Computational Intelligence - Exam Report

> **Luigi Federico, s295740**
>
> - Exam Project Repository (private): https://github.com/LuigiFederico/Computational_Intelligence_Exam
> - Course Repository (public): https://github.com/LuigiFederico/Computational-Intelligence

---

1 - Introduction

The agent is build using an heuristic strategy based on pareto fronts idea and using MinMax with alpha-beta pruning and both horizontal and deep pruning.

The method section illustrates in detail how the agent works and all the strategies that I developed. Next, the Experiment section contains a tuning experiment and the results against the Random Agent. The code is at the end of this report, followed by all the laboratory reports and reviews.

2 - Method

The agent uses two different approaches depending on which mode it is in. There could be two different modes:

- Safe Mode
- Threat Mode

The agent is in Threat Mode if there is at least one **threat piece**. A threat piece is a piece that, if placed in a line with 3 pieces, will result in a quarto. If there are no threat pieces, the agent is in Safe Mode.

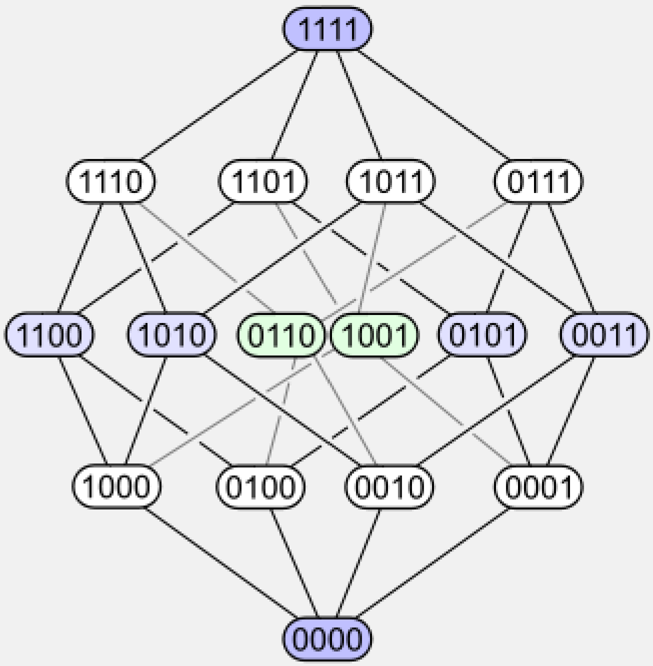
The agent will act differently according to the mode it is in:

- When it is in safe mode it will use an **euristic strategy** that tries to map the board state to a score board, based on the frontiers idea (see 2.1 - Frontiers and Score Board). This stands for both choosing and placing a piece.
- When it is in threat mode it will run a **MinMax algorithm** that uses alpha-beta pruning and both deep and horizontal pruning with a greedy sorting based on the Score Board to speed up the prunings (see 2.2 - MinMax).

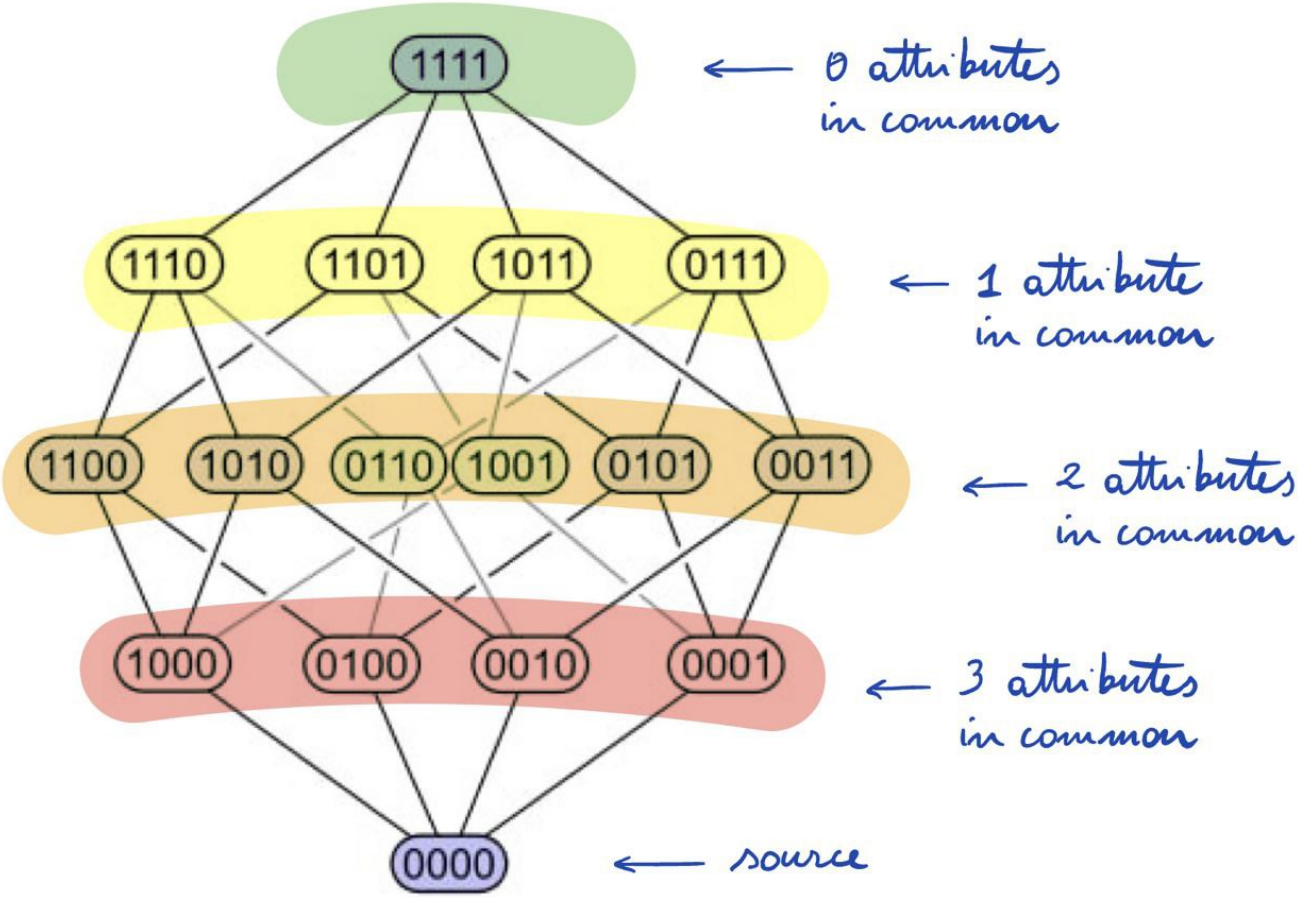
2.1 - Frontiers and Score Board

The idea of piece frontiers comes after reading the reference [1] "Quarto, Part 1 (Theory)" by Steven Morse in which he tries to analyes the math behind quarto. Here is the extract that inspired me:

> [...]
> << We'd like to relate the space to some object, and then use a well known dihedral group to represent the symmetries. (Instead of using a clunky flat matrix.) A good analogy is to use a 4-cube, or tesseract, and think of the binary numbers as the coordinates of the vertices of the cube in 4-space. Then if you choose a vertex *v*, *A_v* are the adjacent vertices, *B_v* are the vertices 2 steps away, *C_v* are

> the vertices 3 steps away, and *D_v* is the vertex on the opposite side of the 4-cube. Thus, *H* seems to be the dihedral group of a 4-cube.
>
> 
>
>
> Like all things, in hindsight, this is quite obvious. We have a 16-element set, and we are trying to identify symmetries. Of course this should correspond to the 16-element set called the coordinates of a 4-cube in Euclidean space! Of course! >>

The idea of the frontiers comes with this hypercube configuration. We can distinguish 4 frontiers, ordered by the number of attributes shared with the source piece. The higher the number of shared attributes, the higher the advantage to place the frontier piece in the same line of the source. Here is a better rapresentation of the frontiers:



If we place the piece with 0 common attributes in the same line of the source piece, we "disable" the line becouse no matter what piece the agent places in that line, it will never give a quarto. If we want to place a piece safely we could choose a disabled line.

For each frontier it doesn't matter which piece you choose, because it will share exactly that number of attributes with the source. This configuration is a sort of **Pareto Front** system.

We can compute the frontiers of more than one piece as source. The sources can be at most three, becouse after that a line is disabled and it doesn't make sense to think about frontiers since you can't place any other piece. There are exacly 696 possible sources (16 with one piece, 120 with two pieces and 560 with three pieces) so they can be easly computed and mantained by the agent during the games.

The code I used to compute statically the frontiers is the following:

```
@staticmethod
def get_frontiers():
    """
        Generates all the frontiers for each possible combination of 1, 2 and 3 elements.

        Returns a dictionary frontiers with:
        - key = tuple containing the source pieces of the respetive frontiers. Could have len = 1, 2 or 3.
        - value: the frontier corresponding to the source pieces as a dictionary with:
```

```
            - key: number of attributes in common with the pieces inside the level
of the frontier. key = 0, 1, 2 or 3.
            - value: pieces that share key elements in common with the source
pieces.
    """

    def get_frontier(p: int):
        """ Frontiers with 1 piece """
        assert p >= 0 and p <= 15

        x = dec_to_bin[p]
        frontier = {
            0: [],
            1: [],
            2: [],
            3: []
        }

        for num, binary in dec_to_bin.items():
            if num == p:
                continue
            cnt = 0
            for i, j in zip(x, binary):
                if i == j:
                    cnt += 1
            frontier[cnt].append(num)

        return frontier

    def get_frontier_2d(a: int, b:int):
        """ Frontiers with 2 pieces """
        assert a >= 0 and a <= 15
        assert b >= 0 and b <= 15
        assert a != b

        x = dec_to_bin[a]
        y = dec_to_bin[b]
        frontier = {
            0: [],
            1: [],
            2: [],
            3: []
        }

        for num, binary in dec_to_bin.items():
            if num == a or num == b:
                continue
            cnt = 0
            for x_i, y_i, j in zip(x, y, binary):
                if x_i == j and y_i == j:
                    cnt += 1
            frontier[cnt].append(num)

        return frontier
```

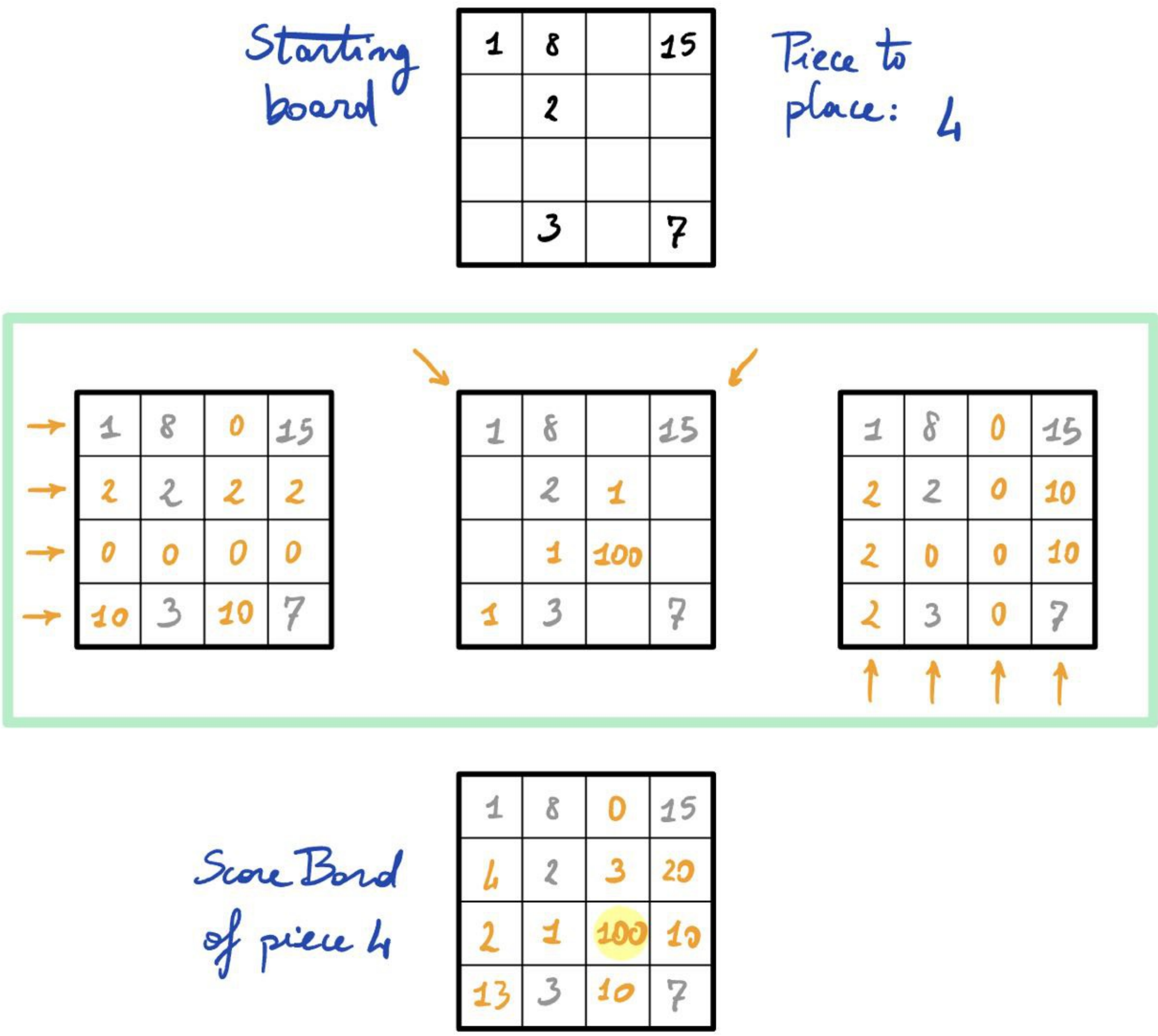
```python
    def get_frontier_3d(a: int, b:int, c: int):
        """ Frontiers with 3 pieces """
        assert a >= 0 and a <= 15
        assert b >= 0 and b <= 15
        assert c >= 0 and c <= 15
        assert a != b and a != c and b != c

        x = dec_to_bin[a]
        y = dec_to_bin[b]
        z = dec_to_bin[c]
        frontier = {
            0: [],
            1: [],
            2: [],
            3: []
        }

        for num, binary in dec_to_bin.items():
            if num == a or num == b or num == c:
                continue
            cnt = 0
            for x_i, y_i, z_i, j in zip(x, y, z, binary):
                if x_i == j and y_i == j and z_i == j:
                    cnt += 1
            frontier[cnt].append(num)

        return frontier

    frontier = {}

    # one piece -> 16
    for p in range(16):
        frontier[(p, )] = get_frontier(p)

    # two pieces -> 120 combinations
    for i in range(16):
        for j in range(i+1, 16):
            frontier[(i, j)] = get_frontier_2d(i, j)

    # three pieces -> 560 combinations
    for i in range(16):
        for j in range(i+1, 16):
            for k in range(j+1, 16):
                frontier[(i, j, k)] = get_frontier_3d(i, j, k)

    return frontier
```

Using frontiers it's easy and fast to distinguish the threat pieces from the safe ones. A threat piece would share at least one attribute with a source of three pieces. If a line has three pieces and there is at least one threat piece, than the agent enters in threat mode.

