Computational Intelligence Exam Report

Enrico Magliano - s295692@studenti.polito.it

January 2023

Contents

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)`) 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 overrided 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.

```python
class RandomPlayer(quarto.Player):

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

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

    def place_piece(self) -> tuple[int, int]:
        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).

```
class MyMinMax(quarto.Player):
    '''My MinMax strategy'''

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.opportunity = {}       #dict key=opportunity level, value= list of tuple of a list of position tuple and int that is charachteristics

    def choose_piece(self) -> int:
        utilities.check_opportunity(self)
        #print(self.opportunity)

        negative_char = []
        positive_char = []
        for e1 in self.opportunity[1]:  #take opportunity level 1 (worse for me)
```

```
            if e1[1] not in negative_char:
                negative_char.append(e1[1])
        for e3 in self.opportunity[3]: #take opportunity level 3
            if e3[1] not  in negative_char:
                negative_char.append(e3[1])
        for e2 in self.opportunity[2]:  #take opportunity level 2 (best for me)
            if e2[1] not in positive_char:
                positive_char.append(e2[1])
        #print("pos ", positive_char)
        #print("neg ", negative_char)

        positive_char = [x for x in positive_char if x not in negative_char] #take only positive char that are not in negative char

        piece_index = utilities.find_piece(self, positive_char, negative_char)
        if piece_index != -1:
                #print("selected piece ", piece_index)
                return piece_index
        else:
            for e4 in self.opportunity[4]: #add level 4 in positive char
                positive_char.append(e4[1])

            positive_char = [x for x in positive_char if x not in negative_char]

            piece_index = utilities.find_piece(self, positive_char, negative_char)
            if piece_index != -1:
                #print("selected piece ", piece_index)
                return piece_index
            else:
                positive_char = range(8) #take all char
                negative_char = [] #reset negative for taking only level 1
                for e1 in self.opportunity[1]:  #take opportunity level 1 (worse for me)
                    if e1[1] not in negative_char:
                        negative_char.append(e1[1])
                positive_char = [x for x in positive_char if x not in negative_char]

                piece_index = utilities.find_piece(self, positive_char, negative_char)
                if piece_index != -1:
                    #print("selected piece ", piece_index)
                    return piece_index



        return random.randint(0, 15)

    def place_piece(self) -> tuple[int, int]: #index are inverted
        #compute opportunity
```

```
        utilities.check_opportunity(self)
        #print(self.opportunity)

        #take selected piece
        piece_index = self.get_game().get_selected_piece()
        piece = self.get_game().get_piece_charachteristics(piece_index)
        #print("index piece ", piece_index)
        piece_char = []

        #take piece char
        if piece.HIGH == True:
            piece_char.append(0)
        else:
            piece_char.append(4)
        if piece.COLOURED == True:
            piece_char.append(1)
        else:
            piece_char.append(5)
        if piece.SOLID == True:
            piece_char.append(2)
        else:
            piece_char.append(6)
        if piece.SQUARE == True:
            piece_char.append(3)
        else:
            piece_char.append(7)
        #print("piece char ", piece_char)


        positive_op = []
        for e1 in self.opportunity[1]:  #take opportunity level 1 (best for me)
            if e1 not in positive_op:
                positive_op.append(e1)
        #print(positive_op)
        #loop over level 1 opportunity until found one with char of selected piece
        for op in positive_op:
            if op[1] in piece_char:
                return op[0][0][1], op[0][0][0]

        #check if need a block and try to block
        if len(positive_op) > 0:
            move = utilities.block_next(self, piece_index)
            if move != None:
                return move

        positive_op = []
        for e3 in self.opportunity[3]:  #take opportunity level 3 (good for me)
            if e3 not in positive_op:
                positive_op.append(e3)
        negative_op_place = []
        for e2 in self.opportunity[2]:  #take opportunity level 2 (good for my opponent)
            for e2_place in e2[0]:    #take only the places not the
```

```
tuple (list_of_place, char)
                if e2_place not in negative_op_place:
                    negative_op_place.append(e2_place)
        #loop over positive opportunity (l3) checking if match with piece char
        for op in positive_op:
            if op[1] in piece_char:
                for place in op[0]: #loop over opportunity places
                    if place not in negative_op_place: #check if place isn't also a place of opportunity of l2
                        return place[1], place[0]

        #loop over free place
        board = self.get_game().get_board_status()
        for i in range(4):
            for j in range(4):
                if board[i][j] == -1 and (i,j) not in negative_op_place: #check if free place isn't a negative opportunity l2
                    return j, i

        #loop over free place
        for i in range(4):
            for j in range(4):
                if board[i][j] == -1:
                    return j, i      #return first free place found
```

Listing 2: My MinMax Player

2.3 Pastimes Player

Pastimes player his 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 agaist myMinMax the wining rate is closed to the 50%.

```python
class MyPastimes(quarto.Player):
    '''Generalization of ourpastimes.com strategy'''
    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.opportunity = {} #dict key=opportunity level, value=list of tuple of a list of position tuple and int that is charachteristics
        self.pieces = {} #dict: key= piece index, value = list of char

    def choose_piece(self) -> int:
        #compute opportunity
        utilities.check_opportunity(self)
        #print(self.opportunity)
        #take free pieces
        self.pieces = utilities.free_pieces(self)

        #save all free piece with l1 char
        returnable_piece = {} #piece withot char in l1
        for index, piece_char in self.pieces.items():
            not_in = True
            for l1 in self.opportunity[1]: #loop over l1 opp
                if l1[1] in piece_char:
                    not_in = False
            if not_in:
```

```python
                returnable_piece[index] = piece_char #add piece without l1 char


        cont_senza_char = {} #dict where key = char and value are free piece without char
        for l2 in self.opportunity[2]:
            cont_senza_char[l2[1]] = sum(1 for p in self.pieces.values() if l2[1] not in p)
        #for each piece cont favor +1 and sfavor as -1
        score = {}
        for p_index, p_char in returnable_piece.items():
            score[p_index] = 0
            for c, n in cont_senza_char.items():
                if n%2 == 0: #if even char in piece is a favor
                    if c in p_char:
                        score[p_index] += 1
                    else:
                        score[p_index] -= 1
                else:      #if odd nor char in piece is a favor
                    if c not in p_char:
                        score[p_index] += 1
                    else:
                        score[p_index] -= 1
        best_piece_index = None
        for p_index, value in score.items():
            if best_piece_index == None or best_piece_index < value:
                best_piece_index = p_index
        if best_piece_index != None:
            #print("choose from l2 char index: ", best_piece_index)  #return the best if exists

            return best_piece_index

        #return from returnable list if it's not empty or choose from free piece
        if len(returnable_piece) > 0:
            choose = random.choice(list(returnable_piece.keys()))
            #print("choose from returnable :) index: ", choose)
            return choose #return a good piece
        else:
            choose = random.choice(list(self.pieces.keys()))
            #print("choose from free piece :( index: ", choose)
            return choose  #return not a good piece


    def place_piece(self) -> tuple[int, int]:
        #compute opportunity
        utilities.check_opportunity(self)
        #print(self.opportunity)
        #take free pieces
        self.pieces = utilities.free_pieces(self)

        #take selected piece
        piece_index = self.get_game().get_selected_piece()
        piece_char = utilities.get_pieces_char(self, piece_index)
```

```
        #print("piece char ", piece_char)

        #consider first l1 opportunities
        positive_op = []
        for e1 in self.opportunity[1]:  #take opportunity level 1 (best for me)
            if e1 not in positive_op:
                positive_op.append(e1)
        #print(positive_op)
        #loop over level 1 opportunity until found one with char of selected piece
        for op in positive_op:
            if op[1] in piece_char:
                return op[0][0][1], op[0][0][0]


        #check if need a block and try to block
        if len(positive_op) > 0:
            move = utilities.block_next(self, piece_index)
            if move != None:
                return move


