Report Computational Intelligence

Laboratory #1 : Set-covering A*

Lab 1.0

This Lab (1.0) was written with the collaboration of

- Riccardo Renda (310383)

The state represents the current configuration of the problem. In this case, it consists of two sets: taken (sets that have been chosen) and not_taken (sets that are yet to be chosen). The goal state is reached when all tiles are covered by the chosen sets.

```
PROBLEM_SIZE = 50
NUM_SETS = 100
SETS = tuple(
    np.array([random() < 0.3 for _ in range(PROBLEM_SIZE)])
    for _ in range(NUM_SETS)
)
State = namedtuple('State', ['taken', 'not_taken'])
```

The goal was to create an A* algorithm that incorporates the actual cost function, denoted as g(), which is determined by the length of sets chosen in the current state. Additionally, a new heuristic cost function, h(), was introduced. This heuristic cost is calculated as the ratio of uncovered tiles to the total number of sets.

```
def goal_check(state):
    return np.all(reduce(
        np.logical_or,
        [SETS[i] for i in state.taken], #for somma le colonne scorrendo
i esimi riga x riga, fa le 2 righe insieme tipo e vede a colonne
        np.array([False for _ in range(PROBLEM_SIZE)]),
    ))

def g(state):
    return len(state.taken)

def h(state):
    sets_tiles = reduce(
            np.logical_or,
            [SETS[i] for i in state.taken],
            np.array([False for _ in range(PROBLEM_SIZE)]),
```

```
        )
    false_tiles = [f for f in sets_tiles if f==False]
    return len(false_tiles)

def cost(state):
    return g(state)+h(state)
```

A priority queue is used to manage the states during the search. States with lower total costs are dequeued and explored first. This ensures that the algorithm explores the most promising states based on the combined cost function.

The A* search loop iterates until the goal state is reached. In each iteration, it dequeues the state with the lowest total cost from the priority queue (frontier). It then generates successor states by applying possible actions (choosing sets) and enqueues them in the priority queue.

```
frontier = PriorityQueue()
#frontier = SimpleQueue()
state = State(set(), set(range(NUM_SETS)))
frontier.put((cost(state), state))

counter = 0
_, current_state = frontier.get()
while not goal_check(current_state):
    counter += 1
    for action in current_state[1]:
        new_state = State(
            current_state.taken ^ {action},
            current_state.not_taken ^ {action},
        )
        frontier.put((cost(new_state), new_state))
    _, current_state = frontier.get()

print(
    f"Solved in {counter:,} steps ({len(current_state.taken)} tiles)"
)
```

The algorithm prints the number of steps (counter) taken to reach the goal and the final state that satisfies the goal condition. The final state includes the sets that have been taken to cover all elements.

Lab 1.1

The goal_check function checks if the current state satisfies the goal condition, i.e., if the union of the selected sets covers all elements in the problem set.

```
# To check if we finished
def goal_check(state):
    # What happens in case of no index? Add initial value
    return np.all(reduce(np.logical_or, [SETS[i] for i in state[0]],
np.array([False for _ in range(PROBLEM_SIZE)])))


assert goal_check((set(range(NUM_SETS)), set())), "Problem not
solvable"
# Sanity check => If i select all sets i must have a solution otherwise
the problem is not solvable
```

The first implementation of the Distance function aims to estimate the remaining cost to reach the goal.

It computes the number of additional sets needed to cover the remaining area, taking into account the sets already selected. The algorithm considers sets sorted in descending order of the area they cover, attempting to minimize the number of sets needed.

```
# The quality of a solution depends on the number of selected tiles so
for each state we compute the cost as the number of selected tiles

def Cost(state):

    return len(state[0])

# Here the distance code seen at lesson

def Distance(state):

    already_covered = reduce(

        np.logical_or,

        [SETS[i] for i in state[0]],

        np.array([False for _ in range(PROBLEM_SIZE)]),

    )
```

```
    if np.all(already_covered):

        return 0

   missing_size = PROBLEM_SIZE - sum(already_covered)

    candidates = sorted((sum(np.logical_and(s,
np.logical_not(already_covered))) for s in SETS), reverse=True)

    taken = 1

    while sum(candidates[:taken]) < missing_size:

        taken += 1

    return taken
```

The second implementation simplifies the heuristic, still estimating the remaining cost.

It computes the area that still needs to be covered and, for each not selected set, calculates the number of uncovered elements it can potentially cover. If the best set already covers everything remaining, the cost is estimated as 1; otherwise, at least 2 sets are needed.

```
def Distance(state):
    # Compute already covered area
    already_covered = reduce(
        np.logical_or,
        [SETS[i] for i in state[0]],
        np.array([False for _ in range(PROBLEM_SIZE)]),
    )

    to_cover = np.logical_not(already_covered)

    # Compute the candidates from the not taken group
    candidates = [sum(np.logical_and(SETS[i], to_cover)) for i in
state[1]]
    best = max(candidates)

    if(best == (PROBLEM_SIZE - sum(already_covered))):
        # We are finished by selecting the best one
        return 1
    else:
        # We need at least two
        return 2
```

Priority queue is used in the A* algorithm to prioritize states based on their total cost (Cost + Distance). States with lower total costs are explored first, improving the efficiency of the A* Algorithm.

The algorithm iteratively explores states until the goal is reached. In each iteration, the algorithm expands the current state by considering possible actions (selecting a set not taken) and adds the resulting states to the priority queue.

The priority queue ensures that states with lower total costs are explored first, guiding the search toward an optimal solution.

The choice of using a priority queue (PriorityQueue) ensures efficient exploration of states with lower costs, which is essential for the A* algorithm.
The heuristic functions (Distance) are designed to estimate the remaining cost efficiently, guiding the search toward promising paths without exhaustively exploring all possibilities.
The algorithm provides insights into the trade-off between the quality of the solution (actual cost) and the efficiency of reaching the goal (heuristic cost).

```
# Path search implementation

frontier = PriorityQueue()

initial_state = (set(), set(range(NUM_SETS)))

# Initial state => no selected sets, all available index

# For starting we add in the frontier the initial state

frontier.put((0, initial_state))

(_, state) = frontier.get()

counter = 0

while not goal_check(state):

    counter += 1

    print(f"{counter}: selected state: {state} with cost {Cost(state)}
and distance {Distance(state)}")

    for action in state[1]: # Compute all the successors and add them
to the queue

        new_state = (state[0] | {action}, state[1] - {action})
```

```
  # For the distance we consider both cost and the heuristic

        frontier.put((Cost(new_state) + Distance(new_state),
new_state))



    (_, state) = frontier.get() # New state



print(f"Solve in {counter+1} step by taking sets {state[0]} with cost
{Cost(state)}")
```

Algorithm prints the step-by-step progress, including the selected state, its cost, and the distance (heuristic estimate). Once the goal is reached, the algorithm prints the number of steps taken and the final state.

The two versions of the Distance function provide different heuristic estimates. The revised version aims to simplify the heuristic computation while still providing a reasonable estimate of the remaining cost.

