Table of Contents

- Table of Contents
- Creating an Agent able to play the game of QUARTO
  - Introduction
    - What is QUARTO?
  - Analyzing the problem
- Proposed Solutions
  - Dumb Player
  - GA approach
    - Results
  - Rule Based Player
    - Results
  - MinMaxPlayer
  - RL Agent
    - Results
  - Mixing strategies
    - Results
  - Mixed strategies with RL policy
- My best player (S309413)
  - Results
- Lab 1 : Set Covering Problem
  - Task
  - Instructions
  - Notes
- Analyzing the Problem
- My proposed solution
  - Reference and material
  - Greedy approach
  - The core of my algorithm
  - Results
    - Using a value of 1 for $threshold$ I obtained the following results:
    - Using a value of 0.5 for $threshold$ I obtained the following results:
    - Testing for different values of threshold
  - Updates after review
  - Peer reviewing
- Lab 2 : Set Covering Problem solved by GA
  - Task
- Analyzing the Problem
  - My proposed solution
    - Reference and material
  - early stages of the solution
  - Genetic Algorithm
  - Offspring generation
  - Fitness function
  - Results
  - Peer reviewing
- Lab 3: Policy Search
  - Task
  - Instructions
  - Notes
  - Deadlines (AoE)
  - Task 3.1
    - An agent using fixed rules based on *nim-sum* (i.e., an *expert system*)
  - Task 3.2
    - An agent using evolved rules
      - Core idea

        - Genome and representation
        - Mutations
        - Evaluations
        - Results and minor tweaks
    - Task 3.3
        - An agent using minmax
    - Task 3.3
        - An agent using reinforcement learning
    - Peer reviewing
- Code for Lab 1
    - setcovering solution
    - dijkstra solution implemented after the peer reviews
- Code for Lab 2
- Code for Lab 3
    - Task 1 (plus other opponents)
    - Task 2 (GA)
    - Task 3 (minmax)
    - Task 4 (RL)
        - Training
        - Agent
    - Best players
- Code for Final Project (Quarto)
    - players
    - GA Agent
    - RL Agent
    - utility functions
    - My best player (S309413)

Creating an Agent able to play the game of QUARTO

Introduction

What is QUARTO?

The game of quarto consist of a board with **16 squares** and **16 pieces**. Each piece has **4 attributes** (height, color, shape, and filling) and each attribute can be either **big** or **small**, **light** or **dark** , **squared** or **round** , **full** or **hollow**.
The goal is to establish a line of four pieces ith at least one common characteristic (attribute) on the board.
Quarto is a turn based. impartial game.
The game start with *player1* choosing a piece to place on the board, then *player2* chooses where to place it and choose an ather piece to be placed on the board. The game continues until one of the players has established a line of four pieces with at least one common attribute.

Analyzing the problem

The first step that i took to search for a solution was to create a playable version of the player in order to sense the problem and to be able to test possible strategies by playing aganinst the random player already created.
The code for this player simply take in input a value from **0** to **15** representing the choosen piece, and **2** values from **0** to **2** representing the row and the column where the piece will be placed.

```
class HumanPlayer(quarto.Player):
    """Random player"""

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

    def choose_piece(self) -> int:
        status = self.get_game().get_board_status()
        used_pieces = {c for x in status for c in x if c > -1}
        usable_pieces = [_ for _ in {x for x in range(16)} - used_pieces]
        print(self.get_game().print())
        print(f"Choose a piece: {usable_pieces}")
        piece = int(input())
        while piece not in usable_pieces:
            print("Invalid piece")
            print(f"Choose a piece in : {usable_pieces}")

        return piece

    def place_piece(self) -> tuple[int, int]:
        status = self.get_game().get_board_status()
        possible_moves = [(c, r) for r in range(4) for c in range(4) if status[r][c] == -1]
        print(self.get_game().print())
        print(f"Choose a move: {possible_moves}")
        move = tuple(map(int, input().split()))
        while move not in possible_moves:
            print("Invalid move")
            print(f"Choose a move in : {possible_moves}")
        return move
```

After playng for some times i started to come up with different options for a strategy.

Proposed Solutions

All of my attempt at creating a wiining player can be found in platyers.py but now I will try to briefly explain the main idea behind each of them.

Dumb Player

This player is really simple but it was really usefull especially in the beggining phases of the progect because it allowed me to have an easbly beatable player to test my strategies againsT, particolarly when i was trying to implement a GA approach.

```
class DumbPlayer(quarto.Player):
    """Dumb player"""

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

    def choose_piece(self) -> int:
        status = self.get_game().get_board_status()
        used_pieces = {c for x in status for c in x if c > -1}
        usable_pieces = [_ for _ in {x for x in range(16)} - used_pieces]
        return usable_pieces[0]

    def place_piece(self) -> tuple[int, int]:
        status = self.get_game().get_board_status()
        possible_moves = [(c, r) for r in range(4) for c in range(4) if status[r][c] == -1]
        return possible_moves[0]
```

When prompted to choose a piece, the player simply choose the first piece that is not already on the board. When prompted to place a piece, the player simply choose the first empty square on the board.

GA approach

The first real attempt at finding a winning strategy was to implement a Genetic Algorithm. Since i had great results with the GA implemented for the third Lab, i decided to try to use it to solve this problem.
The genome of the individuals consists of 5 values :

```
genome = {
            "alpha": -12.228556866040943,
            "beta": 77.44867497850886,
            "gamma": 7.190937752438881,
            "delta": 10.629790618213386,
            "epsilon": 34.46747782969669
        }
```

This parameters are Then used to decide which piece to choose and where to place it on the board. *alpha* and *beta* are used to decide which piece to choose, while *gamma*, *delta* and *epsilon* are used to decide where to place it.
To calculate the piece:

```
def choose_piece(self) -> int:
        data = cook_data_pieces(self)
        res = (
            (a[0], abs(self.genome["alpha"] * a[1] + self.genome["beta"] * b[1]))
            for a, b in zip(data["alpha"], data["beta"])
        )
        choosen_piece = min(res, key=lambda x: x[1])[0]
        return choosen_piece
```

where `data` is a dictionary calculated in the function `cook_data_pieces` where each **key** represents the result of a bitwise operation calculated on the available pieces. For each available piece i remove it from the available piece as if the player decided to choose it, then i calculate the sum of the remaining pieces and the bitwise nand of the remaining pieces. The result of this operation is then used as a **key** in the dictionary `data`. The value associated with each key is a list of tuples where each tuple contains the piece and the result of the operation. The piece is then chosen by taking the piece with the lowest value of the operation:

```
def cook_data_pieces(state: quarto.Quarto) -> dict:
    """provide usefull data for the genetic algorithm whe need to choose a piece"""
    data = {}
    status = state.get_game().get_board_status()
    used_pieces = {c for x in status for c in x if c > -1}
    usable_pieces = {x for x in range(16)} - used_pieces
    alpha = list()
    beta = list()
    if len(usable_pieces) == 1:
        # if there is only one piece left, it is the best choice
        return {"alpha": [(list(usable_pieces)[0], 1)], "beta": [(list(usable_pieces)[0], 1)]}
    for p in usable_pieces:
        usable_pieces.remove(p)
        # remove the selected piece from the usable pieces
        # calculate the sum of the remaining pieces
        *_, a = accumulate([_ for _ in usable_pieces], operator.add)
        usable_pieces.add(p)
        used_pieces.add(p)
        # add the selected piece to the used pieces
        # calculate the nand of the used pieces
        *_, b = accumulate(used_pieces, lambda x, y: ~(x & y))
        used_pieces.remove(p)
        alpha.append((p, a))
        beta.append((p, b))
    data["alpha"] = alpha
    data["beta"] = beta
    return data
```

To calculate the move:

```
def place_piece(self) -> tuple[int, int]:
        data = cook_data_moves(self, self.get_game().get_selected_piece())
        res = (
            (g[0], abs(self.genome["gamma"] * g[1] + self.genome["delta"] * h[1] + self.genome["epsilon"] * i[1]))
            for g, h, i in zip(data["gamma"], data["delta"], data["epsilon"])
        )
        choosen_move = min(res, key=lambda x: x[1])[0]
        return choosen_move
```

In the same way as before `data` is a dictionary calculated in the function `cook_data_moves` where each **key** represents the result of a bitwise operation calculated on the available moves on the board. For each available move i remove it from the available moves as if the player decided to place the piece there, then i calculate the sum of the remaining moves, the bitwise and of the remaining moves and the bitwise or of the remaining moves. The result of this operation is then used as a **key** in the dictionary `data`. The value associated with each key is a list of tuples where each tuple contains the move and the result of the operation. The move is then chosen by taking the move with the lowest value of the operation :

```python
def cook_data_moves(state: quarto.Quarto, piece: int) -> dict:
    """provide usefull data for the genetic algorithm whe need to choose a move"""
    data = {}
    status = state.get_game().get_board_status()
    possible_moves = [(c, r) for r in range(4) for c in range(4) if status[r][c] == -1]
    gamma = list()
    delta = list()
    epsilon = list()
    if len(possible_moves) == 1:
        # if there is only one move left, it is the best choice
        return {
            "gamma": [(possible_moves[0], 1)],
            "delta": [(possible_moves[0], 1)],
            "epsilon": [(possible_moves[0], 1)],
        }
    for p in possible_moves:
        # remove the selected move from the possible moves
        possible_moves.remove(p)
        # calculate the sum of the remaining moves
        # calculate the or of the remaining moves
        # calculate the and of the remaining moves
        *_, g = accumulate([x[0] for x in possible_moves], operator.add)
        *_, d = accumulate([x[1] for x in possible_moves], operator.or_)
        *_, e = accumulate([x[0] for x in possible_moves], operator.and_)
        possible_moves.append(p)
        gamma.append((p, g))
        delta.append((p, d))
        epsilon.append((p, e))
    data["gamma"] = gamma
    data["delta"] = delta
    data["epsilon"] = epsilon

    return data
```

In order to evolve and explore different individuals the algorith is based on the following steps:

1. Generate a population of `POPULATION` individuals
2. Evaluate the fitness of each individual
3. select with a tournament of size `k = 5` an individual from the population
4. mutate the individual's genome
5. calculate the fitness of the mutated individual
6. repeat step 3 to 5 until `OFFSPRING` individuals are generated
7. select the best `POPULATION` individuals from the population and the offspring
8. repeat step 2 to 7 until the maximum number of generations is reached

All the code for the GA can be found in the file GAAgent.py but here is a summary of the main functions:

Step 1:

```python
def generate_population(dim: int) -> list:
    r = []
    for _ in range(dim):
        genome = {
            "alpha": random.uniform(-10, 10),
            "beta": random.uniform(-10, 10),
            "gamma": random.uniform(-10, 10),
            "delta": random.uniform(-10, 10),
            "epsilon": random.uniform(-10, 10),
        }
        fit = fitness(genome)
        r.append((fit, genome))
    return r
```

the genome is generate based on a random uniform distribution between -10 and 10

Step 2:

```python
def fitness(genome):
    game = quarto.Quarto()
    agent = GAPlayer(game)
    agent.set_genome(genome)

    opponent = RandomPlayer(game)
    random_eval = evaluate(game, agent, opponent, NUM_MATCHES)
    game.reset()

    opponent = DumbPlayer(game)
    dumb_eval = evaluate(game, agent, opponent, NUM_MATCHES)
    game.reset()

    opponent = RuleBasedPlayer(game)
    rule_eval = evaluate(game, agent, opponent, NUM_MATCHES)

    return (rule_eval, random_eval, dumb_eval)
```

The fitness is calculated by playing `NUM_MATCHES` against three different opponents: a random player, a dumb player and a rule based player. The fitness is then calculated as the average of the number of wins against each opponent. The fitness is then a tuple of three values where the first value is the fitness against the rule based player, the second value is the fitness against the random player and the third value is the fitness against the dumb player.
For each evaluation against an opponent half of the games are played as *player1* and the other half as *player2* in order to avoid bias.

Step 3:

```python
def tournament(population, tournament_size=5):
    return max(random.choices(population, k=tournament_size), key=lambda i: i[0])
```

Step 4 & 5 & 6:

```python
def generate_offspring(population: list, gen: int) -> list:
    offspring = list()
    for _ in range(OFFSPRING):
        p = tournament(population)

        p[1]["alpha"] += random.gauss(0, 20 / (gen + 1))
        p[1]["beta"] += random.gauss(0, 20 / (gen + 1))
        p[1]["gamma"] += random.gauss(0, 20 / (gen + 1))
        p[1]["delta"] += random.gauss(0, 20 / (gen + 1))
        p[1]["epsilon"] += random.gauss(0, 20 / (gen + 1))
        fit = fitness(p[1])
        offspring.append((fit, p[1]))

    return offspring
```

The genome is mutated by adding a random value sampled from a gaussian distribution with mean 0 and standard deviation 20/(gen+1) where gen is the current generation. The mutation is applied to each gene of the genome.

Step 7

```python
def combine(population, offspring):
    population += offspring
    population = sorted(population, key=lambda i: i[0], reverse=True)[:POPULATION]
    return population
```

Step 8:

```python
def GA():
    i = 0
    best_sol = None
    logging.info("Starting GA")
    if IMPORT_POPULATION:
        with open(POPULATION_FILE, "r") as f:
            pop = json.load(f)
            population = [(fitness(p["genome"]), p["genome"]) for p in pop.values()]
            logging.info(f"population imported")
    else:
        logging.info(f"generating population of {POPULATION} individuals")
        population = generate_population(POPULATION)
        logging.info(f"population generated")

    for _ in range(NUM_GENERATIONS):
        logging.info(f"Generation {_}")
        offspring = generate_offspring(population, _)
        population = combine(population, offspring)
        logging.debug(f"best genome: {population[0][1]}")
        logging.debug(f"best fitness: {population[0][0]}")
        if (_ + 1) % LOG_FREQ == 0:
            with open(f"populations/population_GA_v{i}.json", "w") as f:
                pop = {f"individual_{i:02}": {"fitness": p[0], "genome": p[1]} for i, p in enumerate(population)}
                json.dump(pop, f, indent=4)
                logging.info(f"saved population")
            i += 1

    best_sol = population[0][1]
    return best_sol
```

This is the core of the GA. In order to save particularly instresting genome and their fitness the population is saved every `LOG_FREQ` generations in a json file.

Results

This attempt unfortunatly was not successful. The best genome found was:

```
"genome": {
            "alpha": -94.03014146974122,
            "beta": -107.17350875193313,
            "gamma": 152.6577141347451,
            "delta": -29.856838596915765,
            "epsilon": -12.095960806170313
        }
```

But It yelded pretty poor results against every player except the most simple one (the dumb player). Against the `RandomPlayer` the agent had a winrate between 50 and 60 and against the `RuleBasedPlayer` the winrate was between 20 and 30. The agent was able to beat the `DumbPlayer` with a winrate between 90 and 100.
This are the result of **1000** matches against each player:

	RuleBasedPlayer	RandomPlayer	DumbPlayer
TrainedGAPlayer	282	580	1000

Rule Based Player

The player uses a set of hardcoded rules to decide which piece to play and where to play it.

```
class RuleBasedPlayer(quarto.Player):
    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.previous_board = np.array([[-1, -1, -1, -1], [-1, -1, -1, -1], [-1, -1, -1, -1], [-1, -1, -1, -1]])
        self.previous_move = None
        self.previous_piece = None

    def choose_piece(self) -> int:
        pieces = block_strategy_piece(self)
        if len(pieces) < 3 and len(pieces) > 0:
            return list(pieces.keys())[0]
        piece = mirror_strategy_piece(self)

        return piece

    def place_piece(self) -> tuple[int, int]:
        move = check_for_win(self.get_game())
        if move is not None:
            return move
        move = mirror_strategy_move(self)
        return move
```

The deciosion of which piece to play is based on the following rules:

1. `block_strategy_piece` : the player can recognize when a piece is needed for the opponent to win and if possible will not hand it to player 2, effectively blocking the opponent from winning.

```
def block_strategy_piece(self):
    """check if the are moves that can make the opponent win
    if so, return the piece that can block the move
    param:
        usable_pieces: list of pieces that can be used
        game: current game
    return:
        pieces: list of pieces that will not make the opponent win
    """
    winner = False
    current_board = self.get_game().get_board_status()
    used_pieces = {c for x in current_board for c in x if c > -1}
    usable_pieces = [_ for _ in {x for x in range(16)} - used_pieces]
    possible_moves = [(c, r) for r in range(4) for c in range(4) if current_board[r][c] == -1]
    pieces = {}
    for p in usable_pieces:
        winner = False
        for m in possible_moves:
            # if i choose a piece and the opponent can not make a winning move with that piece add it to the list
            board = deepcopy(self.get_game())

            board.select(p)
            board.place(m[0], m[1])
            if board.check_winner() > -1:
                winner = True
        if not winner:
            pieces[p] = m

    return pieces
```

