 5, 6, 38, 7, 9, 13, 46, 48, 17, 16, 18, 21, 22, 24, 29), (2, 4, 5, 6, 10, 12, 13, 14, 17, 18, 22, 23, 24, 29, 36, 40, 42, 45, 49), (34, 35, 4, 38, 40, 41, 10, 43, 44, 46, 15, 14, 17, 24, 29), ... , (0, 3, 5, 6, 10, 13, 14, 16, 19, 23, 25, 27, 40, 41, 42, 44, 45, 46, 48), (0, 4, 5, 6, 7, 8, 14, 20, 21, 23, 24, 27, 28, 30, 33, 37, 43, 44, 46), (0, 33, 4, 36, 39, 8, 41, 9, 43, 10, 13, 46, 48, 49, 22, 24, 28))
- - Found a solution in 6 steps; visited 3,938,464 states in 104.93217300000833 secs
- - Solution ((), (0, 1, 2, 5, 10, 12, 13, 14, 17, 26, 28, 32, 34, 35, 37, 38, 41, 44, 45, 48), (0, 3, 36, 4, 6, 7, 40, 9, 11, 43, 16, 18, 19, 21, 25, 28, 30, 31), (2, 6, 8, 42, 43, 13, 15, 49, 19, 22, 23, 27), (32, 33, 35, 38, 39, 42, 46, 48, 20, 27, 28, 29), (34, 38, 47, 48, 19, 22, 24, 26)) with weight 70

- 50 WITH LIMITATED ALGORITHM WITH 15% AS WIDTH PARAMETER:
- - Found a solution in 7 steps; visited 1,539,982 states in 69.73105619999114 secs
- - Solution ((), (0, 33, 34, 35, 32, 7, 41, 10, 43, 46, 16, 48, 17, 49, 18, 21, 27, 29), (1, 2, 6, 39, 9, 10, 11, 42, 44, 14, 15, 47, 17, 21, 24, 26, 30), (1, 3, 38, 7, 15, 19, 20, 23, 31), (2, 36, 37, 38, 7, 39, 5, 43, 15, 25, 26), (4, 36, 39, 40, 12, 45, 15, 49, 26, 29, 30), (32, 34, 4, 8, 10, 13, 15, 16, 18, 22, 23, 28)) with weight 78

Here we can see how cutting the width doesn't destroy our solution, with 25% width we still can get 70, 70/65 so the bloat is not huge, also with 15% we don't get much bloat, with 50 we can't run the algorithm without width limitation because of how memory effective it is, while with the proposed solution we get pretty much costant in the space world

---

Let's try with N=100 and a really low width value, 5% (otherwise the problem would be too big)

- 100 WITH THE STANDARD WITH 5% AS WIDTH PARAMETER:
-
    - Found a solution in 8 steps; visited 2,701,485 states in 573.1298935000086 secs
-
    - Solution ((), (0, 2, 28, 34, 38, 45, 48, 49, 53, 55, 61, 62, 68, 69, 73, 74, 77, 89, 94, 95, 96, 97, 98), (3, 4, 8, 13, 19, 20, 22, 24, 25, 28, 29, 30, 31, 34, 35, 39, 42, 44, 48, 51, 52, 58, 60, 65, 71, 79, 82, 84, 85, 89, 94, 95, 98), (3, 6, 10, 16, 17, 21, 23, 27, 33, 35, 41, 43, 48, 49, 50, 51, 54, 58, 59, 68, 72, 75, 79, 82, 84, 85, 86, 92, 97), (5, 8, 10, 12, 15, 20, 24, 26, 29, 33, 34, 35, 37, 45, 46, 47, 48, 58, 68, 70, 73, 79, 80, 81, 82, 84, 85, 87, 89, 90, 93, 98), (7, 8, 9, 10, 17, 19, 20, 27, 29, 30, 31, 33, 34, 35, 36, 40, 42, 49, 52, 53, 54, 56, 57, 59, 60, 62, 64, 66, 69, 70, 78, 80, 81, 88, 90, 91, 92, 95, 96, 98, 99), (8, 11, 17, 18, 27, 28, 31, 33, 34, 40, 46, 50, 51, 54, 58, 63, 65, 68, 71, 72, 74, 82, 83, 91, 95, 96), (32, 1, 67, 68, 70, 8, 76, 14, 80, 49, 48, 19, 20, 54, 87, 59, 92)) with weight 201

The result has weight 201, it is not bad comparing other greedy solutions and the fact that it took only 573 secs and with 5% as width

Lab 2 : Set covering (Genetic Algorithm)

Task

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

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

and that the total numbers of elements in all L_{s_i} is minimum.

Contributions

The code was developed stricly by me but before the final solution I discussed various other solutions with some of my collegues like Diego Gasco, Enrico Magliano, Krzyszstof Kleist, Giovanni Genna, Gabriele Cassetta.

Proposed solution

This solution starts from a population full of empty genomes, so a population that doesn't have any sets taken, the fitness provided return both a fitness score and the # of covered elements.

For the parent selection a wheel roulette is used that is proportionate to the fitness of the individual, I tried to use also a binomial distribution based on ranking of the fitness but in that case if indivuals were really close there wasn't still much diversity, so every individual that has higher fitness has more chances to be picked.

While for the survival selection a different approach is used, in the other solution an approach based on determinism only was used but here if we have a population with a solution in it maybe or a population without *any individual better than the current best solution we favor the fitness score, while otherwise we favor only the # of covered elements.*

So this algorithm find quickly a good solution, then it goes up and down around a solution getting better every once in a while, this algorithm is way faster than the other solution provided and usually gets better solutions.

A variety of crossover functions is provided for recombination but by default the uniform one weigthed on the fitness of parents(the best one has for each gene a 2/3% of getting picked), this is because I think it works better than a split one, then for the mutation we either do a swap in the mutation function or add a new set to the genome in the mutationsol function.

We also keep track of genomes visited, but after a call to the search function we clear it so we don't use too much memory if the # of generations is tuned in a fine way.

Results

N	Weigth	Population	Offspring
5	5	50	200
5	5	50	200
10	10	50	200
10	10	50	200
20	24	50	200
20	24	50	200
50	69	50	200
50	69	50	200
100	181	50	200
100	180	50	200
500	1283	50	200
500	1388	50	200
1000	3318	50	200
1000	3222	50	200

Other solution

This solution uses a uniform crossover by default, and a wheel roulette for the selection of the parents, the algorithm generates for NUM_GENERATIONS a set of OFFSPRING_SIZE offspring, to do so 2 options are available, for 30% of the time a mutation of a parent taken from the population is made, otherwise two parents are selected and crossed over, after this step if we don't have a valid solution but the fitness is still pretty good we try to add to that genome a set until we either reach a bad indivual or we have a valid solution.

Finally we check if the new individual is new or if it's a duplicate, in the last case we try to mutate it, finally if at the end of this process we have a valid solution we add it to the offspring set.

The fitness function proposed is setupped so an optimal solution will have 2 x PROBLEM_SIZE as the fitness, and it tries to give a lot of weigth to repetitions into a genome, so we calculate it as 2(# of covered elements) - total # of repetitions.*

*Lastly in the code we can find various heuristics to try to cut the searching space, in fact we calculate mutations starting from the ceiling of (-(-2 x PROBLEM_SIZE + f)) * PROBLEM_LEN / (10 x MAXREPETITIONS), this is a heuristic designed to remove a lot of sets in the beginning because we have a lot of bad solutions at the beginning because they have too many sets taken, this can be seen as a form of exploration, and then at the end when we try to apply exploitation this heuristic should have a ceiling of 1, so we can do swapping as our mutation.*

This algorithm works pretty well but a limitation is on the speed because requiring 100 new unique valide solutions at each generations is quite expensive, and also trying to turn an invalid solution into a valid one can be expensive as well.

Set covering problem data

Results with N:

N	Weigth	Population	Offspring
50	66	10	100
100	162	10	100
500	1388	10	100
1000	3504	10	100

Lab 3 : Nim

P1 AND P2

Task

Develop a hard coded strategy and an evolved agent for the nim game

---

Contributions

The code was developed stricly by me but before the final solution I discussed various other solutions with some of my collegues like Diego Gasco, Enrico Magliano, Krzyszstof Kleist, Giovanni Genna, Gabriele Cassetta.

Proposed solution

For this solution I started by developing a set of hard coded rules that aims at being pretty balanced, not too overpowered and not too weak.

After this set of rules I tried to build an evolved agent that tries to optimize the order of these rules, I wanted to build an agent that can optimize also the parameters of there rules but I couldn't build in time a set of rules that can be really parametrized, mainly because some of them are appropriate only in a specific moment, but I plan to do it for the next delivery hopefully, so my goal is to develop an agent that can understand both the ordering of rules and the parameters that that rule should use.

For the evolutions I used a population of 1 and an offspring of 5, only mutation is used and survival is based purely on fitness.

Fitness is defined as the # of won games out of 10 against a pure random bot.

Because of time restrictions caused by deliveries of other projects I couldn't develop a great set of rules where each rule is not taken too many times with respect to the other ones and also I couldn't develop a fully fledged evolutionary algorithm.

Results

N	Won games
0 evolutions	4/10
5 evolutions	9/10
10 evolutions	9/10

P3 AND P4

Task

Develop a minmax version and a reinforcement learning version of the algorithm for the Nim game

---

Contributions

The code was developed stricly by me but before the final solution I discussed various other solutions with some of my collegues like Diego Gasco, Enrico Magliano, Krzyszstof Kleist, Giovanni Genna, Gabriele Cassetta.

Proposed solution

Because of the same time restrictions of the other two points I couldn't develop a complete version of the code I wanted to build, for example I developed a minmax agent that calculates everything at the first move and keeps all in memory without using any pruning or smarter choice, in the next weeks hopefully I will be able to change this behaviour, rigth now it is working in the intended way but for nim with a huge number of possibilities the computational capability is huge.

For the reinforcement learning instead I developed the provided code before the lectures about it with just the idea of a reward, so also here I want to do some changes, like giving different rewards per move in the same game, for now I developed an agent that plays a game and at the end of it through a backpropagation it gives rewards to moves used, each state has a fixed # of moves, this was to not require much memory for our *program, in fact every time a significant # of moves are bad according to our scores we refresh them searching for **new** moves, my idea was to give these new moves more probability to be picked and not be useless, I tried to do it but the result was not the expected one so I left that code commented.*

So in the next weeks I hope to be able to deliver a more complete version of both algorithms.

Results

Against	Won games
Gabriele	970/1000
Pure random	600/1000
Optimal agent	0/1000

Peer Review to Collegues

Lab 1: Set Covering

Peer review to https://github.com/SalvatoreAdalberto/computational-intelligence-22-23

GENERAL:

> This solution takes inspiration from the proposed template and personally I think that's a good starting point.

MAJORS:

> The provided solution doesn't generate the optimal solution because the proposed heuristics doesn't considerate the amount of bloat in each state, but other than that on a positive note this algorithm can find a solution really fast even for N=1000 and that's more valuable in my opinion rather than a standard A* algorithm that takes 2 days to compute the optimal one.

MINORS:

> I think the code could have been commented more and have a more expressive README file, and also the structure used to manage states could have been more efficient in my opinion.

FINAL:

> So to wrap this up I really appreciate the developed solution, especially for its computational speed, good job and keep it up!

Lab 2: Set Covering (Genetic Algorithms)

Peer review to https://github.com/karlwen95/CI_22

Major considerations

I really liked that you chose a wheel roulette for this algorithm but I think it would have been nice to give higher probabilities to the numbers of the wheel under POPULATION_SIZE, so having the best individuals would have been more probable, I also like really a lot your fitness choice but on the solution function you don't keep track of the current best solution found so at the end you may have a really good population where no individual is a real solution, I think an approach similar to the comma one where you keep track of the current best solution may have been nice here, I have a doubt about the crossover where the two parents have 2 cutting points, but I think this may generate duplicates in the genome, this happens also on the mutation where there is a swap but this still could cause a duplicate gene.

A missing thing for me is the removal of duplicates in the population and offspring sets, one way to do that could have been to try to mutate the duplicates one until they were new I think.

Minor considerations

I really like the details and comments into the solution and I find it great that you developed both solutions, comma and plus one, so the user may try to see which one fits its need better.

Good work man I really liked your solution because it was fresh and new and you tried a lot of new stuff like the wheel roulette, keep it up man!

Peer review to https://github.com/ben9809/computational_intelligence_s292537

Major considerations

I liked the choice of starting from the onemax solution, but on some added code a little bit more comment either on the code or on the readme would have been appreciated, I'm not sure also about the fact that duplicates are not removed, in this case after a while majority of the population can turn into the same genome and with crossovers there is no evolution anymore so maybe using a self adapting mutation rate would have solved that mayba, I don't know also about the usage of numpy since I'm still new to Python and the usage of lists for the genomes.

Another consideration for me is how the parent_selection is developed, it's called tournament but I'm not sure it's a tournament and also a lot of bad parents can be selected since the tournament size is 2 by default and it's never updated during the calls to that function.

Minor considerations

A minor consideration for me is about the fact that len is called a lot of times when the length of the genome is known since the beginning.

Good work, keep it up man!

Lab 3: Nim

Peer Review to https://github.com/SalvatoreAdalberto/computational-intelligence-22-23 (done on 14/02/2023 😦)

3.1

I really like the thought behind the hard coded strategy diving the rules by time of activation.

I like the aggressive strategy approach but maybe it could have been made even more complex with some other rules.

3.2

I like the main implementation, the two parameters to evolve make sense, maybe you could have used more parameters.

3.3

I think the main work is done pretty well,the only thing that could have been improved I think can maybe the pruning, maybe cutting some branches of the tree could have made the algorithm more efficient.

I like the overall work and the fact that it works

3.4

I think that the generation model could have been improved and used a different approach, generating only a set of states per time, and deleting old ones maybe as well.

I really like the implementation of the reward system tho

FINAL:

> I enjoy most of the approaches taken, good job and keep it up!

Peer Reviews received by Collegues

Lab 1 : Set Covering

Peer Review by @Xiuss

- The README.md file is really detailed. The code is really well commented where needed(like in the choice of the metric).
- The metric, which is "random-reasonably" chosen by the author fits well in the proposed solution representing a good compromise between being easy to compute and efficient.
- The representation of the states using tuples of tuples is well-motivated and, in my opinion, one of the most efficient one.
- The computation of the solution is relatively fast, especially for N>20 and if compared with other solutions (included mine r.i.p.)).
- The other solutions (not proposed as "the" solution but reported in the repository) are really interesting because they provide algorithms that, through a necessary compromise on the optimality of the solution, succeed in computing a solution with N=100 in a reasonable amount of time (~500 s).

Minors: If I must find something that could be better, despite it doesn't add weight, the empty sub-list should not be included in the solution. {Solution (***()***, (0, 33, 3, 35, 6, 7, 41, 9, 11, 44, 14, 15, 48, 49, 25, 27, 29, 31),[...]) with weight 65}

Really nice job.

Peer Review by @shadow036

0. General considerations

First of all I must say that the readme is very well written and it explains very clearly the steps and reasoning behind the chosen algorithms. Also the fact that you tried other solutions as well in addition to the main one is quite admirable.

1. Proposed solution

This solution takes inspiration from the proposed template, although with custom modiffications. I really like the way you expressed in the most concise way operations which otherwise would have needed multiple lines. In addition the comments throughout the code really helps understanding each passage of the program. There are two things whose correctness and efficiency are not maximized in my opinion, even though these kinds of reasonings are quite subjective: a. I don't know if using tuples is really convenient since in this way you have to convert from tuple to list and back every now and then. b.The second point is about the adequateness of an A* algorithm for this kind of problems. As explained in the theory we must make sure that the heuristic part of the A* function never overestimate the actual cost and in this case I don't think it can be known a priori which is a good heuristic function(also in this case the heuristic cost is always equal to 1 which doesn't really discriminates different solutions). Maybe a BFS or even a DFS could be able in average more adequate for this kinds of problems.

2. Other solutions

Same good remarks and opinions about changes from previous section since it's very similar with in addition the advantage of helping to find the solutions for high (>= 100) Ns thanks to one added stopping criterion. The chosen criterion of choosing only a part of the children of a certain state could be good event though I would have chosen something else like a limit to the depth (in this case the number of subtuples) of all solutions, before passing to the next one (but again this is a personal preference).

3. Final remak

In the end I think this is a very good job with very clear explanations and the only advices I gave really depend on the way each person sees the problem.

Lab 2: Set Covering (Genetic Algorithm)

No feedback received 😦

Lab 3: Nim

Peer Review by @KrzysztofKleist

First of all it's good you provided the readme.md and the code is in jupyter notebook files making it easier to read an understand. **Task 3.1** For the hard-coded agent your idea seems to work fine, but you unfortunaly didn't provide the information about how many games did hard-coded agent win. **Task 3.2** Regarding the evolved agent the results are really good. Your evolved agent after the first generation already reaches 90% of succes rate. You could've tried to increase the size of the population and the offspring to introduce bigger variety. **Task 3.3** In minmax example I can see that your code is not finished. **Test 3.4** Your RL agent is the best so far. It reaches way better results than mine, I wasn;t even close to yours, good job!

