Computational Intelligence

Roxane GOFFINET

Fall Session 2023-2024

Report

Supervised by Giovanni SQUILLERO

Acknowledgements

This report, the code produced and the research behind it would not have been possible without the great support of the professor SQUILLERO as well as the lab assistants that have guided us during this semester at Politecnico di Torino. This course was very enriching for me and enabled me to develop my computer skills. It's with great satisfaction that I'll be going back to my home university after this semester, enriched by lots of new learning.

All my code and projects are available on my github :
`https://github.com/RoxaneGoffinet/Computational-Intelligence/tree/main`

1 Pizza-Pub Problem

I started to work on the Pizza-Pub problem that was shown in class, I especially clarified all the constraint of the problem. At first I missed adding the fact that, of course, the bicycle cannot be moved back from the Pub to the Pizza restaurant without someone riding it... The final list of constraints that I found to describe the problem were :
— There must be someone on the bike for it to move
— The number of data scientist and computer scientist in total is fixed
— The number of data scientist should always be lesser in a place that the number of computer scientist
— The tandem bike can only move one or two people at a time
— At the beginning all are at the pizza restaurant
— At the end we want them all to be at the pub

My findings were that not all number of initial people can work. But also it is impossible to resolve the problem if there is more data scientist than computer scientist. And that the solution was obvious whenever there was only one data scientist. However the results were presented in class before I was able to implement properly this problem.

I also at the same time learn how to use github because even though I was registered on the platform for 3 years I never learnt how to use it. It changed my life.

2 Lab1

The first lab goal was to compare and implement different algorithms to explore and cover sets. For that I constructed the **Depth First**, **Breadth First**, **Greedy Best First** and **A***. And then to create also special sets in order to test those algorithm.

The Set Covering Problem is a classic optimization problem that involves selecting a subset of elements from a given collection in such a way that the selected subset covers all the elements in the original collection. This problem is known to be NP-hard, meaning there's no known polynomial-time algorithm that solves all instances of the problem efficiently. That is why, various approximation algorithms, greedy algorithms, and heuristics are commonly used to find near-optimal solutions in a reasonable amount of time, especially for larger problem instances.

— **Depth First** is a fundamental graph traversal algorithm used to explore nodes and edges in a graph structure. It starts at a selected node and explores as far as possible along each branch before backtracking. While being one of the quickest algorithm I have tested Depth First doesn't guarantee the shortest path between nodes in a graph.
— **Breadth First** is another fundamental graph traversal algorithm used to explore nodes and edges in a graph. Unlike Depth-First, BF explores all the neighbor nodes at the present depth before moving on to nodes at the next depth level.
— **Greedy Best-First** is a variant of the Best-First algorithm. It is heuristic-based, it explores the graph by prioritizing nodes based on an evaluation function that estimates how close a node is to the goal state, without considering the total cost from the start node. It's called "greedy" because

at each step, the algorithm chooses the node that appears to be the most promising according to the heuristic, regardless of the path cost to reach that node.

— **A*** is a popular and widely used algorithm that combines the features of both uniform cost search and greedy best-first search by considering both the cost of the path from the start node and an estimated cost to reach the goal node. A* search aims to find the shortest path from a starting node to a goal node in a weighted graph or a grid, where each edge or step between nodes has a specific cost or distance. The evaluation function at a node n can be written as $f(n) = h(n) + g(n)$ with $h(n)$ a heuristic function that estimates the cost or distance from the current node to the goal node and $g(n)$ a cost function that represents the actual cost from the start node to the current node.

For A*, I designed 3 different heuristics in order to see wich one will works best. The first one h_1 was taken as the division of the number of state that has not been covered by the size of the largest set. This formulation of the heuristic is admissible, meaning it never overestimate the true cost to reach the goal, only provide a lower bound. The second one h_2 had the same idea but used a different way to calculate the largest set size. h_3 heuristic idea is a bit different, it iteratively selects sets from the sorted candidates list until the cumulative number of new elements covered by these sets exceeds or equals the missing size. It increments the taken counter until the condition is met. Finally, the function returns the value of taken, which represents the number of sets required to cover enough new elements to reduce the missing size to zero or less. This implementations of the heuristic can be seen under. The $g(n)$ function is given by the number of state already covered, the same as used in Greedy Best First.

```

def h1(state):
    largest_set_size = max(sum(s) for s in SETS)
    missing_size = PROBLEM_SIZE - sum(covered(state))
    estimate = ceil(missing_size / largest_set_size)
    return estimate


def h2(state):
    already_covered = covered(state)
    if np.all(already_covered):
        return 0
    largest_set_size = max(sum(np.logical_and(s, np.logical_not(already_covered))) for s in SETS)
    missing_size = PROBLEM_SIZE - sum(already_covered)
    estimate = ceil(missing_size / largest_set_size)
    return estimate


def h3(state):
    already_covered = covered(state)
    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 special set was designed by putting all "True" in one set and then "False" in every other set.