2. `mirror_strategy_piece` : based on the binary representation of the last piece placed on the board, the player will try to choose a piece that is as much different as possible from the last one. This is done by calculating the **Hamming distance** between the binary representation of the last piece and the binary representation of the possible pieces. The piece with the highest Hamming distance will be chosen.

```
def mirror_strategy_piece(self):
    """mirror the choice of the opponent
    if the opponent choose a piece, i choose the most different one
    """
    piece = random.randint(0, 15)

    current_board = self.get_game().get_board_status()

    if self.previous_piece == None:
        piece = random.randint(0, 15)
        # if i am plain first, i choose a random piece for the first move
    else:
        piece_info = self.get_game().get_piece_charachteristics(self.previous_piece).binary
        used_pieces = {c for x in current_board for c in x if c > -1}
        usable_pieces = [_ for _ in {x for x in range(16)} - used_pieces]
        pieces = list()
        for p in usable_pieces:
            # for each usable pieces find the most different from the previous piece
            p_info = [int(x) for x in format(p, "04b")]
            r = sum([abs(x - y) for x, y in zip(p_info, piece_info)])
            pieces.append((p, r))
            if r == 4:
                piece = p
                break
        piece = max(pieces, key=lambda x: x[1])[0]
    return piece
```

The deciosion of where to place the piece is based on the following rules:

1. `check_for_win`: the player can recognize when a move is needed for the player to win and if possible will play it, effectively winning the game.

```
def check_for_win(gameboard):
    """Check if the gameboard has a winning move. If so, return the move"""
    move = None

    piece = gameboard.get_selected_piece()
    current_board = gameboard.get_board_status()
    possible_moves = [(c, r) for r in range(4) for c in range(4) if current_board[r][c] == -1]
    for m in possible_moves:
        game = deepcopy(gameboard)
        game.select(piece)
        game.place(m[0], m[1])
        if game.check_winner() > -1:
            move = m
            break
    # return the list that can not make the opponent win
    return move
```

2. `mirror_strategy_move`: based on the last move played the player will try to place the piece in the mirroring position on the board.

```
def mirror_strategy_move(self):
    """mirror the choice of the opponent
    if the opponent choose a move, i choose the opposite one if i can
    """
    self.previous_piece = self.get_game().get_selected_piece()
    current_board = self.get_game().get_board_status()
    # find the previous move by vomparing the current board with the previous one and find the first difference
    for i, r in enumerate(zip(self.previous_board, current_board)):
        for j, c in enumerate(zip(r[0], r[1])):
            if c[0] != c[1]:
                self.previous_move = (i, j)
                break
    if self.previous_move != None:
        # if there is a previous move, i choose the opposite one if i can
        possible_moves = [(c, r) for r in range(4) for c in range(4) if self.previous_board[r][c] == -1]
        move = (3 - self.previous_move[1], 3 - self.previous_move[0])
        if move not in possible_moves:
            # if the opposite move is not possible, i try to find a move that is as far as possible from the previous one
            # the distance is calculated usiong the manhattan distance
            manhattan = list()
            for m in possible_moves:
                v = sum(abs(x - y) for x, y in zip(move, self.previous_move))
                manhattan.append((m, v))
            move = max(manhattan, key=lambda x: x[1])[0]
    else:
        move = (random.randint(0, 3), random.randint(0, 3))
    self.previous_board = deepcopy(current_board)  # update the previous board
    self.previous_board[move[1]][move[0]] = self.previous_piece  # update the previous board with the selected move
    return move
```

Results

This player is really effective against the other player tested up to this point, this are the results after **1000** matches:

	TrainedGAPlayer	RandomPlayer	DumbPlayer
RuleBasedPlayer	718	958	693

MinMaxPlayer

As the name suggest the next tecnique that i tried to implement was a minmax player with alpha beta pruning and a memory of previous states. The Player class is defined as follows:

```
class MinMaxPlayer(quarto.Player):
    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.first_move = True
        self.piece_choice = None
        self.move_choice = None
        self.memory = {}

    def choose_piece(self) -> int:
        piece = 0
        if self.first_move:
            piece = random.randint(0, 15)
        else:
            piece = self.piece_choice
        return piece

    def place_piece(self) -> tuple[int, int]:
        move = (0, 0)
        game = deepcopy(self.get_game())
        value, move, piece = minmax(self, depth=1, alpha=-math.inf, beta=math.inf, isMaximizing=True, game=game)
        self.piece_choice = piece
        self.move_choice = move
        return move
```

The `minmax` function is defined as follows:

```python
@cache
def minmax(self, depth, alpha, beta, isMaximizing, last_move=None, last_piece=None, game=None):
    """Minmax to choise the best move to play and piece to use"""
    if (isMaximizing and game.check_winner() > -1) or depth == 0:
        # if winning position or max depth reached
        # evaluate the position and return the value
        evaluation = self.evaluate_board(isMaximizing, game, last_move, last_piece)
        self.memory[(isMaximizing, hash(str(game)))] = evaluation
        return evaluation
    if (isMaximizing, hash(str(game))) in self.memory:
        # if the state is already solved in the memory return the value
        return self.memory[(isMaximizing, hash(str(game)))]

    best_choice = None
    selected_piece = game.get_selected_piece()
    board = game.get_board_status()
    avvailable_piece = get_available_pieces(board, selected_piece)
    available_moves = get_available_moves(board)
    if isMaximizing:
        best_choice = (-math.inf, -1, -1)
        for m in available_moves:
            game_copy = deepcopy(game)
            game_copy.place(m[0], m[1])
            for p in avvailable_piece:
                # for each move and each possible piece minimize the opponent and maximize my score
                if not game_copy.select(p):
                    logging.debug(f"piece {p} not available")
                evaluation = minmax(self, depth - 1, alpha, beta, False, m, p, game_copy)
                best_choice = max(best_choice, evaluation, key=lambda x: x[0])
                alpha = max(alpha, best_choice[0])
                if beta <= alpha or best_choice[0] == 100:  # alpha beta pruning or winning position
                    break
            if best_choice[0] == 100:
                break
        return best_choice

    else:
        best_choice = (math.inf, -1, -1)
        for m in available_moves:
            game_copy = deepcopy(game)
            game_copy.place(m[0], m[1])
            for p in avvailable_piece:
                # for each move and each possible piece minimize the opponent and maximize my score
                if not game_copy.select(p):
                    logging.debug(f"piece {p} not available")
                evaluation = minmax(self, depth - 1, alpha, beta, True, m, p, game_copy)
                best_choice = min(best_choice, evaluation, key=lambda x: x[0])
                beta = min(beta, best_choice[0])
                if beta <= alpha or best_choice[0] == 0:  # alpha beta pruning or losing position for minimizer
                    break
            if best_choice[0] == 0:
                break
        return best_choice
```

The drawback of this player is that it is really slow and can only work with a limited depth.
The evaluation function is defined as follows:

```
def evaluate_board(self, isMaximizing, game, last_move, last_piece):
        """Evaluate the board and return a value
        params:
            isMaximizing: True if the player is maximizing, False otherwise
            game: the game to evaluate
            last_move: the last move made
            last_piece: the last piece used
        return:
            a tuple containing the value of the board, the last move made and the last piece used
        """
        if game.check_winner() > -1:
            return (100, last_move, last_piece)

        usable_pieces = get_available_pieces(game.get_board_status())
        blocking_pieces = blocking_piece(usable_pieces, game)
        # the value is the percentage of blocking pieces if the player is maximizing
        # the value is the percentage of non blocking pieces if the player is minimizing
        if isMaximizing:
            v = len(blocking_pieces) * 100 / len(usable_pieces)
            return (v, last_move, last_piece)
        else:
            v = (len(usable_pieces) - len(blocking_pieces)) * 100 / len(usable_pieces)
            return (v, last_move, last_piece)
```

if the player is maximizin and the evaluating move is winnig the evaluation is 100, otherwise the evaluation is the number of blocking pieces divided by the number of available pieces. where blocking pieces are the pieces that can be used to block the opponent from winning.

```
def blocking_piece(usable_pieces, game):
    """check if the are moves that can make the opponent win
    if so, return the piece that can block the move
    param:
        usable_pieces: list of pieces that can be used
        game: current game
    return:
        pieces: list of pieces that will not make the opponent win
    """
    winner = False
    pieces = {}
    possible_moves = [(c, r) for r in range(4) for c in range(4) if game.get_board_status()[r][c] == -1]
    for p in usable_pieces:
        winner = False
        for m in possible_moves:
            # if i choose a piece and the opponent can not make a winning move with that piece add it to the list
            board = deepcopy(game)
            board.select(p)
            board.place(m[0], m[1])
            if board.check_winner() > -1:
                winner = True
        if not winner:
            pieces[p] = m
    # return the list that can not make the opponent win
    return pieces
```

RL Agent

Since I wasn't able to implement a good minmax player I decided to try to implement a reinforcement learning agent.
The core of the algorithm is expressed in this function:

```python
def reinforcement_training(
    game: quarto.Quarto, agent: ReinforcementLearningAgent, opponent: quarto.Player, num_matches: int
) -> None:
    win = 0
    print(f"training {0} / {NUM_MATCHES}", end="")
    for t in range(num_matches):
        game.reset()
        if t % 2 == 0:
            game.set_players((agent, opponent))
        else:
            game.set_players((opponent, agent))
        winner = game.run()
        if winner == t % 2:
            win += 1
            agent.learn(True)
        else:
            agent.learn(False)

        if SAVE_POLICY and t + 1 % LOG_FREQ == 0:
            print(f"\r\ntraining {t+1} / {NUM_MATCHES} - saving policy\n")
            policy = agent.get_policy()
            pickle.dump(policy, open(f"populations/policy-{t%LOG_FREQ}.pkl", "wb"))
        sys.stdout.flush()
        print(f"\rtraining {t+1} / {NUM_MATCHES}", end="")
    policy = agent.get_policy()
    pickle.dump(policy, open(f"populations/policy-{NUM_MATCHES}.pkl", "wb"))
    return win / num_matches
```

The agent is trained against an opponent for a large number of matches and after every match the agent will update his policy table based on the outcome of the match.
The Agent's class is defined as follows:

```python
class ReinforcementLearningAgent(quarto.Player):
    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.is_learning = False
        self.G = dict()
        self.current_state = dict()
        self.randomness = 0.7
        self.learning_rate = 10e-3

    def get_policy(self) -> dict:
        return self.G

    def set_learning(self, is_learning: bool) -> None:
        self.is_learning = is_learning

    def choose_piece(self) -> int:
        choiche = ...
        ...

        return choice

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

        return choice

    def learn(self, win: bool) -> None:
        for board, value in self.current_state.items():

            update_policy_table
            ...


        self.current_state = dict()
        self.randomness -= self.learning_rate
```

The policy table is defined as a dictionary where the key is the board status and the value is an othe dicvtionary with **2** keys `piece` and `move` and for each key the value is a list of tuple where the first element is the probability of choosing that piece or move and the second element is the value of the move or piece.
In order to update the policy table an history of the choosen pieces and moves is kept in the `current_state` dictionary which is updated in the `choose_piece` and `place_piece` methods as follows:

```
def choose_piece(self) -> int:
        board = self.get_game().get_board_status()
        available_pieces = get_available_pieces(board)
        available_moves = get_available_moves(board)

        if self.is_learning and random.random() < self.randomness:
            choice = random.choice(available_pieces)
            self.current_state[hash(str(board))] = {"piece": choice}
        else:
            if hash(str(board)) in self.G:
                try:
                    possible_chioce = self.G[hash(str(board))]["piece"]
                    choice = max(possible_chioce, key=possible_chioce[1])
                except:
                    choice = random.choice(available_pieces)
            else:
                choice = random.choice(available_pieces)

        return choice
```

```
def place_piece(self) -> tuple[int, int]:
        board = self.get_game().get_board_status()
        available_moves = get_available_moves(board)
        if self.is_learning and random.random() < self.randomness:
            choice = random.choice(available_moves)
            self.current_state[hash(str(board))] = {"move": choice}
        else:
            if hash(str(board)) in self.G:
                try:
                    possible_chioce = self.G[hash(str(board))]["move"]
                    choice = max(possible_chioce, key=possible_chioce[1])
                except:
                    choice = random.choice(available_moves)
            else:
                choice = random.choice(available_moves)
        return choice
```

The agent was trained for **5000** matches against different opponents and the obtained policy saved in a file to be used with a trained player.

Results

Herre we can see the results for **1000** matches between every opponents.

vs	TrainedGAPlayer	RandomPlayer	DumbPlayer	RuleBasedPlayer	TrainedRLPlayer
TrainedGAPlayer		557	1000	292	576
RandomPlayer	443		386	53	473
DumbPlayer	0	614		393	576
RuleBasedPlayer	718	947	607		958
TrainedRLPlayer	424	527	424	42	

Mixing strategies

In order to improve the performance of the agent I decided to mix the strategies of the different players. The first Player that i created is based on the `RuleBasedPlayer` and the `MinimaxPlayer`. Since minmax is really slow, especially for the first moves, I decided to implement a player that at first used a fixed set of rules but towards the end of the game switched to search for optimal solution using minmax. In the latest imlementation the minmax functionality is activated only when the number of available pieces is halfed.
The two main functions of the player are as folows:

```
def choose_piece(self) -> int:
        if self.minmax_piece is not None:
            piece = self.minmax_piece
            self.minmax_piece = None
            return piece

        pieces = block_strategy_piece(self)
        if len(pieces) < 3 and len(pieces) > 0:
            return list(pieces.keys())[0]
        piece = mirror_strategy_piece(self)
        return piece
```

In this function if the minmax is already been activated the function returns the piece choosen by minmax. Otherwise it checks if there is a piece that can block the opponent from winning and if there is it returns that piece. Finally if there is no such piece it checks if there is a piece that can be used to mirror the opponent's piece.

```
def place_piece(self) -> tuple[int, int]:
    move = check_for_win(self.get_game())
    if move is not None:
        return move

    usable_pieces = get_available_pieces(self.get_game().get_board_status())
    if len(usable_pieces) < 8:
        game = deepcopy(self.get_game())
        value, move, piece = minmax(self, depth=4, alpha=-math.inf, beta=math.inf, isMaximizing=True, game=game)
        if piece != -1 and value > 0:
            self.minmax_piece = piece
            self.previous_board[move[0]][move[1]] = self.get_game().get_selected_piece()
            self.previous_piece = piece
            return move
    move = mirror_strategy_move(self)

    return move
```

In this function the first thing that the player will do is to checkl wether there is a move that can be used to win the game. If there is it returns that move. Otherwise it checks if the number of available pieces is less than 8 and if it is it activates the minmax functionality. If the minmax functionality is activated it returns the move choosen by minmax. Finally if the minmax functionality is not activated it returns the move choosen by the mirror strategy.

Results

todo

Mixed strategies with RL policy

Similarly to what I did with the other mixed strategy player I tried to incorporate the policy learned by the RL agent in order to make a better choiche in the first stages of the game.
The two main functions are as follows:

```
    def choose_piece(self) -> int:
        available_pieces = get_available_pieces(self.get_game().get_board_status())

        if self.minmax_piece is not None and self.minmax_piece != -1 and self.minmax_piece in available_pieces:
            minmax_piece = self.minmax_piece
            return minmax_piece

        rl_piece = self.choose_piece_rl()
        if rl_piece is not None:
            return rl_piece
        else:
            mirror_piece = mirror_strategy_piece(self)
            if mirror_piece is not None and mirror_piece in available_pieces:
                return mirror_piece
            else:
                return random.choice(available_pieces)
```

```
def choose_piece_rl(self):
    board = self.get_game().get_board_status()
    if hash(str(board)) in self.G:
        try:
            possible_chioce = self.G[hash(str(board))]["piece"]
            choice = max(possible_chioce, key=possible_chioce[1])
        except:
            choice = None
    else:
        choice = None
    return choice
```

In this function the player first checks if the minmax functionality is activated and if it is it returns the piece choosen by minmax. Otherwise it checks if the RL agent has a policy for the current state and if it does it returns the piece choosen by the RL agent : `choose_piece_rl(self)` . If the RL agent does not have a policy for the current state it checks if there is a piece that can be used to mirror the opponent's piece and if there is it returns that piece. Finally if there is no such piece it returns a random piece.

```
    def place_piece(self) -> tuple[int, int]:
        winning_move = check_for_win(self.get_game())
        if winning_move is not None:
            return winning_move

        usable_pieces = get_available_pieces(self.get_game().get_board_status())

        if len(usable_pieces) < 7:
            game = deepcopy(self.get_game())
            value, minmax_move, minmax_piece = minmax(
                self, depth=4, alpha=-math.inf, beta=math.inf, isMaximizing=True, game=game
            )
            if value > 60 and minmax_piece != -1 and minmax_move != (-1, -1):
                self.minmax_piece = minmax_piece
                self.previous_board[minmax_move[0]][minmax_move[1]] = self.get_game().get_selected_piece()
                self.previous_piece = minmax_piece
                return minmax_move
        rl_move = self.choose_move_rl()
        if rl_move is not None:
            return rl_move
        else:
            mirror_move = mirror_strategy_move(self)
            if mirror_move is not None:
                return mirror_move
            else:
                return random.choice(get_available_moves(self.get_game().get_board_status()))
```

Similarly the second function at first check if there is a move that can be used to win the game. If there is it returns that move. Otherwise it checks if the number of available pieces is less than 7 and if it is it activates the minmax functionality. If the minmax functionality is activated it returns the move choosen by minmax. Otherwise it checks if the RL agent has a policy for the current state and if it does it returns the move choosen by the RL agent. If the RL agent does not have a policy for the current state it checks if there is a move that can be used to mirror the opponent's piece and if there is it returns that move. Finally if there is no such move it returns a random move.

```
def choose_move_rl(self):
    board = self.get_game().get_board_status()
    choice = None
    if hash(str(board)) in self.G:
        try:
            possible_chioce = self.G[hash(str(board))]["move"]
            choice = max(possible_chioce, key=possible_chioce[1])
        except:
            choice = None
    return choice
```

this function simply search in the RLpolicy if there is a policy for the current state and if there is it returns the piece choosen by the RL agent.