        #consider l3 opportunity
        positive_op = []
        for e3 in self.opportunity[3]:  #take opportunity level 3 (good for me)
            if e3 not in positive_op:
                positive_op.append(e3)
        negative_op_place = []
        for e2 in self.opportunity[2]:  #take opportunity level 2 (good for my opponent)
            for e2_place in e2[0]:    #take only the places not the tuple (list_of_place, char)
                if e2_place not in negative_op_place:
                    negative_op_place.append(e2_place)
        #loop over positive opportunity (l3) checking if match with piece char
        for op in positive_op:
            if op[1] in piece_char:
                for place in op[0]: #loop over opportunity places
                    if place not in negative_op_place: #check if place isn't also a place of opportunity of l2
                        return place[1], place[0]

        #loop over free place
        board = self.get_game().get_board_status()
        for i in range(4):
            for j in range(4):
                if board[i][j] == -1 and (i,j) not in negative_op_place: #check if free place isn't a negative opportunity l2
                    return j, i

        #loop over free place
        for i in range(4):
            for j in range(4):
```

```
                if board[i][j] == -1:
                    return j, i      #return first free place found

        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.

```
class Pastimes(quarto.Player):
'''ourpastimes.com strategy'''

def __init__(self, quarto: quarto.Quarto) -> None:
    super().__init__(quarto)
    self.opportunity = {} #dict key=opportunity level, value=list of tuple of a list of position tuple and int that is charachteristics
    self.selected_char = None #select a char

def reset(self):
    '''
    Reset the agent for a new game
    '''

    self.selected_char = random.randint(0, 7)


def choose_piece(self) -> int:
    #print("pastimes choose")
    #check if a new game is started
    if np.all(self.get_game().get_board_status() == -1):
        self.reset()


    #print("sel char ", self.selected_char)
    opposite_char = self.selected_char-4 if self.selected_char > 3 else self.selected_char+4
    utilities.check_opportunity(self)
    #print(self.opportunity)
```

```

        selected_char_l1 = False
        for e1 in self.opportunity[1]:
            if e1[1] == self.selected_char:
                selected_char_l1 = True
                break
        selected_char_l2 = False
        for e2 in self.opportunity[2]:
            if e2[1] == self.selected_char:
                selected_char_l2 = True
                break

        #print("si in l1") if selected_char_l1 else print("no in l1")
        #print("si in l2") if selected_char_l2 else print("no in l2")

        #if there aren't pair (l2) or tripletes (l1) of selected char, return a piece with selected char if doesn't match other l1 char
        if selected_char_l1 == False and selected_char_l2 == False:
            pieces_with_selected_char = utilities.select_pieces(self, self.selected_char)
            #print("piece with selected char ", pieces_with_selected_char)

            returnable = []
            for p in pieces_with_selected_char:
                 if p not in self.get_game().get_board_status() and utilities.check_l1(self, p) == True:
                    returnable.append(p)
            if len(returnable) > 0:
                return random.choice(returnable)       #return a random element from the returnable ones


        #if there are pair(l2) but there aren't tripletes(l1)
        #check number of element without selected char if even(pari)
        #return a piece with selected char, otherwise a piece without it
        if selected_char_l1 == False and selected_char_l2 == True:
            pieces_with_selected_char = utilities.select_pieces(self, self.selected_char)
            #print("opposite char ",opposite_char)

            pieces_without_selected_char_tot = utilities.select_pieces(self, opposite_char)
            pieces_without_selected_char = []
            for p in pieces_without_selected_char_tot: #filter pieces, take only the ones not already in the board
                if p not in self.get_game().get_board_status():
                    pieces_without_selected_char.append(p)
            #print("piece without sel char: ", pieces_without_selected_char)

            if len(pieces_without_selected_char)%2 == 0: #even
```

```
check
                returnable = []
                for p in pieces_with_selected_char:
                    if p not in self.get_game().get_board_status() and utilities.check_l1(self, p) == True:
                        returnable.append(p)
                if len(returnable) > 0:
                    return random.choice(returnable)

            else:
                returnable = []
                for p in pieces_without_selected_char:
                    if p not in self.get_game().get_board_status() and utilities.check_l1(self, p) == True:
                        returnable.append(p)
                if len(returnable) > 0:
                    return random.choice(returnable)


        #if there are tripletes of selected char return a piece without selected char and without l1 char
        returnable = []

        #pieces_without_selected_char = utilities.select_pieces(self, opposite_char) #this worsen the result
        #for p in pieces_without_selected_char:
        for p in range(16):
            if p not in self.get_game().get_board_status() and utilities.check_l1(self, p) == True:
                returnable.append(p)
        if len(returnable) > 0:
            return random.choice(returnable)

        #if there aren't any other posibilities return random
        return random.randint(0, 15)



    def place_piece(self) -> tuple[int, int]:
        #print("pasttimes place")
        if np.all(self.get_game().get_board_status() == -1):
            self.reset()

        #compute opportunity
        utilities.check_opportunity(self)
        #print(self.opportunity)

        #take selected piece
        piece_index = self.get_game().get_selected_piece()
        piece = self.get_game().get_piece_charachteristics(piece_index)
        #print("index piece ", piece_index)
        piece_char = []

        #take piece char
        if piece.HIGH == True:
            piece_char.append(0)
```

```
        else:
            piece_char.append(4)
        if piece.COLOURED == True:
            piece_char.append(1)
        else:
            piece_char.append(5)
        if piece.SOLID == True:
            piece_char.append(2)
        else:
            piece_char.append(6)
        if piece.SQUARE == True:
            piece_char.append(3)
        else:
            piece_char.append(7)
        #print("piece char ", piece_char)


        positive_op = []
        for e1 in self.opportunity[1]:  #take opportunity level 1 (best for me)
            if e1 not in positive_op:
                positive_op.append(e1)
        #print(positive_op)

        #loop over level 1 opportunity until found one with char of selected piece
        for op in positive_op:
            if op[1] in piece_char:
                return op[0][0][1], op[0][0][0]

        #check if need a block and try to block
        if len(positive_op) > 0:
            move = utilities.block_next(self, piece_index)
            if move != None:
                return move


        positive_op = []
        for e3 in self.opportunity[3]:  #take opportunity level 3 (good for me)
            if e3 not in positive_op:
                positive_op.append(e3)
        negative_op_place = []
        for e2 in self.opportunity[2]:  #take opportunity level 2 (good for my opponent)
            for e2_place in e2[0]:    #take only the places not the tuple (list_of_place, char)
                if e2_place not in negative_op_place:
                    negative_op_place.append(e2_place)
        #loop over positive opportunity (l3) checking if match with piece char
        for op in positive_op:
            if op[1] in piece_char:
                for place in op[0]: #loop over opportunity places
                    if place not in negative_op_place: #check if place isn't also a place of opportunity of l2
                        return place[1], place[0]
```

```

        #loop over free place
        board = self.get_game().get_board_status()
        for i in range(4):
            for j in range(4):
                if board[i][j] == -1 and (i,j) not in negative_op_place: #check if free place isn't a negative opportunity l2
                    return j, i

        #loop over free place
        for i in range(4):
            for j in range(4):
                if board[i][j] == -1:
                    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.