Code of Final Project and labs

Final Project: Quarto

> ***main.py*** This file plays a game against a random player with the best player received

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

import logging
import argparse
import quarto
from agents.genetic import GeneticProg as gp
from agents.genome import Genome as myplayer
from agents.genome import RandomPlayer
import dill as pickle
from os.path import isfile


def main(training,best_player_file,generations):
    mp=None
    if training:
        #generate a genetic programming agent
        mp=gp(None,best_player_file,generations)
        print(f'\n\nMy player is {mp.player}\n\nwith fitness: {mp.player.fitness}
and win% of {mp.player.fitness+0.4*mp.player.random_pick}%\n')
    else:
        if isfile(best_player_file):
            #unpickle agent
            mp=pickle.load(open(best_player_file,'rb'))
            print(f'\n\nMy player is {mp}\n\nwith fitness: {mp.fitness} and win%
of {mp.fitness+0.4*mp.random_pick}%\n')
        else:
            print(f'WARNING: Filename provided doesn\'t exists! If you don\'t have
a trained player set the --training flag')
    if mp is not None:
        #play a game
        print(f'Game against random player starting, my player is player 1 ...')
        game = quarto.Quarto()
        game.set_players((RandomPlayer(game), mp))
        mp.set_quarto(game)
        winner = game.run()
        print(f"main: Winner: player {winner}")
```

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-v', '--verbose', action='count',
                        default=0, help='increase log verbosity')
    parser.add_argument('-d',
                        '--debug',
                        action='store_const',
                        dest='verbose',
                        const=2,
                        help='log debug messages (same as -vv)')
    parser.add_argument('-t','--training', action='store_true',dest='training',
                        help='Training phase done if set to True, otherwise read
best player from best_player.p')

    parser.add_argument('-f','--
filename',dest='filename',type=str,default='best_player.p',
                        help='Provide the filename of the best pickled player
already trained or the filename where the best player obtained from training will
be pickled, it defaults to best_player.p')

    parser.add_argument('-g','--
generations',dest='generations',type=int,default=50,
                        help='If training this represents the number of
generations, it defaults to 50')

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

    main(args.training,args.filename,args.generations)
```

> ***genetic.py*** This file contains the training process of a population, and save the best agent through the dill library

```
import logging
import random
import quarto
import numpy as np
from scipy.stats import binom
import dill as pickle
from .genome import Genome,generate_rules,MINRULES,MAXRULES

class GeneticProgramming:
    def __init__(self) -> None:
        #sizes
        self.__POPULATION_SIZE__=5
        self.__OFFSPRING_SIZE__=30
        print(f'Generating initial population ...')
        #generate initial population
        self.population=[Genome(None) for _ in range(self.__POPULATION_SIZE__)]
        #sort the population by fitness
        self.population=sorted(self.population,key=lambda a:
a.fitness,reverse=True)[:self.__POPULATION_SIZE__]
        #weigths roulette, not used actually since the weigths are calculated
differently and not though a binomial function
        self.WEIGHTS_ROULETTE=
[binom.pmf(k=_,n=self.__POPULATION_SIZE__-1,p=1/self.__POPULATION_SIZE__) for _ in
range(self.__POPULATION_SIZE__)]

    def select_parent(self,k=2,weigths=None):
        # Using a wheel roulette TO SELECT K PARENTS, THIS IS WITHOUT REPLACEMENT,
SO THE PARENT CAN'T BE TAKEN MORE THAN ONCE
        # IN A SINGLE CALL
        return [self.population[ind] for ind in
np.random.choice(range(self.__POPULATION_SIZE__),k,p=self.WEIGHTS_ROULETTE if
weigths is None else weigths,replace=False )]


    def cross_oversplit(self,genome1: Genome,genome2: Genome):
        """One point split crossover"""
        #create list of rules
        c_rules=genome1.choose_piece_rules+genome2.choose_piece_rules
        p_rules=genome1.place_piece_rules+genome2.place_piece_rules
        #shuffle lists
        random.shuffle(c_rules)
        random.shuffle(p_rules)
        #take from list a good amount of rules
        return
c_rules[:random.randint(MINRULES,MAXRULES)],p_rules[:random.randint(MINRULES,MAXRU
LES)]

    def cross_oversplit_rules(self,genome: Genome,new_c_rules,new_p_rules):
        """One point split crossover"""
        # FUNCTION NOT USED ANYMORE XXXXXXXX
        #create list of rules
        c_rules=genome.choose_piece_rules+new_c_rules
        random.shuffle(c_rules)
        #shuffle lists
        p_rules=genome.place_piece_rules+new_p_rules
        random.shuffle(p_rules)
        #take from list a good amount of rules
        return
c_rules[:random.randint(MINRULES,MAXRULES)],p_rules[:random.randint(MINRULES,MAXRU
LES)]


    def evolve(self,iterations):
        #evolving algorithm
        offspring=[]
        print(f'Population at the beginning is {self.population_stats()}')
        for i in range(iterations):
            #get minfit and calculate weigths for parent selection probabilities
            minfit,maxfit=min([gen.fitness for gen in
self.population]),max([gen.fitness for gen in self.population])
            weigths=[-minfit+self.population[_].fitness+1 for _ in
range(self.__POPULATION_SIZE__)]
            weigths=[_/sum(weigths) for _ in weigths]
            for o in range(self.__OFFSPRING_SIZE__ if i%6 else
int(self.__OFFSPRING_SIZE__*0.9)):
                #always mutate
                cross=True
                if random.random()<1/3:
                    #mutate tree
                    cross=False
                    parent=self.select_parent(k=1,weigths=weigths)[0]
                    #generate new genome that is the same as the parent

off=Genome(parent.quarto,parent.choose_piece_rules,parent.place_piece_rules)
                    off.mutate()
                else:
                    #crossover tree
                    parents= self.select_parent(k=2,weigths=weigths)
                    #get crossover rules

choose_rules,place_rules=self.cross_oversplit(parents[0],parents[1])
                    #generate new genome
                    off=Genome(parents[0].quarto,choose_rules,place_rules)
                #calculate fitness of new genome and add it to offspring
                off.evaluate_fitness()
                offspring.append(off)
            #get new population
            self.population=sorted(self.population+offspring,key=lambda a:
a.fitness,reverse=True)[:self.__POPULATION_SIZE__]
            if not i%6:
                #once every 6 gens get new visitors to the population
                print(f'Population after {i+1} gens before visitors is
{self.population_stats()}')
                print(f'Visitor from out of the town are arriving')
                #generate new genomes totally from scratch
                off_visitors=[Genome(None) for _ in
range(int(0.1*self.__OFFSPRING_SIZE__))]
                print(f'Now they have meet the population they generated two kids
with fitness of {[o.fitness for o in off_visitors]}')
                #force at least 2 of these visitors to the population

self.population=sorted(self.population[:self.__POPULATION_SIZE__-2]+off_visitors,k
ey=lambda a: a.fitness,reverse=True)[:self.__POPULATION_SIZE__]
            offspring=[]
            print(f'Population after {i+1} gens is {self.population_stats()}')

    def population_stats(self):
        #print purposes
        return [f'fit {h.fitness} rnd {h.random_pick} wins
{h.fitness+0.4*h.random_pick}' for h in self.population]


    def get_best_player(self):
        #get best genome
        return self.population[0]





class GeneticProg(quarto.Player):
    """Genetic Programming player"""

    def __init__(self, quarto: quarto.Quarto,best_player_file,generations) ->
None:
        super().__init__(quarto)
        self.quarto=quarto
        print('Training phase ...')
        #get genetic programming algorithm running and evolve it for x gens
        population=GeneticProgramming()
        population.evolve(generations)
        print(f'Population after {generations} gens is
{population.population_stats()}')
        #get best player
        self.player=population.get_best_player()
        print(f'Player is\n{self.player}')
        try:
            with open(best_player_file,'wb') as file:
                #save best player into file through dill(pickle)
                pickle.dump(self.player, file,protocol=0)
        except OSError as error:
            print(f'Error while pickle saving best player {error}')

    def set_quarto(self,quarto):
        #set quarto to myself and best player
        self.quarto=quarto
        self.player.set_quarto(self.quarto)

    def choose_piece(self) -> int:
        #play with best player
        return self.player.choose_piece()

    def place_piece(self) -> tuple:
        #play with best player
        return self.player.place_piece()
```

> ***genome.py*** This file contains the implementation of a single genome

```
import random
import copy
import quarto
```

```
from .rule import Rule
import agents.quartolib as quartolib
import numpy as np
NUMROWS=4
NUMCOLUMNS=4
NEWLINE="\n"
MINRULES=5
MAXRULES=10

#random choosing function used in case of no good rule found or by random player
def random_choose(quarto) -> int:
    return random.randint(0, 15)
#random placing function used in case of no good rule found or by random player
def random_place(quarto) -> tuple:
    return random.randint(0, 3), random.randint(0, 3)


class RandomPlayer(quarto.Player):
    """Random player"""

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)

    def choose_piece(self) -> int:
        return random.randint(0, 15)

    def place_piece(self) -> tuple:
        return random.randint(0, 3), random.randint(0, 3)

class Genome(quarto.Player):
    def __init__(self,quarto:
quarto.Quarto,choose_piece_rules=None,place_piece_rules=None) -> None:
        super().__init__(quarto)
        self.quarto=quarto
        #set of rules used
        self.choose_piece_rules=copy.deepcopy(choose_piece_rules) if
choose_piece_rules is not None else [Rule(True,None) for _ in
range(random.randint(MINRULES,MAXRULES))]
        self.place_piece_rules=copy.deepcopy(place_piece_rules) if
place_piece_rules is not None else [Rule(False,None) for _ in
range(random.randint(MINRULES,MAXRULES))]
        #rules used during evaluation of the rules
        self.evaluating_choose_piece_rules=self.choose_piece_rules
        self.evaluating_place_piece_rules=self.place_piece_rules
        self.fitness=0
        #am I during the evaluation of some rules?
        self.evaluating=False
        #am I during my own evaluation?
        self.evaluating_genome=False
        #number of times a random rule was used
        self.random_pick=0
        #print(f'\tChoose piece rules :\n {NEWLINE.join([str(rule) for rule in
self.choose_piece_rules])} \n\t;Place piece rules:\n {NEWLINE.join([str(rule) for
rule in self.place_piece_rules])}')
```

```
        if choose_piece_rules is None:
            #I am a completely new genome and I need to be evaluated immediately
            self.evaluate_fitness()
            print(f'Generated genome with fit {self.fitness} rnd 
{self.random_pick}, wins {self.fitness+0.4*self.random_pick}')

    def set_quarto(self,quarto):
        #set quarto for myself + all my rules
        self.quarto=quarto
        for rule in self.choose_piece_rules+self.place_piece_rules:
            rule.set_quarto(self.quarto)

    def choose_piece(self):
        #get possible actions
        board=self.quarto.get_board_status()
        placed_pieces=quartolib.get_placed_pieces(board)
        possible_pieces=[_ for _ in range(NUMROWS*NUMCOLUMNS) if _ not in 
placed_pieces]
        rules_to_use=self.evaluating_choose_piece_rules if self.evaluating else 
self.choose_piece_rules
        for rule in rules_to_use:
            #evaluate the if tree
            val=rule.evaluate()
            if val:
                #if true check the then tree
                act=rule.action()
                if act in possible_pieces:
                    #if the if tree is true and the then tree is possible update 
the stats of the rule and use it
                    if self.evaluating:
                        rule.evaluated(True,True)
                    return act
                else:
                    #if the if tree is true but the then tree is not possible 
update the stats and check next rule
                    if self.evaluating:
                        rule.evaluated(True,False)
            else:
                #if the if tree is not possible update the stats of the rule
                if self.evaluating:
                    rule.evaluated(False,False)
        #no good rule was found, update the stats of the random picks
        if self.evaluating_genome:
            self.random_pick+=1
        return random_choose(self.quarto)

    def place_piece(self):
        #get possible actions
        board=self.quarto.get_board_status()
        possible_placements=[(a[1],a[0]) for a in np.argwhere(board==-1).tolist()]
        rules_to_use=self.evaluating_place_piece_rules if self.evaluating else 
self.place_piece_rules
        for rule in rules_to_use:
            #evaluate the if tree
            val=rule.evaluate()
            if val:
                #if true check the then tree
                act=rule.action()
                if act in possible_placements:
                    #if the if tree is true and the then tree is possible update 
the stats of the rule and use it
                    if self.evaluating:
                        rule.evaluated(True,True)
                    return act
                else:
                    #if the if tree is true but the then tree is not possible 
update the stats and check next rule
                    if self.evaluating:
                        rule.evaluated(True,False)
            else:
                #if the if tree is not possible update the stats of the rule
                if self.evaluating:
                    rule.evaluated(False,False)
        #no good rule was found, update the stats of the random picks
        if self.evaluating_genome:
            self.random_pick+=1
        return random_place(self.quarto)

    def mutate(self):
        #get a random number to choose which combination of mutation to apply
        num=random.randint(0,6)
        if num%2==0:
            #mutate a random choose piece rule

self.choose_piece_rules[random.randint(0,len(self.choose_piece_rules)-1)].mutate()
        if num==1 or num==2 or num==5 or num==6:
            #mutate a random place piece rule

self.place_piece_rules[random.randint(0,len(self.place_piece_rules)-1)].mutate()
        if num>2:
            #do crossover between rules
            self.crossover_rules()

    def crossover_rules(self):
        #pick random rules

chooseind1,chooseind2=random.randint(0,len(self.choose_piece_rules)-1),random.rand
int(0,len(self.choose_piece_rules)-1)

placeind1,placeind2=random.randint(0,len(self.place_piece_rules)-1),random.randint
(0,len(self.place_piece_rules)-1)
        #swap trees between rules

choose_then_node,place_then_node=self.choose_piece_rules[chooseind1].then_node,sel
f.place_piece_rules[placeind1].then_node

self.choose_piece_rules[chooseind1].then_node=self.choose_piece_rules[chooseind2].
then_node
```

```
        self.choose_piece_rules[chooseind1].then_node=choose_then_node

self.place_piece_rules[placeind1].then_node=self.place_piece_rules[placeind2].then
_node
        self.place_piece_rules[placeind2].then_node=place_then_node
        #reset evaluation stats of rules, they need to be reevealuted
        self.choose_piece_rules[chooseind1].reset_evaluation_stats()
        self.choose_piece_rules[chooseind2].reset_evaluation_stats()
        self.place_piece_rules[placeind1].reset_evaluation_stats()
        self.place_piece_rules[placeind2].reset_evaluation_stats()


    def evaluate_fitness(self):
        #evaluate choose piece rules
        self.evaluating=True
        for i in range(len(self.choose_piece_rules)):
            #put rule as first
            self.evaluating_choose_piece_rules= [self.choose_piece_rules[i]] +
self.choose_piece_rules[:i] + self.choose_piece_rules[i+1:]
            make_sense=False
            # run only if the rule needs to be evaluated, so if the rule is
mutated, otherwise use old data
            while not make_sense and
self.choose_piece_rules[i].needs_evaluation():
                #run 3 games with that rule evaluated as the first one
                for _ in range(3):
                    self.choose_piece_rules[i].reset_game_stats()
                    game = quarto.Quarto()
                    playerindex=random.randint(0,1)
                    game.set_players((RandomPlayer(game), self) if playerindex==1
else (self, RandomPlayer(game)))
                    self.set_quarto(game)
                    winner = game.run()
                    #update game stats

self.choose_piece_rules[i].evaluate_game_rule(winner==playerindex)
                #does the rule make sense?
                make_sense=self.choose_piece_rules[i].rule_make_sense and
self.choose_piece_rules[i].action_make_sense
                #if not mutate the rule and rerun the loop
                if not make_sense:

self.choose_piece_rules[i].mutate(self.choose_piece_rules[i].rule_make_sense,self.
choose_piece_rules[i].action_make_sense)
        #reset order of rules
        self.evaluating_choose_piece_rules=self.choose_piece_rules

        for i in range(len(self.place_piece_rules)):
            #put rule as first
            self.evaluating_place_piece_rules= [self.place_piece_rules[i]] +
self.place_piece_rules[:i] + self.place_piece_rules[i+1:]
            make_sense=False
            # run only if the rule needs to be evaluated, so if the rule is
mutated, otherwise use old data
```

```
            while not make_sense and self.place_piece_rules[i].needs_evaluation():
                #run 3 games with that rule evaluated as the first one
                for _ in range(3):
                    self.place_piece_rules[i].reset_game_stats()
                    game = quarto.Quarto()
                    playerindex=random.randint(0,1)
                    game.set_players((RandomPlayer(game), self) if playerindex==1
else (self, RandomPlayer(game)))
                    self.set_quarto(game)
                    winner = game.run()
                    #update game stats

self.place_piece_rules[i].evaluate_game_rule(winner==playerindex)
                #does the rule make sense?
                make_sense=self.place_piece_rules[i].rule_make_sense and
self.place_piece_rules[i].action_make_sense
                #if not mutate the rule and rerun the loop
                if not make_sense:

self.place_piece_rules[i].mutate(self.place_piece_rules[i].rule_make_sense,self.pl
ace_piece_rules[i].action_make_sense)
        #reset order of rules
        self.evaluating_place_piece_rules=self.place_piece_rules

        self.evaluating=False
        #updated priority of rules based on the rule quality parameter
        self.choose_piece_rules=sorted(self.choose_piece_rules,key=lambda a:
a.rule_quality,reverse=True)
        self.place_piece_rules=sorted(self.place_piece_rules,key=lambda a:
a.rule_quality,reverse=True)
        #now evaluate whole genome over a 100 games span
        self.evaluating_genome=True
        wins=0
        for _ in range(100):
            game = quarto.Quarto()
            playerindex=random.randint(0,1)
            game.set_players((RandomPlayer(game), self) if playerindex==1 else
(self, RandomPlayer(game)))
            self.set_quarto(game)
            winner = game.run()
            if winner==playerindex:
                wins+=1
        #fitness of genome
        self.fitness= wins - 0.4*self.random_pick


    def __str__(self):
        return f'\tChoose piece rules :\n{NEWLINE.join([str(rule) for rule in
self.choose_piece_rules])} \n\t; Place piece rules :\n{NEWLINE.join([str(rule) for
rule in self.place_piece_rules])}'

def generate_rules():
    #function that generated a set of rules, not used anymore .... xxxxx
    return [Rule(True,None) for _ in range(random.randint(MINRULES,MAXRULES))],
[Rule(False,None) for _ in range(random.randint(MINRULES,MAXRULES))]
```