```
    # Definition of a special set to test the heuristics

    SETS = [tuple([False] * PROBLEM_SIZE) for _ in range(NUM_SETS)]
    random_set_index = random.randint(0, NUM_SETS - 1)
```

```
SETS[random_set_index] = tuple([True] * PROBLEM_SIZE)
print(f"The set with all True values is at index {random_set_index}")
assert goal_check(State(set(range(NUM_SETS)), set())), "Problem not solvable"
```

Then, I compared the results obtained. What stands out is the Depth first algorithm is very fast but uses a lot of steps and tiles. Greedy Best first is also fast and uses not too much steps and tiles. The A* always give the minimum number of tiles used but generally it takes more steps to get there. The second heuristic seems to provide the best results but the third one also gives promising results. It is also the best algorithm to handle special sets (Depth First is very bad at that). It is worth noting that the results of Best First algorithm seems to be miscalculated on my part, because of their weirdness.

3 Halloween Challenge

The halloween challenge goal was to do Hill climbing and Simulated Annealing with Hill Climbing in order to solve a set covering problem and then compare the results obtained, and especially the number of call to the fitness function that has to done for each.

In order to achieve this, I designed a fitness function whose goal was to rewards solutions that cover more unique elements from the universe. The valid score is based on the maximum coverage of elements by the selected sets. And cost penalizes solutions based on the total number of selected sets. The goal is to minimize the number of selected sets. And used a the Tweak function designed by the professor that changes the value of a state chosen randomly.

```
def fitness(sets, state):
"""Fitness function"""
cost = np.sum(state)
if np.array(state).any():
    valid = sets[np.array(state), :].max(axis=0).sum()
else:
    valid = 0
return valid, -cost


def tweak(state, size):
"""This function changes the value of a state at one index chosen randomly"""
new_state = state.copy()
index = randint(0, size - 1)
new_state[index] = not new_state[index]
return new_state
```

With this two function I could then, designed the pipeline for the Hill Climbing algorithm. Hill Climbing is a local search Algorithm that start with an arbitrary solution and iteratively move toward an optimal solution in the problem space. The idea is to start with an initial solution, and evaluate it's quality using an objective function (fitness). And then explore neighboring solutions to the current solution. At each step we stay where we are or move to the neighboring solution that improves the objective function the most.

```
def hill_climbing(problem_size, num_sets, density, nb_steps = 100000):
"""Resolve a set covering problem of size problem_size and with num_sets sets"""

# Make the problem
sets = make_set_covering_problem(num_sets, num_sets, density).toarray()
# Initialization
```

```
    initial_state = [False for _ in range(num_sets)]
    current_state = initial_state
    fit = fitness(sets, initial_state)
    print(" The fitness of the initial state is : ", fit)
    visited_states = dict()
    visited_states[tuple(current_state)] = fit
    counter  = 0

    for step in range(nb_steps):
        new_state = tweak(current_state, problem_size) # We change the state a little

        if tuple(new_state) in visited_states: # if we already visited this state
            new_fit = visited_states[tuple(new_state)]

        else:   # if it's the first time we visit this state
            new_fit = fitness(sets, new_state)
            counter += 1 #we add one to the counter of times we used the fitness function

            visited_states[tuple(new_state)] = new_fit #we store the new fitness

        if fit <= new_fit:  # if the fitness is better we keep the new state as the the current state

            current_state = new_state
            fit = new_fit


    return fit, counter
```

Hill climbing tends to work well in general but can be really sensitive to the initialization and lead to local optima. That is why Hill Climbing with Simulated Annealing was introduced. It is an optimization algorithm that combines the local search capabilities of Hill Climbing with probabilistic techniques. Simulated Annealing introduces randomness to the search process, allowing the algorithm to escape local optima and explore a broader solution space.

```
    def hill_climbing_with_simulated_annealing(problem_size, num_sets, density, T = 1, Tmin = 0.001, alpha = 0.95, nb_steps = 1000):

    # Make problem
    sets = make_set_covering_problem(num_sets, num_sets, .3).toarray()

    # Initialization
    initial_state = [False for _ in range(num_sets)]
    current_state = initial_state
    fit = fitness(sets, initial_state)
    visited_states = {}
    visited_states[tuple(current_state)] = fit
    counter  = 0
    global_min = ()
    min_fit = (0, 0)

    while T >= Tmin: # Comdition on the temperature parameter
        for step in range(nb_steps):
            new_state = tweak(current_state, problem_size)

            if tuple(new_state) in visited_states:
                new_fit = visited_states[tuple(new_state)]

            else:
                new_fit = fitness(sets, new_state)
                counter += 1
                visited_states[tuple(new_state)] = new_fit

            if fit <= new_fit:
```

```python
                current_state = new_state
                fit = new_fit
                if min_fit < new_fit:
                    global_min = new_state
                    min_fit = new_fit

            else:
                p = np.exp(-(sum(fit) - sum(new_fit)) / T)
                current_state = choices([current_state, new_state], weights=(1 - p, p), k=1)
    [0]

