

# Computational Intelligence Exam Report

Enrico Magliano - s295692@studenti.polito.it

January 2023

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Final Project - Quarto</b>	<b>3</b>
2.1	Random Player . . . . .	3
2.2	My MinMax Player . . . . .	4
2.3	Pastimes Player . . . . .	8
2.3.1	Classic Pastimes strategy . . . . .	11
2.4	Reinforcing Learning Player . . . . .	16
2.5	Genetic Algorithm Player . . . . .	21
2.6	Block Strategy . . . . .	26
2.7	Conclusion . . . . .	28
2.8	Utilities . . . . .	29
<b>3</b>	<b>Laboratory</b>	<b>41</b>
3.1	Lab 1: Set Covering . . . . .	41
3.1.1	Review by Ruggero Nocera . . . . .	43
3.2	Lab 2: Set Covering with Genetic Algorithm . . . . .	45
3.2.1	Review by Gabriele Greco . . . . .	47
3.2.2	Review by Amin Mbare . . . . .	48
3.2.3	Review by Lorenzo Bellino . . . . .	49
3.2.4	Review by Davide Aiello . . . . .	49
3.2.5	Reviw by Diego Gasco . . . . .	50
3.3	Lab 3: Policy Search . . . . .	52
3.3.1	Task 1 . . . . .	52
3.3.2	Task 2 . . . . .	53
3.3.3	Task 3 . . . . .	56
3.3.4	Task 4 . . . . .	58
3.3.5	Review by dfm88 . . . . .	60
3.4	Review made by me . . . . .	62
3.4.1	Review dfm88 lab 1 . . . . .	62
3.4.2	Review s295103 lab 2 . . . . .	62
3.4.3	Review merhametsize lab 2 . . . . .	62
3.4.4	Review Diegomangasco lab3 . . . . .	63

# 1 Introduction

This report includes all my works done in the Computational Intelligence course of Politecnico di Torino. All the laboratory, the reviews which I did to my classmates works and the ones they did to me and the project for the final exam.

all the code was written by me, except for the one provided by the professor during the course, on which I started to implement my works.

All the code is also available on my github: <https://github.com/EnricoMagliano/computational-intelligence>

## 2 Final Project - Quarto

For the final project I have implemented different strategies for creating an agent able to play the Quarto game ([https://en.wikipedia.org/wiki/Quarto\\_\(board\\_game\)](https://en.wikipedia.org/wiki/Quarto_(board_game))) and analysing the results to understand which is the best strategy.

All the agents are an implementations of the generic class `quarto.Player` (provided by the professor), I have overridden the two methods `choose_piece(self)`, that return the index of the piece choose by my agent for the opponent and `place_piece(self)`, that return the coordinates of the position where my agent place piece on the board.

For the evaluation all agent are tested against the random player and the other agents developed by me.

### 2.1 Random Player

Basic random Player (provided by the professor), that select randomly a piece or a place.

```
1 class RandomPlayer(quarto.Player):
2
3     def __init__(self, quarto: quarto.Quarto) -> None:
4         super().__init__(quarto)
5
6     def choose_piece(self) -> int:
7         return random.randint(0, 15)
8
9     def place_piece(self) -> tuple[int, int]:
10        return random.randint(0, 3), random.randint(0, 3)
```

Listing 1: Random Player

## 2.2 My MinMax Player

My MinMax Player is a mix of an hard coded strategy and MinMax strategy. Practically instead of looking for the entire tree, my strategy works on the opportunity that represent a possibility win, if properly exploited.

The opportunity are considered with 4 different level:

- fourth level for the ones made by an empty row or column in the board.
- third level for the ones made by a row or column with only one element.
- second level for the ones made by a row or column with two element with at least one characteristic in common.
- first level for the ones made by a row or column with three element with at least one characteristic in common.

The MinMax idea consists in taking account if is my turn to place or choose the piece, and minimize the probability that an opportunity bring my opponent to win, and maximize the chance that an opportunity bring me to win. For example if I have to choose the piece for my opponent I'll check the second and the fourth level that are best for my, instead of the first and the third that are best for my opponent. While if I have to place the piece, I'll check the first and the third that in this case are the best for my.

This result in a really effective strategy, in 10000 match against the random it win the 98.64% of the games (using the block strategy). It is also computationally very fast compared to a classic MinMax, because instead of exploring the entire tree this strategy just focused on the opportunity, that are a sort of subset of all possibility (like a sort of pruning).

```
1 class MyMinMax(quarto.Player):
2     '''My MinMax strategy'''
3
4     def __init__(self, quarto: quarto.Quarto) -> None:
5         super().__init__(quarto)
6         self.opportunity = {}          #dict key=opportunity level,
7                                     value= list of tuple of a list of position tuple and int that
8                                     is charachteristics
9
10    def choose_piece(self) -> int:
11        utilities.check_opportunity(self)
12        #print(self.opportunity)
13
14        negative_char = []
15        positive_char = []
16        for e1 in self.opportunity[1]: #take opportunity level 1 (
17            worse for me)
```

```

15         if e1[1] not in negative_char:
16             negative_char.append(e1[1])
17         for e3 in self.opportunity[3]: #take opportunity level 3
18             if e3[1] not in negative_char:
19                 negative_char.append(e3[1])
20         for e2 in self.opportunity[2]: #take opportunity level 2 (
best for me)
21             if e2[1] not in positive_char:
22                 positive_char.append(e2[1])
23             #print("pos ", positive_char)
24             #print("neg ", negative_char)
25
26         positive_char = [x for x in positive_char if x not in
negative_char] #take only positive char that are not in
negative char
27
28         piece_index = utilities.find_piece(self, positive_char,
negative_char)
29         if piece_index != -1:
30             #print("selected piece ", piece_index)
31             return piece_index
32         else:
33             for e4 in self.opportunity[4]: #add level 4 in positive
char
34                 positive_char.append(e4[1])
35
36         positive_char = [x for x in positive_char if x not in
negative_char]
37
38         piece_index = utilities.find_piece(self, positive_char,
negative_char)
39         if piece_index != -1:
40             #print("selected piece ", piece_index)
41             return piece_index
42         else:
43             positive_char = range(8) #take all char
44             negative_char = [] #reset negative for taking only
level 1
45             for e1 in self.opportunity[1]: #take opportunity
level 1 (worse for me)
46                 if e1[1] not in negative_char:
47                     negative_char.append(e1[1])
48             positive_char = [x for x in positive_char if x not
in negative_char]
49
50             piece_index = utilities.find_piece(self,
positive_char, negative_char)
51             if piece_index != -1:
52                 #print("selected piece ", piece_index)
53                 return piece_index
54
55
56
57         return random.randint(0, 15)
58
59     def place_piece(self) -> tuple[int, int]: #index are inverted
#compute opportunity
60

```

```

61     utilities.check_opportunity(self)
62     #print(self.opportunity)
63
64     #take selected piece
65     piece_index = self.get_game().get_selected_piece()
66     piece = self.get_game().get_piece_characteristics(
piece_index)
67     #print("index piece ", piece_index)
68     piece_char = []
69
70     #take piece char
71     if piece.HIGH == True:
72         piece_char.append(0)
73     else:
74         piece_char.append(4)
75     if piece.COLOURED == True:
76         piece_char.append(1)
77     else:
78         piece_char.append(5)
79     if piece.SOLID == True:
80         piece_char.append(2)
81     else:
82         piece_char.append(6)
83     if piece.SQUARE == True:
84         piece_char.append(3)
85     else:
86         piece_char.append(7)
87     #print("piece char ", piece_char)
88
89
90     positive_op = []
91     for e1 in self.opportunity[1]: #take opportunity level 1 (
best for me)
92         if e1 not in positive_op:
93             positive_op.append(e1)
94     #print(positive_op)
95     #loop over level 1 opportunity until found one with char of
selected piece
96     for op in positive_op:
97         if op[1] in piece_char:
98             return op[0][0][1], op[0][0][0]
99
100     #check if need a block and try to block
101     if len(positive_op) > 0:
102         move = utilities.block_next(self, piece_index)
103         if move != None:
104             return move
105
106     positive_op = []
107     for e3 in self.opportunity[3]: #take opportunity level 3 (
good for me)
108         if e3 not in positive_op:
109             positive_op.append(e3)
110     negative_op_place = []
111     for e2 in self.opportunity[2]: #take opportunity level 2 (
good for my opponent)
112         for e2_place in e2[0]: #take only the places not the

```

```

113     tuple (list_of_place, char)
114         if e2_place not in negative_op_place:
115             negative_op_place.append(e2_place)
116     #loop over positive opportunity (13) checking if match with
117     piece char
118     for op in positive_op:
119         if op[1] in piece_char:
120             for place in op[0]: #loop over opportunity places
121                 if place not in negative_op_place: #check if
122                     place isn't also a place of opportunity of l2
123                     return place[1], place[0]
124
125     #loop over free place
126     board = self.get_game().get_board_status()
127     for i in range(4):
128         for j in range(4):
129             if board[i][j] == -1 and (i,j) not in
130             negative_op_place: #check if free place isn't a negative
131             opportunity l2
132                 return j, i
133
134     #loop over free place
135     for i in range(4):
136         for j in range(4):
137             if board[i][j] == -1:
138                 return j, i #return first free place found

```

Listing 2: My MinMax Player

## 2.3 Pastimes Player

Pastimes player is based on an idea found on <https://ourpastimes.com/win-quarto-2325455.html>.

This basic idea is to choose a piece characteristic (attribute) and hand your opponent lots of pieces of the same attribute until he places two in the same line. Count the number of pieces remaining that are not in your chosen attribute. Hand your opponent another piece of the chosen attribute if the number of remaining opposite pieces is even, hand her a black piece instead. Continue handing your opponent off-attribute pieces once he has made three in a row of your chosen attribute.

I have revised this idea to keeping track of all piece attributes remains free and opportunity on the board, this basically generalize the basic idea.

This is all implemented in the `choose_piece(self)` method, while the `place_piece(self)` method is more or less the same implemented in the my MinMax Player.

The results are really good and close to the myMinMax player, also due to the fact that they have very similar `place_piece` method, in 10000 match against the random it win the 98.35% of the games (using the block strategy). While against myMinMax the winning rate is closed to the 50%.

```
1 class MyPastimes(quarto.Player):
2     '''Generalization of ourpastimes.com strategy'''
3     def __init__(self, quarto: quarto.Quarto) -> None:
4         super().__init__(quarto)
5         self.opportunity = {} #dict key=opportunity level, value=
6         #list of tuple of a list of position tuple and int that is
7         #characteristics
8         self.pieces = {} #dict: key= piece index, value = list of
9         #char
10
11     def choose_piece(self) -> int:
12         #compute opportunity
13         utilities.check_opportunity(self)
14         #print(self.opportunity)
15         #take free pieces
16         self.pieces = utilities.free_pieces(self)
17
18         #save all free piece with l1 char
19         returnable_piece = {} #piece without char in l1
20         for index, piece_char in self.pieces.items():
21             not_in = True
22             for l1 in self.opportunity[1]: #loop over l1 opp
23                 if l1[1] in piece_char:
24                     not_in = False
25             if not_in:
```



```

23         returnable_piece[index] = piece_char #add piece
24         without l1 char
25
26         cont_senza_char = {} #dict where key = char and value are
27         free piece without char
28         for l2 in self.opportunity[2]:
29             cont_senza_char[l2[1]] = sum(1 for p in self.pieces.
30             values() if l2[1] not in p)
31             #for each piece cont favor +1 and sfavor as -1
32             score = {}
33             for p_index, p_char in returnable_piece.items():
34                 score[p_index] = 0
35                 for c, n in cont_senza_char.items():
36                     if n%2 == 0: #if even char in piece is a favor
37                         if c in p_char:
38                             score[p_index] += 1
39                         else:
40                             score[p_index] -= 1
41                     else:
42                         #if odd nor char in piece is a favor
43                         if c not in p_char:
44                             score[p_index] += 1
45                         else:
46                             score[p_index] -= 1
47                 best_piece_index = None
48                 for p_index, value in score.items():
49                     if best_piece_index == None or best_piece_index < value
50 :
51                     best_piece_index = p_index
52                 if best_piece_index != None:
53                     #print("choose from l2 char index: ", best_piece_index)
54                     #return the best if exists
55
56                 return best_piece_index
57
58         #return from returnable list if it's not empty or choose
59         from free piece
60         if len(returnable_piece) > 0:
61             choose = random.choice(list(returnable_piece.keys()))
62             #print("choose from returnable :) index: ", choose)
63             return choose #return a good piece
64         else:
65             choose = random.choice(list(self.pieces.keys()))
66             #print("choose from free piece :( index: ", choose)
67             return choose #return not a good piece
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