> ***rule.py*** This file contains the whole rule implementation, so an if node and a then node

```
import random
import agents.quartolib as quartolib
#characteristic that are true
TRUE_PROPS=['high','solid','square','coloured']
#is the object a number (float, bool, int) or a string?
def isnumber(a):
    return (isinstance(a,int) or isinstance(a,float) or isinstance(a,bool))
#transform to a numeric representation
def tonum(a):
    if isnumber(a):
        return a
    else:
        return 0
#dict of if operations, each operation is characterized by the name and an
associated lambda that takes two parameters(even the ones that
# use only one like not) and return a value
IF_OPERATIONS={
    'mul': (lambda a,b: a*b if isnumber(a) and isnumber(b) else float(a) if
isnumber(a) else float(b) if isnumber(b) else 0),
    'add': (lambda a,b: a+b if isnumber(a) and isnumber(b) else float(a) if
isnumber(a) else float(b) if isnumber(b) else 0),
    'sub': (lambda a,b: a-b if isnumber(a) and isnumber(b) else float(a) if
isnumber(a) else float(b) if isnumber(b) else 0),
    'not': (lambda a,b: not a),
    'or': (lambda a,b: a or b),
    'truechar':(lambda a,b: a in TRUE_PROPS),
    'falsechar':(lambda a,b: a not in TRUE_PROPS),
    'gt':(lambda a,b:tonum(a)>tonum(b)),
    'lt':(lambda a,b:tonum(a)<tonum(b)),
    'gte':(lambda a,b:tonum(a)>=tonum(b)),
    'lte':(lambda a,b:tonum(a)<=tonum(b)),
    'and': (lambda a,b: a and b),
    'eq': (lambda a,b: a==b),
    'ne': (lambda a,b: a!=b)
}
#then operations in case of place rules, these operations returns either the left
node, or the right node always!
THEN_PLACE_OPERATIONS={
    'colmore': (lambda quarto,a,b: a if
quartolib.compare_elements_in_columns(quarto,a[0],b[0]) else b),
    'colless': (lambda quarto,a,b: b if
quartolib.compare_elements_in_columns(quarto,a[0],b[0]) else a),
    'rowmore': (lambda quarto,a,b: a if
quartolib.compare_elements_in_rows(quarto,a[1],b[1]) else b),
    'rowless': (lambda quarto,a,b: b if
quartolib.compare_elements_in_rows(quarto,a[1],b[1]) else a),
    'diagmore': (lambda quarto,a,b: a if
quartolib.compare_elements_in_diag(quarto,a,b) else b),
    'diagless': (lambda quarto,a,b: b if
quartolib.compare_elements_in_diag(quarto,a,b) else a),
    'antidiagmore': (lambda quarto,a,b: a if
quartolib.compare_elements_in_antidiag(quarto,a,b) else b),
    'antidiagless': (lambda quarto,a,b: b if
quartolib.compare_elements_in_antidiag(quarto,a,b) else a),
    'possible': (lambda quarto,a,b: a if quartolib.place_possible(quarto,a,b) else
b)
}
#then operations in case of choose piece, these operations returns either the left
node, or the right node always!
THEN_CHOOSE_OPERATIONS={
    'moreunique': (lambda quarto,a,b: a if
quartolib.compare_uniqueness(quarto,a,b) else b),
    'lessunique': (lambda quarto,a,b: b if
quartolib.compare_uniqueness(quarto,a,b) else a),
    'trues': (lambda quarto,a,b: a if quartolib.compare_trues(quarto,a,b) else b),
    'falses': (lambda quarto,a,b: b if quartolib.compare_trues(quarto,a,b) else
a),
    'diffinmostusedrownotcomplete': (lambda quarto,a,b: a if
quartolib.more_different_in_most_used_row_not_complete(quarto,a,b) else b),
    'similarinmostusedrownotcomplete': (lambda quarto,a,b: b if
quartolib.more_different_in_most_used_row_not_complete(quarto,a,b) else a),
    'diffinmostusedcolnotcomplete': (lambda quarto,a,b: a if
quartolib.more_different_in_most_used_column_not_complete(quarto,a,b) else b),
    'similarinmostusedcolumnnotcomplete': (lambda quarto,a,b: b if
quartolib.more_different_in_most_used_column_not_complete(quarto,a,b) else a),
    'diffinlessusedrownotcomplete': (lambda quarto,a,b: a if
quartolib.more_different_in_less_used_row(quarto,a,b) else b),
    'similarinlessusedrownotcomplete': (lambda quarto,a,b: b if
quartolib.more_different_in_less_used_row(quarto,a,b) else a),
    'diffinlessusedcolnotcomplete': (lambda quarto,a,b: a if
quartolib.more_different_in_less_used_column(quarto,a,b) else b),
    'similarinlessusedcolumnnotcomplete': (lambda quarto,a,b: b if
quartolib.more_different_in_less_used_column(quarto,a,b) else a),
    'differentdiag': (lambda quarto,a,b: a if
quartolib.more_different_in_diagonal(quarto,a,b) else b),
    'similardiag': (lambda quarto,a,b: b if
quartolib.more_different_in_diagonal(quarto,a,b) else a),
    'differentantidiag': (lambda quarto,a,b: a if
quartolib.more_different_in_antidiagonal(quarto,a,b) else b),
    'similardiag': (lambda quarto,a,b: b if
quartolib.more_different_in_antidiagonal(quarto,a,b) else a),
    'possible': (lambda quarto,a,b: a if quartolib.choose_possible(quarto,a,b)
else b)
}
#then leaf place functions, these functions are the ones in the leafs of the then
trees, they return a placing action
THEN_LEAF_PLACE_FUNCTIONS=quartolib.get_then_place_functions()
#then leaf choose functions, these functions are the ones used in the leafs of the
then trees, they return a choosing action
THEN_LEAF_CHOOSE_FUNCTIONS=quartolib.get_then_choose_functions()
```

```
IF_OPERATIONS_LIST=list(IF_OPERATIONS.keys())
#used choosing operations
IF_OPERATIONS_CHOOSE=['not','or','and','ne','eq','ne']#,'truechar','falsechar']
#used placing operations
IF_OPERATIONS_PLACE=
['mul','add','sub','not','or','gt','lt','gte','lte','and','eq','ne']
#operations that require only one operand
IF_OPERATIONS_WITH_ONE_OPERAND=set(['not','truechar','falsechar'])
#list of functions that the if tree can use, these functions return basic data
about the current state of the game
#they can be seen as a cooked status
IF_CHOOSE_FUNCTIONS=quartolib.get_choose_functions()
IF_PLACE_FUNCTIONS=quartolib.get_place_functions()
#last type of value that a leaf in the if trees can be, a list of constants
IF_PLACE_VALUES=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,True,False]
IF_CHOOSE_VALUES=
['high','not_high','solid','not_solid','coloured','not_coloured','square','not_squ
are',True,False]
#list composing everything in groups
IF_POSSIBLE_CHOOSE_VALUES=[[(op,'operation') for op in IF_OPERATIONS_CHOOSE],
[(func,'function') for func in IF_CHOOSE_FUNCTIONS]+[(val,'value') for val in
IF_CHOOSE_VALUES]]

IF_POSSIBLE_PLACE_VALUES=[[(op,'operation') for op in IF_OPERATIONS_PLACE],
[(func,'function') for func in IF_PLACE_FUNCTIONS]+[(val,'value') for val in
IF_PLACE_VALUES]]

THEN_POSSIBLE_CHOOSE_VALUES=[[(op,'operation') for op in
list(THEN_CHOOSE_OPERATIONS.keys())],[(leaf,'leaf') for leaf in
THEN_LEAF_CHOOSE_FUNCTIONS]]

THEN_POSSIBLE_PLACE_VALUES=[[(op,'operation') for op in
list(THEN_PLACE_OPERATIONS.keys())],[(leaf,'leaf') for leaf in
THEN_LEAF_PLACE_FUNCTIONS]]

#max depth of trees
MAX_DEPTH=6
class ThenNode:
    #the then node is the action used after checking the condition of the rule
    def __init__(self,parent,choose_piece,quarto) -> None:
        self.parent=parent
        #info about parent, choose rule etc.
        self.quarto=quarto
        self.choose_piece=choose_piece
        #childs are other ThenNode elements
        self.childs=[]
        #update depth
        self.depth=0 if self.parent is None else self.parent.depth+1
        #value is a random pick from the possible ones, if the depth is maxed out
pick a leaf node, otherwise random between operation or leaf
        #value=random.choice(THEN_POSSIBLE_CHOOSE_VALUES[0 if self.parent is None
else random.choice([0,1,1]) if self.depth<MAX_DEPTH else 1] if self.choose_piece
else THEN_POSSIBLE_PLACE_VALUES[0 if self.parent is None else
random.choice([0,1,1]) if self.depth<MAX_DEPTH else 1])
```

```
        value=random.choice(THEN_POSSIBLE_CHOOSE_VALUES[0 if self.parent is None
else random.randint(0,1) if self.depth<MAX_DEPTH else 1] if self.choose_piece else
THEN_POSSIBLE_PLACE_VALUES[0 if self.parent is None else random.randint(0,1) if
self.depth<MAX_DEPTH else 1])
        #setup info about the node, like if the node is a leaf etc.
        if self.parent is None:
            value=('possible','operation')
        self.value=value[0]
        self.op=value[1]=='operation'
        self.leaf=value[1]=='leaf'
        self.optdict=None
        if self.op:
            #if the node is an operation generate two childs
            self.optdict=THEN_CHOOSE_OPERATIONS if self.choose_piece else
THEN_PLACE_OPERATIONS
            self.childs.append(ThenNode(self,self.choose_piece,self.quarto))
            self.childs.append(ThenNode(self,self.choose_piece,self.quarto))

    def mutate(self):
        if random.random()<0.5 and self.op:
            #mutate one of the childs or both
            num=random.randint(0,2)
            if num==0 or num==2:
                self.childs[0].mutate()
            if num==1 or num==2:
                self.childs[1].mutate()
        else:
            #mutate myself
            if random.random()<0.5 and self.parent is not None:
                #go full random, don't care about what type of thing I was before

#value=random.choice(THEN_POSSIBLE_CHOOSE_VALUES[random.choice([0,1,1]) if
self.depth<MAX_DEPTH else 1] if self.choose_piece else
THEN_POSSIBLE_PLACE_VALUES[random.choice([0,1,1]) if self.depth<MAX_DEPTH else 1])
                value=random.choice(THEN_POSSIBLE_CHOOSE_VALUES[0 if self.parent
is None else random.randint(0,1) if self.depth<MAX_DEPTH else 1] if
self.choose_piece else THEN_POSSIBLE_PLACE_VALUES[0 if self.parent is None else
random.randint(0,1) if self.depth<MAX_DEPTH else 1])
                self.value=value[0]
                self.op=value[1]=='operation'
                self.leaf=value[1]=='leaf'
                if self.op:
                    self.optdict=THEN_CHOOSE_OPERATIONS if self.choose_piece else
THEN_PLACE_OPERATIONS
                    #if i didn't have any old child or randomly generate new ones
because Im an operation
                    if random.random()<1/3 or len(self.childs)!=2:
                        self.childs=[]

self.childs.append(ThenNode(self,self.choose_piece,self.quarto))

self.childs.append(ThenNode(self,self.choose_piece,self.quarto))
                    elif len(self.childs)==2:
                        #else If I had some childs and the random picked this
```

```
option mutate some childs or none
                        num=random.randint(0,6)
                        if num==0 or num==2:
                            self.childs[0].mutate()
                        if num==1 or num==2:
                            self.childs[1].mutate()
                else:
                    #If im not an operation setup childs as nothing again
                    self.childs=[]
            else:
                #keep my old type(leaf or operation)
                value=random.choice(THEN_POSSIBLE_CHOOSE_VALUES[0 if self.op else
1] if self.choose_piece else THEN_POSSIBLE_PLACE_VALUES[0 if self.op else 1])
                self.value=value[0]
                self.op=value[1]=='operation'
                self.leaf=value[1]=='leaf'
                if self.op:
                    self.optdict=THEN_CHOOSE_OPERATIONS if self.choose_piece else
THEN_PLACE_OPERATIONS
                    if random.random()<1/3:
                        self.childs=[]

self.childs.append(ThenNode(self,self.choose_piece,self.quarto))

self.childs.append(ThenNode(self,self.choose_piece,self.quarto))
                    else:
                        num=random.randint(0,6)
                        if num==0 or num==2:
                            self.childs[0].mutate()
                        if num==1 or num==2:
                            self.childs[1].mutate()

                else:
                    self.childs=[]

    def set_quarto(self,quarto):
        #set quarto for myself and childs
        self.quarto=quarto
        if self.op:
            for child in self.childs:
                child.set_quarto(self.quarto)

    def action(self):
        if not self.op:
            #if Im a leaf call the function that I refer to
            return self.value(self.quarto)
        else:
            #else evaluate childs and then call teh operation
            evals=[child.action() for child in self.childs]
            left_val,rigth_val=evals[0],evals[1]
            return self.optdict[self.value](self.quarto,left_val,rigth_val)

    def __str__(self):
        #string view
        if self.op:
            return f'({str(self.childs[0])}) {self.value} ({str(self.childs[1])})'
        else:
            return f'{self.value.__name__}'


class IfNode:
    #the ifnode node is the condition controlled for the rule, very similar to
thennode
    def __init__(self,parent,choose_piece,quarto) -> None:
        self.parent=parent
        #info
        self.quarto=quarto
        self.choose_piece=choose_piece
        self.childs=[]
        self.depth=0 if self.parent is None else self.parent.depth+1
        #value=random.choice(IF_POSSIBLE_CHOOSE_VALUES[0 if self.parent is None
else random.randint(0,2)] if choose_piece else IF_POSSIBLE_PLACE_VALUES[0 if
self.parent is None else random.randint(0,2)])
        #get random value
        value=random.choice(IF_POSSIBLE_CHOOSE_VALUES[0 if self.parent is None
else random.randint(0,1) if self.depth<MAX_DEPTH else 1] if self.choose_piece else
IF_POSSIBLE_PLACE_VALUES[0 if self.parent is None else random.randint(0,1) if
self.depth<MAX_DEPTH else 1])
        #setup value info
        self.value=value[0]
        self.op=value[1]=='operation'
        self.func=value[1]=='function'
        self.val=value[1]=='value'
        if self.op:
            #print(f'Need a child or two depending if the operation requires one
or two')
            self.childs.append(IfNode(self,self.choose_piece,self.quarto))
            if self.value not in IF_OPERATIONS_WITH_ONE_OPERAND:
                self.childs.append(IfNode(self,self.choose_piece,self.quarto))
        else:
            pass
            #print(f'No need for childs')

    def mutate(self):
        if random.random()<0.5 and self.op:
            #mutate one of the childs or both if I have two and the random pick
that option
            num=random.randint(0,2)
            if (num==0 or num==2) or (self.value in
IF_OPERATIONS_WITH_ONE_OPERAND):
                self.childs[0].mutate()
            if (num==1 or num==2) and (self.value not in
IF_OPERATIONS_WITH_ONE_OPERAND):
                self.childs[1].mutate()
        else:
            #mutate myself
            if random.random()<0.5 and self.parent is not None:
                #go full random, don't care about what type of thing I was before
```

```
#value=random.choice(IF_POSSIBLE_CHOOSE_VALUES[random.randint(0,2)] if
self.choose_piece else IF_POSSIBLE_PLACE_VALUES[random.randint(0,2)])
                value=random.choice(IF_POSSIBLE_CHOOSE_VALUES[0 if self.parent is
None else random.randint(0,1) if self.depth<MAX_DEPTH else 1] if self.choose_piece
else IF_POSSIBLE_PLACE_VALUES[0 if self.parent is None else random.randint(0,1) if
self.depth<MAX_DEPTH else 1])
                self.value=value[0]
                self.op=value[1]=='operation'
                self.func=value[1]=='function'
                self.val=value[1]=='value'
                if self.op:
                    #print(f'Need a child or two')
                    if random.random()<1/3 and len(self.childs)>0:
                        #keep old childs
                        if self.value in IF_OPERATIONS_WITH_ONE_OPERAND and
len(self.childs)==2:
                            #remove one of the two childs in case
                            self.childs.pop(random.randint(0,len(self.childs)-1))
                        elif (self.value not in IF_OPERATIONS_WITH_ONE_OPERAND)
and (len(self.childs)==1):

self.childs.append(IfNode(self,self.choose_piece,self.quarto))
                        num=random.randint(0,6)
                        if num==0 or num==2 or (num==3 and (self.value in
IF_OPERATIONS_WITH_ONE_OPERAND)):
                            self.childs[0].mutate()
                        if (num==1 or num==2) and (self.value not in
IF_OPERATIONS_WITH_ONE_OPERAND):
                            self.childs[1].mutate()
                    else:
                        self.childs=[]

self.childs.append(IfNode(self,self.choose_piece,self.quarto))
                        if self.value not in IF_OPERATIONS_WITH_ONE_OPERAND:

self.childs.append(IfNode(self,self.choose_piece,self.quarto))
                else:
                    self.childs=[]
            else:
                #keep my old type
                value=random.choice(IF_POSSIBLE_CHOOSE_VALUES[0 if self.op else 1]
if self.choose_piece else IF_POSSIBLE_PLACE_VALUES[0 if self.op else 1])
                self.value=value[0]
                self.op=value[1]=='operation'
                self.func=value[1]=='function'
                self.val=value[1]=='value'
                if self.op:
                    #print(f'Need a child or two')
                    if random.random()<1/3 and len(self.childs)>0:
                        #keep old childs
                        if self.value in IF_OPERATIONS_WITH_ONE_OPERAND and
len(self.childs)==2:
                            #remove one of the two childs in case
                            self.childs.pop(random.randint(0,len(self.childs)-1))
                        elif (self.value not in IF_OPERATIONS_WITH_ONE_OPERAND)
and (len(self.childs)==1):

self.childs.append(IfNode(self,self.choose_piece,self.quarto))
                        num=random.randint(0,6)
                        if num==0 or num==2 or (num==3 and (self.value in
IF_OPERATIONS_WITH_ONE_OPERAND)):
                            self.childs[0].mutate()
                        if (num==1 or num==2) and (self.value not in
IF_OPERATIONS_WITH_ONE_OPERAND):
                            self.childs[1].mutate()
                    else:
                        self.childs=[]

self.childs.append(IfNode(self,self.choose_piece,self.quarto))
                        if self.value not in IF_OPERATIONS_WITH_ONE_OPERAND:

self.childs.append(IfNode(self,self.choose_piece,self.quarto))
                else:
                    self.childs=[]

    def set_quarto(self,quarto):
        self.quarto=quarto
        if self.op:
            for child in self.childs:
                child.set_quarto(self.quarto)

    def eval(self):
        #evaluate condition
        if not self.op:
            #if Im a leaf return my constant value or call the function Im
associated with
            if self.val:
                return self.value
            else:
                return self.value(self.quarto)
        else:
            #else evaluate childs and then call operation
            evals=[child.eval() for child in self.childs] + [None,None]
            left_val,rigth_val=evals[0],evals[1]
            return IF_OPERATIONS[self.value](left_val,rigth_val)

    def __str__(self):
        #string view
        if self.op:
            if self.value not in IF_OPERATIONS_WITH_ONE_OPERAND:
                return f'({str(self.childs[0])}) {self.value}
({str(self.childs[1])})'
            else:
                return f'{self.value} ({str(self.childs[0])})'

        elif self.func:
            return f'{self.value.__name__}'
        else:
            return f'{self.value}'

class Rule:
    def __init__(self,choose_piece,quarto):
        #my trees, the if one and the then one and my info, so if Im a choose
piece rule or a place piece one, and my quarto
        self.quarto=quarto
        self.choose_piece=choose_piece
        self.if_node=IfNode(None,self.choose_piece,self.quarto)
        self.then_node=ThenNode(None,self.choose_piece,self.quarto)
        #data and evaluations
        #does the if tree make sense
        self.rule_make_sense=True
        #does the then tree make sense
        self.action_make_sense=True
        self.rule_quality=0
        self.rule_evaluations=0
        self.rule_true=0
        #game evaluations
        self.game_true=0
        self.game_evaluations=0
        #evaluations on then tree
        self.action_possible=0

    def set_quarto(self,quarto):
        #set my quarto and the ones of the if and then trees
        self.quarto=quarto
        self.if_node.set_quarto(self.quarto)
        self.then_node.set_quarto(self.quarto)

    def evaluate(self):
        #check the if node, this function is called to check if the rule is true
or not
        return self.if_node.eval()

    def evaluate_game_rule(self,won):
        # update rule trueness and its evaluations
        self.rule_true+=self.game_true
        self.rule_evaluations+=self.game_evaluations
        #rule make sense check if the if tree make sense, so if it is true at
least one time but not true every single time, the limit
        # is put at 2/3 now, but it can be changed making rules more strict
        self.rule_make_sense= self.rule_true>0 and
self.rule_true<(2/3)*self.rule_evaluations
        #action make sense check if the then tree is plausible, so if the times
that the action returned is possible is pretty high or not
        self.action_make_sense=self.rule_true>0 and self.action_possible>=(2*
(self.rule_true/3))
        #rule quality is a metric that helps rules that are taken a small amount
of time but in those times they perform good, this metric
        # is used to sort rules, because I want to check first rules that are more
precise
        self.rule_quality+=((self.game_evaluations-
self.game_true)/self.game_evaluations) * 10 if won else -
((self.game_true/self.game_evaluations) * 10)

    def mutate(self,rule_make_sense=True,action_possible=True):
        if (random.random()<0.5 and rule_make_sense) or (rule_make_sense and not
action_possible):
            #mutate then tree if the if tree make sense and need to update the
then one, or randomly
            self.then_node.mutate()
        else:
            #mutate if tree
            self.if_node.mutate()
        #reset values
        self.reset_evaluation_stats()

    def evaluated(self,thruth,action):
        #update stats of the game at each pick or choose
        self.game_evaluations+=1
        self.game_true+=thruth
        self.action_possible+=action

    def reset_evaluation_stats(self):
        #reset all the stats, function called after mutations, crossover and stuff
        self.rule_make_sense=True
        self.action_make_sense=True
        self.rule_quality=0
        self.rule_evaluations=0
        self.rule_true=0
        self.game_true=0
        self.game_evaluations=0
        self.action_possible=0

    def reset_game_stats(self):
        #reset just stats of the game
        self.game_true=0
        self.game_evaluations=0

    def needs_evaluation(self):
        # do I need to be evaluated or Im an old rule with already some data
        return self.rule_evaluations==0

    def action(self):
        # check the action associated with the rule, so check the then node
        return self.then_node.action()

    def __str__(self):
        #string view
        return f'Rule: If {str(self.if_node)} --> Then {self.then_node}'
```

