Computational Intelligence - Quarto

Author

Muhammad Sarib Khan s298885

Collaborators

To be completely candid, since there was no one for this session that I could team up with, I contacted my friend, Angelica Ferlin (an erasmus exchange student), who appeared in the January session and discussed my ideas with her.

Angelica Ferlin - Discussed possible solutions/problems and helped me with the management of Q tables

Resources

Links

- https://stackoverflow.com/questions/10016352/convert-numpy-array-to-tuple - If there's an error with the Q-table because numpy arrays are not hashable, you can resolve it by converting the numpy array into a tuple.
- https://stackoverflow.com/questions/4901815/object-of-custom-type-as-dictionary-key - I've found a solution for QTableKey here, where instead of using a tuple, an object is used as the key.
- https://realpython.com/python-is-identity-vs-equality/ - I encountered a memory issue with the QTableKey, and this link clarified that I needed to implement an **eq**() method to enable comparison when hashing.
- Link to final repository: https://github.com/yurnero14/Quarto-Final
- Link to course repo: https://github.com/yurnero14/Computational-Intelligence-Muhammad-Sarib-Khan

Code Development

In the beginning of the project, Angelica did some research on different possible algorithms and Leonor studied the implementation of Minimax. However, after careful consideration, it was decided to implement Reinforcement Learning instead. I took this decision because I am much more confident implementing an RL based solution since I have done a few mini projects on my own in the past based on RL. Additionally, it was anticipated that Minimax would require a significant amount of time, given the large number of possible states in the game of Quarto.

In the process of developing the Reinforcement Learning strategy, a substantial amount of knowledge was gained from the concerned lectures. In all honesty, I am not the best in coding so Angelica helped giving me the direction to develop an algorithm to code for Q-learning.

Regarding the Q-table, the decision was made to combine the selected piece and the placement of the piece into a single move. Consequently, the key for the Q-table was structured as a tuple, incorporating both the current state (comprising the board array and the chosen piece) and the move itself. The Q-values were then associated with this key. To make the current state usable as part of the key, specific hash and equality (eq) functions had to be implemented. This approach was derived from a previously mentioned source.

In addition, following the creation of the RL agent with the goal of training it, another agent was intentionally designed to make poor decisions. This deliberate choice allowed the agent to undergo initial training with the "bad" agent before transitioning to random actions, thereby smoothing the overall learning process.

The ExtendedQuarto class was developed to enhance the integration of the custom-written code with the provided libraries. This class facilitated the ability to switch the current player, a critical aspect for the implemented logic and auxiliary functions to function smoothly.

Following the agent's training, the program stores the results in a file. This document is subsequently utilized by the trained agent class to read the Q-table and apply moves based on the knowledge acquired during training. Unfortunately, some setbacks were encountered due to the size of the file and using the pickle library. This library was generating a file that was too big to simply upload it on github. Hence, by using HFL (Large File Storage), a solution was found.

In addition, in order to get these results, the agent was trained with Genetic Algorith.

Code Map

- *extendedQuarto.py* - An extended version of Quarto was created to incorporate additional functionality and features.
- *testQuarto.py* - An extended version of Quarto was developed to enable more comprehensive move testing and evaluation.
- *rl.py* - A Reinforcement Learning Agent and a corresponding Class designed to be used as a Key for the Q-table were implemented in the project.
- *opponent_agents.py* - An intentionally designed agent was created to deliberately make suboptimal decisions as part of the training or testing process.
- *train_q_learner.py* - includes a function that facilitates running a game between the Q-Learner and an opponent agent. Additionally, there's a strategy in place to guide the Q-Learner's learning process during these games.
- *trained_rl.py* - Class including the trained RL
- *q_table_1.pickle* - The Q-table is saved in a file with the following parameters:
  - Number of games played: 1000
  - Games added per opponent: 500
  - Exploration rate decreases by 0.05 every 100th game.
- *q_table_2.pickle* - The Q-Learner is saved in a file with the specified parameters:
  - Number of games played: 2000

    - Games added per opponent: 700
    - Exploration rate decreases by 0.05 every 200th game.
- *q_table_3.pickle* - The Q-Learner is saved in a file with the following parameters:
    - Number of games played: 3000
    - Games added per opponent: 700
    - Exploration rate decreases by 0.05 every 300th game.

How to run the code

- In train_q_learner.py file, number of games can be set at line 158 of the file (3rd parameter). At line 108, the number of games after which you want to decrease exploration rate can be change
- In rl.py, at line 273, name the pickle file you want to save the q-table in
- in trained_rl, at line 64, keep the name of the file to read same as the last bullet point
- run python main.py

Source code of project:

generate_agent.py:

```
import random
import quarto

class EvolvedAgent(quarto.Player):
    '''Evolved agent using the GA approach'''

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.board = quarto
        self.active_pieces = [i for i in range(16)]

        # Generate the probability of picking each rule for picking and
placing
        pick_prob_not_normalized = [random.random() for i in range(3)]
        pick_prob = [x/sum(pick_prob_not_normalized) for x in
pick_prob_not_normalized]
        self.pick_probability_1 = pick_prob[0]
        self.pick_probability_2 = pick_prob[1]
        self.pick_probability_3 = pick_prob[2]

        place_probability_not_normalized = [random.random() for i in
range(6)]
        place_prob = [x/sum(place_probability_not_normalized) for x in
place_probability_not_normalized]
        self.place_probability_1 = place_prob[0]
        self.place_probability_2 = place_prob[1]
        self.place_probability_3 = place_prob[2]
        self.place_probability_4 = place_prob[3]
        self.place_probability_5 = place_prob[4]
        self.place_probability_6 = place_prob[5]
```

```
        # Create a dictionary containing the attributes of each piece
        self.piece_dict = self.piece_attribute_dict()

        # Create a dictionary containing the dominating lines of each length
        self.dom_line_dict = {}

    def get_pick_prob(self):
        return (self.pick_probability_1, self.pick_probability_2,
self.pick_probability_3)

    def get_place_prob(self):
        return (self.place_probability_1, self.place_probability_2,
self.place_probability_3, \
            self.place_probability_4, self.place_probability_5,
self.place_probability_6)

    def set_pick_prob_1(self, val):
        self.pick_probability_1 = val

    def set_pick_prob_2(self, val):
        self.pick_probability_2 = val

    def set_pick_prob_3(self, val):
        self.pick_probability_3 = val

    def set_place_prob_1(self, val):
        self.place_probability_1 = val

    def set_place_prob_2(self, val):
        self.place_probability_2 = val

    def set_place_prob_3(self, val):
        self.place_probability_3 = val

    def set_place_prob_4(self, val):
        self.place_probability_4 = val

    def set_place_prob_5(self, val):
        self.place_probability_5 = val

    def set_place_prob_6(self, val):
        self.place_probability_6 = val

    def choose_piece(self) -> int:
        '''Decides what rule to be played when picking a piece by the
agent'''

        self.dominating_line()

        # If there are 3 pieces on the same line with at least 1 common
attribute and a 4th free slot
        if len(self.dom_line_dict['3']) > 0:
            return self.desired_piece(self.dom_line_dict['3'])

        # Genereate the random value to decide what simple rule to play
        val = random.random()
```

```
        # Play rule 1
        if val <= self.pick_probability_1:
            #print("Rule 1")
            return self.pick_rule_1()
        # Play rule 2
        elif val <= self.pick_probability_1 + self.pick_probability_2:
            #print("Rule 2")
            return self.pick_rule_2()
        # Play rule 3
        else:
            #print("Rule 3")
            return self.pick_rule_3()

    def place_piece(self) -> tuple([int, int]):
        '''Decides where to place the piece given by the opponent'''

        self.dominating_line()

        # If there are 3 pieces on the same line with at least 1 common
attribute and a 4th free slot
        # and the selected piece shares one of the common attributes
        if len(self.dom_line_dict['3']) > 0:
            for line in self.dom_line_dict['3']:
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        return self.place_piece_specified_line(line)

        # Genereate the random value to decide what simple rule to play
        val = random.random()

        # Play rule 1
        if val <= self.place_probability_1:
            #print("Rule 1")
            return self.place_rule_1()
        # Play rule 2
        elif val <= self.place_probability_1 + self.place_probability_2:
            #print("Rule 2")
            return self.place_rule_2()
        # Play rule 3
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3:
            #print("Rule 3")
            return self.place_rule_3()
        # Play rule 4
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3 + self.place_probability_4:
            #print("Rule 4")
            return self.place_rule_4()
        # Play rule 5
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3 + self.place_probability_4 +
self.place_probability_5:
            #print("Rule 5")
            return self.place_rule_5()
        # Play rule 6
```

```
        else:
            #print("Rule 6")
            return self.place_rule_6()

    def piece_attribute_dict(self) -> dict:
        piece_dict = {}

        for i in range(16):
            piece = self.board.get_piece_charachteristics(i)
            piece_dict[str(i)] = (piece.HIGH, piece.COLOURED, piece.SOLID,
piece.SQUARE)
        return piece_dict

    def dominating_line(self):
        '''Store each row/column/diagonal with at least one shared attribute
in the dom_line_dict based on how long they are.
        Each row, column and diagonal have been assigned a number, where the
rows from top to bottom are
        0 - 3, columns from left to right are 4-7, diagonal 8 and off-
diagonal 9.'''
        board = self.board.get_board_status()

        self.dom_line_dict['1'] = []
        self.dom_line_dict['2'] = []
        self.dom_line_dict['3'] = []
        self.dom_line_dict['4'] = []

        # Check horizontal
        line_counter = 0
        for row in board:
            dominating_result = self.dominating(row, line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check vertical
        for i in range(self.board.BOARD_SIDE):
            dominating_result = self.dominating(board[:,i], line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check diagonal
        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        dominating_result = self.dominating(diag, line_counter)
        if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        dominating_result = self.dominating(off_diag, line_counter+1)
        if dominating_result != None:
```

```
self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        return dominating_result

    def dominating(self, line, line_counter: int) -> list:
        '''Calculate if the current line has a longer line of pieces with at
least one shared attribute
        (higher score) than the previously calculated dominating line. If the
lines contains the same number
        of pieces with at least one shared attribute (same score), then store
both.'''

        line_attributes = []

        # Remove all the empty slots with no piece on it
        if -1 in line:
            line = list(dict.fromkeys(line))
            line.remove(-1)

        # Store the attributes of each piece in the current line
        for elem in line:
            piece = self.board.get_piece_charachteristics(elem)
            line_attributes.append([piece.HIGH, piece.COLOURED, piece.SOLID,
piece.SQUARE])

        # If there are more than 1 piece in the line, check what attributes
are reccuring.
        # Result is a boolean vector where True/False represent the
attributes that are common in the line
        # and None represent an attribute that is not shared. The order is
[High Coloured Solid Square]
        if len(line_attributes) > 1:
            old_result = line_attributes[0]
            for i in range(len(line_attributes)-1):
                result = []
                for j,k in zip(old_result, line_attributes[i+1]):
                    if j==k:
                        result.append(j)
                    else:
                        result.append(None)
                old_result = result
        elif len(line_attributes) == 1:
            # If the line is only 1 piece, all attributes of that piece are
the lines attributes.
            result = line_attributes[0]
        else:
            result = []

        # Check whether the result vector has an element that is not None,
thus yielding a possible new
        # dominating line
        if any(map(lambda x: not x is None, result)):
            return [len(line_attributes), line_counter, result]
        return None

    def check_attributes(self) -> list:
        '''Calculate the number of each attribute on the board.'''
```

```
        board = self.board.get_board_status()
        attribue_values = {}
        attribue_values['High'] = 0
        attribue_values['Low'] = 0
        attribue_values['Color'] = 0
        attribue_values['Noncolor'] = 0
        attribue_values['Solid'] = 0
        attribue_values['Hollow'] = 0
        attribue_values['Square'] = 0
        attribue_values['Circle'] = 0

        for row in board:
            for elem in row:
                if elem >= 0:
                    if self.board.get_piece_charachteristics(elem).HIGH:
                        attribue_values['High'] += 1
                    else:
                        attribue_values['Low'] += 1
                    if self.board.get_piece_charachteristics(elem).COLOURED:
                        attribue_values['Color'] += 1
                    else:
                        attribue_values['Noncolor'] += 1
                    if self.board.get_piece_charachteristics(elem).SOLID:
                        attribue_values['Solid'] += 1
                    else:
                        attribue_values['Hollow'] += 1
                    if self.board.get_piece_charachteristics(elem).SQUARE:
                        attribue_values['Square'] += 1
                    else:
                        attribue_values['Circle'] += 1
        return attribue_values

    def rank_pieces(self, attribute_values: list) -> list:
        '''Ranks the pieces according to their shared attriutes with the
board'''
        val = []

        for i in range(16):
            cur_val = 0
            piece = self.board.get_piece_charachteristics(i)

            if piece.HIGH:
                cur_val += attribute_values['High']
            else:
                cur_val += attribute_values['Low']
            if piece.COLOURED:
                cur_val += attribute_values['Color']
            else:
                cur_val += attribute_values['Noncolor']
            if piece.SOLID:
                cur_val += attribute_values['Solid']
            else:
                cur_val += attribute_values['Hollow']
            if piece.SQUARE:
                cur_val += attribute_values['Square']
            else:
```

```
                cur_val += attribute_values['Circle']
            val.append(cur_val)
        return val

    def desired_piece(self, dom_lines: list) -> list:
        '''This function determines what attributes the piece we choose
should have to avoid a loss'''

        dom_line_attributes = self.dominant_attributes(dom_lines)

        # If one of the attributes and it's corresponding opposite leads to a
win, then we lose no matter what piece we select.
        # E.g. both high and low.
        # If the above is not the case, store the inverse of the attributes
in order to obtain a vector of what attributes
        # we are searching for in a piece. If there's no condition on the
specific attribute, i.e. the piece can be either high or low,
        # store both True and False for the desirec piece in this attribute
spot.
        desired_piece = [[], [], [], []]
        for i in range(len(dom_line_attributes)):
            if len(dom_line_attributes[i]) == 2:
                #print("Lose no matter what")
                return self.pick_rule_3()
            elif len(dom_line_attributes[i]) == 1:
                desired_piece[i].append(not(dom_line_attributes[i][0]))
            else:
                desired_piece[i].append(True)
                desired_piece[i].append(False)

        # Now we look for a piece with the desired attributes
        found, piece_val = self.find_piece_attribute(desired_piece)
        if found:
            #print("Found a desired piece")
            return piece_val
        else:
        # If such a piece does not exist (have been played already), play a
piece at random
            #print("No desired piece left. Picking a piece at random")
            return self.pick_rule_3()

    def dominant_attributes(self, dominant_lines) -> list:
        '''The function returns what are the common attributes in the
dominating lines'''

        dom_line_attributes = [[], [], [], []]
        for i in range(4):
            for elem in dominant_lines:
                if elem[2][i] != None and elem[2][i] not in
dom_line_attributes[i]:
                    dom_line_attributes[i].append(elem[2][i])
        return dom_line_attributes

    def find_piece_attribute(self, desired_piece) -> tuple:
        '''The function looks for a piece with the desired attributes and
returns True and the piece value if found, otherwise
        it returns false and None'''
```

```
        for i in range(16):
            if i not in self.board.get_board_status():
                piece = self.board.get_piece_charachteristics(i)
                if piece.HIGH in desired_piece[0] and piece.COLOURED in
desired_piece[1] \
                    and piece.SOLID in desired_piece[2] and piece.SQUARE in
desired_piece[3]:
                    return (True, i)
        return (False, None)

    def place_piece_specified_line(self, line) -> tuple([int, int]):
        '''This function checks if the selected piece can be placed such that
the agent wins'''
        board = self.board.get_board_status()

        # Check if the piece should be placed in a row
        if line[1] >= 0 and line[1] <= 3:
            # look for an empty spot in the row
            for i in range(4):
                if board[line[1]][i] == -1:
                    return (i, line[1])
        # Check if the piece should be placed in a column
        elif line[1] >= 4 and line[1] <= 7:
            for i in range(4):
                if board[i, line[1]-4] == -1:
                    return (line[1]-4, i)
        # Check if the piece should be placed on the diagonal
        elif line[1] == 8:
            for i in range(4):
                if board[i, i] == -1:
                    return (i, i)
        # Check if the piece should be placed on the off-diagonal
        elif line[1] == 9:
            for i in range(4):
                if board[3-i, i] == -1:
                    return (i, 3-i)

    def count_shared_attributes_in_line(self, piece) -> list:
        '''This function counts the number of shared attributes in the line
with the selected piece. If the piece
        is high and the line contains 2 high pieces, this will count as 2
shared attributes. Thus, longer lines will
        therefore have an advantage in being picked'''
        board = self.board.get_board_status()
        line_val = []

        # For each row
        for row in board:
            line_val.append(self.count_line(row, piece))

        # For each column
        for i in range(self.board.BOARD_SIDE):
            line_val.append(self.count_line(board[:,i], piece))

        # Diagonals
        diag = []
```

```
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        line_val.append(self.count_line(diag, piece))
        line_val.append(self.count_line(off_diag, piece))

        return line_val


    def count_line(self, line, piece) -> int:
        '''Count the number of shared attributes in the line. If there's no
empty spots left in the line, return -1 in order
        to avoid selecting this line'''
        val = -1
        if -1 in line:
            val  = 0
            line = list(dict.fromkeys(line))
            line.remove(-1)

            for elem in line:
                high, color, solid, square = self.piece_dict[str(elem)]
                if high == self.board.get_piece_charachteristics(piece).HIGH:
                    val += 1
                if color ==
self.board.get_piece_charachteristics(piece).COLOURED:
                    val += 1
                if solid ==
self.board.get_piece_charachteristics(piece).SOLID:
                    val += 1
                if square ==
self.board.get_piece_charachteristics(piece).SQUARE:
                    val += 1
        return val

    def length_of_board_lines(self) -> list:
        '''Counts the lengths of all lines with at least 1 free spot'''

        board = self.board.get_board_status()
        length = []

        for row in board:
            length.append(self.length_of_line(row))

        for i in range(self.board.BOARD_SIDE):
            length.append(self.length_of_line(board[:,i]))

        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        length.append(self.length_of_line(diag))
        length.append(self.length_of_line(off_diag))
```

```
        return length

    def length_of_line(self, line) -> int:
        '''Counts the lenght of the current line'''
        if -1 in line:
            line = list(dict.fromkeys(line))
            line.remove(-1)
            return len(line)
        else:
            return self.board.BOARD_SIDE

    def pick_rule_1(self) -> int:
        '''Pick the piece with the most common attributes with the current
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
        max_pos = piece_ranking.index(max(piece_ranking))
        while max_pos in self.board.get_board_status():
            piece_ranking[max_pos] = -1
            max_pos = piece_ranking.index(max(piece_ranking))
        return max_pos


    def pick_rule_2(self) -> int:
        '''Pick the piece with the least common attributes with the current
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
        min_pos = piece_ranking.index(min(piece_ranking))
        while min_pos in self.board.get_board_status():
            piece_ranking[min_pos] = 100
            min_pos = piece_ranking.index(min(piece_ranking))
        return min_pos

    def pick_rule_3(self) -> int:
        '''Pick a piece at random'''
        random_piece = random.randint(0, 15)
        while random_piece in self.board.get_board_status():
            random_piece = random.randint(0, 15)
        return random_piece

    def place_rule_1(self) -> tuple([int, int]):
        '''Place the piece on the longest uninterrupted line that has no
shared attributes with the current piece'''

        # If there is a dominating line with 3 elements in it, we always
place the piece there if it generates a win.
        # Thus, we have already checked that we have no common attributes
with all 3-length dominating lines and can therefore place
        # the piece in this line
        if len(self.dom_line_dict['3']) > 0:
            return
self.place_piece_specified_line(self.dom_line_dict['3'][0])

        # For dominating lines of length less than 3 there first need to be a
check to confirm that there are no shared attributes
        # between the selected piece and the line in order to block it.
```

```
        for length in ['2', '1']:
            for line in self.dom_line_dict[length]:
                call = True
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        call = False
                        break
                if call:
                    return self.place_piece_specified_line(line)

        # If there are no dominating lines of any lenght we can block, place
it randomly
        return self.place_rule_6()

    def place_rule_2(self):
        '''Place the piece on the longest uninterrupted line with a shared
attribute, might not be the dominating line'''

        # There cannot be a uninterrupted line with 3 elements with a shared
attribute with out piece, if that was the
        # case, the piece would already have been placed there!
        # Thus we only need to check uninterrupted lines of length 2 and 1
and see where our piece has a shared attribute

        for length in ['2', '1']:
            for line in self.dom_line_dict[length]:
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        return self.place_piece_specified_line(line)

        # If there are no uninterrupted lines with a shared attribute, place
it randomly
        return self.place_rule_6()

    def place_rule_3(self):
        '''Places the piece on the line with the most shared attributes
independent of length'''

        piece = self.board.get_selected_piece()
        line_values = self.count_shared_attributes_in_line(piece)
        return self.place_piece_specified_line([0,
line_values.index(max(line_values))])

    def place_rule_4(self):
        '''Place the piece on the shortest line, independent of attributes'''

        lengths_on_board = self.length_of_board_lines()
        return self.place_piece_specified_line([0,
lengths_on_board.index(min(lengths_on_board))])

    def place_rule_5(self):
        '''Place the piece on the longest line, independent of attributes'''

        lengths_on_board = self.length_of_board_lines()
```

```
        # Replace all the lines of length 4 with a -1 as we cannot place a
piece on these lines
        if 4 in lengths_on_board:
            for i in range(len(lengths_on_board)):
                if lengths_on_board[i] == 4:
                    lengths_on_board[i] = -1
        return self.place_piece_specified_line([0,
lengths_on_board.index(max(lengths_on_board))])

    def place_rule_6(self):
        '''Place the piece at random'''
        board = self.board.get_board_status()

        x = random.randint(0, 3)
        y = random.randint(0, 3)

        while board[x,y] != -1:
            x = random.randint(0, 3)
            y = random.randint(0, 3)

        return (y, x)
```