The resuls for this player are not improving much if at all compared to the other mixed strategy player. This is probably due to the fact that the RL agent is not learning much in the first stages of the game and therefore the policy is not very good.

My best player (S309413)

In the best player that i could find and implement is the first mixed strategy player that uses minmax, mirroring strategy and blocking strategy in order to choose the piece and the move.

The code for this player can be found in the file bestPlayer.py
but her is also a copy :

```python
class S309413(quarto.Player):
    """S309413 player"""

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.previous_board = np.array([[-1, -1, -1, -1], [-1, -1, -1, -1], [-1, -1, -1, -1], [-1, -1, -1, -1]])
        self.previous_move = None
        self.previous_piece = None
        self.minmax_piece = None
        self.memory = {}

    def choose_piece(self) -> int:
        """Choose a piece to play with:
        if minmax strategy is available, use it
        if not, use the blocking strategy
        if not, use the mirror strategy
        """
        if self.minmax_piece is not None and self.minmax_piece in get_available_pieces(
            self.get_game().get_board_status()
        ):
            piece = self.minmax_piece
            return piece

        pieces = block_strategy_piece(self)
        if len(pieces) < 3 and len(pieces) > 0:
            return list(pieces.keys())[0]
        piece = mirror_strategy_piece(self)
        return piece

    def place_piece(self) -> tuple[int, int]:
        """Place a piece on the board:
        if minmax strategy is available, use it
        if not, use the blocking strategy
        if not, use the mirror strategy
        """
        move = check_for_win(self.get_game())
        if move is not None:
            return move

        usable_pieces = get_available_pieces(self.get_game().get_board_status())
        if len(usable_pieces) < 6:
            game = deepcopy(self.get_game())
            value, move, piece = minmax(self, depth=3, alpha=-math.inf, beta=math.inf, isMaximizing=True, game=game)
            if piece != -1 and value > 0:
                self.minmax_piece = piece
                self.previous_board[move[0]][move[1]] = self.get_game().get_selected_piece()
                self.previous_piece = piece
                return move
        move = mirror_strategy_move(self)

        return move

    def evaluate_board(self, isMaximizing, game, last_move, last_piece):
        """Evaluate the board:
        parameters:
            isMaximizing: bool
            game: quarto.Quarto
            last_move: tuple[int, int]
            last_piece: int
        return:
            tuple[int, tuple[int, int], int] = (value, last_move, last_piece)
        """

        if game.check_winner() > -1:
            return (100, last_move, last_piece)

        usable_pieces = get_available_pieces(game.get_board_status())
        blocking_pieces = blocking_piece(usable_pieces, game)

        # the value is the percentage of blocking pieces if isMaximizing is True
        # the value is the percentage of non blocking pieces if isMaximizing is False

        if isMaximizing:
            v = len(blocking_pieces) * 100 / len(usable_pieces)
            return (v, last_move, last_piece)
        else:
```

```
            v = (len(usable_pieces) - len(blocking_pieces)) * 100 / len(usable_pieces)
            return (v, last_move, last_piece)
```

Results

This are the result for y best player compare to every other opponent tested for **1000** games, playing first for 500 and second for 500

vs	TrainedGAPlayer	RandomPlayer	DumbPlayer	RuleBasedPlayer	TrainedRLPlayer
S309413	692	999	645	593	958

Lab 1 : Set Covering Problem

First lab + peer review. List this activity in your final report, it will be part of your exam.

Task

Given a number N and some lists of integers $P = (L_0, L_1, L_2, ..., L_n)$,
determine, if 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.

Instructions

- Create the directory `lab1` inside the course repo (the one you registered with Andrea)
- Put a `README.md` and your solution (all the files, code and auxiliary data if needed)
- Use `problem` to generate the problems with different N
- In the `README.md`, report the the total numbers of elements in L_{s_i} for problem with $N \in [5, 10, 20, 100, 500, 1000]$ and the total number on $nodes$ visited during the search. Use `seed=42`.
- Use `GitHub Issues` to peer review others' lab

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

- Sunday, October 16th 23:59:59 for the working solution
- Sunday, October 23rd 23:59:59 for the peer reviews

Analyzing the Problem

At first i tought about using one of the algorithm proposed in class such as depth first, breadth first, A* and similiar in order to find an exact solution but i quickly realized that this approach would take too much to compute. Instead I tried to find a good solution in a reasonable amount of time even if it was not a global best solution using a greedy approach. For smaller value of N such as $N = 5$ and $N = 10$ my approach defenitly found a global best since the weight of my solution is the minimum possible. For bigger value of N my algorithm is unable to check if the solution is the global best but i decided to make this tradeoff in order to reach a solution in a reasonable amount of time.

My proposed solution

Reference and material

This solution was developed partially in conjunction with Elia Fontana and partially by searching some online references, which includes:

1. This paper by Drona Pratap Chandu on some greedy implementation of set cover
2. This thread on stackoverflow
3. The slide used by the Professor during lecture, especially the proposed solution for the MinSum problem

Greedy approach

Although I have used this as reference material my solution differ from the other that i have seen. The basic greedy implementation of this algorithm choose the local best set based on the maximum number of new elements that it will cover if inserted in te solution. My implementation instead is based on the percentage of new coverage in each set.
For examples, given:

$N = 10$
$goal = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$result = \{1, 3, 5\}$
$set = \{3, 5, 7, 8\}$

The weight is given by the fact that set has 4 values of which only 2 are not already in the reslut set, so it's weight is $\frac{2}{4} = 0.5$
then the percentage is transformed in a value between 0 and 100 using this formula

```
def calculate_weight(r,g,s = set(),threshold=1):
    w = ceil(100*sum(x in g-r for x in s) / len(s))
    return 100 if w>=threshold*100 else w
```

I decided also to include a treshold in order to speed up the algorithm and to drastically reduce the number of explored states, the default value is one but it can be expressed as a value between 0.01 and 1.
If the value is set as default only when a set has a coverage of 100% is chosen as a local best but if the value of threshold is set to a value < 1 then if the coverage of the set is $\geq threshold * 100$ then it will be considered a local optimum.

This weight is then used to store the sets in a PriorityQueue, in order to use this object i had to create my own class to store the sets with the priority calculate with the formula above, the class is:

```
class PrioritizedSet:
    def __init__(self, priority: int, s: set):
        self.priority = 100 - priority
        self.item = s

    def __lt__(self, other):
        return self.priority < other.priority

    def __eq__(self, other):
        return self.priority == other.priority

    def __gt__(self, other):
        return self.priority > other.priority
```

The priority is insrted as $100 - weight$ because the PriorityQueue insert the smallest priority at the top of the queue.

The core of my algorithm

```
while result is not None and not result_set == goal:
        while not options.empty():
            discovered_state += 1
            s = options.get().item
            coverage = calculate_weight(result_set,goal,s,threshold)
            if coverage == 100:
                result.append(s)
                result_set = result_set.union(s)
                while not unused.empty():
                    pq.put(unused.get())
                break
            elif coverage != 0:
                unused.put(PrioritizedSet(coverage,s))
        else:
            if unused.empty():
                result = None
                break
            else:
                local_best = unused.get().item
                result.append(local_best)
                result_set = result_set.union(local_best)
                while not unused.empty():
                    options.put(unused.get())
```

Until a solution is not reached i extract an element from the queue, then calculate its coverage. If the coverage is 100% the set is appended to the solution otherwise is appended to an other queue of unused set. the unused sets are reinserted in the original queue and the next local best set is calculated.

If there is no set with coverage 100% then the set with the gratest coverage is selected as a local best.

If a set has a coverage of 0 it means that adding it to the result set will only increment the weight of the solution, for this reason if found it is not reinserted in the unused queue but it is discarded.

If after calculating all the new weight and searching for a local best the unused queue is empty it means that there are no solution reachable from that state so the algorithm stops and return `None`

Results

Using a value of 1 for $threshold$ I obtained the following results:

$N = 5$
`Explored states` : 8
`Solution's weight` : 5

$N = 10$
`Explored states` : 52
`Solution's weight` : 10

$N = 20$
`Explored states` : 66
`Solution's weight` : 24

$N = 50$
`Explored states` : 832
`Solution's weight` : 76

$N = 100$
`Explored states` : 2098
`Solution's weight` : 181

$N = 500$
`Explored states` : 17914
`Solution's weight` : 1244

$N = 1000$
`Explored states` : 39631
`Solution's weight` : 2871

For a combined time of 2 minute and 23.2 seconds

Using a value of 0.5 for $threshold$ I obtained the following results:

$N = 5$
`Explored states` : 4
`Solution's weight` : 6

$N = 10$
`Explored states` : 17
`Solution's weight` : 13

$N = 20$
`Explored states` : 41
`Solution's weight` : 32

$N = 50$
`Explored states` : 591
`Solution's weight` : 77

$N = 100$
`Explored states` : 1035
`Solution's weight` : 172

$N = 500$
`Explored states` : 15959
`Solution's weight` : 1309

$N = 1000$
`Explored states` : 35395
`Solution's weight` : 2971

For a combined time of 2 minutes and 7.3 seconds
Which is not much in time saving but the space of the states explored is much smaller and in case for $N = 100$ it found even a better solution

Testing for different values of threshold

In the file results.txt file i have collected the results and timing for each $\forall n \in [0, N-1]$ and $\forall t \in [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$ where $t = threshold$

Updates after review

After receiving the first review i tried to implement the suggestion proposed in te Issues of my peer, in particular i added:

- Added some usefull comment in my code other than the readme in order to make it more readable and understandable
- Changed my approach using an ad hoc class `PrioritizedSet` instead using a tuple representing `(priority,list)`
- Added a file `Dijkstra.py` implementing a vanilla version of Dijkstra's algorithm in order to find a globally optimal solution for values of $N <= 20$

Peer reviewing

In order to review the code produced by some of my peer i opened a pull request or used the Issues tool provided by github,improving some aspects of the code and suggesting some other changes
I was assigned to review the code produced by Omid Mahdavii. The other peer review were chosen randomly from the list of github repos provided by the professor.

For this lab i reviewed:

- Omid Mahdavii forking his repo and creating a new pull request that can be found here

LAB 1 REVIEW

fixes and improvement provided in this pull request

1. `main function` : deleting redundant instruction such as creating a new initial state and appending a new initial value each loop. Added logging instruction instead of prints. Global declaration of global variable should be outside of main.
2. `findCommonElements` : it is more efficient to use set's properties like intersection instead of looping and checking to find common elements.
3. `listToSet` : changed to list comprehension instead of using itertools function, since there is no need for using it if the value returned by the function is not an iterator but a set instead.
4. `mySolution` : general refactoring of code such as deletion of else statement after a return, changed the exit condition from `len(state) >= n` to `len(state) > n` in order to account for valid solution of leght **n**.

Advice and other considerations

Probably recursion is not the best implementation of this solution since generally it is computationally intense and a basic iteration would be more efficient and reach the same goal, maybe using some data structure such as a PriorityQueue which also internally implement a sorting algorithm.

- Shayan Taghinezhad Roudbaraki using github Issues, my review can be found here

General overview

- The code is well written and easy to understand
- The README could be more complete

Major Issue

- There are no major issue in your code

Minor issue and possible improvements

- your result do not include the number of nodes visited searching for the solution, and it would have been better to include those results in the README
- while this approach is completly valid it does not produce the optimal solution nor the solution with less nodes visited. I would suggest to implement some heuristic in order to search through the lists more efficiently. A good starting point might be the lenght of the solution, or maybe the number of new element each list could provide to the solution

---

- RaminHedayatmehr also using github Issues, my review can be found here

General overview

The solution was a bit chaotic especially the one written in `other_algos.ipynb`. The README was not very well written, in particular:

- Try to describe your approach and how you implemented your solution
- Add the results you have found, it would have been good to compare the results obtained with the greedy approach and other algorithms
- Review the syntax for markdown (no space between [...] and (https://...) to make an hyperlink

Greedy approach

Your solution provided in `greedy.ipynb` is a good first attempt at finding a solution in a quick way. The code was well written and understandable.

Minor

- Add in the result the number of nodes visited in order to reach you solution

Other algos

The solution is a bit difficult to understand and even trying to executes it for different value of N was not intuitive:

Major

- If you want to use a notebook split your code in different section adding some description and comments. Otherwise is better to write a python module with `.py` extension
- The problem is not generated using the correct function, instead you used

```
SETS = np.random.random((NUM_SETS, NUM_AREAS)) < 0.5
```

it shoud be:

```
def problem(N, seed=None):
    """Creates a list of lists that contains a random amount of numbers between 0 and N-1."""
    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))
    ]
```

Minor

- As already stated a better description in the README or some comments in the code could make it more readable and easier to understand
- Add in the result the number of nodes visited in order to reach you solution

Lab 2 : Set Covering Problem solved by GA

Task

Given a number N and some lists of integers $P = (L_0, L_1, L_2, ..., L_n)$,
determine, if 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.

Analyzing the Problem

The Problem is the same as Lab1 so it does not need introductions. My approach to a solution was highly based on the olgorithm proposed by the professor during the lectures and then trying to tweak it aiming at better performance overall.

My proposed solution

Reference and material

This solution was developed by myself but the beginning stages of the problem solving were brainstormed togheter with my peers after classes.
The core of the algorithm is based on:

1. the solution proposed by the professor to solve the onemax problem
2. slides presented during lectures
3. my notes that can be found here

early stages of the solution

All my attemps to solve the problem can be found here unfortunately since I was sick during the days before the deadline I had to rush the solution and I did not had enough time to explore more possibilities. For this reason my solution is pretty basic but I think I have found some good strategies. I will try other solution in the future and I will update this file with the results.

Genetic Algorithm

The core of my solution is standard GA with recombination.
For each Problem space I create a population of 300 individuals.
Then I calculate an offspring of 200 individuals that is combined with the parent population. This is done for 100 generations.
To this standard approach I added a check in order to avoid a steady state, if for 7 generation the bfitness of the best individual does not change I combine the current population with a new one generated from scartch.
my approach is defined here:

```
population = generate_population(N, all_lists, maxlen)
bestfit = population[0].fitness
for _ in range(NUM_GENERATIONS):
        offspring = generate_offspring(population, all_lists, N, maxlen)
        population = combine(population, offspring)
        steady += 1
        if population[0].fitness > bestfit:
            bestfit = population[0].fitness
            steady = 0
        if steady == 7:
            population = combine(population, generate_population(N, all_lists, maxlen))
            steady = 0
```

Offspring generation

Each offspring is generated by either **mutation** or **crossover** the parent(s) are chosen with a tournament function with $k = 2$.

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

- **mutation**:
  the offspring is generated starting from the parent and adding or substituting a list from the original problem.
  With a probability of 0.3 the new list is added to the genome of the parent.

  ```
  if random.random() < 0.3:
      o = g[:point] + (random.choice(all_lists),) + g[point:]
  ```

  With a probability of 0.7 the new list is substituted to the genome of the parent.

  ```
  else:
      o = g[:point] + (random.choice(all_lists),) + g[point+1:]
  ```

  In the end with a probability of 0.3 the offspring's genome is shortened by a random amount $l \in [1, len(g)]$

  ```
  if random.random() < 0.3:
      o = o[: -random.randint(1, len(o))]
  ```

- **crossover**:
  The crossover is performed by combining the genome of the two selected parents.
  With a probability of 0.5 the offspring is generated by combining the first half of the first parent with the second half of the second parent.

  ```
  if random.random() < 0.5:
      o = g1[:cut] + g2[cut:]
  ```

  With a probability of 0.5 the offspring is generated by combining the first half of the second parent with the second half of the first parent.

  ```
  else:
      o = g2[:cut] + g1[cut:]
  ```

  In the end similarly to the mutation phase with a probability of 0.3 the offspring's genome is shortened by a random amount $l \in [1, len(g)]$

  ```
  if random.random() < 0.3:
      o = o[: -random.randint(1, len(o))]
  ```

Fitness function

The fitness is expressed as a tuple of three values:

1. `fitness` : the actual fitness calculated by calculating the number of elements already covered by the genome and sumbtracting a penalty that accounts for the repetitions in the genome, the penalty is calculated as `(duplicates * N / maxlen)` where `duplicates` is the number of duplicates in the genome and `maxlen` is the maximum length of the lists in the problem space and `N` is the number of elements to cover.
2. `covered` : the number of elements covered by the genome
3. `- duplicates` : the number of duplicates in the genome as a negative value since the less repetitions the better.

```
covered = len(set(loci for gene in genome for loci in gene))
    duplicates = len([loci for gene in genome for loci in gene]) - covered
    fitness = covered - (duplicates * N / maxlen)
    return (fitness, covered, -duplicates)
```

Results

$N = 5$
`Solution's weight` : 5
`solution's bloat` : 0
$N = 10$
`Solution's weight` : $1o$
`solution's bloat` : 0
$N = 20$
`Solution's weight` : 27
`solution's bloat` : 35
$N = 50$
`Solution's weight` : 79
`solution's bloat` : 58
$N = 100$
`Solution's weight` : 201
`solution's bloat` : 101
$N = 200$
`Solution's weight` : 487
`solution's bloat` : 144
$N = 500$
`Solution's weight` : 1479
`solution's bloat` : 196
$N = 1000$
`Solution's weight` : 3680
`solution's bloat` : 268

Peer reviewing

In order to review the code produced by some of my peer i used the Issues tool provided by github, suggesting some changes and improvements. For this lab i reviewed:

- Enrico Magliano using github Issues, my review can be found here

General

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

Suggestions

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

```
return len(set([loci for gene in sol for loci in gene])) == N
```

3. **fitness**:
   in the same way as for `check_sol` you could have used list comprehension
4. **General**:
   Could have reported the result in the README for better understanding and ease of use

---

- Peppepelle99 using github Issues, my review can be found here
  Here are some fixes and possible improvements to your solution:

Major

- **compute_fitness** : you should not use built-in functions name as variable names for example in

```
def compute_fitness(genome):
    list = gen2List(genome,list_of_lists)
    return (N - len(GOAL - set(list))) - numpy.sqrt(len(list))
```

it wold be better to rename the variable list to something else

Minor

- **gen2list** you could have used itertool compress to achieve the same goal instead of using a for loop:

```
def gen2List(genome,lol):
    l = list(compress(lol, genome))
    return [loci for gene in l for loci in gene]
```

- when computing list of list there is no need for `tmp` variable:

```
list_of_lists = [tuple(x) for x in lol]
```

- minor error in **README**:

- `TOURNAMENT_SIZE` = $\frac{N}{3}$ but in the code you write `TOURNAMENT_SIZE` = $\frac{N}{2}$

Possible improvements

Your algorithm generally find a good solution in the first few generations and then it does not improve for hundreds of generations. Maybe introducing some sort of stop if a steady state is reached or even better some reshuffling or introducing new individuals in the population once a steady state is reached, this will help the algorithm to explore new states maybe.