> ***quartolib.py*** This file contains a set of functions that returns informations that a certain rule can exploit

```
import numpy as np
import random
import quarto
#this library is composed of all the function used by nodes etc., a lot of them
are redundant
NUMROWS=4
NUMCOLUMNS=4
POSITIONS=[(a,b) for a in range(4) for b in range(4)]
def get_placed_pieces(board) :
    #get pieces placed on the board already
    return list(board[board!=-1])

def num_pieces_chosen(quarto) -> int:
    #get num of pieces on the board
    return len(get_placed_pieces(quarto.get_board_status()))

def num_pieces_left(quarto) -> int:
    #num of pieces not on board
    return (NUMROWS*NUMCOLUMNS) -
len(get_placed_pieces(quarto.get_board_status()))

def less_used_characteristic(quarto):
    #get less used characteristic, so maybe high etc.
    board=quarto.get_board_status()
    placed_pieces=get_placed_pieces(board)
    free_pieces=[_ for _ in range(NUMROWS*NUMCOLUMNS) if _ not in placed_pieces]
    placed_pieces_char=
{'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
re':0,'not_square':0}
    for piece in placed_pieces:
        piece_char=quarto.get_piece_charachteristics(piece)
        placed_pieces_char['high']+=piece_char.HIGH
        placed_pieces_char['coloured']+=piece_char.COLOURED
        placed_pieces_char['solid']+=piece_char.SOLID
        placed_pieces_char['square']+=piece_char.SQUARE
        placed_pieces_char['not_high']+=not piece_char.HIGH
        placed_pieces_char['not_coloured']+=not piece_char.COLOURED
        placed_pieces_char['not_solid']+=not piece_char.SOLID
        placed_pieces_char['not_square']+=not piece_char.SQUARE
    minval,minchar=None,'high'
    for k,v in placed_pieces_char.items():
        if minval is None or minval>v:
            minval=v
            minchar=k
    return minchar

def most_used_characteristic(quarto):
    #get most used characteristic, so maybe high etc.
    board=quarto.get_board_status()
    placed_pieces=get_placed_pieces(board)
    free_pieces=[_ for _ in range(NUMROWS*NUMCOLUMNS) if _ not in placed_pieces]
    placed_pieces_char=
{'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
re':0,'not_square':0}
    for piece in placed_pieces:
        piece_char=quarto.get_piece_charachteristics(piece)
        placed_pieces_char['high']+=piece_char.HIGH
        placed_pieces_char['coloured']+=piece_char.COLOURED
        placed_pieces_char['solid']+=piece_char.SOLID
        placed_pieces_char['square']+=piece_char.SQUARE
        placed_pieces_char['not_high']+=not piece_char.HIGH
        placed_pieces_char['not_coloured']+=not piece_char.COLOURED
        placed_pieces_char['not_solid']+=not piece_char.SOLID
        placed_pieces_char['not_square']+=not piece_char.SQUARE
    maxval,maxchar=None,'high'
    for k,v in placed_pieces_char.items():
        if maxval is None or maxval<v:
            maxval=v
            maxchar=k
    return maxchar

def most_used_row(quarto):
    #get which row is the most used one
    return np.argmax(np.count_nonzero(quarto.get_board_status()!=-1,axis=1))

def most_used_column(quarto):
    #get which column is the most used one
    return np.argmax(np.count_nonzero(quarto.get_board_status()!=-1,axis=0))

def less_used_row(quarto):
    #get which row is the least used one
    return np.argmin(np.count_nonzero(quarto.get_board_status()!=-1,axis=1))

def less_used_column(quarto):
    #get which column is the least used one
    return np.argmin(np.count_nonzero(quarto.get_board_status()!=-1,axis=0))

def most_used_column_not_complete(quarto):
    #get which column is the most used one, except for the full ones
    count=np.count_nonzero(quarto.get_board_status()!=-1,axis=0).tolist()
    maxcount=0
    maxcol=0
    for a in range(NUMCOLUMNS):
        if count[a]>=maxcount and count[a]<NUMCOLUMNS:
            maxcount=count[a]
            maxcol=a
    return maxcol

def most_used_row_not_complete(quarto):
    #get which row is the most used one, except for the full ones
    count=np.count_nonzero(quarto.get_board_status()!=-1,axis=1).tolist()
    maxcount=0
    maxrow=0
    for a in range(NUMROWS):
        if count[a]>=maxcount and count[a]<NUMROWS:
            maxcount=count[a]
            maxrow=a
    return maxrow

def num_elements_in_antidiagonal(quarto):
    #num pieces in the antidiagonal
    return np.count_nonzero(np.fliplr(quarto.get_board_status()).diagonal()!=-1)

def num_elements_in_diagonal(quarto):
    #num pieces in the diagonal
    return np.count_nonzero(quarto.get_board_status().diagonal()!=-1)

def num_pieces_in_most_used_row(quarto):
    #num pieces in the most used row
    return np.count_nonzero(quarto.get_board_status()!=-1,axis=1)
[most_used_row(quarto)]

def num_pieces_in_most_used_column(quarto):
    #num pieces in the most used column
    return np.count_nonzero(quarto.get_board_status()!=-1,axis=1)
[most_used_column(quarto)]

def num_pieces_in_most_used_row_not_complete(quarto):
    #num pieces in most used row not complete
    return np.count_nonzero(quarto.get_board_status()!=-1,axis=1)
[most_used_row_not_complete(quarto)]

def num_pieces_in_most_used_column_not_complete(quarto):
    #num pieces in most used column not complete
    return np.count_nonzero(quarto.get_board_status()!=-1,axis=1)
[most_used_column_not_complete(quarto)]

def num_pieces_in_less_used_row(quarto):
    #num pieces in less used row
    return np.count_nonzero(quarto.get_board_status()!=-1,axis=1)
[less_used_column(quarto)]

def num_pieces_in_less_used_column(quarto):
    #num pieces in less used column
    return np.count_nonzero(quarto.get_board_status()!=-1,axis=1)
[less_used_column(quarto)]

def element_in_less_used_column(quarto):
    #pick a place in the less used column
    column=less_used_column(quarto)
    possible_placements=[(a[1],a[0]) for a in
np.argwhere(quarto.get_board_status()==-1).tolist() if a[1]==column]
    return random.choice(possible_placements)

def element_in_less_used_row(quarto):
    #pick a place in the least used row
    row=less_used_row(quarto)
    possible_placements=[(a[1],a[0]) for a in
np.argwhere(quarto.get_board_status()==-1).tolist() if a[0]==row]
    return random.choice(possible_placements)
```

```
def element_in_most_used_row_not_complete(quarto):
    #pick a place in the most used row that is yet to fill
    row=most_used_row_not_complete(quarto)
    possible_placements=[(a[1],a[0]) for a in
np.argwhere(quarto.get_board_status()==-1).tolist() if a[0]==row]
    return random.choice(possible_placements)

def element_in_most_used_column_not_complete(quarto):
    #pick a place in the most used column that is yet to fill
    column=most_used_column_not_complete(quarto)
    possible_placements=[(a[1],a[0]) for a in
np.argwhere(quarto.get_board_status()==-1).tolist() if a[1]==column]
    return random.choice(possible_placements)


def element_in_diagonal(quarto):
    #pick a place in the diagonal
    return random.choice([a for a in POSITIONS if a[0]==a[1]])

def element_in_antidiagonal(quarto):
    #pick a place in the antidiagonal
    return random.choice([a for a in POSITIONS if a[0]+a[1]==NUMROWS-1])

def element_in_corner(quarto):
    #pick a place in the corner, so the border
    return random.choice([a for a in POSITIONS if a[0]==0 or a[0]==NUMCOLUMNS-1 or
a[1]==0 or a[1]==NUMROWS-1])

def element_inside(quarto):
    #pick a place not in the border
    return random.choice([a for a in POSITIONS if a[0]!=0 and a[0]!=NUMCOLUMNS-1
and a[1]!=0 and a[1]!=NUMROWS-1])

def not_high_piece(quarto):
    #pick a short piece, the functions below are all similar
    return random.choice([a for a in range(NUMCOLUMNS*NUMROWS) if not
quarto.get_piece_charachteristics(a).HIGH])

def not_solid_piece(quarto):
    return random.choice([a for a in range(NUMCOLUMNS*NUMROWS) if not
quarto.get_piece_charachteristics(a).SOLID])

def not_coloured_piece(quarto):
    return random.choice([a for a in range(NUMCOLUMNS*NUMROWS) if not
quarto.get_piece_charachteristics(a).COLOURED])

def not_square_piece(quarto):
    return random.choice([a for a in range(NUMCOLUMNS*NUMROWS) if not
quarto.get_piece_charachteristics(a).SQUARE])

def high_piece(quarto):
    return random.choice([a for a in range(NUMCOLUMNS*NUMROWS) if
quarto.get_piece_charachteristics(a).HIGH])
```

```
def solid_piece(quarto):
    return random.choice([a for a in range(NUMCOLUMNS*NUMROWS) if
quarto.get_piece_charachteristics(a).SOLID])

def coloured_piece(quarto):
    return random.choice([a for a in range(NUMCOLUMNS*NUMROWS) if
quarto.get_piece_charachteristics(a).COLOURED])

def square_piece(quarto):
    return random.choice([a for a in range(NUMCOLUMNS*NUMROWS) if
quarto.get_piece_charachteristics(a).SQUARE])

def place_possible(quarto,a,b):
    #get if a is possible, or b is possible
    return a in [(a[1],a[0]) for a in
np.argwhere(quarto.get_board_status()==-1).tolist()]

def choose_possible(quarto,a,b):
    #get if a is possible or b is possible
    return a in [_ for _ in range(NUMROWS*NUMCOLUMNS) if _ not in
get_placed_pieces(quarto.get_board_status())]

def compare_elements_in_columns(quarto,a,b):
    #is a column more full than b one?
    board=quarto.get_board_status()
    ela,elb=np.count_nonzero(board[:,a]!=-1),np.count_nonzero(board[:,b]!=-1)
    return True if ela>elb and ela!=NUMROWS else False


def compare_elements_in_rows(quarto,a,b):
    #is a row more full than b one?
    board=quarto.get_board_status()
    ela,elb=np.count_nonzero(board[a,:]!=-1),np.count_nonzero(board[b,:]!=-1)
    return True if ela>elb and ela!=NUMCOLUMNS else False

def compare_elements_in_diag(quarto,a,b):
    #is a diagonal more full than b one?
    board=quarto.get_board_status()
    ela,elb=np.count_nonzero(np.diagonal(board,a[0]-
a[1])!=-1),np.count_nonzero(np.diagonal(board,b[0]-b[1])!=-1)
    return ela>elb

def compare_elements_in_antidiag(quarto,a,b):
    #is a antidiagonal more full than b one?
    board=quarto.get_board_status()
    ela,elb=np.count_nonzero(np.diagonal(np.fliplr(board),a[0]-
a[1])!=-1),np.count_nonzero(np.diagonal(np.fliplr(board),b[0]-b[1])!=-1)
    return ela>elb

def compare_uniqueness(quarto,a,b):
    #is a more unique than b?

apiece,bpiece=quarto.get_piece_charachteristics(a),quarto.get_piece_charachteristi
cs(b)
```

```
    board=quarto.get_board_status()
    placed_pieces=get_placed_pieces(board)
    placed_pieces_char=
{'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
re':0,'not_square':0}
    for piece in placed_pieces:
        piece_char=quarto.get_piece_charachteristics(piece)
        placed_pieces_char['high']+=piece_char.HIGH
        placed_pieces_char['coloured']+=piece_char.COLOURED
        placed_pieces_char['solid']+=piece_char.SOLID
        placed_pieces_char['square']+=piece_char.SQUARE
        placed_pieces_char['not_high']+=not piece_char.HIGH
        placed_pieces_char['not_coloured']+=not piece_char.COLOURED
        placed_pieces_char['not_solid']+=not piece_char.SOLID
        placed_pieces_char['not_square']+=not piece_char.SQUARE
    aval=placed_pieces_char['high' if apiece.HIGH else 'not_high'] +
placed_pieces_char['coloured' if apiece.COLOURED else 'not_coloured'] +
placed_pieces_char['solid' if apiece.SOLID else 'not_solid']
+placed_pieces_char['square' if apiece.SQUARE else 'not_square']
    bval=placed_pieces_char['high' if bpiece.HIGH else 'not_high'] +
placed_pieces_char['coloured' if bpiece.COLOURED else 'not_coloured'] +
placed_pieces_char['solid' if bpiece.SOLID else 'not_solid']
+placed_pieces_char['square' if bpiece.SQUARE else 'not_square']
    return aval>bval


def compare_trues(quarto,a,b):
    #does a have more true characteristics than b?

apiece,bpiece=quarto.get_piece_charachteristics(a),quarto.get_piece_charachteristi
cs(b)
    return
apiece.HIGH+apiece.COLOURED+apiece.SOLID+apiece.SQUARE>bpiece.HIGH+bpiece.COLOURED
+bpiece.SOLID+bpiece.SQUARE

def more_different_in_most_used_row_not_complete(quarto,a,b):
    #is a more unique in the most used row that is yet to complete than b?

apiece,bpiece=quarto.get_piece_charachteristics(a),quarto.get_piece_charachteristi
cs(b)
    board=quarto.get_board_status()
    placed_pieces=get_placed_pieces(board[most_used_row_not_complete(quarto),:])
    placed_pieces_char=
{'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
re':0,'not_square':0}
    for piece in placed_pieces:
        piece_char=quarto.get_piece_charachteristics(piece)
        placed_pieces_char['high']+=piece_char.HIGH
        placed_pieces_char['coloured']+=piece_char.COLOURED
        placed_pieces_char['solid']+=piece_char.SOLID
        placed_pieces_char['square']+=piece_char.SQUARE
        placed_pieces_char['not_high']+=not piece_char.HIGH
        placed_pieces_char['not_coloured']+=not piece_char.COLOURED
        placed_pieces_char['not_solid']+=not piece_char.SOLID
```

```
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==len(placed_pieces)]
        achars=[k for k,v in {'high':apiece.HIGH,'not_high':not
    apiece.HIGH,'coloured':apiece.COLOURED,'not_coloured':not
    apiece.COLOURED,'solid':apiece.SOLID,'not_solid':not
    apiece.SOLID,'square':apiece.SQUARE,'not_square':not apiece.SQUARE}.items() if v]
        bchars=[k for k,v in {'high':bpiece.HIGH,'not_high':not
    bpiece.HIGH,'coloured':bpiece.COLOURED,'not_coloured':not
    bpiece.COLOURED,'solid':bpiece.SOLID,'not_solid':not
    bpiece.SOLID,'square':bpiece.SQUARE,'not_square':not bpiece.SQUARE}.items() if v]
        aval,bval=sum([k in chars for k in achars]),sum([k in chars for k in bchars])
        return aval>bval

    def more_different_in_most_used_column_not_complete(quarto,a,b):
        #is a more unique in the most used column that is yet to complete than b?

    apiece,bpiece=quarto.get_piece_charachteristics(a),quarto.get_piece_charachteristi
    cs(b)
        board=quarto.get_board_status()

    placed_pieces=get_placed_pieces(board[:,most_used_column_not_complete(quarto)])
        placed_pieces_char=
    {'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
    re':0,'not_square':0}
        for piece in placed_pieces:
            piece_char=quarto.get_piece_charachteristics(piece)
            placed_pieces_char['high']+=piece_char.HIGH
            placed_pieces_char['coloured']+=piece_char.COLOURED
            placed_pieces_char['solid']+=piece_char.SOLID
            placed_pieces_char['square']+=piece_char.SQUARE
            placed_pieces_char['not_high']+=not piece_char.HIGH
            placed_pieces_char['not_coloured']+=not piece_char.COLOURED
            placed_pieces_char['not_solid']+=not piece_char.SOLID
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==len(placed_pieces)]
        achars=[k for k,v in {'high':apiece.HIGH,'not_high':not
    apiece.HIGH,'coloured':apiece.COLOURED,'not_coloured':not
    apiece.COLOURED,'solid':apiece.SOLID,'not_solid':not
    apiece.SOLID,'square':apiece.SQUARE,'not_square':not apiece.SQUARE}.items() if v]
        bchars=[k for k,v in {'high':bpiece.HIGH,'not_high':not
    bpiece.HIGH,'coloured':bpiece.COLOURED,'not_coloured':not
    bpiece.COLOURED,'solid':bpiece.SOLID,'not_solid':not
    bpiece.SOLID,'square':bpiece.SQUARE,'not_square':not bpiece.SQUARE}.items() if v]
        aval,bval=sum([k in chars for k in achars]),sum([k in chars for k in bchars])
        return aval>bval

    def more_different_in_less_used_row(quarto,a,b):
        #is a more unique in the less used row than b?

    apiece,bpiece=quarto.get_piece_charachteristics(a),quarto.get_piece_charachteristi
    cs(b)
        board=quarto.get_board_status()
        placed_pieces=get_placed_pieces(board[less_used_row(quarto),:])
        placed_pieces_char=
    {'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
    re':0,'not_square':0}
        for piece in placed_pieces:
            piece_char=quarto.get_piece_charachteristics(piece)
            placed_pieces_char['high']+=piece_char.HIGH
            placed_pieces_char['coloured']+=piece_char.COLOURED
            placed_pieces_char['solid']+=piece_char.SOLID
            placed_pieces_char['square']+=piece_char.SQUARE
            placed_pieces_char['not_high']+=not piece_char.HIGH
            placed_pieces_char['not_coloured']+=not piece_char.COLOURED
            placed_pieces_char['not_solid']+=not piece_char.SOLID
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==len(placed_pieces)]
        achars=[k for k,v in {'high':apiece.HIGH,'not_high':not
    apiece.HIGH,'coloured':apiece.COLOURED,'not_coloured':not
    apiece.COLOURED,'solid':apiece.SOLID,'not_solid':not
    apiece.SOLID,'square':apiece.SQUARE,'not_square':not apiece.SQUARE}.items() if v]
        bchars=[k for k,v in {'high':bpiece.HIGH,'not_high':not
    bpiece.HIGH,'coloured':bpiece.COLOURED,'not_coloured':not
    bpiece.COLOURED,'solid':bpiece.SOLID,'not_solid':not
    bpiece.SOLID,'square':bpiece.SQUARE,'not_square':not bpiece.SQUARE}.items() if v]
        aval,bval=sum([k in chars for k in achars]),sum([k in chars for k in bchars])
        return aval>bval

    def more_different_in_less_used_column(quarto,a,b):
        #is a more unique in the less used column than b?

    apiece,bpiece=quarto.get_piece_charachteristics(a),quarto.get_piece_charachteristi
    cs(b)
        board=quarto.get_board_status()
        placed_pieces=get_placed_pieces(board[:,less_used_column(quarto)])
        placed_pieces_char=
    {'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
    re':0,'not_square':0}
        for piece in placed_pieces:
            piece_char=quarto.get_piece_charachteristics(piece)
            placed_pieces_char['high']+=piece_char.HIGH
            placed_pieces_char['coloured']+=piece_char.COLOURED
            placed_pieces_char['solid']+=piece_char.SOLID
            placed_pieces_char['square']+=piece_char.SQUARE
            placed_pieces_char['not_high']+=not piece_char.HIGH
            placed_pieces_char['not_coloured']+=not piece_char.COLOURED
            placed_pieces_char['not_solid']+=not piece_char.SOLID
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==len(placed_pieces)]
        achars=[k for k,v in {'high':apiece.HIGH,'not_high':not
    apiece.HIGH,'coloured':apiece.COLOURED,'not_coloured':not
    apiece.COLOURED,'solid':apiece.SOLID,'not_solid':not
    apiece.SOLID,'square':apiece.SQUARE,'not_square':not apiece.SQUARE}.items() if v]
        bchars=[k for k,v in {'high':bpiece.HIGH,'not_high':not
    bpiece.HIGH,'coloured':bpiece.COLOURED,'not_coloured':not
    bpiece.COLOURED,'solid':bpiece.SOLID,'not_solid':not
    bpiece.SOLID,'square':bpiece.SQUARE,'not_square':not bpiece.SQUARE}.items() if v]
        aval,bval=sum([k in chars for k in achars]),sum([k in chars for k in bchars])
        return aval>bval

    def more_different_in_diagonal(quarto,a,b):
        #is a more unique in the diagonal than b?

    apiece,bpiece=quarto.get_piece_charachteristics(a),quarto.get_piece_charachteristi
    cs(b)
        board=quarto.get_board_status()
        placed_pieces=get_placed_pieces(np.diag(board))
        placed_pieces_char=
    {'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
    re':0,'not_square':0}
        for piece in placed_pieces:
            piece_char=quarto.get_piece_charachteristics(piece)
            placed_pieces_char['high']+=piece_char.HIGH
            placed_pieces_char['coloured']+=piece_char.COLOURED
            placed_pieces_char['solid']+=piece_char.SOLID
            placed_pieces_char['square']+=piece_char.SQUARE
            placed_pieces_char['not_high']+=not piece_char.HIGH
            placed_pieces_char['not_coloured']+=not piece_char.COLOURED
            placed_pieces_char['not_solid']+=not piece_char.SOLID
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==len(placed_pieces)]
        achars=[k for k,v in {'high':apiece.HIGH,'not_high':not
    apiece.HIGH,'coloured':apiece.COLOURED,'not_coloured':not
    apiece.COLOURED,'solid':apiece.SOLID,'not_solid':not
    apiece.SOLID,'square':apiece.SQUARE,'not_square':not apiece.SQUARE}.items() if v]
        bchars=[k for k,v in {'high':bpiece.HIGH,'not_high':not
    bpiece.HIGH,'coloured':bpiece.COLOURED,'not_coloured':not
    bpiece.COLOURED,'solid':bpiece.SOLID,'not_solid':not
    bpiece.SOLID,'square':bpiece.SQUARE,'not_square':not bpiece.SQUARE}.items() if v]
        aval,bval=sum([k in chars for k in achars]),sum([k in chars for k in bchars])
        return aval>bval

    def more_different_in_antidiagonal(quarto,a,b):
        #is a more unique in the antidiagonal than b?

    apiece,bpiece=quarto.get_piece_charachteristics(a),quarto.get_piece_charachteristi
    cs(b)
        board=quarto.get_board_status()
        placed_pieces=get_placed_pieces(np.fliplr(board).diagonal())
        placed_pieces_char=
    {'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
    re':0,'not_square':0}
        for piece in placed_pieces:
            piece_char=quarto.get_piece_charachteristics(piece)
            placed_pieces_char['high']+=piece_char.HIGH
            placed_pieces_char['coloured']+=piece_char.COLOURED
            placed_pieces_char['solid']+=piece_char.SOLID
            placed_pieces_char['square']+=piece_char.SQUARE
            placed_pieces_char['not_high']+=not piece_char.HIGH
            placed_pieces_char['not_coloured']+=not piece_char.COLOURED
            placed_pieces_char['not_solid']+=not piece_char.SOLID
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==len(placed_pieces)]
        achars=[k for k,v in {'high':apiece.HIGH,'not_high':not
    apiece.HIGH,'coloured':apiece.COLOURED,'not_coloured':not
    apiece.COLOURED,'solid':apiece.SOLID,'not_solid':not
    apiece.SOLID,'square':apiece.SQUARE,'not_square':not apiece.SQUARE}.items() if v]
        bchars=[k for k,v in {'high':bpiece.HIGH,'not_high':not
    bpiece.HIGH,'coloured':bpiece.COLOURED,'not_coloured':not
    bpiece.COLOURED,'solid':bpiece.SOLID,'not_solid':not
    bpiece.SOLID,'square':bpiece.SQUARE,'not_square':not bpiece.SQUARE}.items() if v]
        aval,bval=sum([k in chars for k in achars]),sum([k in chars for k in bchars])
        return aval>bval

    def characteristic_in_most_used_row_not_complete(quarto):
        #get a characteristic used in the most used row yet to complete
        board=quarto.get_board_status()
        placed_pieces=get_placed_pieces(board[most_used_row_not_complete(quarto),:])
        placed_pieces_char=
    {'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
    re':0,'not_square':0}
        for piece in placed_pieces:
            piece_char=quarto.get_piece_charachteristics(piece)
            placed_pieces_char['high']+=piece_char.HIGH
            placed_pieces_char['coloured']+=piece_char.COLOURED
            placed_pieces_char['solid']+=piece_char.SOLID
            placed_pieces_char['square']+=piece_char.SQUARE
            placed_pieces_char['not_high']+=not piece_char.HIGH
            placed_pieces_char['not_coloured']+=not piece_char.COLOURED
            placed_pieces_char['not_solid']+=not piece_char.SOLID
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==len(placed_pieces)]
        return chars[0] if len(chars)>0 else 'high'

    def characteristic_in_most_used_column_not_complete(quarto):
        #get a characteristic used in the most used column yet to complete
        board=quarto.get_board_status()

    placed_pieces=get_placed_pieces(board[:,most_used_column_not_complete(quarto)])
        placed_pieces_char=
    {'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
    re':0,'not_square':0}
        for piece in placed_pieces:
            piece_char=quarto.get_piece_charachteristics(piece)
            placed_pieces_char['high']+=piece_char.HIGH
            placed_pieces_char['coloured']+=piece_char.COLOURED
            placed_pieces_char['solid']+=piece_char.SOLID
            placed_pieces_char['square']+=piece_char.SQUARE
            placed_pieces_char['not_high']+=not piece_char.HIGH
            placed_pieces_char['not_coloured']+=not piece_char.COLOURED
            placed_pieces_char['not_solid']+=not piece_char.SOLID
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==len(placed_pieces)]
        return chars[0] if len(chars)>0 else 'high'
```

```
def characteristic_in_less_used_column(quarto):
    #get a characteristic used in the least used column
    board=quarto.get_board_status()
    placed_pieces=get_placed_pieces(board[:,less_used_column(quarto)])
    placed_pieces_char=
{'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
re':0,'not_square':0}
    for piece in placed_pieces:
        piece_char=quarto.get_piece_charachteristics(piece)
        placed_pieces_char['high']+=piece_char.HIGH
        placed_pieces_char['coloured']+=piece_char.COLOURED
        placed_pieces_char['solid']+=piece_char.SOLID
        placed_pieces_char['square']+=piece_char.SQUARE
        placed_pieces_char['not_high']+=not piece_char.HIGH
        placed_pieces_char['not_coloured']+=not piece_char.COLOURED
        placed_pieces_char['not_solid']+=not piece_char.SOLID
        placed_pieces_char['not_square']+=not piece_char.SQUARE
    chars=[k for k,v in placed_pieces_char.items() if v==len(placed_pieces)]
    return chars[0] if len(chars)>0 else 'high'

def characteristic_in_less_used_row(quarto):
    #get a characteristic used in the least used row
    board=quarto.get_board_status()
    placed_pieces=get_placed_pieces(board[less_used_row(quarto),:])
    placed_pieces_char=
{'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
re':0,'not_square':0}
    for piece in placed_pieces:
        piece_char=quarto.get_piece_charachteristics(piece)
        placed_pieces_char['high']+=piece_char.HIGH
        placed_pieces_char['coloured']+=piece_char.COLOURED
        placed_pieces_char['solid']+=piece_char.SOLID
        placed_pieces_char['square']+=piece_char.SQUARE
        placed_pieces_char['not_high']+=not piece_char.HIGH
        placed_pieces_char['not_coloured']+=not piece_char.COLOURED
        placed_pieces_char['not_solid']+=not piece_char.SOLID
        placed_pieces_char['not_square']+=not piece_char.SQUARE
    chars=[k for k,v in placed_pieces_char.items() if v==len(placed_pieces)]
    return chars[0] if len(chars)>0 else 'high'

def characteristic_in_diagonal(quarto):
    #get a characteristic used in the diagonal
    board=quarto.get_board_status()
    placed_pieces=get_placed_pieces(np.diag(board))
    placed_pieces_char=
{'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
re':0,'not_square':0}
    for piece in placed_pieces:
        piece_char=quarto.get_piece_charachteristics(piece)
        placed_pieces_char['high']+=piece_char.HIGH
        placed_pieces_char['coloured']+=piece_char.COLOURED
        placed_pieces_char['solid']+=piece_char.SOLID
        placed_pieces_char['square']+=piece_char.SQUARE
        placed_pieces_char['not_high']+=not piece_char.HIGH
```

```
            placed_pieces_char['not_coloured']+=not piece_char.COLOURED
            placed_pieces_char['not_solid']+=not piece_char.SOLID
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==len(placed_pieces)]
        return chars[0] if len(chars)>0 else 'high'

    def characteristic_in_antidiagonal(quarto):
        #get a characteristic used in the antidiagonal
        board=quarto.get_board_status()
        placed_pieces=get_placed_pieces(np.fliplr(board).diagonal())
        placed_pieces_char=
    {'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
    re':0,'not_square':0}
        for piece in placed_pieces:
            piece_char=quarto.get_piece_charachteristics(piece)
            placed_pieces_char['high']+=piece_char.HIGH
            placed_pieces_char['coloured']+=piece_char.COLOURED
            placed_pieces_char['solid']+=piece_char.SOLID
            placed_pieces_char['square']+=piece_char.SQUARE
            placed_pieces_char['not_high']+=not piece_char.HIGH
            placed_pieces_char['not_coloured']+=not piece_char.COLOURED
            placed_pieces_char['not_solid']+=not piece_char.SOLID
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==len(placed_pieces)]
        return chars[0] if len(chars)>0 else 'high'

    def characteristic_not_in_most_used_row_not_complete(quarto):
        #get a characteristic NOT USED in the most used row yet to complete
        board=quarto.get_board_status()
        placed_pieces=get_placed_pieces(board[most_used_row_not_complete(quarto),:])
        placed_pieces_char=
    {'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
    re':0,'not_square':0}
        for piece in placed_pieces:
            piece_char=quarto.get_piece_charachteristics(piece)
            placed_pieces_char['high']+=piece_char.HIGH
            placed_pieces_char['coloured']+=piece_char.COLOURED
            placed_pieces_char['solid']+=piece_char.SOLID
            placed_pieces_char['square']+=piece_char.SQUARE
            placed_pieces_char['not_high']+=not piece_char.HIGH
            placed_pieces_char['not_coloured']+=not piece_char.COLOURED
            placed_pieces_char['not_solid']+=not piece_char.SOLID
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==0]
        return chars[0] if len(chars)>0 else 'high'

    def characteristic_not_in_most_used_column_not_complete(quarto):
        #get a characteristic NOT USED in the most used column yet to complete
        board=quarto.get_board_status()

    placed_pieces=get_placed_pieces(board[:,most_used_column_not_complete(quarto)])
        placed_pieces_char=
    {'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
    re':0,'not_square':0}
        for piece in placed_pieces:
            piece_char=quarto.get_piece_charachteristics(piece)
            placed_pieces_char['high']+=piece_char.HIGH
            placed_pieces_char['coloured']+=piece_char.COLOURED
            placed_pieces_char['solid']+=piece_char.SOLID
            placed_pieces_char['square']+=piece_char.SQUARE
            placed_pieces_char['not_high']+=not piece_char.HIGH
            placed_pieces_char['not_coloured']+=not piece_char.COLOURED
            placed_pieces_char['not_solid']+=not piece_char.SOLID
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==0]
        return chars[0] if len(chars)>0 else 'high'


    def characteristic_not_in_less_used_column(quarto):
        #get a characteristic NOT USED in the least used column
        board=quarto.get_board_status()
        placed_pieces=get_placed_pieces(board[:,less_used_column(quarto)])
        placed_pieces_char=
    {'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
    re':0,'not_square':0}
        for piece in placed_pieces:
            piece_char=quarto.get_piece_charachteristics(piece)
            placed_pieces_char['high']+=piece_char.HIGH
            placed_pieces_char['coloured']+=piece_char.COLOURED
            placed_pieces_char['solid']+=piece_char.SOLID
            placed_pieces_char['square']+=piece_char.SQUARE
            placed_pieces_char['not_high']+=not piece_char.HIGH
            placed_pieces_char['not_coloured']+=not piece_char.COLOURED
            placed_pieces_char['not_solid']+=not piece_char.SOLID
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==0]
        return chars[0] if len(chars)>0 else 'high'

    def characteristic_not_in_less_used_row(quarto):
        #get a characteristic NOT USED in the least used row
        board=quarto.get_board_status()
        placed_pieces=get_placed_pieces(board[less_used_row(quarto),:])
        placed_pieces_char=
    {'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
    re':0,'not_square':0}
        for piece in placed_pieces:
            piece_char=quarto.get_piece_charachteristics(piece)
            placed_pieces_char['high']+=piece_char.HIGH
            placed_pieces_char['coloured']+=piece_char.COLOURED
            placed_pieces_char['solid']+=piece_char.SOLID
            placed_pieces_char['square']+=piece_char.SQUARE
            placed_pieces_char['not_high']+=not piece_char.HIGH
            placed_pieces_char['not_coloured']+=not piece_char.COLOURED
            placed_pieces_char['not_solid']+=not piece_char.SOLID
            placed_pieces_char['not_square']+=not piece_char.SQUARE
        chars=[k for k,v in placed_pieces_char.items() if v==0]
        return chars[0] if len(chars)>0 else 'high'
```

```
def characteristic_not_in_diagonal(quarto):
    #get a characteristic NOT USED in the diagonal
    board=quarto.get_board_status()
    placed_pieces=get_placed_pieces(np.diag(board))
    placed_pieces_char=
{'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
re':0,'not_square':0}
    for piece in placed_pieces:
        piece_char=quarto.get_piece_charachteristics(piece)
        placed_pieces_char['high']+=piece_char.HIGH
        placed_pieces_char['coloured']+=piece_char.COLOURED
        placed_pieces_char['solid']+=piece_char.SOLID
        placed_pieces_char['square']+=piece_char.SQUARE
        placed_pieces_char['not_high']+=not piece_char.HIGH
        placed_pieces_char['not_coloured']+=not piece_char.COLOURED
        placed_pieces_char['not_solid']+=not piece_char.SOLID
        placed_pieces_char['not_square']+=not piece_char.SQUARE
    chars=[k for k,v in placed_pieces_char.items() if v==0]
    return chars[0] if len(chars)>0 else 'high'

def characteristic_not_in_antidiagonal(quarto):
    #get a characteristic NOT USED in the antidiagonal
    board=quarto.get_board_status()
    placed_pieces=get_placed_pieces(np.fliplr(board).diagonal())
    placed_pieces_char=
{'high':0,'not_high':0,'coloured':0,'not_coloured':0,'solid':0,'not_solid':0,'squa
re':0,'not_square':0}
    for piece in placed_pieces:
        piece_char=quarto.get_piece_charachteristics(piece)
        placed_pieces_char['high']+=piece_char.HIGH
        placed_pieces_char['coloured']+=piece_char.COLOURED
        placed_pieces_char['solid']+=piece_char.SOLID
        placed_pieces_char['square']+=piece_char.SQUARE
        placed_pieces_char['not_high']+=not piece_char.HIGH
        placed_pieces_char['not_coloured']+=not piece_char.COLOURED
        placed_pieces_char['not_solid']+=not piece_char.SOLID
        placed_pieces_char['not_square']+=not piece_char.SQUARE
    chars=[k for k,v in placed_pieces_char.items() if v==0]
    return chars[0] if len(chars)>0 else 'high'

#functions that serve kinda as an export, they return function that we want the
genomes to use

def get_then_place_functions():
    return
[element_in_less_used_row,element_in_less_used_column,element_in_most_used_row_not
_complete,element_in_most_used_column_not_complete,element_inside,element_in_diago
nal,element_in_antidiagonal,element_in_corner]

def get_then_choose_functions():
    return
[high_piece,not_high_piece,solid_piece,not_solid_piece,coloured_piece,not_coloured
_piece,square_piece,not_square_piece]
```

```
def get_choose_functions():
    return
[#num_elements_in_diagonal,num_elements_in_antidiagonal,num_pieces_in_less_used_co
lumn,num_pieces_in_less_used_row,num_pieces_in_most_used_column,

#num_pieces_in_most_used_row,num_pieces_in_most_used_column_not_complete,num_piece
s_in_most_used_row_not_complete,
        #num_pieces_chosen,num_pieces_left,

less_used_characteristic,most_used_characteristic,characteristic_in_most_used_row_
not_complete,characteristic_in_diagonal,

characteristic_in_most_used_column_not_complete,characteristic_in_antidiagonal,cha
racteristic_not_in_most_used_row_not_complete,characteristic_not_in_diagonal,

characteristic_not_in_most_used_column_not_complete,characteristic_not_in_antidiag
onal,characteristic_in_less_used_column,characteristic_in_less_used_row,

characteristic_not_in_less_used_column,characteristic_not_in_less_used_row]

def get_place_functions():
    return
[num_elements_in_diagonal,num_elements_in_antidiagonal,num_pieces_in_less_used_col
umn,num_pieces_in_less_used_row,num_pieces_in_most_used_column,

num_pieces_in_most_used_row,num_pieces_in_most_used_column_not_complete,num_pieces
_in_most_used_row_not_complete,

num_pieces_chosen,num_pieces_left,most_used_row,less_used_row,most_used_column,les
s_used_column,most_used_row_not_complete,most_used_column_not_complete]
```