```

74         #print("piece char ", piece_char)
75
76         #consider first l1 opportunities
77         positive_op = []
78         for e1 in self.opportunity[1]: #take opportunity level 1 (
best for me)
79             if e1 not in positive_op:
80                 positive_op.append(e1)
81             #print(positive_op)
82             #loop over level 1 opportunity until found one with char of
selected piece
83             for op in positive_op:
84                 if op[1] in piece_char:
85                     return op[0][0][1], op[0][0][0]
86
87
88             #check if need a block and try to block
89             if len(positive_op) > 0:
90                 move = utilities.block_next(self, piece_index)
91                 if move != None:
92                     return move
93
94
95             #consider l3 opportunity
96             positive_op = []
97             for e3 in self.opportunity[3]: #take opportunity level 3 (
good for me)
98                 if e3 not in positive_op:
99                     positive_op.append(e3)
100             negative_op_place = []
101             for e2 in self.opportunity[2]: #take opportunity level 2 (
good for my opponent)
102                 for e2_place in e2[0]: #take only the places not the
tuple (list_of_place, char)
103                     if e2_place not in negative_op_place:
104                         negative_op_place.append(e2_place)
105             #loop over positive opportunity (l3) checking if match with
piece char
106             for op in positive_op:
107                 if op[1] in piece_char:
108                     for place in op[0]: #loop over opportunity places
109                         if place not in negative_op_place: #check if
place isn't also a place of opportunity of l2
110                             return place[1], place[0]
111
112             #loop over free place
113             board = self.get_game().get_board_status()
114             for i in range(4):
115                 for j in range(4):
116                     if board[i][j] == -1 and (i,j) not in
negative_op_place: #check if free place isn't a negative
opportunity l2
117                         return j, i
118
119             #loop over free place
120             for i in range(4):
121                 for j in range(4):

```

```

122         if board[i][j] == -1:
123             return j, i          #return first free place found
124
125     return random.randint(0, 3), random.randint(0, 3)

```

Listing 3: My Pastimes Player

### 2.3.1 Classic Pastimes strategy

I have also implemented the vanilla version of the Pastimes strategy, the one explained on the site (<https://ourpastimes.com/win-quarto-2325455.html>). On which the agent choose with which attributes play, and keep track only of this attribute.

As expected the result are worse than the generalized pastimes Agent:

- 10000 match against random, 97.95% of winning rate.
- 10000 match against my MinMax, 44.3% of winning rate.
- 10000 match against my pastimes, 44.5% of winning rate.

```

1  class Pastimes(quarto.Player):
2      '''ourpastimes.com strategy'''
3
4      def __init__(self, quarto: quarto.Quarto) -> None:
5          super().__init__(quarto)
6          self.opportunity = {} #dict key=opportunity level, value=
7                                list of tuple of a list of position tuple and int that is
8                                charachteristics
9          self.selected_char = None #select a char
10
11     def reset(self):
12         '''
13         Reset the agent for a new game
14         '''
15
16         self.selected_char = random.randint(0, 7)
17
18     def choose_piece(self) -> int:
19         #print("pastimes choose")
20         #check if a new game is started
21         if np.all(self.get_game().get_board_status() == -1):
22             self.reset()
23
24         #print("sel char ", self.selected_char)
25         opposite_char = self.selected_char-4 if self.selected_char
26         > 3 else self.selected_char+4
27         utilities.check_opportunity(self)
28         #print(self.opportunity)

```

```

28
29     selected_char_l1 = False
30     for e1 in self.opportunity[1]:
31         if e1[1] == self.selected_char:
32             selected_char_l1 = True
33             break
34     selected_char_l2 = False
35     for e2 in self.opportunity[2]:
36         if e2[1] == self.selected_char:
37             selected_char_l2 = True
38             break
39
40     #print("si in l1") if selected_char_l1 else print("no in l1
41 ")
42     #print("si in l2") if selected_char_l2 else print("no in l2
43 ")
44
45     #if there aren't pair (l2) or tripletes (l1) of selected
46     char, return a piece with selected char if doesn't match other
47     l1 char
48     if selected_char_l1 == False and selected_char_l2 == False:
49         pieces_with_selected_char = utilities.select_pieces(
50 self, self.selected_char)
51         #print("piece with selected char ",
52 pieces_with_selected_char)
53
54         returnable = []
55         for p in pieces_with_selected_char:
56             if p not in self.get_game().get_board_status() and
57 utilities.check_l1(self, p) == True:
58                 returnable.append(p)
59             if len(returnable) > 0:
60                 return random.choice(returnable) #return a
61 random element from the returnable ones
62
63
64     #if there are pair(l2) but there aren't tripletes(l1)
65     #check number of element without selected char if even(pari
66 )
67     #return a piece with selected char, otherwise a piece
68     without it
69     if selected_char_l1 == False and selected_char_l2 == True:
70         pieces_with_selected_char = utilities.select_pieces(
71 self, self.selected_char)
72         #print("opposite char ",opposite_char)
73
74         pieces_without_selected_char_tot = utilities.
75 select_pieces(self, opposite_char)
76         pieces_without_selected_char = []
77         for p in pieces_without_selected_char_tot: #filter
78 pieces, take only the ones not already in the board
79             if p not in self.get_game().get_board_status():
80                 pieces_without_selected_char.append(p)
81             #print("piece without sel char: ",
82 pieces_without_selected_char)
83
84         if len(pieces_without_selected_char)%2 == 0: #even

```

```

71     check
72         returnable = []
73         for p in pieces_with_selected_char:
74             if p not in self.get_game().get_board_status()
75 and utilities.check_l1(self, p) == True:
76             returnable.append(p)
77             if len(returnable) > 0:
78                 return random.choice(returnable)
79
80         else:
81             returnable = []
82             for p in pieces_without_selected_char:
83                 if p not in self.get_game().get_board_status()
84 and utilities.check_l1(self, p) == True:
85                 returnable.append(p)
86                 if len(returnable) > 0:
87                     return random.choice(returnable)
88
89         #if there are tripletes of selected char return a piece
90 without selected char and without l1 char
91         returnable = []
92
93         #pieces_without_selected_char = utilities.select_pieces(self
94 , opposite_char) #this worsen the result
95         #for p in pieces_without_selected_char:
96         for p in range(16):
97             if p not in self.get_game().get_board_status() and
98 utilities.check_l1(self, p) == True:
99                 returnable.append(p)
100                 if len(returnable) > 0:
101                     return random.choice(returnable)
102
103         #if there aren't any other possibilities return random
104         return random.randint(0, 15)
105
106
107
108 def place_piece(self) -> tuple[int, int]:
109     #print("pasttimes place")
110     if np.all(self.get_game().get_board_status() == -1):
111         self.reset()
112
113     #compute opportunity
114     utilities.check_opportunity(self)
115     #print(self.opportunity)
116
117     #take selected piece
118     piece_index = self.get_game().get_selected_piece()
119     piece = self.get_game().get_piece_characteristics(
120 piece_index)
121     #print("index piece ", piece_index)
122     piece_char = []
123
124     #take piece char
125     if piece.HIGH == True:
126         piece_char.append(0)

```

```

121         else:
122             piece_char.append(4)
123         if piece.COLOURED == True:
124             piece_char.append(1)
125         else:
126             piece_char.append(5)
127         if piece.SOLID == True:
128             piece_char.append(2)
129         else:
130             piece_char.append(6)
131         if piece.SQUARE == True:
132             piece_char.append(3)
133         else:
134             piece_char.append(7)
135         #print("piece char ", piece_char)
136
137
138         positive_op = []
139         for e1 in self.opportunity[1]: #take opportunity level 1 (
best for me)
140             if e1 not in positive_op:
141                 positive_op.append(e1)
142             #print(positive_op)
143
144         #loop over level 1 opportunity until found one with char of
selected piece
145         for op in positive_op:
146             if op[1] in piece_char:
147                 return op[0][0][1], op[0][0][0]
148
149         #check if need a block and try to block
150         if len(positive_op) > 0:
151             move = utilities.block_next(self, piece_index)
152             if move != None:
153                 return move
154
155
156         positive_op = []
157         for e3 in self.opportunity[3]: #take opportunity level 3 (
good for me)
158             if e3 not in positive_op:
159                 positive_op.append(e3)
160         negative_op_place = []
161         for e2 in self.opportunity[2]: #take opportunity level 2 (
good for my opponent)
162             for e2_place in e2[0]: #take only the places not the
tuple (list_of_place, char)
163                 if e2_place not in negative_op_place:
164                     negative_op_place.append(e2_place)
165         #loop over positive opportunity (l3) checking if match with
piece char
166         for op in positive_op:
167             if op[1] in piece_char:
168                 for place in op[0]: #loop over opportunity places
169                     if place not in negative_op_place: #check if
place isn't also a place of opportunity of l2
170                         return place[1], place[0]

```

```

171
172     #loop over free place
173     board = self.get_game().get_board_status()
174     for i in range(4):
175         for j in range(4):
176             if board[i][j] == -1 and (i,j) not in
negative_op_place: #check if free place isn't a negative
opportunity l2
177                 return j, i
178
179     #loop over free place
180     for i in range(4):
181         for j in range(4):
182             if board[i][j] == -1:
183                 return j, i             #return first free place found

```

Listing 4: Classic Pastimes Strategy

## 2.4 Reinforcing Learning Player

I have also implemented a raw version of a Reinforcement Learning Agent, that during the training phase it learn from the winning matches updating a dictionary, that contains for each board status two move lists one for the piece chosen pieces, the other for the chosen positions, each element with is score. The score is update at the end of every match in this way: +1 if the agent win the match, -1 if the agent lose the match.

During the train phase the random\_factor is decrease to encourage exploitation by moving forward with matches, starting with only exploration, choosing the moves randomly, and finishing exploiting what was previously learned.

Result for this agent:

- 1000 match of training against random, 55% wins against random.
- 10000 match of training against random, 60% wins against random.
- 100000 match of training against random, 57% wins against random.

From this result we can see that our agent is able to learn from the random, increasing the number of training match, the wining rate increase until a threshold (10000 match), after that the overfitting worse the results.

While if I use my MinMax during the training phase, my reinforcement learning agent isn't able to learn, due to the fact that MinMax wins almost always.

- 1000 match of training against MinMax, 47% wins against random.

In general the result are really poor, in particular if compared with the previous Player:

- 1000 match of training against random, 3% wins against MinMax.
- 1000 match of training against MinMax, 4% wins against MinMax.