        T *= alpha  # we decrease the temperature

    return min_fit, counter
```

Then we optimized the parameters by choosing the best α and the best number of steps to optain the best results. We found that $\alpha = 0.7$ and $nb\_steps = 500$ yields the best results.

In the table 1 we can see the comparison of the results obtained with Hill Climbing and Hill Climbing with Simulated Annealing for problem with 0.3 density.

	Hill Climbing	Hill Climbing with SA
number of points : 100	fitness : -8, call to fitness function : 220	fitness : -10, call to fitness function : 760
number of points : 1000	fitness : -15, call to fitness function : 1013	fitness : -19, call to fitness function : 1199
number of points : 5000	fitness : -21, call to fitness function : 6980	fitness : -23, call to fitness function : 4714

Table 1 – Comparison of Hill Climbing and Hill Climbing with Simulated Annealing

What we see is that the fitness of Hill Climbing with Simulated Annealing is better than the one only with Hill Climbing. That makes sense since HC with SA is less likely to get stuck in local minimums. But in the mean time it means that more call to the fitness function are going to be made for small number of points. However that is not true for bigger number of points. We can thus say that HC with SA is an improvement that pays for the shortcomings of HC.

4 Lab2

The goal of Lab2 was to write agents able to play Nim, with an arbitrary number of rows. Nim is a strategical game in which two players take turns removing objects from distinct piles. On each turn, a player must remove at least one object, and may remove any number of objects provided they all come from the same pile. The goal of the game is to avoid taking the last object. We needed to write two agents :

— 1) An agent using fixed rules based on nim-sum (i.e., an expert system)
— 2) An agent using evolved rules using ES

For the first agent, we use different rules to choose the best move. Our goal is to win so if we are in a winning position we want to secure the victory. The two "sure" winning positions are : There is only one remaining line with a number of object >1, hen we take all but one object and we are sure to win as the other player will have to take the last object. The second is there is only two remaining lines with objects, and in at least one of them there is only one object left. In this case by removing all the objects on the line with multiple object, we are sure of winning. Once those two cases are done we use the nim-sum to know if we are on a stable state as our strategy depend on it. If we are, we make only a small perturbation in the row containing the biggest number of object, in hope of staying in a stable state. If we are not in a stable state, but we can make it stable, then it is our best option to do so. And last case, we are not on a stable state and we can't make it stable, then we are in a losing position no matter what we do, so we can choose randomly. The code for the agent is written but we can make it stable, then it is our best option to do so. And last case, we are not on a stable state and we can't make it stable, then we are in a losing position no matter what we do, so we can choose randomly. You can find the code for this agent right under this paragraph.

```
def expert_nim_agent(state: Nim) -> Nimply:
nim = nim_sum(state)
non_null = len([r for r in state.rows if r > 0])

# Case 1: There is only one row with objects, we take all the objects but one --> We win
 if the number of object >1
if non_null == 1:
    max_row = max(state.rows)
    row_index = state.rows.index(max_row)
    return row_index, max_row - 1

# Case 2: There is only 2 row with objects and one with only object in it, we take all
the objects but one in the row with the maximum of object
# --> We win
if non_null == 2 and 1 in state.rows:
    max_row = max(state.rows)
    row_index = state.rows.index(max_row)
    return row_index, max_row

# Case 3: We are on a stable state : we make a little perturbation
if nim == 0:
    max_row = max(state.rows)
    row_index = state.rows.index(max_row)
    return row_index, 1

# Case 4: We are not in the stable state nut we can make it stable by removing some
object
for i, row in enumerate(state.rows):
    if row & nim ^ nim == 0:
        return i, nim

# Case 5: We are in non of the above cases : we are in loosing position. We make a
random move
return pure_random(state)
```

This agent is very effective as we are always winning against the optimal and the pure random strategies already implemented.

Now for the agent using evolving strategies, the idea is to use a different already coded strategy at each move. In order to do that we start from either a random choice of strategy or our best strategy at disposition (here it was optimal) times the number of maximum move allowed in a game. And then the idea is to make it evolved so that at each move we have the best strategy used. In order to make it evolve we use mutation (we modify one strategy randomly) or we do reproduction. We can then fine-tuned the mutation rate to yield the best results. We stop the process when the iterator as reach the maximum number of generation or when our fitness is always of 100%. In the code after we can see the architecture of the code, here the fitness function is defined by the number of match won by the agent over the number of match played by the agent.

```
def train_agent():

# Initialization
pop = generate_population()

# Evolution : for each generation we select the best agents and reproduce them
```

```python
    for generation in range(NB_GENERATIONS):

        # calculate the fitness of each agent
        fit = [fitness(agent) for agent in pop]

        # print the average and max fitness every 10 generations
        if generation % 10 == 0:
            print(" - Generation {} : Average Fitness = {}, Max fitness = {}".format(generation, np.mean(fit), max(fit)))

        # if we have a perfect agent we stop
        if np.mean(fit) == 1:
            break

        # selection of the best agents
        parents = selection(pop, fit)

        # reproduction of the best agents
        new_pop = []
        for i in range(POP_SIZE):
            mut = random.random() < MUTATION_RATE
            if mut:
                new_pop.append(mutation(random.choice(parents)))
            else:
                agent1 = random.choice(parents)
                agent2 = random.choice(parents)
                new_pop.append(reproduction(agent1, agent2))

        pop = new_pop

    return max(pop, key=fitness)