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`.

Deadlines (AoE)

- Sunday, December 4th for Task3.1 and Task3.2
- Sunday, December 11th for Task3.3 and Task3.4
- Sunday, December 18th for all reviews

Task 3.1

An agent using fixed rules based on *nim-sum* (i.e., an *expert system*)

this is the simplest of the tasks, we just need to implement the rules of the game and the agent will play accordingly. The obvious answer is to use nim-sum but we can also try some other tactics.
some examples of agents can be found in the file opponents.py

Task 3.2

An agent using evolved rules

My approach to this task can be seen in the file nim_GA.py

Core idea

The idea is to use a gnetic approach to evolve a genome that will be used to play the game. Given a population size of N, N individuals are randomly generated. Starting from this individuals O offspring are generated applyng some sort of mutation to the genome of the parent. The offspring are then evaluated and the best N individuals are selected to be the parents of the next generation. This process is repeated until a certain number of generations is reached.

Genome and representation

At first I tried to use some simple rules such as *shortest-row* and *longest-row* and using a genome composed of two parameters: `genome = {"p_rule": random.random(), "p_row": random.random()}`. The first parameter is used as a probability of using the first or the second rule, the second parameter is used as a percentage of the row to be removed. The problem with this approach is that the agent is not able to learn the best strategy, it just learns to play a bit better than the random agent.
Then i tried different parameters for different approaches such as `genome = {"aggression": random.random(), "finisher": random.random()}` where the first parameter is used to decide if the agent should play aggressively or not, the second parameter is used to decide if the agent should try to take all objects from a row or leave one object.
Ultimately all of my attempts failed and i decided to use a different approach, finally i came up with a genome inspired by the way of nim-sum is calculated.
the genome is composed of 4 parameters:

```
genome = {
            "alpha": random.uniform(-10, 10),
            "beta": random.uniform(-10, 10),
            "gamma": random.uniform(-10, 10),
            "delta": random.uniform(-10, 10),
        }
```

each of this parameters is then used to calculate the score for a given board in the following way:

```
res = (
            (a[0], abs(alpha * a[1] + beta * b[1] + gamma * c[1] + delta * d[1] + epsilon * e[1]))
            for a, b, c, d in zip(data["and"], data["or"], data["sum"], data["sub"])
        )
```

where `data` is a dictionary calculated in the function `cook_status()`. Each key represent an operation calculated between the different rows of the board in the following way:

1. **sub** : subtraction between the rows

```
def sub_operation(state: Nim) -> int:
    *_, result = accumulate(state.rows, operator.sub)
    return result
```

2. **sum** : sum between the rows

```
sum(state.rows)
```

3. **and** : and between the rows

```
def and_operation(state: Nim) -> int:
    *_, result = accumulate(state.rows, operator.and_)
    return result
```

4. **or** : or between the rows

```
def or_operation(state: Nim) -> int:
    *_, result = accumulate(state.rows, operator.or_)
    return result
```

Finally **res** is a generator where for each move given an initial state the score is calculated based on the parameters of the genome. The score is calculated as the absolute value of the sum of the parameters multiplied by the value of the operation between the rows of the board. The parameters are then used to calculate the score for each move and the move with the lowest score is chosen.

Mutations

The mutations are applied to the genome in the following way:

```
for i in range(OFFSPRING):
        p = tournament(population)

        p[2]["alpha"] += random.gauss(0, 10 / (gen + 1))
        p[2]["beta"] += random.gauss(0, 10 / (gen + 1))
        p[2]["gamma"] += random.gauss(0, 10 / (gen + 1))
        p[2]["delta"] += random.gauss(0, 10 / (gen + 1))
        p[2]["epsilon"] += random.gauss(0, 10 / (gen + 1))

        strat = make_strategy(p[2])
```

for each offspring a parent **p** is selected using the tournament selection method. Then the parameters of the genome are mutated using a gaussian distribution with a standard deviation that decreases with the number of generations. Finally the strategy is created using the new genome.

Evaluations

In order to evaluate the performance of the agent i tested it agains four different opponents:

1. **optimal_strategy**: the optimal strategy for the game using nim-sum
2. **pure_random**: a random agent
3. **gabriele**: an agent that uses the same approach proposed by one of our colleagues
4. **take_one_never_finish**: an agent that takes one object from a random row, if it can it will never remove the last object from a row

Each opponent is played for a fixed number of matches **NUM_MATCHES** and the number of victory with each component is saved in a tuple.

```
eval = (
        evaluate(strat, opponents.optimal_startegy),
        evaluate(strat, opponents.pure_random),
        evaluate(strat, opponents.gabriele),
        evaluate(strat, opponents.take_one_never_finish),
        )
```

Results and minor tweaks

Using this apprach it is possible to evolve a genome able to beat all the opponents consistently, except for nim-sum where it reaches a win rate of roughly 52.
It is important to notes that this result is only reachable in a reasonable amount of time for boards with a fixed size that is not too big. In my case i calculated the best genome for a board with 5 rows:

```
genome = {
        "alpha": 12.812770589035535,
        "beta": -16.051123920350758,
        "gamma": -0.20956437443764508,
        "delta": -8.234717910949916,
    }
```

I tried to calculate a similar genome but for a generic board size but it was not possible to reach the same result in a reasonable amount of time. The best i could do was an agent with a win rate of 1.5 against nim-sum. The genome for this agent is:

```
genome = {
        "alpha": 15.945309194204931,
        "beta": -3.2707966609771746,
        "gamma": -25.708257470959275,
        "delta": 14.81947128092396,
    }
```

I tried also an approach with five parameters, with the fifth one used in the formula above as a constant multiplied by the result of an other function, in particular the nand operation between rows:

```
def nand_operation(state: Nim) -> int:
    *_, result = accumulate(state.rows, lambda x, y: ~(x & y))
    return result
```

The agent with a five parameters genome was able to beat nim-sum with a win rate of 62 for a fixed board size of 5 rows. The genome for this agent is:

```
genome = {
        "alpha": -42.5399485484396,
        "beta": 114.60961375796023,
        "gamma": -52.64808867035252,
        "delta": 0.49668038593870456,
        "epsilon": 18.15686871650329,
    }
```

But the result for a random sized board were similar to the one with four parameters.

All of my previews attempt can be found here and my optimal palyers here

Task 3.3

An agent using minmax

The minmax algorithm is a recursive algorithm that is used to find the best move for a player in a two player game. It is based on the assumption that the opponent will play in the best way possible. The algorith is implemented in the function `minmax()` here
In order too speed up the process of recursion i decided to use a cache to store the results of the function `minmax()` for each state. The cache is implemented as a dictionary where the key is a tuple of (state,player) and the value is a boolean value to represent the outcome of the game starting from that state for the player zero.
An other aspect of the algorith used to speed up the process is the use of alpha-beta pruning. Here is used to prune the branches of the tree that are not useful for the calculation of the best move.
If the player is minimizing means that given a state if there is a move that leads to a loss for the maximising player we don't need to check the other moves because the minimising player will choose the move that leads to the loss. a similar approach is used for the maximising player.

My full algorithm implementation can be found here

Task 3.3

An agent using reinforcement learning

I implemented a simple reinforcement learnming agent, it is implemented in the class

```
class Agent(object):
    def __init__(self, state, alpha=0.15, random_factor=0.2):
        self.state_history = [(tuple(state._rows), 0)]  # state, reward
        self.alpha = alpha
        self.random_factor = random_factor
        self.G = {}
        self.init_reward(state)

    def init_reward(self, state):
        for i in product(*(range(x + 1) for x in state._rows)):
            self.G[i] = random.uniform(0.1, 1)

    def choose_action(self, state, allowedMoves):
        maxG = -10e15
        next_move = None

        randomN = random.random()
        if randomN < self.random_factor:
            random.randint(0, len(allowedMoves) - 1)
            next_move = allowedMoves[random.randrange(len(allowedMoves))]
        else:
            # if exploiting, gather all possible actions and choose one with the highest G (reward)
            for action in allowedMoves:
                stateCopy = deepcopy(state)
                stateCopy.nimming(action)
                if self.G[tuple(stateCopy._rows)] >= maxG:
                    next_move = action
                    maxG = self.G[tuple(stateCopy._rows)]

        return next_move

    def update_state_history(self, state, reward):
        self.state_history.append((tuple(state._rows), reward))

    def learn(self):
        target = 0

        for prev, reward in reversed(self.state_history):
            self.G[prev] = self.G[prev] + self.alpha * (target - self.G[prev])
            target += reward

        self.state_history = []

        self.random_factor -= 10e-5  # decrease random factor each episode of pla

    def get_policy(self):
        return self.G
```

The callable agent i:

```
def RLAgent(G: dict) -> Nimply:
    """the agent related to the task 3.4"""

    def agent(state: Nim):
        possibleStates = cook_data(state)["brute_force"]
        ply = max(((s[0], G[s[1].rows]) for s in possibleStates if s[1].rows in G), key=lambda i: i[1])[0]
        return Nimply(ply[0], ply[1])

    return agent
```

that uses the utility function `cook_data()` to get the possible moves and the next state for each move. Then it uses the dictionary `G` to get the reward for each possible next state and choose the move that leads to the state with the highest reward.

```
def cook_data(state: Nim) -> dict:
    # print(f"rows: {state.rows}")
    bf = []
    data = {}
    possible_moves = [(r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1)]
    for m in possible_moves:
        tmp = deepcopy(state)
        tmp.nimming(m)
        # bf.append(tmp)
        bf.append((m, tmp))
    data["brute_force"] = bf
    data["possible_moves"] = possible_moves

    return data
```

The main loop for the training of the agent is:

```
for t in range(TRAINING_TIME):

        stateCopy = deepcopy(nim)
        while stateCopy:
            possiblePlies = cook_data(stateCopy)["possible_moves"]
            action = agent.choose_action(stateCopy, possiblePlies)
            stateCopy.nimming(action)
            # give a 0 reward if I am winning, -10 if I am losing, and -0.5 if not in a deterministic state
            reward = -10 if sum(i > 0 for i in stateCopy._rows) == 1 else -0.5 * int(sum(stateCopy._rows) > 0)

            agent.update_state_history(stateCopy, reward)

            if sum(stateCopy._rows) == 0:
                break
            stateCopy.nimming(opponent(stateCopy))

        agent.learn()
        sys.stdout.flush()
        print(f"\rtraining {t+1} / {TRAINING_TIME}", end="")
```

Peer reviewing

In order to review the code produced by some of my peer i used the Issues tool provided by github, suggesting some changes and improvements. For this lab i reviewed:

- Leonor Gomes creating a pull request since in the README asked to fix the `minimax_pruning` function, my pull request can be found here

Task 3.3

Fixing minimax_pruning

I have seen that your implementation of minimax with alpha-beta pruninig wasn't working.
The problem was that in the basic minimax implementation you used an integer 1 or 0 in order to distinguish the two players but in the implementation with priuning the vsrisble was a boolean True/False. This caused some problem with the evaluation function, since it expected value (0,1) it worked correctly only for the implementation without pruning. In order to fix the problem I changed your code to always use **True** or **False** to indicates the player and changed the evaluation function like this:

```
def evaluate(nim_rows, player):  # evaluate if the game is finished
    if sum(nim_rows) == 0:
        return -1 if player else 1
    else:
        return None
```

Other small improvements

in the same pull request i have implemented a more efficient version of `nimply_move`:

```
def nimply_move(current_state, new_state):
    for i, r in enumerate(zip(current_state, new_state)):
        if r[0] != r[1]:
            return Nimply(i, r[0] - r[1])
```

in this way you don't have nested loop whic are slower

Task 3.2

I have some small suggestion to maybe(?) improve your implementation:

1. I noted that your mutation implementation always start with the genome of the parents in order to create a new genome, maybe trying to apply a true random mutation maybe with `random.uniform(A,B)` in order to explore more solution
2. Similarly to the first suggestion, when you initialize the population you create all the genome with almost the same parameters `'Rule2': [random.randint(0,1), max_leave]` the first parameters is random picked between 0 and 1 but the second is always `max_leave` maybe you could try to choose a value with `random.randrange(max_leave)`

README

You could have added some results for each strategy and maybe a minimal explaination for the reinforcement learning part could have been usefull

---

- Omid Mahdavii using github Issues, my review can be found here
  The code was well written and uderstandable, I appreciated the inclusion of result in the README but a brief explaination of your solutions would have been welcomed. My main suggestions regards the **Task 3.2**.

Task 3.2

I feel like your solution was overly complicated and could have reached the same or even better result using much less parameter. for what I understood he core idea of the solution is to choose between **longest_row** or **shortest_row** and then choos between **leaving_one**, **taking_all** or **percentage_of_elem** in a row. for this reason i thick you could have used only 3 parameters:

1. $p1$ to choose between longest or shortest row
2. $p2$ to choose between taking all elements or leaving one on the row
3. $p3$ to pick a different number of element like you did here: `num_objects = math.floor(state.rows[row] * genome[8])`

I tried to change your code and run it with this 3 parameters genearet randomly as you did for all the other parameters, and with this genome: $rules = [0.066, 0.94, 0.233]$
and this modified code inside of the `evolvable` function:

```
if random.random() < genome[0]:
            row = data["longestRow"]
        else:
            row = data["shortestRow"]

        if random.random() < genome[1]:
            num_objects = state.rows[row]
        elif random.random() < genome[2]:
            num_objects = 1
        else:
            num_objects = math.floor(state.rows[row] * genome[2])
```

I obtained this result: $resultHistory = [0.85, 0.95, 0.7, 0.85, 0.75, 0.9, 0.95, 0.8, 0.9, 1.0]$ that are even better than what you have reched with 12 parameters
Other than a small improvement in **Task 3.4** I have not much to improve since your code is really good for the other task.