The agent uses the following method to distinguish the pieces:

```
def distinguish_pieces(self, board):
    """ Returns a list of safe pieces and a list of threat pieces """

    threat = set()

    # Rows
    for row in range(board.shape[0]):
        k = tuple(sorted([p for p in board[row, :] if p != -1])) # frontier source
(key)
        if len(k) == 3: # Threat line
            k_threat = set(self.frontiers[k][1]) | set(self.frontiers[k][2])  #
pieces with attributes in common with all of 3
            threat |= k_threat

    # Cols
    for col in range(board.shape[1]):
        k = tuple(sorted([p for p in board[:, col] if p != -1]))
        if len(k) == 3: # Threat line
            k_threat = set(self.frontiers[k][1]) | set(self.frontiers[k][2])  #
pieces with attributes in common with all of 3
            threat |= k_threat

    # Diag 1
    k = tuple(sorted([p for p in board.diagonal() if p != -1]))
    if len(k) == 3: # Threat line
        k_threat = set(self.frontiers[k][1]) | set(self.frontiers[k][2])  # pieces
with attributes in common with all of 3
        threat |= k_threat

    # Diag 2
    k = tuple(sorted([p for p in np.fliplr(board).diagonal() if p != -1]))
    if len(k) == 3: # Threat line
        k_threat = set(self.frontiers[k][1]) | set(self.frontiers[k][2])  # pieces
with attributes in common with all of 3
        threat |= k_threat


    # Clean up
    safe = set(range(16))
    placed = set([int(p) for p in np.nditer(board) if p != -1])
    threat -= placed
    safe -= placed
    safe -= threat

    return list(safe), list(threat)
```

Using the frontiers it is possible to map how much "dangerous" is to place a piece on a specific position of the board. When we have to place a piece, we can compute for each line how many attributes would be shared with the placed pieces on that line, i.e. on which frontier the piece to place is, using the placed pieces as source, and then we multiply that number for a scaling factor. This scaling factor is equal to 100 if the source is compoused of 3 pieces, 10 if the source has 2 pieces and 1 if the source is just one piece. This scaling factor highlights the board positions where is more likely to find a quarto after the placing. If we have a number in the hundreds, we have a winning condition: place the piece there and you will have a quarto! If the number is in the tens, than the higher it is the more probable it is to create a quarto condition on multiple lines. At the end, the scores computer for each line are summed togheter on the board. Here is a quick illustration to visualize the procedure:



The method used by the agent to compute the score board is the following:

```
def __board_intersection_scores(self, piece: int):
    """
        Computes the mapping of the board with scores in it using the frontiers
    """

    def extract_source(board_slice) -> tuple:
        """ Extracts the a key for self.frontiers """
        k = tuple(sorted([p for p in board_slice if p != -1]))
```

```
        return k

    def compute_score(frontier) -> int:
        rank = -1                           # Frontier rank = number of common
attributes with piece
        for i in range(4):
            if piece in frontier[i]:
                rank = i
                break

        rescaling = 10**(len(k) - 1)
        score = rank * rescaling
        return score


    # Setup
    board = self.get_game().get_board_status()
    board_scores = np.zeros(shape=(Quarto.BOARD_SIDE, Quarto.BOARD_SIDE),
dtype=int)   # Initialized mapped board

    # Rows
    for row in range(board.shape[0]):
        k = extract_source(board[row, :])   # Extract the frontier's source
        if k == () or len(k) == 4:
            continue
        score = compute_score(self.frontiers[k])
        board_scores[row, :] += score

    # Cols
    for col in range(board.shape[1]):
        k = extract_source(board[:, col])
        if k == () or len(k) == 4:
            continue
        score = compute_score(self.frontiers[k])
        board_scores[:, col] += score

    # Diag 1
    k = extract_source(board.diagonal())
    if k != () and len(k) != 4:
        score = compute_score(self.frontiers[k])
        tmp = np.zeros((4, 4), dtype=int)   # Update the diagonal
        np.fill_diagonal(tmp, score)
        board_scores += tmp

    # Diag 2
    k = extract_source(np.fliplr(board).diagonal())
    if k != () and len(k) != 4:
        score = compute_score(self.frontiers[k])
        tmp = np.zeros((4, 4), dtype=int)
        np.fill_diagonal(np.fliplr(tmp), score)
        board_scores += tmp

    # Clean up board_scores
    mask = np.array(board == -1, dtype=int)     # 0 if there is a piece in the i,j
```

```
board coordinate, 1 otherwise
    board_scores = board_scores * mask

    return board_scores
```

I used this score board strategy as base line for the heuristic strategy that the agent uses at safe mode:

- When the agent has to place a piece it computes all the possible actions and sorts them by ascending score and takes the action with the lowest score. This should delay as much as possible the threat mode strategy, in order to have a board configuration that is not too heavy for the MinMax algorithm to explore. It is also possible to sort the actions in descending mode, trying to create as soon as possible a threat line. In the experiment section (3 - Experiments) we can see that it is better to delay the MinMax computation in order to have faster agent moves. The method used for the "safe mode place piece" is the following:

```
def __safe_mode_place_piece(self, piece: int, *, delay_minmax = True) ->
tuple[int, int]:

    # Setup
    board = self.get_game().get_board_status()
    possible_actions = self.__list_ply_score(board, piece)

    # Sort the action
    if delay_minmax:
        possible_actions = sorted(possible_actions, key=lambda a: a[1])    #
ascending score (MinMax delayed amap)
    else:
        possible_actions = sorted(possible_actions, key=lambda a: -a[1])   #
descending score (MinMax asap)

    # Check if the action is safe (if there are enough pieces to choose)
    idx = 0
    ply, score = possible_actions[idx]

    while (self.__check_place_ply(ply, piece) == False):
        idx += 1
        if idx < len(possible_actions):
            ply, score = possible_actions[idx]
        else:
            break

    return ply


def __list_ply_score(self, board, piece):
    """
        List all the possible positions where to place the piece with score as
tuples ((x,y), score)
    """
    possible_actions = []
```

```
    score_board = self.__board_intersection_scores(piece)

    for row in range(board.shape[0]):
        for col in range(board.shape[1]):
            if board[row, col] == -1:       # Possible action
                possible_actions.append( ((row, col), score_board[row, col]) )

    return possible_actions

def __check_place_ply(self, ply: tuple[int, int], piece: int, *, safe_threshold =
1):
    """
        Returns True if there are enough safe pieces to choose after placing
        the given piece on board using the ply coordinates.
    """
    board = self.get_game().get_board_status()

    assert board[ply[0], ply[1]] == -1  # The ply must be legal

    board[ply[0], ply[1]] = piece
    safe, _ = self.distinguish_pieces(board)

    # I want enough safe pieces to choose after having placed the given piece on
the board
    if len(safe) >= safe_threshold:
        return True
    else:
        return False
```

- When the agent has to choose a piece in safe mode, it computes for each piece the total score and chooses the one with the minimum. This should result on giving the piece that gives the less advantage possible to the opponent and that delays the threat mode configuration as much as possible. The method that computes this is the following:

```
def __safe_mode_choose_piece(self, safe_pool) -> int:
    """
        Retrieves the piece with the lowest total score computed using the
frontiers.
        This should be the piece that gives to the opponent the lowest advantage.
    """
    best_piece = -1
    best_score = 500

    for p in iter(safe_pool):   # Min search
        board_score = self.__board_intersection_scores(p)
        score = board_score.sum()
        if score < best_score:
            best_piece = p

    return best_piece
```

2.2 - MinMax

When the agent is in Threat Mode it runs the MinMax algorithm to find the best ply possible. In order to have reasonable timings, I adopted the alpha-beta pruning and both deep and horizontal pruning.

Deep pruning follows a static approach: once it reaches `MINMAX_MAX_DEPTH`, it will stop expanding the game tree. After some tuning I found that MINMAX_MAX_DEPTH = 2 was the best compromise between exploration and time efficiency. Note that the reached depth is actually 3, becouse the algorithm starts the search using a counter that starts form 0.

Horizontal pruning uses a greedy heuristics hoping to help the alpha-beta pruning:

- When it has to place a piece, it sorts all the possible actions on descending order, exploring first the plys that could lead faster to a quarto. The horizontal pruning consists on exploring only the plys that has score greater than 0, becouse the cutted plys could need too much exploration and it could not be worth to explore them.
- When it has to choose a piece, it computes the total score for each safe piece and sort them in descending order. It explores first the pieces that could lead faster to a quarto. This is not an horizontal pruning but an heuristic that tries to help the alpha-beta pruning by finding as soon as possible a winning condition.

The used methods are the following:

```
def min_max_place_piece(self, board_now, piece, player, cnt=0):
    """
        MinMax algorithm with alpha-beta pruning, deep pruning, horizontal pruning
and greedy sorting for placing a piece.

        The greedy sorting tryes to speed up MinMax by sorting the possible
actions by
        the corresponding score computed with self.__list_ply_score(•).
        IDEA: Expand first the actions that could lead faster to a winning/drow
condition,
        hoping to favor the alpha-beta pruning.

        Args:
        - board -> current state of the board (numpy.ndarray)
        - piece -> piece to place (int)
        - player -> 0: my agent, 1: opponent

        Returns:
        - ply = coordinates as tuple es. (x, y)
        - piece = best piece to give associated with the best ply
        - score -> WIN = 5, LOSE = -5, DROW = 1
    """

    possible_actions = self.__list_ply_score(board_now, piece)          #
Descending score:
    possible_actions = sorted(possible_actions, key=lambda a: a[1])     # Greedy
```

```
sorting to speed up alpha-beta pruning

    if cnt > self.MINMAX_MAX_DEPTH:
        return possible_actions[0], -1, 0

    possible_actions = [a for a in possible_actions if a[1] > 0]        #
Horizontal pruning
    ply_drow = (-1, -1)
    piece_drow = -1
    best_score = 0

    for ply, _ in iter(possible_actions):
        if board_now[ply[0], ply[1]] != -1:     # Check that the ply is legal
            continue
        board = deepcopy(board_now)             # Ply
        board[ply[0], ply[1]] = piece

        if self.check_finished(board):                # Termination
            if player == 0:
                return ply, -1, WIN
            else:
                return ply, -1, LOSE

        piece, score = self.min_max_choose_piece(board, player, cnt)  # Recursion


        if score == WIN and player == 0:        # Alpha-Beta Pruning
            return ply, piece, WIN
        if score == LOSE and player == 1:
            return ply, piece, LOSE
        # If drow or else, keep looking for a better ply
        ply_drow = ply
        piece_drow = piece
        if score == DROW:
            best_score = DROW

    # No winning condition -> drow
    return ply_drow, piece_drow, best_score


def check_finished(self, board):
    """ Returns True if the board is a winning condition, False otherwise """

    def __check(k):
        for attribute in range(4):
            sum = 0
            for idx in range(4):
                sum += dec_to_bin[ k[idx] ][attribute]
            if sum == 4 or sum == 0:    # Winning condition
                return True

    # Rows
    for row in range(board.shape[0]):
        k = tuple(sorted([p for p in board[row, :] if p != -1]))
```

```
        if len(k) == 4 and __check(k):
            return True

    # Cols
    for col in range(board.shape[1]):
        k = tuple(sorted([p for p in board[:, col] if p != -1]))
        if len(k) == 4 and __check(k):
            return True

    # Diag 1
    k = tuple(sorted([p for p in board.diagonal() if p != -1]))
    if len(k) == 4 and __check(k):
        return True

    # Diag 2
    k = tuple(sorted([p for p in np.fliplr(board).diagonal() if p != -1]))
    if len(k) == 4 and __check(k):
        return True

    return False
```

```
def min_max_choose_piece(self, board, player, cnt=0):
    """
        MinMax algorithm for choosing a piece with alpha-beta pruning and greedy
sorting.

        The greedy sorting tryes to speed up MinMax by sorting the safe pieces
(the only one expanded)
        by decreasing score. The score is computed as sum of the scores inside the
board_score computed
        by self.__board_intersection_scores(p) for each piece p in the safe pool.
        IDEA: look for pieces that offer more aggressive plays hoping to find
faster a winning/drow condition
        in oreder to assist the alpha-beta pruning.

        Args:
        - board -> current state of the board, after having placed the previous
given piece (np.ndarray)
        - player -> 0: my agent, 1: opponent

        Returns:
        - piece = choosen piece
        - score -> WIN = 5, LOSE = -5, DROW = 1
    """

    safe, threat = self.distinguish_pieces(board)   # Threat and safe pieces pools

    # Terminations
    if len(safe) + len(threat) == 0:
        return -1, DROW
```

```
    if len(safe) == 0:
        if player == 0:
            return threat[0], LOSE
        else:
            return threat[0], WIN

    if cnt > self.MINMAX_MAX_DEPTH:
        return safe[0], 0

    # Greedy sorting to speed up MinMax pruning
    piece_score = []
    for p in iter(safe):
        board_score = self.__board_intersection_scores(p)
        piece_score.append((p, board_score.sum()))
    piece_score = sorted(piece_score, key=lambda x: -x[1])   # Descending order

    # Recursion
    best_piece = -1
    best_score = []
    for piece, _ in iter(piece_score):
        _, _, score = self.min_max_place_piece(board, piece, (player + 1) % 2,
cnt+1)

        if player == 0 and score == WIN:        # Alpha-Beta Pruning
            return piece, WIN
        if player == 1 and score == LOSE:
            return piece, LOSE
        # If drow or else, keep looking
        best_piece = piece
        if score == DROW:
            best_score = DROW

    # No winning condition -> drow
    return best_piece, best_score
```

2.3 - Place piece

As already said, if the agent is in threat mode, if it has to place a threat piece, he place it on the spot that creates a quarto, otherwise it runs the MinMax algorithm (`min_max_place_piece(•)`). The MinMax can retrieve also the best piece to give after the placing ply so it saves time avoiding to run again the MinMax aglorithm to choose the piece to give. If the agent is in safe mode, it runs the heuristic strategy.

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

    # Setup
    board = self.get_game().get_board_status()
    piece_to_place = self.get_game().get_selected_piece()
    safe, threat = self.distinguish_pieces(board)

    # Threat mode -> MinMax
```

```
    if len(threat) > 0:
        # If I have to place a threat piece, I win
        if piece_to_place in threat:
            score_board = self.__board_intersection_scores(piece_to_place)
            ply_ = np.unravel_index(np.argmax(score_board), score_board.shape)    #
Max score on the winning position
            ply = (ply_[1], ply_[0])
            return ply

        # MinMax
        ply_, piece, _ = self.min_max_place_piece(board, piece_to_place, player =
0)
        ply = (ply_[1], ply_[0])
        if piece != -1:
            self.piece_to_give = piece  # No need to run MinMax again to choose
the piece

    # Safe mode -> Heuristic strategy
    else:
        ply_ = self.__safe_mode_place_piece(piece_to_place, delay_minmax = True)
        ply = (ply_[1], ply_[0])
    return ply
```

2.4 - Choose piece

When the agent has to choose a piece, if it computed the best piece with the placing ply using MinMax, it gives that piece. Otherwise, if it is in threat mode it runs MinMax, if it is in safe mode it uses the heuristic strategy.

```
def choose_piece(self) -> int:
    # Best piece to give computed inside self.place_piece() with MinMax
    if self.piece_to_give != -1:
        p = self.piece_to_give
        self.piece_to_give = -1
        return p

    # Setup
    board = self.get_game().get_board_status()
    safe, threat = self.distinguish_pieces(board)

    # Threat mode -> MinMax
    if len(threat) > 0:
        piece, _ = self.min_max_choose_piece(board, 0)

    # Safe mode -> Heuristic strategy
    else:
        piece = self.__safe_mode_choose_piece(safe)
```

```
    return piece
```

---

3 - Experiments

I runned some tournaments in order to evaluate how well my agent performs against the RandomPlayer. The code of the tournament is the following:

```
from main import RandomPlayer
from quarto.players import MyPlayer
from quarto.objects import Quarto
from IPython.display import clear_output

def main(reverse = False):
    game = Quarto()
    if reverse:
        game.set_players((RandomPlayer(game), MyPlayer(game)))
    else:
        game.set_players((MyPlayer(game), RandomPlayer(game)))

    winner = game.run()
    clear_output(wait=False)
    print(f"main: Winner: player {winner}")
    return winner

N_MATCHES = 200

cnt = 0
for i in range(N_MATCHES):
    if i < N_MATCHES//2:
        w = main()
        if w == 0:
            cnt += 1
    else:
        w = main(True)
        if w == 1:
            cnt += 1

print(f'WINRATE = {cnt / N_MATCHES}')
```

I wanted to see if there was a difference between sorting all the possible actions (inside the function `__safe_mode_place_piece(•)`) in ascending or descending mode when the agent is in safe mode. I repropose the function:

```
 def __safe_mode_place_piece(self, piece: int, *, delay_minmax = True) ->
tuple[int, int]:
```

```
        # Setup
        board = self.get_game().get_board_status()
        possible_actions = self.__list_ply_score(board, piece)

        # Sort the action
        if delay_minmax:
            possible_actions = sorted(possible_actions, key=lambda a: a[1])    #
ascending score (MinMax delayed amap)
        else:
            possible_actions = sorted(possible_actions, key=lambda a: -a[1])   #
descending score (MinMax asap)

        # Check if the action is safe (if there are enough pieces to choose)
        idx = 0
        ply, score = possible_actions[idx]

        while (self.__check_place_ply(ply, piece) == False):
            idx += 1
            if idx < len(possible_actions):
                ply, score = possible_actions[idx]
            else:
                break

        return ply
```

Looking at the results (table below) it doesn't seem to have a relevant impact on the performance of the agent, maybe becouse the opponent is a Random Agent and, obviously, the game statistics (WinRate and ExecutionTime/#matches) are not really stable.

# matches	delay_minmax = True	WinRate	Execution time	ExecutionTime / # Matches
20	True	0.95	6m 45.8s	20.25s
20	True	0.9	5m 14.2s	15.7s
20	True	1.00	1m 39.8s	4.95s
20	False	0.95	5m 52.0s	17.6s
20	False	1.00	6m 59.4s	20.95s
20	False	0.9	11m 10.8s	33.5s
100	True	0.92	6m 21.2s	3.81s
100	True	0.91	3m 24.5s	2.04s
100	False	0.91	3m 24.5s	2.04s
100	False	0.96	12m 14.0s	7.34s
200	False	0.975	37m 37.5s	11.29s
200	Ture	0.965	13m 1.4s	3.91s

We can observe that the timings don't follow a pattern but they are quite random, as just said.
We can take the averages:

delay_minmax	Avg WinRate	Avg ExecutionTime/#matches
True	0.924	8.44s
False	0.949	15.45s

It seems to have better time performances when we use the greedy sorting to deplay the MinMax algorithm but to observe better statistics it would be better to play against a less random agent.

Overall, the winrate varies between 0.9 and 1.00. The performances against the Random Agent are promising but it would be better to test the agent against a stronger and different agents in order to have a more robust feedback.

We can conclude that the performances against the Random Agent are good, despite the execution times being relatively slow for a machine. This is due to the MinMax algorithm that, dispite the prunings and the euristics to help it to be faster, it can still be slow sometimes.

---

4 - References

1. Steven Morse, "Quarto, Part 1 (Theory)", 01/03/2017, [link]
2. WayBackMachine, "Quarto", 12/10/2004, [link]

---

5 - Code (players.py)

```
from copy import deepcopy
import numpy as np
from .objects import Player, Quarto

WIN = 5
LOSE = -5
DROW = 1

dec_to_bin = {
    0: [0, 0, 0, 0],
    1: [0, 0, 0, 1],
    2: [0, 0, 1, 0],
    3: [0, 0, 1, 1],
    4: [0, 1, 0, 0],
    5: [0, 1, 0, 1],
    6: [0, 1, 1, 0],
    7: [0, 1, 1, 1],
    8: [1, 0, 0, 0],
    9: [1, 0, 0, 1],
    10: [1, 0, 1, 0],
    11: [1, 0, 1, 1],
```

```
    12: [1, 1, 0, 0],
    13: [1, 1, 0, 1],
    14: [1, 1, 1, 0],
    15: [1, 1, 1, 1]
}

class MyPlayer(Player):

    MINMAX_MAX_DEPTH = 2

    def __init__(self, quarto: Quarto) -> None:
        super().__init__(quarto)
        self.frontiers = self.get_frontiers()
        self.piece_to_give = -1


    def choose_piece(self) -> int:
        # Best piece to give computed inside self.place_piece() with MinMax
        if self.piece_to_give != -1:
            p = self.piece_to_give
            self.piece_to_give = -1
            return p

        # Setup
        board = self.get_game().get_board_status()
        safe, threat = self.distinguish_pieces(board)

        # Threat mode -> MinMax
        if len(threat) > 0:
            piece, _ = self.min_max_choose_piece(board, 0)

        # Safe mode -> Heuristic strategy
        else:
            piece = self.__safe_mode_choose_piece(safe)

        return piece


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

        # Setup
        board = self.get_game().get_board_status()
        piece_to_place = self.get_game().get_selected_piece()
        safe, threat = self.distinguish_pieces(board)

        # Threat mode -> MinMax
        if len(threat) > 0:
            # If I have to place a threat piece, I win
            if piece_to_place in threat:
                score_board = self.__board_intersection_scores(piece_to_place)
                ply_ = np.unravel_index(np.argmax(score_board), score_board.shape)
# Max score on the winning position
                ply = (ply_[1], ply_[0])
                return ply
```

```
            # MinMax
            ply_, piece, _ = self.min_max_place_piece(board, piece_to_place,
player = 0)
            ply = (ply_[1], ply_[0])
            if piece != -1:
                self.piece_to_give = piece  # No need to run MinMax again to
choose the piece

        # Safe mode -> Heuristic strategy
        else:
            ply_ = self.__safe_mode_place_piece(piece_to_place, delay_minmax =
True)
            ply = (ply_[1], ply_[0])
        return ply

    @staticmethod
    def get_frontiers():
        """
            Generates all the frontiers for each possible combination of 1, 2 and
3 elements.