Lab 1: Set Covering

Proposed Solution

```
import random
import heapq
import logging
import bisect
from typing import Callable
from time import perf_counter
NUMBER=0
METRIC=()
prob=()

class PriorityQueue:
    """A basic Priority Queue with simple performance optimizations, code taken
from Prof Squillero repo"""

    def __init__(self):
        self._data_heap = list()
        self._data_set = set()

    def __bool__(self):
        return bool(self._data_set)
    def __contains__(self, item):
        return item in self._data_set
    def push(self, item, p=None):
        assert item not in self, f"Duplicated element"
        if p is None:
            p = len(self._data_set)
        self._data_set.add(item)
        heapq.heappush(self._data_heap, (p, item))

    def pop(self):
        p, item = heapq.heappop(self._data_heap)
        self._data_set.remove(item)
        return item

def problem(N, seed=None):
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N // 5, N
// 2))))
        for n in range(random.randint(N, N * 5))
    ]

def __lentup__(a):
    """This functions returns the length of a tuple with multiple tuples inside,
so just the sum of
    the inner tuples length"""
    len_=0
    for i in a:
        len_+=len(i)
    return len_

def actions_(state):
    """This function generates the list of actions that we can take starting from
a state
    , states already seen or duplicates of already seen states shouldn't be
considered"""
    return [tup_ for tup_ in prob if tup_ not in state]

def goal_test(state):
    """This function tests if the current state has all of the numbers needed, to
do that
    we test the length of a set(set so no duplicates) that has all the valid
numbers of this state
    (a valid number is a number that is from 0 to N-1)"""
    return len(set(b for a in state for b in a))==NUMBER

def sequence_found(state):
    """This function returns the set of all the valid numbers found in this set
    (a valid number is a number that is from 0 to N-1)"""
    return set([b for a in state for b in a])
```

```
def result(state,action):
    """This function returns a new state starting from one and it and it adds a
list(action)
    to its data"""
    #generate a state starting from last one
    lt=list(state)
    bisect.insort_right(lt,action)
    #add action to the state
    return tuple(lt)

def search(initial_state:tuple,
    actions:Callable,
    goal_test:Callable,state_cost:dict,
    priority_function:Callable,priority_function_inner:Callable):
    """This function search for a valid solution"""
    frontier = PriorityQueue()
    state_cost.clear()
    state = initial_state
    state_cost[state] = 0
    while state is not None and not goal_test(state):
        #iterate through the actions of the state itself
        for a in actions(state):
            #generate the new state and its cost
            new_state=result(state,a)
            u_cost= 1
            state_cost_new_state=state_cost[state]+u_cost
            prio_=priority_function_inner(new_state)
            #here the METRIC is checked so if in this level we reached a bad
priority we shouldn't
            #even add this state to the frontier and just continue with the next
iteration
            if prio_>METRIC[state_cost_new_state]:
                continue
            if new_state not in state_cost and new_state not in frontier:
                #add this new state to the parent state and state cost
dictionaries
                state_cost_new_state=state_cost[state]+u_cost
                state_cost[new_state]=state_cost_new_state
                #push this new state into the frontier
                frontier.push(new_state,p=priority_function(prio_))
                #logging.info(f"Pushed state {new_state}, with priority {prio_}")
        #if there is an element to be popped from the frontier do it and set it as
the new state
        #otherwise put state as None because no element to expand is left so no
solution has been found
        if frontier:
            state=frontier.pop()
            #logging.info(f"Popped state {state}, with priority
{priority_function_inner(state)}")
            state_cost_new_state=state_cost[state]
        else:
            state=None
            print("Couldn't find any solution")
    #iterate through the state to get its path
    if state:
        print(f"Found a solution in {len(state):,} steps; visited
{len(state_cost):,} states")
    return state

def set_covering(numbers):
    """interface to the system, numbers is the list of size of problems we want to
solve"""
    for number in numbers:
        global NUMBER
        NUMBER=number
        #generate a new problem
        probl=problem(NUMBER,seed=42)
        global prob
        prob=tuple([tuple(a) for a in probl])
        #logging.info(f"Problem with value {NUMBER} : {prob}")
        #METRIC generation, this heuristic was derived from other tries with
growing N
        global METRIC
        METRIC=tuple([0,0,2]+[_*5 for _ in range(1,len(prob))])
        #setup for the search
        #the priority function used start with minus len of the sequence found of
that set
        #so if the state is like [[0,1,5,8]] the sequence found will be [0,1,5] so
its len
        #will be 3, so we have -3, then we add to this the len of the whole state,
len of
        #the State class has been written in a way to sum the len of all of the
lists inside
        #the outer list, so in this case we'll get 4, so -3 + 4 = 1, this will be
the priority
        #of this priority queue, in this way we favorite the states that don't
have any
        #duplicates or non valid numbers(8 in this case), so a state like [[0,3],
[2,5]]
        #will have priority equal to 0 and will be popped before the last one
        state_cost=dict()
        times_=perf_counter()
        state=tuple([()])

sol=search(initial_state=state,actions=actions_,goal_test=goal_test,state_cost=sta
te_cost,
            priority_function=lambda a: a,priority_function_inner=lambda a: -
len(sequence_found(a))+__lentup__(a))
        print(f"The search for problem of value {NUMBER} lasted {perf_counter()-
times_}")
        if sol:
            print(f"Solution {sol} with weight {__lentup__(sol)}")

#call the interface to solve the problems with the list of size of problems
set_covering([50])
```