```
NUM_TRAINING_MATCH = 10000
NUM_EVAL_MATCH = 100

class ReinforcementLearning(quarto.Player):
    '''Reinforcement Learning Agent'''

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.learning = False
        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
```

```python
        self.current = dict() #dict for saving place and piece of the current game
        self.random_factor = 1 #close to 1 -> more exploration, close to 0 more exploitation
        self.num_match = 0
        self.tot_num_matches = NUM_TRAINING_MATCH

    def set_learning(self, value: bool):
        '''set RF agent in learning mode if value equal true, otherwise set it on evaluation mode.'''
        self.learning = value

    def save_knowledge(self, win):
        '''save the move made in this game scoring accordinng with the outcome (win)'''

        #value is a dict that contains choosen piece or place piece
        for board, value in self.current.items(): #loop over all the board status in current dict

            if board in self.knowledge: #check if board already exist in the dict

                if "choose_piece" in value: #check if value is choose_piece type
                    not_in = True
                    for element in self.knowledge[board]["choose_piece"]:
                        if element[0] == value["choose_piece"]: #if piece already exists for board status in knowledge dict
                            not_in = False
                            element[1]+= 1 if win else -1 #update it
                    if not_in: #otherwise add it and initialize
                        self.knowledge[board]["choose_piece"].append([value["choose_piece"], 1 if win else -1])

                else: #check if value is place_piece type
                    not_in = True
                    for element in self.knowledge[board]["place_piece"]:
                        if element[0] == value["place_piece"]: #if place already exists for board status in knowledge dict
                            not_in = False
                            element[1]+= 1 if win else -1 #update it
                    if not_in: #otherwise add it and initialize
                        self.knowledge[board]["place_piece"].append([value["place_piece"], 1 if win else -1])

            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
                self.knowledge[board] = {"choose_piece": list(), "place_piece": list()}
                if "choose_piece" in value: #and add piece or place in the right list, with score according to the outcomes
```

```
                    self.knowledge[board]["choose_piece"].append([value["choose_piece"], 1 if win else -1])
                else:
                    self.knowledge[board]["place_piece"].append([value["place_piece"], 1 if win else -1])

        self.num_match +=1 #update number of played matches
        self.random_factor = 1- 2*(self.num_match/self.tot_num_matches) #update random factor for encrease the exploitation with match prograssion
        self.current = dict() #reset the current dict

    def choose_piece(self) -> int:
        board = self.get_game().get_board_status()
        free_pieces = list(utilities.free_pieces(self).keys())
        choose = random.choice(free_pieces)

        #if random_factor > random(0,1) -> exploration otherwise exploitation
        if self.learning and random.random() < self.random_factor: #if agent is set in learning return a random piece from the free ones
            self.current[np.array2string(board)] = {"choose_piece": choose} #and save the choose in the current dict
            return choose

        else: #if agent is in eval mode select the piece with the highest score (>0) if it exists
            best = None
            if np.array2string(board) in self.knowledge:
                #piece_score is a tuple (piece_index, score)
                for piece_score in self.knowledge[np.array2string(board)]["choose_piece"]:
                    if best == None or best[1] < piece_score[1]:
                        best = piece_score
                if best != None and best[1] > 0: #check if exists a piece with score greater than 0
                    return best[0]
            return choose #if not exist return a random piece

    def place_piece(self) -> tuple[int, int]:

        board = self.get_game().get_board_status()
        free_place = utilities.free_place(self)
        choose = random.choice(free_place)

        #if random_factor > random(0,1) -> exploration otherwise exploitation
        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
            self.current[np.array2string(board)] = {"place_piece": choose} #and save the choose in the current dict
            return choose[1], choose[0]

        else:   #if agent is in eval (or exploitation) mode select the place with the highest score (>0)  if it exists
```

```
            best = None
            if np.array2string(board) in self.knowledge:
                #place_score is a tuple (place_tuple, score)
                for place_score in self.knowledge[np.array2string(board)]["place_piece"]:
                    if best == None or best[1] < place_score[1]:
                        best = place_score
                if best != None and best[1] > 0: #check if exists a place with score greater than 0
                    return best[0][1], best[0][0]
            return choose[1], choose[0] #if not exist return a random place from the free ones

#
# TRAINING STUFF
#

def play_n_game_train(game: quarto.Quarto, RF: ReinforcementLearning, player2: quarto.Player, n: int):
    '''
    Play n games  for training player1 (Reinforcement Learning) against player2, print the winner ratio of player1 over player2, switching the starter at each game
    '''

    win_count = 0
    last_start = 1
    for i in range(n):
        game.reset()

        if last_start == 1:
            game.set_players((RF, player2))
            last_start = 0
        else:
            game.set_players((player2, RF))
            last_start = 1

        winner = game.run()

        if (winner == 0 and last_start == 0) or (winner == 1 and last_start == 1): #player1 win
            win_count+=1
            RF.save_knowledge(True)
        else:
            RF.save_knowledge(False)

    logging.warning(f"main: Winner ratio of RF during training: {win_count/n}")

def play_n_game(game: quarto.Quarto, RF: ReinforcementLearning, player2: quarto.Player, n: int):
    '''
    Play n games player1 (Reinforcement Learning) against player2, print the winner ratio of player1 over player2, switching the starter at each game
    '''

```

```
    win_count = 0
    last_start = 1
    for i in range(n):
        game.reset()

        if last_start == 1:
            game.set_players((RF, player2))
            last_start = 0
        else:
            game.set_players((player2, RF))
            last_start = 1

        winner = game.run()

        if (winner == 0 and last_start == 0) or (winner == 1 and last_start == 1): #player1 win
            win_count+=1

    logging.warning(f"main: Winner ratio of RF evaluation training: {win_count/n}")

def training():
    '''
    training the RF agent and evaluete it
    '''
    game = quarto.Quarto()
    agentReinLear = ReinforcementLearning(game)
    agentReinLear.set_learning(True)
    play_n_game_train(game, agentReinLear, main.RandomPlayer(game), NUM_TRAINING_MATCH)
    agentReinLear.set_learning(False)
    play_n_game(game, agentReinLear, main.RandomPlayer(game), 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%.

```
class GeneticAlgorithm(quarto.Player):
'''Genetic Algorithm agent'''

def __init__(self, quarto: quarto.Quarto) -> None:
    super().__init__(quarto)
    self.genome = None      #genome of the current GA agent
    self.opportunity = {}   #dict key=opportunity level, value=
 list of tuple of a list of position tuple and int that is
charachteristics

def set_genome(self, genome):
```

```python
        '''set the genome for the current GA agent'''
        self.genome = genome

    def choose_piece(self) -> int:
        utilities.check_opportunity(self)

        level = math.ceil(self.genome[0]*4) #level of opportunity take in consideration
        chars = {}      #dict with key the char and value the number of time this char appear on selected value
        for _, char in self.opportunity[level]:
            if char not in chars:
                chars[char] = 1
            else:
                chars[char] += 1

        free_piece = utilities.free_pieces(self)
        best_fit = None
        worse_fit = None
        for p_index, p_char in free_piece.items():  #for each free piece
            score = 0                                  #compute a score as +1 if piece char is in selected level char
            for char in p_char:
                score += chars[char] if char in chars else 0
            if worse_fit == None or worse_fit[1] > score:
                worse_fit = (p_index, score)    #save the worse
            if best_fit == None or best_fit[1] < score:
                best_fit = (p_index, score)     #save the best

        if self.genome[1] < 0.5:    #return best char if exist
            if best_fit != None:
                return best_fit[0]
        else:                       #return worse char if exist
            if worse_fit != None:
                return worse_fit[0]

        #otherwise return random piece
        return random.randint(0, 15)

    def place_piece(self) -> tuple[int, int]:
        utilities.check_opportunity(self)

        #take selected piece
        piece_index = self.get_game().get_selected_piece()
        piece_char = utilities.get_pieces_char(self, piece_index)
        level = math.ceil(self.genome[2]*4)  #level of opportunity take in consideration

        position_pos = []                            #list of place with char in the selected piece
        position_neg = utilities.free_place(self)   #list of palce without char of selected piece
        for pos, char in self.opportunity[level]:
            for p in pos:       #for each place in opportunity of selected level
                if char in piece_char:
```

```
                    position_pos.append(p)  #append place in positive if char match with selected piece
                    if p in position_neg:
                        position_neg.remove(p) #remove from negative if char not match with selected piece

        if self.genome[3] < 0.5:            #return one of the positive if exist
            if len(position_pos) > 0:
                choice = random.choice(position_pos)
                return choice[1], choice[0]
        else:                               #return one of the negative if not exist
            if len(position_neg) > 0:
                choice = random.choice(position_neg)
                return choice[1], choice[0]

        #otherwise return a random place
        return random.randint(0, 3), random.randint(0, 3)



'''
GENOME MEANING:
    genome[0] = select level of the consider opportunity for choose piece, more high -> more high level
    genome[1] = for choose piece, if higher than 0.5 try to destroy the opportunity otherwise try to use it
    genome[2] = select level of the consider opportunity for choose place piece, more high -> more high level
    genome[3] = for place piece, if higher than 0.5 try to destroy the opportunity otherwise try to use it
'''

NUM_TRAINING_MATCH = 10    #to compute fitness
NUM_EVAL_MATCH = 1000       #to evaluate the best genome
NUM_GENERATION = 10
POPULATION_SIZE = 100
OFFSPRING_SIZE = 10

def fitness(genome):
    '''Compute the fitness of the genome as the winning ratio agaisnt random player'''
    game = quarto.Quarto()
    agent = GeneticAlgorithm(game)
    agent.set_genome(genome)
    opponent = main.RandomPlayer(game)
    return play_n_game(game, agent, opponent, NUM_TRAINING_MATCH)

def generatePopulation():
    '''Return list of population,  one individual is a tuple of genome (list of genes) and his fitness'''
    population = list()
    for genome in range(POPULATION_SIZE):
        genome = (random.random(), random.random(), random.random(), random.random())
        population.append((genome, fitness(genome)))
```

```python
    return population

def mutation(g):
    '''Mutation change ramdonly a ramdom gene in the genome'''
    point = random.randint(0,len(g)-1)
    return g[:point] + (random.random(),) + g[point+1:]

def select_parent(population, tornament_size=10):
    '''select parent using a k = 10 tornament, takeing the one with best fitness'''
    return max(random.choices(population, k=tornament_size), key=lambda i: i[1])

def GA():
    '''Run 10 generation for find the best genome'''
    best_sol = None

    population = generatePopulation()
    for generation in range(10):
        offsprings = list()
        for i in range(OFFSPRING_SIZE): #in each generation create OFFSPRING_SIZE offspring by mutation parents
            o = ()
            p = select_parent(population)
            o = mutation(p[0])
            offsprings.append((o, fitness(o)))
        population = population + offsprings
        population = sorted(population, key=lambda i:i[1], reverse=True)[:POPULATION_SIZE]

    best_sol = population[0][0]
    return best_sol


def play_n_game(game: quarto.Quarto, GA: GeneticAlgorithm, player2: quarto.Player, n: int):
    '''
    Play n games player1 (Genetic Algorithm) against player2, print the winner ratio of player1 over player2, switching the starter at each game
    '''

    win_count = 0
    last_start = 1
    for i in range(n):
        game.reset()

        if last_start == 1:
            game.set_players((GA, player2))
            last_start = 0
        else:
            game.set_players((player2, GA))
            last_start = 1

        winner = game.run()

        if (winner == 0 and last_start == 0) or (winner == 1 and
```

```
    last_start == 1): #player1 win
            win_count+=1

    #logging.warning(f"main: Winner ratio of GA evaluation training: {win_count/n}")
    return win_count/n

def training():
    '''
    training the GA agent and evaluete it
    '''
    best_genome = GA()
    game = quarto.Quarto()
    agentGen = GeneticAlgorithm(game)
    agentGen.set_genome(best_genome)
    result = play_n_game(game, agentGen, main.RandomPlayer(game), NUM_EVAL_MATCH)
    print(f"main: Winner ratio of GA: {result}, with genome: {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 codede 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