Task 3.4

The function `init_reward` can be compacted in just 1 loop instead of 2:

```
for i in product(*(range(x + 1) for x in state._rows)):
                self.G[i] = np.random.uniform(low=1.0, high=0.1)
```

Code for Lab 1

setcovering solution

```
import random
import logging
from itertools import groupby
from queue import PriorityQueue
from time import perf_counter_ns
from math import ceil

SEED = 42

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


def problem(N, seed=None):
    """Generate a random problem given a seed
    ...
    Attributes :
        N : int
            The number of elements in the universe
        seed : int
            The seed for the random number generator
            default value = 42
    """
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N // 5, N // 2))))
        for n in range(random.randint(N, N * 5))
    ]


def calculate_weight(result, goal, set_=set(), threshold=1):
    """Calculate the weight of a set
    ...
    Attributes :
        result : set
            The set of elements already covered
        goal : set
            The set of elements to cover
        set_ : set
            The set to evaluate
            default value = set()
        threshold : float
            The threshold for the weight
            default value = 1
    """
    # weight = the percentage of new elements covered by the set in respect to the state of the solution
    weight = ceil(100 * sum(x in goal - result for x in set_) / len(set_))
    # return 0 if weight >= threshold * 100 else 100 - weight
    return 100 if weight >= threshold * 100 else weight


def search(sets, goal, N, threshold=1):
    """Search for a solution to the set covering problem
    ...
    Attributes :
        sets : list
            The list of sets
        goal : set
            The set of elements to cover
        N : int
            The number of elements in the universe
        threshold : float
            The threshold for the weight
            default value = 1
    """
    discovered_state = 0
    options = PriorityQueue()
    unused = PriorityQueue()
    # added a precomputing step:
    # remove empty lists and remove duplicates
    sets = [s for s in sets if s]
    sets = list(sets for sets, _ in groupby(sets))
```

```
    for element in sets:
        options.put((100 - int(100 * len(element) / N), element))
    result = [options.get()[-1]]
    result_set = set().union(result[0])
    while result is not None and not result_set == goal:
        while not options.empty():  # until i have options extract an element
            discovered_state += 1
            s = options.get()[-1]
            coverage = calculate_weight(result_set, goal, s, threshold)
            if coverage == 100:  # if all the elements in the set are new to the solution
                result.append(s)  # add it to the solution
                result_set = result_set.union(s)
                while not unused.empty():  # reinsert unused sets in the options queue
                    options.put(unused.get())
                break
            if coverage != 0:  # if the coverage is > 0% insert it in the unused queue
                unused.put((100 - coverage, s))
        else:  # after checking all options
            if unused.empty():  # if there are no unused sets a result can not be reached
                result = None
                break
            local_best = unused.get()[-1]  # extract the locally best unused set
            result.append(local_best)  # append it to the solution
            result_set = result_set.union(local_best)
            while not unused.empty():
                options.put(unused.get())  # reinsert unused sets in the options queue
    logging.info(f"explored state: {discovered_state}")
    return result


def main():
    """Main function"""
    for n in [5, 10, 20, 50, 100, 500, 1000]:
        sets = problem(n, SEED)
        goal = set(_ for _ in range(n))
        logging.info(f"N = {n}")
        start_time_ns = perf_counter_ns()
        result = search(sets, goal, n, threshold=1)
        end_time_ns = perf_counter_ns()
        logging.info(f"Time: {end_time_ns-start_time_ns} ns")
        if result is None:
            logging.info("No solution found")
        else:
            logging.info(f"the weight of the solution is: {sum(len(s) for s in result)}")


if __name__ == "__main__":
    main()
```

dijkstra solution implemented after the peer reviews

```python
import random
import heapq
from collections import Counter
from typing import Callable
import logging

N = 5
SEED = 42

logging.getLogger().setLevel(logging.DEBUG)


class PriorityQueue:
    """A basic Priority Queue with simple performance optimizations"""

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

    def __bool__(self):
        return bool(self._data_set)

    def __contains__(self, item):
        return item in self._data_set

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

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


def problem(N, seed=None):
    """
    Creates an instance of the problem
    param N: number of elements
    param seed: random seed
    return: list of lists
    """
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N // 5, N // 2))))
        for n in range(random.randint(N, N * 5))
    ]


def state_to_set(state):
    """Converts a state to a set of integers"""
    return set(sum((e for e in state), start=()))


def goal_test(state, GOAL):
    """
    Checks if the state is a goal state
    param state: a state
    param GOAL: the goal state
    return: True if the state is a goal state
    """
    return state_to_set(state) == GOAL


def possible_steps(state, all_lists):
    """
    Returns a list of possible steps
    param state: a state
    param all_lists: a list of lists of possible sets to choose from
    return: a list of possible steps
    """
    current = state_to_set(state)
    return [l for l in all_lists if not set(l) <= current]
```

```
def weight(state):
    """Returns the weight of a state
    param state: a state
    return: the weight of the state as 2 value:
        - the number of repetitions of elements
        - the number of elements with no repetitions as a negative value
    """
    cnt = Counter()
    cnt.update(sum((e for e in state), start=()))
    return sum(cnt[c] - 1 for c in cnt if cnt[c] > 1), -sum(cnt[c] == 1 for c in cnt)


def dijkstra(N, all_lists):
    """
    Vanilla Dijkstra's algorithm
    param N: number of elements
    param all_lists: a list of lists of possible sets to choose from
    return: the solution to the set covering problem"""

    GOAL = set(range(N))
    all_lists = tuple(set(tuple(_) for _ in all_lists))
    frontier = PriorityQueue()
    nodes = 0
    state = tuple()
    while state is not None and not goal_test(state, GOAL):
        nodes += 1
        for s in possible_steps(state, all_lists):
            frontier.push((*state, s), p=weight((*state, s)))
        state = frontier.pop()

    logging.info(f"nodes visited={nodes}; weight={sum(len(_) for _ in state)}")
    return state


def main():
    for N in [5, 10, 20]:
        logging.info(f" Solution for N={N}:")
        solution = dijkstra(N, problem(N, seed=42))


if __name__ == "__main__":
    main()
```

Code for Lab 2

```python
import logging
import random
from collections import namedtuple

POPULATION_SIZE = 300
OFFSPRING_SIZE = 200

NUM_GENERATIONS = 100

DEBUG = False
logging.basicConfig(level=logging.DEBUG if DEBUG else logging.INFO)
# basic data structure for an individual in order to access its parameters
Individual = namedtuple("Individual", ["genome", "fitness"])


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))
    ]


def fitness(genome, N, maxlen):
    """
    calculate the fitness of a genome
    param genome: the genome to calculate the fitness of
    param N: the size of the problem
    param maxlen: the maximum length of a list
    return: the fitness of the genome
    the fitness is a tuple containing:
        the actual fitness calculated as covered elements - a penalty for duplicates
        the number of covered elements
        the number of duplicates as a negative value (the less duplicates the better)
    """

    covered = len(set(loci for gene in genome for loci in gene))
    duplicates = len([loci for gene in genome for loci in gene]) - covered
    fitness = covered - (duplicates * N / maxlen)
    return (fitness, covered, -duplicates)


def tournament(population, tournament_size=2):
    """
    default tournament with k = 2
    select a random individual from the population
    and compare it with another random individual
    return the fittest individual
    """
    return max(random.choices(population, k=tournament_size), key=lambda i: i.fitness)


def cross_over(g1, g2):
    """
    Crossover two genomes by slicing them at a random point
    param g1: genome 1
    param g2: genome 2
    return: a new genome
    with probabiiity 0.5 the first part of g1 is combined with the second part of g2
    with probability 0.5 the first part of g2 is combined with the second part of g1
    with probability 0.3 the genome is shortened by a random amount
    """
    logging.debug(f"crossover lenghts : {len(g1)} and {len(g2)}")
    cut = random.randint(0, min(len(g1), len(g2)))
    if random.random() < 0.5:
        o = g1[:cut] + g2[cut:]
    else:
        o = g2[:cut] + g1[cut:]
    if random.random() < 0.3:
        o = o[: -random.randint(1, len(o))]
    return o


def mutation(g, all_lists):
```

```
    """
    mutate a genome by adding a random gene from all_lists
    param g: genome to mutate
    param all_lists: list of all genes
    return: mutated genome
    with probsbility 0.3 a random list is added to the genome
    with probability 0.7 a random list is substituted with another random list
    with a probability of 0.3 the genome is shortened by a random amount
    """
    logging.debug(f"mutation lenght : {len(g)}")
    point = random.randint(0, len(g) - 1)
    if random.random() < 0.3:
        o = g[:point] + (random.choice(all_lists),) + g[point:]
    else:
        o = g[:point] + (random.choice(all_lists),) + g[point + 1 :]
    if random.random() < 0.3:
        o = o[: -random.randint(1, len(o))]
    return o


def generate_population(N, all_lists, maxlen):
    """
    generate a population of POPULATION_SIZE individuals
    param N: the size of the problem
    param all_lists: the list of all possible lists
    param maxlen: the maximum length of a list
    return:
        a list of POPULATION_SIZE individuals
    """
    population = list()
    for genome in [
        tuple([random.choice(all_lists) for _ in range(random.randint(1, N // 3))]) for _ in range(POPULATION_SIZE)
    ]:
        population.append(Individual(genome, fitness(genome, N, maxlen)))
    return population


def generate_offspring(population, all_lists, N, maxlen):
    """
    param population: the current population
    param all_lists: the list of all possible lists
    param N: the size of the problem
    param maxlen: the maximum length of a list
    return: a list of offspring

    apply mutation with a probability of 0.4
    apply crossover with a probability of 0.6
    """
    offspring = list()
    for i in range(OFFSPRING_SIZE):
        if random.random() < 0.4:
            p = tournament(population)
            o = mutation(p.genome, all_lists)
        else:
            p1 = tournament(population)
            p2 = tournament(population)
            o = cross_over(p1.genome, p2.genome)
        # sometimes crossover or mutation can produce an empty genome
        if len(o) > 0:
            f = fitness(o, N, maxlen)
            offspring.append(Individual(o, f))
    return offspring


def combine(population, offspring):
    """
    combine the current population and a new one into a new population
    """
    population += offspring
    population = sorted(population, key=lambda i: i.fitness, reverse=True)[:POPULATION_SIZE]
    return population


def setCovering(N, all_lists):
    """
    Solve the set covering problem using a genetic algorithm
    for each generation generate offspring and combine them with the population
    and then select the best individuals
    if a steady state is reached where the best_fitness doesn't change
    for 7 iteration than a reshuffle and combination with a new population is performed
```

```
    return:
        the fittest individual after NUM_GENERATIONS
    """
    maxlen = sum(len(l) for l in all_lists)
    population = generate_population(N, all_lists, maxlen)
    bestfit = population[0].fitness
    steady = 0
    for individual in population:
        logging.debug(f"Genome: {individual.genome}, Fitness: {individual.fitness}")

    for _ in range(NUM_GENERATIONS):
        offspring = generate_offspring(population, all_lists, N, maxlen)
        population = combine(population, offspring)
        steady += 1
        if population[0].fitness > bestfit:
            bestfit = population[0].fitness
            steady = 0
        if steady == 7:
            population = combine(population, generate_population(N, all_lists, maxlen))
            steady = 0

    return population[0].genome


def main():
    """
    main function used to iterate over the various problem space
    for N = [5, 10, 20, 50, 100, 200, 500, 1000]
    and logging each result found
    """
    solutions = list()
    for N in [5, 10, 20, 50, 100, 200, 500, 1000]:

        solution = setCovering(N, problem(N, seed=42))
        solutions.append(solution)
        logging.info(
            f" Solution for N={N:,}: "
            + f"w={sum(len(_) for _ in solution):,} "
            + f"(bloat={(sum(len(_) for _ in solution)-N)/N*100:.0f}%)"
        )
    for s in solutions:
        flat = [i for a in s for i in a]
        logging.debug(f"Solution: {sorted(flat)}")


if __name__ == "__main__":
    main()
```

Code for Lab 3

Task 1 (plus other opponents)

```python
import logging
import random
import operator
import json
from collections import namedtuple
from itertools import accumulate
from copy import deepcopy
from functools import cache


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._rows = [i + 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) + ">"

    def __hash__(self):
        return hash(str(self))

    def __eq__(self, other):
        return str(self) == str(other)

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

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

    def nimming(self, ply: Nimply) -> None:
        row, num_objects = ply
        try:
            assert self._rows[row] >= num_objects
        except AssertionError as er:
            logging.debug(f"AssertionError: {er}")
            logging.debug(f"row: {row}, num_objects: {num_objects}")
            logging.debug(f"self : {self}")
            raise er

        assert self._k is None or num_objects <= self._k
        self._rows[row] -= num_objects

    def available_moves(self) -> list:
        return [(r, o) for r, c in enumerate(self.rows) for o in range(1, c + 1) if self._k is None or o <= self._k]

    def get_state_and_reward(self):
        return self.rows, self.give_reward()

    def give_reward(self):
        return 1 if sum(self._rows) == 0 else 0


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

```python
def take_one_never_finish(State: Nim) -> Nimply:
    """
    Player that always take one object without finishing a row if it is possible
    param state: Nim
    return: Nimply
    """
    try:
        move = [(r, o) for r, o in enumerate(State.rows) if o > 1][0]
    except IndexError:
        move = [(r, o) for r, o in enumerate(State.rows) if o > 0][0]
    return Nimply(*move)


def pure_random(state: Nim) -> Nimply:
    """
    Pure random player, pick a random move between all the possible moves
    param state: Nim
    return: Nimply
    """
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    num_objects = random.randint(1, state.rows[row])
    return Nimply(row, num_objects)


def nim_sum(state: Nim) -> int:
    """
    Player with the optimal strategy
    param state: Nim
    return: result: int, the nim sum of the state
    """
    *_, result = accumulate(state.rows, operator.xor)
    return result


def brute_force(state: Nim) -> dict:
    """
    Apply the nimSum to all the possible moves and return a dictionary with the possible moves and the nim sum
    param state: Nim
    return: dict
    """
    bf = list()
    data = dict()
    possible_moves = [
        (r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1) if state._k is None or o <= state._k
    ]
    for m in possible_moves:
        tmp = deepcopy(state)
        tmp.nimming(m)
        bf.append((m, nim_sum(tmp)))
    data["brute_force"] = bf
    return data


def optimal_startegy(state: Nim) -> Nimply:
    """
    Player with the optimal strategy (nimSum)
    param state: Nim
    return: Nimply
    """
    data = brute_force(state)
    return next((bf for bf in data["brute_force"] if bf[1] == 0), random.choice(data["brute_force"]))[0]


def nim_sum(state: Nim) -> int:
    """
    Perform the nim sum operation between the row's objects
    param state: Nim
    return: result: int, the nim sum operation of the state
    """
    *_, result = accumulate(state.rows, operator.xor)
    return result


def human_player(state: Nim) -> Nimply:
    """Human player
    param state: Nim
    return: Nimply
    """
    logging.info(f"\n{state}")
    row = int(input("row: "))
```

```
    num_objects = int(input("num_objects: "))
    return Nimply(row, num_objects)

@cache
def cook_data(state: Nim) -> dict:
    # print(f"rows: {state.rows}")
    bf = []
    data = {}
    possible_moves = [(r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1)]
    for m in possible_moves:
        tmp = deepcopy(state)
        tmp.nimming(m)
        # bf.append(tmp)
        bf.append((m, tmp))
    data["brute_force"] = bf
    data["possible_moves"] = possible_moves

    return data

def rl_agent(state: Nim) -> Nimply:
    possibleStates = cook_data(state)["brute_force"]
    G = json.load(open("policy.json"))
    G = {tuple(int(i) for i in k): v for k, v in G.items()}
    ply = max(((s[0], G[s[1].rows]) for s in possibleStates if s[1].rows in G), key=lambda i: i[1])[0]
    return Nimply(ply[0], ply[1])
```

Task 2 (GA)

```python
import logging
import random
import json
import os
from collections import namedtuple
from typing import Callable

import opponents
from opponents import Nim
from best_player import cook_data


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

logging.getLogger().setLevel(logging.DEBUG)

NUM_MATCHES = 50
NIM_SIZE = 5
POPULATION = 10
NUM_GENERATIONS = 25
OFFSPRING = 7


def make_strategy(genome: dict) -> Callable:
    def evolvable(state: Nim) -> Nimply:
        # data = opponents.cook_data(state)
        data = cook_data(state)
        alpha = genome["alpha"]
        beta = genome["beta"]
        gamma = genome["gamma"]
        delta = genome["delta"]
        epsilon = genome["epsilon"]

        res = (
            (a[0], abs(alpha * a[1] + beta * b[1] + gamma * c[1] + delta * d[1] + epsilon * e[1]))
            for a, b, c, d, e in zip(data["and"], data["or"], data["sum"], data["sub"], data["nand"])
        )
        ply = min(res, key=lambda x: x[1])[0]

        return ply

    return evolvable


def evaluate(strategy: Callable, opponent=opponents.pure_random) -> float:
    strategy = (strategy, opponent)
    won = 0
    for m in range(NUM_MATCHES):
        nim = Nim(random.randint(5, 12))
        player = 0
        while nim:
            ply = strategy[player](nim)
            nim.nimming(ply)
            player = 1 - player
        if player == 1:
            won += 1
    return won


def generate_population(dim: int) -> list:
    r = []
    for _ in range(dim):
        genome = {
            "alpha": random.uniform(-10, 10),
            "beta": random.uniform(-10, 10),
            "gamma": random.uniform(-10, 10),
            "delta": random.uniform(-10, 10),
            "epsilon": random.uniform(-10, 10),
        }
        strat = make_strategy(genome)
        eval = (
            evaluate(strat, opponents.optimal_startegy),
            evaluate(strat, opponents.pure_random),
            evaluate(strat, opponents.gabriele),
            evaluate(strat, opponents.take_one_never_finish),
        )
```