            Returns a dictionary frontiers with:
            - key = tuple containing the source pieces of the respetive frontiers.
Could have len = 1, 2 or 3.
            - value: the frontier corresponding to the source pieces as a
dictionary with:
                - key: number of attributes in common with the pieces inside the
level of the frontier. key = 0, 1, 2 or 3.
                - value: pieces that share key elements in common with the source
pieces.
        """

        def get_frontier(p: int):
            """ Frontiers with 1 piece """
            assert p >= 0 and p <= 15

            x = dec_to_bin[p]
            frontier = {
                0: [],
                1: [],
                2: [],
                3: []
            }

            for num, binary in dec_to_bin.items():
                if num == p:
                    continue
                cnt = 0
                for i, j in zip(x, binary):
                    if i == j:
                        cnt += 1
                frontier[cnt].append(num)
```

```
        return frontier

    def get_frontier_2d(a: int, b:int):
        """ Frontiers with 2 pieces """
        assert a >= 0 and a <= 15
        assert b >= 0 and b <= 15
        assert a != b

        x = dec_to_bin[a]
        y = dec_to_bin[b]
        frontier = {
            0: [],
            1: [],
            2: [],
            3: []
        }

        for num, binary in dec_to_bin.items():
            if num == a or num == b:
                continue
            cnt = 0
            for x_i, y_i, j in zip(x, y, binary):
                if x_i == j and y_i == j:
                    cnt += 1
            frontier[cnt].append(num)

        return frontier

    def get_frontier_3d(a: int, b:int, c: int):
        """ Frontiers with 3 pieces """
        assert a >= 0 and a <= 15
        assert b >= 0 and b <= 15
        assert c >= 0 and c <= 15
        assert a != b and a != c and b != c

        x = dec_to_bin[a]
        y = dec_to_bin[b]
        z = dec_to_bin[c]
        frontier = {
            0: [],
            1: [],
            2: [],
            3: []
        }

        for num, binary in dec_to_bin.items():
            if num == a or num == b or num == c:
                continue
            cnt = 0
            for x_i, y_i, z_i, j in zip(x, y, z, binary):
                if x_i == j and y_i == j and z_i == j:
                    cnt += 1
            frontier[cnt].append(num)
```

```
            return frontier

        frontier = {}

        # one piece -> 16
        for p in range(16):
            frontier[(p, )] = get_frontier(p)

        # two pieces -> 120 combinations
        for i in range(16):
            for j in range(i+1, 16):
                frontier[(i, j)] = get_frontier_2d(i, j)

        # three pieces -> 560 combinations
        for i in range(16):
            for j in range(i+1, 16):
                for k in range(j+1, 16):
                    frontier[(i, j, k)] = get_frontier_3d(i, j, k)

        return frontier


    def distinguish_pieces(self, board):
        """ Returns a list of safe pieces and a list of threat pieces """

        threat = set()

        # Rows
        for row in range(board.shape[0]):
            k = tuple(sorted([p for p in board[row, :] if p != -1])) # frontier
source (key)
            if len(k) == 3: # Threat line
                k_threat = set(self.frontiers[k][1]) | set(self.frontiers[k][2])
# pieces with attributes in common with all of 3
                threat |= k_threat

        # Cols
        for col in range(board.shape[1]):
            k = tuple(sorted([p for p in board[:, col] if p != -1]))
            if len(k) == 3: # Threat line
                k_threat = set(self.frontiers[k][1]) | set(self.frontiers[k][2])
# pieces with attributes in common with all of 3
                threat |= k_threat

        # Diag 1
        k = tuple(sorted([p for p in board.diagonal() if p != -1]))
        if len(k) == 3: # Threat line
            k_threat = set(self.frontiers[k][1]) | set(self.frontiers[k][2])  #
pieces with attributes in common with all of 3
            threat |= k_threat

        # Diag 2
        k = tuple(sorted([p for p in np.fliplr(board).diagonal() if p != -1]))
        if len(k) == 3: # Threat line
```

```
            k_threat = set(self.frontiers[k][1]) | set(self.frontiers[k][2])  #
pieces with attributes in common with all of 3
            threat |= k_threat


        # Clean up
        safe = set(range(16))
        placed = set([int(p) for p in np.nditer(board) if p != -1])
        threat -= placed
        safe -= placed
        safe -= threat

        return list(safe), list(threat)


    def __board_intersection_scores(self, piece: int):
        """
            Computes the mapping of the board with scores in it using the
frontiers
        """

        def extract_source(board_slice) -> tuple:
            """ Extracts the a key for self.frontiers """
            k = tuple(sorted([p for p in board_slice if p != -1]))
            return k

        def compute_score(frontier) -> int:
            rank = -1                           # Frontier rank = number of common
attributes with piece
            for i in range(4):
                if piece in frontier[i]:
                    rank = i
                    break

            rescaling = 10**(len(k) - 1)
            score = rank * rescaling
            return score


        # Setup
        board = self.get_game().get_board_status()
        board_scores = np.zeros(shape=(Quarto.BOARD_SIDE, Quarto.BOARD_SIDE),
dtype=int)   # Initialized mapped board

        # Rows
        for row in range(board.shape[0]):
            k = extract_source(board[row, :])   # Extract the frontier's source
            if k == () or len(k) == 4:
                continue
            score = compute_score(self.frontiers[k])
            board_scores[row, :] += score

        # Cols
        for col in range(board.shape[1]):
```

```
            k = extract_source(board[:, col])
            if k == () or len(k) == 4:
                continue
            score = compute_score(self.frontiers[k])
            board_scores[:, col] += score

        # Diag 1
        k = extract_source(board.diagonal())
        if k != () and len(k) != 4:
            score = compute_score(self.frontiers[k])
            tmp = np.zeros((4, 4), dtype=int)   # Update the diagonal
            np.fill_diagonal(tmp, score)
            board_scores += tmp

        # Diag 2
        k = extract_source(np.fliplr(board).diagonal())
        if k != () and len(k) != 4:
            score = compute_score(self.frontiers[k])
            tmp = np.zeros((4, 4), dtype=int)
            np.fill_diagonal(np.fliplr(tmp), score)
            board_scores += tmp

        # Clean up board_scores
        mask = np.array(board == -1, dtype=int)     # 0 if there is a piece in the
i,j board coordinate, 1 otherwise
        board_scores = board_scores * mask

        return board_scores


    def __safe_mode_choose_piece(self, safe_pool) -> int:
        """
            Retrieves the piece with the lowest total score computed using the
frontiers.
            This should be the piece that gives to the opponent the lowest
advantage.
        """
        best_piece = -1
        best_score = 500

        for p in iter(safe_pool):   # Min search
            board_score = self.__board_intersection_scores(p)
            score = board_score.sum()
            if score < best_score:
                best_piece = p

        return best_piece


    def __safe_mode_place_piece(self, piece: int, *, delay_minmax = True) ->
tuple[int, int]:

        # Setup
        board = self.get_game().get_board_status()
```

```
        possible_actions = self.__list_ply_score(board, piece)

        # Sort the action
        if delay_minmax:
            possible_actions = sorted(possible_actions, key=lambda a: a[1])    #
ascending score (MinMax delayed amap)
        else:
            possible_actions = sorted(possible_actions, key=lambda a: -a[1])   #
descending score (MinMax asap)

        # Check if the action is safe (if there are enough pieces to choose)
        idx = 0
        ply, score = possible_actions[idx]

        while (self.__check_place_ply(ply, piece) == False):
            idx += 1
            if idx < len(possible_actions):
                ply, score = possible_actions[idx]
            else:
                break

        return ply


    def __list_ply_score(self, board, piece):
        """
            List all the possible positions where to place the piece with score as
tuples ((x,y), score)
        """
        possible_actions = []
        score_board = self.__board_intersection_scores(piece)

        for row in range(board.shape[0]):
            for col in range(board.shape[1]):
                if board[row, col] == -1:       # Possible action
                    possible_actions.append( ((row, col), score_board[row, col]) )

        return possible_actions


    def __check_place_ply(self, ply: tuple[int, int], piece: int, *,
safe_threshold = 1):
        """
            Returns True if there are enough safe pieces to choose after placing
            the given piece on board using the ply coordinates.
        """
        board = self.get_game().get_board_status()

        assert board[ply[0], ply[1]] == -1  # The ply must be legal

        board[ply[0], ply[1]] = piece
        safe, _ = self.distinguish_pieces(board)

        # I want enough safe pieces to choose after having placed the given piece
```

```
on the board
        if len(safe) >= safe_threshold:
            return True
        else:
            return False


    def min_max_place_piece(self, board_now, piece, player, cnt=0):
        """
            MinMax algorithm with alpha-beta pruning, deep pruning, horizontal
pruning and greedy sorting for placing a piece.

            The greedy sorting tryes to speed up MinMax by sorting the possible
actions by
            the corresponding score computed with self.__list_ply_score(•).
            IDEA: Expand first the actions that could lead faster to a
winning/drow condition,
            hoping to favor the alpha-beta pruning.

            Args:
            - board -> current state of the board (numpy.ndarray)
            - piece -> piece to place (int)
            - player -> 0: my agent, 1: opponent

            Returns:
            - ply = coordinates as tuple es. (x, y)
            - piece = best piece to give associated with the best ply
            - score -> WIN = 5, LOSE = -5, DROW = 1
        """

        possible_actions = self.__list_ply_score(board_now, piece)          #
Descending score:
        possible_actions = sorted(possible_actions, key=lambda a: a[1])     #
Greedy sorting to speed up alpha-beta pruning

        if cnt > self.MINMAX_MAX_DEPTH:
            return possible_actions[0], -1, 0

        possible_actions = [a for a in possible_actions if a[1] > 0]        #
Horizontal pruning
        ply_drow = (-1, -1)
        piece_drow = -1
        best_score = 0

        for ply, _ in iter(possible_actions):
            if board_now[ply[0], ply[1]] != -1:     # Check that the ply is legal
                continue
            board = deepcopy(board_now)             # Ply
            board[ply[0], ply[1]] = piece

            if self.check_finished(board):              # Termination
                if player == 0:
                    return ply, -1, WIN
                else:
```

```
                    return ply, -1, LOSE

            piece, score = self.min_max_choose_piece(board, player, cnt)  #
Recursion


            if score == WIN and player == 0:        # Alpha-Beta Pruning
                return ply, piece, WIN
            if score == LOSE and player == 1:
                return ply, piece, LOSE
            # If drow or else, keep looking for a better ply
            ply_drow = ply
            piece_drow = piece
            if score == DROW:
                best_score = DROW

        # No winning condition -> drow
        return ply_drow, piece_drow, best_score


    def min_max_choose_piece(self, board, player, cnt=0):
        """
            MinMax algorithm for choosing a piece with alpha-beta pruning and
greedy sorting.

            The greedy sorting tryes to speed up MinMax by sorting the safe pieces
(the only one expanded)
            by decreasing score. The score is computed as sum of the scores inside
the board_score computed
            by self.__board_intersection_scores(p) for each piece p in the safe
pool.
            IDEA: look for pieces that offer more aggressive plays hoping to find
faster a winning/drow condition
            in oreder to assist the alpha-beta pruning.

            Args:
            - board -> current state of the board, after having placed the
previous given piece (np.ndarray)
            - player -> 0: my agent, 1: opponent

            Returns:
            - piece = choosen piece
            - score -> WIN = 5, LOSE = -5, DROW = 1
        """

        safe, threat = self.distinguish_pieces(board)   # Threat and safe pieces
pools

        # Terminations
        if len(safe) + len(threat) == 0:
            return -1, DROW

        if len(safe) == 0:
            if player == 0:
```

```
                return threat[0], LOSE
            else:
                return threat[0], WIN

        if cnt > self.MINMAX_MAX_DEPTH:
            return safe[0], 0

        # Greedy sorting to speed up MinMax pruning
        piece_score = []
        for p in iter(safe):
            board_score = self.__board_intersection_scores(p)
            piece_score.append((p, board_score.sum()))
        piece_score = sorted(piece_score, key=lambda x: -x[1])    # Descending
order

        # Recursion
        best_piece = -1
        best_score = []
        for piece, _ in iter(piece_score):
            _, _, score = self.min_max_place_piece(board, piece, (player + 1) % 2,
cnt+1)

            if player == 0 and score == WIN:        # Alpha-Beta Pruning
                return piece, WIN
            if player == 1 and score == LOSE:
                return piece, LOSE
            # If drow or else, keep looking
            best_piece = piece
            if score == DROW:
                best_score = DROW

        # No winning condition -> drow
        return best_piece, best_score


    def check_finished(self, board):
        """ Returns True if the board is a winning condition, False otherwise """

        def __check(k):
            for attribute in range(4):
                sum = 0
                for idx in range(4):
                    sum += dec_to_bin[ k[idx] ][attribute]
                if sum == 4 or sum == 0:     # Winning condition
                    return True

        # Rows
        for row in range(board.shape[0]):
            k = tuple(sorted([p for p in board[row, :] if p != -1]))
            if len(k) == 4 and __check(k):
                return True

        # Cols
        for col in range(board.shape[1]):
```

```
            k = tuple(sorted([p for p in board[:, col] if p != -1]))
            if len(k) == 4 and __check(k):
                return True

        # Diag 1
        k = tuple(sorted([p for p in board.diagonal() if p != -1]))
        if len(k) == 4 and __check(k):
            return True

        # Diag 2
        k = tuple(sorted([p for p in np.fliplr(board).diagonal() if p != -1]))
        if len(k) == 4 and __check(k):
            return True

        return False
```

Laboratories

Lab 1 - Set Covering

My code

Setup

```
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))
    ]
```

Solution Proposed

```
W_FACTOR = 2
INTW_FACTOR = 10

def cost(l, goal, covered):
  """ Compute the weighted number of digits left to find the goal state + the
weighted len of the list"""
  extended_covered = set(l) | covered   # Covered digits including the list l
  return len(l) * W_FACTOR + len(goal - extended_covered) * INTW_FACTOR


def find_solution(probl, N):
  """ Optimized greedy """

  def print_solution(solution, n_visited):
    w = 0
    for sol_l in solution:
      w += len(sol_l)

    print("We have found a solution!!")
    print(f"Weight = {w}")
    print(f"Number of visited nodes = {n_visited}")
    print(f"\nSolution: {solution}")

  n_visited = 0
  goal = set(range(N))
  covered = set()
  solution = list()
```

```
  frontier = sorted(probl, key=lambda l: cost(l, goal, covered))

  # Max number of iterations = N (worst case: I add ONE new digit to the covered
set every time)
  # Possible becouse we clear the redundant lists
  for i in range(N):
    # Goal reached?
    if goal == covered:
      print_solution(solution, n_visited)
      return

    # Look for another list
    if frontier:
      l = frontier.pop(0)
      n_visited += 1

      # If l is not a subset of covered, e.i. there is a new digit
      if not set(l) < covered:
        solution.append(l)
        covered |= set(l)

      # Clear the redundant lists (lists with only digits already covered)
      for x in frontier:
        if set(x) < covered:
          frontier.remove(x)

      frontier.sort(key=lambda l: cost(l, goal, covered))