best_agent = train_agent()
```

What we found is that the evolved agent has a winning rate of 97.10 % against optimal strategies. To put it in perspective the random agent has a winning rate of 18.10% against the optimal agent, so we achieve results that are correct. However those results are not better than the strict rule agent using nim-sum. It would've been good to compare one to each other to see which one really is better.

4.1 Reviews given

After finishing, I made two reviews for my classmates. The first one was addressed to Rita Mendes :

"First of all I really like that your code is clean and easy to understand. All the questions are implemented and give results. The only remarks I can raise is that maybe you could have better results in the second question by implementing the adaptive function. Also I feel like that maybe would have helped. Another thing I could suggest is printing the final score after your two evolutionary strategy as it helps to understand how well your algorithm is working. That said it is a really nice work."

The second review I issued was addressed to Donato Lanzilotti :

"First of all, I would like to highlight the fact that this is a really nice, clean and commented notebook : very pleasant and agreeable to reed. However I have a few suggestions of amelioration that I can provide. First, your Donato function even though it is a good strategy is not really what was expected for the first question (algorithm with a set a rules based on the nim-sum) as it doesn't use him sum at all. Then, I feel like you should have really tested your strategy on match against other strategies as the weight in your plot only shows that you have convergence but not that you perform well. That said I really appreciate the insights you made and the plots shown."

4.2 Reviews received

I also was given reviews for my work. First one was given by Rita Mendes Review received by Rita Mendes : "Hello, I have read through your code, and I want to share some thoughts : I like that you kept it simple, straight-forward and self-explanatory ; I see that your mutation function only randomly changes a single gene in the genome to a different strategy. Maybe, as an improvement possibility, consider implementing a more sophisticated strategy that changes the whole genome such as Gaussian mutation as an improvement possibility. Nice job, and good luck for the next lab !".

And the second one was given by Francesco Volpi
"Hello, I find your work interesting. The unconventional individual encoding you've employed appears to be quite effective. Additionally, your code is easily comprehensible, thanks to the comments you've provided (the more, the better). I have a few suggestions : Consider adding more match (be aware : this will grow the computation time), that can be useful to have less variations in the fitness function for the same genotype. Can be useful use different enemies (from weaker to stronger ones) in the fitness function instead of using only the optimal one. That said, this is a good job."

5 Lab3 - ex lab9

The goal of the lab 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.

5.1 Methodology and Results

The idea of the Evolutionary algorithm is that it uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the quality of the solutions. Evolution of the population then takes place after the repeated application of the above operators.

In order to do that I designed multiple functions such as :

— mutation : we change one of the boolean of the sequence on a random place
— swap : we exchange two boolean of the sequence from random places
— reversion : we
— crossover : we make a mix of two sequences coming from different individuals by cutting and jointing them at a point chosen randomly
— double crossover : we make a mix of two sequences coming from different individuals by cutting and jointing them between two points chosen randomly

An illustration of the modification of a binary sequence by each function is provided in the figure 1

Sequence A :	00000000001111111111	
Sequence B :	01010101010101010101	
Mutation of A:	00010000001111111111	Alea = 4
Swap of A:	00010000001111110111	Alea = 4, 18
Reversion of A:	00000001110001111111	Alea = 7, 13
Crossover A/B:	00000000010101010101	Alea = 9
Double crossover:	00000101010101011111	Alea = 4, 16

FIGURE 1 – Illustration olf the effects of the functions

I also designed two different selection functions in order to chose a parent. One is chosing the individual with the best fitness and the other one is choosing the last individual with a higher fitness than an individual chosen randomly. Those functions are visible rigth under.

```
def selection(population, fitnesses, fitness):
    """ This function is selecting individual from the population with probability their fitness."""
    ind = choices(population, weights=fitnesses, k=1)[0]
    return ind

def selection2(population, fitnesses, fitness, k=2):
    """ This function is selecting ."""
    ind = choices(range(len(population)), k=k)
    best = ind[0]
    for i in ind:
        if fitnesses[i]> fitnesses[best]:
            best = i
    return population[best]

def xover(ind1, ind2):
    """ This function is performing crossover between two individuals."""
    gen_length = min(len(ind1), len(ind2))
    g = randint(0, gen_length - 1)
    if randint(0, 1) == 0:
        return  ind1[:g] + ind2[g:]
    else:
        return ind2[:g] + ind1[g:]

def double_xover(ind1, ind2):
    """ This function is performing two-point crossover on two individuals."""
    gen_length = min(len(ind1), len(ind2))
    g1 = randint(0, gen_length - 1)
    g2 = randint(g1, gen_length - 1)

    if randint(0, 1) == 0:
        return  ind1[:g1] + ind2[g1:g2] + ind1[g2:]
    else:
        return  ind2[:g1] + ind1[g1:g2] + ind2[g2:]

def mutation(ind):
    """ This function is selecting one random gene and return a mutation of it."""
    gen_length = len(ind)
    new_ind = ind.copy()
    g = randint(0, gen_length-1)
    new_ind[g] = 1 - new_ind[g]
    return new_ind

def swap(ind):
    """ This function is selecting two random genes and return a swap of them."""
    gen_length = len(ind)
    new_ind = ind.copy()
    g1, g2 = tuple(choices(range(0, gen_length), k=2))
    new_ind[g1] = ind[g2]
    new_ind[g2] = ind[g1]
    return new_ind

def reversion(ind):
    """ This function is selecting two random genes and return a reversion of the genome between them."""
    gen_length = len(ind)
    new_ind = ind.copy()
    pos1 = randint(0, gen_length - 2)
    pos2 = randint(pos1, gen_length - 1)
    new_ind[pos1:pos2] = list(reversed(ind[pos1:pos2]))
    return new_ind
```

Once we had all this possible twist at each generation we can wrote the evolution function. To construct the new generation from the previous one we select n_d individuals mutate, swap, reverse, or keep them as they are with a certain rate, and then we make them "reproduce" with crossover or double-crossover. We add those new individuals to our existing population and then we keep the n_p individuals with the best fitness. We store the fitness of the best individual and then we re-start the process.

Because we want as less call to the fitness function as possible we consider early stop in two cases : we reached the best fitness (1.0) or the fitness doesn't improve after x generation where x is a parameter to fine tune.