Other solution

~~~
```python
import random
import heapq
import logging
import bisect
from typing import Callable
from time import perf_counter
NUMBER=0
WIDTH=0
prob=()

class PriorityQueue:
    """A basic Priority Queue with simple performance optimizations, code taken
from Prof Squillero repo"""

    def __init__(self):
        self._data_heap = list()
        self._data_set = set()

    def __bool__(self):
        return bool(self._data_set)
    def __contains__(self, item):
        return item in self._data_set
    def push(self, item, p=None):
        assert item not in self, f"Duplicated element"
        if p is None:
            p = len(self._data_set)
        self._data_set.add(item)
        heapq.heappush(self._data_heap, (p, item))

    def pop(self):
        p, item = heapq.heappop(self._data_heap)
        self._data_set.remove(item)
        return item

def problem(N, seed=None):
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N // 5, N
// 2))))
        for n in range(random.randint(N, N * 5))
    ]

#here are the functions used by the search algorithm used, the mem versions are
just versions that try to use a little bit less memory
#but because of how memory demanding the algorithm used is the impact is still
light
~~~

```
#while the width versions of the actions functions serve to take only the
WIDTH/100 most promising childs, this is done with a heuristics
#calculated in the function that is not aware of how the child will look like, so
it's not the optimal heuristic but it works well
def __lentup__(a):
    """This function is used to calculate the summed length of a tuple that
containes tuples"""
    len_=0
    for i in a:
        len_+=len(i)
    return len_

def __lentupmem__(a):
    len_=0
    for i in a:
        len_+=len(prob[i])
    return len_

def actions_(state):
    """This function generates the list of actions that we can take starting from
a state
    , states already seen or duplicates of already seen states shouldn't be
considered"""
    return [tup_ for tup_ in prob if tup_ not in state]

def set_still_needed(state):
    """This function generates the difference between two sets, so we can get
which numbers we haven't covered yet with our state"""
    return set([_ for _ in range(0,NUMBER)])-set(sequence_found(state))

def actionswidth(state):
    """This function generates the list of actions that we can take starting from
a state
    , states already seen or duplicates of already seen states shouldn't be
considered"""
    set_=set_still_needed(state)
    lensetneeded_=len(set_)
    list_=[]
    for a in prob:
        if a not in state:
            ind_=(-len(set_-set(a))+len(set_))*(len(a)//lensetneeded_)
            bisect.insort_right(list_,a,key=lambda a :ind_)
    return [tup_ for _,tup_ in zip(range(0,int(WIDTH/100*len(list_))),list_)]


def isDupPresentmem(state,cont):
    """Function to check for duplicates in the memory version in the memory
version"""
    for a in state:
        if prob[a]==cont:
            return True
    return False
```

```
def actionsmem(state):
    """This function generates the list of actions that we can take starting from
a state
    , states already seen or duplicates of already seen states shouldn't be
considered"""
    return [prob.index(a) for a in prob if not isDupPresentmem(state,a)]

def set_still_neededmem(state):
    return set([_ for _ in range(0,NUMBER)])-set(sequence_foundmem(state))

def actionsmemwidth(state):
    """This function generates the list of actions that we can take starting from
a state
    , states already seen or duplicates of already seen states shouldn't be
considered"""
    set_=set_still_neededmem(state)
    lensetneeded_=len(set_)
    list_=[]
    for a in prob:
        if not isDupPresentmem(state,a):
            ind_=(-len(set_-set(a))+len(set_))*(len(a)//lensetneeded_)
            bisect.insort_right(list_,prob.index(a),key=lambda a :ind_)
    return [tup_ for _,tup_ in zip(range(0,int(WIDTH/100*len(list_))),list_)]

def goal_test(state):
    """This function tests if the current state has all of the numbers needed, to
do that
    we test the length of a set(set so no duplicates) that has all the valid
numbers of this state
    (a valid number is a number that is from 0 to N-1)"""
    return len(set(b for a in state for b in a))==NUMBER

def goal_testmem(state):
    """This function tests if the current state has all of the numbers needed, to
do that
    we test the length of a set(set so no duplicates) that has all the valid
numbers of this state
    (a valid number is a number that is from 0 to N-1)"""
    return len(set([b for a in state for b in prob[a]]))==NUMBER
def sequence_found(state):
    """This function returns the set of all the valid numbers found in this set
    (a valid number is a number that is from 0 to N-1)"""
    return set([b for a in state for b in a])

def sequence_foundmem(state):
    """This function returns the set of all the valid numbers found in this set
    (a valid number is a number that is from 0 to N-1)"""
    return set([b for a in state for b in prob[a]])
def result(state,action):
    """This function returns a new state starting from one and it and it adds a
list(action)
    to its data"""
    #generate a state starting from last one
    lt=list(state)
    bisect.insort_right(lt,action)
    #add action to the state
    return tuple(lt)

def resultmem(state,action):
    """This function returns a new state starting from one and it and it adds a
list(action)
    to its data"""
    #generate a state starting from last one
    lt=list(state)
    bisect.insort_right(lt,action)
    return tuple(lt)
    #add action to the state

#functions that do the search algorithm
def search(initial_state:tuple,
    actions:Callable,
    goal_test:Callable,
    parent_state:dict,state_cost:dict,
    priority_function:Callable,unit_cost:Callable):
    """This function search for a valid solution"""
    frontier = PriorityQueue()
    parent_state.clear()
    state_cost.clear()
    state = initial_state
    parent_state[state] = None
    state_cost[state] = 0
    #print(f"\n\nNUMBER {NUMBER} WIDTH {WIDTH}\n\n")
    while state is not None and not goal_test(state):
        #iterate through the actions of the state itself
        #print(f"State {state}")
        #for a in actions(state):
        for a in actions(state):
            #generate the new state and its cost
            #print(f"\n\nTuple {a}")
            new_state=result(state,a)
            u_cost= unit_cost(new_state)
            if new_state not in state_cost and new_state not in frontier:
                #add this new state to the parent state and state cost
dictionaries
                parent_state[new_state]=state
                state_cost[new_state]=state_cost[state]+u_cost
                #push this new state into the frontier
                frontier.push(new_state,p=priority_function(new_state))
                #logging.info(f"Pushed state {new_state}, with priority
{priority_function(new_state)}")
        #if there is an element to be popped from the frontier do it and set it as
the new state
        #otherwise put state as None because no element to expand is left so no
solution has been found
        if frontier:
            state=frontier.pop()
            #logging.info(f"Popped state {state}, with priority
{priority_function(state)}")
```

```
        else:
            state=None
            print("Couldn't find any solution")
    #iterate through the state to get its path
    path = list()
    s = state
    while s:
        path.append(s)
        s = parent_state[s]
    print(f"Found a solution in {len(path):,} steps; visited {len(state_cost):,}
states")
    """if state:
        print(f"Found a solution in {len(state):,} steps; visited
{len(state_cost):,} states")
    return state"""
    return list(reversed(path))

def searchmem(initial_state:tuple,
    actions: Callable,
    goal_test:Callable,
    state_cost:dict,
    priority_function:Callable):
    """This function search for a valid solution"""
    frontier = PriorityQueue()
    state_cost.clear()

    state = initial_state
    state_cost[state] = 0
    #seq_found=()
    while state is not None and not goal_test(state):
        #iterate through the actions of the state itself
        for a in actions(state):
            #generate the new state and its cost
            new_state=resultmem(state,a)
            #new_seq=sequence_foundmem(new_state)
            #if new_seq==seq_found:
                #continue
            #u_cost= unit_cost(new_state)
            if new_state not in state_cost and new_state not in frontier:
                #add this new state to the parent state and state cost
dictionaries
                #parent_state[new_state]=state
                state_cost[new_state]=state_cost[state]+1
                #push this new state into the frontier
                frontier.push(new_state,p=priority_function(new_state))
                #logging.info(f"Pushed state {new_state}, with priority
{priority_function(new_state)}")
        #if there is an element to be popped from the frontier do it and set it as
the new state
        #otherwise put state as None because no element to expand is left so no
solution has been found
        if frontier:
            state=frontier.pop()
            #logging.info(f"Popped state {state}, with priority
{priority_function(state)}")
            #seq_found=sequence_foundmem(state)
            #prio_=priority_function(state)
            #if prio_>maxprio:
                #print(f"Popped state {state}, with priority {prio_}")
                #maxprio=prio_
        else:
            state=None
            print("Couldn't find any solution")
    if state:
        print(f"Found a solution in {len(state):,} steps; visited
{len(state_cost):,} states")
    return state

def set_covering(numbers,mem=False,widthex=False,widthval=100):
    """interface to the system, numbers is the list of size of problems we want to
solve , mem is a bool to start the mem version instead of the normal one, and also
widthex is a boolean to select if
the width is limited, and in that case widthval is the value to limit it, so 50
will make us search 50% of the nodes of a child etc.
so with 100 we have the normal version(a little bit slower because of the
additional calculation that without the widthex we don't do)"""
    if widthex:
        global WIDTH
        WIDTH=widthval
    for number in numbers:
        global NUMBER
        NUMBER=number
        #generate a new problem
        probl=problem(NUMBER,seed=42)
        global prob
        prob=tuple([tuple(a) for a in probl])
        print(f"Problem with value {NUMBER} : {prob}")
        #setup for the search
        #the priority function used start with minus len of the sequence found of
that set
        #so if the state is like [[0,1,5,8]] the sequence found will be [0,1,5] so
its len
        #will be 3, so we have -3, then we add to this the len of the whole state,
len of
        #the State class has been written in a way to sum the len of all of the
lists inside
        #the outer list, so in this case we'll get 4, so -3 + 4 = 1, this will be
the priority
        #of this priority queue, in this way we favorite the states that don't
have any
        #duplicates or non valid numbers(8 in this case), so a state like [[0,3],
[2,5]]
        #will have priority equal to 0 and will be popped before the last one
        state_cost=dict()
        times_=perf_counter()
        if mem:
            state=tuple()
            if widthex:
```

```
sol=searchmem(initial_state=state,actions=actionsmemwidth,goal_test=goal_testmem,s
tate_cost=state_cost,
                priority_function=lambda a: -
len(sequence_foundmem(a))+__lentupmem__(a))
            else:

sol=searchmem(initial_state=state,actions=actionsmem,goal_test=goal_testmem,state_
cost=state_cost,
                priority_function=lambda a: -
len(sequence_foundmem(a))+__lentupmem__(a))
            print(f"The search for problem of value {NUMBER} lasted
{perf_counter()-times_}")
            if sol:
                print(f"Solution {tuple([prob[a] for a in sol])} with weight
{__lentupmem__(sol)}")
        else:
            state=tuple([()])
            parent_state=dict()
            if widthex:

sol=search(initial_state=state,actions=actionswidth,goal_test=goal_test,parent_sta
te=parent_state,state_cost=state_cost,
                priority_function=lambda a: -
len(sequence_found(a))+__lentup__(a),unit_cost=lambda a: 1)
            else:

sol=search(initial_state=state,actions=actions_,goal_test=goal_test,parent_state=p
arent_state,state_cost=state_cost,
                priority_function=lambda a: -
len(sequence_found(a))+__lentup__(a),unit_cost=lambda a: 1)
            print(f"The search for problem of value {NUMBER} lasted
{perf_counter()-times_}")
            if len(sol)>0:
                sol_=sol[len(sol)-1]
                print(f"Path {sol} Solution {sol_} with weight
{__lentup__(sol_)}")

#call the function to solve the desired problem
set_covering([100],mem=False,widthex=True,widthval=5)
```

Lab 2: Set Covering (Genetic Algorithm)

Proposed Solution

```
import random
import logging
from collections import Counter
from numpy.random import choice
from scipy.stats import binom
from math import ceil
from typing import Callable
from time import perf_counter
```

```
PROBLEM_SIZE = 1000

OFFSPRING_SIZE = 200

POPULATION_SIZE = 50

NUM_GENERATIONS = 50

# DATA USED TO SEE WHICH TYPE OF CROSSOVER WAS WORKING BETTER, UNIFORM WERE
WORKING BETTER SO I KEPT IT
#winsuni=0
#winssplit=0
```

```
def problem(N, seed=None):
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N // 5, N
// 2))))
        for n in range(random.randint(N, N * 5))
    ]
```

```
problem_=problem(PROBLEM_SIZE,seed=42)
# MAXREPETITIONS REPRESENTS THE WORST CASE, THIS IS USED TO UNDERSTAND RELATIVELY
HOW MANY SETS TO REMOVE EACH TIME
PROBLEM_LEN=len(problem_)
MAXREPETITIONS=sum([len(problem_[ind]) for ind in range(PROBLEM_LEN)])
AVGPERSET=MAXREPETITIONS/PROBLEM_LEN
gen=0
problem_
```

```
def covered_elements(genome):
    return len(set([problem_[i][j] for i in range(PROBLEM_LEN) for j in
range(len(problem_[i])) if genome[i]]))

def goal_test(genome):
    # IS THE GENOME A VALID SOLUTION? TO DO IT JUST CHECK THE LEN OF THE SET
    return covered_elements(genome)==PROBLEM_SIZE

def weight_sol(genome):
    """Returns the weigth of a genome"""
    len_=0
    for i in range(PROBLEM_LEN):
        if genome[i]:
            len_+=len(problem_[i])
    return len_

def solution_sets(genome):
    """Function used to get the sets in a genome so it's used to print genomes at
the end"""
    return [problem_[i] for i in range(PROBLEM_LEN) if genome[i]]

def fitness_bonus_sol(genome):
    # len of set that we are generating minus number of repetitions plus bonus if
solution, so OPTIMAL SOLUTION has
    # fitness of PROBLEM_SIZE*2, because it would have len of the set as
PROBLEM_SIZE(it's a solution so the set would
    # contain each number) then the sum on the dictionary took from the counter of
repetitions would be 0, then
    # a bonus of PROBLEM_SIZE because the solution is valid, so PROBLEM_SIZE-
0+PROBLEM_SIZE = 2*PROBLEM_SIZE
    cov=len(set([problem_[i][j] for i in range(PROBLEM_LEN) for j in
range(len(problem_[i])) if genome[i]]))
    return cov -sum({k:v-1 for k,v in dict(Counter([problem_[i][j] for i in
range(PROBLEM_LEN) for j in range(len(problem_[i])) if
genome[i]])).items()}.values()) + int(cov==PROBLEM_SIZE)*PROBLEM_SIZE,cov

def fitness_bonus_genome(genome):
    """This fitness gives less bonus to the solutions comparing it to non
solution, in fact a genome that has covered PROBLEM_SIZE-1
    elements will get PROBLEM_SIZE-1 bonus points, so the gap between a solution
or not is really tiny, this favors non solutions a lot early on"""
    cov=len(set([problem_[i][j] for i in range(PROBLEM_LEN) for j in
range(len(problem_[i])) if genome[i]]))
    return 2*cov -sum({k:v-1 for k,v in dict(Counter([problem_[i][j] for i in
range(PROBLEM_LEN) for j in range(len(problem_[i])) if
genome[i]])).items()}.values()),cov

# WEIGTHS_ROULETTE IS THE DISTRIBUTION USED BEFORE AND AS DEFAULT(BUT NOW I USE A
UNIFORM ONE KINDA)
# THIS DISTRIBUTION IS A SIMPLE BINOMIAL ONE THAT IS ON THE BASIS OF THE RANKING,
NOW I DON'T USE IT
# BECAUSE WHEN WE HAVE ALL THE POPULATION THAT ARE SIMILAR WE STILL PICK MAINLY
THE FIRST 4 INDIVIDUALS
WEIGHTS_ROULETTE=[binom.pmf(k=_,n=POPULATION_SIZE-1,p=1/POPULATION_SIZE) for _ in
range(POPULATION_SIZE)]
#print(f"Weights roulette {WEIGHTS_ROULETTE}, their sum is
{sum(WEIGHTS_ROULETTE)}")

def select_parent(pop,weigths=WEIGHTS_ROULETTE,k=2):
    # Using a wheel roulette TO SELECT K PARENTS, THIS IS WITHOUT REPLACEMENT, SO
THE PARENT CAN'T BE TAKEN MORE THAN ONCE
    # IN A SINGLE CALL
    return [pop[ind] for ind in
choice(range(POPULATION_SIZE),k,p=weigths,replace=False )]

def cross_overint(genome1,genome2):
    """crossover that takes the union of the genomes and takes from that union the
average # of sets that the genomes take"""
    list_=[ind for ind in range(PROBLEM_LEN) if genome1[ind] or genome2[ind]]
    choices=choice(list_,(sum(genome1)+sum(genome2))//2,p=[1/len(list_) for _ in
range(len(list_))],replace=False)
    return tuple([True if _ in choices else False for _ in range(PROBLEM_LEN)])

def cross_overuni(genome1,genome2):
    """Uniform crossover, this is the one activated by default, it is weigthed"""
    return tuple([random.choice([genome1[ind],genome1[ind],genome2[ind]]) for ind
in range(PROBLEM_LEN)])


def cross_oversplit(genome1,genome2):
    """One point split crossover"""
    point = random.randint(0,PROBLEM_LEN-1)
    return genome1[:point] + genome2[point:] if random.choice([True,False]) else
genome2[:point] + genome1[point:]

def mutationsol(genome):
    """This function just adds one random set to the genome, this is because the
current genome isn't yet
    a solution, so we need to add sets(or swap)"""
    points= random.choice([ind for ind in range(PROBLEM_LEN) if not genome[ind]])
    return genome[:points]+(not genome[points],)+genome[points+1:]

def mutation(genome):
    # MUTATION THAT IS BASED ON A # OF ELEMENTS TO REMOVE, AND USING CERTAIN
HEURISTICS I WEIGTHED PROBABILITIES OF SWAPPING
    # REMOVING ONLY, ADDING ONLY ETC., THIS WORK COULD BE DONE IN A BETTER WAY
THO, MUTATING GENOMES THAT AREN'T SOLUTIONS
    # IN A SMARTER WAY AND ADDING TO THEM ONLY A SET OR SWAPPING, BUT NOT JUST
REMOVING, THAT WOULDN'T MAKE SENSE
    # NOW THIS MUTATIONS TRIES TO WORK IN A SMART WAY REMOVING A TON OF ELEMENTS
AT THE BEGINNING BECAUSE THERE WE ARE JUST DOING
    # EXPLORATION, AND AT THE END MOSTLY SWAPS BECAUSE WE TRY TO DO EXPLOITATION
    #points to remove
    points=set()
    points.add(random.choice([ind for ind in range(PROBLEM_LEN) if genome[ind]]))#
for _ in range(random.choice([1,2]) if sum(genome)>1 else 1)]
    #points to add
    points.add(random.choice([ind for ind in range(PROBLEM_LEN) if not
genome[ind]]))# for _ in range(random.choice([1,2]) if sum(genome)<PROBLEM_LEN-1
else 1)]
    #points to add)
    return tuple([not genome[ind] if ind in points else genome[ind] for ind in
range(PROBLEM_LEN)])
```

```
def gen_population():
    # THIS GEN POPULATION DOES TWO WORKS IN ONE, IT GENERATE A POPULATION AND
CLEAR A LOT OF SETS FROM EACH GENOME
```