```
        r.append((eval, strat, genome))
    return r


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


def combine(population, offspring):
    population += offspring
    population = sorted(population, key=lambda i: i[0], reverse=True)[:POPULATION]
    return population


def generate_offspring(population: list, gen: int) -> list:
    offspring = list()
    for i in range(OFFSPRING):
        p = tournament(population)

        p[2]["alpha"] += random.gauss(0, 20 / (gen + 1))
        p[2]["beta"] += random.gauss(0, 20 / (gen + 1))
        p[2]["gamma"] += random.gauss(0, 20 / (gen + 1))
        p[2]["delta"] += random.gauss(0, 20 / (gen + 1))
        p[2]["epsilon"] += random.gauss(0, 20 / (gen + 1))

        strat = make_strategy(p[2])
        eval = (
            evaluate(strat, opponents.optimal_startegy),
            evaluate(strat, opponents.pure_random),
            evaluate(strat, opponents.gabriele),
            evaluate(strat, opponents.take_one_never_finish),
        )
        offspring.append((eval, strat, p[2]))
    return offspring


def main():
    # if you want to import a population from a previous run, set this to True
    import_population = False

    if import_population and os.path.exists("populationV8.json"):
        with open("populationV8.json", "r") as f:
            loaded_population = json.load(f)
            population = list()
            for i in loaded_population:
                genome = loaded_population[i]["genome"]
                strat = make_strategy(genome)
                eval = (
                    evaluate(strat, opponents.optimal_startegy),
                    evaluate(strat, opponents.pure_random),
                    evaluate(strat, opponents.gabriele),
                    evaluate(strat, opponents.take_one_never_finish),
                )
                population.append((eval, strat, genome))
        logging.info("loaded initial population")
    else:
        logging.info(f"generating initial population of {POPULATION} individuals")
        population = generate_population(POPULATION)
        logging.info("Generated initial population")

    for _ in range(NUM_GENERATIONS):
        logging.info(f"Generation {_}")
        offspring = generate_offspring(population, _)
        population = combine(population, offspring)
        logging.debug(f"best genome: {population[0][2]}")
        logging.debug(f"best fitness: {population[0][0]}")
        if population[0][0][0] > 20:
            break

    logging.info(f"best genome: {population[0][2]}")

    with open("populationV8.json", "w") as f:
        pop = {f"individual_{i:02}": {"fitness": p[0], "genome": p[2]} for i, p in enumerate(population)}
        json.dump(pop, f, indent=4)
    logging.info(f"saved population\nlength: {len(population)}")


if __name__ == "__main__":
```

```
main()
```

Task 3 (minmax)

```python
import logging
import random

from collections import namedtuple
from typing import Callable

from copy import deepcopy

import opponents
from opponents import Nim


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

logging.getLogger().setLevel(logging.DEBUG)

NIM_SIZE = 5


def cook_status(state: Nim) -> dict:
    cooked = dict()
    cooked["possible_moves"] = [
        (r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1) if state._k is None or o <= state._k
    ]
    return cooked


def minmax(state: Nim, player: bool, depth: int = None, memory: dict = {}) -> Nimply:
    """
    recursive minmax function
    parameters:
        state: the current state of the game
        player: True if the player is the maximizing player
        depth: the depth of the search tree
        memory: a dictionary of states and their values
    returns:
        True if the player can win from this state
        False if the player can't win from this state
    """
    if not state:
        return not player

    if (state, player) in memory:
        return memory[(state, player)]

    if depth is not None and depth == -1:
        return True

    possible_moves = cook_status(state)["possible_moves"]
    if player:
        for move in possible_moves:
            new_state = deepcopy(state)
            new_state.nimming(move)
            v = minmax(new_state, False, depth - 1 if depth is not None else None, memory)
            if v:
                # if player is maximising and v is True, then we can win from this state
                break
        if (state, player) not in memory:
            memory[(state, player)] = v
        return v
    else:
        for move in possible_moves:
            new_state = deepcopy(state)
            new_state.nimming(move)
            v = minmax(new_state, True, depth - 1 if depth is not None else None, memory)
            if not v:
                # if player is minimising and v is False, then we can't win from this state
                break
        if (state, player) not in memory:
            memory[(state, player)] = v
        return v


def minmax_choice(state: Nim, player: bool, memory: dict) -> Nimply:
    """
    returns the best move for the player implementing minmax
```

```
    parameters:
        state: the current state of the game
        player: True if the player is the maximizing player
        memory: a dictionary of states and their values
            memory -> key = (state, player)
                   -> value = True if first player can win from this state, False otherwise
    returns:
        Nimply: the best move for the player
    """
    possible_moves = cook_status(state)["possible_moves"]
    # choices = []

    # for values of NIM_SIZE > 6, we need to limit the depth of the search tree
    maxdepth = None
    for move in possible_moves:
        new_state = deepcopy(state)
        new_state.nimming(move)
        result = minmax(new_state, not player, maxdepth, memory)
        if result:
            return move
        # choices.append((result, move))

    # return next((c[1] for c in choices if c[0]), random.choice(choices)[1])


def main():
    nim = Nim(NIM_SIZE)
    player = 0
    logging.info(f"nim: {nim}")
    while nim:
        logging.info(f"player: {player} -> {nim}")
        if player == 0:
            ply = minmax_choice(nim, True, {})
        else:
            ply = opponents.pure_random(nim)
        nim.nimming(ply)
        logging.info(f"ply : {ply}")
        player = 1 - player

    if player == 1:
        logging.info(f"victory")
    else:
        logging.info(f"defeat")


if __name__ == "__main__":

    main()
```

Task 4 (RL)

Training

```
from opponents import pure_random
from RLAgent import *
from collections import namedtuple
from typing import Callable
from opponents import rl_agent
import logging
import sys
import json

NIM_SIZE = 4
NUM_MATCHES = 100
TRAINING_TIME = 50000
SAVE_POLICY = True
Nimply = namedtuple("Nimply", "row, num_objects")
logging.getLogger().setLevel(logging.DEBUG)


def evaluate(strategy: Callable, opponent: Callable = pure_random) -> int:
    strategy = (strategy, opponent)
    won = 0
    for m in range(NUM_MATCHES):
        nim = Nim(NIM_SIZE)
        player = 0
        while nim:
            ply = strategy[player](nim)
            nim.nimming(ply)
            player = 1 - player
        if player == 1:
            won += 1
    return won


if __name__ == "__main__":
    nim = Nim(NIM_SIZE)
    opponent = pure_random
    logging.debug("start training of RL Agent")
    agent = Agent(nim, alpha=0.2, random_factor=0.5)
    opponent = pure_random
    print(f"training {0} / {TRAINING_TIME}", end="")
    for t in range(TRAINING_TIME):

        stateCopy = deepcopy(nim)
        while stateCopy:
            possiblePlies = cook_data(stateCopy)["possible_moves"]
            action = agent.choose_action(stateCopy, possiblePlies)
            stateCopy.nimming(action)
            # give a 0 reward if I am winning, -10 if I am losing, and -0.5 if not in a deterministic state
            reward = -10 if sum(i > 0 for i in stateCopy._rows) == 1 else -0.5 * int(sum(stateCopy._rows) > 0)

            agent.update_state_history(stateCopy, reward)

            if sum(stateCopy._rows) == 0:
                break
            stateCopy.nimming(opponent(stateCopy))

        agent.learn()
        sys.stdout.flush()
        print(f"\rtraining {t+1} / {TRAINING_TIME}", end="")

        if SAVE_POLICY and t%5000 == 0:
            policy = agent.get_policy()
            policy_str = {"".join(str(_) for _ in k): v for k, v in policy.items()}
            json.dump(policy_str, open("policy.json", "w"))

    policy = agent.get_policy()


    print("\n")
    # policy = learning(nim)
    logging.debug("finished training\nstarting tests...")
    print("--------------------")
    for _ in range(10):
        result = evaluate(RLAgent(policy), opponent)
```

```
        logging.info(f"after {NUM_MATCHES} matches, player 0 won {result} times ==> {result / NUM_MATCHES * 100}%")
    print("--------------------")
```

Agent

```python
from collections import namedtuple

# from best_player import cook_data
from opponents import pure_random, Nim
from itertools import product
from copy import deepcopy
from functools import cache
import random


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


@cache
def cook_data(state: Nim) -> dict:
    # print(f"rows: {state.rows}")
    bf = []
    data = {}
    possible_moves = [(r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1)]
    for m in possible_moves:
        tmp = deepcopy(state)
        tmp.nimming(m)
        # bf.append(tmp)
        bf.append((m, tmp))
    data["brute_force"] = bf
    data["possible_moves"] = possible_moves

    return data


def RLAgent(G: dict) -> Nimply:
    """the agent related to the task 3.4"""

    def agent(state: Nim):
        possibleStates = cook_data(state)["brute_force"]
        ply = max(((s[0], G[s[1].rows]) for s in possibleStates if s[1].rows in G), key=lambda i: i[1])[0]
        return Nimply(ply[0], ply[1])

    return agent


class Agent(object):
    def __init__(self, state, alpha=0.15, random_factor=0.2):
        self.state_history = [(tuple(state._rows), 0)]  # state, reward
        self.alpha = alpha
        self.random_factor = random_factor
        self.G = {}
        self.init_reward(state)

    def init_reward(self, state):
        for i in product(*(range(x + 1) for x in state._rows)):
            self.G[i] = random.uniform(0.1, 1)

    def choose_action(self, state, allowedMoves):
        maxG = -10e15
        next_move = None

        randomN = random.random()
        if randomN < self.random_factor:
            random.randint(0, len(allowedMoves) - 1)
            next_move = allowedMoves[random.randrange(len(allowedMoves))]
        else:
            # if exploiting, gather all possible actions and choose one with the highest G (reward)
            for action in allowedMoves:
                stateCopy = deepcopy(state)
                stateCopy.nimming(action)
                if self.G[tuple(stateCopy._rows)] >= maxG:
                    next_move = action
                    maxG = self.G[tuple(stateCopy._rows)]

        return next_move

    def update_state_history(self, state, reward):
        self.state_history.append((tuple(state._rows), reward))

    def learn(self):
        target = 0
```

```
        for prev, reward in reversed(self.state_history):
            self.G[prev] = self.G[prev] + self.alpha * (target - self.G[prev])
            target += reward

        self.state_history = []

        self.random_factor -= 10e-5  # decrease random factor each episode of pla

    def get_policy(self):
        return self.G
```

Best players

```python
import operator
import logging
from opponents import Nim, Nimply

# from operator import and_, or_, sub
from itertools import accumulate
from copy import deepcopy

logging.basicConfig(level=logging.INFO)


def cook_data(state: Nim) -> dict:
    """
    Apply all the possible operation to the state and return a dictionary with the possible moves and the result of the operations
    param state: Nim
    return: dict
    """
    data = dict()
    and_list = list()
    or_list = list()
    sum_list = list()
    sub_list = list()
    nand_list = list()
    possible_moves = [
        (r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1) if state._k is None or o <= state._k
    ]
    for m in possible_moves:
        tmp = deepcopy(state)
        tmp.nimming(m)
        and_list.append((m, and_operation(tmp)))
        or_list.append((m, or_operation(tmp)))
        sum_list.append((m, sum(tmp.rows)))
        sub_list.append((m, sub_operation(tmp)))
        nand_list.append((m, nand_operation(tmp)))

    data["and"] = and_list
    data["or"] = or_list
    data["sum"] = sum_list
    data["sub"] = sub_list
    data["nand"] = nand_list
    # data["possible_moves"] = possible_moves

    return data


def sub_operation(state: Nim) -> int:
    """
    Perform the sub operation between the row's objects
    param state: Nim
    return: result: int, the sub operation of the state
    """
    *_, result = accumulate(state.rows, operator.sub)
    return result


def nand_operation(state: Nim) -> int:
    """
    Perform the nand operation between the row's objects
    param state: Nim
    return: result: int, the nand operation of the state
    """
    *_, result = accumulate(state.rows, lambda x, y: ~(x & y))
    return result


def and_operation(state: Nim) -> int:
    """
    Perform the and operation between the row's objects
    param state: Nim
    return: result: int, the and operation of the state
    """
    *_, result = accumulate(state.rows, operator.and_)
    return result


def or_operation(state: Nim) -> int:
```

```
    """
    Perform the or operation between the row's objects
    param state: Nim
    return: result: int, the or operation of the state
    """
    *_, result = accumulate(state.rows, operator.or_)
    return result


def five_param_5(state: Nim) -> Nimply:
    """
    My best player with 5 parameters
    param state: Nim
    return: Nimply
    """
    genome = {
        "alpha": -42.5399485484396,
        "beta": 114.60961375796023,
        "gamma": -52.64808867035252,
        "delta": 0.49668038593870456,
        "epsilon": 18.15686871650329,
    }
    data = cook_data(state)
    res = (
        (
            a[0],
            abs(
                genome["alpha"] * a[1]
                + genome["beta"] * b[1]
                + genome["gamma"] * c[1]
                + genome["delta"] * d[1]
                + genome["epsilon"] * e[1]
            ),
        )
        for a, b, c, d, e in zip(data["and"], data["or"], data["sum"], data["sub"], data["nand"])
    )
    ply, r = min(res, key=lambda x: x[1])
    logging.info(f"res: {r}")

    return ply


def four_param_5(state: Nim) -> Nimply:
    """
    My best player with 4 parameters
    param state: Nim
    return: Nimply
    """
    genome = {
        "alpha": 12.812770589035535,
        "beta": -16.051123920350758,
        "gamma": -0.20956437443764508,
        "delta": -8.234717910949916,
    }
    data = cook_data(state)
    alpha = genome["alpha"]
    beta = genome["beta"]
    gamma = genome["gamma"]
    delta = genome["delta"]

    res = (
        (a[0], abs(alpha * a[1] + beta * b[1] + gamma * c[1] + delta * d[1]))
        for a, b, c, d in zip(data["and"], data["or"], data["sum"], data["sub"])
    )
    ply = min(res, key=lambda x: x[1])[0]

    return ply


def five_param_generalized(state: Nim) -> Nimply:
    """
    My best player with 5 parameters
    param state: Nim
    return: Nimply
    """
    genome = {
        "alpha": 44.79077594400001,
        "beta": 9.579669386437583,
        "gamma": -5.209762203134689,
        "delta": -9.489475946977137,
```

```
        "epsilon": -31.18441371362716,
    }
    data = cook_data(state)
    res = (
        (
            a[0],
            abs(
                genome["alpha"] * a[1]
                + genome["beta"] * b[1]
                + genome["gamma"] * c[1]
                + genome["delta"] * d[1]
                + genome["epsilon"] * e[1]
            ),
        )
        for a, b, c, d, e in zip(data["and"], data["or"], data["sum"], data["sub"], data["nand"])
    )
    ply, r = min(res, key=lambda x: x[1])
    logging.info(f"res: {r}")

    return ply


def four_param_generalized(state: Nim) -> Nimply:
    """
    My best player with 4 parameters
    param state: Nim
    return: Nimply
    """
    # genome = {
    #     "alpha": 14.325827789108999,
    #     "beta": -5.726164045356429,
    #     "gamma": -31.566080375138124,
    #     "delta": 13.98406203443887,
    # }
    genome = {
        "alpha": 15.945309194204931,
        "beta": -3.2707966609771746,
        "gamma": -25.708257470959275,
        "delta": 14.81947128092396,
    }
    data = cook_data(state)
    alpha = genome["alpha"]
    beta = genome["beta"]
    gamma = genome["gamma"]
    delta = genome["delta"]

    res = (
        (a[0], abs(alpha * a[1] + beta * b[1] + gamma * c[1] + delta * d[1]))
        for a, b, c, d in zip(data["and"], data["or"], data["sum"], data["sub"])
    )
    ply = min(res, key=lambda x: x[1])[0]

    return ply
```

Code for Final Project (Quarto)

players

```
import random
import quarto

import pickle
import math
import numpy as np

from utility import *

RL_POLICY_FILE = "populations/policy-5000.pkl"


class GAPlayer(quarto.Player):
    """Genetic Algorithm agent"""

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

    def set_genome(self, genome):
        self.genome = genome

    def choose_piece(self) -> int:
        data = cook_data_pieces(self)
        res = (
            (a[0], abs(self.genome["alpha"] * a[1] + self.genome["beta"] * b[1]))
            for a, b in zip(data["alpha"], data["beta"])
        )
        choosen_piece = min(res, key=lambda x: x[1])[0]
        return choosen_piece

    def place_piece(self) -> tuple[int, int]:
        data = cook_data_moves(self, self.get_game().get_selected_piece())
        res = (
            (g[0], abs(self.genome["gamma"] * g[1] + self.genome["delta"] * h[1] + self.genome["epsilon"] * i[1]))
            for g, h, i in zip(data["gamma"], data["delta"], data["epsilon"])
        )
        choosen_move = min(res, key=lambda x: x[1])[0]
        return choosen_move


class TrainedGAPlayer(quarto.Player):
    """Genetic Algorithm agent"""

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        # self.genome = {
        #     "alpha": -81.66675697723781,
        #     "beta": -113.69148590004534,
        #     "gamma": -29.32357457256496,
        #     "delta": -171.92595218897023,
        #     "epsilon": -10.935275664842214,
        # }
        self.genome = {
            "alpha": -94.03014146974122,
            "beta": -107.17350875193313,
            "gamma": 152.6577141347451,
            "delta": -29.856838596915765,
            "epsilon": -12.095960806170313,
        }

    def choose_piece(self) -> int:
        data = cook_data_pieces(self)
        res = (
            (a[0], abs(self.genome["alpha"] * a[1] + self.genome["beta"] * b[1]))
            for a, b in zip(data["alpha"], data["beta"])
        )
        choosen_piece = min(res, key=lambda x: x[1])[0]
        return choosen_piece

    def place_piece(self) -> tuple[int, int]:
        data = cook_data_moves(self, self.get_game().get_selected_piece())
        res = (
```

```
            (g[0], abs(self.genome["gamma"] * g[1] + self.genome["delta"] * h[1] + self.genome["epsilon"] * i[1]))
            for g, h, i in zip(data["gamma"], data["delta"], data["epsilon"])
        )
        choosen_move = min(res, key=lambda x: x[1])[0]
        return choosen_move


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

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

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

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


class DumbPlayer(quarto.Player):
    """Dumb player"""

