Lab 1

NOT DONE

Lab 2

Optimal and Random agent classes

```
class Agent:
    _name = "AGENT"

    def __str__(self):
        return f"{self._name}"

    def nim_sum(self, game: Game) -> int: #nim sum
        if game.K is None:
            source = game.Rows #game rows
        else:
            source = [x % (game.K+1) for x in game.Rows] #mex

        tmp = np.array([tuple(int(x) for x in f"{c:032b}") for c in source])
        xor = tmp.sum(axis=0) % 2
        return int("".join(str(_) for _ in xor), base=2)

    def get_possible_moves(self, game: Game) -> dict:
        possible_moves = dict()
        for move in (Move(r, o) for r, c in enumerate(game.Rows) for o in range(1, c + 1)):
            if game.K is not None and move.taken > game.K:
                continue

            game_aftermove = deepcopy(game)
            game_aftermove.action(move)
            possible_moves[move] = self.nim_sum(game_aftermove) #for each move (row, taken)

        return possible_moves

class randomAgent(Agent):
    _name = "RND"

    def strategy(self, game: Game) -> Move:
        possible_moves = super().get_possible_moves(game)
        return random.choice(list(possible_moves.keys()))

class optimalAgent(Agent):
    _name = "OPT"

    def strategy(self, game: Game) -> Move:
        move = None

        possible_moves = Agent.get_possible_moves(self, game)
        optimal_moves = [move for move, nim_sum in possible_moves.items() if nim_sum == 0]
        if len(optimal_moves)>0:
            move = random.choice(optimal_moves)
        else:
            move = random.choice(list(possible_moves.keys()))

        return move
```

EA AGENT

Adopted strategy: 1. a population of individuals is created. The genotype consists of all the possible states of the game with an associated move (randomly generated). 2. To each individual a fitness score is associated. This is given by the following formula: f = #Moves_With_Resulting_NimSum_0 * 10 + #Items_removed. A little bonus is given the more items you take in order to promote faster wins. This bonus must be of a smaller magnitude wrt to the bonus given by the number of optimal moves since if not so we would have final individuals that just take the highest possible number of items per turn.
3. After some generations, mutations and crossovers the most promising individual is returned

```
class eaAgent(Agent):
    _POPULATION_SIZE = 1000
    _OFFSPRING_SIZE = 100
    _MUTATION_PROBABILITY = .3
    _TOURNAMENT_SIZE = 10
    _GENERATIONS = 200

    def __init__(self, N, k) -> None:
        self._N = N
        self._k = k
        self._name = "EA"
        self._std_genotype = self.generate_genotype(N)
        self.strongest = self.train()

    def generate_genotype(self, N):
        sets = []
        for i in range(N):
            # Each set has numbers from 0 to i*2+1
            sets.append(list(range(i * 2 + 2)))

        # Use itertools.product to generate all combinations
        all_combinations = list(product(*sets))
        all_states = [Field(c, self._k) for c in all_combinations if sum(c)>0]
        genes = dict()

        for s in all_states:
            genes[s]=None

        return genes

    def nim_sum(self, gene:tuple[Field, Move]) -> int: #nim sum
        field = deepcopy(gene[0])
        field.nimming(gene[1])

        if field.K is None:
            source = field.Rows #game rows
        else:
            source = [x % (field.K+1) for x in field.Rows] #mex

        tmp = np.array([tuple(int(x) for x in f"{c:032b}") for c in source])
        xor = tmp.sum(axis=0) % 2
        return int("".join(str(_) for _ in xor), base=2)

    def initialize_genes(self):
        geno = deepcopy(self._std_genotype)
        for g in geno:
            target = random.choice([(i,r) for i, r in enumerate(g._rows) if r>0])
            items_to_take = random.randint(1, min([target[1], self._k]))
            assert(g._rows[target[0]]>=items_to_take)
            geno[g]=Move(target[0], items_to_take)

        return geno

    @dataclass
    class Individual:
        fitness: int
        genotype: dict


    def fitness(self, genotype:dict[Field, Move]):
        score = 0
        for k,v in genotype.items():
            score += int(self.nim_sum((k,v))==0)*10 + v.taken

        return score

    def select_parent(self, pop):
        pool = random.sample(pop, self._TOURNAMENT_SIZE)
        champion = max(pool, key=lambda i: i.fitness)
        return champion

    def mutate(self, ind: Individual) -> Individual:
        offspring = deepcopy(ind)
        key = random.choice([key for key in ind.genotype.keys()])

        target = random.choice([(i,r) for i, r in enumerate(key._rows) if r>0])
        items_to_take = random.randint(1, min([target[1], self._k]))

        assert(key._rows[target[0]]>=items_to_take)
        offspring.genotype[key]=Move(target[0], items_to_take)

        offspring.fitness = self.fitness(offspring.genotype)
        return offspring

    def one_cut_xover(self, ind1: Individual, ind2: Individual) -> Individual:
        incoming_genes = random.randint(1, len(ind1.genotype.keys()))
        crossing_genes = random.sample([key for key in ind1.genotype.keys()], k=incoming_ge
        offspring = self.Individual(fitness=0, genotype=deepcopy(ind1.genotype))
        for k in crossing_genes:
            offspring.genotype[k]=deepcopy(ind2.genotype[k])

        offspring.fitness=self.fitness(offspring.genotype)

        return offspring

    def train(self) -> Individual:
        population = [ self.Individual(genotype=self.initialize_genes(), fitness=0) for _ i

        for p in population:
            p.fitness = self.fitness(p.genotype)

        print(f"Generations: {self._GENERATIONS}")
        print("Training EA...")
        for gen in range(self._GENERATIONS):
            print(f"\rG{gen+1}", end='', flush=True)
            offspring = []
            for counter in range(self._OFFSPRING_SIZE):
                if random.random() < self._MUTATION_PROBABILITY:
                    # mutation
                    p = self.select_parent(population)
                    o = self.mutate(p)
                else:
                    # xover
                    p1 = self.select_parent(population)
                    p2 = self.select_parent(population)
                    o = self.one_cut_xover(p1, p2)
                offspring.append(o)

            population.extend(offspring)
            population.sort(key=lambda i: i.fitness, reverse=True)
            population = population[:self._POPULATION_SIZE]

        print("\nDone...")

        return max(population, key=lambda i: i.fitness)

    def strategy(self, game:Game) -> Move:
        return self.strongest.genotype[game.field]
```