```
    # THIS IS BECAUSE STARTING WITH LOTS OF SETS IN A GENOME IS STUPID, WE KNOW
THAT GOOD SOLUTIONS USUALLY HAVE
    # A LOW AMOUNT OF SETS TAKEN, EX. FROM 4 TO 10 MAX USUALLY, SO WE FOCUS ON
GIVING THE FALSE A HIGHER CHANCE
    # HERE, THIS WAY OF THINKING CAN BE SEEN ALSO IN THE WAY toRemove IS
FORMULATED AFTER, WE TRY TO REMOVE A LOT
    # OF SETS AT THE BEGINNING, BECAUSE SOLUTIONS WITH 50% OF SETS TAKEN ARE
ABSURD
    return [tuple(False for _ in range(PROBLEM_LEN)) for __ in
range(POPULATION_SIZE)]

fitness_used=fitness_bonus_genome
xover_used=cross_overuni
# GEN POPULATION, GIVE IT FITNESS VALUES AND SORT IT
population_=tuple(sorted([(genome,fitness_used(genome)) for genome in
gen_population()],key=lambda a: a[1],reverse=True))
# PRINT POPULATION BY FITNESS VALUES
print([_[1] for _ in population_])
sol=(tuple(False for _ in range(PROBLEM_LEN)),(-MAXREPETITIONS,0))
visited_genomes=set()
visited_genomes.add(population_[0])
```

```
def search(population__,best_sol,fitness_function: Callable,xover: Callable):
    #logging.info(f"At gen 0 we started with the population {[_[1] for _ in
population__]} while best sol is {best_sol[1]}")
    end=False
    for g in range(NUM_GENERATIONS):
        # GET THE WEIGTHS FOR THE PARENT SELECTION, I CHOSE A KINDA UNIFORM
DISTRIBUTION
        # BASED ON FITNESS, SO THE FITTER AN INDIVIDUAL IS RELATIVELY TO OTHER
ONES THE
        # HIGHER ARE HIS CHANCES TO BE PICKED
        offspring,minfit,maxfit=set(),min([_[1][0] for _ in
population__]),max([_[1][0] for _ in population__])
        weigths=[-minfit+population__[_][1][0]+1 for _ in range(POPULATION_SIZE)]
        weigths=[_/sum(weigths) for _ in weigths]
        solfound=any(_[1][1]==PROBLEM_SIZE for _ in population__)
        for i in range(OFFSPRING_SIZE):
            # MUTATION
            if random.random()<0.5 or maxfit==minfit:
                # GET A PARENT AND MUTATE IT
                parent=select_parent(population__,weigths,k=1)[0]
                if parent[1][1]==PROBLEM_SIZE:
                    o = mutation(parent[0])
                else:
                    o = mutationsol(parent[0])
                f = fitness_function(o)
            else:
                # GETS TWO PARENTS AND USE THEM
                parents= sorted(select_parent(population__,weigths,k=2),key=lambda
a: a[1],reverse=True)
```

```
                o = xover(parents[0][0], parents[1][0])
                f=fitness_function(o)
                # RECOMBINATION + MUTATION
                if random.random()<0.3:
                    if f[1]==PROBLEM_SIZE:
                        o1 = mutation(o)
                    else:
                        o1=mutationsol(o)
                    f1=fitness_function(o1)
                    # IF THE MUTATION DESTROYS THE RECOMBINATION JUST KEEP THE
RECOMBINATION
                    # OTHERWISE SAVE THIS MUTATION
                    if f1>f and (f1[1]==PROBLEM_SIZE)>=(f[1]==PROBLEM_SIZE):
                        o,f=o1,f1
            while (o,f) in visited_genomes:
                # NO DUPLICATES ALLOWED SO MUTATE
                if f[1]==PROBLEM_SIZE:
                    o = mutation(o)
                else:
                    o=mutationsol(o)
                f=fitness_function(o)
            offspring.add((o,f))
            visited_genomes.add((o,f))
            if f>best_sol[1] and f[1]==PROBLEM_SIZE:
                best_sol=(o,f)
            if f==2*PROBLEM_SIZE:
                end=True
                break
        if solfound or best_sol[1][0]>maxfit:
            # Favoring fitness
            if best_sol[1][0]>(-MAXREPETITIONS):
                population__ = tuple(sorted(list(population__)+list(offspring),
key=lambda i: i[1], reverse=True)[:POPULATION_SIZE])
            else:
                population__ = tuple(sorted(list(offspring), key=lambda i: i[1],
reverse=True)[:POPULATION_SIZE])
        else:
            # favoring more covering
            if best_sol[1][0]>(-MAXREPETITIONS):
                population__ = tuple(sorted(list(population__)+list(offspring),
key=lambda i: i[1][1], reverse=True)[:POPULATION_SIZE])
            else:
                population__ = tuple(sorted(list(offspring), key=lambda i: i[1]
[1], reverse=True)[:POPULATION_SIZE])
        if best_sol[1][0] > maxfit and all(not _[1][1]==PROBLEM_SIZE for _ in
population__):
            # Got worse and don't have any sol in my thing so let's put the best
sol back in the population
            list_=[]
            list_.append(best_sol)
            population__ = tuple(list_+list(population__)[:POPULATION_SIZE-1])
        #logging.info(f"At gen {g} we finished with the population {[_[1] for _ in
population__]} while best sol is {best_sol[1]}")
        if end:
            break
    return best_sol,population__
```

```
MAX_GENERATIONS=NUM_GENERATIONS
while sol[1][0]<1.2*PROBLEM_SIZE:
    gen+=NUM_GENERATIONS

sol,population_=search(population__=population_,best_sol=sol,fitness_function=fitn
ess_used,xover=xover_used)
    population_=tuple(sorted(list(population_), key=lambda i: i[1], reverse=True))
    visited_genomes=set()
    print(f"After #{gen} generations the current population has the following
fitness values( 2 * PROBLEM_SIZE IS OPTIMAL FITNESS, SO IN THIS CASE
{PROBLEM_SIZE+PROBLEM_SIZE} ) \n{[_[1] for _ in population_]}")
    if gen>=MAX_GENERATIONS and sol[1][0]<1.2*PROBLEM_SIZE:
        # GIVE UP IF NO SOLUTION WITH GOOD BLOAT IS IN THE POPULATION AFTER MAX
GENERATIONS
        print(f"Couldn't find a solution with less weigth than
{int(3*PROBLEM_SIZE-1.2*PROBLEM_SIZE)} in #{gen} generations (more than MAX), this
is the best solution found so far")
        break
```

```
print(f"Best solution found has fitness {sol[1]} individual with weight
{weight_sol(sol[0])} is {solution_sets(sol[0])}\nand it is a VALID SOLUTION?
{goal_test(sol[0])}\nbecause this genome covers {len(set([problem_[i][j] for i in
range(len(sol[0])) for j in range(len(problem_[i])) if sol[0][i]]))} numbers")
print(f"Found in #{gen} generations")
```

Other Solution

```
import random
import logging
from collections import Counter
from numpy.random import choice
from scipy.stats import binom
from math import ceil
from typing import Callable
from time import perf_counter
```

```
PROBLEM_SIZE = 50

OFFSPRING_SIZE = 1000

POPULATION_SIZE = 10
```

```
NUM_GENERATIONS = 50

# DATA USED TO SEE WHICH TYPE OF CROSSOVER WAS WORKING BETTER, UNIFORM WERE
WORKING BETTER SO I KEPT IT
#winsuni=0
#winssplit=0
```

```
def problem(N, seed=None):
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N // 5, N
// 2))))
        for n in range(random.randint(N, N * 5))
    ]
```

```
problem_=problem(PROBLEM_SIZE,seed=42)
# MAXREPETITIONS REPRESENTS THE WORST CASE, THIS IS USED TO UNDERSTAND RELATIVELY
HOW MANY SETS TO REMOVE EACH TIME
PROBLEM_LEN=len(problem_)
MAXREPETITIONS=sum([len(problem_[ind]) for ind in range(PROBLEM_LEN)])
AVGPERSET=MAXREPETITIONS/PROBLEM_LEN
gen=0
problem_
```

```
def goal_test(genome):
    # IS THE GENOME A VALID SOLUTION? TO DO IT JUST CHECK THE LEN OF THE SET
    return len(set([problem_[i][j] for i in range(PROBLEM_LEN) for j in
range(len(problem_[i])) if genome[i]]))==PROBLEM_SIZE

def weight_sol(genome):
    """Returns the weigth of a genome"""
    len_=0
    for i in range(PROBLEM_LEN):
        if genome[i]:
            len_+=len(problem_[i])
    return len_
def solution_sets(genome):
    """Function used to get the sets in a genome so it's used to print genomes at
the end"""
    return [problem_[i] for i in range(PROBLEM_LEN) if genome[i]]

def fitness_bonus_sol(genome):
    # len of set that we are generating minus number of repetitions plus bonus if
solution, so OPTIMAL SOLUTION has
    # fitness of PROBLEM_SIZE*2, because it would have len of the set as
```

```
PROBLEM_SIZE(it's a solution so the set would
    # contain each number) then the sum on the dictionary took from the counter of
repetitions would be 0, then
    # a bonus of PROBLEM_SIZE because the solution is valid, so PROBLEM_SIZE-
0+PROBLEM_SIZE = 2*PROBLEM_SIZE
    return len(set([problem_[i][j] for i in range(PROBLEM_LEN) for j in
range(len(problem_[i])) if genome[i]])) -sum({k:v-1 for k,v in
dict(Counter([problem_[i][j] for i in range(PROBLEM_LEN) for j in
range(len(problem_[i])) if genome[i]])).items()}.values()) +
int(goal_test(genome))*PROBLEM_SIZE

def fitness_bonus_genome(genome):
    """This fitness gives less bonus to the solutions comparing it to non
solution, in fact a genome that has covered PROBLEM_SIZE-1
    elements will get PROBLEM_SIZE-1 bonus points, so the gap between a solution
or not is really tiny, this favors non solutions a lot early on"""
    return 2*len(set([problem_[i][j] for i in range(PROBLEM_LEN) for j in
range(len(problem_[i])) if genome[i]])) -sum({k:v-1 for k,v in
dict(Counter([problem_[i][j] for i in range(PROBLEM_LEN) for j in
range(len(problem_[i])) if genome[i]])).items()}.values())

# WEIGTHS_ROULETTE IS THE DISTRIBUTION USED BEFORE AND AS DEFAULT(BUT NOW I USE A
UNIFORM ONE KINDA)
# THIS DISTRIBUTION IS A SIMPLE BINOMIAL ONE THAT IS ON THE BASIS OF THE RANKING,
NOW I DON'T USE IT
# BECAUSE WHEN WE HAVE ALL THE POPULATION THAT ARE SIMILAR WE STILL PICK MAINLY
THE FIRST 4 INDIVIDUALS
WEIGHTS_ROULETTE=[binom.pmf(k=_,n=POPULATION_SIZE-1,p=1/POPULATION_SIZE) for _ in
range(POPULATION_SIZE)]
#print(f"Weights roulette {WEIGHTS_ROULETTE}, their sum is
{sum(WEIGHTS_ROULETTE)}")

def select_parent(pop,weigths=WEIGHTS_ROULETTE,k=2):
    # Using a wheel roulette TO SELECT K PARENTS, THIS IS WITHOUT REPLACEMENT, SO
THE PARENT CAN'T BE TAKEN MORE THAN ONCE
    # IN A SINGLE CALL
    return [pop[ind] for ind in
choice(range(POPULATION_SIZE),k,p=weigths,replace=False )]

def cross_overint(genome1,genome2):
    """crossover that takes the union of the genomes and takes from that union the
average # of sets that the genomes take"""
    list_=[ind for ind in range(PROBLEM_LEN) if genome1[ind] or genome2[ind]]
    choices=choice(list_,(sum(genome1)+sum(genome2))//2,p=[1/len(list_) for _ in
range(len(list_))],replace=False)
    return tuple([True if _ in choices else False for _ in range(PROBLEM_LEN)])

def cross_overuni(genome1,genome2):
    """Uniform crossover, this is the one activated by default, it is weigthed"""
    return tuple([random.choice([genome1[ind],genome1[ind],genome2[ind]]) for ind
in range(PROBLEM_LEN)])


def cross_oversplit(genome1,genome2):
    """One point split crossover"""
    point = random.randint(0,PROBLEM_LEN-1)
    return genome1[:point] + genome2[point:] if random.choice([True,False]) else
genome2[:point] + genome1[point:]

"""def covered_elements(genome):
    return len(set([problem_[i][j] for i in range(PROBLEM_LEN) for j in
range(len(problem_[i])) if genome[i]]))"""

def mutationsol(genome):
    """This function just adds one random set to the genome, this is because the
current genome isn't yet
    a solution, so we need to add sets(or swap)"""
    point= random.choice([ind for ind in range(len(genome)) if not genome[ind]])
    return genome[:point]+(not genome[point],)+genome[point+1:]

def mutation(genome,toRemove):
    # MUTATION THAT IS BASED ON A # OF ELEMENTS TO REMOVE, AND USING CERTAIN
HEURISTICS I WEIGTHED PROBABILITIES OF SWAPPING
    # REMOVING ONLY, ADDING ONLY ETC., THIS WORK COULD BE DONE IN A BETTER WAY
THO, MUTATING GENOMES THAT AREN'T SOLUTIONS
    # IN A SMARTER WAY AND ADDING TO THEM ONLY A SET OR SWAPPING, BUT NOT JUST
REMOVING, THAT WOULDN'T MAKE SENSE
    # NOW THIS MUTATIONS TRIES TO WORK IN A SMART WAY REMOVING A TON OF ELEMENTS
AT THE BEGINNING BECAUSE THERE WE ARE JUST DOING
    # EXPLORATION, AND AT THE END MOSTLY SWAPS BECAUSE WE TRY TO DO EXPLOITATION
    #points to remove
    points= [random.choice([ind for ind in range(PROBLEM_LEN) if genome[ind]]) for
_ in range(choice([toRemove-1,toRemove],1,p=[0.2,0.8])[0] if toRemove>0 else
toRemove)]
    #points to add
    points.append(random.choice([ind for ind in range(PROBLEM_LEN) if not
genome[ind]]))
    return tuple([not genome[ind] if ind in points else genome[ind] for ind in
range(PROBLEM_LEN)])
```

```

def gen_population():
    """Generate a population"""
    return [tuple(random.choice([True,False]) for _ in range(PROBLEM_LEN)) for __
in range(POPULATION_SIZE)]
# DEFINITION OF FUNCTIONS USED
fitness_used=fitness_bonus_genome
xover_used=cross_overuni
# GEN POPULATION, GIVE IT FITNESS VALUES AND SORT IT
population_=tuple(sorted([(genome,fitness_used(genome)) for genome in
gen_population()],key=lambda a: a[1],reverse=True))
# PRINT POPULATION BY FITNESS VALUES
print([_[1] for _ in population_])
```

```
def search(population__,fitness_function: Callable,xover: Callable):
    """starting from a population try for NUM_GENERATIONS to get a new
population"""
    end=False
    for g in range(NUM_GENERATIONS):
        # GET THE WEIGTHS FOR THE PARENT SELECTION, I CHOSE A DISTRIBUTION
PROPORTIONAL
        # BASED ON FITNESS, SO THE FITTER AN INDIVIDUAL IS RELATIVELY TO OTHER
ONES THE
        # HIGHER ARE HIS CHANCES TO BE PICKED
        offspring,minfit=set(),population__[POPULATION_SIZE-1][1]
        weigths=[-minfit+population__[_][1]+1 for _ in range(POPULATION_SIZE)]
        weigths=[_/sum(weigths) for _ in weigths]
        for _ in range(OFFSPRING_SIZE):
            # MUTATION
            if random.random()<0.3:
                # GET A PARENT AND MUTATE IT
                parent=select_parent(population__,weigths,k=1)[0]
                o=mutation(parent[0],ceil((-(-PROBLEM_SIZE-
PROBLEM_SIZE+parent[1]))*PROBLEM_LEN/(10*MAXREPETITIONS)))
                f=fitness_function(o)
            else:
                # GETS TWO PARENTS AND USE THEM
                parents= sorted(select_parent(population__,weigths,k=2),key=lambda
a: a[1],reverse=True)
                o = xover(parents[0][0], parents[1][0])
                f=fitness_function(o)
                # RECOMBINATION + MUTATION
                if random.random()<0.3:
                    o1=mutation(o,ceil((-(-PROBLEM_SIZE-
PROBLEM_SIZE+f))*PROBLEM_LEN/(10*MAXREPETITIONS)))
                    f1=fitness_function(o1)
                    if (f1>minfit and f<minfit) or (f1>f and goal_test(o1)):
                        o,f=o1,f1
            while f>=minfit+AVGPERSET/2 and not goal_test(o):
                # THIS INDIVIDUAL IS REALLY CLOSE TO A GOOD SOLUTION, BUT IS NOT A
SOLUTION YET
                # AND ITS FITNESS IS ENOUGH TO MAKE IT SURVIVE, SO UNTIL
                # MUTATIONS KEEPS THE INDIVIDUAL IN THAT RANGE AND THE INDIVUAL IS
NO A SOLUTION YET
                # TRY AGAIN, A MUTATION COULD BREAK IT OR MAKE IT
                o= mutationsol(o)
                f=fitness_function(o)
            while (o,f) in offspring or (o,f) in population__:
                # NO DUPLICATES ALLOWED SO MUTATE
                o=mutation(o,ceil((-(-PROBLEM_SIZE-
PROBLEM_SIZE+f))*PROBLEM_LEN/(10*MAXREPETITIONS)))
                f=fitness_function(o)
            if goal_test(o):
                offspring.add((o,f))
            if f==2*PROBLEM_SIZE:
                end=True
                break
        population__ = tuple(sorted(list(offspring), key=lambda i: i[1], 
reverse=True)[:POPULATION_SIZE])
        print(f"At gen {g} we finished with the fitness pop of {[_[1] for _ in 
population__]}")
        if end:
            break
    return population__
```

```
MAX_GENERATIONS=NUM_GENERATIONS
while population_[0][1]<1.2*PROBLEM_SIZE:
    gen+=NUM_GENERATIONS
    
population_=search(population__=population_,xover=cross_oversplit,fitness_function
=fitness_used)
    print(f"After #{gen} generations the current population has the following 
fitness values( 2 * PROBLEM_SIZE IS OPTIMAL FITNESS, SO IN THIS CASE 
{PROBLEM_SIZE+PROBLEM_SIZE} ) \n{[_[1] for _ in population_]}")
    if gen>=MAX_GENERATIONS and population_[0][1]<1.2*PROBLEM_SIZE:
        # GIVE UP IF NO SOLUTION WITH GOOD BLOAT IS IN THE POPULATION AFTER MAX 
GENERATIONS
        print(f"Couldn't find a solution with less bloat in #{gen} generations 
(more than MAX), this is the best solution found so far")
        break
```

```
print(f"Best individual has fitness {population_[0][1]} individual is 
{solution_sets(population_[0][0])}\nwith weight {weight_sol(population_[0]
[0])}\nand it is a VALID SOLUTION? {goal_test(population_[0][0])}\nbecause this 
genome covers {len(set([problem_[i][j] for i in range(len(population_[0][0])) for 
j in range(len(problem_[i])) if population_[0][0][i]]))} numbers\nwith 
{sum(population_[0][0])} sets taken")
print(f"Found in #{gen} generations")
```

Lab 3: Nim

Lab 3.1 and 3.2 solution, hard coded agent and evolved agent

```
import logging
from collections import namedtuple
import random
import math
from numpy.random import choice
import functools
```

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

class Nim:
    def __init__(self, num_rows: int, k: int = None) -> None:
        self._rows = [i * 2 + 1 for i in range(num_rows)]
        self._k = k

    def __bool__(self):
        return sum(self._rows) > 0

    def __str__(self):
        return "<" + " ".join(str(_) for _ in self._rows) + ">"

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

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

```
def _nimsum(state):
    return functools.reduce(lambda a,b : a^b,state)

def pure_random(state: Nim) -> Nimply:
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    num_objects = random.randint(1, state.rows[row])
    return Nimply(row, num_objects)