```
def evolution(population, fitness, early_stop = 10, select = 1):

if select == 1: #choice of selection function
    select = selection
else:
    select = selection2


stop = 0
best_fit = float("-inf")
list_fit = []
fitnesses = [fitness(i) for i in population]
for generation in range(NUM_GENERATIONS):
    descendants = []

    for _ in range(DESCENDANTS_SIZE):
        p1 = select(population, fitnesses, fitness)
        if random() < MUTATION_RATE:
            p1 = mutation(p1)
        elif random() < SWAP_RATE:
            p1 = swap(p1)
        elif random() < REVERSION_RATE:
            p1 = reversion(p1)
        else :
            p1 = p1


        if random() < CROSSOVER_RATE:
            p2 = select(population, fitnesses, fitness)
            d = xover(p1, p2)
        elif random() < DOUBLE_CROSSOVER_RATE:
            p2 = select(population, fitnesses, fitness)
            d = double_xover(p1, p2)
        else:
            d = p1

        descendants.append(d)


    new_population = population + descendants
    fitnesses = [fitness(i) for i in new_population]
    combined = list(zip(new_population, fitnesses))
    sorted_combined = sorted(combined, key=lambda x: x[1], reverse=True)
    new_population = [elem[0] for elem in sorted_combined]
    population = new_population[:POP_SIZE]
    fitnesses = [elem[1] for elem in sorted_combined]
    fitnesses = fitnesses[:POP_SIZE]
    best_indiv = population[0]

    if fitnesses[0] > best_fit:
        best_fit = fitnesses[0]
        stop = 0

    else:
        stop += 1
```

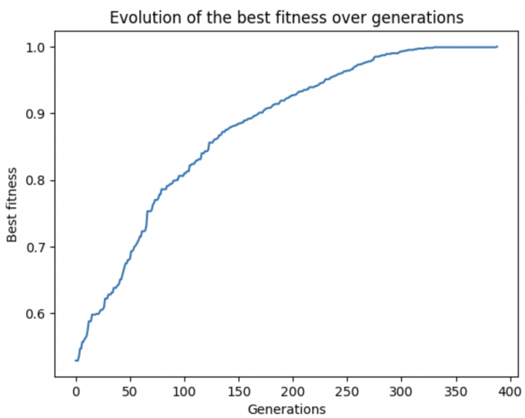
```

        list_fit.append(best_fit)

        if best_fit == 1:
            print(f"Best individual: {best_indiv}")
            print(f"Best fitness: 100%")
            print(f"Converged at generation {generation}")
            break
        if stop >= early_stop:
            print(f"Converged at generation {generation}")
            print(f"Best individual: {best_indiv}")
            print(f"Best fitness: {fitnesses[0]:.2%}")
            break
        if generation == NUM_GENERATIONS - 1:
            print(f"Best individual: {best_indiv}")
            print(f"Best fitness: {fitnesses[0]:.2%}")
            print(f"Converged at generation {generation}")

    plt.plot(range(0, generation + 1), list_fit)
    plt.xlabel("Generations")
    plt.ylabel("Best fitness")
    plt.title("Evolution of the best fitness over generations")
    plt.show()
```

The results obtained after training with parameters MUTATION_RATE = 0.5, SWAP_RATE = 0.5, REVERSION_RATE = 0.5, CROSSOVER_RATE = 0.6, DOUBLE_CROSSOVER_RATE = 0.5. They are visible in the table 2 and in the figure 2

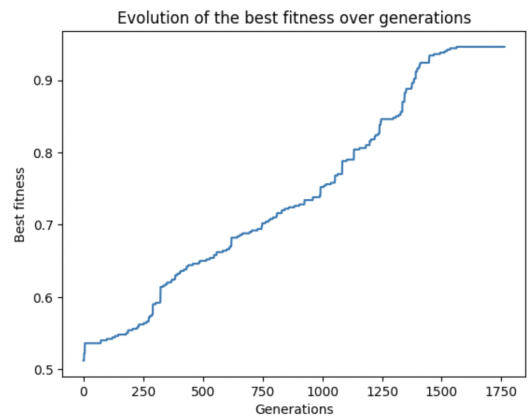
Problem size	best fitness	number of iterations
1	100%	388
2	94.60%	1763
5	43.19%	996
10	29.00%	1481

Table 2 – Comparison of the EA results on different problem size

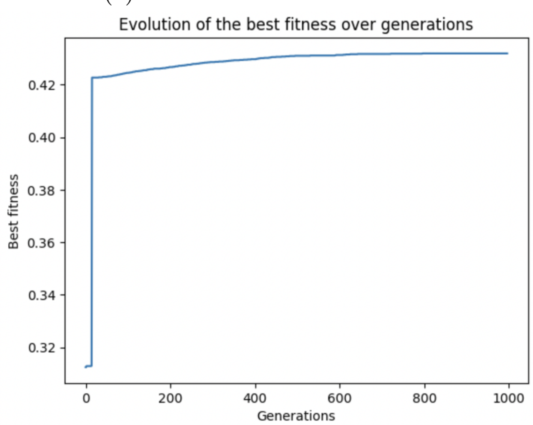
Finally I fine-tuned my parameters by testing different rates in order to find the better ones.



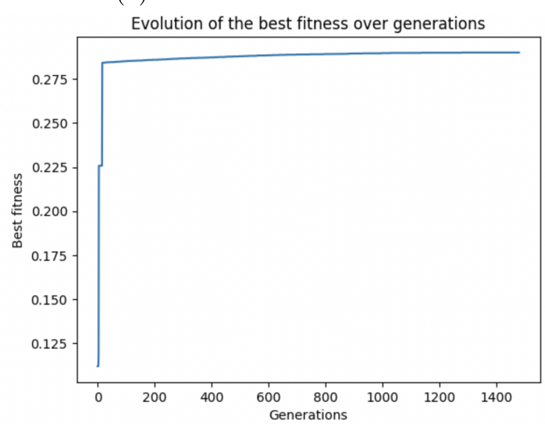
(a) Problem size = 1



(b) Problem size = 2



(c) Problem size = 5



(d) Problem size = 10

FIGURE 2 – Evolution of the fitness as a function of iteration for different problem sizes