    # No list available
    else:
      print("No solution has beed found")
      return

  if goal == covered:
    print_solution(solution, n_visited)
  else:
    print("No solution has beed found")
```

My REDME

> # Lab1: Set Covering
>
> > Author: Luigi Federico
> >
> > Computational Intelligence 2022/23
> > Prof: G. Squillero
>
> ---
>
> ## Task

Given a number N and some lists of integers $P = (L_0, L_1, L_2, ..., L_n)$, determine is possible $S = (L_{s_0}, L_{s_1}, L_{s_2}, ..., L_{s_n})$

such that each number between 0 and $N-1$ appears in at least one list

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

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

Solution proposed

To find a solution I did as follows:

1. I sorted the lists $L_0, L_1, L_2, ..., L_n$ using the cost function (explained below). This sorted lists are the frontier.
2. For a maximum of N iterations, the algorithm pops out the list with the lowest cost and adds it to the solution.
3. It clears from the frontier every list that is a strict subset of the covered set, becouse it can't contribute to build a solution.
4. The frontier is now resorted using the cost function.
5. If there will be at least a not redundant list inside the frontier, it will be considered to build the solution in the next iteration. If the frontier is empty after the next loop and we didn't find the solution I declare it.

> **Why at most N iterations?**
> The worst case in this approach occurs when we take at each step just one new digit into the covered set. If at some point we want to append a new list to the solution, we surely want to add the one that gives at least one new digit to the covered set. If we do more then N operations, it's garanteed that at least one of the lists is redundant, i.e. it contains only digits that was already added to the solution.

Cost function

```
def cost(l, goal, covered):
  extended_covered = set(l) | covered
  return len(l) * W_FACTOR + len(goal - extended_covered) * INTW_FACTOR
```

The idea behind this function is to consider:

- the length of the current list *l*
- the number of digits left to achive the solution if the list *l* was included in the *covered* set.

I wanted to give more importance to the amount of contribution that the given list *l* brings to the solution (in terms of number of new digits offered to the *covered* set) rather then the weight of *l*, i.e. the length of the list.
To do this, I scaled the weight and the number of new digits obtained adding *l* to the solution with **W_FACTOR**=2 (weight factor) and **INTW_FACTOR**=10 (internal weight factor), respectively.

> Note that if we use as scale factor W_FACTOR = INTW_FACTOR, the algorithm will favorite short lists over longer ones, even if the first ones contribute with fewer new digits. I found it not ideal.

Results

Running the algorithm with different values of N we obtain the following results.

> Of course they depend on the generated list set, but they should be similar to the peer rewiever results

N = 5

Solution found:

- Weight of the solution = 5
- Execution time = 0.1s
- Number of visited nodes = 3

N = 10

Solution found:

- Weight of the solution = 10
- Execution time = 0.1s
- Number of visited nodes = 3

N = 20

Solution found:

- Weight of the solution = 28
- Execution time = 0.1s
- Number of visited nodes = 4

N = 100

Solution found:

- Weight of the solution = 181
- Execution time = 0.1s
- Number of visited nodes = 6

N = 500

Solution found:

- Weight of the solution = 1419
- Execution time = 1.6s
- Number of visited nodes = 11

N = 1000

> Solution found:
>
> - Weight of the solution = 3110
> - Execution time = 6.9s
> - Number of visited nodes = 13

Review 1

- Review to Andrea D'Attila (s303339)

> ## Major issue: **What if there is no valid solution?**
>
> You didn't consider this case in your algorithm.
> Imagine that the function generates lists in which the value `k` is never present, >given k as integer s.t. 0<= k <N. The result is that your cost function `h` will always return `float('inf')` once it has covered all elements except k, since `new_elements = 0` anyhow.
> Your algorithm will provide a new list that will be appended to the solution until the generated problem list will be empty. **Here comes the real problem**: the `min(*iterable*, *[, _key, default_])` function raises a 'ValueError' exception if the provided iterable is empty and there is no default value specified. Give a look at the function doc for more info.
>
> Since you don't handle this problem, **your code could end badly!**
>
> > Possible solution:
> >
> > You could simply check if the cost function returns `float('inf')`: if so, break the loop and declare that there is not a valid solution.
>
> ---
>
> ## Minor issue: **Pay attention to redundancies!**
>
> The lists `flat_sol` and `sol` exploit the same functionality, i.e. they are redundant duplicates. Morover, you do two different operation that should be done on the same and unique "solution list" but once using `flat_sol` and another using `sol`: you compute the cost on the `flat_sol` list but you return the `sol` list.
>
> In this case it's not a problem since it's easy to maintain allined the two lists but per sè it's a best practice to avoid duplicates when it's possibile since it could give you some problem and it could waste your time!

Code reviewed

This is from his jupyter notebook file

```
import logging
import copy

def h(sol, current_x):
    common_elements = len(set(sol) & set(current_x))
```

```
    new_elements = len(current_x) - common_elements
    if (new_elements == 0):
        return float('inf')
    return common_elements/new_elements

def greedy(N):
    goal = set(range(N))

    lists = sorted(problem(N, seed=42), key=lambda l: -len(l))

    starting_x = lists.pop(0)

    sol = list()
    sol.append(starting_x)

    flat_sol = list(starting_x)
    nodes = 1

    covered = set(starting_x)
    while goal != covered:
        most_promising_x = min(lists, key = lambda x: h(flat_sol, x))
        lists.remove(most_promising_x)

        flat_sol.extend(most_promising_x)
        sol.append(most_promising_x)
        nodes = nodes + 1

        covered |= set(most_promising_x)

    w = len(flat_sol)

    logging.info(
        f"Greedy solution for N={N}: w={w} (bloat={(w-N)/N*100:.0f}%) - visited
{nodes} nodes"
    )
    logging.debug(f"{sol}")
    return sol
```

Review 2

- Review to Francesco Carlucci

> # Major issues
>
> ## Generators are slower then sets
>
> In order to check if you reached the goal you use a generator like the following one to compare the elements already covered with another set:

```
set([item for sublist in selected for item in sublist])
```

You do this:

- inside the `goal_test(slz)` function;
- inside the `priority_function(newState)` function;
- inside the inner loop of the `tree_search2(...)` function.

This approach is not optimal since there is a better object that could perform those operations way faster: **sets**!

You could have used a set variable `covered` to hold the set of digits already covered by your solution. To update this set with the new digits added with an expanction you could do:

```
covered |= newlist
```

To verify if your selection of lists is a solution, you could use this simple and optimised piece of code:

```
if coveder == set(range(N))
   return True
else:
   return False
```

To verify if the the new list has new digits to offer to the solution, i.e. if the new list is not a strict subset of the partial solution you could use:

```
if not set(newlist) < covered:
   ....
```

This way should be faster, cleaner and more elegant! 😄

Useless check

Inside the function `tree_search2`, when you find a solution you check if the the current solution `selected` costs less then `solution`:

```
if slzCost( selected ) < slzCost(solution):
   solution = sekected
break
```

This if is useless because `solution=lists` always. That means that if you find a solution that is a strick subset of the list of lists generated by the problem (from now on "problem list") it will necessary have a

> lower cost computed by `slzCost()`. This check could be useful if you try to find other solutions after you have discovered this one, but you don't (because of the `break`).
>
> > **Possible solution**
> >
> > If you mean to mantain the `break`, just don't check the costs with that function.
> >
> > ```
> > if goal_test(selected):
> >     logging.info(f"Founded!")
> >    return selected
> > ```
>
> # Minor issue
>
> It's better to don't import stuff that you will not use: the imports executes code, thus it will slow your script and will in vain take up space. It's just a best practice since if you import a whole library, you could import bad staff or, even worst, there could be an open script that you will run (maybe without knowing what he is doing).
>
> Just clean up the code skeleton before the final commit and your code will be safer and more readable! 😄

Code Reviewed

This is from his the jupyter notebook file

```
from queue import PriorityQueue

def tree_search2(lists, goal_test, priority_function):
    frontier = PriorityQueue()

    state=(set(),(), lists) #initial state

    n=0
    while state is not None:

        selected,solution,available=state

        if goal_test(selected):
            logging.info(f"Found a solution in {n:,} steps: {solution}")
            break
        n+=1

        for i,newlist in enumerate(available):
            if not set(newlist) < selected:

                newState=(selected | set(newlist),solution+
(newlist,),available[i+1 :])
```

```
                frontier.put((priority_function(selected,solution+
(newlist,)),newState))

        if frontier:
            state = frontier.get()[1]
        else:
            state = None

    return solution

    def goal_test_gen(N):
    def goal_test(selected):
        return selected==set(range(N))

    return goal_test

# ----

def priority_function(selected,solution):
    newlist=solution[-1]
    return len(set(newlist)&selected),-len(set(newlist)|selected)

def priority_dijkstra(_,solution):
    cnt = Counter()
    cnt.update(sum((e for e in solution), start=()))
    return sum(cnt[c] - 1 for c in cnt if cnt[c] > 1), -sum(cnt[c] == 1 for c in
cnt)

for N in [5, 10, 20]:
    lists = sorted(problem(N, seed=42), key=lambda l: len(l))
    filteredLists=sorted(list(list(_) for _ in set(tuple(l) for l in lists)),
key=lambda l:len(l))

    tuples=tuple(tuple(sublist) for sublist in filteredLists)

    solution=tree_search2(tuples, goal_test_gen(N), lambda a,b:
priority_function(a,b))
    print(f"Solution for N={N}: w={sum(len(_) for _ in solution)} (bloat=
{(sum(len(_) for _ in solution)-N)/N*100:.0f}%)")

    solution2=tree_search2(tuples, goal_test_gen(N), lambda a,b:
priority_dijkstra(a,b))
    print(f"Dijkstra Solution for N={N}: w={sum(len(_) for _ in solution2)}
(bloat={(sum(len(_) for _ in solution2)-N)/N*100:.0f}%)")

# ---------
# GREEDY
# ---------

def greedy(N, all_lists):
    """Vanilla greedy algorithm"""

    goal = set(range(N))
```

```
    covered = set()
    solution = list()
    all_lists = sorted(all_lists, key=lambda l: len(l))
    while goal != covered:
        x = all_lists.pop(0)
        if not set(x) < covered:
            solution.append(x)
            covered |= set(x)
    logging.debug(f"{solution}")
    return solution

for N in [5, 10, 20,100,500,1000]:
    solution = greedy(N, problem(N, seed=42))
    logging.info(
        f" Greedy solution for N={N:,}: "
        + f"w={sum(len(_) for _ in solution):,} "
        + f"(bloat={(sum(len(_) for _ in solution)-N)/N*100:.0f}%)"
    )
```

Lab 2 - Set covering using a Genetic Algorithm

My code

Gene.py

```python
def create_dict_genes(problem):
  id_to_genes = dict()
  id = 0

  for l in problem:
    id_to_genes[id] = Gene(id, l)
    id += 1

  return id_to_genes

# ---------------------------------------- #

class Gene:
  def __init__(self, id, values):
    self.id = id
    self.values = set(values)

  def __len__(self):
    return len(self.values)

  def __str__(self):
    return f'{self.values}'

  def display(self):
    print(f"{sorted(self.values)}")
```

Genome.py

```python


from copy import deepcopy
import random
from Gene import Gene


class Genome:
  """
  Genome attributes:
  - genome: list of genes
  """

  def __init__(self, genes):
    self.genome = genes
```

```
    def __len__(self):
        len_ = 0
        for gene in self.genome:
            len_ += len(gene)
        return len_


    def covered_values(self):
        covered = set()
        for gene in self.genome:
            covered |= gene.values

        return covered


    def cross_over(self, partner_genome, id_to_genes: dict):
        """
        returns the genome obtained from the crossover of the current genome
        with the genome of the partner individual
        """
        # randomly choose genes (choices between len(genome)//2 and len(genome))
        g_self = random.choices(self.genome,
                                k=random.randint(len(self.genome)//2,
len(self.genome)))
        g_parent = random.choices(self.genome,
                                  k=random.randint(len(partner_genome)//2,
len(partner_genome)))
        survivals = set([gene.id for gene in g_self + g_parent])

        return Genome([id_to_genes[g_id] for g_id in survivals])

    def smart_reproduction(self, partner_genome, id_to_genes: dict, N):
        # filter the duplicates ids -> exstract the candidate genes
        candidates = [id_to_genes[id_] for id_ in set([g.id for g in self.genome +
partner_genome.genome])]

        stop = max(N, len(candidates))
        goal = set(range(N))
        covered = set()
        new_genome = list()

        for i in range(stop):
            best = max(candidates, key=lambda gene: (len(goal) - len(covered |
gene.values), -len(gene))) # max based on how much the gene would contribute to
the solution
            new_genome.append(best)
            candidates.remove(best)
            if not candidates:
                break
        return Genome(new_genome)
```

```
  def mutate(self, id_to_genes: dict):
    genome = deepcopy(self.genome)
    point = random.randint(1, len(self.genome)-1)  # point of mutation
    candidates = set(id_to_genes.keys()) - set([gene.id for gene in self.genome])
# set of genes not present in self.genome

    genome[point] = id_to_genes[random.choice([c for c in candidates])] # Update
the genome
    return Genome(genome)


  def display(self):
    for gene in self.genome:
      gene.display()
```

Individual.py

```
from collections import namedtuple
from Genome import Genome

W_FACTOR = 2
INTW_FACTOR = 10

class Individual:
  """
  Individual:
  - genome = list of genes
  - covered: set of covered values between 0 and N-1
  - cost: weighted number of digits left to find the goal state + the weighted len
of the list
  """
  def ideal_cost(N):
    """
    The cost is computed as follows:
        cost = len(genome) * W_FACTOR + len(goal - self.covered) * INTW_FACTOR
    The ideal cost will have:
    - len(genome) = N (ideal minimum)
    - len(goal - self.covered) = 0
    """
    return N * W_FACTOR


  def __init__(self, genome: Genome, N):
    self.genome = genome    # list of genes
    self.covered = genome.covered_values() # set
    goal = set(range(N))
    self.cost = len(genome) * W_FACTOR + len(goal - self.covered) * INTW_FACTOR


  def __len__(self):
    return len(self.genome)
```

```
    def is_healty(self, N):
      """
        The individual is healty if it's genome contains
        all the numbers between 0 and N-1.
      """
      return len(self.covered) == N


    def fight(self, opponent):
      if self.cost < opponent.cost:
        return self
      else:
        return opponent


    #def reproduce(self, partner, id_to_genes: dict):
    #  return Individual(self.genome.cross_over(partner.genome, id_to_genes))


    def reproduce(self, partner, id_to_genes, N):
        return Individual(self.genome.smart_reproduction(partner.genome,
  id_to_genes, N), N)


    def mutate(self, id_to_genes, N):
      return Individual(self.genome.mutate(id_to_genes), N)


    def display(self):
      return self.genome.display()
```

lab2.ipynb

```
# ---
# Setup
# ---
import random
from matplotlib import pyplot as plt
from tqdm import tqdm   # pip install
"git+https://github.com/tqdm/tqdm.git@devel#egg=tqdm"

from Gene import create_dict_genes
from Individual import Individual
from Genome import Genome

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))
  ]

# ---
# Global functionalities and parameters
# ---
POPULATION_SIZE = 5
OFFSPRING_SIZE = 3
NUM_GENERATIONS = 1000

def tournament(population):
  x, y = tuple(random.choices(population, k=2))
  return x.fight(y)

def initial_population(id_to_genes, N):
  population = list()
  genes = list(id_to_genes.values())
  tot_genes = len(id_to_genes)

  for i in range(POPULATION_SIZE):
    genes = random.choices(genes, k=random.randint(1, N))
    population.append(Individual(Genome(genes), N))

  return population

def plot_gen_best(fitness_log):
  gen_best = [max(f[1] for f in fitness_log if f[0] == x) for x in
range(NUM_GENERATIONS)]

  plt.figure(figsize=(15, 6))
  plt.ylabel("cost")
  plt.xlabel("generations")
  #plt.scatter([x for x, _ in fitness_log], [y for _, y in fitness_log],
marker=".", label='fitness_log')
  plt.plot([x for x, _ in enumerate(gen_best)], [y for _, y in
enumerate(gen_best)], label='gen_best')
  plt.legend()

def print_statistics(winner, N):
  print(f"Genetic Algorithm (N={N}):")
  print(f"\tsuccess = {len(winner.covered)*100/N}%")
  print(f"\tgenome length = {len(winner)}")
  print(f"\tlength - idael_length = {len(winner) - N}")
  print(f"Genome:")
  winner.display()

# ---
# Evolution
# ---

def genetic_algorithm(population, id_to_genes, N):
  fitness_log = [(0, i.cost) for i in population]

  for g in tqdm(range(NUM_GENERATIONS)):
```

```
    offspring = list()
    for i in range(OFFSPRING_SIZE):
      p1 = tournament(population)
      p2 = tournament(population)
      o = p1.reproduce(p2, id_to_genes, N)

      if random.random() < 0.3:
        o = o.mutate(id_to_genes, N)

      fitness_log.append((g+1, o.cost))
      offspring.append(o)
    population += offspring
    population = sorted(population, key=lambda i: i.cost)[:POPULATION_SIZE]

  winner = max(population, key=lambda i: i.cost)
  return winner, fitness_log

# ---
# Main
# ---

N = 5

clean_problem = set([tuple(sorted(l)) for l in problem(N, 42)])
id_to_genes = create_dict_genes(clean_problem) # Dictionaries to map genes to ids
and vice versa
population = initial_population(id_to_genes, N)
winner, fitness_log = genetic_algorithm(population, id_to_genes, N)

print_statistics(winner, N)
plot_gen_best(fitness_log)
```

My README

> # Lab2 - Set covering using a Genetic Algorithm
>
> > Author: Luigi Federico
> >
> > Computational Intelligence 2022/23
> > Prof: G. Squillero
>
> ---
>
> ## Solution proposed
>
> I implemented a genetic algorithm that operates as follows:
>
> > For each generation:
> >
> > 1. Randomply select two parents, both through a tournament
> > 2. Let those parents reproduce, generating the offspring

> 3. With a certain probability, the offspring could mutate
> 4. After having generated the entire offspring, they are added to the starting population and sorted baserd on the individual cost. Cost-based evolutionary selection is applied to have the population of the next generation.
>
> The best individual of the last generation will be the winner of the selection, i.e. the proposed solution

Reproduction

The reproduction is "smart", i.e. it's not a random crossover but there is a selection of the genes that will compose the genome of the offspring individual.

The reproduction between two individuals works as follows:

1. All the genes of both the genomes are grouped toghether and the duplicate are removed.
2. The genes are iterativelly selected by looking for the best gene among the candidates. The selection is based on how much the gene would contribute to the solution and on the length of the gene.