Lab 2 peer reviews

Sent to Filippo Greco

The optimal strategy works correctly for both classic (wins who takes the last item) and misere (looses who takes the last item). You correctly handled the final stage of the misere version in which you must not follow the strategy of making moves with resulting nim sums equal to 1 when you reach THE critical point (all remaining rows but one have just a single element left). You correctly followed the idea described in class regarding the EA strategy where each agent can play different strategies (gabriele, random and optimal) with different probabilities. Looking at the results the agents correctly tend to play the optimal strategy since the fitness function rewards who wins the most games and obviously the ones with higher probability of playing the optimal strategy will be favored. Good job!

Sent to Festa Shabani

You worked on the "misere" version of the game in which the player that takes the last item looses. The implemented optimal strategy in not complete. You should make the moves with resulting nim sum different from 0 until you reach a critical point (all remaining rows but 1 have a single item left). If you apply this change you will see that the player with optimal strategy will win more games against the random player.

About the EA approach the code follows correctly the evolutionary strategy patterns. The agents have 3 strategies available with a corresponding probability of being used in a game. The population correctly approach the optimal strategy since the fitness function promotes the agents with high number of winning games and obviously this is obtained when the optimal strategy is adopted.

It's interesting to see how we adopted 2 opposite fitness functions. You promote agents that win more games and play longer games trying to encourage resilience and adaptability whereas I try to promote shorter games and thus favoring optimal moves with the highest possible number of taken items. I think that since Nim has only 2 ways of playing (optimal or not optimal) both our “little tweaks” are not that important since the nim sum is the only thing that matters. Nevertheless it shows that we really tried to understand the problem and build from scratch a personal solution.