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

    def choose_piece(self) -> int:
        status = self.get_game().get_board_status()
        used_pieces = {c for x in status for c in x if c > -1}
        usable_pieces = [_ for _ in {x for x in range(16)} - used_pieces]
        return usable_pieces[0]

    def place_piece(self) -> tuple[int, int]:
        status = self.get_game().get_board_status()
        possible_moves = [(c, r) for r in range(4) for c in range(4) if status[r][c] == -1]
        return possible_moves[0]


class HumanPlayer(quarto.Player):
    """Human player
    A player that asks for input from the user
    """

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

    def choose_piece(self) -> int:
        status = self.get_game().get_board_status()
        used_pieces = {c for x in status for c in x if c > -1}
        usable_pieces = [_ for _ in {x for x in range(16)} - used_pieces]
        print(self.get_game().print())
        print(f"Choose a piece: {usable_pieces}")
        piece = int(input())
        while piece not in usable_pieces:
            print("Invalid piece")
            print(f"Choose a piece in : {usable_pieces}")

        return piece

    def place_piece(self) -> tuple[int, int]:
        status = self.get_game().get_board_status()
        possible_moves = [(c, r) for r in range(4) for c in range(4) if status[r][c] == -1]
        print(self.get_game().print())
        print(f"Choose a move: {possible_moves}")
        move = tuple(map(int, input().split()))
        while move not in possible_moves:
            print("Invalid move")
            print(f"Choose a move in : {possible_moves}")
        return move


class RuleBasedPlayer(quarto.Player):
    """Rule based player"""

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.previous_board = np.array([[-1, -1, -1, -1], [-1, -1, -1, -1], [-1, -1, -1, -1], [-1, -1, -1, -1]])
        self.previous_move = None
        self.previous_piece = None
```

```
    def choose_piece(self) -> int:
        pieces = block_strategy_piece(self)
        if len(pieces) < 3 and len(pieces) > 0:
            return list(pieces.keys())[0]
        piece = mirror_strategy_piece(self)

        return piece

    def place_piece(self) -> tuple[int, int]:
        move = check_for_win(self.get_game())
        if move is not None:
            return move
        move = mirror_strategy_move(self)
        return move


class MinMaxPlayer(quarto.Player):
    """Minmax player"""

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.first_move = True
        self.piece_choice = None
        self.move_choice = None
        self.memory = {}

    def choose_piece(self) -> int:
        piece = 0
        if self.first_move:
            piece = random.randint(0, 15)
        else:
            piece = self.piece_choice
        return piece

    def place_piece(self) -> tuple[int, int]:
        move = (0, 0)
        game = deepcopy(self.get_game())
        value, move, piece = minmax(self, depth=1, alpha=-math.inf, beta=math.inf, isMaximizing=True, game=game)
        self.piece_choice = piece
        self.move_choice = move
        return move

    def evaluate_board(self, isMaximizing, game, last_move, last_piece):
        """Evaluate the board and return a value
        params:
            isMaximizing: True if the player is maximizing, False otherwise
            game: the game to evaluate
            last_move: the last move made
            last_piece: the last piece used
        return:
            a tuple containing the value of the board, the last move made and the last piece used
        """
        if game.check_winner() > -1:
            return (100, last_move, last_piece)

        usable_pieces = get_available_pieces(game.get_board_status())
        blocking_pieces = blocking_piece(usable_pieces, game)
        # the value is the percentage of blocking pieces if the player is maximizing
        # the value is the percentage of non blocking pieces if the player is minimizing
        if isMaximizing:
            v = len(blocking_pieces) * 100 / len(usable_pieces)
            return (v, last_move, last_piece)
        else:
            v = (len(usable_pieces) - len(blocking_pieces)) * 100 / len(usable_pieces)
            return (v, last_move, last_piece)


class MixedStrategyPlayer(quarto.Player):
    """Mixed strategy player:
    A player that uses a mix of rule based and minmax strategies
    """

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.previous_board = np.array([[-1, -1, -1, -1], [-1, -1, -1, -1], [-1, -1, -1, -1], [-1, -1, -1, -1]])
        self.previous_move = None
        self.previous_piece = None
        self.minmax_piece = None
        self.memory = {}
```

```
    def choose_piece(self) -> int:
        """Choose a piece to play
        if the minmax player has already chosen a piece, use it
        else use block strategy
        else use mirror strategy
        """
        if self.minmax_piece is not None and self.minmax_piece in get_available_pieces(
            self.get_game().get_board_status()
        ):
            piece = self.minmax_piece
            return piece

        pieces = block_strategy_piece(self)
        if len(pieces) < 3 and len(pieces) > 0:
            return list(pieces.keys())[0]
        piece = mirror_strategy_piece(self)
        return piece

    def place_piece(self) -> tuple[int, int]:
        """Place a piece on the board
        if the minmax player has already chosen a move, use it
        else use block strategy
        else use mirror strategy
        """
        move = check_for_win(self.get_game())
        if move is not None:
            return move

        usable_pieces = get_available_pieces(self.get_game().get_board_status())
        if len(usable_pieces) < 6:
            game = deepcopy(self.get_game())
            value, move, piece = minmax(self, depth=3, alpha=-math.inf, beta=math.inf, isMaximizing=True, game=game)
            if piece != -1 and value > 0:
                self.minmax_piece = piece
                self.previous_board[move[0]][move[1]] = self.get_game().get_selected_piece()
                self.previous_piece = piece
                return move
        move = mirror_strategy_move(self)

        return move

    def evaluate_board(self, isMaximizing, game, last_move, last_piece):
        """Evaluate the board and return a value
        params:
            isMaximizing: True if the player is maximizing, False otherwise
            game: the game to evaluate
            last_move: the last move made
            last_piece: the last piece used
        return:
            a tuple containing the value of the board, the last move made and the last piece used
        """
        if game.check_winner() > -1:
            return (100, last_move, last_piece)

        usable_pieces = get_available_pieces(game.get_board_status())
        blocking_pieces = blocking_piece(usable_pieces, game)
        # the value is the percentage of blocking pieces if the player is maximizing
        # the value is the percentage of non blocking pieces if the player is minimizing
        if isMaximizing:
            v = len(blocking_pieces) * 100 / len(usable_pieces)
            return (v, last_move, last_piece)
        else:
            v = (len(usable_pieces) - len(blocking_pieces)) * 100 / len(usable_pieces)
            return (v, last_move, last_piece)


class TrainedRLPlayer(quarto.Player):
    """Trained RL player:
    A player that uses a trained RL policy
    """

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.is_learning = False
        self.G = pickle.load(open(RL_POLICY_FILE, "rb"))
        self.current_state = dict()
        self.randomness = 0.7
        self.learning_rate = 10e-3

    def choose_piece(self) -> int:
```

```
        board = self.get_game().get_board_status()
        available_pieces = get_available_pieces(board)

        if hash(str(board)) in self.G:
            try:
                possible_chioce = self.G[hash(str(board))]["piece"]
                choice = max(possible_chioce, key=possible_chioce[1])
            except:
                choice = random.choice(available_pieces)
        else:
            choice = random.choice(available_pieces)

        return choice

    def place_piece(self) -> tuple[int, int]:
        board = self.get_game().get_board_status()
        available_moves = get_available_moves(board)
        if hash(str(board)) in self.G:
            try:
                possible_chioce = self.G[hash(str(board))]["move"]
                choice = max(possible_chioce, key=possible_chioce[1])
            except:
                choice = random.choice(available_moves)
        else:
            choice = random.choice(available_moves)

        return choice


class MixedStrategyRL(quarto.Player):
    """Mixed strategy RL player:
    A player that uses a mix of rule,  RL strategies and minmax
    """

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.previous_board = np.array([[-1, -1, -1, -1], [-1, -1, -1, -1], [-1, -1, -1, -1], [-1, -1, -1, -1]])
        self.previous_move = None
        self.previous_piece = None
        self.minmax_piece = None
        self.memory = {}
        self.G = pickle.load(open(RL_POLICY_FILE, "rb"))
        self.current_state = dict()
        self.randomness = 0.7
        self.learning_rate = 10e-3

    def choose_piece(self) -> int:
        available_pieces = get_available_pieces(self.get_game().get_board_status())

        if self.minmax_piece is not None and self.minmax_piece != -1 and self.minmax_piece in available_pieces:
            minmax_piece = self.minmax_piece
            return minmax_piece

        rl_piece = self.choose_piece_rl()
        if rl_piece is not None:
            return rl_piece
        else:
            mirror_piece = mirror_strategy_piece(self)
            if mirror_piece is not None and mirror_piece in available_pieces:
                return mirror_piece
            else:
                return random.choice(available_pieces)

    def place_piece(self) -> tuple[int, int]:
        winning_move = check_for_win(self.get_game())
        if winning_move is not None:
            return winning_move

        usable_pieces = get_available_pieces(self.get_game().get_board_status())

        rl_move = self.choose_move_rl()

        if len(usable_pieces) < 6:
            game = deepcopy(self.get_game())
            value, minmax_move, minmax_piece = minmax(
                self, depth=3, alpha=-math.inf, beta=math.inf, isMaximizing=True, game=game
            )
            if value > 60 and minmax_piece != -1 and minmax_move != (-1, -1):
                self.minmax_piece = minmax_piece
                self.previous_board[minmax_move[0]][minmax_move[1]] = self.get_game().get_selected_piece()
```

```
            self.previous_piece = minmax_piece
            return minmax_move
    if rl_move is not None:
        return rl_move
    else:
        mirror_move = mirror_strategy_move(self)
        if mirror_move is not None:
            return mirror_move
        else:
            return random.choice(get_available_moves(self.get_game().get_board_status()))

def choose_piece_rl(self):
    """choose piece using rl policy"""
    board = self.get_game().get_board_status()
    if hash(str(board)) in self.G:
        try:
            possible_chioce = self.G[hash(str(board))]["piece"]
            choice = max(possible_chioce, key=possible_chioce[1])
        except:
            choice = None
    else:
        choice = None
    print(f"rl choiche piece{choice}")
    return choice

def choose_move_rl(self):
    """choose move using rl policy"""
    board = self.get_game().get_board_status()
    choice = None
    print(f"g : {self.G.keys()}")
    print(f"b : {hash(str(board))}")
    print(f"s : {str(board)}")
    print(f"len : {len(self.G)}")
    input()
    if hash(str(board)) in self.G:
        try:
            possible_chioce = self.G[hash(str(board))]["move"]
            choice = max(possible_chioce, key=possible_chioce[1])
        except:
            choice = None
    print(f"rl choiche move -> {choice}")
    return choice

def evaluate_board(self, isMaximizing, game, last_move, last_piece):
    """Evaluate the board and return a value
    params:
        isMaximizing: True if the player is maximizing, False otherwise
        game: the game to evaluate
        last_move: the last move made
        last_piece: the last piece used
    return:
        a tuple containing the value of the board, the last move made and the last piece used
    """
    if game.check_winner() > -1:
        return (100, last_move, last_piece)

    usable_pieces = get_available_pieces(game.get_board_status())
    blocking_pieces = blocking_piece(usable_pieces, game)
    # the value is the percentage of blocking pieces if the player is maximizing
    # the value is the percentage of non blocking pieces if the player is minimizing
    if isMaximizing:
        v = len(blocking_pieces) * 100 / len(usable_pieces)
        return (v, last_move, last_piece)
    else:
        v = (len(usable_pieces) - len(blocking_pieces)) * 100 / len(usable_pieces)
        return (v, last_move, last_piece)
```

GA Agent

```python
import random
import logging
import json

import quarto
from players import DumbPlayer, RandomPlayer, GAPlayer, RuleBasedPlayer

logging.getLogger().setLevel(logging.DEBUG)

IMPORT_POPULATION = False
NUM_MATCHES = 25
POPULATION = 20
NUM_GENERATIONS = 50
OFFSPRING = 10
LOG_FREQ = 10
POPULATION_FILE = "populations/population_GA_v3.json"


def fitness(genome):
    game = quarto.Quarto()
    agent = GAPlayer(game)
    agent.set_genome(genome)

    opponent = RandomPlayer(game)
    random_eval = evaluate(game, agent, opponent, NUM_MATCHES)
    game.reset()

    opponent = DumbPlayer(game)
    dumb_eval = evaluate(game, agent, opponent, NUM_MATCHES)
    game.reset()

    opponent = RuleBasedPlayer(game)
    rule_eval = evaluate(game, agent, opponent, NUM_MATCHES)

    return (rule_eval, random_eval, dumb_eval)


def generate_population(dim: int) -> list:
    r = []
    for _ in range(dim):
        genome = {
            "alpha": random.uniform(-10, 10),
            "beta": random.uniform(-10, 10),
            "gamma": random.uniform(-10, 10),
            "delta": random.uniform(-10, 10),
            "epsilon": random.uniform(-10, 10),
        }
        fit = fitness(genome)
        r.append((fit, genome))
    return r


def tournament(population, tournament_size=5):
    return max(random.choices(population, k=tournament_size), key=lambda i: i[0])


def combine(population, offspring):
    population += offspring
    population = sorted(population, key=lambda i: i[0], reverse=True)[:POPULATION]
    return population


def generate_offspring(population: list, gen: int) -> list:
    offspring = list()
    for _ in range(OFFSPRING):
        p = tournament(population)

        p[1]["alpha"] += random.gauss(0, 20 / (gen + 1))
        p[1]["beta"] += random.gauss(0, 20 / (gen + 1))
        p[1]["gamma"] += random.gauss(0, 20 / (gen + 1))
        p[1]["delta"] += random.gauss(0, 20 / (gen + 1))
        p[1]["epsilon"] += random.gauss(0, 20 / (gen + 1))
        fit = fitness(p[1])
        offspring.append((fit, p[1]))

    return offspring
```

```python
def GA():
    i = 0
    best_sol = None
    logging.info("Starting GA")
    if IMPORT_POPULATION:
        with open(POPULATION_FILE, "r") as f:
            pop = json.load(f)
            population = [(fitness(p["genome"]), p["genome"]) for p in pop.values()]
            logging.info(f"population imported")
    else:
        logging.info(f"generating population of {POPULATION} individuals")
        population = generate_population(POPULATION)
        logging.info(f"population generated")

    for _ in range(NUM_GENERATIONS):
        logging.info(f"Generation {_}")
        offspring = generate_offspring(population, _)
        population = combine(population, offspring)
        logging.debug(f"best genome: {population[0][1]}")
        logging.debug(f"best fitness: {population[0][0]}")
        if (_ + 1) % LOG_FREQ == 0:
            with open(f"populations/population_GA_v{i}.json", "w") as f:
                pop = {f"individual_{i:02}": {"fitness": p[0], "genome": p[1]} for i, p in enumerate(population)}
                json.dump(pop, f, indent=4)
                logging.info(f"saved population")
            i += 1

    best_sol = population[0][1]
    return best_sol


def evaluate(game: quarto.Quarto, player1: GAPlayer, player2: quarto.Player, n: int):
    """Evaluate the performance of a player against another player
    param:
        game: the game to be played
        player1: the first player
        player2: the second player
        n: the number of matches to be played
    return:
        the ratio of matches won by player1
    """
    win = 0
    for _ in range(n):
        game.reset()
        if _ % 2 == 0:
            game.set_players((player1, player2))
        else:
            game.set_players((player2, player1))
        winner = game.run()
        if _ % 2 == winner:
            win += 1
    return win / n


def training():
    best_genome = GA()
    game = quarto.Quarto()
    agentGen = GAPlayer(game)
    agentGen.set_genome(best_genome)
    result = evaluate(game, agentGen, RandomPlayer(game), NUM_MATCHES)
    logging.info(f"main: Winner ratio of GA: {result} -- RANDOM")
    game.reset()
    result = evaluate(game, agentGen, DumbPlayer(game), NUM_MATCHES)
    logging.info(f"main: Winner ratio of GA: {result} -- DUMB")
    game.reset()
    result = evaluate(game, agentGen, RuleBasedPlayer(game), NUM_MATCHES)
    logging.info(f"main: Winner ratio of GA: {result} -- RULE BASED")
    game.reset()


if __name__ == "__main__":
    training()
```

RL Agent

```
import logging
import random
import numpy as np
import operator as op
import quarto
import sys
import pickle

from players import RuleBasedPlayer, DumbPlayer, RandomPlayer
from utility import *

NUM_MATCHES = 5000
SAVE_POLICY = True
LOG_FREQ = 1000


class ReinforcementLearningAgent(quarto.Player):
    """A reinforcement learning agent that learns to play Quarto."""