```
best = max(candidates, key=lambda gene: (len(goal) - len(covered |
gene.values), -len(gene)))
```

3. When there are no candidates left or it has selected a maximum of N genes, the genome of the new individual is ready.

Mutation

The mutation concerns a randomly chosen gene of the genome that is replaced with another gene (different from the other already present inside the genome)

Results

Running the algorithm with different values of N we obtain the following results.

> The winner statistics are the following:
>
> - Success = 100 * number_of_covered_values / N
> - Cost = weighted number of digits left to find the goal state + the weighted len of the list
> - Ideal_Cost = W_FACTOR * N
> - Genome = set of covered values

N=5

Winner individual:

▶ Genome (click me):

- Success = 100.0%
- Genome Length = 5
- Length - Ideal_Length = 0
- Cost of best individual per generation: Best individuals per generation

---

N=10

Winner individual:

▶ Genome (click me):

- Success = 100.0%
- Genome Length = 13
- Length - Ideal_Length = 3
- Cost of best individual per generation: Best individuals per generation

---

N=20

Winner individual:

▶ Genome (click me):

- Success = 100.0%
- Genome Length = 29
- Length - Ideal_Length = 9
- Cost of best individual per generation: Best individuals per generation

---

N=100

Winner individual:

- Success = 100.0%
- Genome Length = 261
- Length - Ideal_Length = 161
- Cost of best individual per generation: Best individuals per generation

---

N=500

- Success = 100.0%
- Genome Length = 11757
- Length - Ideal_Length = 11257
- Cost of best individual per generation: Best individuals per generation

---

> **N=1000**
>
> - Success = 100.0%
> - Genome Length = 24505
> - Length - Ideal_Length = 23505
> - Cost of best individual per generation: Best individuals per generation
>
> ---

Review 1

- Review to Francesco Scalera

> The algorithm appear well thought and with good strategies, so: **well done!** 👏😃
> The following "issues" are just minor suggestions or small possible improvements.
>
> ## The algorithm seems to be slow
>
> Your algorithm could be very slow if `N` is really big because of the fact that you want only feasible solutions inside your population. This constrain of "only feasible solutions" represents the bottleneck. The thing is that you could loop for a really high amount of time, also if you have an individual that needs to tweak correctly just one list!
>
> You could see this behavior inside the algorithm when, after creating the offspring, you check if this is feasible. If it's not, you discard it. The thing is that you loop until you have `OFFSPRING_SIZE` new individuals and if your mutations/crossover is unlucky you could discard a lot of individuals before finding a good population. This could be very time consuming.
>
> My suggestion is to add a mechanism that pick a new list in a smart way, maybe if your loop didn't generate any good individual after *n* attempts.
>
> > By the way, I'm not very sure about how well it could improve or if it is really a performance issue... It should be tested to understand how much is probable to find a feasible solution instead of an unfeasible one, just to understand if this approach is performing well or it could be way more faster.
>
> ## Better use of sets
>
> Inside the function `check_feasible(•)` you use two nested loops to extract the single numbers covered by the individual.
>
> ```
> def check_feasibile(individual, N):
>     '''From np array of Lists and size of problem, returns if it provides a possible solution '''
>     goal = set(list(range(N)))
>     coverage = set()
>     for list_ in individual:
>         for num in list_:
>             coverage.add(num)
>         if coverage == goal:
> ```

```
            return True
    return False
```

You could use the optimization that comes with sets to achieve the same result, but in a more fast and elegant way. You could use the `|=` operator to compute the intersection between two sets just like this:

```
for list_ in individual:
    coverage |= set(list_)
    if coverage == goal:
        return True
return False
```

The readability could be improved

Different names for the same object

There is some confusion between the names of some data structures that affect the readability.
In your algorithm you use a list called `mutation_probability_list`. This suggests that it will contain a list of mutation probabilities but when you pass to the function `calculate_mutation_probabilityDet2(•)` you alias this list as `best_candidate_list`. This name seems more appropriate since the structure actually contains only the individual with the same best fitness.

My suggestion is to maintain parallel the names if the function is exactly the same, just to avoid redundancies.

Another thing about this list: maybe it could be replaced with a counter and a variable. This should improve readability since your use of this list is to count how many times your best fitness repeats. So, why don't just use a counter to count? 😉

Not ideal use of the Jupyter Notebook format

The jupyter notebook format is a really powerful tool to divide code in blocks, inserting some text to better structure the code. Your [lab2.ipynb] file looks like a .py file more than a .ipynb file!

My suggestion is to isolate all the single pieces that is logically self-complete.

- You could separate in different blocks all the functions, giving a title to it if needed.
- You could isolate the part with all the parameters, so if you want to change/add just one parameter you don't have to run again all the code since you can run just that part alone!

Another cool thing that you could do is to save the output of your code inside the .ipynb file, so who wants to read your code could see the output without running it.

Code reviewed

This is from his jupyter notebook file

```
from base64 import decode
import random
import numpy as np
from sklearn.model_selection import ParameterGrid
from tqdm import tqdm
import time
# import sys

# Function for the problem
def problem(N, seed=42):
    """Generates the problem, also makes all blocks generated unique"""
    random.seed(seed)
    blocks_not_unique = [
        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))
    ]
    blocks_unique = np.unique(np.array(blocks_not_unique, dtype=object))
    return blocks_unique.tolist()

def check_feasibile(individual, N):
    '''From np array of Lists and size of problem, returns if it provides a
possible solution '''
    goal = set(list(range(N)))
    coverage = set()
    for list_ in individual:
        for num in list_:
            coverage.add(num)
        if coverage == goal:
            return True
    return False

def createFitness(individual):
    fitness = 0
    for list_ in individual:
        fitness += len(list_)
    return fitness

def select_parent(population, tournament_size = 2):
    subset = random.choices(population, k = tournament_size)
    return min(subset, key=lambda i: i [0])

def cross_over(g1,g2, len_):
    cut = random.randint(0,len_-1)
    return g1[:cut] + g2[cut:]
# cross_over con più tagli

def mutation(g, len_):
    point = random.randint(0,len_-1)
    return g[:point] + [not g[point]] + g[point+1:]

def calculate_mutation_probability(best_candidate, N):
    distance = abs(N - best_candidate[0])
```

```
    return 1-(distance/N)

best_candidate_option = ""

def calculate_mutation_probabilityDet2(best_candidate, N, best_candidate_list):
    global best_candidate_option

    probability_selected = 0.5
    probability_reason = ""

    # check if best changed (based on fitness func)
    if not best_candidate[0] == best_candidate_list[-1][0]:
        best_candidate_list.clear()
        best_candidate_list.append(best_candidate)
    else:
        best_candidate_list.append(best_candidate)

    # if list is bigger than 10 select opositive of current best
    if len(best_candidate_list) > 10:

        if len(best_candidate_list) < 21:
            if best_candidate[2] == "mutation":
                probability_reason= "cross"
                probability_selected = 0.1
            else:
                probability_reason= "mutation"
                probability_selected = 0.9
        else:
            probability_reason = best_candidate_option

        if len(best_candidate_list) % 20 == 0:
            if best_candidate_option == "mutation":
                probability_reason= "cross"
                probability_selected = 0.1
            else:
                probability_reason= "mutation"
                probability_selected = 0.9
    else:
        probability_reason = "distance-based"
        probability_selected = calculate_mutation_probability(best_candidate, N)

    best_candidate_option = probability_reason
    return probability_selected

PARAMETERS = {
    "N":[20, 100, 500, 1000, 5000],
    "POPULATION_SIZE":[50, 200, 300, 500, 600, 1000, 2000, 3000, 5000],
    "OFFSPRING_SIZE":[int(50*2/3), int(200*2/3), int(300*2/3), int(500*2/3),
int(600*2/3), int(1000*2/3), int(2000*2/3), int(3000*2/3), 5000*(2/3)]
    # number of iterations? as 1000 is too small for some N values
}

configurations = {"configurations": []}
my_configs = ParameterGrid(PARAMETERS)
```

```
for config in my_configs:
    configurations["configurations"].append(config)

#Inital list of lists
random.seed(42)

with open("results.csv", "a") as csvf:
    header="N,POPULATION_SIZE,OFFSPRING_SIZE,fitness\n"
    csvf.write(header)

    for idx in tqdm(range(len(configurations["configurations"]))):

        config = configurations["configurations"][idx]

        start = time.time()

        initial_formulation = problem(config['N'])
        initial_formulation_np = np.array(initial_formulation, dtype=object)

        mutation_probability_list = list()
        mutation_probability_list.append((None, None, ""))
        population = list()

        # we use a while since if the checks will give always false, i can also
have a population that too little in size
        while len(population) != (config['POPULATION_SIZE']):
            # list of random indexes
            # this avoid duplicate samples of the same index when initializing the
first individuals
            random_choices = random.choices([True, False],
k=len(initial_formulation))
            # np array of lists based on random indexes
            individual_lists = initial_formulation_np[random_choices]
            if check_feasibile(individual_lists,config['N']) == True:
                population.append((createFitness(individual_lists),
random_choices, ""))

        for _ in range(1000):
            # print(f"interation {_}; w:{population[0][0]}; best calculated:
{population[0][2]}")
            sum_of_cross = 0
            sum_of_mut = 0
            offspring_pool = list()
            offspring_pool_mask = list()
            i = 0
            mutation_probability =
calculate_mutation_probabilityDet2(population[0], config['N'],
mutation_probability_list)
            while len(offspring_pool) != config['OFFSPRING_SIZE']:
                reason = ""
                if random.random() < mutation_probability:
                    p = select_parent(population)
                    sum_of_mut += 1
                    offspring_mask = mutation(p[1], len(initial_formulation))
```

```
                    offspring_mask = mutation(offspring_mask,
len(initial_formulation))
                    reason = "mutation"
                else:
                    p1 = select_parent(population)
                    p2 = select_parent(population)
                    sum_of_cross += 1
                    offspring_mask = cross_over(p1[1],p2[1],
len(initial_formulation))
                    reason = "cross"

                offspring_lists = initial_formulation_np[offspring_mask]
                if check_feasibile(offspring_lists, config['N']) == True and
offspring_mask not in offspring_pool_mask:
                    offspring_pool.append((createFitness(offspring_lists),
offspring_mask, reason))
                    offspring_pool_mask.append(offspring_mask)

            population = population + offspring_pool
            unique_population = list()
            unique_population_mask = list()
            for ind in population:
                if ind[1] not in unique_population_mask:
                    unique_population.append(ind)
                    unique_population_mask.append(ind[1])
            unique_population=list(unique_population)
            unique_population.sort(key=lambda x: x[0])
            # take the fittest individual
            population = unique_population[:config['POPULATION_SIZE']]

        end = time.time()
        csvf.write(f"{config['N']},{config['POPULATION_SIZE']},
{config['OFFSPRING_SIZE']},{population[0][0]},{end-start}\n")
```

Review 2

- Review to Leonor Gomes

> # Major issues
>
> ## Crossover
>
> After splitting the parents, you generate the children by gluing the splitted genomes **without checking if the offspring contains duplicate lists**.
> This affects badly your fitness function because the length of the genome will increase:
>
> ```
> fitness = N * size_genome /unique_values
> ```

> This could be a problem if the fitness becomes worse just because of the noisy duplicates, also if the solution was a really good one if it had not the clones.
>
> **Exepli Gratia**:
>
> > N = 5 Individual with noise:
> > --> [0] [1, 2, 3] [3, 4] **[1, 2, 3]** --> fitness = 5 * 8 / 5 = 8
> > Individual without noise: --> [0] [1, 2, 3] [3, 4] --> fitness = 5 * 5 / 5 = 5 Worst individual but with the same fitness of the noisy good individual: --> [0, 1] [1, 2, 3] [2, 3, 4] --> fitness = 5 * 8 / 5 = 8
> >
> > The quality of these two individuals is identical for your algorithm!
>
> ### Possible solutions:
>
> You could choose different strategies. Here are some:
>
> 1. If the list is already selected, delete the list from the individual.
>
> > This could be seen as a mutation.
> >
> > - Pro: this kind of mutation could generate offspring that could lead you out of a local optimum. This could be useful after a lot of generations, when the individual are quite all the same, favoring exploitation.
> > - Con: If that gene was a really good one, the risk is to affect badly the individual fitness.
>
> 2. If the list is already selected, just don't insert it again!
>
> > The most basic approach!
> >
> > - Pro: easy to implement and correct the noise problem described above.
> > - Con: If you use this approach with the mutation (read the next point), you just waste the mutation!
>
> ## Mutation
>
> When you select a random list of the genome to perform the mutation, you pick the new list from the entire pool of lists with the possibility of pick a noisy duplicate that affect badly the fitness of the individual. **It's just the same problem as before!**
>
> ## Genetic Algorithm
>
> If you obtain 10 times the same best fitness, you mutate the entire population.
> Issue: you don't keep track of the absolute best solution that your algorithm generated so far. What if it generates the global optimum solution and the fitness will be the same for the next generations? You will discard it, losing the best solution! And you have no guarantee that you will find it again!
>
> ### Possible solution:
>
> You could keep a variable that contains the best individual so far and update it every generation. In this way, not only you will always have a trace of the best individual generated by your algorithm, but if you

> change the whole population, you don't forget that individual! This is good if the mutations are worsening the entire individual population.
>
> # Minor issues
>
> ## Generation of the next population
>
> It might be more optimal to merge the current population with the offspring and to perform a single sort to let survive the best solutions among the merged population. What if the entire offspring population just generated have a very bad fitness if compared with the parent population? With your way of generating the next population, you are discarding part of the population with better fitness than the offspring!
>
> > Let's notice that those "weak" individual will die in the next generation anyway.
>
> **Is it worth sacrificing potentially better solutions?** If your intention was to also retain individuals with potentially pejorative fitness than the previous generation average, then it can be an interesting strategy hoping that a crossover in the next generation will lead to a better individual.
>
> We should test and understand which of the two approaches is better: always select the absolute best or favor individuals potentially worse than others?
>
> ## Poorly readable README
>
> If you want to paste the genome you could use the following trick to make it collapsable inside a markdown file.
>
> Just wrap your genome like this:
>
> ```
> <details>
>  <summary> Click me to display the genome </summary>
>
>  genome lists
>
> </details>
> ```
>
> Feel free to look at my lab2 README to see how it works! 😃
>
> This should make your README more readable and more interactive!

Code reviewed

This is from her jupyter notebook file

```
import logging
import random
from copy import copy
```

```
from collections import namedtuple
from operator import attrgetter


def problem(N, seed=None):
    """Creates an instance of the problem"""

    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))
    ]

Individual = namedtuple('Individual', ['genome', 'fitness'])
TOURNAMENT_SIZE = 5
POPULATION_SIZE = 30


#part of code from https://stackoverflow.com/questions/952914/how-do-i-make-a-
flat-list-out-of-a-list-of-lists

def unique(sol): # number of unique values in a possible solution/individual
   unique = set([item for sublist in sol for item in sublist])
   return len(unique)

#unique values
#size of solution
#we want to minimize fitness
def fitness(genome, N):
  fit = 0
  size = len([item for sublist in genome for item in sublist])

  unique_values = unique(genome)
  fit = N*(size/unique_values) #relation between current size of the list and the
amount of unique values
  values_left = N - unique_values

  if (values_left > 0):
    fit = fit + N*values_left #adds values left

  return fit

#mutates a random gene(list) and substitutes it with a possible list from the
problem
def mutation(individual, P, N):
  index = random.randrange(len(individual.genome))
  old_gene = individual.genome.pop(index)

  P_index = random.randrange(len(P))
  new_gene = P[P_index]

  new_genome = individual.genome + [new_gene]
  new_fitness = fitness(new_genome, N)
```

```
  new_individual = Individual(new_genome, new_fitness)

  return new_individual

# takes a random interval and generates two children, mixed from two parents
def crossover(first_individual, second_individual, N):
  min_size = min(len(first_individual.genome), len(second_individual.genome))

  interval = random.randrange(min_size) #interval can't be bigger than one of the individuals

  first_child_genome = first_individual.genome[:interval] + second_individual.genome[interval:]
  second_child_genome = second_individual.genome[:interval] + first_individual.genome[interval:]

  first_child = Individual(first_child_genome, fitness(first_child_genome, N))
  second_child = Individual(second_child_genome, fitness(second_child_genome, N))


  return first_child, second_child

# generates a random individual from the problem lists
def generate_individual(P):

  individual_size = random.randrange(1, len(P))

  individual = random.sample(P, individual_size) #gets a random sized sample of lists from the problem

  return individual

# generates random initial population
def generate_population(P, N):
  population = []

  for i in range(POPULATION_SIZE):
    new_individual = generate_individual(P)

    if new_individual not in population: #checks if the individual is already in the population
      population.append(Individual(new_individual, fitness(new_individual, N)))

  return population


# tournament selection
def select_parent_tournament(population):
  tournament_selection = []

  while len(tournament_selection) != TOURNAMENT_SIZE: #randomly select a small subset of the population to compete against each other
    id = random.randrange(len(population))
```

```
    tournament_selection.append(population[id])

  tournament_selection.sort(key=attrgetter('fitness'))

  return tournament_selection[0] #returns the fittest from the subset of the
tournament


def should_mutate(): #given a small possibility, should we mutate or not?
  if random.random() < 0.3:
    return True
  return False