Laboratory #2 : Nim GA

From this laboratory on (including the project) we have always worked in a team of 4 people:

- Giacomo Fantino (s310624)
- Giacomo Cauda (s317977)
- Lorenzo Bonannella (s317985)
- Farisan Fekri (to_add)

The objective of the lab was using GA to solve the Nim game.
We were given the **Nim class** and some opponents : **Random**, **Gabriele** and **Optimal.**

Let's begin by looking at our fitness function. To avoid a flat landscape each agent plays 18 rounds where 6 are against the Random Player, 6 against Gabriele and 6 against the Optimal.
Also if an agent loses, instead of giving him 0 points, we increase the score depending on the number of moves played.
Between two losing agents we prefer the one with the **maximum** number of moves played.ù

The following Hyperparameters have been chosen after some trial and error approach:

```
POPULATION_SIZE = 20
OFFSPRING_SIZE = 20
TOURNAMENT_SIZE = 3  #number of players in the game
MUTATION_PROBABILITY = .15 #prob. of an individual of being mutated
```

```
NUM_ROWS = 5  #number rows of stick in the game
NUM_MOVES = sum([i * 2 + 1 for i in range(NUM_ROWS)])//2
UPPERBOUND_K = 3 #max number of stick that a player can select

def fitness(genotype):
    strategy = (pure_random, gabriele, optimal)
    score = 0
    for i in range(18): #number of rounds
        nim = Nim(5)
        index = 0
        player = 0
        num_moves = 0
        while nim:
            if player == 0:
                ply = genotype[index]
                index += 1

                while nim._rows[ply[0]] < ply[1]:
                    #spare move
                    ply = (randint(0, NUM_ROWS-1), 1) #spare move

                num_moves += 1
            else:
                ply = strategy[i%3](nim) #pick which strategy we are
fighting this time

            nim.nimming(ply)
            player = 1 - player

        if player == 0: #we won
            score = score + 10
        else:
              #some points ==> if we played a lot of moves is better
than losing after a couple of moves
            score = score + num_moves*0.01 #between 0 and 10
    return score
```

We decided to code the ***genotype*** of an individual as a list of tuples of integers.
Each tuple represents in the first value the selected row, and in the second value the number of sticks that the Individual wants to take.

min(randint(1, row * 2 + 1), UPPERBOUND_K) → The number of sticks that we want to pick is minimum between: an upper bound value (which is the max number of stick that we

can take, in every selection) and a random number between 1 and the maximum number of sticks for that line

```
@dataclass
class Individual:
    fitness: int
    genotype: list[(int, int)]

population = [
    Individual(
        genotype=[],
        fitness=None,
    )
    for _ in range(POPULATION_SIZE)
]

for i in population:
    for _ in range(NUM_MOVES):
        row = randint(0, NUM_ROWS-1)
        i.genotype.extend(
            [
                (row, min(randint(1, row * 2 + 1), UPPERBOUND_K))
            ]) #what we are doing here, it is avoiding situations like
picking 5 elements from row 1 which has at most 1 element
    i.fitness = fitness(i.genotype)
```

For the parent selection we do a regular tournament where we select the parent with the maximum fitness.

For the mutation, we don't change the row in the genotype, but only the number of sticks by adding or removing 1.

For the crossover, we select a random point in the genotype and then we apply one cut crossover.

```
def select_parent(pop):
    pool = [choice(pop) for _ in range(TOURNAMENT_SIZE)]
    champion = max(pool, key=lambda i: i.fitness)
    return champion

def mutate(ind: Individual) -> Individual:
    offspring = copy(ind)
    pos = randint(0, len(offspring.genotype)-1)
```

```
    temp = 0
    while temp <= 0 or temp > UPPERBOUND_K:
        temp = offspring.genotype[pos][1] + choice([-1, 1])
    offspring.genotype[pos] = (offspring.genotype[pos][0], temp)
    offspring.fitness = None
    return offspring


def one_cut_xover(ind1: Individual, ind2: Individual) -> Individual:
    cut_point = randint(0, len(ind1.genotype))
    offspring = Individual(fitness=None,
genotype=ind1.genotype[:cut_point] + ind2.genotype[cut_point:])
    return offspring
```

Speaking of the implementation of the Genetic Algorithm, we used the usual schema.
For creating new offsprings we used a MUTATION_PROBABILITY fixed at 0.15.

```
for generation in range(100):
    offspring = list()
    for counter in range(OFFSPRING_SIZE):
        if random.random() < MUTATION_PROBABILITY:  # self-adapt
mutation probability
            # mutation  # add more clever mutations
            p = select_parent(population)
            o = mutate(p)
        else:
            # xover # add more xovers
            p1 = select_parent(population)
            p2 = select_parent(population)
            o = one_cut_xover(p1, p2)
        offspring.append(o)

    for i in offspring:
        i.fitness = fitness(i.genotype)
    population.extend(offspring)
    population.sort(key=lambda i: i.fitness, reverse=True)
    population = population[:POPULATION_SIZE]
    print(population[0].fitness)
```

With the following code we tested the best player:

```
#let's see the results against a random player
wins = 0
```

```
for _ in range(100):
    index = 0
    player = 0
    nim = Nim(5)
    while nim:
        if player == 0:
            ply = best_agent.genotype[index]
            index += 1

            while nim._rows[ply[0]] < ply[1]:
                #spare move
                ply = (randint(0, NUM_ROWS-1), 1) #spare move
        else:
            ply = pure_random(nim)

        nim.nimming(ply)
        player = 1 - player

    if player == 0:
        wins += 1

print(f'we won {wins} time')
```

And got an average win time of 70% over 100 matches.
Giving 10 points for each match won (on a maximum of 18 wins) the best value of the fitness function that we reached is 170.11.

Peer reviews received

By Andrea Sillano

The code is clear and self explanatory, I liked the way in the fitness function the time spent playing is considered since it wasn't required to the win with the least amount of moves. This may exclude more efficient solution but it could find alternative strategies.

In my opinion the tournament selection is pretty good and it always choose the strongest individual keeping the fitness high.
It would have been nice to present some statistics/performances to understand how the individual compete against on the the provide strategies.

Overall the implementation seems good.
Good luck with the next lab!

By Festa Shabani

The way the Evolutionary Algorithm is set up seems good. It utilizes basic evolutionary principles to evolve strategies.
I like that you rewarded the system to encourage longer games instead of shorter games and faster wins.
One thing I would recommend, is to put some graphs so I can see how the algorithm evolves.

By s316467

The code you've developed for playing Nim using both expert systems and evolutionary strategies showcases a commendable level of skill and understanding. What stands out is the structured approach you've taken. The clear division into different sections and the use of specific classes for game elements like Nim and Nimply not only enhances readability but also adds a modular charm to the code. The various strategies implemented, ranging from the randomness of pure_random to the calculated moves of optimal, reflect a deep grasp of game tactics. Particularly intriguing is the use of evolutionary strategies. This method, with its promise of learning and adaptation, brings a layer of sophistication to the AI agents.

However, this promising work could be elevated with a few adjustments. The code, rich in logic and strategy, would greatly benefit from increased documentation and comments. This addition would not only aid others in navigating through the complexities of your logic but also serve as a valuable reference for you in the future. While the evolutionary strategies are a highlight, they present an opportunity for refinement. The current mutation strategy, for instance, is somewhat basic and could be enhanced with more intricate approaches. Experimenting with parameters such as population size and mutation probability might also yield more effective learning outcomes.

Moreover, your testing regimen, though robust, could be expanded to pit the AI against a wider array of strategies. This would provide a more comprehensive evaluation of the agents' capabilities. On the performance front, there seems to be room for optimization, particularly in the evolutionary sections like the fitness calculation. Such enhancements could significantly streamline the code's efficiency.

Lastly, incorporating robust error handling and input validation would make your code more resilient, especially when faced with unexpected or invalid inputs.