```
1 NUM_TRAINING_MATCH = 10000
2 NUM_EVAL_MATCH = 100
3
4 class ReinforcementLearning(quarto.Player):
5     '''Reinforcement Learning Agent'''
6
7     def __init__(self, quarto: quarto.Quarto) -> None:
8         super().__init__(quarto)
9         self.learning = False
10        self.knowledge = dict() #dict with board status as key and
                                two list as value one for the score of all place and one for
                                the piece
```



```

11     self.current = dict() #dict for saving place and piece of
the current game
12     self.random_factor = 1 #close to 1 -> more exploration,
close to 0 more exploitation
13     self.num_match = 0
14     self.tot_num_matches = NUM_TRAINING_MATCH
15
16     def set_learning(self, value: bool):
17         '''set RF agent in learning mode if value equal true,
otherwise set it on evaluation mode.'''
18         self.learning = value
19
20     def save_knowledge(self, win):
21         '''save the move made in this game scoring accordingn with
the outcome (win)'''
22
23         #value is a dict that contains choosen piece or place piece
24         for board, value in self.current.items(): #loop over all
the board status in current dict
25
26             if board in self.knowledge: #check if board already
exist in the dict
27
28                 if "choose_piece" in value: #check if value is
choose_piece type
29                     not_in = True
30                     for element in self.knowledge[board]["
choose_piece"]:
31                         if element[0] == value["choose_piece"]: #if
piece already exists for board status in knowledge dict
32                             not_in = False
33                             element[1] += 1 if win else -1 #update
it
34                             if not_in: #otherwise add it and initialize
35                                 self.knowledge[board]["choose_piece"].
append([value["choose_piece"], 1 if win else -1])
36
37                             else: #check if value is place_piece type
38                                 not_in = True
39                                 for element in self.knowledge[board]["
place_piece"]:
40                                     if element[0] == value["place_piece"]: #if
place already exists for board status in knowledge dict
41                                         not_in = False
42                                         element[1] += 1 if win else -1 #update
it
43                                         if not_in: #otherwise add it and initialize
44                                             self.knowledge[board]["place_piece"].append
([value["place_piece"], 1 if win else -1])
45
46                                     else: #if board status isn't already in the knowledge
dict add it creating the 2 list, one for piece scoring the
other for place scoring
47                                         self.knowledge[board] = {"choose_piece": list(), "
place_piece": list()}
48                                         if "choose_piece" in value: #and add piece or place
in the right list, with score according to the outcomes

```

```

49         self.knowledge[board]["choose_piece"].append([
value["choose_piece"], 1 if win else -1])
50         else:
51             self.knowledge[board]["place_piece"].append([
value["place_piece"], 1 if win else -1])
52
53         self.num_match +=1 #update number of played matches
54         self.random_factor = 1- 2*(self.num_match/self.
tot_num_matches) #update random factor for encrease the
exploitation with match prograssion
55         self.current = dict() #reset the current dict
56
57     def choose_piece(self) -> int:
58         board = self.get_game().get_board_status()
59         free_pieces = list(utilities.free_pieces(self).keys())
60         choose = random.choice(free_pieces)
61
62         #if random_factor > random(0,1) -> exploration otherwise
exploitation
63         if self.learning and random.random() < self.random_factor:
#if agent is set in learning return a random piece from the
free ones
64             self.current[np.array2string(board)] = {"choose_piece":
choose} #and save the choose in the current dict
65             return choose
66
67         else: #if agent is in eval mode select the piece with the
highest score (>0) if it exists
68             best = None
69             if np.array2string(board) in self.knowledge:
#piece_score is a tuple (piece_index, score)
70                 for piece_score in self.knowledge[np.array2string(
board)]["choose_piece"]:
71                     if best == None or best[1] < piece_score[1]:
72                         best = piece_score
73                     if best != None and best[1] > 0: #check if exists a
piece with score greater than 0
74                         return best[0]
75             return choose #if not exist return a random piece
76
77     def place_piece(self) -> tuple[int, int]:
78
79         board = self.get_game().get_board_status()
80         free_place = utilities.free_place(self)
81         choose = random.choice(free_place)
82
83         #if random_factor > random(0,1) -> exploration otherwise
exploitation
84         if self.learning and random.random() < self.random_factor:
#if agent is set in learning and exploration return a random
place from the free ones
85             self.current[np.array2string(board)] = {"place_piece":
choose} #and save the choose in the current dict
86             return choose[1], choose[0]
87
88         else: #if agent is in eval (or exploitation) mode select
the place with the highest score (>0) if it exists
89

```

```

90         best = None
91         if np.array2string(board) in self.knowledge:
92             #place_score is a tuple (place_tuple, score)
93             for place_score in self.knowledge[np.array2string(
board)]:
94                 if best == None or best[1] < place_score[1]:
95                     best = place_score
96                 if best != None and best[1] > 0: #check if exists a
place with score greater than 0
97                     return best[0][1], best[0][0]
98                 return choose[1], choose[0] #if not exist return a
random place from the free ones
99
100 #
101 # TRAINING STUFF
102 #
103
104 def play_n_game_train(game: quarto.Quarto, RF:
ReinforcementLearning, player2: quarto.Player, n: int):
105     '''
106     Play n games for training player1 (Reinforcement Learning)
against player2, print the winner ratio of player1 over player2
, switching the starter at each game
107     '''
108
109     win_count = 0
110     last_start = 1
111     for i in range(n):
112         game.reset()
113
114         if last_start == 1:
115             game.set_players((RF, player2))
116             last_start = 0
117         else:
118             game.set_players((player2, RF))
119             last_start = 1
120
121         winner = game.run()
122
123         if (winner == 0 and last_start == 0) or (winner == 1 and
last_start == 1): #player1 win
124             win_count+=1
125             RF.save_knowledge(True)
126         else:
127             RF.save_knowledge(False)
128
129         logging.warning(f"main: Winner ratio of RF during training: {
win_count/n}")
130
131 def play_n_game(game: quarto.Quarto, RF: ReinforcementLearning,
player2: quarto.Player, n: int):
132     '''
133     Play n games player1 (Reinforcement Learning) against player2,
print the winner ratio of player1 over player2, switching the
starter at each game
134     '''
135

```

```

136 win_count = 0
137 last_start = 1
138 for i in range(n):
139     game.reset()
140
141     if last_start == 1:
142         game.set_players((RF, player2))
143         last_start = 0
144     else:
145         game.set_players((player2, RF))
146         last_start = 1
147
148     winner = game.run()
149
150     if (winner == 0 and last_start == 0) or (winner == 1 and
151 last_start == 1): #player1 win
152         win_count+=1
153
154 logging.warning(f"main: Winner ratio of RF evaluation training:
155 {win_count/n}")
156
157 def training():
158     '''
159     training the RF agent and evaluate it
160     '''
161     game = quarto.Quarto()
162     agentReinLear = ReinforcementLearning(game)
163     agentReinLear.set_learning(True)
164     play_n_game_train(game, agentReinLear, main.RandomPlayer(game),
165 NUM_TRAINING_MATCH)
166     agentReinLear.set_learning(False)
167     play_n_game(game, agentReinLear, main.RandomPlayer(game),
168 NUM_EVAL_MATCH)

```

Listing 5: Reinforcement Learning Player

## 2.5 Genetic Algorithm Player

The last approach that I have try is an Genetic algorithm agent, for evolving an hard coded strategy, similar to MinMax. where the genome is made by four gene:

- First gene select level of the consider opportunity for choosing piece, more high means more high level.
- Second gene is for choosing piece, if higher than 0.5 try to destroy the opportunity otherwise try to use it.
- Third gene select level of the consider opportunity for placing piece, more high means more high level.
- Fourth gene is for placing piece, if higher than 0.5 try to destroy the opportunity otherwise try to use it.

This algorithm works for 10 generation on which the offsprings are made by mutations, this solution is based on the code developed for the second laboratory.

In this case the result are really interesting, not for the effectiveness of the player, his performance is still above the MinMax and pastimes hard coded strategy, but because the best genome that is founds bring the GA agent to use hard coded rules really similar to the ones used in the MinMax. This means that my MinMax implementation are more close to the optimal strategy.

For example this genome: (0.140, 0.619, 0.007, 0.263), that is one of the best and has a winner ratio of GA: 97.8%, is trying to block the level one opportunity when I'm choosing the piece for my opponent and use the level one opportunity when I'm placing the piece, very similar to idea under the minmax. Some other genome, like this: (0.427, 0.859, 0.162, 0.359), are quite good, with a winning ratio of 85.5%.

In any case hard coded rules find by GA Player has good results against random, but has poor result agaist my MinMax, this best genome (0.140, 0.619, 0.007, 0.263) against my MinMax has a winning ratio of 27.6%.

```
1 class GeneticAlgorithm(quarto.Player):
2     '''Genetic Algorithm agent'''
3
4     def __init__(self, quarto: quarto.Quarto) -> None:
5         super().__init__(quarto)
6         self.genome = None           #genome of the current GA agent
7         self.opportunity = {}        #dict key=opportunity level, value=
                                     #list of tuple of a list of position tuple and int that is
                                     #characteristics
8
9     def set_genome(self, genome):
```

```

10     '''set the genome for the current GA agent'''
11     self.genome = genome
12
13     def choose_piece(self) -> int:
14         utilities.check_opportunity(self)
15
16         level = math.ceil(self.genome[0]*4) #level of opportunity
17         take in consideration
18         chars = {} #dict with key the char and value the
19         number of time this char appear on selected value
20         for _, char in self.opportunity[level]:
21             if char not in chars:
22                 chars[char] = 1
23             else:
24                 chars[char] += 1
25
26         free_piece = utilities.free_pieces(self)
27         best_fit = None
28         worse_fit = None
29         for p_index, p_char in free_piece.items(): #for each free
30             piece
31             score = 0 #compute a
32             score as +1 if piece char is in selected level char
33             for char in p_char:
34                 score += chars[char] if char in chars else 0
35             if worse_fit == None or worse_fit[1] > score:
36                 worse_fit = (p_index, score) #save the worse
37             if best_fit == None or best_fit[1] < score:
38                 best_fit = (p_index, score) #save the best
39
40             if self.genome[1] < 0.5: #return best char if exist
41                 if best_fit != None:
42                     return best_fit[0]
43             else: #return worse char if exist
44                 if worse_fit != None:
45                     return worse_fit[0]
46
47             #otherwise return random piece
48             return random.randint(0, 15)
49
50     def place_piece(self) -> tuple[int, int]:
51         utilities.check_opportunity(self)
52
53         #take selected piece
54         piece_index = self.get_game().get_selected_piece()
55         piece_char = utilities.get_pieces_char(self, piece_index)
56         level = math.ceil(self.genome[2]*4) #level of opportunity
57         take in consideration
58
59         position_pos = [] #list of place
60         with char in the selected piece
61         position_neg = utilities.free_place(self) #list of palce
62         without char of selected piece
63         for pos, char in self.opportunity[level]:
64             for p in pos: #for each place in opportunity of
65                 selected level
66                 if char in piece_char:

```

```

59         position_pos.append(p) #append place in
        positive if char match with selected piece
60         if p in position_neg:
61             position_neg.remove(p) #remove from
        negative if char not match with selected piece
62
63         if self.genome[3] < 0.5: #return one of the
        positive if exist
64             if len(position_pos) > 0:
65                 choice = random.choice(position_pos)
66                 return choice[1], choice[0]
67             else: #return one of the
        negative if not exist
68                 if len(position_neg) > 0:
69                     choice = random.choice(position_neg)
70                     return choice[1], choice[0]
71
72         #otherwise return a random place
73         return random.randint(0, 3), random.randint(0, 3)
74
75
76
77 '''
78 GENOME MEANING:
79     genome[0] = select level of the consider opportunity for choose
        piece, more high -> more high level
80     genome[1] = for choose piece, if higher than 0.5 try to destroy
        the opportunity otherwise try to use it
81     genome[2] = select level of the consider opportunity for choose
        place piece, more high -> more high level
82     genome[3] = for place piece, if higher than 0.5 try to destroy
        the opportunity otherwise try to use it
83 '''
84
85 NUM_TRAINING_MATCH = 10 #to compute fitness
86 NUM_EVAL_MATCH = 1000 #to evaluate the best genome
87 NUM_GENERATION = 10
88 POPULATION_SIZE = 100
89 OFFSPRING_SIZE = 10
90
91 def fitness(genome):
92     '''Compute the fitness of the genome as the winning ratio
        agaisnt random player'''
93     game = quarto.Quarto()
94     agent = GeneticAlgorithm(game)
95     agent.set_genome(genome)
96     opponent = main.RandomPlayer(game)
97     return play_n_game(game, agent, opponent, NUM_TRAINING_MATCH)
98
99 def generatePopulation():
100     '''Return list of population, one individual is a tuple of
        genome (list of genes) and his fitness'''
101     population = list()
102     for genome in range(POPULATION_SIZE):
103         genome = (random.random(), random.random(), random.random(),
            random.random())
104         population.append((genome, fitness(genome)))

```

```

105     return population
106
107 def mutation(g):
108     '''Mutation change randomly a random gene in the genome'''
109     point = random.randint(0, len(g)-1)
110     return g[:point] + (random.random(),) + g[point+1:]
111
112 def select_parent(population, tournament_size=10):
113     '''select parent using a k = 10 tournament, taking the one
114     with best fitness'''
115     return max(random.choices(population, k=tournament_size), key=
116                 lambda i: i[1])
117
118 def GA():
119     '''Run 10 generations to find the best genome'''
120     best_sol = None
121
122     population = generatePopulation()
123     for generation in range(10):
124         offsprings = list()
125         for i in range(OFFSPRING_SIZE): #in each generation create
126             OFFSPRING_SIZE offspring by mutation parents
127             o = ()
128             p = select_parent(population)
129             o = mutation(p[0])
130             offsprings.append((o, fitness(o)))
131             population = population + offsprings
132             population = sorted(population, key=lambda i: i[1], reverse=
133                                 True)[:POPULATION_SIZE]
134
135     best_sol = population[0][0]
136     return best_sol
137
138 def play_n_game(game: quarto.Quarto, GA: GeneticAlgorithm, player2:
139                 quarto.Player, n: int):
140     '''
141     Play n games player1 (Genetic Algorithm) against player2, print
142     the winner ratio of player1 over player2, switching the
143     starter at each game
144     '''
145
146     win_count = 0
147     last_start = 1
148     for i in range(n):
149         game.reset()
150
151         if last_start == 1:
152             game.set_players((GA, player2))
153             last_start = 0
154         else:
155             game.set_players((player2, GA))
156             last_start = 1
157
158     winner = game.run()
159
160     if (winner == 0 and last_start == 0) or (winner == 1 and

```