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

def optimal(state: Nim) -> Nimply:
    nimsum=_nimsum(state._rows)
    if nimsum==0:
        return pure_random(state)
    else:
        for _ in reversed(range(len(state._rows))):
            if state._rows[_]^nimsum<state._rows[_]:
                return (_,state._rows[_]-(state._rows[_]^nimsum))
        return pure_random(state)
```

Hard coded agent

```
# DATA AND PLYS


def number_active_rows(nim):
    return sum([_>0 for _ in nim.rows])


def indexes_active_rows(nim):
    return [_ for _ in range(len(nim.rows)) if nim.rows[_]>0]

def els_active_rows(nim):
    return [_ for _ in nim.rows if _>0]

def els_max_row(nim):
    return max(els_active_rows(nim))

def avg_els_per_row(nim):
    return sum(nim.rows)/len(nim.rows)

def els_min_row(nim):
    return min(els_active_rows(nim))

def els_second_max_row(nim):
    return sorted(els_active_rows(nim),key=lambda a:a,reverse=True)[1]

def index_max_row(nim):
    return sorted(indexes_active_rows(nim),key=lambda a:nim.rows[a],reverse=True)
[0]

def index_min_row(nim):
    return sorted(indexes_active_rows(nim),key=lambda a:nim.rows[a])[0]

def index_row_middle_els(nim):
    return sorted(indexes_active_rows(nim),key=lambda a:nim.rows[a])
[int(len(indexes_active_rows(nim))/2)]

def index_second_max_row(nim):
    return sorted(indexes_active_rows(nim),key=lambda a:nim.rows[a],reverse=True)
[1]

def els_row_middle(nim):
    return nim.rows[sorted(indexes_active_rows(nim),key=lambda a:nim.rows[a])
[int(len(indexes_active_rows(nim))/2)]]

def sum_rows_odd(nim):
    return sum([True for _ in range(len(nim.rows)) if not nim.rows[_]%2])

def indexes_odd_rows(nim):
    return [_ for _ in range(len(nim.rows)) if not nim.rows[_]%2]

def index_max_odd_row(nim):
    return sorted(indexes_odd_rows(nim),key=lambda a:nim.rows[a],reverse=True)[0]
```

```
def index_first_untouched_row(nim,starting_rows):
    for index in range(len(nim.rows)):
        if nim.rows[index]==starting_rows[index]:
            return index

def els_max_odd_row(nim):
    return nim.rows[index_max_odd_row(nim)]

def number_of_rows_with_same_elements_as_starting_elements(nim,starting_rows):
    return sum([nim.rows[_]==starting_rows[_] for _ in range(len(nim.rows))])

def only_odd_rows(nim):
    return all([not nim.rows[_]%2 for _ in indexes_odd_rows(nim)])

# HARD CODED STRAT

def hard_coded_strategy(nim,max_start,starting_rows,rows):
    if number_active_rows(nim)<2:
        return (index_max_row(nim),els_max_row(nim))
    if
number_of_rows_with_same_elements_as_starting_elements(nim,starting_rows)>=rows/2:
        return (index_first_untouched_row(nim,starting_rows), 1)
    if number_active_rows(nim)>2:
        return (index_max_row(nim),els_second_max_row(nim))
    if avg_els_per_row(nim)>(max_start/4) and avg_els_per_row(nim)<(max_start/2):
        return (index_max_row(nim),els_max_row(nim)//2)
    if number_active_rows(nim)>rows/2:
        return (index_row_middle_els(nim),els_row_middle(nim))
    if sum_rows_odd(nim):
        return (index_max_odd_row(nim),els_max_odd_row(nim))
    if only_odd_rows(nim) and number_active_rows(nim)>1:
        return (index_max_row(nim),els_second_max_row(nim))



    else:
        return pure_random(nim)
```

Evolved Agent

```
# MOVES

def final_move(nim,rows,starting_rows,max_start):
    if number_active_rows(nim)<2:
        return (index_max_row(nim),els_max_row(nim))
    return (0,0)


def same_as_starting_move(nim,rows,starting_rows,max_start):
    if
number_of_rows_with_same_elements_as_starting_elements(nim,starting_rows)>=rows/2:
        return (index_first_untouched_row(nim,starting_rows), 1)
    return (0,0)

def pick_max_row_second_max_row(nim,rows,starting_rows,max_start):
    if number_active_rows(nim)>2:
        return (index_max_row(nim),els_second_max_row(nim))
    return (0,0)

def avg_move(nim,rows,starting_rows,max_start):
    if avg_els_per_row(nim)>(max_start/4) and avg_els_per_row(nim)<(max_start/2):
        return (index_max_row(nim),els_max_row(nim)//2)
    return (0,0)

def middle_move(nim,rows,starting_rows,max_start):
    if number_active_rows(nim)>rows/2:
        return (index_row_middle_els(nim),els_row_middle(nim))
    return (0,0)

def odd_move(nim,rows,starting_rows,max_start):
    if sum_rows_odd(nim):
        return (index_max_odd_row(nim),els_max_odd_row(nim))
    return (0,0)

def only_odd_move(nim,rows,starting_rows,max_start):
    if only_odd_rows(nim) and number_active_rows(nim)>1:
        return (index_max_row(nim),els_second_max_row(nim))
    return (0,0)


moves=
[final_move,same_as_starting_move,pick_max_row_second_max_row,avg_move,middle_move
,odd_move,only_odd_move]


def evolved_move(plays,nim,max_start,starting_rows,rows):
    #print(f"Move indexes {}")
    for move in plays:
        ply=moves[move](nim,rows,starting_rows,max_start)
        #print("Ply obtained",ply)
        if ply[1]>0:
            return ply
    return pure_random(nim)

class EvAlgRules:
    def __init__(self,rows,moves):
        self._population_size_=1
        self._offspring_size_=5
        self._games_per_genome_=10
        self._moves=moves
        self._nummoves=len(moves)
        self._number_all_genomes=math.factorial(self._nummoves)
        self._offspring=[]
        self._num_rows_=rows
        self._genomes=[tuple([i for i in range(self._nummoves)]) for _ in
range(self._population_size_)]
        self._genomes_visited_already_=set(_ for _ in self._genomes)
        self._population=[(g,self._fitness(g)) for g in self._genomes]
        self._population=sorted(self._population,key=lambda a: a[1],reverse=True)

    def _mutate(self,genome):
        points=choice(range(self._nummoves),2,p=[1/self._nummoves for _ in
range(self._nummoves)],replace=False)
        return tuple([genome[_] if _ not in points else (genome[points[1]] if
_==points[0] else genome[points[0]]) for _ in range(self._nummoves)])


    def get_best_player(self):
        return self._population[0][0]

    def pick_move(self,genome,nim,max_start,starting_rows,rows):
        return evolved_move(genome,nim,max_start,starting_rows,rows)

    def _fitness(self,genome):
        player=0
        starting=random.choice([True,False])
        wins=0
        NUM_GAMES=20
        NUM_ROWS=11
        MAX_START=21
        for _ in range(self._games_per_genome_):
            nim=Nim(11)
            #logging.debug(f"In this game I'm player #{0 if starting else 1}")
            starting_rows=[_ for _ in nim.rows]
            while nim:
                if starting!=player:

ply=self.pick_move(genome,nim,MAX_START,starting_rows,NUM_ROWS)
                else:
                    ply=pure_random(nim)
                nim.nimming(ply)
                #logging.debug(f"After player {player} move now rows are {nim}")
                player=1-player
            winner=1-player
            won=(winner==0 and starting) or (winner==1 and not starting)
            if won:
                wins+=1
            starting=random.choice([True,False])
        return wins/self._games_per_genome_



    def __str__(self):
        return f"The best player right now won {self._population[0]
[1]*self._games_per_genome_} against the pure random bot!"

    def evolve(self,gens):
        print("At beginning the ev was",self)
        for _ in range(gens):
            for __ in range(self._offspring_size_):
                #print(f"Found {len(self._genomes_visited_already_)} states 
already while max are {self._number_all_genomes}")
                if len(self._genomes_visited_already_)==self._number_all_genomes:
                    print(f"Found all possible combinations already!")
                    print(f"Finished gen {_}, now ev is",self)
                    return
                new_genome=None
                while new_genome is None or new_genome in self._offspring or 
new_genome in self._population:
                    new_genome=self._mutate(random.choice(self._population)[0])
                self._offspring.append((new_genome,self._fitness(new_genome)))
                self._genomes_visited_already_.add(new_genome)

self._population=tuple(sorted(list(self._population)+self._offspring,key=lambda a: 
a[1],reverse=True)[:self._population_size_])
            self._offspring=[]
            print(f"Finished gen {_}, now ev is",self)


myevplayer=EvAlgRules(4,moves)
myevplayer.evolve(10)
print(myevplayer)
best_player=myevplayer.get_best_player()
print(f"Best player is {best_player}")
```

```
wins=0
logging.getLogger().setLevel(logging.DEBUG)
starting=random.choice([True,False])
player=0
NUM_GAMES=20
NUM_ROWS=11
MAX_START=21
for _ in range(NUM_GAMES):
    nim=Nim(NUM_ROWS)
    starting_rows=[_ for _ in nim.rows]
    logging.debug(f"In this game I'm player #{0 if starting else 1}")
    logging.debug(f"Game starting, AM I STARTING??? {starting} with these rows 
{nim} and this nimsum {_nimsum(nim.rows)}")
    while nim:
        if starting!=player:

ply=myevplayer.pick_move(best_player,nim,MAX_START,starting_rows,NUM_ROWS)
        else:
            ply=gabriele(nim)
        nim.nimming(ply)
        logging.debug(f"After player {player} play {ply} move now rows are {nim} 
and nimsum is {_nimsum(nim.rows)}")
        player=1-player
    winner=1-player
    won=(winner==0 and starting) or (winner==1 and not starting)
    if won:
        wins+=1
    starting=random.choice([True,False])

logging.debug(f"After {NUM_GAMES} my player won {wins} games!")
```

Lab 3.3 : Minmax agent

```
import logging
from collections import namedtuple
import random
from numpy.random import choice
import functools
import copy
```

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

class Nim:
    def __init__(self, num_rows: int, k: int = None) -> None:
        self._rows = [i * 2 + 1 for i in range(num_rows)]
        self._k = k

    def __setRows__(self, rows):
        self._rows=copy.deepcopy(rows)

    def __bool__(self):
        return sum(self._rows) > 0

    def __str__(self):
        return "<" + " ".join(str(_) for _ in self._rows) + ">"

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

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

```
class MinMaxAgent:
    def __init__(self,num_rows):
        self._num_rows_=num_rows
        self._analyzed_moves_=dict()
        self._rules=dict()
    def pickmove(self,nim):
        if tuple(nim._rows) in self._analyzed_moves_:
            return self._analyzed_moves_[tuple(nim._rows)][0]
        moves=self.__possiblemoves(tuple(nim._rows))
        evals=[]
        for move in moves:
            nimCopy=Nim(self._num_rows_)
            nimCopy.__setRows__(nim._rows)
            #print(f"Move to ev {move} while nim rows are {nimCopy._rows} and
level is {0}")
            nimCopy.nimming(move)
            evals.append(self.minmax(nimCopy,1,False))
        max,maxIndex=None,None
        #logging.debug(f"PICKMOVE Finished evaluations for state {nim} and myturn
{True} with moves {moves} and evals {evals}")
        for _ in range(len(evals)):
            if max is None or evals[_]>max:
                max=evals[_]
                maxIndex=_
        return moves[maxIndex]


    def minmax(self,nim,val,myTurn):
        if not nim:
            return -1*val if myTurn else int(100/val)
        else:
            if myTurn and tuple(nim._rows) in self._analyzed_moves_:
                return self._analyzed_moves_[tuple(nim._rows)][1]
            moves=self.__possiblemoves(tuple(nim._rows))
            evals=[]
            for move in moves:
                nimCopy=Nim(self._num_rows_)
                nimCopy.__setRows__(nim._rows)
                #print(f"Move to ev {move} while nim rows are {nimCopy._rows} and
level is {val}")
                nimCopy.nimming(move)
                evals.append(self.minmax(nimCopy,val+1,not myTurn))
            #print(f"Finished evaluating level {val}")
            #logging.debug(f"Finished evaluations for state {nim} and myturn
{myTurn} with moves {moves} and evals {evals}")
            if myTurn:
                max,maxIndex=None,None
                for _ in range(len(evals)):
                    if max is None or evals[_]>max:
                        max=evals[_]
                        maxIndex=_
                self._analyzed_moves_[tuple(nim._rows)]=(moves[maxIndex],max)
                return max
            else:
                return min(evals)
    def __possiblemoves(self,state):
        #moves=[(row,toTake,0,0) for row in range(self._num_rows_) for toTake in
range(state[row])]
        #logging.debug(f"Moves for state {state} have len{len(moves)}")
        #return moves
        return [(row,toTake+1) for row in range(self._num_rows_) for toTake in
range(state[row])]
```

```
def gabriele(state: Nim) -> Nimply:
    """Pick always the maximum possible number of the lowest row"""
    possible_moves = [(r, o) for r, c in enumerate(state.rows) for o in range(1, c
+ 1)]
    return Nimply(*max(possible_moves, key=lambda m: (-m[0], m[1])))

def _nimsum(state):
    return functools.reduce(lambda a,b : a^b,state)

def pure_random(state: Nim) -> Nimply:
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    num_objects = random.randint(1, state.rows[row])
    return Nimply(row, num_objects)

def optimal(state: Nim) -> Nimply:
    nimsum=_nimsum(state._rows)
    #logging.debug(f"For state {state} nimsum is {nimsum}")
    if nimsum==0:
        return pure_random(state)
    else:
        for _ in reversed(range(len(state._rows))):
            #logging.debug(f"Nimsum between {state._rows[_]} and {nimsum} is
{state._rows[_]^nimsum}")
            if state._rows[_]^nimsum<state._rows[_]:
                return (_,state._rows[_]-(state._rows[_]^nimsum))
        return pure_random(state)
```

```
wins=0
logging.getLogger().setLevel(logging.DEBUG)
myPlayer=MinMaxAgent(11)
starting=True#random.choice([True,False])
player=0
NUM_GAMES=1
for _ in range(NUM_GAMES):
    nim=Nim(11)
    logging.debug(f"In this game I'm player #{0 if starting else 1} and nim is
{nim} with nimsum {_nimsum(nim._rows)}")
    while nim:
        if starting!=player:
            ply=myPlayer.pickmove(nim)
        else:
            ply=optimal(nim)
        nim.nimming(ply)
        logging.debug(f"After player {player} move now rows are {nim} with nimsum
{_nimsum(nim._rows)}")
        player=1-player
    winner=1-player
    won=(winner==0 and starting) or (winner==1 and not starting)
    if won:
        wins+=1
    starting=True#random.choice([True,False])

logging.debug(f"After {NUM_GAMES} my player won {wins} games!")
```

Lab 3.4: Reinforcement Learning

```
import logging
from collections import namedtuple
import random
from numpy.random import choice
import functools
```

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

class Nim:
    def __init__(self, num_rows: int, k: int = None) -> None:
        self._rows = [i * 2 + 1 for i in range(num_rows)]
        self._k = k

    def __bool__(self):
        return sum(self._rows) > 0

    def __str__(self):
        return "<" + " ".join(str(_) for _ in self._rows) + ">"

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

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

```
def _nimsum(state):
    return functools.reduce(lambda a,b : a^b,state)

def pure_random(state: Nim) -> Nimply:
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    num_objects = random.randint(1, state.rows[row])
    return Nimply(row, num_objects)

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

def optimal(state: Nim) -> Nimply:
    nimsum=_nimsum(state._rows)
    if nimsum==0:
        return pure_random(state)
    else:
        for _ in reversed(range(len(state._rows))):
            if state._rows[_]^nimsum<state._rows[_]:
                return (_,state._rows[_]-(state._rows[_]^nimsum))
        return pure_random(state)
```

```

class ReinforcementLearningAgent:
    def __init__(self,rows):
        self._rules=dict()
        self._game=[]
        self._num_rows_=rows

    def evaluate_game(self,won):
        for state,move in self._game:
            self._rules[state]=tuple(sorted([rule if rule!=move else
            ((rule[0],rule[1],rule[2]+1,rule[3]+1) if won else
(rule[0],rule[1],rule[2]-2,rule[3]+1))
             for rule in self._rules[state]],key=lambda a: a[2],reverse=True))
        self._game=[]

    def pickmove(self,state):
        #logging.debug(f"In pickmove with state {state}, state in self.rules?
{state in self._rules}")
        if state not in self._rules:
            #logging.debug(f"New state found {state}")

self._rules[state]=tuple(random.sample(self.__possiblemoves(state),self.__lenpossi
blemoves(state))[:5 if self.__lenpossiblemoves(state)>5 else
self.__lenpossiblemoves(state)])
            #logging.debug(f"Now moves for state {state} are
{self._rules[state]}")
        else:
            #logging.debug(f"Old state {state}")
            if any([rule[2]<0 and rule[3]>5 for rule in self._rules[state]]):
                #logging.debug(f"State has all moves evaluated already {state} ->
{self._rules[state]}")

new_rules=random.sample(self.__newpossiblemoves(state),len(self.__newpossiblemoves
(state)))[:5 if len(self.__newpossiblemoves(state))>5 else
len(self.__newpossiblemoves(state))]
                badcurrmoves=sum([rule[2]<0 and rule[3]>5 for rule in
self._rules[state]])
                len_new_rules=len(new_rules)
                if len_new_rules>0:
                    self._rules[state]=tuple(sorted(list(self._rules[state])
[:len(self._rules[state])-badcurrmoves]+new_rules[:badcurrmoves if
len_new_rules>badcurrmoves else len_new_rules],key=lambda a: a[2],reverse=True))
                #logging.debug(f"Now fixed and state {state} has moves ->
{self._rules[state]}")
        #logging.debug(f"Before picking a move the rules for state {state} are
{self._rules[state]}")
        #if any([rule[3]<3 for rule in self._rules[state]]):
            #picked_move=random.choice([rule for rule in self._rules[state] if
rule[3]<3])
        #else:
        minfit=min([rule[2] for rule in self._rules[state]])
        weigths=[-minfit+rule[2]+1 for rule in self._rules[state]]
        weigths=[_/sum(weigths) for _ in weigths]
        picked_move_index=choice(list(range(len(weigths))),1,p=weigths)[0]
        picked_move=self._rules[state][picked_move_index]
        self._game.append((state,picked_move))
        #logging.debug(f"Picked move {picked_move} for state {state}")
        return Nimply(picked_move[0],picked_move[1])

    def __possiblemoves(self,state):
        #moves=[(row,toTake,0,0) for row in range(self._num_rows_) for toTake in
range(state[row])]
        #logging.debug(f"Moves for state {state} have len{len(moves)}")
        #return moves
        return [(row,toTake+1,0,0) for row in range(self._num_rows_) for toTake in
range(state[row])]

    def __newpossiblemoves(self,state):
        return [(row,toTake+1,0,0) for row in range(self._num_rows_) for toTake in
range(state[row]) if (row,toTake+1) not in [(_[0],_[1]) for _ in
self._rules[state]]]

    def __lenpossiblemoves(self,state):
        #logging.debug(f"LEN POSSIBLE MOVES FOR STATE {state} is {sum(state)}")
        return sum(state)
```

```
mp=ReinforcementLearningAgent(11)
```

```
wins=0
logging.getLogger().setLevel(logging.DEBUG)
starting=random.choice([True,False])
player=0
NUM_GAMES=1000
for _ in range(NUM_GAMES):
    nim=Nim(11)
    #logging.debug(f"In this game I'm player #{0 if starting else 1}")
    while nim:
        if starting!=player:
            ply=mp.pickmove(nim.rows)
        else:
            ply=gabriele(nim)
        nim.nimming(ply)
        #logging.debug(f"After player {player} move now rows are {nim}")
        player=1-player
    winner=1-player
    won=(winner==0 and starting) or (winner==1 and not starting)
    mp.evaluate_game(won)
    if won:
        wins+=1
    starting=random.choice([True,False])

logging.debug(f"After {NUM_GAMES} my player won {wins} games!")
```