In summary, your approach to developing Nim-playing agents is quite impressive. The careful structuring, the diverse strategies, and the innovative use of evolutionary strategies all speak to a thoughtful and skilled approach. With a bit more refinement in documentation, strategy complexity, and performance optimization, your project could reach new heights of efficiency and effectiveness. Your work is not just a demonstration of coding proficiency but also a testament to creative problem-solving in AI.

By Massimo Porcheddu

The code is clean and seems well commented (although I would have written the comments only in English due to the nature of the course)

The strategy you implemented actually uses ES but your genotype are moves so I don't understand how it is possible to predict how many moves are needed to finish a game.

Furthermore, it seems to me that these moves do not refer to the current situation of the nim as they are calculated a priori, therefore the final result seems to me to be somewhat like a random strategy.

Finally I tried to run it to understand the win rate (I recommend you enter it to test it next time) but it wasn't very easy

I hope this review can be useful to you!

By lucabubi

Hello there 👋,

First of all I wanna thank you for sharing your code with us.

You've added a reasonable amount of comments to the code, which is a good thing.

The algorithm you provided seems fine but I would suggest you some changes regarding different aspects of the code.

I suggest changing the line

NUM_MOVES = sum([i * 2 + 1 for i in range(NUM_ROWS)])//2

to

MAX_NUM_MOVES = sum([i * 2 + 1 for i in range(NUM_ROWS)])//2

for better understanding. This reflects a limit-case because it's not typical for both your algorithm and the opponent to remove only one object per turn in each round.

Regarding the _**fitness**_ function, I geniunely appriciate the change of strategy stated by

ply = strategy[i%3](nim)

because that ensures a rotational pattern for the oppent's strategy selection.

I also like a lot this part

if player == 0:

 score = score + 10

else:

 score = score + num_moves*0.01

in which you seem to give importance to the number of moves. A higher number of moves may contribute to a higher score, reflecting a preference not only for winning strategies but also for strategies that involve more moves in a game.

I suggest you to change

for i in range(18):

maybe into

NUM_ROUNDS = 18

for i in range(NUM_ROUNDS)

alwayss for better comprehension.

I also would have suggested you to increase this number but it would be probabily useless because I see a "main problem" I think you should take into consideration:

I notice that your algorithm's moves are determined **prior to the beginning of the game**, as stated by the following code

```
for _ in range(NUM_MOVES):

 row = randint(0, NUM_ROWS-1)

 i.genotype.extend([(row, min(randint(1, row * 2 + 1), UPPERBOUND_K))])
```

without considering the current state of the game or the moves executed by your opponent. These moves are not adjusted based on these parameters during each turn.

So, basically, your algorithm just randomly makes moves without considering what's happening in the game or what the opponent is doing.

With that said, I think you've done a good job, although there's room for improvement.

Good luck for the next labs! 👋

Peer reviews issued

For LeoDardanello

- This code implements a rule as the professor suggested in the customization of the move function. I personally think this an advanced choice as the professor's choose to have one sigma per each individual. It seems to update the performance of the corresponding solution. If you consider the population as a list of percentage of the probabilities then they are not independent because no matter what is the percentage of the first element is, the other elements are constraint to have the rest of the percentage at most, so it was a good choice to disjoint them.

- Therefore; when it is mutating to create a new offspring, all the probabilities are completely free and they are not bounded so is the sigma. Then it chooses a random number between the length of the first individual which is the number of move the player can have. In the end, it has one probability for each move which is remarkable.

- It was considered to have the move operation on different rows. Each move calculates the row to do the move and in the end it returns an index object which is the number of objects to remove - the peer was careful that the player shouldn't make 0 moves which means he is not playing at all and it would be a disadvantage to the other players.

- This code has the formula from the theory (exponential of the Scaling Factor and a random Gaussian). It picks form a standard Gaussian and selects n rules per each sample and scale each sample with this factor and then raise it to the exponent to avoid having negative values in the variance. The peer mentions that he tried to use an absolute value but he was obtaining worse results using the exponent. It would've been great to find an alternative way to use the absolute value and make it work more efficiently but the main solution is good as well.

- For the next move, there is set a debugging move to check if the ES algorithm is working optimally. If at the end of the generation our move has a higher probability corresponding to the second move, it means it is working and it should be chosen most of the times. It is overall a decent way to check if we are winning or losing the game.

- In summary, the peer has demonstrated both theoretical understanding and practical implementation skills in evolutionary strategies. The code exhibits a balance between complexity and clarity, making it a commendable solution in optimizing strategies for the given problem.

For TheMassimo

- This code is well-organized for figuring out the best strategies in the game of Nim. The functions are clear and separated, making it easy to follow. However, it might be helpful to add comments in English to make it even clearer.

- The peer did an impressive job coming up with six different strategies. They tested each of these strategies by playing 100 games for every move to figure out which one works the best. Even though the win rate is around 50% and there's a suggestion to use a different method called "nim_sum" for testing, I think it's a good call by the peer to explore other Evolutionary Strategies (ES) that are more adaptive. This makes sense because relying solely on "nim_sum" might not always be the best approach, especially in different game scenarios. The peer's effort in trying different strategies and seeking adaptability is a positive aspect of their work.

- One thing to note is that the code runs a bit slowly because of the many games played for each move in the "adaptive" strategy. Despite this, it's understandable considering the thorough evaluation process.

- One small improvement could be adding a metric for the percentage of victories. Despite this suggestion, the code already prints the number of victories for both players after each set of 100 games, providing useful information about how well the strategies are performing.

Laboratory #9 : Fewest fitness call with EA

The Goal of this laboratory was to write a local-search algorithm (eg. an EA) able to solve the **Problem** instances 1, 2, 5, and 10 on a 1000-loci genomes, using a minimum number of fitness calls.

Our idea was to get the best results without considering the fitness function, then decreasing the number of calls while getting the same results.
We have applied different approaches, and we obtained different results for each strategy.

Simple EA approach

The genotype represents the input to the function, and it is made of an array of a thousand 1 and zeros.
In this case, the **mutation** and **crossover** are simple: In the first one we flip a random bit, in the second we use one-cut crossover.

For the selection of the parents we employed tournament selection with TOURNAMENT_SIZE = 3.

```
POPULATION_SIZE = 20
OFFSPRING_SIZE = 20
TOURNAMENT_SIZE = 3
MUTATION_PROBABILITY = .2

@dataclass
class Individual:
    fitness: int
    genotype: list[int]

population = [
    Individual(
        genotype=choices([0, 1], k=1000),
        fitness=None,
    )
    for _ in range(POPULATION_SIZE)
]

for i in population:
```

```
    i.fitness = fitness(i.genotype)

def select_parent(pop):
    pool = [choice(pop) for _ in range(TOURNAMENT_SIZE)]
    champion = max(pool, key=lambda i: i.fitness)
    return champion

def mutate(ind: Individual) -> Individual:
    offspring = copy(ind)
    pos = randint(0, len(offspring.genotype)-1)

    offspring.genotype[pos] = not offspring.genotype[pos]
    offspring.fitness = None
    return offspring


def one_cut_xover(ind1: Individual, ind2: Individual) -> Individual:
    cut_point = randint(0, len(ind1.genotype))
    offspring = Individual(fitness=None,
                           genotype=ind1.genotype[:cut_point] +
ind2.genotype[cut_point:])
    return offspring

"""
we use a really big value for the generations (100k) in order to find
an upperbound of the value of the fitness function without thinking
about the
number of fitness calls
"""

# for generation in range(10_000): used in the first part
for generation in range(1_500):
    offspring = list()
    for counter in range(OFFSPRING_SIZE):
        if random() < MUTATION_PROBABILITY:
            p = select_parent(population)
            o = mutate(p)
        else:
            # xover # add more xovers
            p1 = select_parent(population)
            p2 = select_parent(population)
            o = one_cut_xover(p1, p2)
        offspring.append(o)
```

```
    for i in offspring:
        i.fitness = fitness(i.genotype)
    population.extend(offspring)
    population.sort(key=lambda i: i.fitness, reverse=True)
    population = population[:POPULATION_SIZE]
    print(f"{population[0].fitness:.2%}")


print(fitness.calls)
fitness._calls = 0
```

ES with Diversity in Parent selection

In this strategy we simply added to the previous strategy a diversity promotion technique, which selects the most diverse parents to be used for the generation of the new offsprings.