generate_agent_dumb.py:

```
import random
import quarto

class DumbAgent(quarto.Player):
    '''Evolved agent using the GA approach'''

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.board = quarto
        self.active_pieces = [i for i in range(16)]

        # Generate the probability of picking each rule for picking and
placing
        pick_prob_not_normalized = [random.random() for i in range(3)]
        pick_prob = [x/sum(pick_prob_not_normalized) for x in
pick_prob_not_normalized]
        self.pick_probability_1 = pick_prob[0]
        self.pick_probability_2 = pick_prob[1]
        self.pick_probability_3 = pick_prob[2]

        place_probability_not_normalized = [random.random() for i in
range(6)]
        place_prob = [x/sum(place_probability_not_normalized) for x in
place_probability_not_normalized]
        self.place_probability_1 = place_prob[0]
        self.place_probability_2 = place_prob[1]
        self.place_probability_3 = place_prob[2]
        self.place_probability_4 = place_prob[3]
        self.place_probability_5 = place_prob[4]
        self.place_probability_6 = place_prob[5]
```

```python
        # Create a dictionary containing the attributes of each piece
        self.piece_dict = self.piece_attribute_dict()

        # Create a dictionary containing the dominating lines of each length
        self.dom_line_dict = {}

    def get_pick_prob(self):
        return (self.pick_probability_1, self.pick_probability_2,
self.pick_probability_3)

    def get_place_prob(self):
        return (self.place_probability_1, self.place_probability_2,
self.place_probability_3, \
            self.place_probability_4, self.place_probability_5,
self.place_probability_6)

    def set_pick_prob_1(self, val):
        self.pick_probability_1 = val

    def set_pick_prob_2(self, val):
        self.pick_probability_2 = val

    def set_pick_prob_3(self, val):
        self.pick_probability_3 = val

    def set_place_prob_1(self, val):
        self.place_probability_1 = val

    def set_place_prob_2(self, val):
        self.place_probability_2 = val

    def set_place_prob_3(self, val):
        self.place_probability_3 = val

    def set_place_prob_4(self, val):
        self.place_probability_4 = val

    def set_place_prob_5(self, val):
        self.place_probability_5 = val

    def set_place_prob_6(self, val):
        self.place_probability_6 = val

    def choose_piece(self) -> int:
        '''Decides what rule to be played when picking a piece by the
agent'''

        self.dominating_line()

        '''# If there are 3 pieces on the same line with at least 1 common
attribute and a 4th free slot
        if len(self.dom_line_dict['3']) > 0:
            return self.desired_piece(self.dom_line_dict['3'])'''

        # Genereate the random value to decide what simple rule to play
        val = random.random()
```

```
        # Play rule 1
        if val <= self.pick_probability_1:
            #print("Rule 1")
            return self.pick_rule_1()
        # Play rule 2
        elif val <= self.pick_probability_1 + self.pick_probability_2:
            #print("Rule 2")
            return self.pick_rule_2()
        # Play rule 3
        else:
            #print("Rule 3")
            return self.pick_rule_3()

    def place_piece(self) -> tuple([int, int]):
        '''Decides where to place the piece given by the opponent'''

        self.dominating_line()

        '''# If there are 3 pieces on the same line with at least 1 common
attribute and a 4th free slot
        # and the selected piece shares one of the common attributes
        if len(self.dom_line_dict['3']) > 0:
            for line in self.dom_line_dict['3']:
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        return self.place_piece_specified_line(line)'''

        # Genereate the random value to decide what simple rule to play
        val = random.random()

        # Play rule 1
        if val <= self.place_probability_1:
            #print("Rule 1")
            return self.place_rule_1()
        # Play rule 2
        elif val <= self.place_probability_1 + self.place_probability_2:
            #print("Rule 2")
            return self.place_rule_2()
        # Play rule 3
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3:
            #print("Rule 3")
            return self.place_rule_3()
        # Play rule 4
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3 + self.place_probability_4:
            #print("Rule 4")
            return self.place_rule_4()
        # Play rule 5
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3 + self.place_probability_4 +
self.place_probability_5:
            #print("Rule 5")
            return self.place_rule_5()
        # Play rule 6
        else:
```

```
            #print("Rule 6")
            return self.place_rule_6()

    def piece_attribute_dict(self) -> dict:
        piece_dict = {}

        for i in range(16):
            piece = self.board.get_piece_charachteristics(i)
            piece_dict[str(i)] = (piece.HIGH, piece.COLOURED, piece.SOLID,
piece.SQUARE)
        return piece_dict

    def dominating_line(self):
        '''Store each row/column/diagonal with at least one shared attribute
in the dom_line_dict based on how long they are.
        Each row, column and diagonal have been assigned a number, where the
rows from top to bottom are
        0 - 3, columns from left to right are 4-7, diagonal 8 and off-
diagonal 9.'''
        board = self.board.get_board_status()

        self.dom_line_dict['1'] = []
        self.dom_line_dict['2'] = []
        self.dom_line_dict['3'] = []
        self.dom_line_dict['4'] = []

        # Check horizontal
        line_counter = 0
        for row in board:
            dominating_result = self.dominating(row, line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check vertical
        for i in range(self.board.BOARD_SIDE):
            dominating_result = self.dominating(board[:,i], line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check diagonal
        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        dominating_result = self.dominating(diag, line_counter)
        if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        dominating_result = self.dominating(off_diag, line_counter+1)
        if dominating_result != None:
```

```
self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        return dominating_result

    def dominating(self, line, line_counter: int) -> list:
        '''Calculate if the current line has a longer line of pieces with at
least one shared attribute
        (higher score) than the previously calculated dominating line. If the
lines contains the same number
        of pieces with at least one shared attribute (same score), then store
both.'''

        line_attributes = []

        # Remove all the empty slots with no piece on it
        if -1 in line:
            line = list(dict.fromkeys(line))
            line.remove(-1)

        # Store the attributes of each piece in the current line
        for elem in line:
            piece = self.board.get_piece_charachteristics(elem)
            line_attributes.append([piece.HIGH, piece.COLOURED, piece.SOLID,
piece.SQUARE])

        # If there are more than 1 piece in the line, check what attributes
are reccuring.
        # Result is a boolean vector where True/False represent the
attributes that are common in the line
        # and None represent an attribute that is not shared. The order is
[High Coloured Solid Square]
        if len(line_attributes) > 1:
            old_result = line_attributes[0]
            for i in range(len(line_attributes)-1):
                result = []
                for j,k in zip(old_result, line_attributes[i+1]):
                    if j==k:
                        result.append(j)
                    else:
                        result.append(None)
                old_result = result
        elif len(line_attributes) == 1:
            # If the line is only 1 piece, all attributes of that piece are
the lines attributes.
            result = line_attributes[0]
        else:
            result = []

        # Check whether the result vector has an element that is not None,
thus yielding a possible new
        # dominating line
        if any(map(lambda x: not x is None, result)):
            return [len(line_attributes), line_counter, result]
        return None

    def check_attributes(self) -> list:
        '''Calculate the number of each attribute on the board.'''
```

```python
        board = self.board.get_board_status()
        attribue_values = {}
        attribue_values['High'] = 0
        attribue_values['Low'] = 0
        attribue_values['Color'] = 0
        attribue_values['Noncolor'] = 0
        attribue_values['Solid'] = 0
        attribue_values['Hollow'] = 0
        attribue_values['Square'] = 0
        attribue_values['Circle'] = 0

        for row in board:
            for elem in row:
                if elem >= 0:
                    if self.board.get_piece_charachteristics(elem).HIGH:
                        attribue_values['High'] += 1
                    else:
                        attribue_values['Low'] += 1
                    if self.board.get_piece_charachteristics(elem).COLOURED:
                        attribue_values['Color'] += 1
                    else:
                        attribue_values['Noncolor'] += 1
                    if self.board.get_piece_charachteristics(elem).SOLID:
                        attribue_values['Solid'] += 1
                    else:
                        attribue_values['Hollow'] += 1
                    if self.board.get_piece_charachteristics(elem).SQUARE:
                        attribue_values['Square'] += 1
                    else:
                        attribue_values['Circle'] += 1
        return attribue_values

    def rank_pieces(self, attribute_values: list) -> list:
        '''Ranks the pieces according to their shared attriutes with the 
board'''
        val = []

        for i in range(16):
            cur_val = 0
            piece = self.board.get_piece_charachteristics(i)

            if piece.HIGH:
                cur_val += attribute_values['High']
            else:
                cur_val += attribute_values['Low']
            if piece.COLOURED:
                cur_val += attribute_values['Color']
            else:
                cur_val += attribute_values['Noncolor']
            if piece.SOLID:
                cur_val += attribute_values['Solid']
            else:
                cur_val += attribute_values['Hollow']
            if piece.SQUARE:
                cur_val += attribute_values['Square']
            else:
                cur_val += attribute_values['Circle']
```

```
            val.append(cur_val)
        return val

    def desired_piece(self, dom_lines: list) -> list:
        '''This function determines what attributes the piece we choose
should have to avoid a loss'''

        dom_line_attributes = self.dominant_attributes(dom_lines)

        # If one of the attributes and it's corresponding opposite leads to a
win, then we lose no matter what piece we select.
        # E.g. both high and low.
        # If the above is not the case, store the inverse of the attributes
in order to obtain a vector of what attributes
        # we are searching for in a piece. If there's no condition on the
specific attribute, i.e. the piece can be either high or low,
        # store both True and False for the desirec piece in this attribute
spot.
        desired_piece = [[], [], [], []]
        for i in range(len(dom_line_attributes)):
            if len(dom_line_attributes[i]) == 2:
                #print("Lose no matter what")
                return self.pick_rule_3()
            elif len(dom_line_attributes[i]) == 1:
                desired_piece[i].append(not(dom_line_attributes[i][0]))
            else:
                desired_piece[i].append(True)
                desired_piece[i].append(False)

        # Now we look for a piece with the desired attributes
        found, piece_val = self.find_piece_attribute(desired_piece)
        if found:
            #print("Found a desired piece")
            return piece_val
        else:
        # If such a piece does not exist (have been played already), play a
piece at random
            #print("No desired piece left. Picking a piece at random")
            return self.pick_rule_3()

    def dominant_attributes(self, dominant_lines) -> list:
        '''The function returns what are the common attributes in the
dominating lines'''

        dom_line_attributes = [[], [], [], []]
        for i in range(4):
            for elem in dominant_lines:
                if elem[2][i] != None and elem[2][i] not in
dom_line_attributes[i]:
                    dom_line_attributes[i].append(elem[2][i])
        return dom_line_attributes

    def find_piece_attribute(self, desired_piece) -> tuple:
        '''The function looks for a piece with the desired attributes and
returns True and the piece value if found, otherwise
        it returns false and None'''
```

```
        for i in range(16):
            if i not in self.board.get_board_status():
                piece = self.board.get_piece_charachteristics(i)
                if piece.HIGH in desired_piece[0] and piece.COLOURED in
desired_piece[1] \
                    and piece.SOLID in desired_piece[2] and piece.SQUARE in
desired_piece[3]:
                    return (True, i)
        return (False, None)

    def place_piece_specified_line(self, line) -> tuple([int, int]):
        '''This function checks if the selected piece can be placed such that
the agent wins'''
        board = self.board.get_board_status()

        # Check if the piece should be placed in a row
        if line[1] >= 0 and line[1] <= 3:
            # look for an empy spot in the row
            for i in range(4):
                if board[line[1]][i] == -1:
                    return (i, line[1])
        # Check if the piece should be placed in a column
        elif line[1] >= 4 and line[1] <= 7:
            for i in range(4):
                if board[i, line[1]-4] == -1:
                    return (line[1]-4, i)
        # Check if the piece should be placed on the diagonal
        elif line[1] == 8:
            for i in range(4):
                if board[i, i] == -1:
                    return (i, i)
        # Check if the piece should be placed on the off-diagonal
        elif line[1] == 9:
            for i in range(4):
                if board[3-i, i] == -1:
                    return (i, 3-i)

    def count_shared_attributes_in_line(self, piece) -> list:
        '''This function counts the number of shared attributes in the line
with the selected piece. If the piece
        is high and the line contains 2 high pieces, this will count as 2
shared attributes. Thus, longer lines will
        therefore have an advantage in being picked'''
        board = self.board.get_board_status()
        line_val = []

        # For each row
        for row in board:
            line_val.append(self.count_line(row, piece))

        # For each column
        for i in range(self.board.BOARD_SIDE):
            line_val.append(self.count_line(board[:,i], piece))

        # Diagonals
        diag = []
        off_diag = []
```

```
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        line_val.append(self.count_line(diag, piece))
        line_val.append(self.count_line(off_diag, piece))

        return line_val


    def count_line(self, line, piece) -> int:
        '''Count the number of shared attributes in the line. If there's no
empty spots left in the line, return -1 in order
        to avoid selecting this line'''
        val = -1
        if -1 in line:
            val  = 0
            line = list(dict.fromkeys(line))
            line.remove(-1)

            for elem in line:
                high, color, solid, square = self.piece_dict[str(elem)]
                if high == self.board.get_piece_charachteristics(piece).HIGH:
                    val += 1
                if color ==
self.board.get_piece_charachteristics(piece).COLOURED:
                    val += 1
                if solid ==
self.board.get_piece_charachteristics(piece).SOLID:
                    val += 1
                if square ==
self.board.get_piece_charachteristics(piece).SQUARE:
                    val += 1
        return val

    def length_of_board_lines(self) -> list:
        '''Counts the lengths of all lines with at least 1 free spot'''

        board = self.board.get_board_status()
        length = []

        for row in board:
            length.append(self.length_of_line(row))

        for i in range(self.board.BOARD_SIDE):
            length.append(self.length_of_line(board[:,i]))

        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        length.append(self.length_of_line(diag))
        length.append(self.length_of_line(off_diag))

        return length
```

```
    def length_of_line(self, line) -> int:
        '''Counts the lenght of the current line'''
        if -1 in line:
            line = list(dict.fromkeys(line))
            line.remove(-1)
            return len(line)
        else:
            return self.board.BOARD_SIDE

    def pick_rule_1(self) -> int:
        '''Pick the piece with the most common attributes with the current
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
        max_pos = piece_ranking.index(max(piece_ranking))
        while max_pos in self.board.get_board_status():
            piece_ranking[max_pos] = -1
            max_pos = piece_ranking.index(max(piece_ranking))
        return max_pos


    def pick_rule_2(self) -> int:
        '''Pick the piece with the least common attributes with the current
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
        min_pos = piece_ranking.index(min(piece_ranking))
        while min_pos in self.board.get_board_status():
            piece_ranking[min_pos] = 100
            min_pos = piece_ranking.index(min(piece_ranking))
        return min_pos

    def pick_rule_3(self) -> int:
        '''Pick a piece at random'''
        random_piece = random.randint(0, 15)
        while random_piece in self.board.get_board_status():
            random_piece = random.randint(0, 15)
        return random_piece

    def place_rule_1(self) -> tuple([int, int]):
        '''Place the piece on the longest uninterrupted line that has no
shared attributes with the current piece'''

        # If there is a dominating line with 3 elements in it, we always
place the piece there if it generates a win.
        # Thus, we have already checked that we have no common attributes
with all 3-length dominating lines and can therefore place
        # the piece in this line
        if len(self.dom_line_dict['3']) > 0:
            return
self.place_piece_specified_line(self.dom_line_dict['3'][0])

        # For dominating lines of length less than 3 there first need to be a
check to confirm that there are no shared attributes
        # between the selected piece and the line in order to block it.
        for length in ['2', '1']:
```

```
            for line in self.dom_line_dict[length]:
                call = True
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        call = False
                        break
                if call:
                    return self.place_piece_specified_line(line)

        # If there are no dominating lines of any lenght we can block, place
it randomly
        return self.place_rule_6()

    def place_rule_2(self):
        '''Place the piece on the longest uninterrupted line with a shared
attribute, might not be the dominating line'''

        # There cannot be a uninterrupted line with 3 elements with a shared
attribute with out piece, if that was the
        # case, the piece would already have been placed there!
        # Thus we only need to check uninterrupted lines of length 2 and 1
and see where our piece has a shared attribute

        for length in ['2', '1']:
            for line in self.dom_line_dict[length]:
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        return self.place_piece_specified_line(line)

        # If there are no uninterrupted lines with a shared attribute, place
it randomly
        return self.place_rule_6()

    def place_rule_3(self):
        '''Places the piece on the line with the most shared attributes
independent of length'''

        piece = self.board.get_selected_piece()
        line_values = self.count_shared_attributes_in_line(piece)
        return self.place_piece_specified_line([0,
line_values.index(max(line_values))])

    def place_rule_4(self):
        '''Place the piece on the shortest line, independent of attributes'''

        lengths_on_board = self.length_of_board_lines()
        return self.place_piece_specified_line([0,
lengths_on_board.index(min(lengths_on_board))])

    def place_rule_5(self):
        '''Place the piece on the longest line, independent of attributes'''

        lengths_on_board = self.length_of_board_lines()

        # Replace all the lines of length 4 with a -1 as we cannot place a
```

```
piece on these lines
        if 4 in lengths_on_board:
            for i in range(len(lengths_on_board)):
                if lengths_on_board[i] == 4:
                    lengths_on_board[i] = -1
        return self.place_piece_specified_line([0,
lengths_on_board.index(max(lengths_on_board))])

    def place_rule_6(self):
        '''Place the piece at random'''
        board = self.board.get_board_status()

        x = random.randint(0, 3)
        y = random.randint(0, 3)

        while board[x,y] != -1:
            x = random.randint(0, 3)
            y = random.randint(0, 3)

        return (y, x)
```