Well done!

Lab 9

In order to write a local-search algorithm able to solve the *Problem* instances 1, 2, 5, and 10 using a minimum number of fitness calls I adopted the following strategies:

Mutation: randomly mutate 1 gene

```
def mutate(ind: Individual) -> Individual:
    offspring = deepcopy(ind)
    pos = randint(0, K-1)
    offspring.genotype[pos] = 1 - offspring.genotype[pos]
    offspring.fitness = -1
    return offspring
```

Reorder: shuffle genes

```
def reorder(ind: Individual) -> Individual:
    offspring = deepcopy(ind)
    shuffle(offspring.genotype)
    return offspring
```

xover: randomly crossover N genes with another individual, with N randomly generated

```
def xover(ind1: Individual, ind2: Individual) -> Individual:
    how_many = randint(1, K)
    targets = sample(range(K), k=how_many)
    offspring = deepcopy(ind1)
    for t in targets:
        offspring[t] = ind2[t]
    assert len(offspring.genotype) == K
    return offspring
```

Here is presented the **main code**:

```
for s in PROBLEM_SIZE:
    fitness = make_problem(s)
    pop = deepcopy(population)
    maxF = -1
    result = "Failure"
```

```
for p in pop:
    p.fitness = fitness(p)
    if p.fitness>maxF:
        maxF = p.fitness

g=0
stillNoPerfectIndividual = True
while g < GENERATIONS and stillNoPerfectIndividual:
    offspring = list()
    for counter in range(OFFSPRING_SIZE):
        o = None

        if random() < REORDER_PROBABILITY:
            if o is not None:
                o = reorder(o)
            else:
                p = select_parent(pop)
                o = reorder(p)

        if random() < MUTATION_PROBABILITY:
            if o is not None:
                o = mutate(o)
            else:
                p = select_parent(pop)
                o = mutate(p)

        if random() < XOVER_PROBABILITY:
            # xover # add more xovers
            if o is not None:
                p2 = select_parent(pop)
                o = xover(o, p2)
            else:
                p1 = select_parent(pop)
                p2 = select_parent(pop)
                o = xover(p1, p2)

        if o is not None:
          offspring.append(o)

    for o in offspring:
        o.fitness = fitness(o.genotype)

        if o.fitness>maxF:
            maxF = o.fitness
        if maxF == 1.0:
            stillNoPerfectIndividual=False
            result = "Success"
            break

    if not stillNoPerfectIndividual:
        break

    pop.extend(offspring)
    pop.sort(key=lambda i: i.fitness, reverse=True)
    pop = pop[:POPULATION_SIZE]

    g+=1

    print(f"S={s}\t{round(g/GENERATIONS*100, 2)}%\t\tmaxFitness:{round(maxF,2)}\t\tcalls

print(f"S={s}\t\tresult:{result}\t\tmaxFitness:{round(maxF,2)}\t\tcalls:{fitness.calls}
```

parameters: maximum probability for xover, mutation and reordering led to the best results

```
POPULATION_SIZE = 300
OFFSPRING_SIZE = 30
MUTATION_PROBABILITY = 1
XOVER_PROBABILITY = 1
REORDER_PROBABILITY = 1
TOURNAMENT_SIZE = 300
GENERATIONS = 1000
PROBLEM_SIZE = [1,2,5,10]
K=1000
```

RESULTS

Problem	Result	maxFitness	Calls
S=1	Success	1.0	1217
S=2	Success	1.0	6302
S=5	Failure	0.6	30010
S=10	Failure	0.38	30010

Lab 9 peer reviews

Sent to Filippo Greco

Firstly i want to say that even if simulated anealing and the island model didn't deliver the wanted results you showed that you deeply understood them. They are solutions that just do not suit well for this problem. Regarding the tabu search, like you said in the readme, the probability of getting 2 individuals with the same genotype is too low and so it just adds computations without any valuable effect. Since we worked together on the following parts I can just confirm the effectiveness of shuffling the genes randomly and applying a crossover on random genes. Very clever of you to tarminate the program when the fitness is not improving anymore. Good job!