```

155     last_start == 1): #player1 win
156         win_count+=1
157
158     #logging.warning(f"main: Winner ratio of GA evaluation training
159     : {win_count/n}")
160     return win_count/n
161
162 def training():
163     '''
164     training the GA agent and evaluate it
165     '''
166     best_genome = GA()
167     game = quarto.Quarto()
168     agentGen = GeneticAlgorithm(game)
169     agentGen.set_genome(best_genome)
170     result = play_n_game(game, agentGen, main.RandomPlayer(game),
171     NUM_EVAL_MATCH)
172     print(f"main: Winner ratio of GA: {result}, with genome: {
173     best_genome}")

```

Listing 6: Genetic Algorithm Player

## 2.6 Block Strategy

The Block strategy, already mentioned for MinMax and pastimes Player is a side strategy applicable as extension on `place_piece(self)` for hard coded strategy.

Basically is consists in try to place the piece to block a possible win of out opponent in his next turn. Practically, after check if I can't win placing the selected piece, the block strategy check if all remain piece can fit in a level 1 opportunity (the ones with 3 pieces with at least one common attribure), and so bring my opponent to the victory, in this case the block strategy place the selected piece in a level 1 opportunity to allow the `choose_piece(self)` method to select a piece that doesn't fit in any level 1 opportunity.

This extension increase a little bit the performance of the minMax and pastimes hard code strategies:

- my MinMax against random:
  - 10000 match without block strategy = 98.11% winning rate
  - 10000 match with block strategy = 98.64% winning rate
- my pastimes against random:
  - 10000 match without block strategy = 97.95% winning rate
  - 10000 match with block strategy = 98.35% winning rate

```
1 def block_next(self, sel_piece_index) -> tuple[int, int]:
2     '''
3     Check if next turn, I have to choose a piece that let my
4     opponent win.
5     In this case return a position when place piece to block
6     the winning.
7     Otherwise return None
8     '''
9     sel_piece = self.get_game().get_piece_characteristics(
10    sel_piece_index)
11
12    positive_char_opponent = {} #dict where key is l1 char and
13    value the number of place
14    for e1 in self.opportunity[1]:
15        if e1[1] not in positive_char_opponent:
16            positive_char_opponent[e1[1]] = 1
17        else:
18            positive_char_opponent[e1[1]] += 1
19
20    #print("in block", positive_char_opponent)
21
22    #take all piece indexes not already placed in the board
23    free_pieces = list(range(16))
24    free_pieces.remove(sel_piece_index) #remove selected piece
```

```

21         for r in self.get_game().get_board_status():
22             for p in r:
23                 if p != -1:
24                     free_pieces.remove(p)
25
26             for p in free_pieces:
27                 match = False
28                 for c in positive_char_opponent:
29                     if c == 0:
30                         if self.get_game().get_piece_characteristics(p
21 ).HIGH == True:
31                             match = True
32                         elif c == 1:
33                             if self.get_game().get_piece_characteristics(p
21 ).COLOURED == True:
34                             match = True
35                         elif c == 2:
36                             if self.get_game().get_piece_characteristics(p
21 ).SOLID == True:
37                             match = True
38                         elif c == 3:
39                             if self.get_game().get_piece_characteristics(p
21 ).SQUARE == True:
40                             match = True
41                         elif c == 4:
42                             if self.get_game().get_piece_characteristics(p
21 ).HIGH == False:
43                             match = True
44                         elif c == 5:
45                             if self.get_game().get_piece_characteristics(p
21 ).COLOURED == False:
46                             match = True
47                         elif c == 6:
48                             if self.get_game().get_piece_characteristics(p
21 ).SOLID == False:
49                             match = True
50                         elif c == 7:
51                             if self.get_game().get_piece_characteristics(p
21 ).SQUARE == False:
52                             match = True
53
54                 if match == False: #find a piece that doesn't match
55                     #print("find piece not match ", p)
56                     return None #no need block, find a piece without
char in l1
57
58             #search char with one place
59             blockable_char = []
60             for c in positive_char_opponent:
61                 if positive_char_opponent[c] == 1: #try to block char c
if have one place
62                     blockable_char.append(c)
63                     #print("blockable char ", blockable_char)
64
65
66
67             for b_c in blockable_char:

```

```

68         for l1 in self.opportunity[1]:
69             if l1[1] == b_c:
70                 place = l1[0][0]
71                 if simulation(self, place, sel_piece_index,
free_pieces):
72                     return place[1], place[0]
73         return None #if there aren't any single place for one char
-> unblockable -> return None

```

Listing 7: Block Strategy

## 2.7 Conclusion

At the end I can say that for this task the best solution is an hard coded strategy, like my MinMax and my pastimes. In particular my MinMax, using the block strategy is the best (the name is due to the fact that is inspired to minMax).

## 2.8 Utilities

In this section there are the common functions used in my project by the strategy, in the file utilities.py

```
1 import logging
2 import argparse
3 import random
4 import quarto
5 import numpy as np
6 import operator as op
7
8 def get_pieces_char(agent, index):
9     '''
10     Return array of the characteristic of the index piece
11     '''
12     piece = agent.get_game().get_piece_characteristics(index)
13     #print("index piece ", piece_index)
14     piece_char = []
15
16     #take piece char
17     if piece.HIGH == True:
18         piece_char.append(0)
19     else:
20         piece_char.append(4)
21     if piece.COLOURED == True:
22         piece_char.append(1)
23     else:
24         piece_char.append(5)
25     if piece.SOLID == True:
26         piece_char.append(2)
27     else:
28         piece_char.append(6)
29     if piece.SQUARE == True:
30         piece_char.append(3)
31     else:
32         piece_char.append(7)
33
34     #print("piece char ", piece_char)
35     return piece_char
36
37
38 def char_l1(self):
39     '''
40     return a list of l1 char
41     '''
42     array = list()
43     for l1 in self.opportunity[1]:
44         if l1[1] not in array:
45             array.append(l1[1])
46     return array
47
48 def free_pieces(agent):
49     '''
50     Return a dict of free piece indexes as value and array of
```

```

51     charateristic as value
52     '''
53     board = agent.get_game().get_board_status()
54     pieces = dict()
55     for p in range(16):
56         if p not in board:
57             pieces[p] = get_pieces_char(agent, p)
58
59     return pieces
60
61 def free_place(agent):
62     '''
63     Return a list tuple of free place in the board
64     '''
65     board = agent.get_game().get_board_status()
66     free_place = []
67     for i in range(4):
68         for j in range(4):
69             if board[i][j] == -1:
70                 free_place.append((i, j))
71     return free_place
72
73 def block_next(self, sel_piece_index) -> tuple[int, int]:
74     '''
75     Check if next turn, I have to choose a piece that let my
76     opponent win.
77     In this case return a position when place piece to block
78     the winning.
79     Otherwise return None
80     '''
81     sel_piece = self.get_game().get_piece_characteristics(
82         sel_piece_index)
83
84     positive_char_opponent = {} #dict where key is l1 char and
85     value the number of place
86     for e1 in self.opportunity[1]:
87         if e1[1] not in positive_char_opponent:
88             positive_char_opponent[e1[1]] = 1
89         else:
90             positive_char_opponent[e1[1]] += 1
91
92     #print("in block", positive_char_opponent)
93
94     #take all piece indexes not already placed in the board
95     free_pieces = list(range(16))
96     free_pieces.remove(sel_piece_index) #remove selected piece
97     for r in self.get_game().get_board_status():
98         for p in r:
99             if p != -1:
100                 free_pieces.remove(p)
101
102     for p in free_pieces:
103         match = False
104         for c in positive_char_opponent:
105             if c == 0:
106                 if self.get_game().get_piece_characteristics(p
107 ).HIGH == True:

```

```

102         match = True
103     elif c == 1:
104         if self.get_game().get_piece_characteristics(p
105 ).COLOURED == True:
106             match = True
107     elif c == 2:
108         if self.get_game().get_piece_characteristics(p
109 ).SOLID == True:
110             match = True
111     elif c == 3:
112         if self.get_game().get_piece_characteristics(p
113 ).SQUARE == True:
114             match = True
115     elif c == 4:
116         if self.get_game().get_piece_characteristics(p
117 ).HIGH == False:
118             match = True
119     elif c == 5:
120         if self.get_game().get_piece_characteristics(p
121 ).COLOURED == False:
122             match = True
123     elif c == 6:
124         if self.get_game().get_piece_characteristics(p
125 ).SOLID == False:
126             match = True
127     elif c == 7:
128         if self.get_game().get_piece_characteristics(p
129 ).SQUARE == False:
130             match = True
131
132     if match == False: #find a piece that doesn't match
133         #print("find piece not match ", p)
134         return None #no need block, find a piece without
135 char in l1
136
137     #search char with one place
138     blockable_char = []
139     for c in positive_char_opponent:
140         if positive_char_opponent[c] == 1: #try to block char c
141             if have one place
142                 blockable_char.append(c)
143             #print("blockable char ", blockable_char)
144
145
146     for b_c in blockable_char:
147         for l1 in self.opportunity[1]:
148             if l1[1] == b_c:
149                 place = l1[0][0]
150                 if simulation(self, place, sel_piece_index,
151 free_pieces):
152                     return place[1], place[0]
153         return None #if there aren't any single place for one char
154     -> unblockable -> return None
155
156     '''
157     for c in blockable_char:

```

```

148         place = None
149         not_in_l2 = True
150         for e1 in self.opportunity[1]:
151             if e1[1] == c:
152                 place = e1[0][0]
153         for e2 in self.opportunity[2]:
154             for place_2 in e2[0]:
155                 if place_2 == place:
156                     not_in_l2 = False
157         if not_in_l2:
158             #print("not in l2, ", place)
159             return place[1], place[0]
160
161         if len(blockable_char) > 0:
162             place = None
163             random_choose = random.choice(blockable_char)
164             for e1 in self.opportunity[1]:
165                 if e1[1] == random_choose:
166                     place = e1[0][0]
167             #print("place in l2, ", place)
168             return place[1], place[0]
169         return None #if there aren't any single place for one char
170         -> unblockable -> return None
171         '''
172
173     def simulation(self, place, piece_index, free_piece):
174         '''
175         Simulate placing the selected piece in the selected place.
176         If this doesn't compromise nothing return true, otherwise
177         return False.
178         '''
179         for l2 in self.opportunity[2]:
180             if place in l2[0] and l2[1] in get_pieces_char(self,
181                 piece_index):
182                 piece_no_match = False
183                 for p in free_piece:
184                     if l2[1] not in get_pieces_char(self, p):
185                         piece_no_match = True
186                 if piece_no_match == False:
187                     return False
188         return True
189
190     def check_l1(self, piece) -> bool:
191         '''
192         Return true if piece doesn't have characteristic in l1,
193         otherwise return false
194         '''
195         #print("in check l1")
196         l1 = []
197         for e1 in self.opportunity[1]:
198             if e1[1] not in l1:
199                 l1.append(e1[1])
200         if self.get_game().get_piece_characteristics(piece).HIGH ==
201             True:
202             if 0 in l1:
203                 return False
204         else:

```