generate_agent_least_dumb.py:

```
import random
import quarto

class LeastDumbAgent(quarto.Player):
    '''Evolved agent using the GA approach'''

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.board = quarto
        self.active_pieces = [i for i in range(16)]

        # Generate the probability of picking each rule for picking and
placing
        pick_prob_not_normalized = [random.random() for i in range(3)]
        pick_prob = [x/sum(pick_prob_not_normalized) for x in
pick_prob_not_normalized]
        self.pick_probability_1 = pick_prob[0]
        self.pick_probability_2 = pick_prob[1]
        self.pick_probability_3 = pick_prob[2]

        place_probability_not_normalized = [random.random() for i in
range(6)]
        place_prob = [x/sum(place_probability_not_normalized) for x in
place_probability_not_normalized]
        self.place_probability_1 = place_prob[0]
        self.place_probability_2 = place_prob[1]
        self.place_probability_3 = place_prob[2]
        self.place_probability_4 = place_prob[3]
        self.place_probability_5 = place_prob[4]
        self.place_probability_6 = place_prob[5]

        # Create a dictionary containing the attributes of each piece
```

```
        self.piece_dict = self.piece_attribute_dict()

        # Create a dictionary containing the dominating lines of each length
        self.dom_line_dict = {}

    def get_pick_prob(self):
        return (self.pick_probability_1, self.pick_probability_2,
self.pick_probability_3)

    def get_place_prob(self):
        return (self.place_probability_1, self.place_probability_2,
self.place_probability_3, \
            self.place_probability_4, self.place_probability_5,
self.place_probability_6)

    def set_pick_prob_1(self, val):
        self.pick_probability_1 = val

    def set_pick_prob_2(self, val):
        self.pick_probability_2 = val

    def set_pick_prob_3(self, val):
        self.pick_probability_3 = val

    def set_place_prob_1(self, val):
        self.place_probability_1 = val

    def set_place_prob_2(self, val):
        self.place_probability_2 = val

    def set_place_prob_3(self, val):
        self.place_probability_3 = val

    def set_place_prob_4(self, val):
        self.place_probability_4 = val

    def set_place_prob_5(self, val):
        self.place_probability_5 = val

    def set_place_prob_6(self, val):
        self.place_probability_6 = val

    def choose_piece(self) -> int:
        '''Decides what rule to be played when picking a piece by the
agent'''

        self.dominating_line()

        # If there are 3 pieces on the same line with at least 1 common
attribute and a 4th free slot
        if len(self.dom_line_dict['3']) > 0:
            return self.desired_piece(self.dom_line_dict['3'])

        # Genereate the random value to decide what simple rule to play
        val = random.random()

        # Play rule 1
```

```
        if val <= self.pick_probability_1:
            #print("Rule 1")
            return self.pick_rule_1()
        # Play rule 2
        elif val <= self.pick_probability_1 + self.pick_probability_2:
            #print("Rule 2")
            return self.pick_rule_2()
        # Play rule 3
        else:
            #print("Rule 3")
            return self.pick_rule_3()

    def place_piece(self) -> tuple([int, int]):
        '''Decides where to place the piece given by the opponent'''

        self.dominating_line()



        # Genereate the random value to decide what simple rule to play
        val = random.random()

        # Play rule 1
        if val <= self.place_probability_1:
            #print("Rule 1")
            return self.place_rule_1()
        # Play rule 2
        elif val <= self.place_probability_1 + self.place_probability_2:
            #print("Rule 2")
            return self.place_rule_2()
        # Play rule 3
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3:
            #print("Rule 3")
            return self.place_rule_3()
        # Play rule 4
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3 + self.place_probability_4:
            #print("Rule 4")
            return self.place_rule_4()
        # Play rule 5
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3 + self.place_probability_4 +
self.place_probability_5:
            #print("Rule 5")
            return self.place_rule_5()
        # Play rule 6
        else:
            #print("Rule 6")
            return self.place_rule_6()

    def piece_attribute_dict(self) -> dict:
        piece_dict = {}

        for i in range(16):
            piece = self.board.get_piece_charachteristics(i)
            piece_dict[str(i)] = (piece.HIGH, piece.COLOURED, piece.SOLID,
```

```
piece.SQUARE)
        return piece_dict

    def dominating_line(self):
        '''Store each row/column/diagonal with at least one shared attribute
in the dom_line_dict based on how long they are.
        Each row, column and diagonal have been assigned a number, where the
rows from top to bottom are
        0 - 3, columns from left to right are 4-7, diagonal 8 and off-
diagonal 9.'''
        board = self.board.get_board_status()

        self.dom_line_dict['1'] = []
        self.dom_line_dict['2'] = []
        self.dom_line_dict['3'] = []
        self.dom_line_dict['4'] = []

        # Check horizontal
        line_counter = 0
        for row in board:
            dominating_result = self.dominating(row, line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check vertical
        for i in range(self.board.BOARD_SIDE):
            dominating_result = self.dominating(board[:,i], line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check diagonal
        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        dominating_result = self.dominating(diag, line_counter)
        if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        dominating_result = self.dominating(off_diag, line_counter+1)
        if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        return dominating_result

    def dominating(self, line, line_counter: int) -> list:
        '''Calculate if the current line has a longer line of pieces with at
least one shared attribute
        (higher score) than the previously calculated dominating line. If the
lines contains the same number
        of pieces with at least one shared attribute (same score), then store
```

```
both.'''

        line_attributes = []

        # Remove all the empty slots with no piece on it
        if -1 in line:
            line = list(dict.fromkeys(line))
            line.remove(-1)

        # Store the attributes of each piece in the current line
        for elem in line:
            piece = self.board.get_piece_charachteristics(elem)
            line_attributes.append([piece.HIGH, piece.COLOURED, piece.SOLID,
piece.SQUARE])

        # If there are more than 1 piece in the line, check what attributes
are reccuring.
        # Result is a boolean vector where True/False represent the
attributes that are common in the line
        # and None represent an attribute that is not shared. The order is
[High Coloured Solid Square]
        if len(line_attributes) > 1:
            old_result = line_attributes[0]
            for i in range(len(line_attributes)-1):
                result = []
                for j,k in zip(old_result, line_attributes[i+1]):
                    if j==k:
                        result.append(j)
                    else:
                        result.append(None)
                old_result = result
        elif len(line_attributes) == 1:
            # If the line is only 1 piece, all attributes of that piece are
the lines attributes.
            result = line_attributes[0]
        else:
            result = []

        # Check whether the result vector has an element that is not None,
thus yielding a possible new
        # dominating line
        if any(map(lambda x: not x is None, result)):
            return [len(line_attributes), line_counter, result]
        return None

    def check_attributes(self) -> list:
        '''Calculate the number of each attribute on the board.'''

        board = self.board.get_board_status()
        attribue_values = {}
        attribue_values['High'] = 0
        attribue_values['Low'] = 0
        attribue_values['Color'] = 0
        attribue_values['Noncolor'] = 0
        attribue_values['Solid'] = 0
        attribue_values['Hollow'] = 0
        attribue_values['Square'] = 0
```

```
        attribue_values['Circle'] = 0

        for row in board:
            for elem in row:
                if elem >= 0:
                    if self.board.get_piece_charachteristics(elem).HIGH:
                        attribue_values['High'] += 1
                    else:
                        attribue_values['Low'] += 1
                    if self.board.get_piece_charachteristics(elem).COLOURED:
                        attribue_values['Color'] += 1
                    else:
                        attribue_values['Noncolor'] += 1
                    if self.board.get_piece_charachteristics(elem).SOLID:
                        attribue_values['Solid'] += 1
                    else:
                        attribue_values['Hollow'] += 1
                    if self.board.get_piece_charachteristics(elem).SQUARE:
                        attribue_values['Square'] += 1
                    else:
                        attribue_values['Circle'] += 1
        return attribue_values

    def rank_pieces(self, attribute_values: list) -> list:
        '''Ranks the pieces according to their shared attriutes with the
board'''
        val = []

        for i in range(16):
            cur_val = 0
            piece = self.board.get_piece_charachteristics(i)

            if piece.HIGH:
                cur_val += attribute_values['High']
            else:
                cur_val += attribute_values['Low']
            if piece.COLOURED:
                cur_val += attribute_values['Color']
            else:
                cur_val += attribute_values['Noncolor']
            if piece.SOLID:
                cur_val += attribute_values['Solid']
            else:
                cur_val += attribute_values['Hollow']
            if piece.SQUARE:
                cur_val += attribute_values['Square']
            else:
                cur_val += attribute_values['Circle']
            val.append(cur_val)
        return val

    def desired_piece(self, dom_lines: list) -> list:
        '''This function determines what attributes the piece we choose
should have to avoid a loss'''

        dom_line_attributes = self.dominant_attributes(dom_lines)
```

```
        # If one of the attributes and it's corresponding opposite leads to a
win, then we lose no matter what piece we select.
        # E.g. both high and low.
        # If the above is not the case, store the inverse of the attributes
in order to obtain a vector of what attributes
        # we are searching for in a piece. If there's no condition on the
specific attribute, i.e. the piece can be either high or low,
        # store both True and False for the desirec piece in this attribute
spot.
        desired_piece = [[], [], [], []]
        for i in range(len(dom_line_attributes)):
            if len(dom_line_attributes[i]) == 2:
                #print("Lose no matter what")
                return self.pick_rule_3()
            elif len(dom_line_attributes[i]) == 1:
                desired_piece[i].append(not(dom_line_attributes[i][0]))
            else:
                desired_piece[i].append(True)
                desired_piece[i].append(False)

        # Now we look for a piece with the desired attributes
        found, piece_val = self.find_piece_attribute(desired_piece)
        if found:
            #print("Found a desired piece")
            return piece_val
        else:
        # If such a piece does not exist (have been played already), play a
piece at random
            #print("No desired piece left. Picking a piece at random")
            return self.pick_rule_3()

    def dominant_attributes(self, dominant_lines) -> list:
        '''The function returns what are the common attributes in the
dominating lines'''

        dom_line_attributes = [[], [], [], []]
        for i in range(4):
            for elem in dominant_lines:
                if elem[2][i] != None and elem[2][i] not in
dom_line_attributes[i]:
                    dom_line_attributes[i].append(elem[2][i])
        return dom_line_attributes

    def find_piece_attribute(self, desired_piece) -> tuple:
        '''The function looks for a piece with the desired attributes and
returns True and the piece value if found, otherwise
        it returns false and None'''

        for i in range(16):
            if i not in self.board.get_board_status():
                piece = self.board.get_piece_charachteristics(i)
                if piece.HIGH in desired_piece[0] and piece.COLOURED in
desired_piece[1] \
                    and piece.SOLID in desired_piece[2] and piece.SQUARE in
desired_piece[3]:
                    return (True, i)
        return (False, None)
```

```python
    def place_piece_specified_line(self, line) -> tuple([int, int]):
        '''This function checks if the selected piece can be placed such that
the agent wins'''
        board = self.board.get_board_status()

        # Check if the piece should be placed in a row
        if line[1] >= 0 and line[1] <= 3:
            # look for an empy spot in the row
            for i in range(4):
                if board[line[1]][i] == -1:
                    return (i, line[1])
        # Check if the piece should be placed in a column
        elif line[1] >= 4 and line[1] <= 7:
            for i in range(4):
                if board[i, line[1]-4] == -1:
                    return (line[1]-4, i)
        # Check if the piece should be placed on the diagonal
        elif line[1] == 8:
            for i in range(4):
                if board[i, i] == -1:
                    return (i, i)
        # Check if the piece should be placed on the off-diagonal
        elif line[1] == 9:
            for i in range(4):
                if board[3-i, i] == -1:
                    return (i, 3-i)

    def count_shared_attributes_in_line(self, piece) -> list:
        '''This function counts the number of shared attributes in the line
with the selected piece. If the piece
        is high and the line contains 2 high pieces, this will count as 2
shared attributes. Thus, longer lines will
        therefore have an advantage in being picked'''
        board = self.board.get_board_status()
        line_val = []

        # For each row
        for row in board:
            line_val.append(self.count_line(row, piece))

        # For each column
        for i in range(self.board.BOARD_SIDE):
            line_val.append(self.count_line(board[:,i], piece))

        # Diagonals
        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        line_val.append(self.count_line(diag, piece))
        line_val.append(self.count_line(off_diag, piece))

        return line_val
```

```
    def count_line(self, line, piece) -> int:
        '''Count the number of shared attributes in the line. If there's no
empty spots left in the line, return -1 in order
        to avoid selecting this line'''
        val = -1
        if -1 in line:
            val  = 0
            line = list(dict.fromkeys(line))
            line.remove(-1)

            for elem in line:
                high, color, solid, square = self.piece_dict[str(elem)]
                if high == self.board.get_piece_charachteristics(piece).HIGH:
                    val += 1
                if color ==
self.board.get_piece_charachteristics(piece).COLOURED:
                    val += 1
                if solid ==
self.board.get_piece_charachteristics(piece).SOLID:
                    val += 1
                if square ==
self.board.get_piece_charachteristics(piece).SQUARE:
                    val += 1
        return val

    def length_of_board_lines(self) -> list:
        '''Counts the lengths of all lines with at least 1 free spot'''

        board = self.board.get_board_status()
        length = []

        for row in board:
            length.append(self.length_of_line(row))

        for i in range(self.board.BOARD_SIDE):
            length.append(self.length_of_line(board[:,i]))

        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        length.append(self.length_of_line(diag))
        length.append(self.length_of_line(off_diag))

        return length

    def length_of_line(self, line) -> int:
        '''Counts the lenght of the current line'''
        if -1 in line:
            line = list(dict.fromkeys(line))
            line.remove(-1)
            return len(line)
        else:
            return self.board.BOARD_SIDE
```

```
    def pick_rule_1(self) -> int:
        '''Pick the piece with the most common attributes with the current
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
        max_pos = piece_ranking.index(max(piece_ranking))
        while max_pos in self.board.get_board_status():
            piece_ranking[max_pos] = -1
            max_pos = piece_ranking.index(max(piece_ranking))
        return max_pos


    def pick_rule_2(self) -> int:
        '''Pick the piece with the least common attributes with the current
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
        min_pos = piece_ranking.index(min(piece_ranking))
        while min_pos in self.board.get_board_status():
            piece_ranking[min_pos] = 100
            min_pos = piece_ranking.index(min(piece_ranking))
        return min_pos

    def pick_rule_3(self) -> int:
        '''Pick a piece at random'''
        random_piece = random.randint(0, 15)
        while random_piece in self.board.get_board_status():
            random_piece = random.randint(0, 15)
        return random_piece

    def place_rule_1(self) -> tuple([int, int]):
        '''Place the piece on the longest uninterrupted line that has no
shared attributes with the current piece'''

        # If there is a dominating line with 3 elements in it, we always
place the piece there if it generates a win.
        # Thus, we have already checked that we have no common attributes
with all 3-length dominating lines and can therefore place
        # the piece in this line
        if len(self.dom_line_dict['3']) > 0:
            return
self.place_piece_specified_line(self.dom_line_dict['3'][0])

        # For dominating lines of length less than 3 there first need to be a
check to confirm that there are no shared attributes
        # between the selected piece and the line in order to block it.
        for length in ['2', '1']:
            for line in self.dom_line_dict[length]:
                call = True
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        call = False
                        break
                if call:
                    return self.place_piece_specified_line(line)
```

```
        # If there are no dominating lines of any lenght we can block, place
it randomly
        return self.place_rule_6()

    def place_rule_2(self):
        '''Place the piece on the longest uninterrupted line with a shared
attribute, might not be the dominating line'''

        # There cannot be a uninterrupted line with 3 elements with a shared
attribute with out piece, if that was the
        # case, the piece would already have been placed there!
        # Thus we only need to check uninterrupted lines of length 2 and 1
and see where our piece has a shared attribute

        for length in ['2', '1']:
            for line in self.dom_line_dict[length]:
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        return self.place_piece_specified_line(line)

        # If there are no uninterrupted lines with a shared attribute, place
it randomly
        return self.place_rule_6()

    def place_rule_3(self):
        '''Places the piece on the line with the most shared attributes
independent of length'''

        piece = self.board.get_selected_piece()
        line_values = self.count_shared_attributes_in_line(piece)
        return self.place_piece_specified_line([0,
line_values.index(max(line_values))])

    def place_rule_4(self):
        '''Place the piece on the shortest line, independent of attributes'''

        lengths_on_board = self.length_of_board_lines()
        return self.place_piece_specified_line([0,
lengths_on_board.index(min(lengths_on_board))])

    def place_rule_5(self):
        '''Place the piece on the longest line, independent of attributes'''

        lengths_on_board = self.length_of_board_lines()

        # Replace all the lines of length 4 with a -1 as we cannot place a
piece on these lines
        if 4 in lengths_on_board:
            for i in range(len(lengths_on_board)):
                if lengths_on_board[i] == 4:
                    lengths_on_board[i] = -1
        return self.place_piece_specified_line([0,
lengths_on_board.index(max(lengths_on_board))])

    def place_rule_6(self):
```

```
        '''Place the piece at random'''
        board = self.board.get_board_status()

        x = random.randint(0, 3)
        y = random.randint(0, 3)

        while board[x,y] != -1:
            x = random.randint(0, 3)
            y = random.randint(0, 3)

        return (y, x)
```