```
LEN_GEN = 1000
POP_SIZE = 50
PROBLEM_SIZE = [1, 2, 5,10]
DESCENDANTS_SIZE = 25
NUM_GENERATIONS = 5000


for mutation_rate in [0.5, 0.6,0.7]:
    for swap_rate in [0.3,0.5,0.7]:
        for reversion_rate in [0.3,0.5,0.7]:
                for crossover_rate in [0.5,0.6,0.7]:
                    for double_crossover_rate in [0.5,0.6,0.7]:
                        print(f"Mutation rate: {mutation_rate}")
                        MUTATION_RATE = mutation_rate
                        print(f"Swap rate: {swap_rate}")
                        SWAP_RATE = swap_rate
                        print(f"Reversion rate: {reversion_rate}")
                        REVERSION_RATE = reversion_rate
                        print(f"Crossover rate: {crossover_rate}")
                        CROSSOVER_RATE = crossover_rate
                        print(f"Double crossover rate: {double_crossover_rate}")
                        DOUBLE_CROSSOVER_RATE = double_crossover_rate
                        fitness = lab9_lib.make_problem(5)
                        evolution(population, fitness, early_stop=200,select=2)
                        print("Number of fitness call : ", fitness.calls)
                        print("------------------------------")
```

5.2 Reviews given

I gave one review to Giuseppe Nicola Natalizio. "Hey, nice code. I really appreciate the fact that you tried multiple method in order to get the best results. The later, also seem satisfactory. However I think that the number of calls to the fitness function could and should be reduced as we want to avoid having too many of them. Looking at the graph we can see that for the first two techniques there is no evolution of fitness after a certain time, maybe implementing an early stop would have save you time and memory. I really liked the idea of the migration phenomenon and didn't thought of it myself, well done, I could also suggest that you try implementing other form of transformation such as double mutation, double crossover, reversion of a part of the genome as maybe it can boost your fitness faster (and maybe even higher). Nonetheless, it is great work."

And one to Luca Pastore : "Hey. I found your results obtained in this lab really great! But, I would have appreciated a code better commented, your readme file was a great idea but the presentation is not as readable as we would like. Also I think that some of your libraries imports are not necessary so you should clean it. I think that you could try different things to have better results. First, try to find other way to make the genome change, you could try multiple, mutation, different crossover, reversion, or other process inspired by biology. I also think that your crossover function shouldn't try to force the cross over to always increase fitness because this can lead to local maxima. It's normal and good that you explore the surroundings of the actual solution. Anyway you have a lot of great idea and effective code writing skills so overall it's great work."

5.3 Reviews received

I only received a review from Marcello Vitaggio :

"Hi!

Your code is quite extensive and solid! Your selection methods, crossovers, mutations, and reversions seem well-defined and clear in their purposes.

The evolution function is well-structured. However, consolidating the population sorting and truncation into a single step can save computation. Instead of sorting and then slicing the population, consider using a selection algorithm (like tournament selection). You also used a convergence criteria to early stopping based on no more fitness increment, I should have used it too, nice touch.

Your approach of iterating through different parameter combinations and problem sizes is a great way to fine-tune your algorithm.

Regarding the code outputs, your code generates lengthy output due to the iterations and print statements, making it challenging to spot some code snippets within the outputs."

6 Lab4 - ex Lab10

The goal of lab 4 is to implement a Reinforcement Learning Player for a game with the best winning ratio. In order to do this I had to design a reward function named fitness function. My fitness function is actually quite simple : it is equal to 1 if we win the game, -1 if we loose and 0 if it's draw.

The entire implementation of the RL class is visible here :