Sento to Lorenzo Fezza

Your implementation correclty follows the EA pattern. It is very straightforward how you computed the diversity between 2 individuals: the highest is the number of 1s in the resulting XOR operation the higher is the number of different genes. The mutation strategy adopted (0.5 probability of inverting each gene) it allows to increase the exploration maybe by reducing a bit the exploitation. About the crossover, performing it over only two points doesn't alllow to converge as fast as a n-crossover would which is beneficial when you are trying to minimize the calls to the fitness function. Probably the island model wasn't the best one to use for this kind of problem: it performes a lot of fitness calls without converging that fast to the best solution. I think that you should have played a bit more with your crossover and mutation strategies trying to find one that delivered better results. Overall very well done, managing to implement the island model shows that you deeply understood it. Good job!!

Lab 10

Overview on Q-Learning

The core idea behind reinforcement learning is that agents learn by interacting with the environment, receiving feedback in the form of rewards or punishments based on their actions.

Q-learning is a popular technique in reinforcement learning that allows agents to learn optimal actions through trial and error. The algorithm employs a value function known as the Q-function, which represents the expected cumulative reward for taking a particular action in a given state. By iteratively updating the Q-function, Q-learning enables agents to make better decisions over time.

Main classes

The class **TicTacToe** allows to play the game: we have methods that print the game field, check if a move is valid, check the winner and so on.

The class **TicTacToeRND** implements an agent playing randomly.

The class **TicTacToeLR** implements an agent adopting Q-learning. The Q-function is represented using a Q-table, a two-dimensional table where rows correspond to states, and columns correspond to actions. The Q-value for a state-action pair (s, a) in the Q-table denotes the expected cumulative reward an agent can achieve by taking action ‘a’ in state ‘s’. Initially, the Q-table is populated with zero

Training

The Q-learning algorithm follows the following iterative steps: 1. q_table is a dictionary containing the visited (states, actions) pairs. Not visisted pairs will have a 0 q-score. 2. Observe the current state of the environment. 3. Choose an action to take based on a trade-off between exploration and exploitation. This is done using an epsilon-greedy strategy, where the agent selects a random action with a certain probability and chooses the action with the highest Q-value with a complementary probability. 4. Perform the chosen action and observe the reward and the resulting next state. 5. Update the Q-value of the state-action pair using the Q-learning update rule:
Q(s, a) = (1 — A) * Q(s, a) + A * (R + Y * max(Q(s’, a’)))
A (alpha) is the learning rate, **Y** (gamma) is the discount factor that determines the importance of future rewards, **R** is the immediate reward obtained, and **max(Q(s’, a’))** represents the maximum Q-value for the next state. 6. Repeat steps 2–5 until predefined number of iterations (300_000).

The agent, while training, randomly starts first or second in order to learn playing in both situations.

Scoring

The rewards are exponentially spaced: first (action, states) pairs are rewarded less with respect to the final ones wich are more decisive.
For example if the agents wins in 4 moves the array of rewards are:
[0.01, 0.16, 0.49, 1.]
If the agent started first and the got a reward the scores are:
[0.0001, 0.0016, 0.0049, 0.01]
Wheras if it started second and got a draw:
[0.0001, 0.0114, 0.0413, 0.09]
I reward more a draw in the case the agent starts second since starting second is a disadventageous condition.

Validation

In order to choose the best hyper-parameters values the validate function tried every possibile combination. Among all the best options the following configuraiton was chosen [.5, .7, .1] for learning rate, discount factor and exploration probability.