generate_agent_less_dumb.py:

```
import random
import quarto

class LessDumbAgent(quarto.Player):
    '''Evolved agent using the GA approach'''

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.board = quarto
        self.active_pieces = [i for i in range(16)]

        # Generate the probability of picking each rule for picking and
placing
        pick_prob_not_normalized = [random.random() for i in range(3)]
        pick_prob = [x/sum(pick_prob_not_normalized) for x in
pick_prob_not_normalized]
        self.pick_probability_1 = pick_prob[0]
        self.pick_probability_2 = pick_prob[1]
        self.pick_probability_3 = pick_prob[2]

        place_probability_not_normalized = [random.random() for i in
range(6)]
        place_prob = [x/sum(place_probability_not_normalized) for x in
place_probability_not_normalized]
        self.place_probability_1 = place_prob[0]
        self.place_probability_2 = place_prob[1]
        self.place_probability_3 = place_prob[2]
        self.place_probability_4 = place_prob[3]
        self.place_probability_5 = place_prob[4]
        self.place_probability_6 = place_prob[5]

        # Create a dictionary containing the attributes of each piece
        self.piece_dict = self.piece_attribute_dict()

        # Create a dictionary containing the dominating lines of each length
        self.dom_line_dict = {}

    def get_pick_prob(self):
        return (self.pick_probability_1, self.pick_probability_2,
self.pick_probability_3)
```

```python
    def get_place_prob(self):
        return (self.place_probability_1, self.place_probability_2,
self.place_probability_3, \
            self.place_probability_4, self.place_probability_5,
self.place_probability_6)

    def set_pick_prob_1(self, val):
        self.pick_probability_1 = val

    def set_pick_prob_2(self, val):
        self.pick_probability_2 = val

    def set_pick_prob_3(self, val):
        self.pick_probability_3 = val

    def set_place_prob_1(self, val):
        self.place_probability_1 = val

    def set_place_prob_2(self, val):
        self.place_probability_2 = val

    def set_place_prob_3(self, val):
        self.place_probability_3 = val

    def set_place_prob_4(self, val):
        self.place_probability_4 = val

    def set_place_prob_5(self, val):
        self.place_probability_5 = val

    def set_place_prob_6(self, val):
        self.place_probability_6 = val

    def choose_piece(self) -> int:
        '''Decides what rule to be played when picking a piece by the
agent'''

        self.dominating_line()

        '''# If there are 3 pieces on the same line with at least 1 common
attribute and a 4th free slot
        if len(self.dom_line_dict['3']) > 0:
            return self.desired_piece(self.dom_line_dict['3'])'''

        # Genereate the random value to decide what simple rule to play
        val = random.random()

        # Play rule 1
        if val <= self.pick_probability_1:
            #print("Rule 1")
            return self.pick_rule_1()
        # Play rule 2
        elif val <= self.pick_probability_1 + self.pick_probability_2:
            #print("Rule 2")
            return self.pick_rule_2()
        # Play rule 3
        else:
```

```
            #print("Rule 3")
            return self.pick_rule_3()

    def place_piece(self) -> tuple([int, int]):
        '''Decides where to place the piece given by the opponent'''

        self.dominating_line()

        # If there are 3 pieces on the same line with at least 1 common
attribute and a 4th free slot
        # and the selected piece shares one of the common attributes
        if len(self.dom_line_dict['3']) > 0:
            for line in self.dom_line_dict['3']:
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        return self.place_piece_specified_line(line)

        # Genereate the random value to decide what simple rule to play
        val = random.random()

        # Play rule 1
        if val <= self.place_probability_1:
            #print("Rule 1")
            return self.place_rule_1()
        # Play rule 2
        elif val <= self.place_probability_1 + self.place_probability_2:
            #print("Rule 2")
            return self.place_rule_2()
        # Play rule 3
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3:
            #print("Rule 3")
            return self.place_rule_3()
        # Play rule 4
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3 + self.place_probability_4:
            #print("Rule 4")
            return self.place_rule_4()
        # Play rule 5
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3 + self.place_probability_4 +
self.place_probability_5:
            #print("Rule 5")
            return self.place_rule_5()
        # Play rule 6
        else:
            #print("Rule 6")
            return self.place_rule_6()

    def piece_attribute_dict(self) -> dict:
        piece_dict = {}

        for i in range(16):
            piece = self.board.get_piece_charachteristics(i)
            piece_dict[str(i)] = (piece.HIGH, piece.COLOURED, piece.SOLID,
piece.SQUARE)
```

```
        return piece_dict

    def dominating_line(self):
        '''Store each row/column/diagonal with at least one shared attribute
in the dom_line_dict based on how long they are.
        Each row, column and diagonal have been assigned a number, where the
rows from top to bottom are
        0 - 3, columns from left to right are 4-7, diagonal 8 and off-
diagonal 9.'''
        board = self.board.get_board_status()

        self.dom_line_dict['1'] = []
        self.dom_line_dict['2'] = []
        self.dom_line_dict['3'] = []
        self.dom_line_dict['4'] = []

        # Check horizontal
        line_counter = 0
        for row in board:
            dominating_result = self.dominating(row, line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check vertical
        for i in range(self.board.BOARD_SIDE):
            dominating_result = self.dominating(board[:,i], line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check diagonal
        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        dominating_result = self.dominating(diag, line_counter)
        if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        dominating_result = self.dominating(off_diag, line_counter+1)
        if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        return dominating_result

    def dominating(self, line, line_counter: int) -> list:
        '''Calculate if the current line has a longer line of pieces with at
least one shared attribute
        (higher score) than the previously calculated dominating line. If the
lines contains the same number
        of pieces with at least one shared attribute (same score), then store
both.'''
```

```
        line_attributes = []

        # Remove all the empty slots with no piece on it
        if -1 in line:
            line = list(dict.fromkeys(line))
            line.remove(-1)

        # Store the attributes of each piece in the current line
        for elem in line:
            piece = self.board.get_piece_charachteristics(elem)
            line_attributes.append([piece.HIGH, piece.COLOURED, piece.SOLID,
piece.SQUARE])

        # If there are more than 1 piece in the line, check what attributes
are reccuring.
        # Result is a boolean vector where True/False represent the
attributes that are common in the line
        # and None represent an attribute that is not shared. The order is
[High Coloured Solid Square]
        if len(line_attributes) > 1:
            old_result = line_attributes[0]
            for i in range(len(line_attributes)-1):
                result = []
                for j,k in zip(old_result, line_attributes[i+1]):
                    if j==k:
                        result.append(j)
                    else:
                        result.append(None)
                old_result = result
        elif len(line_attributes) == 1:
            # If the line is only 1 piece, all attributes of that piece are
the lines attributes.
            result = line_attributes[0]
        else:
            result = []

        # Check whether the result vector has an element that is not None,
thus yielding a possible new
        # dominating line
        if any(map(lambda x: not x is None, result)):
            return [len(line_attributes), line_counter, result]
        return None

    def check_attributes(self) -> list:
        '''Calculate the number of each attribute on the board.'''

        board = self.board.get_board_status()
        attribue_values = {}
        attribue_values['High'] = 0
        attribue_values['Low'] = 0
        attribue_values['Color'] = 0
        attribue_values['Noncolor'] = 0
        attribue_values['Solid'] = 0
        attribue_values['Hollow'] = 0
        attribue_values['Square'] = 0
        attribue_values['Circle'] = 0
```

```
        for row in board:
            for elem in row:
                if elem >= 0:
                    if self.board.get_piece_charachteristics(elem).HIGH:
                        attribue_values['High'] += 1
                    else:
                        attribue_values['Low'] += 1
                    if self.board.get_piece_charachteristics(elem).COLOURED:
                        attribue_values['Color'] += 1
                    else:
                        attribue_values['Noncolor'] += 1
                    if self.board.get_piece_charachteristics(elem).SOLID:
                        attribue_values['Solid'] += 1
                    else:
                        attribue_values['Hollow'] += 1
                    if self.board.get_piece_charachteristics(elem).SQUARE:
                        attribue_values['Square'] += 1
                    else:
                        attribue_values['Circle'] += 1
        return attribue_values

    def rank_pieces(self, attribute_values: list) -> list:
        '''Ranks the pieces according to their shared attriutes with the
board'''
        val = []

        for i in range(16):
            cur_val = 0
            piece = self.board.get_piece_charachteristics(i)

            if piece.HIGH:
                cur_val += attribute_values['High']
            else:
                cur_val += attribute_values['Low']
            if piece.COLOURED:
                cur_val += attribute_values['Color']
            else:
                cur_val += attribute_values['Noncolor']
            if piece.SOLID:
                cur_val += attribute_values['Solid']
            else:
                cur_val += attribute_values['Hollow']
            if piece.SQUARE:
                cur_val += attribute_values['Square']
            else:
                cur_val += attribute_values['Circle']
            val.append(cur_val)
        return val

    def desired_piece(self, dom_lines: list) -> list:
        '''This function determines what attributes the piece we choose
should have to avoid a loss'''

        dom_line_attributes = self.dominant_attributes(dom_lines)

        # If one of the attributes and it's corresponding opposite leads to a
```

```
win, then we lose no matter what piece we select.
        # E.g. both high and low.
        # If the above is not the case, store the inverse of the attributes
in order to obtain a vector of what attributes
        # we are searching for in a piece. If there's no condition on the
specific attribute, i.e. the piece can be either high or low,
        # store both True and False for the desirec piece in this attribute
spot.
        desired_piece = [[], [], [], []]
        for i in range(len(dom_line_attributes)):
            if len(dom_line_attributes[i]) == 2:
                #print("Lose no matter what")
                return self.pick_rule_3()
            elif len(dom_line_attributes[i]) == 1:
                desired_piece[i].append(not(dom_line_attributes[i][0]))
            else:
                desired_piece[i].append(True)
                desired_piece[i].append(False)

        # Now we look for a piece with the desired attributes
        found, piece_val = self.find_piece_attribute(desired_piece)
        if found:
            #print("Found a desired piece")
            return piece_val
        else:
        # If such a piece does not exist (have been played already), play a
piece at random
            #print("No desired piece left. Picking a piece at random")
            return self.pick_rule_3()

    def dominant_attributes(self, dominant_lines) -> list:
        '''The function returns what are the common attributes in the
dominating lines'''

        dom_line_attributes = [[], [], [], []]
        for i in range(4):
            for elem in dominant_lines:
                if elem[2][i] != None and elem[2][i] not in
dom_line_attributes[i]:
                    dom_line_attributes[i].append(elem[2][i])
        return dom_line_attributes

    def find_piece_attribute(self, desired_piece) -> tuple:
        '''The function looks for a piece with the desired attributes and
returns True and the piece value if found, otherwise
        it returns false and None'''

        for i in range(16):
            if i not in self.board.get_board_status():
                piece = self.board.get_piece_charachteristics(i)
                if piece.HIGH in desired_piece[0] and piece.COLOURED in
desired_piece[1] \
                    and piece.SOLID in desired_piece[2] and piece.SQUARE in
desired_piece[3]:
                    return (True, i)
        return (False, None)
```

```python
    def place_piece_specified_line(self, line) -> tuple([int, int]):
        '''This function checks if the selected piece can be placed such that
the agent wins'''
        board = self.board.get_board_status()

        # Check if the piece should be placed in a row
        if line[1] >= 0 and line[1] <= 3:
            # look for an empy spot in the row
            for i in range(4):
                if board[line[1]][i] == -1:
                    return (i, line[1])
        # Check if the piece should be placed in a column
        elif line[1] >= 4 and line[1] <= 7:
            for i in range(4):
                if board[i, line[1]-4] == -1:
                    return (line[1]-4, i)
        # Check if the piece should be placed on the diagonal
        elif line[1] == 8:
            for i in range(4):
                if board[i, i] == -1:
                    return (i, i)
        # Check if the piece should be placed on the off-diagonal
        elif line[1] == 9:
            for i in range(4):
                if board[3-i, i] == -1:
                    return (i, 3-i)

    def count_shared_attributes_in_line(self, piece) -> list:
        '''This function counts the number of shared attributes in the line
with the selected piece. If the piece
        is high and the line contains 2 high pieces, this will count as 2
shared attributes. Thus, longer lines will
        therefore have an advantage in being picked'''
        board = self.board.get_board_status()
        line_val = []

        # For each row
        for row in board:
            line_val.append(self.count_line(row, piece))

        # For each column
        for i in range(self.board.BOARD_SIDE):
            line_val.append(self.count_line(board[:,i], piece))

        # Diagonals
        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        line_val.append(self.count_line(diag, piece))
        line_val.append(self.count_line(off_diag, piece))

        return line_val
```

```python
    def count_line(self, line, piece) -> int:
        '''Count the number of shared attributes in the line. If there's no
empty spots left in the line, return -1 in order
        to avoid selecting this line'''
        val = -1
        if -1 in line:
            val  = 0
            line = list(dict.fromkeys(line))
            line.remove(-1)

            for elem in line:
                high, color, solid, square = self.piece_dict[str(elem)]
                if high == self.board.get_piece_charachteristics(piece).HIGH:
                    val += 1
                if color ==
self.board.get_piece_charachteristics(piece).COLOURED:
                    val += 1
                if solid ==
self.board.get_piece_charachteristics(piece).SOLID:
                    val += 1
                if square ==
self.board.get_piece_charachteristics(piece).SQUARE:
                    val += 1
        return val

    def length_of_board_lines(self) -> list:
        '''Counts the lengths of all lines with at least 1 free spot'''

        board = self.board.get_board_status()
        length = []

        for row in board:
            length.append(self.length_of_line(row))

        for i in range(self.board.BOARD_SIDE):
            length.append(self.length_of_line(board[:,i]))

        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        length.append(self.length_of_line(diag))
        length.append(self.length_of_line(off_diag))

        return length

    def length_of_line(self, line) -> int:
        '''Counts the lenght of the current line'''
        if -1 in line:
            line = list(dict.fromkeys(line))
            line.remove(-1)
            return len(line)
        else:
            return self.board.BOARD_SIDE
```

```python
    def pick_rule_1(self) -> int:
        '''Pick the piece with the most common attributes with the current
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
        max_pos = piece_ranking.index(max(piece_ranking))
        while max_pos in self.board.get_board_status():
            piece_ranking[max_pos] = -1
            max_pos = piece_ranking.index(max(piece_ranking))
        return max_pos


    def pick_rule_2(self) -> int:
        '''Pick the piece with the least common attributes with the current
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
        min_pos = piece_ranking.index(min(piece_ranking))
        while min_pos in self.board.get_board_status():
            piece_ranking[min_pos] = 100
            min_pos = piece_ranking.index(min(piece_ranking))
        return min_pos

    def pick_rule_3(self) -> int:
        '''Pick a piece at random'''
        random_piece = random.randint(0, 15)
        while random_piece in self.board.get_board_status():
            random_piece = random.randint(0, 15)
        return random_piece

    def place_rule_1(self) -> tuple([int, int]):
        '''Place the piece on the longest uninterrupted line that has no
shared attributes with the current piece'''

        # If there is a dominating line with 3 elements in it, we always
place the piece there if it generates a win.
        # Thus, we have already checked that we have no common attributes
with all 3-length dominating lines and can therefore place
        # the piece in this line
        if len(self.dom_line_dict['3']) > 0:
            return
self.place_piece_specified_line(self.dom_line_dict['3'][0])

        # For dominating lines of length less than 3 there first need to be a
check to confirm that there are no shared attributes
        # between the selected piece and the line in order to block it.
        for length in ['2', '1']:
            for line in self.dom_line_dict[length]:
                call = True
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        call = False
                        break
                if call:
                    return self.place_piece_specified_line(line)
```

```
        # If there are no dominating lines of any lenght we can block, place 
it randomly
        return self.place_rule_6()

    def place_rule_2(self):
        '''Place the piece on the longest uninterrupted line with a shared 
attribute, might not be the dominating line'''

        # There cannot be a uninterrupted line with 3 elements with a shared 
attribute with out piece, if that was the
        # case, the piece would already have been placed there!
        # Thus we only need to check uninterrupted lines of length 2 and 1 
and see where our piece has a shared attribute

        for length in ['2', '1']:
            for line in self.dom_line_dict[length]:
                for j,k in zip(line[2], 
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        return self.place_piece_specified_line(line)

        # If there are no uninterrupted lines with a shared attribute, place 
it randomly
        return self.place_rule_6()

    def place_rule_3(self):
        '''Places the piece on the line with the most shared attributes 
independent of length'''

        piece = self.board.get_selected_piece()
        line_values = self.count_shared_attributes_in_line(piece)
        return self.place_piece_specified_line([0, 
line_values.index(max(line_values))])

    def place_rule_4(self):
        '''Place the piece on the shortest line, independent of attributes'''

        lengths_on_board = self.length_of_board_lines()
        return self.place_piece_specified_line([0, 
lengths_on_board.index(min(lengths_on_board))])

    def place_rule_5(self):
        '''Place the piece on the longest line, independent of attributes'''

        lengths_on_board = self.length_of_board_lines()

        # Replace all the lines of length 4 with a -1 as we cannot place a 
piece on these lines
        if 4 in lengths_on_board:
            for i in range(len(lengths_on_board)):
                if lengths_on_board[i] == 4:
                    lengths_on_board[i] = -1
        return self.place_piece_specified_line([0, 
lengths_on_board.index(max(lengths_on_board))])

    def place_rule_6(self):
        '''Place the piece at random'''
```

```
        board = self.board.get_board_status()

        x = random.randint(0, 3)
        y = random.randint(0, 3)

        while board[x,y] != -1:
            x = random.randint(0, 3)
            y = random.randint(0, 3)

        return (y, x)
```

generate_individuals.py:

```
from generate_agent import EvolvedAgent
import quarto
import random
import itertools

def fitness(individuals: list, game: quarto.Quarto, num_games: int) -> list:
    '''The fitness is determined as the number of wins in case of a tie the
number of draws
    between two individuals. The function thus lets all individuals play
eachother num_games
    number of games and stores (win, draw) in a tuple and returns a list
containing all
    individuas and the respective fitness'''

    # Create a dict for each individual to store the score
    individual_score = {}
    for individual in individuals:
        individual_score[individual] = [0, 0]

    # Let all individuals play against each other num_games number of times
to determine the fitness
    # of each individual
    for individual_1, individual_2 in itertools.combinations(individuals, 2):

        play_1_win, play_2_win, draw = gameplay(individual_1, individual_2,
game, num_games)

        # Store the score for each individual
        individual_score[individual_1][0] += play_1_win
        individual_score[individual_1][1] += draw
        individual_score[individual_2][0] += play_2_win
        individual_score[individual_2][1] += draw

        # Store the individual and it's corresponding score together
        individuals_and_score = []
        for individual in individuals:
            score = individual_score[individual]
            individuals_and_score.append([score, individual])

    return individuals_and_score

def gameplay(individual_1: EvolvedAgent, individual_2: EvolvedAgent, game:
```

```python
quarto.Quarto, num_games: int) -> tuple:
    # Keep track of the result of each match
    draw = 0
    play_1_win = 0
    play_2_win = 0

    # Set players
    game.set_players((individual_1, individual_2))

    # Play num_games games between each individual pair
    for _ in range(num_games):
        game.reset()
        result = game.run()
        if result == -1:
            draw += 1
        elif result == 0:
            play_1_win += 1
        elif result == 1:
            play_2_win += 1

    return (play_1_win, play_2_win, draw)

def population_init(mu: int, game: quarto.Quarto) -> list:
    '''Initialize the population'''
    individuals = []

    for i in range(mu):
        individuals.append(EvolvedAgent(game))

    individuals = fitness(individuals, game, 1)
    return individuals

def roulette(individuals: list) -> EvolvedAgent:
    '''The roulette gives an individual a chance to be selected as a parent
in relation to it's fitness'''

    tot_wins = 0
    # Calculate the sum of all wins
    for individual in individuals:
        tot_wins += individual[0][0]

    # Spin the wheel, i.e. generate a random number between 0 and tot_wins to
find out the winner
    parent_val = random.randint(0, tot_wins)

    for individual in individuals:
        if parent_val <= individual[0][0]:
            return individual[1]
        else:
            parent_val -= individual[0][0]

def offspring(individuals: list, game: quarto.Quarto, mutation: int) -> list:
    '''Generate offspring of the population'''

    offspring = []

    for i in range(int(len(individuals)/2)):
```

```python
        p1 = roulette(individuals)
        p2 = roulette(individuals)

        # Generate children
        child1, child2 = crossover(p1, p2, game)

        # Mutation
        if random.random() <= mutation:
            mutate(child1)

        if random.random() <= mutation:
            mutate(child2)

        # Add the children to the offspring
        offspring.append(child1)
        offspring.append(child2)

    return offspring


def mutate(individual: EvolvedAgent):
    '''Mutate the individual'''

    # Determine what picking and placing rule to mutate
    pick_rule = random.randint(0,2)
    place_rule = random.randint(0,5)

    # change said rules
    pick_val = list(individual.get_pick_prob())
    pick_val[pick_rule] = random.random()

    place_val = list(individual.get_place_prob())
    place_val[place_rule] = random.random()

    set_pick_and_place(individual, pick_val, place_val)

def crossover(p1: EvolvedAgent, p2: EvolvedAgent, game: quarto.Quarto) ->
tuple:
    '''Creates two children to two parents'''
    pick_cross = random.randint(0, 3)
    place_cross = random.randint(0, 6)

    parent_1 = p1.get_pick_prob()
    parent_2 = p2.get_pick_prob()
    child_1_pick = []
    child_2_pick = []
    child_1_place = []
    child_2_place = []

    for i in range(pick_cross):
        child_1_pick.append(parent_1[i])
        child_2_pick.append(parent_2[i])

    for i in range(3 - pick_cross):
        child_1_pick.append(parent_2[i+pick_cross])
        child_2_pick.append(parent_1[i+pick_cross])
```

```python
    parent_1 = p1.get_place_prob()
    parent_2 = p2.get_place_prob()

    for i in range(place_cross):
        child_1_place.append(parent_1[i])
        child_2_place.append(parent_2[i])

    for i in range(6 - place_cross):
        child_1_place.append(parent_2[i+place_cross])
        child_2_place.append(parent_1[i+place_cross])

    child_1 = EvolvedAgent(game)
    child_2 = EvolvedAgent(game)

    set_pick_and_place(child_1, child_1_pick, child_1_place)
    set_pick_and_place(child_2, child_2_pick, child_2_place)

    return child_1, child_2


def set_pick_and_place(individual: EvolvedAgent, pick_prob: list, place_prob:
list):
    '''Sets the probabilites of pick and place according to the parameter
values'''

    # Normalize the rules
    pick = [prob/sum(pick_prob) for prob in pick_prob]
    place = [prob/sum(place_prob) for prob in place_prob]

    individual.set_pick_prob_1(pick[0])
    individual.set_pick_prob_2(pick[1])
    individual.set_pick_prob_3(pick[2])

    individual.set_place_prob_1(place[0])
    individual.set_place_prob_2(place[1])
    individual.set_place_prob_3(place[2])
    individual.set_place_prob_4(place[3])
    individual.set_place_prob_5(place[4])
    individual.set_place_prob_6(place[5])

def run_evolution():
    '''This function runs the evolution algorithm in order to obtain the best
agent'''

    # Initial parameters
    mu = 10
    mutate_rate = 0.25
    iterations = 100
    game = quarto.Quarto()

    individuals = population_init(mu, game)

    for _ in range(iterations):
        print("Iteration: " + str(_))

        # Generate offspring
        offspring_individuals = offspring(individuals, game, mutate_rate)
```

```
        # Combine the offspring with the initial population
        for individual in individuals:
            offspring_individuals.append(individual[1])

        # Determine fitness of said offspring
        individuals = fitness(offspring_individuals, game, 1)

        # Choose the mu best individuals and keep going to the next iteration
        individuals.sort(key = lambda x: x[0], reverse=True)
        individuals = individuals[0:mu]

    # Determine the fitness one last time, this time with 10 matches against
each agent to
    # reduce some of the variance in each game
    final_individuals = []
    for individual in individuals:
        final_individuals.append(individual[1])

    individuals = fitness(final_individuals, game, 10)
    individuals.sort(key = lambda x: x[0], reverse=True)
    return individuals


if __name__ == "__main__":
    individuals = run_evolution()
    best_agent = individuals[0][1]
    print(best_agent)
```