```python
    def block_next(self, sel_piece_index) -> tuple[int, int]:
        '''
        Check if next turn, I have to choose a piece that let my opponent win.
        In this case return a position when place piece to block the winning.
        Otherwise return None
        '''
        sel_piece = self.get_game().get_piece_charachteristics(sel_piece_index)

        positive_char_opponent = {} #dict where key is l1 char and value the number of place
        for e1 in self.opportunity[1]:
            if e1[1] not in positive_char_opponent:
                positive_char_opponent[e1[1]] = 1
            else:
                 positive_char_opponent[e1[1]] += 1

        #print("in block", positive_char_opponent)

        #take all piece indexes not already placed in the board
        free_pieces = list(range(16))
        free_pieces.remove(sel_piece_index) #remove selected piece
```

```
    for r in self.get_game().get_board_status():
        for p in r:
            if p != -1:
                free_pieces.remove(p)

    for p in free_pieces:
        match = False
        for c in positive_char_opponent:
            if c == 0:
                if self.get_game().get_piece_charachteristics(p).HIGH == True:
                    match = True
            elif c == 1:
                if self.get_game().get_piece_charachteristics(p).COLOURED == True:
                    match = True
            elif c == 2:
                if self.get_game().get_piece_charachteristics(p).SOLID == True:
                    match = True
            elif c == 3:
                if self.get_game().get_piece_charachteristics(p).SQUARE == True:
                    match = True
            elif c == 4:
                if self.get_game().get_piece_charachteristics(p).HIGH == False:
                    match = True
            elif c == 5:
                if self.get_game().get_piece_charachteristics(p).COLOURED == False:
                    match = True
            elif c == 6:
                if self.get_game().get_piece_charachteristics(p).SOLID == False:
                    match = True
            elif c == 7:
                if self.get_game().get_piece_charachteristics(p).SQUARE == False:
                    match = True

        if match == False: #find a piece that doesn't match
            #print("find piece not match ", p)
            return None   #no need block, find a piece without char in l1

    #search char with one place
    blockable_char = []
    for c in positive_char_opponent:
        if positive_char_opponent[c] == 1: #try to block char c if have one place
            blockable_char.append(c)
    #print("blockable char ", blockable_char)