Agent code

```python
class TicTacToeRL:
    def __init__(self, learning_rate=0.5, discount_factor=0.7, exploration_prob=0.1):
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_prob = exploration_prob

        # Q-table to store Q-values for state-action pairs
        self.q_table = {}

    def get_state_key(self, state):
        return str(state)

    def get_valid_actions(self, state):
        # Return a list of valid actions for the given state
        a =  [(i, j) for i in range(3) for j in range(3) if state[i][j] == 0]
        return a

    def get_q_value(self, state, action):
        state_key = self.get_state_key(state)
        return self.q_table.get((state_key, action), 0.0)

    def update_q_value(self, state, action, new_value):
        state_key = self.get_state_key(state)
        self.q_table[(state_key, action)] = new_value

    def choose_action(self, state, testing):
        # Epsilon-greedy strategy for action selection
        if random.uniform(0, 1) <= self.exploration_prob and testing == 0:
            return random.choice(self.get_valid_actions(state))
        else:
            # Choose the action with the highest Q-value
            q_values = [self.get_q_value(state, action) for action in self.get_valid_actions
            max_q_value = max(q_values, default=0.0)
            best_actions = [action for action, q_value in zip(self.get_valid_actions(state)
            return random.choice(best_actions)

    def train(self, state, action, reward, next_state):
        cur_qv = self.get_q_value(state, action)
        max_qv_next = max([self.get_q_value(next_state, next_action) for next_action in self

        new_qv = (1-self.learning_rate)*cur_qv + self.learning_rate * (reward + self.discou
        self.update_q_value(state, action, new_qv)

def tic_tac_toe_rl_train():
    # Initialize the RL model
    agents = (TicTacToeRL(0.5, 0.7, 0.1), TicTacToeRL(0.5, 0.7, 0.1))
    # Training loop
    num_episodes = 300_000
```

```
for ep in range(num_episodes):
    if ep < num_episodes-1:
        print(f"TRAINING: {ep+1}/{num_episodes}", end="\r")
    else:
        print(f"TRAINING COMPLETED\n")

    history = ([],[])
    start = random.choice([0,1])
    turn = start
    cntTurn = 0
    game = TicTacToe()
    state = [[0,0,0],[0,0,0],[0,0,0]]
    while True:
        # Player 1 (RL agent) move
        action = agents[turn].choose_action(state,0)
        game.make_move(action[0], action[1])
        history[turn].append((state, action, game.board))

        cntTurn+=1

        if cntTurn>=5:

            # Check for a winner or a tie
            winner = game.check_winner()

            if winner:
                loser = 1 if winner == 2 else 2
                n=len(history[winner-1])

                for i, h in enumerate(history[winner-1]):
                    scores = linspace(0.1,1,num=n)**2
                    agents[winner-1].train(h[0], h[1], scores[i], h[2])

                n=len(history[loser-1])
                for i,h in enumerate(history[loser-1]):
                    scores = linspace(-.1,-.1,num=n)**2
                    agents[loser-1].train(h[0], h[1], scores[i], h[2])

                break

            if game.is_board_full():
                n=len(history[start])
                for i, h in enumerate(history[start]):
                    scores = linspace(-.1,-.5,num=n)**2
                    agents[start].train(h[0], h[1], scores[i], h[2])
                    n=len(history[1-start])
                    for i, h in enumerate(history[1-start]):
                        scores = linspace(-.01,-3.,num=n)**2
                        agents[1-start].train(h[0], h[1], scores[i], h[2])

                    break

            state = deepcopy(game.board)

            turn = 1 - turn

    return agents[0]

def tic_tac_toe_rl_train_random(x):
    # Initialize the RL model
    agents = (TicTacToeRL(*x), TicTacToeRND())

    # Training loop
    num_episodes = 100_000
    for ep in range(num_episodes):
        if ep <num_episodes-1:
            print(f"TRAINING: {round((ep+1)/num_episodes*100, 2)}%", end="\r")
        else:
            print(f"TRAINING COMPLETED!\n")

        game = TicTacToe()
        state = [[0,0,0],[0,0,0],[0,0,0]]
        history = ([],[])
        start = random.choice([0,1])
        turn = start
        cntTurn = 0
        while True:
            # Player 1 (RL agent) move
            action = agents[turn].choose_action(state, 0)
            game.make_move(action[0], action[1])
            history[turn].append((state, action, game.board))

            cntTurn+=1

            if cntTurn>=5:

                # Check for a winner or a tie
                winner = game.check_winner()

                if winner == 1:
                    n=len(history[0])
                    for i, h in enumerate(history[0]):
                        scores = linspace(.1,1,num=n)**2
                        agents[0].train(h[0], h[1], scores[i], h[2])

                    break

                if winner == 2:
                    n=len(history[0])
                    for i, h in enumerate(history[0]):
                        scores = linspace(-.01,-.1,num=n)**2
                        agents[0].train(h[0], h[1], scores[i], h[2])

                    break

                if game.is_board_full():
                    n=5
                    scoreMAX = 0.1 if start == 0 else 0.3
                    scores = linspace(.01,scoreMAX,num=n)**2
                    for i, h in enumerate(history[0]):
                        agents[0].train(h[0], h[1], scores[i], h[2])

                    break

            state = deepcopy(game.board)

            turn = 1 - turn

    return agents[0]
```