extendedquarto.py:

```
import quarto
from copy import deepcopy
import numpy as np
import testQuarto

class ExtendedQuarto(quarto.Quarto):
    '''
    Extended version of Quarto for more function implementation
    '''
    def __init__(self):
        super().__init__()

    def set_current_player(self, player: int):
        '''
        Function to set current player in the board
        '''
        self._current_player = player


    def get_unchosen_pieces(self) -> list:
        '''
        Get a list of all unchosen pieces
        '''
        unchosen_pieces = list(range(16))
```

```
        for y, row in enumerate(self._board):
            for x, index_at_place in enumerate(row):
                if index_at_place == -1: #empty place
                    continue
                else:
                    unchosen_pieces.remove(index_at_place) #piece already
played

        return unchosen_pieces

    def check_if_possible_to_win(self, piece_idx: int) -> bool:
        '''
        Given a piece, checks if it's possible to win with that piece
        '''
        for y, row in enumerate(self._board):
            for x, index_at_place in enumerate(row):
                if index_at_place == -1:

                    new_arr_board = deepcopy(self._board)
                    new_board = testQuarto.TestQuarto(new_arr_board) #creates
a test quarto to apply the possible move

                    new_board.select(piece_idx) # select the piece we want to
place
                    new_board.place(x,y) # place the piece

                    if (new_board.check_finished() or
new_board.check_winner() != -1): #if the game is over and it's not a tie
                        return True

        return False
```

ga_agent.py:

```
import random
import quarto

class EvolvedAgent(quarto.Player):
    '''Evolved agent using the GA approach'''

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.board = quarto
        self.active_pieces = [i for i in range(16)]

        # Generate the probability of picking each rule for picking and
placing
        pick_prob_not_normalized = [random.random() for i in range(3)]
        pick_prob = [x/sum(pick_prob_not_normalized) for x in
pick_prob_not_normalized]
        self.pick_probability_1 = pick_prob[0]
        self.pick_probability_2 = pick_prob[1]
        self.pick_probability_3 = pick_prob[2]
```

```
        place_probability_not_normalized = [random.random() for i in
range(6)]
        place_prob = [x/sum(place_probability_not_normalized) for x in
place_probability_not_normalized]
        self.place_probability_1 = place_prob[0]
        self.place_probability_2 = place_prob[1]
        self.place_probability_3 = place_prob[2]
        self.place_probability_4 = place_prob[3]
        self.place_probability_5 = place_prob[4]
        self.place_probability_6 = place_prob[5]

        # Create a dictionary containing the attributes of each piece
        self.piece_dict = self.piece_attribute_dict()

        # Create a dictionary containing the dominating lines of each length
        self.dom_line_dict = {}

    def get_pick_prob(self):
        return (self.pick_probability_1, self.pick_probability_2,
self.pick_probability_3)

    def get_place_prob(self):
        return (self.place_probability_1, self.place_probability_2,
self.place_probability_3, \
            self.place_probability_4, self.place_probability_5,
self.place_probability_6)

    def set_pick_prob_1(self, val):
        self.pick_probability_1 = val

    def set_pick_prob_2(self, val):
        self.pick_probability_2 = val

    def set_pick_prob_3(self, val):
        self.pick_probability_3 = val

    def set_place_prob_1(self, val):
        self.place_probability_1 = val

    def set_place_prob_2(self, val):
        self.place_probability_2 = val

    def set_place_prob_3(self, val):
        self.place_probability_3 = val

    def set_place_prob_4(self, val):
        self.place_probability_4 = val

    def set_place_prob_5(self, val):
        self.place_probability_5 = val

    def set_place_prob_6(self, val):
        self.place_probability_6 = val

    def choose_piece(self) -> int:
        '''Decides what rule to be played when picking a piece by the
agent'''
```

```
        self.dominating_line()

        # If there are 3 pieces on the same line with at least 1 common
attribute and a 4th free slot
        if len(self.dom_line_dict['3']) > 0:
            return self.desired_piece(self.dom_line_dict['3'])

        # Genereate the random value to decide what simple rule to play
        val = random.random()

        # Play rule 1
        if val <= self.pick_probability_1:
            #print("Rule 1")
            return self.pick_rule_1()
        # Play rule 2
        elif val <= self.pick_probability_1 + self.pick_probability_2:
            #print("Rule 2")
            return self.pick_rule_2()
        # Play rule 3
        else:
            #print("Rule 3")
            return self.pick_rule_3()

    def place_piece(self) -> tuple([int, int]):
        '''Decides where to place the piece given by the opponent'''

        self.dominating_line()

        # If there are 3 pieces on the same line with at least 1 common
attribute and a 4th free slot
        # and the selected piece shares one of the common attributes
        if len(self.dom_line_dict['3']) > 0:
            for line in self.dom_line_dict['3']:
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        return self.place_piece_specified_line(line)

        # Genereate the random value to decide what simple rule to play
        val = random.random()

        # Play rule 1
        if val <= self.place_probability_1:
            #print("Rule 1")
            return self.place_rule_1()
        # Play rule 2
        elif val <= self.place_probability_1 + self.place_probability_2:
            #print("Rule 2")
            return self.place_rule_2()
        # Play rule 3
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3:
            #print("Rule 3")
            return self.place_rule_3()
        # Play rule 4
        elif val <= self.place_probability_1 + self.place_probability_2 +
```

```
self.place_probability_3 + self.place_probability_4:
            #print("Rule 4")
            return self.place_rule_4()
        # Play rule 5
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3 + self.place_probability_4 +
self.place_probability_5:
            #print("Rule 5")
            return self.place_rule_5()
        # Play rule 6
        else:
            #print("Rule 6")
            return self.place_rule_6()

    def piece_attribute_dict(self) -> dict:
        piece_dict = {}

        for i in range(16):
            piece = self.board.get_piece_charachteristics(i)
            piece_dict[str(i)] = (piece.HIGH, piece.COLOURED, piece.SOLID,
piece.SQUARE)
        return piece_dict

    def dominating_line(self):
        '''Store each row/column/diagonal with at least one shared attribute
in the dom_line_dict based on how long they are.
        Each row, column and diagonal have been assigned a number, where the
rows from top to bottom are
        0 - 3, columns from left to right are 4-7, diagonal 8 and off-
diagonal 9.'''
        board = self.board.get_board_status()

        self.dom_line_dict['1'] = []
        self.dom_line_dict['2'] = []
        self.dom_line_dict['3'] = []
        self.dom_line_dict['4'] = []

        # Check horizontal
        line_counter = 0
        for row in board:
            dominating_result = self.dominating(row, line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check vertical
        for i in range(self.board.BOARD_SIDE):
            dominating_result = self.dominating(board[:,i], line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check diagonal
        diag = []
        off_diag = []
```

```
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        dominating_result = self.dominating(diag, line_counter)
        if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        dominating_result = self.dominating(off_diag, line_counter+1)
        if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        return dominating_result

    def dominating(self, line, line_counter: int) -> list:
        '''Calculate if the current line has a longer line of pieces with at
least one shared attribute
        (higher score) than the previously calculated dominating line. If the
lines contains the same number
        of pieces with at least one shared attribute (same score), then store
both.'''

        line_attributes = []

        # Remove all the empty slots with no piece on it
        if -1 in line:
            line = list(dict.fromkeys(line))
            line.remove(-1)

        # Store the attributes of each piece in the current line
        for elem in line:
            piece = self.board.get_piece_charachteristics(elem)
            line_attributes.append([piece.HIGH, piece.COLOURED, piece.SOLID,
piece.SQUARE])

        # If there are more than 1 piece in the line, check what attributes
are reccuring.
        # Result is a boolean vector where True/False represent the
attributes that are common in the line
        # and None represent an attribute that is not shared. The order is
[High Coloured Solid Square]
        if len(line_attributes) > 1:
            old_result = line_attributes[0]
            for i in range(len(line_attributes)-1):
                result = []
                for j,k in zip(old_result, line_attributes[i+1]):
                    if j==k:
                        result.append(j)
                    else:
                        result.append(None)
                old_result = result
        elif len(line_attributes) == 1:
            # If the line is only 1 piece, all attributes of that piece are
the lines attributes.
            result = line_attributes[0]
        else:
            result = []
```

```
        # Check whether the result vector has an element that is not None,
thus yielding a possible new
        # dominating line
        if any(map(lambda x: not x is None, result)):
            return [len(line_attributes), line_counter, result]
        return None

    def check_attributes(self) -> list:
        '''Calculate the number of each attribute on the board.'''

        board = self.board.get_board_status()
        attribue_values = {}
        attribue_values['High'] = 0
        attribue_values['Low'] = 0
        attribue_values['Color'] = 0
        attribue_values['Noncolor'] = 0
        attribue_values['Solid'] = 0
        attribue_values['Hollow'] = 0
        attribue_values['Square'] = 0
        attribue_values['Circle'] = 0

        for row in board:
            for elem in row:
                if elem >= 0:
                    if self.board.get_piece_charachteristics(elem).HIGH:
                        attribue_values['High'] += 1
                    else:
                        attribue_values['Low'] += 1
                    if self.board.get_piece_charachteristics(elem).COLOURED:
                        attribue_values['Color'] += 1
                    else:
                        attribue_values['Noncolor'] += 1
                    if self.board.get_piece_charachteristics(elem).SOLID:
                        attribue_values['Solid'] += 1
                    else:
                        attribue_values['Hollow'] += 1
                    if self.board.get_piece_charachteristics(elem).SQUARE:
                        attribue_values['Square'] += 1
                    else:
                        attribue_values['Circle'] += 1
        return attribue_values

    def rank_pieces(self, attribute_values: list) -> list:
        '''Ranks the pieces according to their shared attriutes with the
board'''
        val = []

        for i in range(16):
            cur_val = 0
            piece = self.board.get_piece_charachteristics(i)

            if piece.HIGH:
                cur_val += attribute_values['High']
            else:
                cur_val += attribute_values['Low']
            if piece.COLOURED:
```

```
                cur_val += attribute_values['Color']
            else:
                cur_val += attribute_values['Noncolor']
            if piece.SOLID:
                cur_val += attribute_values['Solid']
            else:
                cur_val += attribute_values['Hollow']
            if piece.SQUARE:
                cur_val += attribute_values['Square']
            else:
                cur_val += attribute_values['Circle']
            val.append(cur_val)
        return val

    def desired_piece(self, dom_lines: list) -> list:
        '''This function determines what attributes the piece we choose
should have to avoid a loss'''

        dom_line_attributes = self.dominant_attributes(dom_lines)

        # If one of the attributes and it's corresponding opposite leads to a
win, then we lose no matter what piece we select.
        # E.g. both high and low.
        # If the above is not the case, store the inverse of the attributes
in order to obtain a vector of what attributes
        # we are searching for in a piece. If there's no condition on the
specific attribute, i.e. the piece can be either high or low,
        # store both True and False for the desirec piece in this attribute
spot.
        desired_piece = [[], [], [], []]
        for i in range(len(dom_line_attributes)):
            if len(dom_line_attributes[i]) == 2:
                #print("Lose no matter what")
                return self.pick_rule_3()
            elif len(dom_line_attributes[i]) == 1:
                desired_piece[i].append(not(dom_line_attributes[i][0]))
            else:
                desired_piece[i].append(True)
                desired_piece[i].append(False)

        # Now we look for a piece with the desired attributes
        found, piece_val = self.find_piece_attribute(desired_piece)
        if found:
            #print("Found a desired piece")
            return piece_val
        else:
        # If such a piece does not exist (have been played already), play a
piece at random
            #print("No desired piece left. Picking a piece at random")
            return self.pick_rule_3()

    def dominant_attributes(self, dominant_lines) -> list:
        '''The function returns what are the common attributes in the
dominating lines'''

        dom_line_attributes = [[], [], [], []]
        for i in range(4):
```

```
            for elem in dominant_lines:
                if elem[2][i] != None and elem[2][i] not in
dom_line_attributes[i]:
                    dom_line_attributes[i].append(elem[2][i])
        return dom_line_attributes

    def find_piece_attribute(self, desired_piece) -> tuple:
        '''The function looks for a piece with the desired attributes and
returns True and the piece value if found, otherwise
        it returns false and None'''

        for i in range(16):
            if i not in self.board.get_board_status():
                piece = self.board.get_piece_charachteristics(i)
                if piece.HIGH in desired_piece[0] and piece.COLOURED in
desired_piece[1] \
                    and piece.SOLID in desired_piece[2] and piece.SQUARE in
desired_piece[3]:
                    return (True, i)
        return (False, None)

    def place_piece_specified_line(self, line) -> tuple([int, int]):
        '''This function checks if the selected piece can be placed such that
the agent wins'''
        board = self.board.get_board_status()

        # Check if the piece should be placed in a row
        if line[1] >= 0 and line[1] <= 3:
            # look for an empy spot in the row
            for i in range(4):
                if board[line[1]][i] == -1:
                    return (i, line[1])
        # Check if the piece should be placed in a column
        elif line[1] >= 4 and line[1] <= 7:
            for i in range(4):
                if board[i, line[1]-4] == -1:
                    return (line[1]-4, i)
        # Check if the piece should be placed on the diagonal
        elif line[1] == 8:
            for i in range(4):
                if board[i, i] == -1:
                    return (i, i)
        # Check if the piece should be placed on the off-diagonal
        elif line[1] == 9:
            for i in range(4):
                if board[3-i, i] == -1:
                    return (i, 3-i)

    def count_shared_attributes_in_line(self, piece) -> list:
        '''This function counts the number of shared attributes in the line
with the selected piece. If the piece
        is high and the line contains 2 high pieces, this will count as 2
shared attributes. Thus, longer lines will
        therefore have an advantage in being picked'''
        board = self.board.get_board_status()
        line_val = []
```

```
        # For each row
        for row in board:
            line_val.append(self.count_line(row, piece))

        # For each column
        for i in range(self.board.BOARD_SIDE):
            line_val.append(self.count_line(board[:,i], piece))

        # Diagonals
        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        line_val.append(self.count_line(diag, piece))
        line_val.append(self.count_line(off_diag, piece))

        return line_val


    def count_line(self, line, piece) -> int:
        '''Count the number of shared attributes in the line. If there's no
empty spots left in the line, return -1 in order
        to avoid selecting this line'''
        val = -1
        if -1 in line:
            val  = 0
            line = list(dict.fromkeys(line))
            line.remove(-1)

            for elem in line:
                high, color, solid, square = self.piece_dict[str(elem)]
                if high == self.board.get_piece_charachteristics(piece).HIGH:
                    val += 1
                if color ==
self.board.get_piece_charachteristics(piece).COLOURED:
                    val += 1
                if solid ==
self.board.get_piece_charachteristics(piece).SOLID:
                    val += 1
                if square ==
self.board.get_piece_charachteristics(piece).SQUARE:
                    val += 1
        return val

    def length_of_board_lines(self) -> list:
        '''Counts the lengths of all lines with at least 1 free spot'''

        board = self.board.get_board_status()
        length = []

        for row in board:
            length.append(self.length_of_line(row))

        for i in range(self.board.BOARD_SIDE):
            length.append(self.length_of_line(board[:,i]))
```

```
        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        length.append(self.length_of_line(diag))
        length.append(self.length_of_line(off_diag))

        return length

    def length_of_line(self, line) -> int:
        '''Counts the lenght of the current line'''
        if -1 in line:
            line = list(dict.fromkeys(line))
            line.remove(-1)
            return len(line)
        else:
            return self.board.BOARD_SIDE

    def pick_rule_1(self) -> int:
        '''Pick the piece with the most common attributes with the current
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
        max_pos = piece_ranking.index(max(piece_ranking))
        while max_pos in self.board.get_board_status():
            piece_ranking[max_pos] = -1
            max_pos = piece_ranking.index(max(piece_ranking))
        return max_pos


    def pick_rule_2(self) -> int:
        '''Pick the piece with the least common attributes with the current
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
        min_pos = piece_ranking.index(min(piece_ranking))
        while min_pos in self.board.get_board_status():
            piece_ranking[min_pos] = 100
            min_pos = piece_ranking.index(min(piece_ranking))
        return min_pos

    def pick_rule_3(self) -> int:
        '''Pick a piece at random'''
        random_piece = random.randint(0, 15)
        while random_piece in self.board.get_board_status():
            random_piece = random.randint(0, 15)
        return random_piece

    def place_rule_1(self) -> tuple([int, int]):
        '''Place the piece on the longest uninterrupted line that has no
shared attributes with the current piece'''

        # If there is a dominating line with 3 elements in it, we always
place the piece there if it generates a win.
```

```
        # Thus, we have already checked that we have no common attributes
with all 3-length dominating lines and can therefore place
        # the piece in this line
        if len(self.dom_line_dict['3']) > 0:
            return
self.place_piece_specified_line(self.dom_line_dict['3'][0])

        # For dominating lines of length less than 3 there first need to be a
check to confirm that there are no shared attributes
        # between the selected piece and the line in order to block it.
        for length in ['2', '1']:
            for line in self.dom_line_dict[length]:
                call = True
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        call = False
                        break
                if call:
                    return self.place_piece_specified_line(line)

        # If there are no dominating lines of any lenght we can block, place
it randomly
        return self.place_rule_6()

    def place_rule_2(self):
        '''Place the piece on the longest uninterrupted line with a shared
attribute, might not be the dominating line'''

        # There cannot be a uninterrupted line with 3 elements with a shared
attribute with out piece, if that was the
        # case, the piece would already have been placed there!
        # Thus we only need to check uninterrupted lines of length 2 and 1
and see where our piece has a shared attribute

        for length in ['2', '1']:
            for line in self.dom_line_dict[length]:
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        return self.place_piece_specified_line(line)

        # If there are no uninterrupted lines with a shared attribute, place
it randomly
        return self.place_rule_6()

    def place_rule_3(self):
        '''Places the piece on the line with the most shared attributes
independent of length'''

        piece = self.board.get_selected_piece()
        line_values = self.count_shared_attributes_in_line(piece)
        return self.place_piece_specified_line([0,
line_values.index(max(line_values))])

    def place_rule_4(self):
        '''Place the piece on the shortest line, independent of attributes'''
```

```
        lengths_on_board = self.length_of_board_lines()
        return self.place_piece_specified_line([0,
lengths_on_board.index(min(lengths_on_board))])

    def place_rule_5(self):
        '''Place the piece on the longest line, independent of attributes'''

        lengths_on_board = self.length_of_board_lines()

        # Replace all the lines of length 4 with a -1 as we cannot place a
piece on these lines
        if 4 in lengths_on_board:
            for i in range(len(lengths_on_board)):
                if lengths_on_board[i] == 4:
                    lengths_on_board[i] = -1
        return self.place_piece_specified_line([0,
lengths_on_board.index(max(lengths_on_board))])

    def place_rule_6(self):
        '''Place the piece at random'''
        board = self.board.get_board_status()

        x = random.randint(0, 3)
        y = random.randint(0, 3)

        while board[x,y] != -1:
            x = random.randint(0, 3)
            y = random.randint(0, 3)

        return (y, x)


def set_pick_and_place(individual: EvolvedAgent, pick_prob: list, place_prob:
list):
    '''Sets the probabilites of pick and place according to the parameter
values'''

    # Normalize the rules
    pick = [prob/sum(pick_prob) for prob in pick_prob]
    place = [prob/sum(place_prob) for prob in place_prob]

    individual.set_pick_prob_1(pick[0])
    individual.set_pick_prob_2(pick[1])
    individual.set_pick_prob_3(pick[2])

    individual.set_place_prob_1(place[0])
    individual.set_place_prob_2(place[1])
    individual.set_place_prob_3(place[2])
    individual.set_place_prob_4(place[3])
    individual.set_place_prob_5(place[4])
    individual.set_place_prob_6(place[5])

'''
game = quarto.Quarto()
plus_agent_025 = EvolvedAgent(game)
```

```
plus_pick_025 = (0.06369372715277863, 0.8847749515760825,
0.051531321271138734)
plus_place_025 = (0.5117837231872915, 3.563965752635712e-05,
0.00032016772834300385, 0.07394996224198032, 0.38933362248027764,
0.024576884704581324)


set_pick_and_place(plus_agent_025, plus_pick_025, plus_place_025)
'''
```