```
class RLearning:
    def __init__(self, alpha, epsilon, d):
        self.Q = defaultdict(float)
        self.alpha = alpha
        self.epsilon = epsilon #control the exploration
        self.d = d

    def get_Q(self, state, action):
        state_key = (tuple(state.x), tuple(state.o))
        return self.Q[(state_key, action)]


    def choose_action(self, state, available):
        """ This functions serves to choose the actions in function of the actual
    probabilities"""

        if random.random() < self.epsilon: # we explore epsilon percent of the time
            return random.choice(available)
        else:
            Q_vals= [self.get_Q(state, action) for action in available] # we retrieve the
    probabilities
            max_Q = max(Q_vals)
            best_moves = [i for i in range(len(available)) if Q_vals[i] == max_Q] # we get
    the best moves
            index = random.choice(best_moves)
            return available[index]

    def update_Q(self, state, action, reward, next_state, available):
    """ This function update the probabilities."""

        state_key = (tuple(state.x), tuple(state.o))
        next_Q_vals = [self.get_Q(next_state, next_action) for next_action in available]
        max_next_Q = max(next_Q_vals, default=0.0)
```

```python
        self.Q[(state_key, action)] = (1 - self.alpha) * self.Q[(state_key, action)] + self.alpha * (reward + self.d * max_next_Q)



def fitness(pos: State, player):
    """ Evaluate state: +1 first player wins """

    if check_win(pos.x):
        if player == 'x':
            return 1
        else:
            return -1
    elif check_win(pos.o):
        if player == 'o':
            return 1
        else:
            return -1
    else:
        return 0


def train(nb_train, alpha, epsilon, d, player):
    """ This function trains the agent"""

    agent = RLearning(alpha, epsilon, d)
    for i in range(nb_train):
        state = State(set(), set())
        available = list(range(1, 10))
        player_turn = 'x'
        while available and not check_win(state):
            if player_turn == player:
                action = agent.choose_action(state, available)
            else:
                action = random.choice(available)

            previous_state = deepcopy(state)
            if player_turn == 'x':
                state.x.add(action)
            else:
                state.o.add(action)
            available.remove(action)
            reward = fitness(state, player)
            agent.update_Q(previous_state, action, reward, state, available)
            player_turn = 'o' if player_turn == 'x' else 'x'
        player = 'x' if player == 'o' else 'o'

    return agent

def optimization(epsilons, alphas, ds, player,nb_train=NB_TRAIN):
        """ This function serves to optimize the parameters alpha, epsilon and d to yield the best results"""
    best_agent = None
    best_percentage_win_agent = 0

    for epsilon in epsilons:
        for alpha in alphas:
            for d in ds:
                agent = train(nb_train=nb_train, alpha=alpha, epsilon=epsilon, d=d, player=player)
                win_agent, win_random, no_win = count_win(agent, player,  fitness)
                tot_games = win_agent + win_random + no_win
                percentage_win_agent = (win_agent / tot_games) * 100

                if percentage_win_agent > best_percentage_win_agent: #if the current agent is better than the previous one, we update the best agent and the best percentage
                    best_percentage_win_agent = percentage_win_agent
                    best_agent = agent
```

```
    return best_agent
```

My RL agent achieve some great results. During the optimization we find that the best epsilon is 0.1, the best alpha is 0.6 and the best discount Factor is 0.8. Then for the testing we test the RL Player against a Random Player on 1000 games. The RL agent wins 91.10000000000001 % of the time, while the percentage wins of the random player is 5.1%, and the percentage of draws is 3.8 %. So that means that the percentage of wins of the agent with respect to the random player : 94.670 %. This very encouraging ! However it would have been better to compare the agent against a non random player as of course we expect the random player to be kinda dumb.

6.1 Reviews

I received one review from Luca Catalano "Hey Roxane ! Fantastic work on your recent lab ! The code is really well-organized and clean. I've got a few suggestions for you :

Consider adding a readme file where you can showcase your results and detail the process. Make sure your results are clearly visible within the .ipynb file. Experiment with implementing a different strategy in the Q learning algorithm, maybe explore a strategy that aims to create multiple potential paths to victory. Have you thought about creating a human player version and testing it against the algorithm ? I hope these suggestions are helpful for your progress, and wishing you the best of luck on your exam ! Luca"

I took it into account and added a readme file to make it clearer. I also received a review from Donato Lanzillotti

"First of all, congratulations for what you have done in LAB10. Some suggestions that could be useful also in the future.

The code is very clear, but not commented. I would suggest you to add a *readme.txt* file to explain in detail your strategy and also the organization of your code.

Showing the results you obtained could help the comprehension of the effectiveness of your strategy.

From what I understood, it seems that your agent is trained also as first player. It could useful, in order to increase the robustness of your player, training the agent also as second player.

The training of the agent is done only against a random player. It could be useful, in order to learn new strategies, training your agent also against an 'expert' player or in general against a player that does not play completely random.

Overall, you have done a great job. Sorry if I have misunderstood something."

7 Project

The goal of the final project was to design a player that would win against random in a Quixo game.

In order to do so, I designed two players. The first one is using reinforcement learning and the second one a Minmax strategy. Both players have been tested against a random player as well as between themselves.

7.1 Implementation

I used multiple class, one for each type of player (RL, Minmax and Random) to implement properly the strategies.

7.1.1 Reinforcement Learning

Here is an overview of the RL player class :