SInce our goal is to find the highest numbers of local-optimum, we tried to implement an approach that promotes diversity.

For evaluating how much a parent is different from another one, we had written two functions:

- *most_diverse_couple* : creates K different parents couples randomly. And then, select the one with highest diversity.
- *diversity* : It is the count of how many genes are different in two genotypes compared

```
def most_diverse_couple(k=5):
    best_couple = None
    best_diversity = -1
    for _ in range(k):
        #select a random couple of parents
        p1, p2 = choice(population), choice(population)
        diversity_value = diversity(p1, p2)
        if diversity_value > best_diversity:
            best_couple = p1, p2
            best_diversity = diversity_value
    return best_couple

def diversity(ind1: Individual, ind2: Individual):
    diff = 0.0
    for i in range(0, len(ind1.genotype)):
        if ind1.genotype[i] != ind2.genotype[i]:
            diff = diff + 1
```

```
    return float(diff)/float(len(ind1.genotype))

#SAME GENERATION AS BEFORE, THE ONLY DIFFERENCE IT IS IN THE PARENTS
SELECTION
   p1, p2 = most_diverse_couple()
```

ES + Adaptiveness

In this strategy we take the simple EA approach and we introduced the idea of adaptiveness
In particular, if we see a stagnation situation, meaning that the fitness function is not improving, we adaptively change the mutation rate.
It is possible to see it in the mutation function, with the parameter **n_mutation** which is initially set to 1, and then it can increase.

We increase the mutation rate in this way `n_mutation = min(1000, n_mutation+1)`

```
n_mutation = 1
def mutate(ind: Individual) -> Individual:
    offspring = copy(ind)
    pos = sample(range(1000), k=n_mutation)
    for p in pos:
        offspring.genotype[p] = 1-offspring.genotype[p]
    offspring.fitness = None
    return offspring
```

How do we keep track of the improvement ?
The approach is to keep track of the previous best fitness values. If our current best fitness is the same as the previous one, we start to increase the mutation rate.

```
previous_fitness = -1
counter_same_fitness = 0
for generation in range(10_000):
    offspring = list()
    for counter in range(OFFSPRING_SIZE):
        if random() < MUTATION_PROBABILITY:
            p = select_parent(population)
            o = mutate(p)
        else:
            # xover # add more xovers
            p1 = select_parent(population)
            p2 = select_parent(population)
            o = one_cut_xover(p1, p2)
        offspring.append(o)
```

```
    for i in offspring:
        i.fitness = fitness(i.genotype)
    population.extend(offspring)
    population.sort(key=lambda i: i.fitness, reverse=True)
    population = population[:POPULATION_SIZE]


    if population[0].fitness>previous_fitness:
        print(f"generation {generation} fitness
{population[0].fitness:.2%}")
        previous_fitness = population[0].fitness
        counter_same_fitness = 0
        n_mutation = 1 #immediately back to exploitation
    else:
        counter_same_fitness += 1
        print(f"generation {generation} mutate {n_mutation} fitness
{population[0].fitness:.2%}")
        if counter_same_fitness >= 5 :
            n_mutation = min(1000, n_mutation+1)

print(fitness.calls)
fitness._calls = 0
```

ES + Diversity + Adaptiveness

Finally, we tried all the approaches together.

Results
Here, there are the results with the same configurations.

Conf	ES	ES+Div	ES+Adapt	ES+Div+Adapt
1	94.60%	89.60%	89.50%	81.65%
2	96.04%	93.00%	94.72%	87.42%
5	98.78%	96.11%	98.06%	88.28%
10	98.89%	94.57%	98.45%	87.11%

The simpler approach is the winning one.

A couple of comments:

1. We were not able to reduce the calls to the fitness function since the other algorithms gave similar or worse results compared to ES.
2. Applying diversity in parent selection wasn't effective. Maybe the choice of ordering tuple based on diversity wasn't the best for this particular task.
3. Applying Diversity with an adaptive strategy always returns the worst results.
4. In general rerunning multiple times the same algorithm can yield results that different from a factor up to -1.5 and +1.5%.

Since the best algorithm is ES we can now try to reduce the number of calls for the fitness function in each configuration:

Conf	Fitness	Calls
1	94.40%	30020
2	96.46%	30020
5	98.28%	30020
10	98.11%	30020

To lower the fitness call we set the population and offspring size equal to 20 and limited the number of generations to 1_500. This led to very similar results with a great reduction to the number of calls to the fitness function. This same configuration for the other 3 approaches still gave worse results.
With lower values the evolutionary process is stopped way too early and thus the results are worse.

Peer reviews received

By s316467

The provided code demonstrates a well-structured approach to solving optimization problems using evolutionary algorithms, with a clear segmentation for different algorithmic strategies. It effectively utilizes Python's dataclass for representing individuals, enhancing readability and maintainability. The modular design of functions like select_parent, mutate, and one_cut_xover is commendable, as it allows for reusability and scalability in the code. Additionally, the systematic approach to experimentation and analysis, testing different configurations and thoughtfully examining the results, is a strong aspect of this work, showcasing a practical problem-solving mindset.
However, there are areas where the code could benefit from improvements. The use of hardcoded values such as population size and mutation probability limits the flexibility of the

algorithms. Parameterizing these values would allow for more dynamic experimentation. While the code is well-organized, it would benefit significantly from more in-depth comments, especially in complex sections, to aid in understanding the underlying logic and purpose. There is noticeable repetition in the initialization of populations across different algorithms, which could be streamlined into a single function to reduce redundancy and improve the code's conciseness.
Performance optimization is another area that could be enhanced, particularly in functions that are frequently called, such as mutate and diversity. Implementing error handling and input validation would also make the code more robust and resistant to unexpected inputs or runtime errors.
To improve this code further, converting hardcoded values into parameters or configuration settings would be beneficial. Enhancing documentation with comprehensive in-line comments would aid in clarifying the code's intentions and functionalities. Refactoring to abstract common functionality into functions could reduce redundancy and improve the code's maintainability. Performance profiling to identify and optimize computational bottlenecks would enhance the efficiency of the algorithms. Implementing a testing framework would ensure the correctness and reliability of the algorithms across different scenarios. Furthermore, exploring advanced genetic operators or selection strategies could potentially yield better results in fewer fitness function calls. Lastly, adding visualizations to depict the evolution process and outcomes can provide more intuitive insights into the algorithms' performance.
In summary, while the code shows a solid foundation in applying evolutionary algorithms to optimization problems, enhancements in parameterization, documentation, optimization, and robustness could elevate its effectiveness and usability.