ga_dumb.py:

```
import random
import quarto

class UntrainedGAAgent(quarto.Player):
    '''Evolved agent using the GA approach'''

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.board = quarto
        self.active_pieces = [i for i in range(16)]

        # Generate the probability of picking each rule for picking and
placing
        pick_prob_not_normalized = [random.random() for i in range(3)]
        pick_prob = [x/sum(pick_prob_not_normalized) for x in
pick_prob_not_normalized]
        self.pick_probability_1 = pick_prob[0]
        self.pick_probability_2 = pick_prob[1]
        self.pick_probability_3 = pick_prob[2]

        place_probability_not_normalized = [random.random() for i in
range(6)]
        place_prob = [x/sum(place_probability_not_normalized) for x in
place_probability_not_normalized]
        self.place_probability_1 = place_prob[0]
        self.place_probability_2 = place_prob[1]
        self.place_probability_3 = place_prob[2]
        self.place_probability_4 = place_prob[3]
        self.place_probability_5 = place_prob[4]
        self.place_probability_6 = place_prob[5]

        # Create a dictionary containing the attributes of each piece
        self.piece_dict = self.piece_attribute_dict()

        # Create a dictionary containing the dominating lines of each length
        self.dom_line_dict = {}

    def get_pick_prob(self):
        return (self.pick_probability_1, self.pick_probability_2,
self.pick_probability_3)

    def get_place_prob(self):
        return (self.place_probability_1, self.place_probability_2,
```

```
self.place_probability_3, \
            self.place_probability_4, self.place_probability_5,
self.place_probability_6)

    def set_pick_prob_1(self, val):
        self.pick_probability_1 = val

    def set_pick_prob_2(self, val):
        self.pick_probability_2 = val

    def set_pick_prob_3(self, val):
        self.pick_probability_3 = val

    def set_place_prob_1(self, val):
        self.place_probability_1 = val

    def set_place_prob_2(self, val):
        self.place_probability_2 = val

    def set_place_prob_3(self, val):
        self.place_probability_3 = val

    def set_place_prob_4(self, val):
        self.place_probability_4 = val

    def set_place_prob_5(self, val):
        self.place_probability_5 = val

    def set_place_prob_6(self, val):
        self.place_probability_6 = val

    def choose_piece(self) -> int:
        '''Decides what rule to be played when picking a piece by the
agent'''

        self.dominating_line()

        '''# If there are 3 pieces on the same line with at least 1 common
attribute and a 4th free slot
        if len(self.dom_line_dict['3']) > 0:
            return self.desired_piece(self.dom_line_dict['3'])'''

        # Genereate the random value to decide what simple rule to play
        val = random.random()

        # Play rule 1
        if val <= self.pick_probability_1:
            #print("Rule 1")
            return self.pick_rule_1()
        # Play rule 2
        elif val <= self.pick_probability_1 + self.pick_probability_2:
            #print("Rule 2")
            return self.pick_rule_2()
        # Play rule 3
        else:
            #print("Rule 3")
            return self.pick_rule_3()
```

```
    def place_piece(self) -> tuple([int, int]):
        '''Decides where to place the piece given by the opponent'''

        self.dominating_line()

        '''# If there are 3 pieces on the same line with at least 1 common
attribute and a 4th free slot
        # and the selected piece shares one of the common attributes
        if len(self.dom_line_dict['3']) > 0:
            for line in self.dom_line_dict['3']:
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        return self.place_piece_specified_line(line)'''

        # Genereate the random value to decide what simple rule to play
        val = random.random()

        # Play rule 1
        if val <= self.place_probability_1:
            #print("Rule 1")
            return self.place_rule_1()
        # Play rule 2
        elif val <= self.place_probability_1 + self.place_probability_2:
            #print("Rule 2")
            return self.place_rule_2()
        # Play rule 3
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3:
            #print("Rule 3")
            return self.place_rule_3()
        # Play rule 4
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3 + self.place_probability_4:
            #print("Rule 4")
            return self.place_rule_4()
        # Play rule 5
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3 + self.place_probability_4 +
self.place_probability_5:
            #print("Rule 5")
            return self.place_rule_5()
        # Play rule 6
        else:
            #print("Rule 6")
            return self.place_rule_6()

    def piece_attribute_dict(self) -> dict:
        piece_dict = {}

        for i in range(16):
            piece = self.board.get_piece_charachteristics(i)
            piece_dict[str(i)] = (piece.HIGH, piece.COLOURED, piece.SOLID,
piece.SQUARE)
        return piece_dict
```

```
    def dominating_line(self):
        '''Store each row/column/diagonal with at least one shared attribute
in the dom_line_dict based on how long they are.
        Each row, column and diagonal have been assigned a number, where the
rows from top to bottom are
        0 - 3, columns from left to right are 4-7, diagonal 8 and off-
diagonal 9.'''
        board = self.board.get_board_status()

        self.dom_line_dict['1'] = []
        self.dom_line_dict['2'] = []
        self.dom_line_dict['3'] = []
        self.dom_line_dict['4'] = []

        # Check horizontal
        line_counter = 0
        for row in board:
            dominating_result = self.dominating(row, line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check vertical
        for i in range(self.board.BOARD_SIDE):
            dominating_result = self.dominating(board[:,i], line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check diagonal
        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        dominating_result = self.dominating(diag, line_counter)
        if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        dominating_result = self.dominating(off_diag, line_counter+1)
        if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        return dominating_result

    def dominating(self, line, line_counter: int) -> list:
        '''Calculate if the current line has a longer line of pieces with at
least one shared attribute
        (higher score) than the previously calculated dominating line. If the
lines contains the same number
        of pieces with at least one shared attribute (same score), then store
both.'''

        line_attributes = []
```

```
        # Remove all the empty slots with no piece on it
        if -1 in line:
            line = list(dict.fromkeys(line))
            line.remove(-1)

        # Store the attributes of each piece in the current line
        for elem in line:
            piece = self.board.get_piece_charachteristics(elem)
            line_attributes.append([piece.HIGH, piece.COLOURED, piece.SOLID,
piece.SQUARE])

        # If there are more than 1 piece in the line, check what attributes
are reccuring.
        # Result is a boolean vector where True/False represent the
attributes that are common in the line
        # and None represent an attribute that is not shared. The order is
[High Coloured Solid Square]
        if len(line_attributes) > 1:
            old_result = line_attributes[0]
            for i in range(len(line_attributes)-1):
                result = []
                for j,k in zip(old_result, line_attributes[i+1]):
                    if j==k:
                        result.append(j)
                    else:
                        result.append(None)
                old_result = result
        elif len(line_attributes) == 1:
            # If the line is only 1 piece, all attributes of that piece are
the lines attributes.
            result = line_attributes[0]
        else:
            result = []

        # Check whether the result vector has an element that is not None,
thus yielding a possible new
        # dominating line
        if any(map(lambda x: not x is None, result)):
            return [len(line_attributes), line_counter, result]
        return None

    def check_attributes(self) -> list:
        '''Calculate the number of each attribute on the board.'''

        board = self.board.get_board_status()
        attribue_values = {}
        attribue_values['High'] = 0
        attribue_values['Low'] = 0
        attribue_values['Color'] = 0
        attribue_values['Noncolor'] = 0
        attribue_values['Solid'] = 0
        attribue_values['Hollow'] = 0
        attribue_values['Square'] = 0
        attribue_values['Circle'] = 0

        for row in board:
```

```python
            for elem in row:
                if elem >= 0:
                    if self.board.get_piece_charachteristics(elem).HIGH:
                        attribue_values['High'] += 1
                    else:
                        attribue_values['Low'] += 1
                    if self.board.get_piece_charachteristics(elem).COLOURED:
                        attribue_values['Color'] += 1
                    else:
                        attribue_values['Noncolor'] += 1
                    if self.board.get_piece_charachteristics(elem).SOLID:
                        attribue_values['Solid'] += 1
                    else:
                        attribue_values['Hollow'] += 1
                    if self.board.get_piece_charachteristics(elem).SQUARE:
                        attribue_values['Square'] += 1
                    else:
                        attribue_values['Circle'] += 1
        return attribue_values

    def rank_pieces(self, attribute_values: list) -> list:
        '''Ranks the pieces according to their shared attriutes with the 
board'''
        val = []

        for i in range(16):
            cur_val = 0
            piece = self.board.get_piece_charachteristics(i)

            if piece.HIGH:
                cur_val += attribute_values['High']
            else:
                cur_val += attribute_values['Low']
            if piece.COLOURED:
                cur_val += attribute_values['Color']
            else:
                cur_val += attribute_values['Noncolor']
            if piece.SOLID:
                cur_val += attribute_values['Solid']
            else:
                cur_val += attribute_values['Hollow']
            if piece.SQUARE:
                cur_val += attribute_values['Square']
            else:
                cur_val += attribute_values['Circle']
            val.append(cur_val)
        return val

    def desired_piece(self, dom_lines: list) -> list:
        '''This function determines what attributes the piece we choose 
should have to avoid a loss'''

        dom_line_attributes = self.dominant_attributes(dom_lines)

        # If one of the attributes and it's corresponding opposite leads to a 
win, then we lose no matter what piece we select.
        # E.g. both high and low.
```

```
        # If the above is not the case, store the inverse of the attributes
in order to obtain a vector of what attributes
        # we are searching for in a piece. If there's no condition on the
specific attribute, i.e. the piece can be either high or low,
        # store both True and False for the desirec piece in this attribute
spot.
        desired_piece = [[], [], [], []]
        for i in range(len(dom_line_attributes)):
            if len(dom_line_attributes[i]) == 2:
                #print("Lose no matter what")
                return self.pick_rule_3()
            elif len(dom_line_attributes[i]) == 1:
                desired_piece[i].append(not(dom_line_attributes[i][0]))
            else:
                desired_piece[i].append(True)
                desired_piece[i].append(False)

        # Now we look for a piece with the desired attributes
        found, piece_val = self.find_piece_attribute(desired_piece)
        if found:
            #print("Found a desired piece")
            return piece_val
        else:
        # If such a piece does not exist (have been played already), play a
piece at random
            #print("No desired piece left. Picking a piece at random")
            return self.pick_rule_3()

    def dominant_attributes(self, dominant_lines) -> list:
        '''The function returns what are the common attributes in the
dominating lines'''

        dom_line_attributes = [[], [], [], []]
        for i in range(4):
            for elem in dominant_lines:
                if elem[2][i] != None and elem[2][i] not in
dom_line_attributes[i]:
                    dom_line_attributes[i].append(elem[2][i])
        return dom_line_attributes

    def find_piece_attribute(self, desired_piece) -> tuple:
        '''The function looks for a piece with the desired attributes and
returns True and the piece value if found, otherwise
        it returns false and None'''

        for i in range(16):
            if i not in self.board.get_board_status():
                piece = self.board.get_piece_charachteristics(i)
                if piece.HIGH in desired_piece[0] and piece.COLOURED in
desired_piece[1] \
                    and piece.SOLID in desired_piece[2] and piece.SQUARE in
desired_piece[3]:
                    return (True, i)
        return (False, None)

    def place_piece_specified_line(self, line) -> tuple([int, int]):
        '''This function checks if the selected piece can be placed such that
```

```
the agent wins'''
        board = self.board.get_board_status()

        # Check if the piece should be placed in a row
        if line[1] >= 0 and line[1] <= 3:
            # look for an empy spot in the row
            for i in range(4):
                if board[line[1]][i] == -1:
                    return (i, line[1])
        # Check if the piece should be placed in a column
        elif line[1] >= 4 and line[1] <= 7:
            for i in range(4):
                if board[i, line[1]-4] == -1:
                    return (line[1]-4, i)
        # Check if the piece should be placed on the diagonal
        elif line[1] == 8:
            for i in range(4):
                if board[i, i] == -1:
                    return (i, i)
        # Check if the piece should be placed on the off-diagonal
        elif line[1] == 9:
            for i in range(4):
                if board[3-i, i] == -1:
                    return (i, 3-i)

    def count_shared_attributes_in_line(self, piece) -> list:
        '''This function counts the number of shared attributes in the line
with the selected piece. If the piece
        is high and the line contains 2 high pieces, this will count as 2
shared attributes. Thus, longer lines will
        therefore have an advantage in being picked'''
        board = self.board.get_board_status()
        line_val = []

        # For each row
        for row in board:
            line_val.append(self.count_line(row, piece))

        # For each column
        for i in range(self.board.BOARD_SIDE):
            line_val.append(self.count_line(board[:,i], piece))

        # Diagonals
        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        line_val.append(self.count_line(diag, piece))
        line_val.append(self.count_line(off_diag, piece))

        return line_val


    def count_line(self, line, piece) -> int:
        '''Count the number of shared attributes in the line. If there's no
```

```
empty spots left in the line, return -1 in order
        to avoid selecting this line'''
        val = -1
        if -1 in line:
            val  = 0
            line = list(dict.fromkeys(line))
            line.remove(-1)

            for elem in line:
                high, color, solid, square = self.piece_dict[str(elem)]
                if high == self.board.get_piece_charachteristics(piece).HIGH:
                    val += 1
                if color ==
self.board.get_piece_charachteristics(piece).COLOURED:
                    val += 1
                if solid ==
self.board.get_piece_charachteristics(piece).SOLID:
                    val += 1
                if square ==
self.board.get_piece_charachteristics(piece).SQUARE:
                    val += 1
        return val

    def length_of_board_lines(self) -> list:
        '''Counts the lengths of all lines with at least 1 free spot'''

        board = self.board.get_board_status()
        length = []

        for row in board:
            length.append(self.length_of_line(row))

        for i in range(self.board.BOARD_SIDE):
            length.append(self.length_of_line(board[:,i]))

        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        length.append(self.length_of_line(diag))
        length.append(self.length_of_line(off_diag))

        return length

    def length_of_line(self, line) -> int:
        '''Counts the lenght of the current line'''
        if -1 in line:
            line = list(dict.fromkeys(line))
            line.remove(-1)
            return len(line)
        else:
            return self.board.BOARD_SIDE

    def pick_rule_1(self) -> int:
        '''Pick the piece with the most common attributes with the current
```

```
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
        max_pos = piece_ranking.index(max(piece_ranking))
        while max_pos in self.board.get_board_status():
            piece_ranking[max_pos] = -1
            max_pos = piece_ranking.index(max(piece_ranking))
        return max_pos


    def pick_rule_2(self) -> int:
        '''Pick the piece with the least common attributes with the current
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
        min_pos = piece_ranking.index(min(piece_ranking))
        while min_pos in self.board.get_board_status():
            piece_ranking[min_pos] = 100
            min_pos = piece_ranking.index(min(piece_ranking))
        return min_pos

    def pick_rule_3(self) -> int:
        '''Pick a piece at random'''
        random_piece = random.randint(0, 15)
        while random_piece in self.board.get_board_status():
            random_piece = random.randint(0, 15)
        return random_piece

    def place_rule_1(self) -> tuple([int, int]):
        '''Place the piece on the longest uninterrupted line that has no
shared attributes with the current piece'''

        # If there is a dominating line with 3 elements in it, we always
place the piece there if it generates a win.
        # Thus, we have already checked that we have no common attributes
with all 3-length dominating lines and can therefore place
        # the piece in this line
        if len(self.dom_line_dict['3']) > 0:
            return
self.place_piece_specified_line(self.dom_line_dict['3'][0])

        # For dominating lines of length less than 3 there first need to be a
check to confirm that there are no shared attributes
        # between the selected piece and the line in order to block it.
        for length in ['2', '1']:
            for line in self.dom_line_dict[length]:
                call = True
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        call = False
                        break
                if call:
                    return self.place_piece_specified_line(line)

        # If there are no dominating lines of any lenght we can block, place
it randomly
```

```
        return self.place_rule_6()

    def place_rule_2(self):
        '''Place the piece on the longest uninterrupted line with a shared
attribute, might not be the dominating line'''

        # There cannot be a uninterrupted line with 3 elements with a shared
attribute with out piece, if that was the
        # case, the piece would already have been placed there!
        # Thus we only need to check uninterrupted lines of length 2 and 1
and see where our piece has a shared attribute

        for length in ['2', '1']:
            for line in self.dom_line_dict[length]:
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        return self.place_piece_specified_line(line)

        # If there are no uninterrupted lines with a shared attribute, place
it randomly
        return self.place_rule_6()

    def place_rule_3(self):
        '''Places the piece on the line with the most shared attributes
independent of length'''

        piece = self.board.get_selected_piece()
        line_values = self.count_shared_attributes_in_line(piece)
        return self.place_piece_specified_line([0,
line_values.index(max(line_values))])

    def place_rule_4(self):
        '''Place the piece on the shortest line, independent of attributes'''

        lengths_on_board = self.length_of_board_lines()
        return self.place_piece_specified_line([0,
lengths_on_board.index(min(lengths_on_board))])

    def place_rule_5(self):
        '''Place the piece on the longest line, independent of attributes'''

        lengths_on_board = self.length_of_board_lines()

        # Replace all the lines of length 4 with a -1 as we cannot place a
piece on these lines
        if 4 in lengths_on_board:
            for i in range(len(lengths_on_board)):
                if lengths_on_board[i] == 4:
                    lengths_on_board[i] = -1
        return self.place_piece_specified_line([0,
lengths_on_board.index(max(lengths_on_board))])

    def place_rule_6(self):
        '''Place the piece at random'''
        board = self.board.get_board_status()
```

```
        x = random.randint(0, 3)
        y = random.randint(0, 3)

        while board[x,y] != -1:
            x = random.randint(0, 3)
            y = random.randint(0, 3)

        return (y, x)
```