#generates new offspring

def create_offspring(population, P, N):
  #generate POPULATION_SIZE offspring

  offspring = []

  for i in range(int(POPULATION_SIZE/2)): #for each iteration -> 2 new children
    tournament_parent = select_parent_tournament(population) #selects first parent
from tournament

    random_id = random.randrange(POPULATION_SIZE)
    random_parent = population[random_id] #selects second parent randomly

    first_child, second_child = crossover(tournament_parent, random_parent, N)
#gets two new children from crossover

    if should_mutate():  #given a small possibility -> mutate new child
      first_child = mutation(first_child, P, N)
    if should_mutate(): #given a small possibility -> mutate new child
      second_child = mutation(second_child, P, N)

    offspring.append(first_child) #add new child to offspring
    offspring.append(second_child) #add new child to offspring

  return offspring


#sorts the current population and the new offspring - gets the best half from the
current population and the best half from the offspring
def get_new_population(population, offspring):
  new_population = []

  population.sort(key=attrgetter('fitness'))
  offspring.sort(key=attrgetter('fitness'))

  best_fitness = min(population[0].fitness, offspring[0].fitness)

  new_population = population[:int(POPULATION_SIZE/2)] +
```

```
offspring[:int(POPULATION_SIZE/2)]
  return new_population, best_fitness


def print_fitness(population): #debug function to print the fitness of a
population
  for i in range(len(population)):
    print(population[i].fitness)


def escape_local_optimum(population, P, N): #mutate the entire population in the
hopes of escaping local optimum
  new_population = []

  for i in range(len(population)):
    new_individual = mutation(population[i], P, N)
    new_population.append(new_individual)

  return new_population


#steady state

def genetic_algorithm(P, N, generations = 100):

  #first generate a population
  population = generate_population(P, N)

  population.sort(key=attrgetter('fitness'))

  current_best_fitness = population[0].fitness #var to check if the solution is
improving
  counter = 0

  #termination criteria - number of desired generations
  for i in range(generations):

    offspring = create_offspring(population, P, N) #creates offspring
    population, new_best_fitness = get_new_population(population, offspring) #gets
the new population with half from the fittest parents and the other half with the
fittest children

    if (new_best_fitness == current_best_fitness):
      counter += 1
    else:
      counter = 0
      current_best_fitnes = new_best_fitness

    if (counter == 10):
      population = escape_local_optimum(population, P, N) #if the solution hasn't
improved for 10 generations, we try to escape local optimum

  population.sort(key=attrgetter('fitness')) #sorts the population by fitness
```

```python
    return population[0] #returns the individual with best fitness


def get_results(N, number_generations = 100):
    p = problem(N, 42)
    solution = genetic_algorithm(p, N, number_generations)
    print(f'LEN: {sum([len(l) for l in solution.genome])}')
    return solution
```

Lab 3 - Policy Search

My code

nim.py

```
# Nim class created by professor Giovanni Squillero:
#    Copyright **`(c)`** 2022 Giovanni Squillero `<squillero@polito.it>`
#    [`https://github.com/squillero/computational-intelligence`]
(https://github.com/squillero/computational-intelligence)
#    Free for personal or classroom use; see [`LICENSE.md`]
(https://github.com/squillero/computational-intelligence/blob/master/LICENSE.md)
for details.

from collections import namedtuple

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

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

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

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

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

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

    def is_game_over(self):
        # This method is made by me
        return sum(self._rows) == 0
```

play_nim.py

```python
import random
import logging
from copy import deepcopy
from nim import Nimply, Nim


def nim_sum(elem: list):
  x = 0
  for e in elem:
    x = e ^ x
  return x

##################
##  STRATEGIES  ##
##################

# Level 0: Easy
def dumb_action(nim: Nim):
  """
  Always takes one obj from the first row available
  """
  row = [r for r, n in enumerate(nim._rows) if n > 0][0]
  return Nimply(row, 1)


# Level 1: Medium
def dumb_random_action(nim: Nim):
  """
  There is 0.5 of probability to make a dumb action or a random action
  """
  if random.random() < 0.5:
    return dumb_action(nim)
  else: return random_action(nim)


# Level 2: Medium-Advanced
def random_action(nim: Nim):
  """
    The agent performs a random action
  """
  row = random.choice([r for r, n in enumerate(nim._rows) if n > 0])
  if nim._k:
    num_obj = random.randint(1, min(nim._k, nim._rows[row]))
  else:
    num_obj = random.randint(1, nim._rows[row])

  return Nimply(row, num_obj)


# Level 3: Medium-Advanced
def layered_action(nim: Nim):
  """
```

```
    Always takes the whole row's objs choosing randomly the row
    """
    row, num_obj = random.choice([(r, n) for r, n in enumerate(nim._rows) if n > 0])
    if nim._k and num_obj > nim._k:
      return Nimply(row, nim._k)
    return Nimply(row, num_obj)


# Level 4: DEMIGOD
def demigod_action(nim: Nim, prob_god=0.5):
    """
      There is a probability prob to play an expert move and a chance to play
randomly
    """
    if random.random() < prob_god:
      return expert_action(nim)
    else: return random_action(nim)


# Level 5: GOD
def expert_action(nim: Nim):
    """
      The agent uses fixed rules based on nim-sum (expert-system)
      Returns the index of the pile and the number of pieces removed as a Nimply
namedtuple
    """
    board = nim._rows
    k = nim._k

    # Winning move if there is only one row left
    tmp = [(i, r) for i, r in enumerate(board) if r > 0]
    if len(tmp) == 1:
      row, num_obj = tmp[0]
      if not k or num_obj <= k:
        return Nimply(row, num_obj) # Take the entire row


    # Compute the nim-sum of all the heap sizes
    x = nim_sum(board)

    if x > 0:
      # Current player on a insucure position -> is winning
      # --> Has to generate a secure position (bad for the other player)
      # --> Find a heap where the nim-sum of X and the heap-size is less than the
heap-size.
      # --> Then play on that heap, reducing the heap to the nim-sum of its original
size with X

      good_rows = [] # A list is needed because of k
      for row, row_size in enumerate(board):
        if row_size == 0:
          continue
        ns = row_size ^ x # nim sum
        if ns < row_size:
```

```
            good_rows.append((row, row_size)) # This row will have nim sum = 0

        for row, row_size in good_rows:
          board_tmp = deepcopy(board)
          for i in range(row_size):
           board_tmp[row] -= 1
           if nim_sum(board_tmp) == 0:  # winning move
            num_obj = abs(board[row] - board_tmp[row])
            if not k or num_obj <= k:
              return Nimply(row, num_obj)

    # x == 0 or k force a bad move to the player
    # Current player on a secure position or on a bad position bc of k -> is losing
    # --> Can only generate an insicure position (good for the other player)
    # --> Perform a random action bc it doesn't matter
    return random_action(nim)


opponents = {
    1: dumb_action,
    2: dumb_random_action,
    3: random_action,
    4: layered_action,
    5: demigod_action,
    6: expert_action
}



####################
##  PLAY MATCHES  ##
####################

def evaluate(nim: Nim, n_matches=20, *, my_action, opponent_action=random_action,
debug=False):
    """
        This function let you evaluate how many matches your strategy wins against an
opponent.
        You are player 0.
        Input:
          - nim: Nim
          - n_matches=20
          - my_action
          - opponent_action=random_action
          - debug=False # let's you display all the match moves for each match
        Output:
          - Percentage of won matches (number of wins / number of matches)
    """

    if debug:
        logging.getLogger().setLevel(logging.DEBUG)

    player_action = {
        0: my_action, # our champion
```

```
      1: opponent_action # our opponent
      }

    won = 0

    for m in range(n_matches):
      # Setup match
      nim_tmp = deepcopy(nim)
      if m/n_matches > 0.5:
        player = 1  # You start
      else:
        player = 0  # Opponent starts

      logging.debug(f'Board -> {nim_tmp}\tk = {nim_tmp._k}')
      logging.debug(f'Player {1-player} starts\n')

      # Play the match
      while not sum(nim_tmp._rows) == 0:
        player = 1 - player
        ply = player_action[player](nim_tmp)
        #logging.debug(f'Action P{player} = {ply}')
        nim_tmp.nimming(ply)
        logging.debug(f'player {player} -> {nim_tmp}\tnim_sum =
{nim_sum(nim_tmp._rows)}')

      logging.debug(f'\n### Player {player} won ###\n')
      if player == 0:
        won += 1

    return won/n_matches
```

lab3_task1.ipynb

```
# ---
# Task 3.1 - An agent using fixed rules based on nim-sum
# Based on the explanation available here: https://en.wikipedia.org/wiki/Nim
#
# It wants to finish every move with a nim-sum of 0, called 'secure position'
(then it will win if it does not make mistakes).
# ---

import logging
from copy import deepcopy
from nim import Nimply, Nim
from play_nim import nim_sum, random_action, evaluate

logging.basicConfig(format="%(message)s", level=logging.INFO)

# ---
```

```
# Implementation
# ---

def expert_action(nim: Nim):
  """
    The agent uses fixed rules based on nim-sum (expert-system)

    Returns the index of the pile and the number of pieces removed as a Nimply
namedtuple
  """
  board = nim._rows
  k = nim._k

  # Winning move if there is only one row left
  tmp = [(i, r) for i, r in enumerate(board) if r > 0]
  if len(tmp) == 1:
    row, num_obj = tmp[0]
    if not k or num_obj <= k:
      return Nimply(row, num_obj) # Take the entire row


  # Compute the nim-sum of all the heap sizes
  x = nim_sum(board)

  if x > 0:
    # Current player on a insucure position -> is winning
    # --> Has to generate a secure position (bad for the other player)
    # --> Find a heap where the nim-sum of X and the heap-size is less than the
heap-size.
    # --> Then play on that heap, reducing the heap to the nim-sum of its original
size with X

    good_rows = [] # A list is needed because of k
    for row, row_size in enumerate(board):
      if row_size == 0:
        continue
      ns = row_size ^ x # nim sum
      if ns < row_size:
        good_rows.append((row, row_size)) # This row will have nim sum = 0

    for row, row_size in good_rows:
      board_tmp = deepcopy(board)
      for i in range(row_size):
       board_tmp[row] -= 1
       if nim_sum(board_tmp) == 0:  # winning move
        num_obj = abs(board[row] - board_tmp[row])
        if not k or num_obj <= k:
          return Nimply(row, num_obj)

  # x == 0 or k force a bad move to the player
  # Current player on a secure position or on a bad position bc of k -> is losing
  # --> Can only generate an insicure position (good for the other player)
  # --> Perform a random action bc it doesn't matter
  return random_action(nim)
```

```
# ---
# Play
# ---
nim = Nim(7)
evaluate(nim, 20, my_action=expert_action)
```

lab3_task2.ipynb

```
# ---
# Task 3.2 - An agent using evolved rules
# Here for semplicity, I will consider the parameter k always equal to None
# ---

import logging
import random
from tqdm import tqdm
from matplotlib import pyplot as plt

from nim import Nimply, Nim
from play_nim import *
# inside play_nim:
#   Functions:  nim_sum, dumb_action, dumb_random_action,
#               random_action, layered_action, demigod_action,
#               expert_action, evaluate
#   Dictionary: opponents

logging.basicConfig(format="%(message)s", level=logging.INFO)


# ---
# Implementation
# ---

# Individual
class EvolvedPlayer():
  """
    This played uses GA to evolve some rules
    that lets him play the game (hopefully better every time).

    The genome will be a list of rules that will be applyed
    in order. The information lies inside the order of the rules, that
    can change with genetic operations (XOVER and MUT).
  """

  def __init__(self, nim: Nim, genome=None):
    self._k = nim._k
    self.score = -1
    self.__collect_info(nim) # Cooked info
```

```
    self.rules = self.__rules()
    if genome:
      assert len(genome) == len(self.rules)
      self.genome = genome
    else:
      self.genome = list(self.rules.keys())
      random.shuffle(self.genome)


  def __collect_info(self, nim: Nim):
    """
      Collects some info:
      - number of not zero rows
      - sorted rows by number of objects
      - average number of objects per row
    """
    self.n_rows_left = len([r for r in nim._rows if r > 0])
    self.sorted_rows = sorted([(r, n) for r, n in enumerate(nim._rows) if n > 0],
key=lambda r: -r[1])
    self.avg_obj_per_row = sum(nim._rows) / len(nim._rows)


  def __rules(self):
    """
    Returns a set of fixed rules as a dicitonary.
    The dictionary will be as follows:
      - key: id as incremental number
      - value: tuple with (condition, action), where
        - condition = boolean condition that has to be true in order to perform
the action
        - action = Nimply action
    """

    assert self.n_rows_left
    assert self.sorted_rows
    assert self.avg_obj_per_row

    ### Conditions and Actions ###

    # 1 row left --> take the entire row
    def c1(self):
      return self.n_rows_left == 1
    def a1(self):
      return Nimply(self.sorted_rows[0][0], self.sorted_rows[0][1])

    # 2 rows left --> leave the same number of objs
    def c2(self):
      return self.n_rows_left == 2 and self.sorted_rows[0][1] !=
self.sorted_rows[1][1]
    def a2(self):
      num_obj = self.sorted_rows[0][1] - self.sorted_rows[1][1]
      return Nimply(self.sorted_rows[0][0], num_obj)

    # 2 rows left and len longest row > avg --> leave one obj in the higher row
```

```
        def c3(self):
          return self.n_rows_left == 2 and self.sorted_rows[0][1] >
self.avg_obj_per_row and self.sorted_rows[0][1]>1
        def a3(self):
          return Nimply(self.sorted_rows[0][0], self.sorted_rows[0][1] - 1)

        # 3 rows left --> take the entire max row
        def c4(self):
          return self.n_rows_left == 3
        def a4(self):
          return Nimply(self.sorted_rows[0][0], self.sorted_rows[0][1])

        # 3 rows left --> take leave the longest row with one obj
        def c5(self):
          return self.n_rows_left == 3 and self.sorted_rows[0][1] > 1
        def a5(self):
          return Nimply(self.sorted_rows[0][0], self.sorted_rows[0][1] - 1)

        # avg < max+1 --> take the longest row
        def c6(self):
          return self.avg_obj_per_row < self.sorted_rows[0][1] + 1
        def a6(self):
          return Nimply(self.sorted_rows[0][0], self.sorted_rows[0][1])

        # default -> take one obj from the longest row
        def c7(self):
          return True
        def a7(self):
          return Nimply(self.sorted_rows[0][0], 1)


        ### Rule dictionary ###
        dict_rules = {
          1: (c1, a1),
          2: (c2, a2),
          3: (c3, a3),
          4: (c4, a4),
          5: (c5, a5),
          6: (c6, a6),
          7: (c7, a7)
        }

        return dict_rules


      def ply(self, nim: Nim):
        """
          Check the rules in order. The first rule that matches will be applyed
        """
        # Update the informations
        self.__collect_info(nim)

        # Apply a rule
        for key in self.genome:
```

```
        cond, act = self.rules[key]
        if cond(self):
          logging.debug(f'--> RULE number {key} applyed')
          return act(self)


    def set_score(self, score):
      self.score = score


    def clear_score(self):
      self.score = -1


    def cross_over(self, partner, nim):
      """
        Cycle crossover: choose two loci l1 and l2 (not included) and copy the
segment
        between them from p1 to p2, then copy the remaining unused values
      """
      locus1 = random.randint(0, len(self.genome)-1)
      while (locus2 := random.randint(0, len(self.genome)-1)) == locus1:
        pass
      if locus1 > locus2:
        tmp = locus1
        locus1 = locus2
        locus2 = tmp

      # Segment extraction
      segment_partner = partner.genome[locus1:locus2] # slice
      alleles_left = [a for a in self.genome if a not in segment_partner]
      #random.shuffle(alleles_left)

      # Create the offspring genome
      piece1 = alleles_left[:locus1]
      piece2 = alleles_left[locus1:]
      offspring_genome = piece1 + segment_partner + piece2

      return EvolvedPlayer(nim, offspring_genome)


    def mutation(self):
      """
        Swap mutation: alleles in two random loci are swapped
      """
      locus1 = random.randint(0, len(self.genome)-1)
      while (locus2 := random.randint(0, len(self.genome)-1)) == locus1:
        pass

      # Swap mutation
      tmp = self.genome[locus1]
      self.genome[locus1] = self.genome[locus2]
      self.genome[locus2] = tmp
```

```
# Evolution
def initial_population(population_size: int, nim: Nim):
  population = []
  for i in range(population_size):
    population.append(EvolvedPlayer(nim))

  return population


def tournament(population, tournament_size=2):
  return max(random.choices(population, k=tournament_size), key=lambda i: -
i.score)


def island(nim, population, generations=1, *, opponent, selective_pressure,
mut_prob, matches_1v1, evolve=True, display_matches=False,
display_survivals=False):

  for gen in range(generations):
    # Play
    for player in population:
      won_p = evaluate(nim,
                       n_matches=matches_1v1,
                       my_action=player.ply,
                       opponent_action=opponent)
      player.set_score(won_p)
      if display_matches:
        logging.info(f'Player {player.genome} has a win rate={player.score}')

    # Evolution

    best_population = [i for i in population if i.score > selective_pressure]
    offspring_size = len(population) - len(best_population)

    if display_survivals:
      logging.info(f'- Survivals = {len(best_population)}')

    if evolve:
      for i in range(offspring_size):
        p1 = tournament(best_population)
        p2 = tournament(best_population)
        o = p1.cross_over(p2, nim)       # XOVER
        if random.random() < mut_prob:   # MUT
          p1.mutation()
        best_population.append(o)
      population = best_population

    return population


def genetic_algorithm(nim: Nim, population, *, generations=100, matches_island=5,
matches_1v1=20, display_survivals=False):
  """
```