```

200         if 4 in l1:
201             return False
202     if self.get_game().get_piece_characteristics(piece).COLOURED
== True:
203         if 1 in l1:
204             return False
205     else:
206         if 5 in l1:
207             return False
208     if self.get_game().get_piece_characteristics(piece).SOLID ==
True:
209         if 2 in l1:
210             return False
211     else:
212         if 6 in l1:
213             return False
214     if self.get_game().get_piece_characteristics(piece).SQUARE ==
True:
215         if 3 in l1:
216             return False
217     else:
218         if 7 in l1:
219             return False
220     return True
221
222 def select_pieces(self, char):
223     '''
224     return a list of all index of piece with char
225     '''
226     select_pieces = []
227     for i in range(16):
228         if char == 0:
229             if self.get_game().get_piece_characteristics(i).HIGH
== True:
230                 select_pieces.append(i)
231             elif char == 1:
232                 if self.get_game().get_piece_characteristics(i).
COLOURED == True:
233                     select_pieces.append(i)
234             elif char == 2:
235                 if self.get_game().get_piece_characteristics(i).SOLID
== True:
236                     select_pieces.append(i)
237             elif char == 3:
238                 if self.get_game().get_piece_characteristics(i).SQUARE
== True:
239                     select_pieces.append(i)
240             elif char == 4:
241                 if self.get_game().get_piece_characteristics(i).HIGH
== False:
242                     select_pieces.append(i)
243             elif char == 5:
244                 if self.get_game().get_piece_characteristics(i).
COLOURED == False:
245                     select_pieces.append(i)
246             elif char == 6:
247                 if self.get_game().get_piece_characteristics(i).SOLID

```

```

248 == False:
249     select_pieces.append(i)
250     elif char == 7:
251         if self.get_game().get_piece_characteristics(i).SQUARE
252         == False:
253             select_pieces.append(i)
254
255 return select_pieces
256
257 def find_piece(self, positive_char, negative_char) -> int:
258     '''
259     Return the index of a piece that satisfies positive char and
260     doesn't have negative char
261     Return -1 if there aren't pieces like that
262     '''
263
264     # take all pieces not in board
265     pieces_not_in_board = [x for x in range(16) if x not in self.
266     get_game().get_board_status()]
267     piesces_match_char = []
268     for i in pieces_not_in_board: #take pieces that have at least
269     one positive char
270         for c in positive_char:
271             if c == 0:
272                 if self.get_game().get_piece_characteristics(i).
273                 HIGH == True and i not in piesces_match_char:
274                     piesces_match_char.append(i)
275             elif c == 1:
276                 if self.get_game().get_piece_characteristics(i).
277                 COLOURED == True and i not in piesces_match_char:
278                     piesces_match_char.append(i)
279             elif c == 2:
280                 if self.get_game().get_piece_characteristics(i).
281                 SOLID == True and i not in piesces_match_char:
282                     piesces_match_char.append(i)
283             elif c == 3:
284                 if self.get_game().get_piece_characteristics(i).
285                 SQUARE == True and i not in piesces_match_char:
286                     piesces_match_char.append(i)
287             elif c == 4:
288                 if self.get_game().get_piece_characteristics(i).
289                 HIGH == False and i not in piesces_match_char:
290                     piesces_match_char.append(i)
291             elif c == 5:
292                 if self.get_game().get_piece_characteristics(i).
293                 COLOURED == False and i not in piesces_match_char:
294                     piesces_match_char.append(i)
295             elif c == 6:
296                 if self.get_game().get_piece_characteristics(i).
297                 SOLID == False and i not in piesces_match_char:
298                     piesces_match_char.append(i)
299             elif c == 7:
300                 if self.get_game().get_piece_characteristics(i).
301                 SQUARE == False and i not in piesces_match_char:
302                     piesces_match_char.append(i)
303         for c in negative_char:
304             if c == 0:

```

```

292         if self.get_game().get_piece_characteristics(i).
HIGH == True and i in piesces_match_char:
293             piesces_match_char.remove(i)
294         elif c == 1:
295             if self.get_game().get_piece_characteristics(i).
COLOURED == True and i in piesces_match_char:
296                 piesces_match_char.remove(i)
297         elif c == 2:
298             if self.get_game().get_piece_characteristics(i).
SOLID == True and i in piesces_match_char:
299                 piesces_match_char.remove(i)
300         elif c == 3:
301             if self.get_game().get_piece_characteristics(i).
SQUARE == True and i in piesces_match_char:
302                 piesces_match_char.remove(i)
303         elif c == 4:
304             if self.get_game().get_piece_characteristics(i).
HIGH == False and i in piesces_match_char:
305                 piesces_match_char.remove(i)
306         elif c == 5:
307             if self.get_game().get_piece_characteristics(i).
COLOURED == False and i in piesces_match_char:
308                 piesces_match_char.remove(i)
309         elif c == 6:
310             if self.get_game().get_piece_characteristics(i).
SOLID == False and i in piesces_match_char:
311                 piesces_match_char.remove(i)
312         elif c == 7:
313             if self.get_game().get_piece_characteristics(i).
SQUARE == False and i in piesces_match_char:
314                 piesces_match_char.remove(i)
315
316         if len(piesces_match_char) > 0: #if there are at least one
matched element return it
317         return piesces_match_char[0]
318     return -1 #return -1 if there isn't any piece that match the
char and isn't already place
319
320
321 def save_opportunity(agent, vet, i, verticale, char) -> None:
322     free_places = []
323     ind_diag = 0
324     ind_diag_rev = 3
325     for j in range(4):
326         if vet[j] == -1: #check free place
327             if verticale == 0: #vet is horiz
328                 if i == -1: #check if is main diag
329                     free_places.append((ind_diag, ind_diag))
330                 else:
331                     free_places.append((i, j))
332             else:
333                 if i == -1: #check if is antidiag
334                     free_places.append((ind_diag, ind_diag_rev))
335                 else:
336                     free_places.append((j, i))
337     ind_diag+=1
338     ind_diag_rev-=1

```

```

339     agent.opportunity[len(free_places)].append((free_places,
340     char)) #append tupla in the correct dict list
341     #print("save ", (free_places, char))
342 def check_opportunity(agent) -> None:
343     '''
344     return a dict with all the opportunity on the board.
345     Key is the level of the opportunity
346     Value is an array of tuple (array of free position of the
347     opp, char of opp)
348     '''
349     agent.opportunity = {1: [], 2: [], 3: [], 4: []} #reset
350     opportunity vector
351     mat = agent.get_game().get_board_status() #get board
352     #print("mat in check ")
353     #print(mat)
354
355     for i in range(4):
356         horiz = mat[i]
357         vert = mat[:,i]
358
359         #HORIZ
360         #check if in horiz there are not element without char
361         HIGH
362         if sum(1 for x in horiz if not agent.get_game().
363         get_piece_characteristics(x).HIGH and x != -1) == 0:
364
365             save_opportunity(agent, horiz, i, 0, 0)
366
367         #check if in horiz there are not element without char
368         COLOURED
369         if sum(1 for x in horiz if not agent.get_game().
370         get_piece_characteristics(x).COLOURED and x != -1) == 0:
371
372             save_opportunity(agent, horiz, i, 0, 1)
373
374         #check if in horiz there are not element without char
375         SOLID
376         if sum(1 for x in horiz if not agent.get_game().
377         get_piece_characteristics(x).SOLID and x != -1) == 0:
378
379             save_opportunity(agent, horiz, i, 0, 2)
380
381         #check if in horiz there are not element without char
382         SQUARE
383         if sum(1 for x in horiz if not agent.get_game().
384         get_piece_characteristics(x).SQUARE and x != -1) == 0:
385
386             save_opportunity(agent, horiz, i, 0, 3)
387
388         #check if in horiz there are not element with char HIGH
389         -> are all low
390         if sum(1 for x in horiz if agent.get_game().
391         get_piece_characteristics(x).HIGH and x != -1) == 0:

```

```

383         save_opportunity(agent, horiz, i, 0, 4)
384
385         #check if in horiz there are not element with char
386         COLOURED -> are all WHITE
387         if sum(1 for x in horiz if agent.get_game().
388 get_piece_characteristics(x).COLOURED and x != -1) == 0:
389
390         save_opportunity(agent, horiz, i, 0, 5)
391
392         #check if in horiz there are not element with char
393         SOLID -> are all holled
394         if sum(1 for x in horiz if agent.get_game().
395 get_piece_characteristics(x).SOLID and x != -1) == 0:
396
397         save_opportunity(agent, horiz, i, 0, 6)
398
399         #check if in horiz there are not element with char
400         SQUARE -> are all CIRCUL
401         if sum(1 for x in horiz if agent.get_game().
402 get_piece_characteristics(x).SQUARE and x != -1) == 0:
403
404         save_opportunity(agent, horiz, i, 0, 7)
405
406         #VERT
407         #check if in vert there are not element without char
408         HIGH
409         if sum(1 for x in vert if not agent.get_game().
410 get_piece_characteristics(x).HIGH and x != -1) == 0:
411
412         save_opportunity(agent, vert, i, 1, 0)
413
414         #check if in vert there are not element without char
415         COLOURED
416         if sum(1 for x in vert if not agent.get_game().
417 get_piece_characteristics(x).COLOURED and x != -1) == 0:
418
419         save_opportunity(agent, vert, i, 1, 1)
420
421         #check if in vert there are not element without char
422         SOLID
423         if sum(1 for x in vert if not agent.get_game().
get_piece_characteristics(x).SOLID and x != -1) == 0:

```

```

424         save_opportunity(agent, vert, i, 1, 4)
425
426         #check if in vert there are not element with char
427         COLOURED -> are all WHITE
428         if sum(1 for x in vert if agent.get_game().
429 get_piece_characteristics(x).COLOURED and x != -1) == 0:
430
431             save_opportunity(agent, vert, i, 1, 5)
432
433             #check if in vert there are not element with char SOLID
434             -> are all holled
435             if sum(1 for x in vert if agent.get_game().
436 get_piece_characteristics(x).SOLID and x != -1) == 0:
437
438                 save_opportunity(agent, vert, i, 1, 6)
439
440                 #check if in vert there are not element with char
441                 SQUARE -> are all CIRCUL
442                 if sum(1 for x in vert if agent.get_game().
443 get_piece_characteristics(x).SQUARE and x != -1) == 0:
444
445                     save_opportunity(agent, vert, i, 1, 7)
446
447                     diag = mat.diagonal() #take main diagonal
448                     #Diag
449                     #check if in diag there are not element without char HIGH
450                     if sum(1 for x in diag if not agent.get_game().
451 get_piece_characteristics(x).HIGH and x != -1) == 0:
452
453                         save_opportunity(agent, diag, -1, 0, 0)
454
455                         #check if in diag there are not element without char
456                         COLOURED
457                         if sum(1 for x in diag if not agent.get_game().
458 get_piece_characteristics(x).COLOURED and x != -1) == 0:
459
460                             save_opportunity(agent, diag, -1, 0, 1)
461
462                             #check if in diag there are not element without char SOLID
463                             if sum(1 for x in diag if not agent.get_game().
464 get_piece_characteristics(x).SOLID and x != -1) == 0:
465
466                                 save_opportunity(agent, diag, -1, 0, 2)
467
468                                 #check if in diag there are not element without char SQUARE
469                                 if sum(1 for x in diag if not agent.get_game().
470 get_piece_characteristics(x).SQUARE and x != -1) == 0:
471
472                                     save_opportunity(agent, diag, -1, 0, 3)
473
474                                     #check if in diag there are not element with char HIGH ->
475                                     are all low
476                                     if sum(1 for x in diag if agent.get_game().
477 get_piece_characteristics(x).HIGH and x != -1) == 0:

```

```

468         save_opportunity(agent, diag, -1, 0, 4)
469
470         #check if in diag there are not element with char COLOURED
471         -> are all WHITE
472         if sum(1 for x in diag if agent.get_game().
473         get_piece_characteristics(x).COLOURED and x != -1) == 0:
474
475         save_opportunity(agent, diag, -1, 0, 5)
476
477         #check if in diag there are not element with char SOLID ->
478         are all holled
479         if sum(1 for x in diag if agent.get_game().
480         get_piece_characteristics(x).SOLID and x != -1) == 0:
481
482         save_opportunity(agent, diag, -1, 0, 6)
483
484         #check if in diag there are not element with char SQUARE ->
485         are all CIRCUL
486         if sum(1 for x in diag if agent.get_game().
487         get_piece_characteristics(x).SQUARE and x != -1) == 0:
488
489         save_opportunity(agent, diag, -1, 0, 7)
490
491         diag = np.fliplr(mat).diagonal() #take anti diagonal
492         if sum(1 for x in diag if not agent.get_game().
493         get_piece_characteristics(x).HIGH and x != -1) == 0:
494
495         save_opportunity(agent, diag, -1, 1, 0)
496
497         #check if in diag there are not element without char
498         COLOURED
499         if sum(1 for x in diag if not agent.get_game().
500         get_piece_characteristics(x).COLOURED and x != -1) == 0:
501
502         save_opportunity(agent, diag, -1, 1, 1)
503
504         #check if in diag there are not element without char SOLID
505         if sum(1 for x in diag if not agent.get_game().
506         get_piece_characteristics(x).SOLID and x != -1) == 0:
507
508         save_opportunity(agent, diag, -1, 1, 2)
509
510         #check if in diag there are not element without char SQUARE
511         if sum(1 for x in diag if not agent.get_game().
512         get_piece_characteristics(x).SQUARE and x != -1) == 0:
513
514         save_opportunity(agent, diag, -1, 1, 3)
515
516         #check if in diag there are not element with char HIGH ->
517         are all low
518         if sum(1 for x in diag if agent.get_game().
519         get_piece_characteristics(x).HIGH and x != -1) == 0:
520
521         save_opportunity(agent, diag, -1, 1, 4)
522
523         #check if in diag there are not element with char COLOURED
524         -> are all WHITE

```