Peer reviews issued

For TheMassimo

Your approach in tweaking the evolutionary algorithm by adding new ways genes mutate and crossover is great. It shows you're trying different things to make the algorithm explore more possibilities. I like that you're also considering diversity with the extinction function and exploring island models. However, it would be awesome if you could share more specific results, like how exactly these changes improved problems 1, 2, and 5. Also, for Problem 10 size, it's not clear why the island and segregation models didn't work. Including a bit more detail on that would make your explanation more complete and easier to understand. Overall, you've done a lot of cool stuff, and maybe sharing some visuals or graphs could help visualize the solutions better. Keep up the good work!

For Simone Giambrone

The code uses an EA, adding a cross over with a number of cuts equal to 5 and using a mutation with a for loop. Also if after a 100 generations the standard deviations is low than we eliminate part of the population and we replace it.
I only have some suggestions:

- using a loop in the mutation may happens that an index is used multiple times.

- a bit of more comments would have helped.
- You could have employed a grid search to find the best hyperparameters (#generation, mutation prob ecc).
- Maybe adding a table with the results for each generation (value of fitness and fitness calls would have helped).
  That being said this a solid work that really show your dedication to the course.

For s310168

The code is very clean and the results well presented. The idea of tweking the probability of the best parent is interesting, but it may be that if the best individual is in a local optimum you may have some problems escaping it (maybe it's the reason why your results aren't very high for some problems). Maybe you could have used it with also an island approach, so that if in one island you are in a local optimum it doesn't propagate. Lastly for the mutation you could have added an adaptive strategy, increasing the number of mutated genes if the individuals aren't getting better after a while.

Laboratory #10 : Reinforcement learning for Tic Tac Toe

In this laboratory the goal was to use reinforcement learning to devise a tic-tac-toe player. In particular, our work was focused on:

1. Creating an action-value function
2. Designing a hybrid Monte Carlo approach: when a certain threshold MAX_NUM_ACTIONS=3 is reached, a full-game simulation is applied

The **win function** : if the player has selected some squares such that with 3 squares the sum is 15 he won.
we have to define the squares in tic tac toe such that this function make sense
The **state_value** : it is used to evaluate if we have won or not, and gives us the reward which is 1 if we have won, -1 if we have lost, 0 in the case of a draw

```
def win(squares):
    return any(sum(c) == 15 for c in combinations(squares, 3))

Position = namedtuple('Position', ['x', 'o'])

#this state value can become the evaluation function
def state_value(position : Position):
    if win(position.x):
        return 1
    elif win(position.o):
        return -1
    else:
        return 0
```

```
#magic board that is used for check, every value correspond to a square
MAGIC = [2, 7, 6, 9, 5, 1, 4, 3, 8]
```

Agent strategy #1 : Action Value function

Starting from the function random_game we want to play some random matches to explore the possibilities.
We can see that random games return the trajectory, which is the history of the game, and the final state (win, loose, drawn) which is useful for the evaluation.

```
#game with two random players
def random_game():
    state = Position(set(), set())
    available = set(range(1, 10)) #from 1 to 9 which are the squares
    trajectory = []

    while True:
        #first player
        x = choice(list(available))
        trajectory.append((deepcopy(state), x)) #current state + chosen
                                                                action
        state.x.add(x)
        available.remove(x)
        if win(state.x):
            break
        elif len(available) == 0:
            break

        y = choice(list(available))
        state.o.add(y)
        available.remove(y)

        if win(state.o):
            break
        elif len(available) == 0:
            break
      #sequence of move for computing the state value function
      (trajectory) + final result (state)
```

```
    return trajectory, state
```

This is the actual training phase
We play 4000 random games and each time we update the Q function with the results of the game.

```
Q_func = defaultdict(float)
epsilon = 0.01
for step in range(4_000):
    trajectory, last_state = random_game()
    final_reward = state_value(last_state)
```

Frozenset allows you to use the set as a key, it is hashable. This is done because state is not hashable otherwise

```
    for (state, action) in trajectory:
        hash_state = (frozenset(state.x), frozenset(state.o), action)
        difference = (final_reward - Q_func[hash_state])
        Q_func[hash_state] = Q_func[hash_state] + epsilon*difference
```

Agent strategy #2: Hybrid Monte Carlo simulation with full game simulation

As a second agent we have created a hybrid of MonteCarlo and Full game: we start by exploring a certain path but after a certain level of depth, which is indicated with the variable **`MAX_NUM_ACTIONS`** we switch to exploitation and explore the entire subtree.
The reward is the average of all possible terminal states and that's the value that will be propagated.
For nodes below the level we use a pessimistic strategy, assigning the worst reward to the node.

```
Q_func = defaultdict(float)
epsilon = 0.01  #determine how much impact the update of a value
#game with two random players using a hybrid
def full_game(state, available):
    turn_player = 'x' if len(state.x) == len(state.o) else 'o'

    if win(state.x):
        return [1]
    elif win(state.o):
        return [-1]
    elif len(available) == 0:
        return [0]

    general_list = []
    for act in available:
        new_available = deepcopy(available)
        new_state = deepcopy(state)
```

```
        if turn_player == 'x':
            new_state.x.add(act)
        else:
            new_state.o.add(act)
        new_available.remove(act)
        list_act = full_game(new_state, new_available)




        if turn_player == 'x':
            #we use min to consider the worst case so the agent will
            choose more safe actions
            hash_state = (frozenset(state.x), frozenset(state.o),
            action)
            Q_func[hash_state] = min(list_act)
        general_list.extend(list_act)
    return general_list

# after 3 iterations of MC we start doing full_game
MAX_NUM_ACTIONS = 3
def random_game():
    state = Position(set(), set())
    available = set(range(1, 10))
    trajectory = []
    num_actions = 0

    while num_actions < MAX_NUM_ACTIONS:
        #first player
        x = choice(list(available))
        #current state + chosen action
        trajectory.append((deepcopy(state), x))
        state.x.add(x)
        available.remove(x)

        if win(state.x):
            break
        elif len(available) == 0:
            break

        #opponent player (doesn't consider the state)
        y = choice(list(available))
```

```
        state.o.add(y)
        available.remove(y)
        if win(state.o):
            break
        elif len(available) == 0:
            break

        num_actions += 1



    #here we change strategy if we have not finished the game yet, we
    start full game
    if win(state.x):
        reward = 1
    elif win(state.o):
        reward = -1
    elif len(available) == 0:
        reward = 0
    else:
        #game not finished: exploitation of the subtree
        #list of all rewards, one for each terminal state
        reward_list = full_game(state, available)
        #let's compute the average result
        reward = sum(reward_list)/len(reward_list)
    return trajectory, reward

#Q_func → key: (current_situation of the board, action); value: reward
for step in range(100_000):
    trajectory, final_reward = random_game()
    for (state, action) in trajectory:
        hash_state = (frozenset(state.x), frozenset(state.o), action)
        difference = (final_reward - Q_func[hash_state])
        #we are updating the reward, if we had the same key
        #if the key, for the Q_func is new, the value is epsilon*diff.
        Q_func[hash_state] = Q_func[hash_state] + epsilon*difference
```

Testing the agents

For each possible move given, a state picks the one with the higher expected reward.

```
def agent(Q_dict, state, available):
    current_reward = -1000
    action = -1
    for i in available:
```

```
        hash_state = (frozenset(state.x), frozenset(state.o), i)
        reward = Q_dict[hash_state]
        if reward > current_reward:
            current_reward = reward
            action = i
    return action
```

Results

We have tried for each agent to change the number of the iterations and then tried to play 100_000 games with a random player. Here the results (for each agent we present the number of wins and draws in percentage):
We have the following conclusions:

- Thanks to a better exploration of the tree the second agent can achieve good results with a lower number of iterations.
- The first agent can converge quickly to almost perfect performance (with 100_000 iterations) while the second agent is struggling. This may be due to how we treated nodes at a lower level, or maybe the epsilon value is too low.