    for b_c in blockable_char:
```

```
            for l1 in self.opportunity[1]:
                if l1[1] == b_c:
                    place = l1[0][0]
                    if simulation(self, place, sel_piece_index, free_pieces):
                         return place[1], place[0]
        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

```python
import logging
import argparse
import random
import quarto
import numpy as np
import operator as op

def get_pieces_char(agent, index):
    '''
    Return array of the characteristic of the index piece
    '''
    piece = agent.get_game().get_piece_charachteristics(index)
    #print("index piece ", piece_index)
    piece_char = []

    #take piece char
    if piece.HIGH == True:
        piece_char.append(0)
    else:
        piece_char.append(4)
    if piece.COLOURED == True:
        piece_char.append(1)
    else:
        piece_char.append(5)
    if piece.SOLID == True:
        piece_char.append(2)
    else:
        piece_char.append(6)
    if piece.SQUARE == True:
        piece_char.append(3)
    else:
        piece_char.append(7)

    #print("piece char ", piece_char)
    return piece_char


def char_l1(self):
    '''
    return a list of l1 char
    '''
    array = list()
    for l1 in self.opportunity[1]:
        if l1[1] not in array:
            array.append(l1[1])
    return array

def free_pieces(agent):
    '''
    Return a dict of free piece indexes as value and array of
```

```
    charateristic as value
    '''
    board = agent.get_game().get_board_status()
    pieces = dict()
    for p in range(16):
        if p not in board:
            pieces[p] = get_pieces_char(agent, p)

    return pieces

def free_place(agent):
    '''
    Return a list tuple of free place in the board
    '''
    board = agent.get_game().get_board_status()
    free_place = []
    for i in range(4):
        for j in range(4):
            if board[i][j] == -1:
                free_place.append((i, j))
    return free_place

def block_next(self, sel_piece_index) -> tuple[int, int]:
        '''
        Check if next turn, I have to choose a piece that let my
    opponent win.
        In this case return a position when place piece to block
    the winning.
        Otherwise return None
        '''
        sel_piece = self.get_game().get_piece_charachteristics(
    sel_piece_index)

        positive_char_opponent = {} #dict where key is l1 char and
    value the number of place
        for e1 in self.opportunity[1]:
            if e1[1] not in positive_char_opponent:
                positive_char_opponent[e1[1]] = 1
            else:
                 positive_char_opponent[e1[1]] += 1

        #print("in block", positive_char_opponent)

        #take all piece indexes not already placed in the board
        free_pieces = list(range(16))
        free_pieces.remove(sel_piece_index) #remove selected piece
        for r in self.get_game().get_board_status():
            for p in r:
                if p != -1:
                    free_pieces.remove(p)

        for p in free_pieces:
            match = False
            for c in positive_char_opponent:
                if c == 0:
                    if self.get_game().get_piece_charachteristics(p
    ).HIGH == True:
```

```
                            match = True
                    elif c == 1:
                        if self.get_game().get_piece_charachteristics(p).COLOURED == True:
                            match = True
                    elif c == 2:
                        if self.get_game().get_piece_charachteristics(p).SOLID == True:
                            match = True
                    elif c == 3:
                        if self.get_game().get_piece_charachteristics(p).SQUARE == True:
                            match = True
                    elif c == 4:
                        if self.get_game().get_piece_charachteristics(p).HIGH == False:
                            match = True
                    elif c == 5:
                        if self.get_game().get_piece_charachteristics(p).COLOURED == False:
                            match = True
                    elif c == 6:
                        if self.get_game().get_piece_charachteristics(p).SOLID == False:
                            match = True
                    elif c == 7:
                        if self.get_game().get_piece_charachteristics(p).SQUARE == False:
                            match = True

            if match == False: #find a piece that doesn't match
                #print("find piece not match ", p)
                return None   #no need block, find a piece without char in l1

        #search char with one place
        blockable_char = []
        for c in positive_char_opponent:
            if positive_char_opponent[c] == 1: #try to block char c if have one place
                blockable_char.append(c)
        #print("blockable char ", blockable_char)



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

        '''
        for c in blockable_char:
```

```
            place = None
            not_in_l2 = True
            for e1 in self.opportunity[1]:
                if e1[1] == c:
                    place = e1[0][0]
            for e2 in self.opportunity[2]:
                for place_2 in e2[0]:
                    if place_2 == place:
                        not_in_l2 = False
            if not_in_l2:
                #print("not il l2, ", place)
                return place[1], place[0]

        if len(blockable_char) > 0:
            place = None
            random_choose = random.choice(blockable_char)
            for e1 in self.opportunity[1]:
                if e1[1] == random_choose:
                    place = e1[0][0]
            #print("place in l2, ", place)
            return place[1], place[0]
        return None  #if there aren't any single place for one char -> unblockable -> return None
        '''

def simulation(self, place, piece_index, free_piece):
    '''
    Simulate placing the selected piece in the selected place.
    If this doesn't compromise nothing return true, otherwise return False.
    '''
    for l2 in self.opportunity[2]:
        if place in l2[0] and l2[1] in get_pieces_char(self, piece_index):
            piece_no_match = False
            for p in free_piece:
                if l2[1] not in get_pieces_char(self, p):
                    piece_no_match = True
            if piece_no_match == False:
                return False
    return True

def check_l1(self, piece) -> bool:
    '''
    Return true if piece doesn't have charateristic in l1, otherwise return false
    '''
    #print("in check l1")
    l1 = []
    for e1 in self.opportunity[1]:
        if e1[1] not in l1:
            l1.append(e1[1])
    if self.get_game().get_piece_charachteristics(piece).HIGH == True:
        if 0 in l1:
            return False
    else:
```

```
        if 4 in l1:
            return False
    if self.get_game().get_piece_charachteristics(piece).COLOURED
    == True:
        if 1 in l1:
            return False
    else:
        if 5 in l1:
            return False
    if self.get_game().get_piece_charachteristics(piece).SOLID ==
    True:
        if 2 in l1:
            return False
    else:
        if 6 in l1:
            return False
    if self.get_game().get_piece_charachteristics(piece).SQUARE ==
    True:
        if 3 in l1:
            return False
    else:
        if 7 in l1:
            return False
    return True

def select_pieces(self, char):
    '''
    return a list of all index of piece with char
    '''
    select_pieces = []
    for i in range(16):
        if char == 0:
            if self.get_game().get_piece_charachteristics(i).HIGH
    == True:
                select_pieces.append(i)
        elif char == 1:
            if self.get_game().get_piece_charachteristics(i).
    COLOURED == True:
                select_pieces.append(i)
        elif char == 2:
            if self.get_game().get_piece_charachteristics(i).SOLID
    == True:
                select_pieces.append(i)
        elif char == 3:
            if self.get_game().get_piece_charachteristics(i).SQUARE
     == True:
                select_pieces.append(i)
        elif char == 4:
            if self.get_game().get_piece_charachteristics(i).HIGH
    == False:
                select_pieces.append(i)
        elif char == 5:
            if self.get_game().get_piece_charachteristics(i).
    COLOURED == False:
                select_pieces.append(i)
        elif char == 6:
            if self.get_game().get_piece_charachteristics(i).SOLID
```

```
== False:
                select_pieces.append(i)
        elif char == 7:
            if self.get_game().get_piece_charachteristics(i).SQUARE == False:
                select_pieces.append(i)

    return select_pieces

def find_piece(self, positive_char, negative_char) -> int:
    '''
    Return the index of a piece that satisfies positive char and doesn't have negative char
    Rturn -1 if there aren't pieces like that
    '''

    # take all pieces not in board
    pieces_not_in_board = [x for x in range(16) if x not in self.get_game().get_board_status()]
    piesces_match_char = []
    for i in pieces_not_in_board:  #take pieces that have at least one positive char
        for c in positive_char:
            if c == 0:
                if self.get_game().get_piece_charachteristics(i).HIGH == True and i not in piesces_match_char:
                    piesces_match_char.append(i)
            elif c == 1:
                if self.get_game().get_piece_charachteristics(i).COLOURED == True and i not in piesces_match_char:
                    piesces_match_char.append(i)
            elif c == 2:
                if self.get_game().get_piece_charachteristics(i).SOLID == True and i not in piesces_match_char:
                    piesces_match_char.append(i)
            elif c == 3:
                if self.get_game().get_piece_charachteristics(i).SQUARE == True and i not in piesces_match_char:
                    piesces_match_char.append(i)
            elif c == 4:
                if self.get_game().get_piece_charachteristics(i).HIGH == False and i not in piesces_match_char:
                    piesces_match_char.append(i)
            elif c == 5:
                if self.get_game().get_piece_charachteristics(i).COLOURED == False and i not in piesces_match_char:
                    piesces_match_char.append(i)
            elif c == 6:
                if self.get_game().get_piece_charachteristics(i).SOLID == False and i not in piesces_match_char:
                    piesces_match_char.append(i)
            elif c == 7:
                if self.get_game().get_piece_charachteristics(i).SQUARE == False and i not in piesces_match_char:
                    piesces_match_char.append(i)
        for c in negative_char:
            if c == 0:
```

```
                if self.get_game().get_piece_charachteristics(i).HIGH == True and i in piesces_match_char:
                    piesces_match_char.remove(i)
            elif c == 1:
                if self.get_game().get_piece_charachteristics(i).COLOURED == True and i in piesces_match_char:
                    piesces_match_char.remove(i)
            elif c == 2:
                if self.get_game().get_piece_charachteristics(i).SOLID == True and i in piesces_match_char:
                    piesces_match_char.remove(i)
            elif c == 3:
                if self.get_game().get_piece_charachteristics(i).SQUARE == True and i in piesces_match_char:
                    piesces_match_char.remove(i)
            elif c == 4:
                if self.get_game().get_piece_charachteristics(i).HIGH == False and i in piesces_match_char:
                    piesces_match_char.remove(i)
            elif c == 5:
                if self.get_game().get_piece_charachteristics(i).COLOURED == False and i in piesces_match_char:
                    piesces_match_char.remove(i)
            elif c == 6:
                if self.get_game().get_piece_charachteristics(i).SOLID == False and i in piesces_match_char:
                    piesces_match_char.remove(i)
            elif c == 7:
                if self.get_game().get_piece_charachteristics(i).SQUARE == False and i in piesces_match_char:
                    piesces_match_char.remove(i)

    if len(piesces_match_char) > 0: #if there are at least one matched element return it
        return piesces_match_char[0]
    return -1 #return -1 if thera isn't any piece that match the char and isn't already place


def save_opportunity(agent, vet, i, verticale, char) -> None:
        free_places = []
        ind_diag = 0
        ind_diag_rev = 3
        for j in range(4):
            if vet[j] == -1: #check free place
                if verticale == 0: #vet is horiz
                    if i == -1: #check if is main diag
                        free_places.append((ind_diag,ind_diag))
                    else:
                        free_places.append((i, j))
                else:
                    if i == -1: #check if is antidiag
                        free_places.append((ind_diag,ind_diag_rev))
                    else:
                        free_places.append((j,i))
            ind_diag+=1
            ind_diag_rev-=1
```

```
        agent.opportunity[len(free_places)].append((free_places, char)) #append tupla in the correct dict list
        #print("save ", (free_places, char))

def check_opportunity(agent) -> None:
        '''
        return a dict with all the opportunity on the board.
        Key is the level of the opportunity
        Value is an arrey of tuple (array of free position of the opp, char of opp)
        '''
        agent.opportunity = {1: [], 2: [], 3: [], 4: []} #reset opportunity vector
        mat = agent.get_game().get_board_status() #get board
        #print("mat in check ")
        #print(mat)


        for i in range(4):
            horiz = mat[i]
            vert = mat[:,i]


            #HORIZ
            #check if in horiz there are not element without char HIGH
            if sum(1 for x in horiz if not agent.get_game().get_piece_charachteristics(x).HIGH and x != -1) == 0:

                save_opportunity(agent, horiz, i, 0, 0)

            #check if in horiz there are not element without char COLOURED
            if sum(1 for x in horiz if not agent.get_game().get_piece_charachteristics(x).COLOURED and x != -1) == 0:

                save_opportunity(agent, horiz, i, 0, 1)

            #check if in horiz there are not element without char SOLID
            if sum(1 for x in horiz if not agent.get_game().get_piece_charachteristics(x).SOLID and x != -1) == 0:

                save_opportunity(agent, horiz, i, 0, 2)

            #check if in horiz there are not element without char SQUARE
            if sum(1 for x in horiz if not agent.get_game().get_piece_charachteristics(x).SQUARE and x != -1) == 0:

                save_opportunity(agent, horiz, i, 0, 3)

            #check if in horiz there are not element with char HIGH -> are all low
            if sum(1 for x in horiz if agent.get_game().get_piece_charachteristics(x).HIGH and x != -1) == 0:

```

```
            save_opportunity(agent, horiz, i, 0, 4)

        #check if in horiz there are not element with char COLOURED -> are all WHITE
        if sum(1 for x in horiz if agent.get_game().get_piece_charachteristics(x).COLOURED and x != -1) == 0:

            save_opportunity(agent, horiz, i, 0, 5)

        #check if in horiz there are not element with char SOLID -> are all holled
        if sum(1 for x in horiz if agent.get_game().get_piece_charachteristics(x).SOLID and x != -1) == 0:

            save_opportunity(agent, horiz, i, 0, 6)

        #check if in horiz there are not element with char SQUARE -> are all CIRCUL
        if sum(1 for x in horiz if agent.get_game().get_piece_charachteristics(x).SQUARE and x != -1) == 0:

            save_opportunity(agent, horiz, i, 0, 7)


        #VERT
        #check if in vert there are not element without char HIGH
        if sum(1 for x in vert if not agent.get_game().get_piece_charachteristics(x).HIGH and x != -1) == 0:

            save_opportunity(agent, vert, i, 1, 0)

        #check if in vert there are not element without char COLOURED
        if sum(1 for x in vert if not agent.get_game().get_piece_charachteristics(x).COLOURED and x != -1) == 0:

            save_opportunity(agent, vert, i, 1, 1)

        #check if in vert there are not element without char SOLID
        if sum(1 for x in vert if not agent.get_game().get_piece_charachteristics(x).SOLID and x != -1) == 0:

            save_opportunity(agent, vert, i, 1, 2)

        #check if in vert there are not element without char SQUARE
        if sum(1 for x in vert if not agent.get_game().get_piece_charachteristics(x).SQUARE and x != -1) == 0:

            save_opportunity(agent, vert, i, 1, 3)

        #check if in vert there are not element with char HIGH -> are all low
        if sum(1 for x in vert if agent.get_game().get_piece_charachteristics(x).HIGH and x != -1) == 0:
```

```

                save_opportunity(agent, vert, i, 1, 4)

            #check if in vert there are not element with char COLOURED -> are all WHITE
            if sum(1 for x in vert if agent.get_game().get_piece_charachteristics(x).COLOURED and x != -1) == 0:

                save_opportunity(agent, vert, i, 1, 5)

            #check if in vert there are not element with char SOLID -> are all holled
            if sum(1 for x in vert if agent.get_game().get_piece_charachteristics(x).SOLID and x != -1) == 0:

                save_opportunity(agent, vert, i, 1, 6)

            #check if in vert there are not element with char SQUARE -> are all CIRCUL
            if sum(1 for x in vert if agent.get_game().get_piece_charachteristics(x).SQUARE and x != -1) == 0:

                save_opportunity(agent, vert, i, 1, 7)


        diag = mat.diagonal() #take main diagonal
        #Diag
        #check if in diag there are not element without char HIGH
        if sum(1 for x in diag if not agent.get_game().get_piece_charachteristics(x).HIGH and x != -1) == 0:

            save_opportunity(agent, diag, -1, 0, 0)

        #check if in diag there are not element without char COLOURED
        if sum(1 for x in diag if not agent.get_game().get_piece_charachteristics(x).COLOURED and x != -1) == 0:

            save_opportunity(agent, diag, -1, 0, 1)

        #check if in diag there are not element without char SOLID
        if sum(1 for x in diag if not agent.get_game().get_piece_charachteristics(x).SOLID and x != -1) == 0:

            save_opportunity(agent, diag, -1, 0, 2)

        #check if in diag there are not element without char SQUARE
        if sum(1 for x in diag if not agent.get_game().get_piece_charachteristics(x).SQUARE and x != -1) == 0:

            save_opportunity(agent, diag, -1, 0, 3)

        #check if in diag there are not element with char HIGH -> are all low
        if sum(1 for x in diag if agent.get_game().get_piece_charachteristics(x).HIGH and x != -1) == 0:

```

```
        save_opportunity(agent, diag, -1, 0, 4)

    #check if in diag there are not element with char COLOURED -> are all WHITE
    if sum(1 for x in diag if agent.get_game().get_piece_charachteristics(x).COLOURED and x != -1) == 0:

        save_opportunity(agent, diag, -1, 0, 5)

    #check if in diag there are not element with char SOLID -> are all holled
    if sum(1 for x in diag if agent.get_game().get_piece_charachteristics(x).SOLID and x != -1) == 0:

        save_opportunity(agent, diag, -1, 0, 6)

    #check if in diag there are not element with char SQUARE -> are all CIRCUL
    if sum(1 for x in diag if agent.get_game().get_piece_charachteristics(x).SQUARE and x != -1) == 0:

        save_opportunity(agent, diag, -1, 0, 7)

    diag = np.fliplr(mat).diagonal() #take anti diagonal
    if sum(1 for x in diag if not agent.get_game().get_piece_charachteristics(x).HIGH and x != -1) == 0:

        save_opportunity(agent, diag, -1, 1, 0)

    #check if in diag there are not element without char COLOURED
    if sum(1 for x in diag if not agent.get_game().get_piece_charachteristics(x).COLOURED and x != -1) == 0:

        save_opportunity(agent, diag, -1, 1, 1)

    #check if in diag there are not element without char SOLID
    if sum(1 for x in diag if not agent.get_game().get_piece_charachteristics(x).SOLID and x != -1) == 0:

        save_opportunity(agent, diag, -1, 1, 2)

    #check if in diag there are not element without char SQUARE
    if sum(1 for x in diag if not agent.get_game().get_piece_charachteristics(x).SQUARE and x != -1) == 0:

        save_opportunity(agent, diag, -1, 1, 3)

    #check if in diag there are not element with char HIGH -> are all low
    if sum(1 for x in diag if agent.get_game().get_piece_charachteristics(x).HIGH and x != -1) == 0:

        save_opportunity(agent, diag, -1, 1, 4)

    #check if in diag there are not element with char COLOURED -> are all WHITE
```

```
        if sum(1 for x in diag if agent.get_game().
    get_piece_charachteristics(x).COLOURED and x != -1) == 0:

            save_opportunity(agent, diag, -1, 1, 5)

        #check if in diag there are not element with char SOLID ->
    are all holled
        if sum(1 for x in diag if agent.get_game().
    get_piece_charachteristics(x).SOLID and x != -1) == 0:

            save_opportunity(agent, diag, -1, 1, 6)

        #check if in diag there are not element with char SQUARE ->
     are all CIRCUL
        if sum(1 for x in diag if agent.get_game().
    get_piece_charachteristics(x).SQUARE and x != -1) == 0:

            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.