Result

On 1000 matches the agent was able to win 98% of the times starting first and 96% of the times starting second

Lab 10 peer reviews

Sent to to Filippo Greco

Overview Your RL strategy consists in assigning a value to the different states with a score that is bigger or lower depending on wether the agent won or lost. The formula you used for updating the state values is the following: **Q(s) = Q(s) + A * (Y * R - Q(s))**
A (alpha) is the learning rate, **Y** (gamma) is the decay factor that determines the importance of future rewards, **R** is the immediate reward obtained and **Q(s)** is the value of the associated state **s**

Each time the agent plays it take the action that will bring him to the state with higher value (it saves the couples state and state value in a dictionary).

Final comment Your final score is impressive, reaching a winning rate that is $> 97\%$. Very good job, I cannot find any critique since you solution is woking really well.

Quixo

Q-Learning - 1st attempt

Since i have worked on q_learning for the tictactoe lab I decided to try it on quixo.
Ufortunately I kept getting very modest results vs the random player (around 55% of winnings) so I tried to apply some heuristics in order to help the agent to choose the best move: give a bonus when 4 consecutive pieces have been placed, when a 4-consecutive piece of the opponent is disrupted and give a score to each single state in order to promote better moves but still no improvement at all. I sterted to think that maybe the low performances were due to the enourmous number of possibe states of the game. So I last tried to take into consideration also equivalent states (rotations of the board, flipping u/d and l/r and inverting the pieces) but still no significant boost of performances. So I decided to change completely the approach and I opted for the minimax algorithm with the alpha beta pruning optimization. I wasn't too concerned to change strategy since minimax is relatively fast to implement and I could bring the heuristics I studied when implementing Q-Learning into the new model. I left the code inside the folder "other_attempts" in order to give an idea to the work I have done. It shouldn't run since there have been updates to the game.py file and I have not kept the code udpated.

Minimax with Alpha Beta pruning - final attempt

I immediately got very high results with a first version versus the random player and so I decided to keep going and improve the performances of the code.

Key points

Evaluation: Here there is the code of the evaluation function:

```
def evaluate(self, board):
        def get_score(id):

            base = 3
            score = 0

            boardt = board.T
            boardf = npfliplr(board)
            #rows and columns
            r0, rt0, r1, rt1, r2, rt2, r3, rt3, r4, rt4 = board[0], boardt[0], board[1], boa
            #main and second diagonals
            md = [board[0][0], board[1][1], board[2][2], board[3][3], board[4][4]]
            sd = [boardf[0][0], boardf[1][1], boardf[2][2], boardf[3][3], boardf[4][4]]

            #rows
            factor = npsum(r0 == id)
            score += factor*base**factor
            factor = npsum(r1 == id)
            score += factor*base**factor
            factor = npsum(r2 == id)
            score += factor*base**factor
            factor = npsum(r3 == id)
            score += factor*base**factor
            factor = npsum(r4 == id)
            score += factor*base**factor

            #columns
            factor = npsum(rt0 == id)
            score += factor*base**factor
            factor = npsum(rt1 == id)
            score += factor*base**factor
            factor = npsum(rt2 == id)
            score += factor*base**factor
            factor = npsum(rt3 == id)
            score += factor*base**factor
            factor = npsum(rt4 == id)
            score += factor*base**factor

            #main diagonal
            factor = npsum(md == id)
            score += factor*base**factor

            #diagonale secondaria
            factor = npsum(sd == id)
            score += factor*base**factor

            return score

        my_bonus, opp_bonus = get_score(self.player_id), get_score(1-self.player_id)

        return my_bonus-opp_bonus
```