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.is_learning = False
        self.G = dict()
        self.current_state = dict()
        self.randomness = 0.7
        self.learning_rate = 10e-3

    def get_policy(self) -> dict:
        """Returns the policy of the agent."""
        return self.G

    def set_learning(self, is_learning: bool) -> None:
        """Sets the learning mode of the agent."""
        self.is_learning = is_learning

    def choose_piece(self) -> int:
        board = self.get_game().get_board_status()
        available_pieces = get_available_pieces(board)
        available_moves = get_available_moves(board)

        if self.is_learning and random.random() < self.randomness:
            # if is learning choose a ranfdom piece and add it to the current state
            choice = random.choice(available_pieces)
            self.current_state[hash(str(board))] = {"piece": choice}
        else:
            # if not learning choose the best piece if it is in the policy
            if hash(str(board)) in self.G:
                try:
                    possible_chioce = self.G[hash(str(board))]["piece"]
                    choice = max(possible_chioce, key=possible_chioce[1])
                except:
                    choice = random.choice(available_pieces)
            else:
                choice = random.choice(available_pieces)

        return choice

    def place_piece(self) -> tuple[int, int]:
        board = self.get_game().get_board_status()
        available_moves = get_available_moves(board)
        # if is learning choose a random move and add it to the current state
        if self.is_learning and random.random() < self.randomness:
            choice = random.choice(available_moves)
            self.current_state[hash(str(board))] = {"move": choice}
        else:
            # if not learning choose the best move if it is in the policy
            if hash(str(board)) in self.G:
                try:
                    possible_chioce = self.G[hash(str(board))]["move"]
                    choice = max(possible_chioce, key=possible_chioce[1])
                except:
                    choice = random.choice(available_moves)
            else:
                choice = random.choice(available_moves)
        return choice
```

```
    def learn(self, win: bool) -> None:
        for board, value in self.current_state.items():
            # current_state = {board: {"piece": [[piece, score], ..], "move": [[move, score], ..]}}
            flag = False
            if board in self.G:  # if the board is in the policy
                # if the update is for a piece
                if "piece" in value and "piece" in self.G[board]:
                    for p in self.G[board]["piece"]:  # self.G[board]["piece"] = [[piece, score], ..]
                        if p[0] == value["piece"]:
                            if win:
                                p[1] += 1  # if win increase the score by 1
                            else:
                                p[1] -= 1  # if lose decrease the score by 1
                            flag = True
                            break
                    if not flag:  # if the piece is not in the policy add it
                        self.G[board]["piece"].append([value["piece"], 1 if win else -1])
                # if the update is for a move
                elif "move" in value and "move" in self.G[board]:
                    for p in self.G[board]["move"]:  # self.G[board]["move"] = [[move, score], ..]
                        if p[0] == value["move"]:
                            if win:
                                p[1] += 1  # if win increase the score by 1
                            else:
                                p[1] -= 1  # if lose decrease the score by 1
                            flag = True
                            break
                    if not flag:  # if the move is not in the policy add it
                        self.G[board]["move"].append([value["move"], 1 if win else -1])
            else:  # if the board is not in the policy
                if "piece" in value:
                    self.G[board] = {"piece": [[value["piece"], 1 if win else -1]]}  # add a new entry for the piece
                elif "move" in value:
                    self.G[board] = {"move": [[value["move"], 1 if win else -1]]}  # add a new entry for the move

        self.current_state = dict()  # reset the current state
        self.randomness -= self.learning_rate  # decrease the randomness


def reinforcement_training(
    game: quarto.Quarto, agent: ReinforcementLearningAgent, opponent: quarto.Player, num_matches: int
) -> None:
    win = 0
    print(f"training {0} / {NUM_MATCHES}", end="")
    for t in range(num_matches):
        game.reset()
        if t % 2 == 0:
            game.set_players((agent, opponent))
        else:
            game.set_players((opponent, agent))
        winner = game.run()
        if winner == t % 2:
            win += 1
            agent.learn(True)
        else:
            agent.learn(False)

        if SAVE_POLICY and t + 1 % LOG_FREQ == 0:
            print(f"\r\ntraining {t+1} / {NUM_MATCHES} - saving policy\n")
            policy = agent.get_policy()
            pickle.dump(policy, open(f"populations/policy-{t%LOG_FREQ}.pkl", "wb"))
        sys.stdout.flush()
        print(f"\rtraining {t+1} / {NUM_MATCHES}", end="")
    policy = agent.get_policy()
    pickle.dump(policy, open(f"populations/policy-{NUM_MATCHES}.pkl", "wb"))
    return win / num_matches


def test(game: quarto.Quarto, agent: ReinforcementLearningAgent, opponent: quarto.Player, num_matches: int) -> None:
    win = 0
    print(f"testing {0} / {NUM_MATCHES}", end="")
    for t in range(num_matches):
        game.reset()
        if t % 2 == 0:
            game.set_players((agent, opponent))
        else:
            game.set_players((opponent, agent))
        winner = game.run()
        if winner == t % 2:
            win += 1
```

```
        sys.stdout.flush()
        print(f"\rtesting {t+1} / {NUM_MATCHES}", end="")
    return win / num_matches


def train():
    game = quarto.Quarto()
    agent = ReinforcementLearningAgent(game)
    agent.set_learning(True)
    opponent = RuleBasedPlayer(game)
    winratio = reinforcement_training(game, agent, opponent, NUM_MATCHES)
    logging.info(f"\nwin ratio after traing: {winratio}")
    agent.set_learning(False)
    logging.info(f"tersting after training")
    winratio = test(game, agent, opponent, NUM_MATCHES)
    logging.info(f"\nwin ratio after testing: {winratio}")


if __name__ == "__main__":
    train()
```

utility functions

```
import quarto
import operator
import random
import math
from copy import deepcopy
import logging
from itertools import accumulate
from functools import cache


def cook_data_pieces(state: quarto.Quarto) -> dict:
    """provide usefull data for the genetic algorithm whe need to choose a piece"""
    data = {}
    status = state.get_game().get_board_status()
    used_pieces = {c for x in status for c in x if c > -1}
    usable_pieces = {x for x in range(16)} - used_pieces
    alpha = list()
    beta = list()
    if len(usable_pieces) == 1:
        # if there is only one piece left, it is the best choice
        return {"alpha": [(list(usable_pieces)[0], 1)], "beta": [(list(usable_pieces)[0], 1)]}
    for p in usable_pieces:
        usable_pieces.remove(p)
        # remove the selected piece from the usable pieces
        # calculate the sum of the remaining pieces
        *_, a = accumulate([_ for _ in usable_pieces], operator.add)
        usable_pieces.add(p)
        used_pieces.add(p)
        # add the selected piece to the used pieces
        # calculate the nand of the used pieces
        *_, b = accumulate(used_pieces, lambda x, y: ~(x & y))
        used_pieces.remove(p)
        alpha.append((p, a))
        beta.append((p, b))
    data["alpha"] = alpha
    data["beta"] = beta
    return data


def cook_data_moves(state: quarto.Quarto, piece: int) -> dict:
    """provide usefull data for the genetic algorithm whe need to choose a move"""
    data = {}
    status = state.get_game().get_board_status()
    possible_moves = [(c, r) for r in range(4) for c in range(4) if status[r][c] == -1]
    gamma = list()
    delta = list()
    epsilon = list()
    if len(possible_moves) == 1:
        # if there is only one move left, it is the best choice
        return {
            "gamma": [(possible_moves[0], 1)],
            "delta": [(possible_moves[0], 1)],
            "epsilon": [(possible_moves[0], 1)],
        }
    for p in possible_moves:
        # remove the selected move from the possible moves
        possible_moves.remove(p)
        # calculate the sum of the remaining moves
        # calculate the or of the remaining moves
        # calculate the and of the remaining moves
        *_, g = accumulate([x[0] for x in possible_moves], operator.add)
        *_, d = accumulate([x[1] for x in possible_moves], operator.or_)
        *_, e = accumulate([x[0] for x in possible_moves], operator.and_)
        possible_moves.append(p)
        gamma.append((p, g))
        delta.append((p, d))
        epsilon.append((p, e))
    data["gamma"] = gamma
    data["delta"] = delta
    data["epsilon"] = epsilon

    return data


def mirror_strategy_piece(self):
    """mirror the choice of the opponent
```

```
    if the opponent choose a piece, i choose the most different one
    """
    piece = random.randint(0, 15)

    current_board = self.get_game().get_board_status()

    if self.previous_piece == None:
        piece = random.randint(0, 15)
        # if i am plain first, i choose a random piece for the first move
    else:
        piece_info = self.get_game().get_piece_charachteristics(self.previous_piece).binary
        used_pieces = {c for x in current_board for c in x if c > -1}
        usable_pieces = [_ for _ in {x for x in range(16)} - used_pieces]
        pieces = list()
        for p in usable_pieces:
            # for each usable pieces find the most different from the previous piece
            p_info = [int(x) for x in format(p, "04b")]
            r = sum([abs(x - y) for x, y in zip(p_info, piece_info)])
            pieces.append((p, r))
            if r == 4:
                piece = p
                break
        piece = max(pieces, key=lambda x: x[1])[0]
    return piece


def mirror_strategy_move(self):
    """mirror the choice of the opponent
    if the opponent choose a move, i choose the opposite one if i can
    """
    self.previous_piece = self.get_game().get_selected_piece()
    current_board = self.get_game().get_board_status()
    # find the previous move by vomparing the current board with the previous one and find the first difference
    for i, r in enumerate(zip(self.previous_board, current_board)):
        for j, c in enumerate(zip(r[0], r[1])):
            if c[0] != c[1]:
                self.previous_move = (i, j)
                break
    if self.previous_move != None:
        # if there is a previous move, i choose the opposite one if i can
        possible_moves = [(c, r) for r in range(4) for c in range(4) if self.previous_board[r][c] == -1]
        move = (3 - self.previous_move[1], 3 - self.previous_move[0])
        if move not in possible_moves:
            # if the opposite move is not possible, i try to find a move that is as far as possible from the previous one
            # the distance is calculated usiong the manhattan distance
            manhattan = list()
            for m in possible_moves:
                v = sum(abs(x - y) for x, y in zip(move, self.previous_move))
                manhattan.append((m, v))
            move = max(manhattan, key=lambda x: x[1])[0]
    else:
        move = (random.randint(0, 3), random.randint(0, 3))
    self.previous_board = deepcopy(current_board)  # update the previous board
    self.previous_board[move[1]][move[0]] = self.previous_piece  # update the previous board with the selected move
    return move


def block_strategy_piece(self):
    """check if the are moves that can make the opponent win
    if so, return the piece that can block the move
    param:
        usable_pieces: list of pieces that can be used
        game: current game
    return:
        pieces: list of pieces that will not make the opponent win
    """
    winner = False
    current_board = self.get_game().get_board_status()
    used_pieces = {c for x in current_board for c in x if c > -1}
    usable_pieces = [_ for _ in {x for x in range(16)} - used_pieces]
    possible_moves = [(c, r) for r in range(4) for c in range(4) if current_board[r][c] == -1]
    pieces = {}
    for p in usable_pieces:
        winner = False
        for m in possible_moves:
            # if i choose a piece and the opponent can not make a winning move with that piece add it to the list
            board = deepcopy(self.get_game())

            board.select(p)
            board.place(m[0], m[1])
```

```
            if board.check_winner() > -1:
                winner = True
        if not winner:
            pieces[p] = m

    return pieces


def check_for_win(gameboard):
    """Check if the gameboard has a winning move. If so, return the move"""
    move = None

    piece = gameboard.get_selected_piece()
    current_board = gameboard.get_board_status()
    possible_moves = [(c, r) for r in range(4) for c in range(4) if current_board[r][c] == -1]
    for m in possible_moves:
        game = deepcopy(gameboard)
        game.select(piece)
        game.place(m[0], m[1])
        if game.check_winner() > -1:
            move = m
            break
    # return the list that can not make the opponent win
    return move


def blocking_piece(usable_pieces, game):
    """check if the are moves that can make the opponent win
    if so, return the piece that can block the move
    param:
        usable_pieces: list of pieces that can be used
        game: current game
    return:
        pieces: list of pieces that will not make the opponent win
    """
    winner = False
    pieces = {}
    possible_moves = [(c, r) for r in range(4) for c in range(4) if game.get_board_status()[r][c] == -1]
    for p in usable_pieces:
        winner = False
        for m in possible_moves:
            # if i choose a piece and the opponent can not make a winning move with that piece add it to the list
            board = deepcopy(game)
            board.select(p)
            board.place(m[0], m[1])
            if board.check_winner() > -1:
                winner = True
        if not winner:
            pieces[p] = m
    # return the list that can not make the opponent win
    return pieces


@cache
def minmax(self, depth, alpha, beta, isMaximizing, last_move=None, last_piece=None, game=None):
    """Minmax to choise the best move to play and piece to use"""
    if (isMaximizing and game.check_winner() > -1) or depth == 0:
        # if winning position or max depth reached
        # evaluate the position and return the value
        evaluation = self.evaluate_board(isMaximizing, game, last_move, last_piece)
        self.memory[(isMaximizing, hash(str(game)))] = evaluation
        return evaluation
    if (isMaximizing, hash(str(game))) in self.memory:
        # if the state is already solved in the memory return the value
        return self.memory[(isMaximizing, hash(str(game)))]

    best_choice = None
    selected_piece = game.get_selected_piece()
    board = game.get_board_status()
    avvailable_piece = get_available_pieces(board, selected_piece)
    available_moves = get_available_moves(board)
    if isMaximizing:
        best_choice = (-math.inf, -1, -1)
        for m in available_moves:
            game_copy = deepcopy(game)
            game_copy.place(m[0], m[1])
            for p in avvailable_piece:
                # for each move and each possible piece minimize the opponent and maximize my score
                if not game_copy.select(p):
                    logging.debug(f"piece {p} not available")
```

```
                evaluation = minmax(self, depth - 1, alpha, beta, False, m, p, game_copy)
                best_choice = max(best_choice, evaluation, key=lambda x: x[0])
                alpha = max(alpha, best_choice[0])
                if beta <= alpha or best_choice[0] == 100:  # alpha beta pruning or winning position
                    break
            if best_choice[0] == 100:
                break
        return best_choice

    else:
        best_choice = (math.inf, -1, -1)
        for m in available_moves:
            game_copy = deepcopy(game)
            game_copy.place(m[0], m[1])
            for p in avvailable_piece:
                # for each move and each possible piece minimize the opponent and maximize my score
                if not game_copy.select(p):
                    logging.debug(f"piece {p} not available")
                evaluation = minmax(self, depth - 1, alpha, beta, True, m, p, game_copy)
                best_choice = min(best_choice, evaluation, key=lambda x: x[0])
                beta = min(beta, best_choice[0])
                if beta <= alpha or best_choice[0] == 0:  # alpha beta pruning or losing position for minimizer
                    break
            if best_choice[0] == 0:
                break
        return best_choice


def get_available_pieces(board, selected_piece=None):
    """return a list of pieces that are not used in the board
    param:
        board: current board
        selected_piece: piece selected by the player
    return:
        list of pieces that are not used in the board
    """
    used_pieces = {c for x in board for c in x if c > -1}
    if selected_piece is not None:
        used_pieces.add(selected_piece)
    return [_ for _ in {x for x in range(16)} - used_pieces]


def get_available_moves(board):
    """return a list of moves that are not used in the board
    param:
        board: current board
    return:
        list of moves that are not used in the board
    """
    return [(c, r) for r in range(4) for c in range(4) if board[r][c] == -1]
```

My best player (S309413)

```python
import quarto
import numpy as np
from utility import *


class S309413(quarto.Player):
    """S309413 player"""

    def __init__(self, quarto: quarto.Quarto) -> None:
        super().__init__(quarto)
        self.previous_board = np.array([[-1, -1, -1, -1], [-1, -1, -1, -1], [-1, -1, -1, -1], [-1, -1, -1, -1]])
        self.previous_move = None
        self.previous_piece = None
        self.minmax_piece = None
        self.memory = {}

    def choose_piece(self) -> int:
        """Choose a piece to play with:
        if minmax strategy is available, use it
        if not, use the blocking strategy
        if not, use the mirror strategy
        """
        if self.minmax_piece is not None and self.minmax_piece in get_available_pieces(
            self.get_game().get_board_status()
        ):
            piece = self.minmax_piece
            return piece

        pieces = block_strategy_piece(self)
        if len(pieces) < 3 and len(pieces) > 0:
            return list(pieces.keys())[0]
        piece = mirror_strategy_piece(self)
        return piece

    def place_piece(self) -> tuple[int, int]:
        """Place a piece on the board:
        if minmax strategy is available, use it
        if not, use the blocking strategy
        if not, use the mirror strategy
        """
        move = check_for_win(self.get_game())
        if move is not None:
            return move

        usable_pieces = get_available_pieces(self.get_game().get_board_status())
        if len(usable_pieces) < 6:
            game = deepcopy(self.get_game())
            value, move, piece = minmax(self, depth=3, alpha=-math.inf, beta=math.inf, isMaximizing=True, game=game)
            if piece != -1 and value > 0:
                self.minmax_piece = piece
                self.previous_board[move[0]][move[1]] = self.get_game().get_selected_piece()
                self.previous_piece = piece
                return move
        move = mirror_strategy_move(self)

        return move

    def evaluate_board(self, isMaximizing, game, last_move, last_piece):
        """Evaluate the board:
        parameters:
            isMaximizing: bool
            game: quarto.Quarto
            last_move: tuple[int, int]
            last_piece: int
        return:
            tuple[int, tuple[int, int], int] = (value, last_move, last_piece)
        """

        if game.check_winner() > -1:
            return (100, last_move, last_piece)

        usable_pieces = get_available_pieces(game.get_board_status())
        blocking_pieces = blocking_piece(usable_pieces, game)

        # the value is the percentage of blocking pieces if isMaximizing is True
        # the value is the percentage of non blocking pieces if isMaximizing is False
```

```
    if isMaximizing:
        v = len(blocking_pieces) * 100 / len(usable_pieces)
        return (v, last_move, last_piece)
    else:
        v = (len(usable_pieces) - len(blocking_pieces)) * 100 / len(usable_pieces)
        return (v, last_move, last_piece)
```