```

511     if sum(1 for x in diag if agent.get_game().
512         get_piece_characteristics(x).COLOURED and x != -1) == 0:
513         save_opportunity(agent, diag, -1, 1, 5)
514
515         #check if in diag there are not element with char SOLID ->
516         are all holled
517         if sum(1 for x in diag if agent.get_game().
518             get_piece_characteristics(x).SOLID and x != -1) == 0:
519             save_opportunity(agent, diag, -1, 1, 6)
520
521             #check if in diag there are not element with char SQUARE ->
522             are all CIRCUL
523             if sum(1 for x in diag if agent.get_game().
524                 get_piece_characteristics(x).SQUARE and x != -1) == 0:
525                 save_opportunity(agent, diag, -1, 1, 7)

```

Listing 8: utilities.py



## 3 Laboratory

### 3.1 Lab 1: Set Covering

For this problem I have implemented a Uniform Cost Search Algorithm, in jupyter notebook is the function UCF().

In which the cost (weight) is the number of repeated element. For example with  $N = 5$  and solution = [ [0,2], [1,2,3,4] ] the cost is 1, due to the fact that the number 2 is repeated.

I have also create a specific class MyNode for store all the data structures that I need to represent a node. I based my solution on the Pseudocode found in <https://python.plainenglish.io/uniform-cost-search-ucs-algorithm-in-python-ec3ee03fca9f>.

Numbers of visited nodes for different N values:

- $N = 5$ , 122 nodes visited
- $N = 10$ , 874 nodes visited
- $N = 20$ , 23.416 nodes visited

With bigger N is not feasible the execution.

```
1 import random
2
3 def problem(N, seed=42):
4     random.seed(seed)
5     return [
6         list(set(random.randint(0, N - 1) for n in range(random.
7             randint(N // 5, N // 2))))
8         for n in range(random.randint(N, N * 5))
9     ]
10
11 from select import select
12
13 class MyNode:
14     weight = 0
15     lists = []
16     numbers = []
17
18     def __init__(self, weight = 0, lists = [], numbers = []):
19         self.weight = weight
20         self.lists = lists
21         self.numbers = numbers
22
23     def calculate_weight(self):
24         weight = 0
25         for inner_list in self.lists:
26             for x in inner_list:
27                 if x in self.numbers:
```

```

27         weight+=1
28     else:
29         self.numbers.append(x)
30     self.weight = weight
31
32     def check_sol(self, N):
33         sol = True
34         for i in range(N):
35
36             if i not in self.numbers:
37                 sol = False
38         return sol
39
40
41     def get_min_node(opened_list, N):
42         min_weight = -1
43         sel_node = []
44         for node in opened_list:
45             if min_weight == -1 or min_weight > node.weight:
46                 min_weight = node.weight
47                 sel_node = node
48         return sel_node
49
50     def get_child(select_nodes, listOfList, opened_list):
51         for l in listOfList:
52             if l not in select_nodes.lists:
53                 new_list = list()
54                 for l2 in select_nodes.lists:
55                     new_list.append(l2)
56                 new_list.append(l)
57                 node = MyNode(0, new_list, list())
58                 node.calculate_weight()
59                 opened_list.append(node)
60
61         return opened_list
62
63
64
65     def UCF(listOfList, N):
66
67         first_node = MyNode()
68         opened_list = list()
69         closed_list = list()
70         opened_list.append(first_node)
71
72         num_visited_nodes = 0
73         while True:
74             select_node = get_min_node(opened_list, N)
75             num_visited_nodes+=1
76             if select_node.check_sol(N):
77                 print("sol: ", select_node.lists)
78                 print("weight: ", select_node.weight)
79                 print("number of nodes visited: ",
num_visited_nodes)
80                 return
81                 closed_list.append(select_node)
82                 opened_list.remove(select_node)

```

```

83         opened_list = get_child(select_node, listOfList,
opened_list)
84         #print("after child gen ----")
85         #for c in opened_list:
86             #    print(c.lists)
87             #    print("weight: ", c.weight)
88
89
90
91
92     def main(N):
93         listOfList = problem(N)
94         print(listOfList)
95         print("start UCF")
96         UCF(listOfList, N)

```

### 3.1.1 Review by Ruggero Nocera

<https://github.com/EnricoMagliano/computational-intelligence/issues/1>

The first thing I noticed is that running codes multiple times it seldom raises an `AttributeError`.

Click to see the output After investigating a bit, I realized this is the default behaviour when the problem doesn't have a solution. I would have designed it in a different way.

The code keeps track of "closed nodes" (the ones removed from the frontier and processed) as in standard implementation of UFC, but in the rest of the code this variable is unused, so why keeping it?

The Priority Queue is not implemented, instead the code uses a list and search for the every time (`get_min_node`). Using a priority queue could improve the execution time.

The `__init__` method of class `MyNode` has a default parameter `weight=0`, and when the code calls `MyNode`'s constructor it calls immediately after the `calculate_weight` method. From my perspective it would be better to "hide" this method and call it inside `__init__`.

There is an `from select import select` at the begin of the second cell, probably an auto-import by some IDE?

The code is an implementation of Uniform Cost Search, it provides the best solution, so it reaches its goal, nothing to say about that. However the style of the code could be made more pythonic.

This review allowed me to study in deep the differences (and the similarities!) between Dijkstra Algorithm and Uniform Cost Search.

### 3.2 Lab 2: Set Covering with Genetic Algorithm

For this problem I have implemented a very simple genetic algorithm, that use for 50% mutations and for 50% crossover in 100 generations.

I based my solution on the Code of the prof. Squillero found in <https://github.com/squillero/computational-intelligence/blob/master/2022-23/one-max.ipynb>.

This solution works very fast instead of Dijkstra's algorithm but doesn't found the best solution, anyway it can be run also with large N like 50, 100 in a reasonable time.

```
1 import logging
2 from collections import namedtuple
3 import random
4 from itertools import compress
5 import sys
6
7 POPULATION_SIZE = 200
8 OFFSPRING_SIZE = 50
9 N = 50
10
11 def problem(seed=42):
12     random.seed(seed)
13     return [
14         list(set(random.randint(0, N - 1) for n in range(random.
15             randint(N // 5, N // 2))))
16         for n in range(random.randint(N, N * 5))
17     ]
18 def takeList(listOfList, mask): #given a mask array of 1 and 0 get
19     the corresponding list in listOfList
20     return list(compress(listOfList, mask))
21
22 def fitness(genome, listOfList): # calculate the fitness
23     lists = takeList(listOfList, genome)
24     cov = list()
25     fitness = 0
26     for innerList in lists:
27         for x in innerList:
28             if x in cov:
29                 fitness-=1
30             else:
31                 fitness+=1
32                 cov.append(x)
33     return fitness #fitness = num. of set covering - num. of
34     duplicates
35 def generatePopulation(SIZE, listOfList): #return population, one
36     individual is a tuple of a mask array of the list taken and his
37     fitness
38     population = list()
39     for genome in [tuple([random.choice([1,0]) for _ in range(SIZE)
40         ]) for _ in range(POPULATION_SIZE)]:
```

```

38     population.append((genome, fitness(genome, listOfList)))
39     return population
40
41 def select_parent(population, tournament_size=10):
42     return max(random.choices(population, k=tournament_size), key=
43                 lambda i: i[1])
44
45 def cross_over(g1, g2):
46     cut = random.randint(0, len(g1))
47     ng = g1[:cut] + g2[cut:]
48     return ng
49
50 def mutation(g):
51     point = random.randint(0, len(g)-1)
52     return g[:point] + (1-g[point],) + g[point+1:]
53
54 def check_sol(sol): #check if a genoma is a solution
55     cov = list()
56     for innerList in sol:
57         for x in innerList:
58             if x not in cov:
59                 cov.append(x)
60     if len(cov) == N:
61         return True
62     else:
63         return False
64
65 def GA(listOfList):
66     best_sol = None
67     best_sol_fit = None
68
69     population = generatePopulation(len(listOfList), listOfList)
70     for generation in range(100):
71         offsprings = list()
72         for i in range(OFFSPRING_SIZE):
73             o = ()
74             if random.random() < 0.5:
75                 p = select_parent(population)
76                 o = mutation(p[0])
77             else:
78                 p1 = select_parent(population)
79                 p2 = select_parent(population)
80                 o = cross_over(p1[0], p2[0])
81             offsprings.append((o, fitness(o, listOfList)))
82         population = population + offsprings
83         population = sorted(population, key=lambda i: i[1], reverse=
84                             True)[:POPULATION_SIZE]
85         for i in population:
86             genome_sol = takeList(listOfList, i[0])
87             if check_sol(genome_sol):
88                 fit_of_sol = fitness(i[0], listOfList)
89                 if best_sol_fit is not None:
90                     if fit_of_sol > best_sol_fit:
91                         best_sol = genome_sol
92                         best_sol_fit = fit_of_sol
93                 else:
94                     best_sol = genome_sol

```