ga_less_dumb.py:

```
import random
import quarto

class LeastDumbAgent(quarto.Player):
    '''Evolved agent using the GA approach'''

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.board = quarto
        self.active_pieces = [i for i in range(16)]

        # Generate the probability of picking each rule for picking and
placing
        pick_prob_not_normalized = [random.random() for i in range(3)]
        pick_prob = [x/sum(pick_prob_not_normalized) for x in
pick_prob_not_normalized]
        self.pick_probability_1 = pick_prob[0]
        self.pick_probability_2 = pick_prob[1]
        self.pick_probability_3 = pick_prob[2]

        place_probability_not_normalized = [random.random() for i in
range(6)]
        place_prob = [x/sum(place_probability_not_normalized) for x in
place_probability_not_normalized]
        self.place_probability_1 = place_prob[0]
        self.place_probability_2 = place_prob[1]
        self.place_probability_3 = place_prob[2]
        self.place_probability_4 = place_prob[3]
        self.place_probability_5 = place_prob[4]
        self.place_probability_6 = place_prob[5]

        # Create a dictionary containing the attributes of each piece
        self.piece_dict = self.piece_attribute_dict()

        # Create a dictionary containing the dominating lines of each length
        self.dom_line_dict = {}

    def get_pick_prob(self):
        return (self.pick_probability_1, self.pick_probability_2,
self.pick_probability_3)

    def get_place_prob(self):
        return (self.place_probability_1, self.place_probability_2,
self.place_probability_3, \
            self.place_probability_4, self.place_probability_5,
self.place_probability_6)
```

```python
    def set_pick_prob_1(self, val):
        self.pick_probability_1 = val

    def set_pick_prob_2(self, val):
        self.pick_probability_2 = val

    def set_pick_prob_3(self, val):
        self.pick_probability_3 = val

    def set_place_prob_1(self, val):
        self.place_probability_1 = val

    def set_place_prob_2(self, val):
        self.place_probability_2 = val

    def set_place_prob_3(self, val):
        self.place_probability_3 = val

    def set_place_prob_4(self, val):
        self.place_probability_4 = val

    def set_place_prob_5(self, val):
        self.place_probability_5 = val

    def set_place_prob_6(self, val):
        self.place_probability_6 = val

    def choose_piece(self) -> int:
        '''Decides what rule to be played when picking a piece by the
agent'''

        self.dominating_line()

        # If there are 3 pieces on the same line with at least 1 common
attribute and a 4th free slot
        if len(self.dom_line_dict['3']) > 0:
            return self.desired_piece(self.dom_line_dict['3'])

        # Genereate the random value to decide what simple rule to play
        val = random.random()

        # Play rule 1
        if val <= self.pick_probability_1:
            #print("Rule 1")
            return self.pick_rule_1()
        # Play rule 2
        elif val <= self.pick_probability_1 + self.pick_probability_2:
            #print("Rule 2")
            return self.pick_rule_2()
        # Play rule 3
        else:
            #print("Rule 3")
            return self.pick_rule_3()

    def place_piece(self) -> tuple([int, int]):
        '''Decides where to place the piece given by the opponent'''
```

```
        self.dominating_line()

        '''# If there are 3 pieces on the same line with at least 1 common
attribute and a 4th free slot
        # and the selected piece shares one of the common attributes
        if len(self.dom_line_dict['3']) > 0:
            for line in self.dom_line_dict['3']:
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        return self.place_piece_specified_line(line)'''

        # Genereate the random value to decide what simple rule to play
        val = random.random()

        # Play rule 1
        if val <= self.place_probability_1:
            #print("Rule 1")
            return self.place_rule_1()
        # Play rule 2
        elif val <= self.place_probability_1 + self.place_probability_2:
            #print("Rule 2")
            return self.place_rule_2()
        # Play rule 3
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3:
            #print("Rule 3")
            return self.place_rule_3()
        # Play rule 4
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3 + self.place_probability_4:
            #print("Rule 4")
            return self.place_rule_4()
        # Play rule 5
        elif val <= self.place_probability_1 + self.place_probability_2 +
self.place_probability_3 + self.place_probability_4 +
self.place_probability_5:
            #print("Rule 5")
            return self.place_rule_5()
        # Play rule 6
        else:
            #print("Rule 6")
            return self.place_rule_6()

    def piece_attribute_dict(self) -> dict:
        piece_dict = {}

        for i in range(16):
            piece = self.board.get_piece_charachteristics(i)
            piece_dict[str(i)] = (piece.HIGH, piece.COLOURED, piece.SOLID,
piece.SQUARE)
        return piece_dict

    def dominating_line(self):
        '''Store each row/column/diagonal with at least one shared attribute
in the dom_line_dict based on how long they are.
```

```
        Each row, column and diagonal have been assigned a number, where the
rows from top to bottom are
        0 - 3, columns from left to right are 4-7, diagonal 8 and off-
diagonal 9.'''
        board = self.board.get_board_status()

        self.dom_line_dict['1'] = []
        self.dom_line_dict['2'] = []
        self.dom_line_dict['3'] = []
        self.dom_line_dict['4'] = []

        # Check horizontal
        line_counter = 0
        for row in board:
            dominating_result = self.dominating(row, line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check vertical
        for i in range(self.board.BOARD_SIDE):
            dominating_result = self.dominating(board[:,i], line_counter)
            if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
            line_counter += 1

        # Check diagonal
        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        dominating_result = self.dominating(diag, line_counter)
        if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        dominating_result = self.dominating(off_diag, line_counter+1)
        if dominating_result != None:

self.dom_line_dict[str(dominating_result[0])].append(dominating_result)
        return dominating_result

    def dominating(self, line, line_counter: int) -> list:
        '''Calculate if the current line has a longer line of pieces with at
least one shared attribute
        (higher score) than the previously calculated dominating line. If the
lines contains the same number
        of pieces with at least one shared attribute (same score), then store
both.'''

        line_attributes = []

        # Remove all the empty slots with no piece on it
        if -1 in line:
```

```
            line = list(dict.fromkeys(line))
            line.remove(-1)

        # Store the attributes of each piece in the current line
        for elem in line:
            piece = self.board.get_piece_charachteristics(elem)
            line_attributes.append([piece.HIGH, piece.COLOURED, piece.SOLID,
piece.SQUARE])

        # If there are more than 1 piece in the line, check what attributes
are reccuring.
        # Result is a boolean vector where True/False represent the
attributes that are common in the line
        # and None represent an attribute that is not shared. The order is
[High Coloured Solid Square]
        if len(line_attributes) > 1:
            old_result = line_attributes[0]
            for i in range(len(line_attributes)-1):
                result = []
                for j,k in zip(old_result, line_attributes[i+1]):
                    if j==k:
                        result.append(j)
                    else:
                        result.append(None)
                old_result = result
        elif len(line_attributes) == 1:
            # If the line is only 1 piece, all attributes of that piece are
the lines attributes.
            result = line_attributes[0]
        else:
            result = []

        # Check whether the result vector has an element that is not None,
thus yielding a possible new
        # dominating line
        if any(map(lambda x: not x is None, result)):
            return [len(line_attributes), line_counter, result]
        return None

    def check_attributes(self) -> list:
        '''Calculate the number of each attribute on the board.'''

        board = self.board.get_board_status()
        attribue_values = {}
        attribue_values['High'] = 0
        attribue_values['Low'] = 0
        attribue_values['Color'] = 0
        attribue_values['Noncolor'] = 0
        attribue_values['Solid'] = 0
        attribue_values['Hollow'] = 0
        attribue_values['Square'] = 0
        attribue_values['Circle'] = 0

        for row in board:
            for elem in row:
                if elem >= 0:
                    if self.board.get_piece_charachteristics(elem).HIGH:
```

```python
                        attribue_values['High'] += 1
                    else:
                        attribue_values['Low'] += 1
                    if self.board.get_piece_charachteristics(elem).COLOURED:
                        attribue_values['Color'] += 1
                    else:
                        attribue_values['Noncolor'] += 1
                    if self.board.get_piece_charachteristics(elem).SOLID:
                        attribue_values['Solid'] += 1
                    else:
                        attribue_values['Hollow'] += 1
                    if self.board.get_piece_charachteristics(elem).SQUARE:
                        attribue_values['Square'] += 1
                    else:
                        attribue_values['Circle'] += 1
        return attribue_values

    def rank_pieces(self, attribute_values: list) -> list:
        '''Ranks the pieces according to their shared attriutes with the
board'''
        val = []

        for i in range(16):
            cur_val = 0
            piece = self.board.get_piece_charachteristics(i)

            if piece.HIGH:
                cur_val += attribute_values['High']
            else:
                cur_val += attribute_values['Low']
            if piece.COLOURED:
                cur_val += attribute_values['Color']
            else:
                cur_val += attribute_values['Noncolor']
            if piece.SOLID:
                cur_val += attribute_values['Solid']
            else:
                cur_val += attribute_values['Hollow']
            if piece.SQUARE:
                cur_val += attribute_values['Square']
            else:
                cur_val += attribute_values['Circle']
            val.append(cur_val)
        return val

    def desired_piece(self, dom_lines: list) -> list:
        '''This function determines what attributes the piece we choose
should have to avoid a loss'''

        dom_line_attributes = self.dominant_attributes(dom_lines)

        # If one of the attributes and it's corresponding opposite leads to a
win, then we lose no matter what piece we select.
        # E.g. both high and low.
        # If the above is not the case, store the inverse of the attributes
in order to obtain a vector of what attributes
        # we are searching for in a piece. If there's no condition on the
```

```
specific attribute, i.e. the piece can be either high or low,
        # store both True and False for the desirec piece in this attribute
spot.
        desired_piece = [[], [], [], []]
        for i in range(len(dom_line_attributes)):
            if len(dom_line_attributes[i]) == 2:
                #print("Lose no matter what")
                return self.pick_rule_3()
            elif len(dom_line_attributes[i]) == 1:
                desired_piece[i].append(not(dom_line_attributes[i][0]))
            else:
                desired_piece[i].append(True)
                desired_piece[i].append(False)

        # Now we look for a piece with the desired attributes
        found, piece_val = self.find_piece_attribute(desired_piece)
        if found:
            #print("Found a desired piece")
            return piece_val
        else:
        # If such a piece does not exist (have been played already), play a
piece at random
            #print("No desired piece left. Picking a piece at random")
            return self.pick_rule_3()

    def dominant_attributes(self, dominant_lines) -> list:
        '''The function returns what are the common attributes in the
dominating lines'''

        dom_line_attributes = [[], [], [], []]
        for i in range(4):
            for elem in dominant_lines:
                if elem[2][i] != None and elem[2][i] not in
dom_line_attributes[i]:
                    dom_line_attributes[i].append(elem[2][i])
        return dom_line_attributes

    def find_piece_attribute(self, desired_piece) -> tuple:
        '''The function looks for a piece with the desired attributes and
returns True and the piece value if found, otherwise
        it returns false and None'''

        for i in range(16):
            if i not in self.board.get_board_status():
                piece = self.board.get_piece_charachteristics(i)
                if piece.HIGH in desired_piece[0] and piece.COLOURED in
desired_piece[1] \
                    and piece.SOLID in desired_piece[2] and piece.SQUARE in
desired_piece[3]:
                    return (True, i)
        return (False, None)

    def place_piece_specified_line(self, line) -> tuple([int, int]):
        '''This function checks if the selected piece can be placed such that
the agent wins'''
        board = self.board.get_board_status()
```

```
        # Check if the piece should be placed in a row
        if line[1] >= 0 and line[1] <= 3:
            # look for an empy spot in the row
            for i in range(4):
                if board[line[1]][i] == -1:
                    return (i, line[1])
        # Check if the piece should be placed in a column
        elif line[1] >= 4 and line[1] <= 7:
            for i in range(4):
                if board[i, line[1]-4] == -1:
                    return (line[1]-4, i)
        # Check if the piece should be placed on the diagonal
        elif line[1] == 8:
            for i in range(4):
                if board[i, i] == -1:
                    return (i, i)
        # Check if the piece should be placed on the off-diagonal
        elif line[1] == 9:
            for i in range(4):
                if board[3-i, i] == -1:
                    return (i, 3-i)

    def count_shared_attributes_in_line(self, piece) -> list:
        '''This function counts the number of shared attributes in the line
with the selected piece. If the piece
        is high and the line contains 2 high pieces, this will count as 2
shared attributes. Thus, longer lines will
        therefore have an advantage in being picked'''
        board = self.board.get_board_status()
        line_val = []

        # For each row
        for row in board:
            line_val.append(self.count_line(row, piece))

        # For each column
        for i in range(self.board.BOARD_SIDE):
            line_val.append(self.count_line(board[:,i], piece))

        # Diagonals
        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        line_val.append(self.count_line(diag, piece))
        line_val.append(self.count_line(off_diag, piece))

        return line_val


    def count_line(self, line, piece) -> int:
        '''Count the number of shared attributes in the line. If there's no
empty spots left in the line, return -1 in order
        to avoid selecting this line'''
        val = -1
```

```
        if -1 in line:
            val  = 0
            line = list(dict.fromkeys(line))
            line.remove(-1)

            for elem in line:
                high, color, solid, square = self.piece_dict[str(elem)]
                if high == self.board.get_piece_charachteristics(piece).HIGH:
                    val += 1
                if color ==
self.board.get_piece_charachteristics(piece).COLOURED:
                    val += 1
                if solid ==
self.board.get_piece_charachteristics(piece).SOLID:
                    val += 1
                if square ==
self.board.get_piece_charachteristics(piece).SQUARE:
                    val += 1
        return val

    def length_of_board_lines(self) -> list:
        '''Counts the lengths of all lines with at least 1 free spot'''

        board = self.board.get_board_status()
        length = []

        for row in board:
            length.append(self.length_of_line(row))

        for i in range(self.board.BOARD_SIDE):
            length.append(self.length_of_line(board[:,i]))

        diag = []
        off_diag = []
        for i in range(len(board)):
            diag.append(board[i,i])
            off_diag.append(board[self.board.BOARD_SIDE-1-i, i])

        length.append(self.length_of_line(diag))
        length.append(self.length_of_line(off_diag))

        return length

    def length_of_line(self, line) -> int:
        '''Counts the lenght of the current line'''
        if -1 in line:
            line = list(dict.fromkeys(line))
            line.remove(-1)
            return len(line)
        else:
            return self.board.BOARD_SIDE

    def pick_rule_1(self) -> int:
        '''Pick the piece with the most common attributes with the current
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
```

```
        max_pos = piece_ranking.index(max(piece_ranking))
        while max_pos in self.board.get_board_status():
            piece_ranking[max_pos] = -1
            max_pos = piece_ranking.index(max(piece_ranking))
        return max_pos


    def pick_rule_2(self) -> int:
        '''Pick the piece with the least common attributes with the current
board'''
        attribute_values = self.check_attributes()
        piece_ranking = self.rank_pieces(attribute_values)
        min_pos = piece_ranking.index(min(piece_ranking))
        while min_pos in self.board.get_board_status():
            piece_ranking[min_pos] = 100
            min_pos = piece_ranking.index(min(piece_ranking))
        return min_pos

    def pick_rule_3(self) -> int:
        '''Pick a piece at random'''
        random_piece = random.randint(0, 15)
        while random_piece in self.board.get_board_status():
            random_piece = random.randint(0, 15)
        return random_piece

    def place_rule_1(self) -> tuple([int, int]):
        '''Place the piece on the longest uninterrupted line that has no
shared attributes with the current piece'''

        # If there is a dominating line with 3 elements in it, we always
place the piece there if it generates a win.
        # Thus, we have already checked that we have no common attributes
with all 3-length dominating lines and can therefore place
        # the piece in this line
        if len(self.dom_line_dict['3']) > 0:
            return
self.place_piece_specified_line(self.dom_line_dict['3'][0])

        # For dominating lines of length less than 3 there first need to be a
check to confirm that there are no shared attributes
        # between the selected piece and the line in order to block it.
        for length in ['2', '1']:
            for line in self.dom_line_dict[length]:
                call = True
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        call = False
                        break
                if call:
                    return self.place_piece_specified_line(line)

        # If there are no dominating lines of any lenght we can block, place
it randomly
        return self.place_rule_6()

    def place_rule_2(self):
```

```
        '''Place the piece on the longest uninterrupted line with a shared
attribute, might not be the dominating line'''

        # There cannot be a uninterrupted line with 3 elements with a shared
attribute with out piece, if that was the
        # case, the piece would already have been placed there!
        # Thus we only need to check uninterrupted lines of length 2 and 1
and see where our piece has a shared attribute

        for length in ['2', '1']:
            for line in self.dom_line_dict[length]:
                for j,k in zip(line[2],
list(self.piece_dict[str(self.board.get_selected_piece())])):
                    if j != None and j == k:
                        return self.place_piece_specified_line(line)

        # If there are no uninterrupted lines with a shared attribute, place
it randomly
        return self.place_rule_6()

    def place_rule_3(self):
        '''Places the piece on the line with the most shared attributes
independent of length'''

        piece = self.board.get_selected_piece()
        line_values = self.count_shared_attributes_in_line(piece)
        return self.place_piece_specified_line([0,
line_values.index(max(line_values))])

    def place_rule_4(self):
        '''Place the piece on the shortest line, independent of attributes'''

        lengths_on_board = self.length_of_board_lines()
        return self.place_piece_specified_line([0,
lengths_on_board.index(min(lengths_on_board))])

    def place_rule_5(self):
        '''Place the piece on the longest line, independent of attributes'''

        lengths_on_board = self.length_of_board_lines()

        # Replace all the lines of length 4 with a -1 as we cannot place a
piece on these lines
        if 4 in lengths_on_board:
            for i in range(len(lengths_on_board)):
                if lengths_on_board[i] == 4:
                    lengths_on_board[i] = -1
        return self.place_piece_specified_line([0,
lengths_on_board.index(max(lengths_on_board))])

    def place_rule_6(self):
        '''Place the piece at random'''
        board = self.board.get_board_status()

        x = random.randint(0, 3)
        y = random.randint(0, 3)
```

```
        while board[x,y] != -1:
            x = random.randint(0, 3)
            y = random.randint(0, 3)

        return (y, x)
```

main.py:

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

import logging
import argparse
import random
import quarto
import ga_agent
from trained_rl import *
from opponent_agents import DumbAgent


class RandomPlayer(quarto.Player):
    """Random player"""
    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)

    def choose_piece(self) -> int:
        test = random.randint(0, 15)
        #logging.debug(f"Selected piece random player: {test}")
        return test

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


def main():
    game = quarto.Quarto()
    rl_agent = TrainedRL(game)
    random_agent = RandomPlayer(game)
    game.set_players((random_agent, rl_agent))
    winner = game.run()
    logging.warning(f"main: Winner: player {winner}")


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-v', '--verbose', action='count', default=1,
help='increase log verbosity')
    parser.add_argument('-d',
                        '--debug',
                        action='store_const',
                        dest='verbose',
                        const=2,
                        help='log debug messages (same as -vv)')
    args = parser.parse_args()

    if args.verbose == 0:
        logging.getLogger().setLevel(level=logging.WARNING)
```

```
    elif args.verbose == 1:
        logging.getLogger().setLevel(level=logging.INFO)
    elif args.verbose == 2:
        logging.getLogger().setLevel(level=logging.DEBUG)

    main()
```

Opponent_Agents.py:

```
import extendedQuarto
import random
from quarto import Player


class DumbAgent(Player):
  '''
  Agent that purposefully makes bad decisions
  '''
  def __init__(self, quarto: extendedQuarto.ExtendedQuarto) -> None:
        super().__init__(quarto)


  def choose_piece(self) -> int:
    '''
    If possible, chooses a piece that the opponent can win with,
    otherwise chooses a random one
    '''
    board = self.get_game()

    unchosen_pieces = board.get_unchosen_pieces() #gets pieces that are left to choose


    if (len(unchosen_pieces) > 0):

      for piece in unchosen_pieces:
        win = board.check_if_possible_to_win(piece) #checks if it's possible to win with said piece

        if (win == True):
          return piece

        else: #chosenPieces
          p = random.randint(0, len(unchosen_pieces) -1) # between 0 and 15
          return unchosen_pieces[p]

    else:
      return -1 # In case the board is full


  def place_piece(self) -> tuple[int, int]:
    '''
    Place the piece in a random place
    '''
    return random.randint(0, 3), random.randint(0, 3)
```

rl.py:

```
import logging
import random
from quarto import Player
import extendedQuarto
import numpy as np
import pickle
from testQuarto import TestQuarto


class QTableKey(object):
    '''
    Class to use as Key for the q-table
    '''
    def __init__(self, board, selected_piece):
        self.board = board
        self.selected_piece = selected_piece

    def __hash__(self):
        '''
        Hash defined to be possible using it as a key in a dictionary
        '''
        return hash((hash(self.board.tostring()), self.selected_piece))

    def __eq__(self, other):
        '''
        Eq defined to be possible using it as a key in a dictionary
        '''
        return ((self.board & other.board).any()) and (self.selected_piece ==
other.selected_piece)

    def get_board_str(self):
        '''
        Gets board as string
        '''
        board_string = "["
        for y, row in enumerate(self.board):
            board_string += "["
            for x, index_at_place in enumerate(row):
                board_string += str(index_at_place)
                board_string += " "
            board_string += "]"
        board_string += "]"
        return board_string

    def get_selected_piece_str(self):
        '''
        Gets selected piece as string
        '''
        return str(self.selected_piece)
```

```python
class RLPlayer(Player):
    '''
    Reinforcement Learning Agent
    '''
    REWARD = 1 #Reward value for winning the game
    PENALTY = -1 #Penalty value for losing the game
    DRAW_REWARD = 0.5 #Reward value for drawing the game
    previous_state = None
    previous_move = None


    def __init__(self, quarto: extendedQuarto.ExtendedQuarto, learning_rate:
float, discount_rate: float, exploration_rate: float) -> None:
        super().__init__(quarto)
        q = {}  # {(QTableKey(board, selected piece), move) -> value}
        self.q = q
        self.learning_rate = learning_rate
        self.discount_rate = discount_rate
        self.exploration_rate = exploration_rate
        self.place_chosen = None # x,y
        self.chosen_piece = None # id


    def choose_piece(self) -> int:
        '''
        Function that returns the chosen piece
        '''
        if (self.chosen_piece == None): # q-learner is starting
            self.chosen_piece = random.randint(0, 15)

        return self.chosen_piece

    def place_piece(self) -> tuple[int, int]:
        '''
        Function that returns the place to put the piece
        '''
        x,y = self.place_chosen[0], self.place_chosen[1]
        return x,y

    def clear_previous_vars(self) -> None:
        '''
        Clears the variables to start a different game
        '''
        self.previous_state = None
        self.previous_move = None
        self.place_chosen = None # x,y
        self.chosen_piece = None # id

    def get_q_length(self) -> int:
        '''
        Gets size of the q-table
        '''
        return len(self.q)


    def generate_possible_moves(self) -> list:
```

```
        """
        Return a list of possible moves, [(x,y,id),(x,y,id),...,(x,y,id)]
        """

        current_state = self.get_game()
        board = current_state.get_board_status()
        selected_piece = current_state.get_selected_piece()

        empty_places = [] # list of empty places which are (x, y)
        not_selected_pieces = list(range(16)) #generates a list with values 0
to 15

        if (selected_piece != -1): # why are we doing this.
            not_selected_pieces.remove(selected_piece)

        for y, row in enumerate(board):
            for x, index_at_place in enumerate(row): # for all places in row,
index_on_place = -1 if no piece on place
                                                     #otherwise the index of
the piece
                if (index_at_place == -1):
                    empty_places.append((x,y)) #adds the empty place to the
list

                else: # if there is a piece at the place, remove it from
not_selected_pieces
                    not_selected_pieces.remove(index_at_place)

        possible_moves = []