Iterations	Agent1	Agent2
3_000	(81.5, 7.5)	(82, 6)
4_000	(83, 8.5)	(85, 7)
5_000	(86, 6)	(88, 8.5)
10_000	(90, 5.4)	(90, 7)
20_000	(95.3, 3.3)	(87, 8)
50_000	(97, 2.5)	(90, 8)
100_000	(98.7, 1.3)	(93.4, 5.5)
200_000	(99, 1)	(95, 4)
500_000	(99, 1)	(95.6, 4)

Peer reviews received

By s316467

The code in this lab is a well-thought-out application of reinforcement learning (RL) to develop a Tic-Tac-Toe player. This work reflects a collaborative effort by all group members. Their approach includes two distinct methods: an action-value function agent and a hybrid Monte Carlo agent with full-game simulation. This dual-method approach is an innovative way to explore different RL strategies within the relatively simple domain of Tic-Tac-Toe.

The first part of the code focuses on developing an action-value function for the Tic-Tac-Toe game. The use of the magic square and the win function to determine the game's outcome is a clever application of classical problem-solving in combinatorics. This method transforms the game into a numerical puzzle, simplifying the process of determining the winner. The action-value function, updated through repeated random gameplay, serves as a foundational element for the AI's decision-making process. This part of the implementation stands out for its simplicity and effectiveness.

The second part introduces a more complex agent that combines Monte Carlo methods with a full-game simulation strategy. This agent starts by exploring certain paths and, upon reaching a threshold (MAX_NUM_ACTIONS=3), switches to a full-game simulation to evaluate the outcomes. This strategy allows for a deeper exploration of possible game states and outcomes, potentially leading to more informed decision-making by the AI. However, this complexity also introduces the challenge of managing a larger state space and ensuring efficient computation.

Experimentally, the team has conducted extensive trials to evaluate the performance of both agents. The results, presented in a tabulated format, offer a clear comparison between the two methods across various iteration counts. These results show that while the first agent can quickly converge to near-perfect performance, the second agent struggles to reach the same level of proficiency. This could be due to the way lower-level nodes are treated or the chosen value of epsilon.

Overall, the code demonstrates a strong grasp of RL principles and their application to a classic game. The innovative use of a magic square in the first agent and the exploration-exploitation balance in the second agent are commendable. However, the complexity and computational efficiency of the second agent could be areas for further investigation and optimization. Additionally, the experiment's design and the presentation of results provide a clear and thorough understanding of the agents' performances, offering valuable insights into the strengths and limitations of each approach.

In conclusion, this project stands as a comprehensive and insightful exploration into applying reinforcement learning techniques to a well-known game. The dual-agent approach not only provides a robust test bed for comparing different RL strategies but also offers valuable lessons in game theory, algorithm design, and experimental analysis.

By Enrico Capuano s317038

First of all, the code is a good application of reinforcement learning principles.
You smartly used two different approaches, to explore different RL strategies in Tic Tac Toe game.

The first approach uses an action-value function and simplifies the process of the determination of a winner, transforming the game into a numerical puzzle.
This is very effective and clear.

In the second approach, you used in a very good way the Monte Carlo method with full game strategy.
Although it increases the method complexity, it allows to go deeper in the solution space, finding more game states and outcomes.

Overall, the code shows a good understading of all RL principles, also using the magic square in an effective way.
Moreover, I really appreciate the representation of all results in a table, that is very clear to read and understand and allows the reader to compare different scores.

In conclusion, you did a very good job exploring different RL techniques, so well done!
I hope my review will be useful to you, thank you for reading me!

Peer reviews issued

For Alessandro Chiabodo

I think this code is good. It makes a Tic-Tac-Toe game and has different kinds of players, like random ones and a smart one that learns by playing. The game training part looks cool, where the smart player learns and gets better over time. However, in some places, simpler words or comments could make it easier to understand. For example, in the ReinforcedPlayer class, explaining a bit more about why it's updating the Q-values could be helpful. Also, in the training loop, using more clear names for variables could make it easier to follow. Overall, it's a nice code for a game and learning player.

For TheMassimo

I think this code is good. It sets up a Q-learning agent for Tic-Tac-Toe in a clear way using the QAgent class. The use of named tuples for the game state is nice and makes the code easy to follow. The random_game function is handy for creating random game situations during training.

However, I think there are areas to make it even better. The code doesn't handle when the game is finished properly, and it needs a way to recognize that. Adding a function for that would help. Also, using more descriptive names for variables and adding more comments inside the QAgent class methods, especially for the Q-value updates, would make it easier for others to understand. Despite these points, the code is a good starting point for a Tic-Tac-Toe agent using Q-learning.

Final Project : Quixo game

The goal of the final project was to write one or more agents able to play the Quixo game by employing some strategies that we studied during the course.
The agents we developed will be tested against a random opponent.

```
class RandomPlayer(Player):
    def __init__(self) -> None:
        super().__init__()

    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        from_pos = (random.randint(0, 4), random.randint(0, 4))
        move = random.choice([Move.TOP, Move.BOTTOM, Move.LEFT,
        Move.RIGHT])
        return from_pos, move
```

Agent strategy #1 : MinMax

We started by using a MinMax strategy.
Since the solution tree is huge we have to set a limit to the depth (cut-off). By doing some tests we found that the evaluation of non terminal nodes could use the difference between zeroes and ones: in this way we are forcing our agent to choose to place more zeroes by picking neutral elements than picking zeroes tiles at the beginning of the games.
The idea is that by putting more zeroes the chance of winning later is higher.