```

93         best_sol_fit = fit_of_sol
94         break
95
96     print("after 100 generations")
97     print("best solution:")
98     print(best_sol)
99     print(best_sol_fit)
100
101
102 def main():
103     listOfList = problem()
104     print(listOfList)
105     print("start GA")
106     GA(listOfList) #start GA with the listOfList generated by the
                    #problem

```

Listing 9: Lab 2: Set Covering with Genetic Algorithm

### 3.2.1 Review by Gabriele Greco

<https://github.com/EnricoMagliano/computational-intelligence/issues/3>

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

About your solution:

- Your solution is similar to mine, I started from one max solution of professor too, but changed it a little bit mutation rate of 0.5 is good, I used 0.55, you could try new values. With higher values it tend to step away from optimal solution with higher values of N, so a lower value is the best choice
- the for loop to check if you found the solution can be expensive for large population and with large num of generations. You can evaluate the new gen after crossover or mutation after the if loop and search for the best solution afterwards
- about the fitness function, the number of set cover minus the number of duplicates it's a good idea. Another one can be to compute the number of distinct elements and the number of total elements of your genome
- You can add more info about your code and solution in the readme like for example your results with different kind of iterations, different value of offspring, mutation rate etc...
- Good use of comments, I advice to add more
- You can use a better output format to print solutions in order to compare them and make them readable. For example INFO:root: Solution for N=10: w=10 (bloat=0%) Fitness calls=1512, you can add in the main a for that loop for

different value of N and print different solution. Here's an example: for N in [5, 10, 20, 50, 100, 500, 1000]: GA\_function(N) logging.info(f" Solution for N=N:," + f"w=w " + f"(bloat=(w-N)/N\*100.0f%) " + f"Fitness calls=len(fitness\_log)")

Overall, good job, keep working like this and improving :)

### 3.2.2 Review by Amin Mbare

<https://github.com/EnricoMagliano/computational-intelligence/issues/6>

The code is readable and clearly written, great job on that. the idea of using `itertools.compress` to take the sublists you used for your solution is very nice , I am sure it will be very useful for me in future projects.

I wanted to first address the fact that you are iterating over all the population to check whether their feasibility and if they are the best solution or not . So one side of me thinks that iterating over all the population could be not very useful since you have already sorted your population based on their fitness. so if the first solution isn't feasible logically the second wouldn't be feasible as well. However another side of me thinks it is necessary because maybe the first solution in the population could have a high fitness since it doesn't contain redundant sub-lists but doesn't cover all the set and the second solution could have a lower fitness because it contains redundant sublists but instead it covers all the set.

If I had to suggest another idea, I would say maybe trying to apply the second strategy of GA suggested by prof on the slides, decrease the randomness between mutation and cross-over but make them sequential instead:

1. choose parents
2. apply genetic operator
3. mutation

when you iterate over population to find the best solution ,you are recomputing the fitness again after you check the feasibility. I believe it might be unnecessary since the fitness of that solution is already known, just to decrease the computational load.

I could as well suggest one minor addition , I have tried to run your genetic algorithm on the N=100 instance , the total number of elements in the best solution was 353 , the fitness was -153 . I used 1000 generations and kept the other parameters. Just looking at the best solution's fitness I am sure that it contains some genes that cover twice the elements to be covered . I could suggest making a heuristic that cleans your best solution by deleting these genes.



Maybe try to provide the parameters for each instance ,So we can test the algorithm for different instances.

A more elaborated readme file would have been appreciated by providing the results you have obtained for every instance or explaining the encoding for your solution . Just to improve the readability and comprehension of the code.

Overall very good job.

### 3.2.3 Review by Lorenzo Bellino

<https://github.com/EnricoMagliano/computational-intelligence/issues/5>

Your solution was well written and the code readable and understandable and it was a good idea to use a mask in order to compute the solution and using itertools built-in function (I might implement it myself), none the less i have some suggestion for your code:

GA main function: You don't need to check all the individuals after each generation to check if you have found an optimal solution, since the list is sorted by te fitness if the first one is not an optimal solution is impossible that an individual with a lower fitness score will be optimal. This will improve the runtime also beacause after each generation you recalculate the fitness for each individual but you could simply access it without recalculating with `i[1]`.

`check_sol`: this function could be improved by using list comprehension in order to improve runtime for example: `return len(set([loci for gene in sol for loci in gene])) == N`

`fitness`: in the same way as for `check_sol` you could have used list comprehension

General: Could have reported the result in the README for better understanding and ease of use

### 3.2.4 Review by Davide Aiello

<https://github.com/EnricoMagliano/computational-intelligence/issues/4>

Good rappresentation of the problem and smart approach to convert the genome to a list of lists by exploitying the compress function of itertools (I think I will steal this idea) but there are some things to highlight:

Major:

- Modify the tournament size according to the value of  $N$  could improve your solution (by choosing a value of the tournament size equal to  $N/2$ , you move from  $w = 129$  to  $w = 120$  for  $N = 50$  and from  $w = 2388$  to  $w = 1557$  for  $N = 100$ )
- Instead of iterate on the population and then break the loop after the first item, you can just pick the first element of the list to obtain the best one for that generation since the population is already sorted according to the fitness of each individual
- Inside the loop on the population, for the best individual you recompute the fitness (which exploits an expensive double loop), but this information is embedded in the individual itself: accessing to the value with index 1 is enough.

Minor

- Since you're comparing just the fitness of the new best individual with the fitness of the old best one, you can call the function `takeList` at the end of the generations in order to print the correct list and save instead the previous best individual.
- In order to reduce the number of duplicated inner lists of the problem, it would have been better to build a set from the list generated by the problem.
- You should print the value of  $w$ , that is the list length of the solution you found.
- Adding more comment could help in code readability
- You should import only the things you use
- Instead of using a double loop to check the solution, you can use list comprehension with a double loop inside and then create a set from it ( `set([x for y in sol for x in y])` ).

Good work and good improvements

### 3.2.5 Review by Diego Gasco

<https://github.com/EnricoMagliano/computational-intelligence/issues/2>

Hi @EnricoMagliano I decided to review your code about the set covering problem using Genetic Algorithm! I hope that comments and suggestions I left below can be useful for you.

Major:

- First of all, you use the same heuristic that I used for the fitness function in my solution. After some trials I think that is a good choice for this kind of problem.
- Another thing similar to mine is the random initialization of the population genomes at the beginning. I saw different approaches in other solutions (all genes to False, all genes to False except for one, etc...), but also in this case, after some trials, I think that this idea can be considered quite good.
- Mutation in this problem can be a double-edged sword, because if the rate is high we have the risk of getting away from possible solution paths. My suggestion is to lower the mutation rate and give more space to crossover. Another interesting thing can be to variate this rate so keep it higher at the beginning (exploration) and lower it during the computation (exploitation).
- Instead of having a loop over the population for searching a solution, you can just check every time the new genome create after mutation or crossover and the new fitness. In this way you have fitness and genome for doing comparisons and searching a local minimum.
- I would like to see more details in the README, for example, for each N the parameters used and the solution main factors (weight and steps).

Minor:

- The fact that there are some comments in the code is very good, I suggest you to add a few more.

Conclusions: Your approach is quite good, you can think about playing with random parameters and checking if better solutions can be reached! Good job ;)

### 3.3 Lab 3: Policy Search

#### 3.3.1 Task 1

For the first task I have implemented the enrico strategy, in which simply checks if there is only one active row, if true takes all element and win, instead takes all element from the shortest row. This strategy lose against gabriele one, but perform better than pure random one.

```
1 def nim_sum(state: Nim) -> int:
2     _, result = accumulate(state.rows, xor)
3     return result
4
5
6 def cook_status(state: Nim) -> dict:
7     cooked = dict()
8     cooked["possible_moves"] = [
9         (r, o) for r, c in enumerate(state.rows) for o in range(1,
10            c + 1) if state.k is None or o <= state.k
11     ]
12     cooked["active_rows_number"] = sum(o > 0 for o in state.rows)
13     cooked["shortest_row"] = min((x for x in enumerate(state.rows)
14         if x[1] > 0), key=lambda y: y[1])[0]
15     cooked["longest_row"] = max((x for x in enumerate(state.rows)),
16         key=lambda y: y[1])[0]
17     cooked["nim_sum"] = nim_sum(state)
18
19     brute_force = list()
20     for m in cooked["possible_moves"]:
21         tmp = deepcopy(state)
22         tmp.nimbling(m)
23         brute_force.append((m, nim_sum(tmp)))
24     cooked["brute_force"] = brute_force
25
26     return cooked
27
28 def pure_random(state: Nim) -> Nimply:
29     row = random.choice([r for r, c in enumerate(state.rows) if c >
30         0])
31     num_objects = random.randint(1, state.rows[row])
32     return Nimply(row, num_objects)
33
34 def gabriele(state: Nim) -> Nimply:
35     """Pick always the maximum possible number of the lowest row"""
36     possible_moves = [(r, o) for r, c in enumerate(state.rows) for
37         o in range(1, c + 1)]
38     return Nimply(*max(possible_moves, key=lambda m: (-m[0], m[1])))
39
40 def enrico(state: Nim) -> Nimply: #my strategy
41     data = cook_status(state)
42     if data["active_rows_number"] == 1:
43         return (data["longest_row"], state._rows[data["longest_row"]])
44     else:
45         return (data["shortest_row"], state._rows[data["shortest_row"]])
46
47 def optimal_strategy(state: Nim) -> Nimply:
48     data = cook_status(state)
```

```

40     return next((bf for bf in data["brute_force"] if bf[1] == 0),
    random.choice(data["brute_force"]))[0]

```

Listing 10: Lab 3: Task 1

### 3.3.2 Task 2

For the second task, I have implemented an hard coded strategy, that use 3 parameters (3 floating between 0 to 1):

- aggressive: more aggressive strategy has an higher probability to remove an entire line.
- pref\_long: higher probability to work on a longest line.
- remove\_perc: higher remove\_perc, more element are removed in a line in a non aggressive ply.

For evolving this hard coded strategy, I have used an Evolutionary Algorithm, where the genome is tuple of the 3 parameters, while fitness is the percentage of matches won against the opponent (10 matches on which for an half of the games start the opponent and for the other half start the genome). This algorithm works for 10 generation on which the offsprings are made by mutations.

Results, percentage matches won by my best genome against different opponents:

- gabriele: 0.7 enrico: 0.9
- optimal: 0.0
- pure random: 0.8

I based my solution on the Code developed in lab2 <https://github.com/EnricoMagliano/computational-intelligence/edit/main/lab2> for the EA, and on the prof. Squillero code [https://github.com/squillero/computational-intelligence/blob/master/2022-23/lab3\\_nim.ipynb](https://github.com/squillero/computational-intelligence/blob/master/2022-23/lab3_nim.ipynb) for the Nim game structure.

```

1 import logging
2 from collections import namedtuple
3 import random
4 from typing import Callable
5 from copy import deepcopy
6 from itertools import accumulate
7 from operator import xor
8
9 Nimply = namedtuple("Nimply", "row, num_objects")
10 class Nim:
11     def __init__(self, num_rows: int, k: int = None) -> None: #k is
12         the max number of object that can be removed from a line
13         self._rows = [i * 2 + 1 for i in range(num_rows)]
14         self._k = k

```

```

14
15 def __bool__(self):
16     return sum(self._rows) > 0
17
18 def __str__(self):
19     return "<" + " ".join(str(_) for _ in self._rows) + ">"
20
21 @property
22 def rows(self) -> tuple:
23     return tuple(self._rows)
24
25 @property
26 def k(self) -> int:
27     return self._k
28
29 def nimming(self, ply: Nimply) -> None:
30     row, num_objects = ply
31     assert self._rows[row] >= num_objects
32     assert self._k is None or num_objects <= self._k
33     self._rows[row] -= num_objects
34
35 def nim_sum(state: Nim) -> int:
36     _, result = accumulate(state.rows, xor)
37     return result
38
39
40 def cook_status(state: Nim) -> dict:
41     cooked = dict()
42     cooked["possible_moves"] = [
43         (r, o) for r, c in enumerate(state.rows) for o in range(1,
44             c + 1) if state.k is None or o <= state.k
45     ]
46     cooked["active_rows_number"] = sum(o > 0 for o in state.rows)
47     cooked["shortest_row"] = min((x for x in enumerate(state.rows)
48         if x[1] > 0), key=lambda y: y[1][0])
49     cooked["longest_row"] = max((x for x in enumerate(state.rows)),
50         key=lambda y: y[1][0])
51     cooked["nim_sum"] = nim_sum(state)
52
53     brute_force = list()
54     for m in cooked["possible_moves"]:
55         tmp = deepcopy(state)
56         tmp.nimming(m)
57         brute_force.append((m, nim_sum(tmp)))
58     cooked["brute_force"] = brute_force
59
60     return cooked
61
62 def hard_coded_strategy(state: Nim, genome) -> Nimply:
63     data = cook_status(state)
64     aggressive, pref_long, remove_perc = genome
65     if data["active_rows_number"] == 1: #win conditions
66         return (data["longest_row"], state._rows[data["longest_row"]])
67
68     if random.random() > 1-aggressive: #follow an aggressive
69         strategy remove an entire line

```

```

66         if random.random() > 1-pref_long:
67             return (data["longest_row"], state._rows[data["
longest_row"]])
68         else:
69             return (data["shortest_row"], state._rows[data["
shortest_row"]])
70     else: #follow an conservative strategy remove only a few
element in a line
71         if random.random() > 1-pref_long:
72             num_of_rem = round(state._rows[data["longest_row"]]*
remove_perc)
73             if num_of_rem == 0:
74                 num_of_rem+=1
75             return (data["longest_row"], num_of_rem)
76         else:
77             num_of_rem = round(state._rows[data["shortest_row"]]*
remove_perc)
78             if num_of_rem == 0:
79                 num_of_rem+=1
80             return (data["shortest_row"], num_of_rem)
81     return pure_random(state)
82
83 def make_strategy(genome: dict) -> Callable:
84     def evolvable(state: Nim) -> Nimply:
85         data = cook_status(state)
86
87         if random.random() < genome["p"]:
88             ply = Nimply(data["shortest_row"], random.randint(1,
state.rows[data["shortest_row"]]))
89         else:
90             ply = Nimply(data["longest_row"], random.randint(1,
state.rows[data["longest_row"]]))
91
92         return ply
93
94     return evolvable
95 NUM_MATCHES = 10
96 NIM_SIZE = 10
97
98
99 def evaluate(opponent: Callable, genome) -> float:
100     won = 0
101     last_player_start = 1
102     for m in range(NUM_MATCHES):
103         nim = Nim(NIM_SIZE)
104         player = 1 - last_player_start
105         last_player_start = player
106         while nim:
107             if player == 0:
108                 ply = opponent(nim)
109             else:
110                 ply = hard_coded_strategy(nim, genome)
111                 nim.nimming(ply)
112                 player = 1 - player
113             if player == 0:
114                 won += 1
115

```