```
import random

def problem(N, seed=42):
random.seed(seed)
return [
    list(set(random.randint(0, N - 1) for n in range(random.
randint(N // 5, N // 2))))
    for n in range(random.randint(N, N * 5))
]

from select import select

class MyNode:
    weight = 0
    lists = []
    numbers = []

    def __init__(self, weight = 0, lists = [], numbers = []):
        self.weight = weight
        self.lists = lists
        self.numbers = numbers

    def calculate_weight(self):
        weight = 0
        for inner_list in self.lists:
            for x in inner_list:
                if x in self.numbers:
```

```
                        weight+=1
                    else:
                        self.numbers.append(x)
            self.weight = weight

        def check_sol(self, N):
            sol = True
            for i in range(N):

                if i not in self.numbers:
                    sol = False
            return sol


    def get_min_node(opened_list, N):
        min_weigth = -1
        sel_node = []
        for node in opened_list:
            if min_weigth == -1 or min_weigth > node.weight:
                min_weigth = node.weight
                sel_node = node
        return sel_node

    def get_child(select_nodes, listOfList, opened_list):
        for l in listOfList:
            if l not in select_nodes.lists:
                new_list = list()
                for l2 in select_nodes.lists:
                    new_list.append(l2)
                new_list.append(l)
                node = MyNode(0, new_list, list())
                node.calculate_weight()
                opened_list.append(node)

        return opened_list



    def UCF(listOfList, N):

        first_node = MyNode()
        opened_list = list()
        closed_list = list()
        opened_list.append(first_node)

        num_visited_nodes = 0
        while True:
            select_node = get_min_node(opened_list, N)
            num_visited_nodes+=1
            if select_node.check_sol(N):
                print("sol: ", select_node.lists)
                print("weight: ", select_node.weight)
                print("number of nodes visited: ",
    num_visited_nodes)
                return
            closed_list.append(select_node)
            opened_list.remove(select_node)
```

```
            opened_list = get_child(select_node, listOfList,
    opened_list)
            #print("after child gen ----")
            #for c in opened_list:
            #    print(c.lists)
            #    print("weight: ", c.weight)




def main(N):
    listOfList = problem(N)
    print(listOfList)
    print("start UCF")
    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 in 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.

```
import logging
from collections import namedtuple
import random
from itertools import compress
import sys

POPULATION_SIZE = 200
OFFSPRING_SIZE = 50
N = 50

def problem(seed=42):
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.
    randint(N // 5, N // 2))))
        for n in range(random.randint(N, N * 5))
    ]

def takeList(listOfList, mask): #given a mask array of 1 and 0 get
    the corresponding list in listOfList
    return list(compress(listOfList, mask))

def fitness(genome, listOfList): # calcolate the fisness
    lists = takeList(listOfList, genome)
    cov = list()
    fitness = 0
    for innerList in lists:
        for x in innerList:

            if x in cov:
                fitness-=1
            else:
                fitness+=1
                cov.append(x)
    return fitness      #fitness = num. of set covering - num. of
    duplicates

def generatePopulation(SIZE, listOfList): #return population, one
    individual is a tuple of a mask array of the list taken and his
     fitness
    population = list()
    for genome in [tuple([random.choice([1,0]) for _ in range(SIZE)
    ]) for __ in range(POPULATION_SIZE)]:
```

```
        population.append((genome, fitness(genome, listOfList)))
    return population

def select_parent(population, tornament_size=10):
    return max(random.choices(population, k=tornament_size), key=lambda i: i[1])

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

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

def check_sol(sol): #check if a genoma is a solution
    cov = list()
    for innerList in sol:
        for x in innerList:
            if x not in cov:
                cov.append(x)
    if len(cov) == N:
        return True
    else:
        return False

def GA(listOfList):
    best_sol = None
    best_sol_fit = None

    population = generatePopulation(len(listOfList), listOfList)
    for generation in range(100):
        offsprings = list()
        for i in range(OFFSPRING_SIZE):
            o = ()
            if random.random() < 0.5:
                p = select_parent(population)
                o = mutation(p[0])
            else:
                p1 = select_parent(population)
                p2 = select_parent(population)
                o = cross_over(p1[0],p2[0])
            offsprings.append((o, fitness(o, listOfList)))
        population = population + offsprings
        population = sorted(population, key=lambda i:i[1], reverse=True)[:POPULATION_SIZE]
        for i in population:
            genome_sol = takeList(listOfList, i[0])
            if check_sol(genome_sol):
                fit_of_sol = fitness(i[0], listOfList)
                if best_sol_fit is not None:
                    if fit_of_sol > best_sol_fit:
                            best_sol = genome_sol
                            best_sol_fit = fit_of_sol
                else:
                    best_sol = genome_sol
```

```
                    best_sol_fit = fit_of_sol
                break

    print("after 100 generations")
    print("best solution:")
    print(best_sol)
    print(best_sol_fit)


def main():
    listOfList = problem()
    print(listOfList)
    print("start GA")
    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 form w = 2388 to w = 1557 for N = 100)
- Instead of iterate on the population and the 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 enougth.

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 helps 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 Reviw 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 shortes row. This strategy lose agaist grabriele one, but perform better than pure random one.

```python
def nim_sum(state: Nim) -> int:
    *_, result = accumulate(state.rows, xor)
    return result


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

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

    return cooked
def pure_random(state: Nim) -> Nimply:
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    num_objects = random.randint(1, state.rows[row])
    return Nimply(row, num_objects)
def gabriele(state: Nim) -> Nimply:
    """Pick always the maximum possible number of the lowest row"""
    possible_moves = [(r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1)]
    return Nimply(*max(possible_moves, key=lambda m: (-m[0], m[1])))
def enrico(state: Nim) -> Nimply: #my strategy
    data = cook_status(state)
    if data["active_rows_number"] == 1:
        return (data["longest_row"], state._rows[data["longest_row"]])
    else:
        return (data["shortest_row"], state._rows[data["shortest_row"]])
def optimal_strategy(state: Nim) -> Nimply:
    data = cook_status(state)
```

```
    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 aggresive 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 matchs won agains 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 matchs 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. Squilero code `https://github.com/squillero/computational-intelligence/blob/master/2022-23/lab3_nim.ipynb` for the Nim game structure.

```
import logging
from collections import namedtuple
import random
from typing import Callable
from copy import deepcopy
from itertools import accumulate
from operator import xor

Nimply = namedtuple("Nimply", "row, num_objects")
class Nim:
    def __init__(self, num_rows: int, k: int = None) -> None: #k is
     the max number of object that can be removed from a line
        self._rows = [i * 2 + 1 for i in range(num_rows)]
        self._k = k
```

```python

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

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

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

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

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

def nim_sum(state: Nim) -> int:
    *_, result = accumulate(state.rows, xor)
    return result


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

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

    return cooked

def hard_coded_strategy(state: Nim, genome) -> Nimply:
    data = cook_status(state)
    aggressive, pref_long, remove_perc = genome
    if data["active_rows_number"] == 1: #win conditions
        return (data["longest_row"], state._rows[data["longest_row"
    ]])

    if random.random() > 1-aggressive: #follow an aggressive
    strategy remove an entire line
```

```python
        if random.random() > 1-pref_long:
            return (data["longest_row"], state._rows[data["longest_row"]])
        else:
            return (data["shortest_row"], state._rows[data["shortest_row"]])
    else:   #follow an conservative strattegy remove only a few element in a line
        if random.random() > 1-pref_long:
            num_of_rem = round(state._rows[data["longest_row"]]*remove_perc)
            if num_of_rem == 0:
                num_of_rem+=1
            return (data["longest_row"], num_of_rem)
        else:
            num_of_rem = round(state._rows[data["shortest_row"]]*remove_perc)
            if num_of_rem == 0:
                num_of_rem+=1
            return (data["shortest_row"], num_of_rem)
    return pure_random(state)

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

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

        return ply

    return evolvable
NUM_MATCHES = 10
NIM_SIZE = 10


def evaluate(opponent: Callable, genome) -> float:
    won = 0
    last_player_start = 1
    for m in range(NUM_MATCHES):
        nim = Nim(NIM_SIZE)
        player = 1 - last_player_start
        last_player_start = player
        while nim:
            if player == 0:
                ply = opponent(nim)
            else:
                ply = hard_coded_strategy(nim, genome)
            nim.nimming(ply)
            player = 1 - player
        if player == 0:
            won += 1