get_new_board → With this function we create a new board, apply the given moves and then we return it. This function is used to create the MinMax tree where every board represents a node of the tree.
acceptable_move → Given a position and a move, it returns true if the move is valid
get_possible_moves → Given a state, it return the acceptables moves

```
class MinMaxPlayer(Player):
    def __init__(self, max_depth=2) -> None:
        super().__init__()
        self.MAXIMUM_DEPTH = max_depth

    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        _, combination = self.minmax(game, True, 0)
        return combination

    def minmax(self, game, maximizing_player, depth):
        # The MinMax algorithm function
        # It returns an evaluation which a score for a certain player
        and the best move it can do
        if (game.check_winner() != -1 or depth==self.MAXIMUM_DEPTH):
```

```
            if game.check_winner() == 0:
                return 1, None
            elif game.check_winner() == 1:
                return -1, None
            else: # depth==self.MAXIMUM_DEPTH:
                #more zeroes the better since we are player 0
                num_zeros = np.count_nonzero(game._board == 0)
                num_ones = np.count_nonzero(game._board == 1)
                return (num_zeros - num_ones)/100, None
                #we take the solution with the highest number of 0

        depth = depth + 1 #increase depth
        if maximizing_player:
            max_eval = float('-inf')
            best_move = None
            for possible_move in self.get_possible_moves(game, 0):
                new_game = self.get_new_board(game, possible_move, 0)
                eval, _ = self.minmax(new_game, False, depth)
                if eval > max_eval:
                    max_eval = eval
                    best_move = possible_move
            return max_eval, best_move
        else:
            min_eval = float('inf')
            best_move = None
            for possible_move in self.get_possible_moves(game, 1):
                new_game = self.get_new_board(game, possible_move, 1)
                eval, _ = self.minmax(new_game, True, depth)
                if eval < min_eval:
                    min_eval = eval
                    best_move = possible_move
            return min_eval, best_move

    def get_new_board(self, game, possible_move, player):
        #We do a deep copy of the game object because in MinMax we
        change the game not only the board attribute
        new_game = deepcopy(game)
        pos, move = possible_move
        res = new_game.move(pos, move, player) #we are trying the move
        if not res:
            print(f'ERROR: SHOULDNT RECEIVE FALSE on {pos} and {move}
            player {player}')
            game.print()
```

```
            sys.exit(1)
        return new_game

    def acceptable_move(self, pos, mov):
         return not ((pos[1] == 0 and mov == Move.TOP)
                     or (pos[1] == 4 and mov == Move.BOTTOM)
                     or (pos[0] == 0 and mov == Move.LEFT)
                     or (pos[0] == 4 and mov == Move.RIGHT))

    def get_possible_moves(self, game, player):
        possible_pos = [(i, j) for i in range(0,5) for j in range(0, 5)
                            if (i == 0 or i == 4 or j == 0 or j == 4)
                            and (game._board[j,i] == -1 or
                            game._board[j,i] == player)]

        possible_moves_pos = [(pos, mov) for pos in possible_pos for
        mov in [Move.TOP, Move.BOTTOM, Move.LEFT, Move.RIGHT]]
        #check if position and moves are feasible
        possible_moves_pos = [(pos, mov) for pos, mov in
        possible_moves_pos if self.acceptable_move(pos, mov)]
        return possible_moves_pos
```

To decide the limit of the depth we measured the time to evaluate the initial state (being that is the one with higher number of possible moves) for each possible depth. By looking at the results we decided to use depth equal to 2.

```
initial_game = Game() #the empty board is the one it takes longer time
for depth in range(1, 4):
    player = MinMaxPlayer(max_depth=depth)
    start_time = time.time()

    _ = player.make_move(initial_game)

    end_time = time.time()

    print(f"Execution time for depth {depth}: {end_time - start_time}
    seconds")
```

Here we can see the result for compute a move with Max_depth of the MinMax tree : 1,2,3.

```
Execution time for depth 1: 0.001154184341430664 seconds
Execution time for depth 2: 0.5223915576934814 seconds
Execution time for depth 3: 18.050224542617798 seconds
```

This is the function for testing our algorithm against a random player. Our results were almost 100% of the winning rate.

```
wins = 0
player1 = MinMaxPlayer()
player2 = RandomPlayer()
for i in range(1000):
    print(f'starting game {i}')
    g = Game()
    winner = g.play(player1, player2)

    if winner == 0:
        wins += 1

print(f'I won {wins} time out of 1000')
```

Agent strategy #2 : MinMax with Alpha-Beta pruning

Since the depth is very low due to the branching factor of the algorithm we decided to use alpha beta pruning to avoid computing most of the subtrees.
The strategy is the same as before (picking more zeros than one) but now the *_minmax* function includes the alpha beta pruning.
The basic idea is: in the case of the max player we receive from the parent node (min player) in the current minimum value, in the variable beta. If we find a value which is higher than the minimum we can immediately stop because we won't change the decision of the parent node. The same happens when we are computing the min player.

```
   def minmax_alpha_beta(self, game, depth, alpha, beta,
   maximizing_player):
        # The MinMax algorithm function with Alpha-Beta Pruning
        # alpha = current maximum value that we have found on that
        sub-branch
        # beta  = current minimum value that we have on that sub-branch
        # Base case: If the game is over or maximum depth is reached
        if (game.check_winner() != -1 or depth==self.MAXIMUM_DEPTH):
            if game.check_winner() == 0:
                return 1, None
            elif game.check_winner() == 1:
                return -1, None
            else: # depth==self.MAXIMUM_DEPTH
                #more zeroes the better since we are player 0
                num_zeros = np.count_nonzero(game._board == 0)
                num_ones = np.count_nonzero(game._board == 1)
                return (num_zeros - num_ones)/100, None
            #we take the solution with the highest number of 0
        depth = depth + 1 #increase depth
```

```
        if maximizing_player:
            max_eval = float('-inf')
            best_move = None
            for possible_move in self.get_possible_moves(game, 0):
                new_game = self.get_new_board(game, possible_move, 0)
                eval, _ = self.minmax_alpha_beta(new_game, depth,
                alpha, beta, False)
                if eval > max_eval:
                    max_eval = eval
                    best_move = possible_move
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break  # Beta cutoff
            return max_eval, best_move
        else:
            min_eval = float('inf')
            best_move = None
            for possible_move in self.get_possible_moves(game, 1):
                new_game = self.get_new_board(game, possible_move, 1)
                eval, _ = self.minmax_alpha_beta(new_game, depth,
                alpha, beta, True)
                if eval < min_eval:
                    min_eval = eval
                    best_move = possible_move
                beta = min(beta, eval)
                if alpha > beta:
                    break  # Alpha cutoff
            return min_eval, best_move
```

We can see the improvement, with the same function used before for evaluating the computation time for a single move .

```
Execution time for depth 1: 0.0142581462860107 42 seconds
Execution time for depth 2: 0.037334203720092 77 seconds
Execution time for depth 3: 0.4546060562133789 seconds
Execution time for depth 4: 1.267000436782837 seconds
Execution time for depth 5: 21.52586340904236 seconds
```

By using the same test function as before, the results we obtained are the same but by printing the board after each move we concluded that the chosen moves done by minmax were more rational.

Agent strategy #3 : Reinforcement Learning with Monte-Carlo approach

We decided to test Reinforcement learning with this game: the agent will play a certain number of games in a training phase to estimate a Q_function. We assume that by playing a lot of games the Q value of a node will converge to the expected reward of that node. Still we had some doubt since the tree is huge, so it may take a lot of iterations to actually explore it.

The Monte Carlo approach is the building base of this algorithm: we play a set of random games, see the results and use those to update all the nodes in the trajectories. We used a learning rate (the update weight) of 0.01.

Also the RL approach inherit the functions : **acceptable_move, get_possible_moves**

```
from random import choice
from collections import defaultdict

class RL_Player(Player):
    def __init__(self) -> None:
        super().__init__()
        self.Q_func = defaultdict(float)

    '''
    Once we trained the agent, we use this function to play
    '''
    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        #game.print()
        moves = self.get_possible_moves(game, 0)
        #we assume we want the maximum reward possible
        reward = -float('inf')
        best_comb = None
        board_tuple = tuple(map(tuple, game._board))
        #let's get the best action for this scenario from the
        Q-function
        for mov in moves:
            rew = self.Q_func[(board_tuple, mov)]
            if rew > reward:
                reward = rew
                best_comb = mov

        from_pos, move = best_comb
        return from_pos, move

    def __random_game(self):
        state = Game()
```

```
        trajectory = []

        while True:
            #first player is US
            x = choice(self.get_possible_moves(state, 0))
            trajectory.append((deepcopy(state), x))
            #current state + chosen action
            pos, move = x
            state.move(pos, move, 0)

            #check_winner != -1 because i can make a bad action
            if state.check_winner() != -1:
                break

            #opponent player (doesn't consider the state)
            y = choice(self.get_possible_moves(state, 1))
            pos, move = y
            state.move(pos, move, 1)

            if state.check_winner() != -1:
                break
        #sequence of move for computing the state value function
        return trajectory, state

    def state_value(self, state):
        return 1 if state.check_winner() == 0 else -1

    def train_agent(self, iterations, epsilon = 0.01):
        for _ in range(iterations):
            #print(f"ITERATION {i}")
            trajectory, last_state = self.__random_game()
            final_reward = self.state_value(last_state)
            for (state, action) in trajectory:
                #convert numpy to tuple to make it hashable
                board_tuple = tuple(map(tuple, state._board))
                hash_state = (board_tuple, action)
                difference = (final_reward - self.Q_func[hash_state])
                self.Q_func[hash_state] = self.Q_func[hash_state] +
                epsilon*difference
```