```

116         return won / NUM_MATCHES #percentage of match won against the
            opponent
117 POPULATION_SIZE = 20
118 OFFSPRING_SIZE = 10
119 OPPONENT = gabriele
120 def fitness(genome): # calculate the fitness
121     return evaluate(OPPONENT, genome)
122 def generatePopulation(): #return population, one individual is a
            tuple of a mask array of the list taken and his fitness
123     population = list()
124     for genome in range(POPULATION_SIZE):
125         genome = (random.random(), random.random(), random.random()
            )
126         population.append((genome, fitness(genome)))
127     return population
128 def mutation(g):
129
130     point = random.randint(0, len(g)-1)
131     return g[:point] + (random.random(),) + g[point+1:]
132 def select_parent(population, tournament_size=10):
133     return max(random.choices(population, k=tournament_size), key=
        lambda i: i[1])
134 def GA():
135     best_sol = None
136     best_sol_fit = None
137
138     population = generatePopulation()
139     for generation in range(10):
140         offsprings = list()
141         for i in range(OFFSPRING_SIZE):
142             o = ()
143             p = select_parent(population)
144             o = mutation(p[0])
145             offsprings.append((o, fitness(o)))
146         population = population + offsprings
147         population = sorted(population, key=lambda i:i[1], reverse=
            True)[:POPULATION_SIZE]
148
149     best_sol = population[0][0]
150     best_sol_fit = population[0][1]
151     print("after 100 generations")
152     print("best solution:")
153     print(best_sol)
154     return best_sol
155 best_sol= GA()
156 print("win-rate:")
157 evaluate(gabriele, best_sol)

```

Listing 11: Lab 3: Task 2

### 3.3.3 Task 3

MinMax Agent



```

1 class Agent():
2     def __init__(self, state: Nim):
3         self.nim = state
4         self.tree = None
5
6     def createTree(self):
7         creation = True
8         self.tree = Node('root', value = list(self.nim.rows))
9         parents = [self.tree]
10        childs = []
11        level = 1
12        my_turn = True
13        while creation:
14            #print("level:", level)
15            #print("parents: ", parents)
16            count_parents_no_child = 0
17            for parent in parents:
18                #print(parent)
19                moves = [(r, o) for r, c in enumerate(parent.value)
20                for o in range(1, c + 1) ]
21                #print("moves", moves)
22                if len(moves) == 0:
23                    count_parents_no_child+=1
24                else:
25                    for ply in moves:
26                        aus = parent.value.copy()
27                        aus[ply[0]] = (aus[ply[0]] - ply[1])
28                        child = Node(level,value=aus, after_me=
29                        level%2, parent=parent)
30                        childs.append(child)
31                        if count_parents_no_child == len(parents):
32                            creation = False #no more node to explore
33                        level+=1
34                        my_turn != my_turn
35                        parents = childs
36                        childs = []
37                        #print(RenderTree(self.tree))
38
39        def findMybestNextMove(self, root: Node) -> Nimply:
40            d = search.findall(root, filter_=lambda node: node.value ==
41            [0]*NIM_SIZE and node.after_me == 1)
42            best = min(d, key=lambda n: n.name)[0]
43
44            return pure_random(self.nim)
45
46        def minmax(self) -> Nimply:
47
48            if self.tree == None:
49                self.createTree() #create tree for first time
50            else:
51                for child in self.tree.children: #updaate
52                    tree root with the current nim state
53                    if child.value == list(self.nim.rows):
54                        self.tree = child

```

```

54         self.findMybestNextMove(self.tree)
55
56         return pure_random(self.nim)

```

Listing 12: Lab 3: Task 3

### 3.3.4 Task 4

For the fourth task, I have implemented an agent using reinforcement learning. In particular my agent during the training phase learning from the winning matches updating a dictionary, `self.learned`, where saves the relation between nim status (element per row) and the winning ply. Obviously use another dictionary, `self.current_move`, to keep track of the ply of the current match attending for the match's outcome. During the train phase the `random_factor` is decrease to encourage exploitation by moving forward with matches, and also the opponent is switched starting from the silliest one (pure random) to the optimal strategy.

Despite I think this is not a bad idea, the results are really bad, this agent is little better than pure random strategy, but worse than all the others. I would really appreciate if someone could improve it.

```

1 class Agent():
2     def __init__(self, nim: Nim, num_tot_matches):
3         self.nim = nim          #to be update at each game
4         self.random_factor = 1  #at the beginning is set to 1 -> 100
5         explore
6         self.learned = dict()    #key is the nim status (nim._rows)
7         the value is a dict of ( key: ply, value: score) from previous
8         games
9         self.current_move = dict() #key is the nim status, value
10        is the ply performed in the current game
11        self.num_matches = 0
12        self.num_tot_matched = num_tot_matches
13
14    def play(self) -> Nimply: #return a move
15        selected_ply = None
16        if random.random() > self.random_factor: # exploitation:
17            select best move in same status situation if exists(score must
18            be grater that 0)
19            if self.nim.print_state() in self.learned.keys():
20                moves = self.learned[self.nim.print_state()]
21                best = None
22                max = 0
23                for move, score in moves.items():
24                    if score > max:
25
26                        best = move
27                        max = score
28
29            if best == None:
30                selected_ply = pure_random(self.nim)
31            else:
32                selected_ply = best

```

```

27
28         else:
29             selected_ply = pure_random(self.nim)
30     else: #exploration
31         selected_ply = pure_random(self.nim)
32
33     self.current_move[self.nim.print_state()] = selected_ply
34
35     return selected_ply
36
37 def update_score(self, win): #in learned update score +1 if
38     agent wins or -1 if loses
39     self.random_factor = 1 - 2*(self.num_matches/self.
40     num_tot_matched) #update random factor for encrease the
41     exploitation the matches prograssion
42
43     for nim_state, move in self.current_move.items():
44         if nim_state in self.learned.keys():
45             if move in self.learned[nim_state].keys():
46                 if win:
47                     self.learned[nim_state][move] += 1
48                 else:
49                     self.learned[nim_state][move] -= 1
50             else:
51                 if win:
52                     self.learned[nim_state][move] = 1
53                 else:
54                     self.learned[nim_state][move] = -1
55         else:
56             if win:
57                 self.learned[nim_state] = {move: 1}
58             else:
59                 self.learned[nim_state] = {move: -1}
60
61 NUM_MATCHES_EVAL = 100
62 NIM_SIZE = 5
63 NUM_MATCHES_TRAINING = 5000
64 OPPONENT_TRAIN = [pure_random, enrico, gabriele, optimal_strategy]
65 OPPONENT_EVAL = enrico
66 def training(robot):
67     won=0
68     last_player_start = 1
69     for i in range(NUM_MATCHES_TRAINING):
70         nim = Nim(NIM_SIZE)
71         robot.nim = nim
72         robot.num_matches = i
73         player = 1 - last_player_start #for switching the starter
74         last_player_start = player
75         while nim:
76             if player == 0:
77                 ply = OPPONENT_TRAIN[int(i/(NUM_MATCHES_TRAINING/
78                 len(OPPONENT_TRAIN)))](nim) #select opponent starting from the
79                 silliest
80             else:
81                 ply = robot.play()
82                 nim.nimming(ply)
83                 player = 1 - player

```

```

79         if player == 0: #robot win
80             won+=1
81             robot.update_score(1)
82         else: #robot lose
83             robot.update_score(0)
84     print("won in training: ", won)
85 def evaluate(robot) -> float:
86     won = 0
87     last_player_start = 1
88     for m in range(NUM_MATCHES_EVAL):
89         nim = Nim(NIM_SIZE)
90         robot.nim = nim
91         player = 1 - last_player_start
92         last_player_start = player
93         while nim:
94             if player == 0:
95                 ply = OPPONENT_EVAL(nim)
96             else:
97                 ply = robot.play()
98                 nim.nimming(ply)
99                 player = 1 - player
100         if player == 0:
101             won += 1
102
103     return won / NUM_MATCHES_EVAL #percentage of match won against
104                                     the opponent
105 def main():
106     robot = Agent(None, NUM_MATCHES_TRAINING)
107
108     training(robot)
109
110     robot.random_factor = 0 #only exploitation
111     res = evaluate(robot)
112     print(res)

```

Listing 13: Lab 3: Task 4

### 3.3.5 Review by dfm88

<https://github.com/EnricoMagliano/computational-intelligence/issues/7>

In general I can say that the approach to the lab was good in terms of ideas. I noticed that you never set the upper bound  $k$  to the number of removable stick, this makes easier for the optimal strategy to win since can almost always play the best move, while in presence of an upper bound you can limit its choices.

Point 2: I used a very similar strategy, using a list of probability as genome, as a rule to choice the most probable strategy between a set of hardcoded strategies, of course this could have also being made parametrizing the parameters of these strategies but it was just another way to do it.

Point 4: I can't strictly suggest you how to improve your RL Agent since I had

also really poor results. I can tell you what I did differently:

- First of all I noticed that you made different behavior on the training based on the actual turn of the players, I simply made the agent play against itself without evaluating which move to do based on turn but simply giving more reward to the move that a NimSum agent would have done.

- I also noticed that it seems that you evaluated the whole set of moves as +1 or -1 based on the final match result, I preferred to evaluate every single move at runtime always based on what a NimSum agent would have done (+1 if my agent did the same move of NimSum, +0.3 if my agent did any move, but also the NimSum wouldn't have been able to do an optimal move -you can have this case more often if set a k boundary- and -1 if my agent would have the opportunity to do a NimSum move but didn't

Code: I suggest to parametrize more the function and evaluate to use a class from some function with lots of variables. The README was quite explicative, but remember also to put some more comments on the key logic

### 3.4 Review made by me

#### 3.4.1 Review dfm88 lab 1

[https://github.com/dfm88/computational\\_intelligence\\_2022/issues/2](https://github.com/dfm88/computational_intelligence_2022/issues/2)

The choice of using A\* as algorithm is good and also his implementation. You can reach the optima solution in a reasonable time, except for large N, in any case comparing with my Dijkstra solution, yours is better. The heuristic is also correct, simple but functional.

I have found some issue in the numpy import with poetry (but this could be some of my configuration problems). I have also found some difficulty in understanding the code, due to the fact that there aren't so many comment and you use too many classes and and data structures, that provides a good generalization but for this single task they are not all needed.

#### 3.4.2 Review s295103 lab 2

[https://github.com/s295103/ci\\_2022\\_s295103/issues/3](https://github.com/s295103/ci_2022_s295103/issues/3)

The algorithm looks correct and also the result are good. In detail: the choice of the fitness is correct but only in this case because we select only solutions as individual, also the selective pressure to create the next generation is correct. Recombination and mutation are basically correct but they are really simple, a more specific solution could be find for this problem. The choice to keep only the solutions could lead to the removal of individuals potentially close to the best solution, but i think that in this case is not a problem, due to the fact that the result are good.

Minors:

- ReadMe file is really well written and useful.
- I didn't understand how the parameters were chosen, like tournament size, population size and the ratio between mutation and recombination.
- The class structures are really useful? Maybe a simple tuple is enough to represent the individuals.

#### 3.4.3 Review merhametsize lab 2

<https://github.com/merhametsize/Computational-intelligence/issues/4>

The code and the logic are correct, good choice for individual representation and fitness. I like the idea of create an individual for each list and also the mutation realization, this following something like a grow approach. While

crossover is really basic, maybe something more specific for this task could be done. One other thing that i don't have understood is the parameters choices, like tournament size or the ratio between mutation and crossover.

Minors:

- The code is pythonic
- The ReadMe file could be done better, with more detail and some explanation of the choices made

#### 3.4.4 Review Diegomangasco lab3

[https://github.com/Diegomangasco/Computational\\_Intelligence/issues/5](https://github.com/Diegomangasco/Computational_Intelligence/issues/5)

In general the code is well organized and the readme file contains usefull explanations, while the commets in the code are poorly.

The evaluation is made using only one match, this is affect by randomness that can result in misleading results. Is better use more match and the win/number\_of\_match ratio (switching the starter).

All the solutions are made without considering the parameter k.

Task 1: Nothing to say, good implementations of the optimal strategy.

Task 2: Good hard coded strategy evolved by GA, the fitness is correct, in general all the structure of the evolution is correct except for the mutation that change the entire genome, while it should change only a part.

Task 3: Also here nothing to say, I really like your recursive function to explore the tree without directly create it.

Task 4: Good code structure of the evolved strategy, but I didn't understand how the reward are updates and why randomly initialize the rewards?

In conclusion I can say that this work is reasoned and well implemented.

Good job