```
    The algorithm is based on islands.
    Each island have a different opponent to match with increasing difficulty.
    It goes from the dumb strategy to the god strategy (expert agent).
    There will be matches_island matches on each island and the genetic operations
    will be applyed after the end of the competition. The best individuals (the
ones
    that won more matches) will pass to the next generation, while the others will
perish.
    The offsprings will replace the missing individuals.
  """
  # I imported the opponents and the evaluation function from play_nim.py
  log_best1 = []
  log_best2 = []
  log_best3 = []
  log_best4 = []
  log_best5 = []
  selective_pressure = 0.5
  mut_prob = 0.01

  for i in tqdm(range(generations)):

    ####  ISLAND 1: population vs dumb agent  ####
    if display_survivals:
      logging.info(f'ISLAND 1: population vs dumb agent')

    survivals1 = island(nim, population, matches_island,
                        opponent=opponents[1],
                        selective_pressure=0.7,
                        mut_prob=mut_prob,
                        matches_1v1=matches_1v1,
                        display_survivals=display_survivals)
    log_best1.append((i, max([p for p in survivals1 if p.score>0], key=lambda i: -
i.score)))


    ####  ISLAND 2: population vs dumb random agent  ####
    if display_survivals:
      logging.info(f'ISLAND 2: population vs dumb random agent')

    survivals2 = island(nim, survivals1,  matches_island,
                        opponent=opponents[2],
                        selective_pressure=selective_pressure,
                        mut_prob=mut_prob,
                        matches_1v1=matches_1v1,
                        display_survivals=display_survivals)
    log_best2.append((i, max([p for p in survivals2 if p.score>0], key=lambda i: -
i.score)))

    ####  ISLAND 3: population vs dumb random agent  ####
    if display_survivals:
      logging.info(f'ISLAND 3: population vs random agent')

    survivals3 = island(nim, survivals2, matches_island,
                        opponent=opponents[3],
```

```
                             selective_pressure=selective_pressure,
                             mut_prob=mut_prob,
                             matches_1v1=matches_1v1,
                             display_survivals=display_survivals)
    log_best3.append((i, max([p for p in survivals3 if p.score>0], key=lambda i: -
i.score)))

    ####  ISLAND 4: population vs layered agent  ####
    if display_survivals:
      logging.info(f'ISLAND 4: population vs layered agent')

    survivals4 = island(nim, survivals3, matches_island,
                             opponent=opponents[4],
                             selective_pressure=selective_pressure,
                             mut_prob=mut_prob,
                             matches_1v1=matches_1v1,
                             display_survivals=display_survivals)
    log_best4.append((i, max([p for p in survivals4 if p.score>0], key=lambda i: -
i.score)))

    ####  ISLAND 5: population vs demigod agent  ####
    if display_survivals:
      logging.info(f'ISLAND 5: population vs demigod agent')

    survivals5 = island(nim, survivals4, matches_island,
                             opponent=opponents[5],
                             selective_pressure=selective_pressure,
                             mut_prob=mut_prob,
                             matches_1v1=matches_1v1,
                             display_survivals=display_survivals)
    log_best5.append((i, max([p for p in survivals5 if p.score>0], key=lambda i: -
i.score)))

    population = survivals5

  ####  ISLAND 6: population vs god agent  ####
  logging.info(f'ISLAND 6: population vs god agent')
  survivals6 = island(nim, survivals5,
                           opponent=opponents[6],
                           selective_pressure=0,
                           mut_prob=0,
                           matches_1v1=matches_1v1,
                           display_matches=False,
                           display_survivals=True,
                           evolve=False)
  defeated_god = [p for p in survivals6 if p.score > 0]
  if defeated_god:
    for champion in defeated_god:
      logging.info(f'CHAMPION {champion.genome} defeated GOD (score=
{champion.score})')
  else:
    logging.info(f'God won\n')
```

```
  log_best_generation = (log_best1, log_best2, log_best3, log_best4, log_best5)
  return population, log_best_generation


# ---
# Play
# ---

POPULATION_SIZE = 500

nim = Nim(7)
population = initial_population(POPULATION_SIZE, nim)

survivals, log_best = genetic_algorithm(nim, population,
                                        matches_island=50,
                                        matches_1v1=10,
                                        display_survivals=False)

parent_survivals = [p for p in survivals if p.score > 0]
logging.info('Survivals:')
for i in range(len(parent_survivals)):
  logging.info(f'- Player {parent_survivals[i].genome} with score
{parent_survivals[i].score}')

log1, log2, log3, log4, log5 = log_best
bests = []
bests.append(max(log1, key=lambda x: x[1].score))
bests.append(max(log2, key=lambda x: x[1].score))
bests.append(max(log3, key=lambda x: x[1].score))
bests.append(max(log4, key=lambda x: x[1].score))
bests.append(max(log5, key=lambda x: x[1].score))

logging.info('Best players:')
for i in range(5):
  logging.info(f'Island {i} - player {bests[i][1].genome} with score {bests[i]
[1].score} at generation {bests[i][0]}')
```

lab3_task3.ipynb

```


# ---
# Task3.3: An agent using minmax
# ---

import logging
import random
from copy import deepcopy
from nim import Nimply, Nim

logging.basicConfig(format="%(message)s", level=logging.INFO)
```

```
# ---
# Implementation
# ---

def hash_id(state: list, player: int):
  """
    Computes the hash of the tuple tuple(state) + (player, ), where:
    - state is the list of rows, i.e. the board
    - player is either 0 or 1
  """
  assert player == 1 or player == 0
  return hash(tuple(sorted(state)) + (player, ))

# ---

class Node():
  """
    State of the grapth that contains:
    - id: hash of tuple(state)+(player,)
    - state: copy of the state (nim._rows)
    - player: either 0 or 1
    - utility: value initialized to 0, becomes either -inf or +inf
    - children: list of nodes
    - parents: list of nodes
    - actions: list of possible actions as Nimply objects
  """

  def __init__(self, state: list, player: int):
    assert player == 1 or player == 0

    self.id = hash_id(state, player)
    self.state = deepcopy(state)
    self.player = player # Me (0) -> max ; Opponent (1) -> min

    self.utility = 0  # -inf if I lose, +inf if I win
    self.children = []
    self.parents = []
    self.possible_acitions() # creates self.actions


  def __eq__(self, other):
    return isinstance(other, Node) and self.state == other.state and self.player
== other.player


  def link_parent(self, parent):
    """
      Links the actual node with the parent node
    """
    assert isinstance(parent, Node)
    assert self.player != parent.player

    if parent not in self.parents:
```

```
      self.parents.append(parent)


  def link_child(self, child):
    """
      Links the child node to the actual node
    """
    assert isinstance(child, Node)
    assert self.player != child.player

    if child not in self.children:
      self.children.append(child)


  def is_leaf(self):
    return sum(self.state) == 0


  def leaf_utility(self):
    """
      Returns the utility of a leaf:
      - player 0 on leaf --> I lost, then utility = -inf
      - player 1 on leaf --> I won, then utility = +inf
    """
    if self.is_leaf():
      if self.player == 0:
        return float('-inf')     # I lost (the opponent took the last piece)
      else: return float('+inf') # I won


  def possible_acitions(self, k=None):
    """
      Computes all the possible action reachable from the actual node
      and saves them inside self.actions
    """
    self.actions = []

    if self.is_leaf():
      return

    not_zero_rows = [(r, n) for r, n in enumerate(self.state) if n > 0]
    for row, num_obj in not_zero_rows:
      while num_obj > 0:
        if k and num_obj > k:
          num_obj = k
          continue
        self.actions.append(Nimply(row, num_obj))
        num_obj -= 1

# ---

class GameTree():
  """
    Game Tree comosed of nodes that could have multiple parents and multiple
```

```
children.

    The roots is one:
    - Starting state + starting player = 0
    The leafs are two:
    - State of all zeros + finish player = 0  (I lose)
    - State of all zeros + finish player = 1  (I win)

    The class contains the following attributs:
    - k: nim._k
    - start_player: either 0 or 1
    - dict_id_node: dictionary that maps the node id to the actual node
    - dict_id_utility_action: dictionary that maps the node id to a tuple
(utility, action), where:
      - utility: utility of the node
      - action: better action to take (Nimply object)
    - root: root node (Node object)
  """

  def __init__(self, nim: Nim, start_player=0):
    self.k = nim._k
    self.start_player = start_player
    self.dict_id_node = {}
    self.dict_id_utility_action = {}

    self.root = Node(nim._rows, start_player)
    self.dict_id_node[self.root.id] = self.root


  def min_max(self):
    """
      MinMax using a recursive function that expands a node by trying every
possible action of that node.

      The recursive function returns the utility of the children and the parent
will select
       the best utility according to who is playing at that layer:
      - if player 1 is playing, than minimize the reward (look for utility = -inf)
      - if player 0 is playing, than maximize the reward (look for utility = +inf)

      The alpha-beta pruning is implemented:
       if the player finds a child with the desired utility, it stops looking
      becouse he will win choosing that action to go to that state.
    """

    def recursive_min_max(node: Node):
      # Stop condition
      if node.id in self.dict_id_utility_action:
        logging.debug(f'State {node.state} ({node.player}) already computed:
{self.dict_id_utility_action[node.id][0]}')
        return self.dict_id_utility_action[node.id][0] # just the utility value

      if node.is_leaf():
        node.utility = node.leaf_utility()
```

```
    logging.debug(f'Leaf player {node.player}')
    return node.utility


   # Recursive part
   for ply in node.actions:
    row, num_obj = ply

    # Check rules
    assert node.state[row] >= num_obj
    assert self.k is None or num_obj <= self.k

    # Create the child
    child_state = deepcopy(node.state)
    child_state[row] -= num_obj # nimming
    child_id = hash_id(child_state, 1 - node.player)
    if child_id in self.dict_id_node: # node already exists
     child = self.dict_id_node[child_id]
    else: # create the new node
     child = Node(child_state, 1 - node.player)

    # Link parent and child
    node.link_child(child)
    child.link_parent(node)

    # Recursion
    best_utility = recursive_min_max(child)

    # Update the values
    opp_wins = node.player == 1 and best_utility == float('-inf')  # opponent
will win
    i_win = node.player == 0 and best_utility == float('+inf')  # I will win
    if i_win or opp_wins:
     node.utility = best_utility
     self.dict_id_utility_action[node.id] = (node.utility, ply)
     return node.utility

   # This player will surelly lose otherwise he would have returned before
   node.utility = best_utility
   ply = random.choice(node.actions) # it doesn't matter the ply, he will lose
   self.dict_id_utility_action[node.id] = (node.utility, ply)

   return node.utility


  utility = recursive_min_max(self.root)
  if self.start_player == 0 and utility == float('+inf'):
   logging.info('The starting player will WIN')
   logging.info(f'--> move {self.dict_id_utility_action[self.root.id][1]}')
   return self.dict_id_utility_action[self.root.id]
  else:
   logging.info('The starting player will LOSE')
   return self.dict_id_utility_action[self.root.id]
```

```
    def best_action(self, node: Node):
      """
        Returns the best aciton at that state
      """
      assert self.root.id in self.dict_id_utility_action
      assert node.id in self.dict_id_utility_action

      return self.dict_id_utility_action[node.id]

# ---
# Play
# ---

nim = Nim(5)
game_tree0 = GameTree(nim, start_player=0) # I start
game_tree1 = GameTree(nim, start_player=1) # Opponent starts

game_tree0.min_max()
game_tree1.min_max()
```

lab3_task4.ipynb

```
# ---
# Task3.4: An agent using reinforcement learning
# ---

import logging
import random
from copy import deepcopy
import matplotlib.pyplot as plt
from nim import Nimply, Nim
from play_nim import opponents, evaluate

logging.basicConfig(format="%(message)s", level=logging.INFO)

# ---
# Implementation
# ---

def hash_id(state: list, player: int):
  """
    Computes the hash of the tuple tuple(state) + (player, ), where:
    - state is the list of rows, i.e. the board
    - player is either 0 or 1
  """
  assert player == 1 or player == 0
  return hash(tuple(state) + (player, ))
```

```
# ---
# Node Class from Task 3
# ---

class Node():
  """
    State of the grapth that contains:
    - id: hash of tuple(state)+(player,)
    - state: copy of the state (nim._rows)
    - player: either 0 or 1
    - reward: value initialized to 0, becomes either 2 (win) or -2 (lose)
    - children: list of nodes
    - parents: list of nodes
    - actions: list of possible actions as Nimply objects
  """

  def __init__(self, state: list, player: int):
    assert player == 1 or player == 0

    self.id = hash_id(state, player)
    self.state = deepcopy(state)
    self.player = player # Me (0) -> max ; Opponent (1) -> min

    self.reward = self.give_reward()
    self.children = []
    self.parents = []
    self.possible_acitions() # creates self.actions


  def __eq__(self, other):
    return isinstance(other, Node) and self.state == other.state and self.player
== other.player


  def link_parent(self, parent):
    """
      Links the actual node with the parent node
    """
    assert isinstance(parent, Node)
    assert self.player != parent.player

    if parent not in self.parents:
      self.parents.append(parent)


  def link_child(self, child):
    """
      Links the child node to the actual node
    """
    assert isinstance(child, Node)
    assert self.player != child.player

    if child not in self.children:
```

```
        self.children.append(child)


    def is_game_over(self):
      return sum(self.state) == 0


    def give_reward(self):
      """
      Returns the reward of the node
      - not end -> reward = -1
      - win -> reward = 2
      - lose -> reward = -2
      """
      if not self.is_game_over():
        #return -1
        return random.uniform(-1, 1)
      if self.player == 0: # I lose
        return -2
      return 2 # I win


    def possible_acitions(self, k=None):
      """
        Computes all the possible action reachable from the actual node
        and saves them inside self.actions
      """
      self.actions = []

      if self.is_game_over():
        return

      not_zero_rows = [(r, n) for r, n in enumerate(self.state) if n > 0]
      for row, num_obj in not_zero_rows:
        while num_obj > 0:
          if k and num_obj > k:
            num_obj = k
            continue
          self.actions.append(Nimply(row, num_obj))
          num_obj -= 1

# ---
# Game Tree (builded recursively, such us in task 3)
# ---

class GameTree():
  """
    Game Tree comosed of nodes that could have multiple parents and multiple
children.
    Differently from task 3, this class expands the tree considering both player 1
and player 0 starting.

    The roots are two:
    - Starting state + starting player = 0
```

```
    - Starting state + starting player = 1
    The leafs are two:
    - State of all zeros + finish player = 0  (Agent loses)
    - State of all zeros + finish player = 1  (Agent wins)

    The class contains the following attributs:
    - k: nim._k
    - dict_id_node: dictionary that maps the node id to the actual node
    - dict_id_reward: dictionary that maps the node id to the state reward
    - root0: root node (Node object) when player 0 starts
    - root1: root node (Node object) when player 1 starts

  """

  def __init__(self, nim: Nim):
    self.k = nim._k
    self.dict_id_node = {}
    self.dict_id_reward = {}

    # Build tree
    self.root0 = Node(nim._rows, player=0)
    self.root1 = Node(nim._rows, player=1)
    self.dict_id_node[self.root0.id] = self.root0
    self.dict_id_node[self.root1.id] = self.root1

    logging.info(f'Building the tree...')
    self.build_tree(self.root0)
    self.build_tree(self.root1)
    logging.info('Done')


  def build_tree(self, root):
    """
      Builds the tree using a recursive function that expands a node by trying
every possible action of that node.

      The nodes are likend to each other, starting by the given root node.
    """

    def recursive(node: Node):
      # Stop condition
      if node.id in self.dict_id_reward:
        return

      if node.is_game_over():
        node.reward = node.give_reward()
        self.dict_id_reward[node.id] = node.reward
        return


      # Recursive part
      for ply in node.actions:
        row, num_obj = ply
```

```
          # Check rules
          assert node.state[row] >= num_obj
          assert self.k is None or num_obj <= self.k

          # Create the child
          child_state = deepcopy(node.state)
          child_state[row] -= num_obj # nimming
          child_id = hash_id(child_state, 1 - node.player)
          if child_id in self.dict_id_node: # node already exists
            child = self.dict_id_node[child_id]
          else: # create the new node
            child = Node(child_state, 1 - node.player)
            self.dict_id_node[child_id] = child

          # Link parent and child
          node.link_child(child)
          child.link_parent(node)

          # Recursion
          recursive(child)

        # Reward of the node (-1)
        node.reward = node.give_reward()
        self.dict_id_reward[node.id] = node.reward

        return

      recursive(root)
      root.reward = root.give_reward()

# ---
# Agent
# ---

class Agent():
  """
    The agent that will use Reinforcement Learning to learn to play nim.
    This class in based on the maze example given by the professor.

    Attributes:
    alpha: learning rate
    random_factor: probability of making a random action
    state_history: history of the match played before learning
    G: dictionary that maps the id node to the expected reward (initialized
randomly)
  """

  def __init__(self, game_tree: GameTree, alpha=0.5, random_factor=0.2):
    self.alpha = alpha
    self.random_factor = random_factor

    self.state_history = [] # node -> inside has state and reward
    self.G = {} # (k, v) = id_node, expected reward
    for id, node in game_tree.dict_id_node.items():
```

```
        self.G[id] = random.uniform(1.0, 0.1)


  def choose_action(self, node: Node):
    """
      Returns a Nimply by choosing the move that gives the maximum reward.
      With self.random_factor probability returns a random move.
    """
    maxG = -10e15
    next_move = None

    # Random action
    if random.random() < self.random_factor:
      next_move = random.choice(node.actions)
    # Action with highest G (reward)
    else:
      for a in node.actions:  # a is a Nimply obj
        new_state = deepcopy(node.state)
        new_state[a.row] -= a.num_objects
        new_state_id = hash_id(new_state, player=1) # opponent's state
        if self.G[new_state_id] >= maxG:
          next_move = a
          maxG = self.G[new_state_id]

    return next_move


  def update_history(self, node: Node):
    self.state_history.append(node)


  def learn(self):
    """
      Update the internal G function by looking at the past
       using the formula G[s] = G[s] + a * (v - G[s]), where:
       - G[s] is the expected reward for the state s
       - a is alpha, the learning rate
       - v is the actual value associated to that state

      After the learning part, it reset the history and decreases the random
factor by 10e-5
    """
    target = 0

    for node in reversed(self.state_history):
      self.G[node.id] = self.G[node.id] + self.alpha * (target - self.G[node.id])
      target += node.reward
      #print(f'player {node.player}: {node.state} - {self.G[node.id]}')

    self.state_history = []      # Restart
    self.random_factor -= 10e-5 # Decrease random factor each episode of play

# ---
# Evaluatation function
```

```
# ---

def play(nim: Nim, game_tree: GameTree, agent: Agent, n_matches=40, *,
opponent_action: callable, alternate_turns=True):
  """
    This function simulated n_matches games between the agent and the given
opponent.