```

```python
    return won / NUM_MATCHES #percentage of match won agaist the opponent
POPULATION_SIZE = 20
OFFSPRING_SIZE = 10
OPPONENT = gabriele
def fitness(genome): # calcolate the fisness
    return evaluate(OPPONENT, genome)
def generatePopulation(): #return population, one individual is a tuple of a mask array of the list taken and his fitness
    population = list()
    for genome in range(POPULATION_SIZE):
        genome = (random.random(), random.random(), random.random())
        population.append((genome, fitness(genome)))
    return population
def mutation(g):

    point = random.randint(0,len(g)-1)
    return g[:point] + (random.random(),) + g[point+1:]
def select_parent(population, tornament_size=10):
    return max(random.choices(population, k=tornament_size), key=lambda i: i[1])
def GA():
    best_sol = None
    best_sol_fit = None

    population = generatePopulation()
    for generation in range(10):
        offsprings = list()
        for i in range(OFFSPRING_SIZE):
            o = ()
            p = select_parent(population)
            o = mutation(p[0])
            offsprings.append((o, fitness(o)))
        population = population + offsprings
        population = sorted(population, key=lambda i:i[1], reverse=True)[:POPULATION_SIZE]

    best_sol = population[0][0]
    best_sol_fit = population[0][1]
    print("after 100 generations")
    print("best solution:")
    print(best_sol)
    return best_sol
best_sol= GA()
print("win-rate:")
evaluate(gabriele, best_sol)
```

Listing 11: Lab 3: Task 2

3.3.3 Task 3

MinMax Agent

```python
class Agent():
    def __init__(self, state: Nim):
        self.nim = state
        self.tree = None

    def createTree(self):
        creation = True
        self.tree = Node('root', value = list(self.nim.rows))
        parents = [self.tree]
        childs = []
        level = 1
        my_turn = True
        while creation:
            #print("level:", level)
            #print("parents: ", parents)
            count_parents_no_child = 0
            for parent in parents:
                #print(parent)
                moves = [(r, o) for r, c in enumerate(parent.value) for o in range(1, c + 1) ]
                #print("moves", moves)
                if len(moves) == 0:
                    count_parents_no_child+=1
                else:
                    for ply in moves:
                        aus = parent.value.copy()

                        aus[ply[0]] = (aus[ply[0]] - ply[1])
                        child = Node(level,value=aus, after_me=level%2, parent=parent)
                        childs.append(child)
            if count_parents_no_child == len(parents):
                creation = False #no more node to explore
            level+=1
            my_turn != my_turn
            parents = childs
            childs = []
        #print(RenderTree(self.tree))

    def findMybestNextMove(self, root: Node) -> Nimply:
        d = search.findall(root, filter_=lambda node: node.value == [0]*NIM_SIZE and node.after_me == 1)
        best = min(d, key=lambda n: n.name)[0]


        return pure_random(self.nim)

    def minmax(self) -> Nimply:

        if self.tree == None:
            self.createTree()        #create tree for first time
        else:
            for child in self.tree.children:             #updaate tree root with the current nim state
                if child.value == list(self.nim.rows):
                    self.tree = child

```

```python
        self.findMybestNextMove(self.tree)

        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 particolar 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 keed 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.

Depsite 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.

```python
class Agent():
    def __init__(self, nim: Nim, num_tot_matches):
        self.nim = nim          #to be update at each game
        self.random_factor = 1  #at the begining is set to 1 -> 100 explore
        self.learned = dict()    #key is the nim status (nim._rows) the value is a dict of ( key: ply, value: score) from previus games
        self.current_move = dict()  #key is the nim status, value is the ply performed in the current game
        self.num_matches = 0
        self.num_tot_matched = num_tot_matches

    def play(self) -> Nimply:  #return a move
        selected_ply = None
        if random.random() > self.random_factor: # exploitation: select best move in same status situation if exists(score must be grater that 0)
            if self.nim.print_state() in self.learned.keys():
                moves = self.learned[self.nim.print_state()]
                best = None
                max = 0
                for move, score in moves.items():
                    if score > max:

                        best = move
                        max = score

                if best == None:
                    selected_ply = pure_random(self.nim)
                else:
                    selected_ply = best
```

```

            else:
                selected_ply = pure_random(self.nim)
        else:   #exploration
            selected_ply = pure_random(self.nim)

        self.current_move[self.nim.print_state()] = selected_ply

        return selected_ply

    def update_score(self, win):    #in learned update score +1 if agent wins or -1 if loses
        self.random_factor = 1 -2*(self.num_matches/self.num_tot_matched) #update random factor for encrease the exploitation the matches progression

        for nim_state, move in self.current_move.items():
            if nim_state in self.learned.keys():
                if move in self.learned[nim_state].keys():
                    if win:
                        self.learned[nim_state][move]+=1
                    else:
                        self.learned[nim_state][move]-=1
                else:
                    if win:
                        self.learned[nim_state][move] = 1
                    else:
                        self.learned[nim_state][move] = -1
            else:
                if win:
                    self.learned[nim_state] = {move: 1}
                else:
                    self.learned[nim_state] = {move: -1}

NUM_MATCHES_EVAL = 100
NIM_SIZE = 5
NUM_MATCHES_TRAINING = 5000
OPPONENT_TRAIN = [pure_random, enrico, gabriele, optimal_strategy]
OPPONENT_EVAL = enrico
def training(robot):
    won=0
    last_player_start = 1
    for i in range(NUM_MATCHES_TRAINING):
        nim = Nim(NIM_SIZE)
        robot.nim = nim
        robot.num_matches = i
        player = 1 - last_player_start  #for switching the starter
        last_player_start = player
        while nim:
            if player == 0:
                ply = OPPONENT_TRAIN[int(i//(NUM_MATCHES_TRAINING/len(OPPONENT_TRAIN)))](nim)  #select opponent starting from the silliest
            else:
                ply = robot.play()
            nim.nimming(ply)
            player = 1 - player
```

```
        if player == 0: #robot win
            won+=1
            robot.update_score(1)
        else:   #robot lose
            robot.update_score(0)
    print("won in training: ", won)
def evaluate(robot) -> float:
    won = 0
    last_player_start = 1
    for m in range(NUM_MATCHES_EVAL):
        nim = Nim(NIM_SIZE)
        robot.nim = nim
        player = 1 - last_player_start
        last_player_start = player
        while nim:
            if player == 0:
                ply = OPPONENT_EVAL(nim)
            else:
                ply = robot.play()
            nim.nimming(ply)
            player = 1 - player
        if player == 0:
            won += 1

    return won / NUM_MATCHES_EVAL #percentage of match won agaist the opponent
def main():
    robot = Agent(None, NUM_MATCHES_TRAINING)

    training(robot)

    robot.random_factor = 0 #only exploitation
    res = evaluate(robot)
    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 very 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

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 generalizzation 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

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`

In general the code is well organized and the readme file contains usefull explainations, 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