Here the code to train and then test our agent:

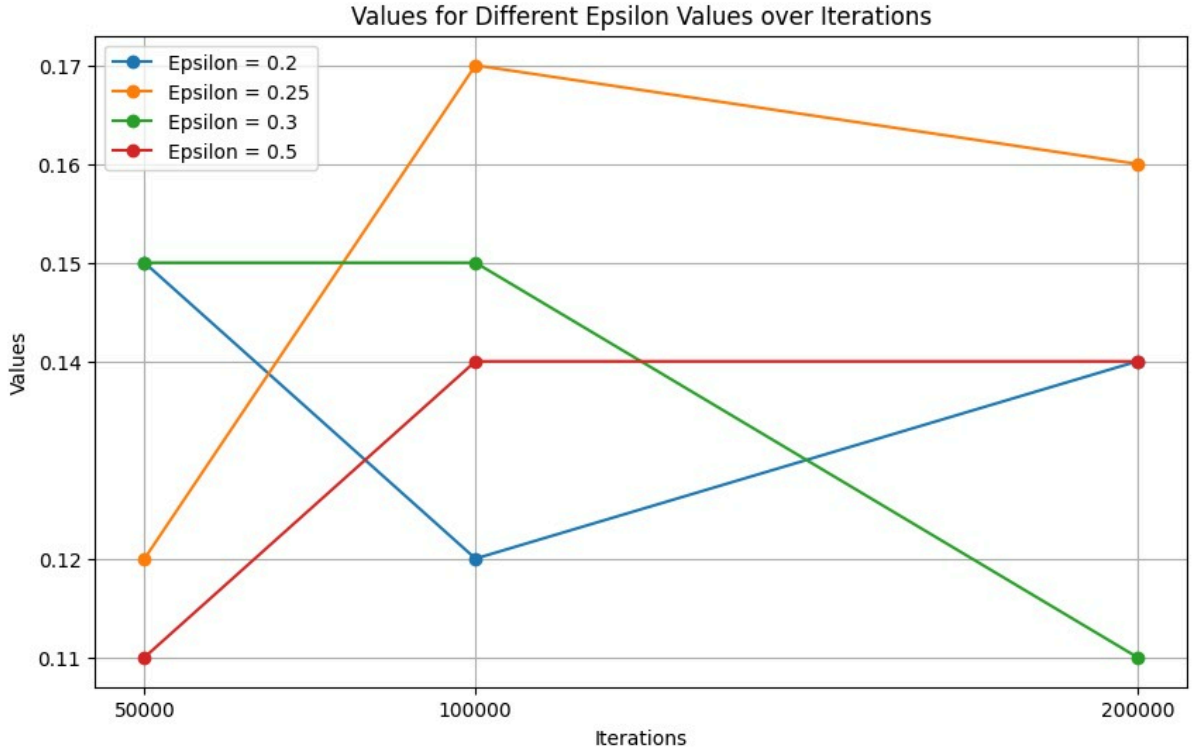
```
player1 = RL_Player()
player2 = RandomPlayer()

training_its = 200_000
player1.train_agent(training_its)

win = 0
for _ in range(1000):
    g = Game()
    winner = g.play(player1, player2)
    if winner == 0:
        win += 1

print(f' We won {win} out of 1000 games')
```

Our doubts were correct: with 200_000 iterations the process took more than one hour and the results were not interesting. We tried changing the iterations and the learning rate:



Agent strategy #4 : RL with Monte-Carlo and parallelization

For this last agent we added two strategies to the RL:

- now it's possible to save the Q function and reload it later to be trained. This gave us the possibilities to train our agent for many iterations without our computers crashing.
- Since the computation of a single game is independent from the others we used Thread Parallelization to speed up the execution.

As we can see from the code we used two optimizations: ThreadPool and Future functions.

1. With threadpool we don't create a thread for each function call but only a fixed amount. Each thread will look at the queue and execute the function.
2. With Future we also parallelize the processing of the results: instead of waiting for all the threads to finish each time a thread stops, we immediately process the result in a callback-fashion.
3.

Since the Q_func becomes too heavy for a single computation, with the result of saturating all the RAM available in our computer, we decided to save the Q_func (which is stored in a dictionary) inside a pickle file; so that we can later reload it and resume the computation.
In this way, we can create bigger Q_func through several computations instead of a single one.

```
def train_agent(self, iterations, epsilon=0.01, save_dictionary=False):
        with concurrent.futures.ThreadPoolExecutor() as executor:
            futures = [executor.submit(self._random_game) for _ in
            range(iterations)]

            i = 1
            for future in concurrent.futures.as_completed(futures):
                i+=1
                print(f' process {i} finished')
                try:
                    trajectory, last_state = future.result()
                except Exception as e:
                    print(f' Error in process {i}: {e}')
              #Continue to the next iteration if there was an exception
                    continue
                final_reward = self.state_value(last_state)
                for (state, action) in trajectory:
                    board_tuple = tuple(map(tuple, state._board))
                    hash_state = (board_tuple, action)
                    self.update_Q_func(hash_state, final_reward,
                    epsilon)
        if save_dictionary:
        # Specify the file path where you want to save the dictionary
            file_path = 'my_dictionary.pickle'
```

```
            #in case erase the old file
            pickle.dump(self.Q_func, open(file_path, 'wb'))

    def load_from_dictionary(self, file_path = 'my_dictionary.pickle'):
        # Load the dictionary from the file
        self.Q_func = pickle.load(open(file_path, 'rb'))
```

Train of the agent for multiple times:

```
#training phase
player1 = RL_Player()
player2 = RandomPlayer()

iterations = 200_000
player1.load_from_dictionary()
player1.train_agent(training_its, save_dictionary=True)
#save_dictionary: if it's true it overrides the file with the
dictionary with the new one
```

Results for Reinforcement Learning

We can start by seeing the improvements in the time:

Iterations	old RL	improvements
50_000	3 min 40 sec	-30 sec
200_000	1 hour 5 min	-20 min

We can see correlations between iterations and saved time.

Performances wise we were able to reach more of a million iterations by retraining the agent more times. Even by exploring a lot (the pickle file became 5GB) still the exploration was not enough and the results were only roughly 33% of winnings.

Final considerations

These are the results for MinMax with Alpha-Beta pruning
We can say that we have reached optimal result both in terms of time and winning rate

```
starting game 0
starting game 1
starting game 2
starting game 3
starting game 4
...
...
...
I won 1000 time out of 1000
```

For the Reinforcement Learning we confirm the problem of exploring the entire Quixo tree. This leads to obtain worst results.