The idea is to give a bonus for each group of pieces owned in rows, columns and diagonals which exponentially increases with the number of pieces. Doing so promotes having more pieces in the board but mainly reaching 5 consecutive pieces. We calculate with the same strategy also the score of the opponent and we subtruct it to the score of the player. In this way we promote winning and keeping the opponent in a loosing state. Positive scores are indicating a winning state for my agent whilst negative score indicate a winning state for the opponent The code isn't scalable but I written it in this way since I tried to make it faster since this function will bee called many times

minimax algorithm: here we can see the implementation of the minimax algoirthm. I tried various settings and the following proved to be the most efficient. The terminal conditions for the recursive functions are either one of the 2 players won or the maximum depth is reached. I iterate through the moves and generate the state that comes with the current move: doing so i avoid generating states (which is computationally expensive) that i don't need to generate when i'm not traversing some subtree thanks to alpha beta pruning. The minimax function returns both with the state value the related move. In this way it immediately and elegantely returns the action to take at the end. If it is the first time playing we just make a move.

```
def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:

        if self.start==1:
            optimal_move = self.get_first_move(game._board, self.player_id)
            self.start=0
        else:
            root = self.Node(game._board, None)
            _, optimal_move = self.minimax(root, self.depth, float('-inf'), float('inf'), Tr

        return tuple(reversed(optimal_move[0])), optimal_move[1]

def minimax(self, node, depth, alpha, beta, maximizing):
        # check if opponent won
        if self.check_if_winner(node.board, 1-self.player_id):
            return float("-inf"), node.move

        # check if I won
        if self.check_if_winner(node.board, self.player_id):
            return float("inf"), node.move

        # when max depth is reached return the score obtained with corresponding move
        if depth == 0:
            return self.evaluate(node.board), node.move

        # if maximizing the current player is myself, if not the current player is the oppo
        player_id = self.player_id if maximizing else 1-self.player_id
        proto_game = Game()
        moves = self.get_valid_moves(node.board, player_id)

        best_eval = float('-inf') if maximizing else float('inf')
        optimal_move = node.move

        for m in moves:
            proto_game._board = copy(node.board)
            proto_game._Game__move(tuple(reversed(m[0])), m[1], player_id)
            child = self.Node(proto_game._board, m)

            eval, _ = self.minimax(child, depth - 1, alpha, beta, not maximizing)

            if maximizing and eval > best_eval:
                optimal_move = child.move
                best_eval = eval

                alpha = max(alpha, best_eval)
            elif (not maximizing) and eval < best_eval:
                best_eval = eval
                optimal_move = child.move

                beta = min(beta, best_eval)

            if beta <= alpha:
                break

        return (best_eval, optimal_move)
```

get_valid_moves: in order to get the valid moves i used the following lines of code:

```
pool_moves = [
    *itproduct([(0,1),(0,2),(0,3)], [Move.BOTTOM, Move.LEFT, Move.RIGHT]),
    *itproduct([(4,1),(4,2),(4,3)], [Move.LEFT, Move.RIGHT, Move.TOP]),
    *itproduct( [(1,0),(2,0),(3,0)],  [Move.BOTTOM, Move.TOP, Move.RIGHT]),
    *itproduct([(1,4),(2,4),(3,4)], [Move.BOTTOM, Move.LEFT, Move.TOP]),
    ((0,0),Move.BOTTOM), ((0,0),Move.RIGHT), ((0,4),Move.BOTTOM), ((0,4),Move.LEFT), ((4,0)
]

def get_valid_moves(self, board, player_id):
    return [x for x in pool_moves if board[x[0]] == -1 or board[x[0]] == player_id]
```

pool_moves contains all the possible moves one can make (44 in total) and then with **get_valid_moves** we get the moves that we can do given the board sate and our player_id

results vs random player: 100% win rate