    If alternate_turns == True, than the games will have as strating player the
player 0
     50 % of the time and player 1 50% of the time.

    If alternate_turns == False, the player 0 will always start the match.
  """

  # Agent is player 0
  won = 0
  for m in range(n_matches):
    # Setup the match
    nim_tmp = deepcopy(nim)

    if alternate_turns:
      if m/n_matches > 0.5:
        player = 1 # The agent starts
      else:
        player = 0 # Opponent starts
    else:
      player = 0 # The agent always starts

    # Play the match
    while not nim_tmp.is_game_over():
      player = 1 - player
      if player == 1:
        ply = opponent_action(nim_tmp)
      else: # player 0
        state_id = hash_id(nim_tmp._rows, player)
        node = game_tree.dict_id_node[state_id]
        ply = agent.choose_action(node)
      nim_tmp.nimming(ply)

    if player == 0:
      won += 1

  return won/n_matches

# ---
# Reinforcement Learning algorithm
# ---

def RL_nim(nim: Nim, game_tree: GameTree, agent: Agent, opponent: callable,
episodes = 5000, alternate_turns=True):
  """
    A match is played by the agent aganst the given opponent episodes times.
```

```
    At each episode, the agent learns looking at the rewards that it receved.

    Every 50 epochs, the agent plays versus the opponent using the function
play(•) above,
    in order to look how many matches it wins.

    The win rates are sotred inside a log list that is returned

  """
  log_winrate = [] # episode, value

  for e in range(episodes):
    # Play a game
    episode_nim = deepcopy(nim)

    if alternate_turns:
      if e % 2 == 0:
        state = game_tree.root0 # the agent starts
      else:
        state = game_tree.root1 # the opponent starts
    else:
      state = game_tree.root0 # The agent always starts

    agent.update_history(state)
    while not episode_nim.is_game_over():
      # My turn
      if state.player == 0:
        my_action = agent.choose_action(state) # Choose an action
        episode_nim.nimming(my_action)         # Update the state
        new_state_id = hash_id(episode_nim._rows, player = 1)
        state = game_tree.dict_id_node[new_state_id]

      # Opponent turn
      else:
        opp_action = opponent(episode_nim)
        episode_nim.nimming(opp_action)
        new_state_id = hash_id(episode_nim._rows, player = 0)
        state = game_tree.dict_id_node[new_state_id]

      agent.update_history(state)

    if state.player == 0:
      agent.update_history(state)
    agent.learn()

    # Log
    if e % 50 == 0:
      winrate = play(nim, game_tree, agent, opponent_action=opponent,
alternate_turns=alternate_turns)
      logging.debug(f'{e}: winrate = {winrate}')
      log_winrate.append((e, winrate))

  return log_winrate
```

```
# ---
# Plot funcion
# ---

def plot_agent_winrates(log_winrate):
    x = [e for e, w in log_winrate]
    y = [w for e, w in log_winrate]
    plt.xlabel('Episodes')
    plt.ylabel('Win rate')
    plt.plot(x, y)
    plt.show()

# ---
# Play
# ---

nim = Nim(5)
game_tree= GameTree(nim)
agent = Agent(game_tree)

# ---
# Agent vs Dumb player
# ---
opponent = opponents[1]
log_lv1 = RL_nim(nim, game_tree, agent, opponent=opponent, episodes=10000,
alternate_turns=False)

logging.info(f'Agent winrate from {log_lv1[0]} to {log_lv1[len(log_lv1)-1]} ')
vals = [val for _, val in log_lv1]
logging.info(f'Avg score = {sum(vals) / len(vals)}')
plot_agent_winrates(log_lv1)

# Agent winrate from (0, 0.825) to (9950, 1.0)
# Avg score = 0.993125

# ---
# Agent vs Dumb random player
# ---
opponent = opponents[2]
log_lv2 = RL_nim(nim, game_tree, agent, opponent=opponent, episodes=10000,
alternate_turns=False)

logging.info(f'Agent winrate from {log_lv2[0]} to {log_lv2[len(log_lv2)-1]} ')
vals = [val for _, val in log_lv2]
logging.info(f'Avg score = {sum(vals) / len(vals)}')
plot_agent_winrates(log_lv2)

# Agent winrate from (0, 0.9) to (9950, 1.0)
# Avg score = 0.9764999999999988

# ---
# Agent vs Random player
# ---
```

```
opponent = opponents[3]
log_lv3 = RL_nim(nim, game_tree, agent, opponent=opponent, episodes=10000,
alternate_turns=False)

logging.info(f'Agent winrate from {log_lv3[0]} to {log_lv3[len(log_lv3)-1]} ')
vals = [val for _, val in log_lv3]
logging.info(f'Avg score = {sum(vals) / len(vals)}')
plot_agent_winrates(log_lv3)

# Agent winrate from (0, 0.95) to (9950, 0.975)
# Avg score = 0.9664999999999988



# ---
# Agent vs Layered player
# ---
opponent = opponents[4]
log_lv4 = RL_nim(nim, game_tree, agent, opponent=opponent, episodes=10000,
alternate_turns=False)

logging.info(f'Agent winrate from {log_lv4[0]} to {log_lv4[len(log_lv4)-1]} ')
vals = [val for _, val in log_lv4]
logging.info(f'Avg score = {sum(vals) / len(vals)}')
plot_agent_winrates(log_lv4)

# Agent winrate from (0, 1.0) to (9950, 1.0)
# Avg score = 0.9996250000000001

# ---
# Agent vs Demigod player (50% random - 50% num-sum)
# ---

opponent = opponents[5]
log_lv5 = RL_nim(nim, game_tree, agent, opponent=opponent, episodes=10000,
alternate_turns=False)

logging.info(f'Agent winrate from {log_lv5[0]} to {log_lv5[len(log_lv5)-1]} ')
vals = [val for _, val in log_lv5]
logging.info(f'Avg score = {sum(vals) / len(vals)}')
plot_agent_winrates(log_lv5)

# Agent winrate from (0, 0.775) to (9950, 0.825)
# Avg score = 0.7732500000000002

# ---
# Agent vs God player (nim-sum)
# ---

opponent = opponents[6]
log_lv6 = RL_nim(nim, game_tree, agent, opponent=opponent, episodes=10000,
alternate_turns=False)

logging.info(f'Agent winrate from {log_lv6[0]} to {log_lv6[len(log_lv6)-1]} ')
vals = [val for _, val in log_lv6]
```

```
logging.info(f'Avg score = {sum(vals) / len(vals)}')
plot_agent_winrates(log_lv6)

# Agent winrate from (0, 0.0) to (9950, 0.0)
# Avg score = 0.0
```

My README

Lab 3: Policy Search

Task

Write agents able to play *Nim*, with an arbitrary number of rows and an upper bound k on the number of objects that can be removed in a turn (a.k.a., *subtraction game*).

The player **taking the last object wins**.

- Task3.1: An agent using fixed rules based on *nim-sum* (i.e., an *expert system*)
- Task3.2: An agent using evolved rules
- Task3.3: An agent using minmax
- Task3.4: An agent using reinforcement learning

Instructions

- Create the directory `lab3` inside the course repo
- Put a `README.md` and your solution (all the files, code and auxiliary data if needed)

Notes

- Working in group is not only allowed, but recommended (see: Ubuntu and Cooperative Learning). Collaborations must be explicitly declared in the `README.md`.
- Yanking from the internet is allowed, but sources must be explicitly declared in the `README.md`.

Deadline

- Tasks 3.1 and 3.2 --> 4/12
- Tasks 3.3 and 3.4 --> 11/12

Note

The documentation is inside every function. The readme gives just the high level idea of the strategy, while the functions contains an in depth documentation of what they do!

Collaborations:

- Paolo Drago Leon

> ## Task 3.1 - An agent using fixed rules based on nim-sum
>
> The agent uses an expert strategy based on nim-sum. If the agent is on a position with not zero nim-sum, then he will always win.
>
> Based on the explanation available here: https://en.wikipedia.org/wiki/Nim
>
> ## Task 3.2 - An agent using evolved rules
>
> The agent uses evolved hard-coded rules. Those rules are labeled with a number from 1 to 7. The order of the rules represents the genome of the agent.
>
> I used a hyerarcical island strategy to let evolve the initial population. Only the best individual surviving at each island where able to reproduce.
>
> The opponents where the following, ordered from the lowest island to the highest:
>
> - Dumb opponent: always takes one obj from the first row available.
> - Dumb random opponent: there is 0.5 of probability to make a dumb action or a random action.
> - Random opponent: the opponent performs a random action.
> - Layered opponent: it always takes the whole row's objs choosing randomly the row.
> - Demigod opponent: there is a probability to play an expert move and a chance to play randomly
> - God opponent: it is the expert agent developed at task3.1, the one that uses nim-sum.
>
> ## Task 3.3 - An agent using minmax
>
> The agent uses the MinMax strategy applyed on the game tree, a tree with all the possible states of nim.
>
> I used the alpha-best pruning in order to reduce the time complexity of the function. The nodes of the tree stops expanding if there is at least a child already expanded that has the best result expected for the player at that layer.
>
> > If the layer is played by the opponent and a child has -inf utility, than the opponent will stop expanding the current state because it knows that it will win with that move. The same if for the layer played by the agent: if there is at least a child with +inf utility, than stop expanding the subtree.
>
> ## Task 3.4 - An agent using reinforcement learning
>
> The solution is inspired by the maze example given by the professor.
>
> The states are encoded inside a game tree similar to the one of the previous task and the reward is set to:
>
> - 100 if the agent wins
> - -100 if the agent loses
> - -1 if the state is not a win or lose state

Review 1

- Review to Paolo Drago Leon

> Overall you did a splendid job! So **well done**! 👏
>
> # Task 1 - Expert agent
>
> ### If a row has more than k objects, you force a move without looking at nim-sum
>
> At the beginning of the expert_strategy(•) you verify if there is a row with a number of objects grater than k. If so, than you check:
>
> - `if board.rows[i] % (board.k + 1) != 0`, you choose `Nimply(i, board.rows[i] % (board.k + 1))` as ply.
> - if `board.rows[i] % (board.k + 1) == 0`, you just pick k objects from the maximum row.
>
> The problem is that **you actually choose a move that is not optimal!** Look at this example:
>
> > k = 3 After some moves we have: `[0, 3, 5, 0, 0]` --> nim-sum = 6 Row 2 has num_objects > k, so you apply the first part of the strategy:
> >
> > ```
> > if board.rows[i] % (board.k + 1) != 0:
> >     ply = Nimply(i, board.rows[i] % board.k + 1))
> > ```
> >
> > That means:
> >
> > ```
> > if 5 % (3 + 1) != 0: # True bc 5 % 4 = 1
> >     ply = Nimply(2, 1)
> > ```
> >
> > So your next state will be:
> > `[0, 3, 4, 0, 0]` --> nim-sum = 7 !!!
> >
> > Instead, the best move would have been this: `Nimply(2, 2)` -> `[0, 3, 3, 0, 0]` --> nim-sum=0
>
> So the thing is your expert agent will do a not optimal move becouse it doesn't check nim-sum first.
>
> The same happens if you find a row with num_objects > k but `board.rows[i] % (board.k + 1)` is always False, than you apply a 'default' rule: `Nimply(max_row, board.k)
>
> Here is an example:
>
> > k = 4 After some point we have: `[0, 3, 5, 0, 0 ]` --> nim-sum=6 `board.rows[i] % (board.k + 1)` = 5 % (4+1) = 0 --> False
> > but `one_gt_k is True` (you found a row with num_objs > k)
> >
> > Than you apply the 'defaul' rule, so your ply will be `Nimply(2, 4)` because k=4 The next state will be `[0, 3, 1, 0, 0]` --> nim-sum = 2 !!!

> Using the nim-sum approach you would have played `Nimply(2, 2)` Next state -> `[0, 3, 3, 0 , 0]` --> nim-sum = 0

You could verify that the limit on k is respected within the nim-sum algorithm and if you don't find any better move, than apply this strategy.

Redundant actions that the strategy would have covered

In the second part of the algorithm, after this line

```
board_nimsum = calc_nimsum(board.rows)
```

you handle the cases where the board has (or you can force the board to have) at most one object at each row.

This is not an issue per se, **it is just redundant** becouse the nim-sum algorithm would have covered this case by making you choose the best move (the same one you choose here).

The code would be less loaded and the computational complexity would not change because you iterate over the entire array in the `all_ones()` function and with the `functools.reduce()` function.

Task 2 - Evolved agent

The strategy is well thought out and from the results I have to say that it works better than mine. I haven't found any problems in the code, except for two little things

SemiExpert B agent

Here you don't take into account the ending strategies but the nim-sum algorithm covers that case so **this 'semi-expert' agent is actually an expert agent!**

Rule 5 inside class Rules

This rule does nothing becouse of that `return None` before the condition check. I think it was intentional becouse you were using the expert strategy (nim-sum) as rule inside the individual, but leaving that will steale clock cycles. Not a big issue, but cleaning it could save some time (maybe).

Task 3 - MinMax

Nothing to say about this task, it seems to be by the book!

Task 4 - Reinforcement Learning

> Here again it is a good work! your plots have really nice shapes and the results ara really good. Good job again!

NOTE: The cose is really long and verbose and since my review should be quite detailed and self-explanatory, I don't report Paolo's code hoping it won't be a problem.

Review 2

- Review to Flavio Patti

> # Task 1 - Expert player using nim-sum strategy
>
> Here you have not implemented what was asked, but instead you implemented an hard coded rule-based agent.
>
> The thing is that **this agent will always lose against an expert agent that uses the nim-sum strategy** becouse your code just looks for a winning move only if you are in an final state that your rules can handle.
>
> Furthermore, your agent is very computational expencive since it's a mix between a recursive function, that goes on until it finds a good move, and 10 entire iterations over the state board.
>
> The nim-sum strategy only does 2 iterations over the state board and will always force the opponent in a state where he can't win; moreover it is easy to implement and to read.
>
> # Task 2 - Evolved agent
>
> You kinda implemented the optimal strategy inside task3.2.ipynb but trying all the possible states without using the real strategy to look for the only row that can give you nim-sum = 0. Again, following the algorithm that you can find online, you can have a more performant agent.
>
> Your evolved agent uses just 4 hard-coded rules that are very generic. Without other actions, your agent will always perform poorly. You could have used some of the hard-coded rules that you implemented in task3.1. Maybe you could have added a gamma variable in order to use 9 hard-coded rules. This could be beneficial for your agent.
>
> Maybe it was intetional, but the ifs inside the function that chooses which ply the agent will perform are not mutually exclusive according to alpha and beta.

```
def evolvable(state: Nim, genome: tuple):
    threshold_alpha = 0.5
    threshold_beta = 0.5

    #choose the strategy to use based on the parameters inside the genome
    if threshold_alpha <= genome[0] and threshold_beta <= genome[1]:
        ply = dumb_PCI_max_longest(state)
    if threshold_alpha <= genome[0] and threshold_beta >= genome[1]:
```

```
        ply = dump_PCI_min_longest(state)
    if threshold_alpha >= genome[0] and threshold_beta <= genome[1]:
        ply = dump_PCI_max_lowest(state)
    if threshold_alpha >= genome[0] and threshold_beta >= genome[1]:
        ply = dumb_PCI_min_lowest(state)

    return ply
```

If you have alpha and/or beta equal to 0.5, than the first rules will always be replaced by the ones below. If it was not willful, be aware that this could change the expected behaviour of you agent.

Task 3 - MinMax agent

You adopt a 'deep pruning' combined with alpha-beta pruning in order to cut off the search and reduce the computational complexity.

An approach that you did not considered was to not consider superimposable states. Here is an example:

> `[1, 0, 3, 1, 2]` The above state will have the same outcome of the following states:
>
> - `[1, 0, 3, 2, 1]`
> - `[3, 1, 2, 0 , 1]`
> - `[2, 1, 3, 0, 1]`
> - etc.. with every possible combination of those rows
>
> All the states can be represented by a single state that has a sorted number of objects: `[3, 2, 1, 1, 0]`.

So, the idea is to map the actual state with his 'ordered version' in and take the result inside a dictionary. Doing so, you can avoid using the deep pruning and you can expand the tree to actually verify if the agent wins or not. With this strategy I runend MinMax on Nim with 8 rows and it took only 10 min (without deep pruning).

Task 4 - LR agent

Nothing to say here, it looks a good implementation. The only think that i could suggest is to train the RL agent with more than one opponent in a gradual way. First against an easy-to-beat agent and than gradually towards the expert. It should make more robust the learning of the agent.

NOTE: Same here, the cose is really long and verbose and since my review should be quite detailed and self-explanatory, I don't report Flavio's code hoping it won't be a problem.