        # add all possible moves
        for empty_place in empty_places:
            if len(not_selected_pieces) > 0:
                for piece in not_selected_pieces:
                    possible_moves.append((empty_place[0], empty_place[1],
piece))
            else:
                possible_moves.append((empty_place[0], empty_place[1], -1))
#when there is only one piece left

        return possible_moves


    def add_new_state_move(self) -> None:
        '''
        Adds new state, move combinations to the q-learner table
        '''
        possible_moves = self.generate_possible_moves()
        current_state = self.get_game() # type -> ExtendedQuarto
        board = current_state.get_board_status() # the list with the board
        selected_piece = current_state.get_selected_piece()


        for move in possible_moves: # adds the combination state, move to the
q

            current_key = QTableKey(board, selected_piece) #creates key
```

```
            if (current_key, move) not in self.q:
                self.q[(current_key, move)] = np.random.uniform(
                    0.0, 0.01)  # attribute a small random value


    def policy(self) -> tuple:
        '''
        Gets the move to apply
        '''
        possible_moves = self.generate_possible_moves()
        current_state = self.get_game() # type -> ExtendedQuarto
        board = current_state.get_board_status() # the list with the board
        selected_piece = current_state.get_selected_piece()

        if np.random.random() > self.exploration_rate: # Exploitation

            q_val_list = [self.q[(QTableKey(board, selected_piece), move)]
                          for move in possible_moves] # list of the values of
state and action

            max_val_index = np.argmax(q_val_list)    # returns the index of
the max element of the array

            return possible_moves[max_val_index] # returns the move with the
biggest q_value

        else:  # Exploration - returns a random possible move
            return random.sample(possible_moves, 1)[0]


    def set_move(self, move) -> None:
        '''
        Sets the move for chosen and place piece
        '''
        if(move != None):
            self.chosen_piece = move[2]
            self.place_chosen = move[0], move[1]

    def update_when_draw(self) -> None:
        '''
        Updates the q-table when the game draws
        '''
        q_value = self.q[(self.previous_state, self.previous_move)]
        #self.q[(self.previous_state, self.previous_move)] += \
                    #self.learning_rate * (self.DRAW_REWARD -
                                          #self.q[(self.previous_state,
self.previous_move)])
        self.q[(self.previous_state, self.previous_move)] += \
                    self.learning_rate * (self.DRAW_REWARD +
(self.discount_rate * q_value) -
                                        self.q[(self.previous_state,
self.previous_move)])

        self.clear_previous_vars()

    def update_when_lost(self)->None:
```

```
        '''
        Updates the q-table when the agent loses
        '''
        self.q[(self.previous_state, self.previous_move)] += \
            self.learning_rate * \
            (self.PENALTY -
            self.q[(self.previous_state, self.previous_move)])

        self.clear_previous_vars()


    def update_q(self) -> tuple:
        """
        Updated the q-table and returns the chosen piece and the coordinates
for the piece that
        should be placed in a tuple (chosen_piece: int, x: int, y: int)
        """
        current_move = None
        current_state = self.get_game()
        board = current_state.get_board_status() # the list with the board
        selected_piece = current_state.get_selected_piece() # gets the
selected piece of the board


        self.add_new_state_move()  # adds the new state, moves

        current_move = self.policy()  # gets the move that we want to use

        if self.previous_move is not None:  # if it is not the first move

            game = self.get_game()
            b = game.get_board_status()
            next_state = TestQuarto(b)


            next_state.select(game.get_selected_piece()) # set the selected
piece in the copied board to the same one in the original one

            next_state.place(current_move[0], current_move[1]) #apply move

            reward = 0

            # check winner or draw -> change reward.
            if (next_state.check_finished() and (next_state.check_winner() ==
-1)): # check if draw
                reward = self.DRAW_REWARD


            if (next_state.check_winner() >= 0): # check if winner
                reward = self.REWARD


            possible_moves = self.generate_possible_moves()

            max_q = max([self.q[(QTableKey(board, selected_piece), move)]
                        for move in possible_moves]) # max qvalue from the
possible moves of the current_state
```

```
            self.q[(self.previous_state, self.previous_move)] += \
                    self.learning_rate * (reward + (self.discount_rate *
max_q) -
                                        self.q[(self.previous_state,
self.previous_move)])


        self.set_move(current_move)

        self.previous_state, self.previous_move = QTableKey(board,
selected_piece), current_move

        return current_move

    def save_q_table(self):
        '''
        Save q-table in a file
        '''
        with open('q_table_3.pickle', 'wb') as handle:
            pickle.dump(self.q, handle, protocol=pickle.HIGHEST_PROTOCOL)


        f = open("doc_path.txt", "w")
        state = 0
        move = 1
        for key, value in self.q.items():
            elem = ""
            elem += key[state].get_board_str() + ";"
            elem += key[state].get_selected_piece_str() + ";"
            for i in range(3):
                elem += str(key[move][i]) + ";"
            elem += str(value) + "\n"
            f.write(elem)
        f.close()
```

testQuarto.py:

```
import quarto
import numpy as np


class TestQuarto(quarto.Quarto):
    '''
    Extended version of Quarto for move testing
    '''
    def __init__(self, board: np):
        super(TestQuarto, self).__init__()
        self._board = board

    def get_test_board_status(self):
        '''
        gets the board
```

```
        '''
        return self._board
```

train_Q_learner:

```
import extendedQuarto
from rl import *
from main import RandomPlayer
from quarto import Player
import logging
from trained_rl import TrainedRL
from opponent_agents import DumbAgent
from ga_agent import *
from tqdm import tqdm
from ga_dumb import UntrainedGAAgent
from ga_less_dumb import LeastDumbAgent

def train_q_learner(game: extendedQuarto.ExtendedQuarto, q_learner: RLPlayer,
external_agent: Player, q_learner_turn: int) -> int:
    '''
    Function to run a game between the Q-Learner and an opponent agent
    '''

    player = 0
    winner = -1

    if(q_learner_turn == 0):
        is_q_learner = True
        players = (q_learner, external_agent)
        q_learner_player = 0
    else:
        is_q_learner = False
        players = (external_agent, q_learner)
        q_learner_player = 1


    while not game.check_finished() and game.check_winner() == -1:

        piece_ok = False

        while not piece_ok:
            piece_ok = game.select(players[player].choose_piece())

        piece_ok = False

        if player == 0: #switch players
            player = 1
            game.set_current_player(1)
            if(q_learner_player == 1):
                is_q_learner = True
            else:
                is_q_learner = False
        else:
            player = 0
            game.set_current_player(0)
```

```python
            if(q_learner_player == 0):
                is_q_learner = True
            else:
                is_q_learner = False

        if (is_q_learner == True):
            q_learner.update_q()


        while not piece_ok:
            x, y = players[player].place_piece()
            piece_ok = game.place(x, y)


        winner = game.check_winner()


    if (winner != q_learner_turn): # q-learner didn't win
        if (winner == -1): # draw
            q_learner.update_when_draw()
        else:
            q_learner.update_when_lost()

    q_learner.clear_previous_vars()
    return winner


learning_rate = 0.8
discount_rate = 0.2
exploration_rate = 0.7

def q_learning_strategy(game: extendedQuarto.ExtendedQuarto, q_learner: RLPlayer, num_games: int):
    '''
    Strategy for making q-learner learn. Q-learner plays first against Dumb and then against Random
    Progression of difficulty
    '''
    # creat trained GA
    trained_evolved = EvolvedAgent(game) # performs better as second
    plus_pick_025 = (0.06369372715277863, 0.8847749515760825, 0.051531321271138734)
    plus_place_025 = (0.5117837231872915, 3.563965752635712e-05, 0.00032016772834300385, 0.07394996224198032, 0.38933362248027764, 0.024576884704581324)
    set_pick_and_place(trained_evolved, plus_pick_025, plus_place_025)

    results = []
    OPPONENTS = [DumbAgent(game), RandomPlayer(game), UntrainedGAAgent(game), LeastDumbAgent(game), EvolvedAgent(game), trained_evolved]

    for opponent in OPPONENTS:
        game.set_players((opponent, q_learner))
        won = 0
        lost = 0
        draw = 0
        games_run = 0
```

```
        q_learner.exploration_rate = exploration_rate
        print("HERE", q_learner.exploration_rate)

        for _ in tqdm(range(num_games)):
            game.reset()

            if (q_learner.exploration_rate >= 0.2): # change the exploration
rate to do more exploitation the more we play
                if (games_run == 300):
                    q_learner.exploration_rate -= 0.05 # migt end up in local
optima

            q_learner_turn = 1
            winner = train_q_learner(game, q_learner, opponent,
q_learner_turn)
            if (winner == q_learner_turn):
                results.append("won")

            elif (winner == -1):
                results.append("draw")

            else:
                results.append("lost")

            games_run += 1

        num_games += 700
        print(num_games)

        for result in results:
            if result == "draw":
                draw += 1
            elif result == "lost":
                lost += 1
            else:
                won += 1

        #prints results
        print("Won: " + str(won))
        print("Lost: " + str(lost))
        print("Draw: " + str(draw))

        won = 0
        lost = 0
        draw = 0
        results = []


    return won, lost, draw #returns results


def run():
    '''
    Runs everything
    '''
    logging.getLogger().setLevel(level=logging.INFO)
```

```
    game_train = extendedQuarto.ExtendedQuarto()
    q_learner = RLPlayer(game_train, learning_rate, discount_rate,
exploration_rate)

    q_learning_strategy(game_train, q_learner, 3000)

    q_learner.save_q_table() #saves table after training q-learner
    # q_table_1: num_games = 1000, games added per opponent = 500,
exploration rate = decreases by 0,05 every 100th game
    # q_table_2: num_games = 2000, games added per opponent = 700,
exploration rate = decreases by 0,05 every 200th game
    # q_table_3: num_games = 3000, games added per opponent = 700,
exploration rate = decreases by 0,05 every 300th game


run()
```

trained_rl:

```
from rl import QTableKey
from quarto import Quarto
from quarto import Player
import numpy as np
import pickle
import random

class TrainedRL(Player):
  '''
  Class with the trained RL
  '''
  def __init__(self, quarto: Quarto) -> None:
    super().__init__(quarto)
    self.q = {}
    self.place_chosen = None
    self.chosen_piece = None

    self.build_q()  # when this agent is inited -> q-table is created


  def choose_piece(self) -> int:
      if (self.chosen_piece == None): # we are starting
        return random.randint(0, 15)
      else:
        return self.chosen_piece


  def place_piece(self) -> tuple[int, int]:
      possible_moves = self.generate_possible_moves()
      current_state = self.get_game() # type -> Extended Quarto
      board = current_state.get_board_status() # the list with the board
      selected_piece = current_state.get_selected_piece()

      state = QTableKey(board, selected_piece)  # list of the values of state
and action
```

```
        for move in possible_moves:  # list of the values of state and action

              current_key = QTableKey(board, selected_piece) # adds the
combination state, move to the q

              if (current_key, move) not in self.q:
                self.q[(current_key, move)] = np.random.uniform(
                      0.0, 0.01)  # attribute a small random value


        q_val_list = [self.q[(state, move)]
              for move in possible_moves]

        max_val_index = np.argmax(q_val_list) # returns the index of the max
element of the array

        move = possible_moves[max_val_index] # returns the move with the
biggest q_value

        x = move[0]
        y = move[1]

        self.place_chosen = x, y
        self.chosen_piece = move[2]

        return x, y

    def build_q(self):
        '''
        Builds q-table from document
        '''
        file_to_read = open("q_table_3.pickle", "rb")

        self.q = pickle.load(file_to_read)


        f = open("doc_path.txt", "r")
        for row in f:
          elems = row.split(";")
          board_str = elems[0]
          BOARD_SIDE = 4

          board_str = board_str.replace("[", "")
          board_str = board_str.replace("]", "")
          numbers = board_str.split(" ")


          board = np.ones(
              shape=(BOARD_SIDE, BOARD_SIDE), dtype=int)

          i = 0
          for y in range(4):
              for x in range(4):
                  board[y,x] = int(numbers[i])
                  i += 1
```

```
        given_piece = int(elems[1])
        x = int(elems[2])
        y = int(elems[3])
        selected_piece = int(elems[4])
        q_value = float(elems[5])
        key = QTableKey(board, given_piece)
        move = (x,y,selected_piece)
        self.q[(key,move)] = q_value
      f.close()




  def generate_possible_moves(self) -> list:
      """
      Return a list of possible moves, [(x,y,id),(x,y,id),...,(x,y,id)]
      """
      current_state = self.get_game() # type -> ExtendedQuarto
      board = current_state.get_board_status() # the list with the board
      selected_piece = current_state.get_selected_piece()

      empty_places = []  # list of empty places which are (x, y)
      not_selected_pieces = list(range(16)) #generates a list fom 0 to 15
      if (selected_piece != -1):
          not_selected_pieces.remove(selected_piece)

      for y, row in enumerate(board):
          for x, index_at_place in enumerate(row): # for all places in row,
index_on_place = -1 if no piece on place
                                                   #otherwise the index of
the piece
              if (index_at_place == -1):
                  empty_places.append((x,y))
              else: # if there is a piece at the place, remove it from
not_selected_pieces
                  not_selected_pieces.remove(index_at_place)

      possible_moves = []

      for empty_place in empty_places: # add all possible moves
          if len(not_selected_pieces) > 0:
              for piece in not_selected_pieces:
                  possible_moves.append((empty_place[0], empty_place[1],
piece))
          else:
              possible_moves.append((empty_place[0], empty_place[1], -1))
#when there is only one piece left

      return possible_moves
```

SOURCE CODE FOR LAB 1:

List_generation_and_class_State.py:

```
# 'Taking inspiration from the puzzle problem and professor's code, defining class 'State'
import logging
from random import seed, choice
from typing import Callable
from gx_utils import *
import random


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


class State:
    def __init__(self, data: set):
        self._data = set(data)

    # 'hashing by Frozenset to avoid any mutation'

    def __hash__(self):
        return hash(frozenset(self._data))

    # since the length of lists will be important to find the solution

    def __len__(self):
        return len(self._data)

    def __eq__(self, other):
        return self._data == other._data

    def __lt__(self, other):
        return self._data < other._data

    def __or__(self, other):
```

```python
        return State(self._data | other._data)

    def __and__(self, other):
        return State(self._data & other._data)

    def __sub__(self, other):
        return State(self._data - other._data)

    def __str__(self):
        return str(self._data)

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

    def data(self):
        return self._data

    def copy_data(self):
        return self._data.copy()


# 'Writing Search algorithm for subsequent searches to find the final solution'

def search(
        initial_state: State,
        goal_test: Callable,
        parent_state: dict,
        state_cost: dict,
        priority_function: Callable,
        unit_cost: Callable,
):
    frontier = PriorityQueue()
    parent_state.clear()
    state_cost.clear()

    state = initial_state
    parent_state[state] = None
    state_cost[state] = 0

    while state is not None and not goal_test(state):
```

```
        for a in possible_actions():
            new_state = result(state, a)
            cost = unit_cost(a)
            if new_state not in state_cost and new_state not in frontier:
                parent_state[new_state] = state
                state_cost[new_state] = state_cost[state] + cost
                frontier.push(new_state, p=priority_function(new_state))
                logging.debug(f"Added new node to frontier (cost={state_cost[new_state]})")
            elif new_state in frontier and state_cost[new_state] > state_cost[state] + cost:
                old_cost = state_cost[new_state]
                parent_state[new_state] = state
                state_cost[new_state] = state_cost[state] + cost
                logging.debug(f"Updated node cost in frontier: {old_cost} ->
{state_cost[new_state]}")
        if frontier:
            state = frontier.pop()
        else:
            state = None

    path = list()
    s = state
    while s:
        path.append(s.copy_data())
        s = parent_state[s]

    logging.info(f"Found a solution in {len(path):,} steps; The number of visited states:
{len(state_cost):,} ")
    return list(reversed(path))


def goal_test(state):
    return state == GOAL


def possible_actions():
    return (State(x) for x in P)


def result(state, action):
    return state | action
```

```
# 'Breadth First'
logging.getLogger().setLevel(logging.INFO)
for N in [5, 10, 20, 100, 500, 1000]:
    parent_state = dict()
    state_cost = dict()
    GOAL = State(set(range(N)))
    P = problem(N, seed=42)


    def h(state):
        return len(state)


    INITIAL_STATE = State(set())
    logging.info(f'N = {N}')
    final = search(
        INITIAL_STATE,
        goal_test=goal_test,
        parent_state=parent_state,
        state_cost=state_cost,
        priority_function=lambda s: h(s),
        unit_cost=lambda a: 1,
    )
    logging.debug(final)

# 'DEPTH FIRST
logging.getLogger().setLevel(logging.INFO)
for N in [5, 10, 20, 100, 500, 1000]:
    parent_state = dict()
    state_cost = dict()
    GOAL = State(set(range(N)))
    P = problem(N, seed=42)




    INITIAL_STATE = State(set())
    logging.info(f'N = {N}')
    final = search(
```

```
        INITIAL_STATE,
        goal_test=goal_test,
        parent_state=parent_state,
        state_cost=state_cost,
        priority_function=lambda s: -len(state_cost),
        unit_cost=lambda a:1
    )
    logging.debug(final)

import heapq
from collections import Counter


class PriorityQueue:
    """A basic Priority Queue with simple performance optimizations"""

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

    def __bool__(self):
        return bool(self._data_set)

    def __contains__(self, item):
        return item in self._data_set

    def __len__(self):
        return len(self._data_set)

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

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

```
class Multiset:
    """Multiset"""

    def __init__(self, init=None):
        self._data = Counter()
        if init:
            for item in init:
                self.add(item)

    def __contains__(self, item):
        return item in self._data and self._data[item] > 0

    def __getitem__(self, item):
        return self.count(item)

    def __iter__(self):
        return (i for i in sorted(self._data.keys()) for _ in range(self._data[i]))

    def __len__(self):
        return sum(self._data.values())

    def __copy__(self):
        t = Multiset()
        t._data = self._data.copy()
        return t

    def __str__(self):
        return f"M{{{', '.join(repr(i) for i in self)}}}"

    def __repr__(self):
        return str(self)

    def __or__(self, other: "Multiset"):
        tmp = Multiset()
        for i in set(self._data.keys()) | set(other._data.keys()):
            tmp.add(i, cnt=max(self[i], other[i]))
        return tmp
```

```
def __and__(self, other: "Multiset"):
    return self.intersection(other)

def __add__(self, other: "Multiset"):
    return self.union(other)

def __sub__(self, other: "Multiset"):
    tmp = Multiset(self)
    for i, n in other._data.items():
        tmp.remove(i, cnt=n)
    return tmp

def __eq__(self, other: "Multiset"):
    return list(self) == list(other)

def __le__(self, other: "Multiset"):
    for i, n in self._data.items():
        if other.count(i) < n:
            return False
    return True

def __lt__(self, other: "Multiset"):
    return self <= other and not self == other

def __ge__(self, other: "Multiset"):
    return other <= self

def __gt__(self, other: "Multiset"):
    return other < self

def add(self, item, *, cnt=1):
    assert cnt >= 0, "Can't add a negative number of elements"
    if cnt > 0:
        self._data[item] += cnt

def remove(self, item, *, cnt=1):
    assert item in self, f"Item not in collection"
    self._data[item] -= cnt
    if self._data[item] <= 0:
        del self._data[item]
```

```
def count(self, item):
    return self._data[item] if item in self._data else 0

def union(self, other: "Multiset"):
    t = Multiset(self)
    for i in other._data.keys():
        t.add(i, cnt=other[i])
    return t

def intersection(self, other: "Multiset"):
    t = Multiset()
    for i in self._data.keys():
        t.add(i, cnt=min(self[i], other[i]))
    return t
```