```
class RLPlayer(Player):
    def __init__(self) -> None:
        super().__init__()
        self.moves = [Move.TOP, Move.BOTTOM, Move.LEFT, Move.RIGHT] # available moves
        self.col = range(5) #size of the game
        self.row = self.col
        self.len_move = 16 #number of pieces on the border (playable)
        self.ind = [[(i,j) for i in self.row] for j in self.col] # coordinates of each position
        self.last_move: [[Move]] # last move chosen for each position
        self.picks: [[int]] # number of times each position was chosen to play
        self.pos_proba: [[float]] #probability of choosing each position
        self.move_proba: [[[float]]] # probability pf each move for each position
        self.id = -1 #actual player
        self.training_size = 15000
        self.testing_size = 3000
        self.lr = 0.2 #learning_rate
        self.wins = 0 # number of wins achieved
        self.nb_games = 0 # number of training games played
        self.last_reward = 0 # last reward obtained on a training game
        self.epsilon = 1.0 #exploration parameter
        self.decay_rate = 0.05 # rate of decay for the epsilon parameter


    def is_playable(self, position: tuple[int, int], player_id: int, board) -> bool:
        '''This function check that the piece we want to use is within the border limit and is not occupied by the opponent'''

        border = False
        available = False
        # Check error of position
        if (position[0] >=5 or position[1]>=5 or position[0]<0 or position[1]<0):
            print("There is an error the row/col cannot be greater or equal to 5 ")
            border = False
        # Check border
        elif (position[0] == 0 or position[0] == 4 or position[1] == 0 or position[1] == 4):
            border = True
        # Check availability
        if (board[position] == player_id or board[position] == -1):
            available = True
        return (border and available)



    def get_move(self):
            """ This function predict the best piece and move with respect to the probability"""
            flat_proba = [item for row in self.pos_proba for item in row]
```

```
            proba = softmax(flat_proba) # we want them to sum up to one
            flat_ind = [item for row in self.ind for item in row]
            valid_idx = [0, 1, 2, 3, 4, 5, 9, 10, 14, 15, 19, 20, 21, 22, 23, 24] # we can only move pieces on the border
            table_idx = [i for i in  range(len(flat_proba))]
            ind = np.random.choice(table_idx, p=proba)
            while ind not in valid_idx:
                ind = np.random.choice(table_idx, p=proba)

            row = flat_ind[ind][0]
            col = flat_ind[ind][1]
            moves = [Move.TOP, Move.BOTTOM, Move.LEFT, Move.RIGHT]
            move = np.random.choice(moves, p = softmax(self.move_proba[row][col]))
            return row, col, move




    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        """ This function is determining the next move"""

        self.id = game.get_current_player() #update the actual player
        if random.uniform(0,1) < self.epsilon:   #random exploration
            row = random.randint(0,4)
            col = random.randint(0,4)
            while not self.is_playable((row,col),self.id, game.get_board()):
                row = random.randint(0,4)
                col = random.randint(0,4)

            if (row == 0 and col == 0):
                move =  random.choice([Move.BOTTOM, Move.RIGHT])
            elif (row == 0 and col == 4):
                move = random.choice([Move.BOTTOM, Move.LEFT])
            elif (row == 4 and col == 0):
                move = random.choice([Move.TOP, Move.RIGHT])
            elif (row == 4 and col == 4):
                move = random.choice([Move.TOP, Move.LEFT])
            elif (row == 0):
                move = random.choice([Move.BOTTOM, Move.LEFT, Move.RIGHT])
            elif (row == 4):
                 move = random.choice([Move.TOP, Move.LEFT, Move.RIGHT])
            elif (col == 0):
                move = random.choice([Move.TOP, Move.BOTTOM, Move.RIGHT])
            elif (col == 4):
                move = random.choice([Move.TOP, Move.BOTTOM, Move.LEFT])

            self.picks[row][col] += 1 #1
            self.last_move[row][col] = move

        else:
            row, col, move = self.get_move()
            while game.get_board()[row][col] == 1 - self.id:
                row, col, move = self.get_move()
            self.picks[row][col] += 1
            self.last_move[row][col] = move

        return (col,row), move





    def init_proba(self):
        """ This function initialize every attributs"""
        self.pos_proba = [[0 for _ in self.col] for _ in self.row] # proba of each position
        self.picks = [[0 for i in self.row] for j in self.col] # Number of picks for each position
        self.last_move = [[0 for i in self.row] for j in self.col] # last move of each
```

```
position
        self.move_proba = [[[0 for _ in range(4)] for _ in self.col] for _ in self.row] # proba of TOP, BOTTOM, LEFT, RIGHT for each position

        for i in range(5):
            self.pos_proba[0][i] = 1/self.len_move
            self.pos_proba[4][i] = 1/self.len_move
            self.pos_proba[i][0] = 1/self.len_move
            self.pos_proba[i][4] = 1/self.len_move

        # if we are in a corner there is only two directions where we can move
        self.move_proba[0][0] = [0, 0.5, 0, 0.5]
        self.move_proba[0][4] = [0, 0.5, 0.5, 0]
        self.move_proba[4][0] = [0.5, 0, 0, 0.5]
        self.move_proba[4][4] = [0.5, 0, 0.5, 0]

        self.last_move[0][0] = random.choice([Move.BOTTOM, Move.RIGHT])
        self.last_move[0][4] = random.choice([Move.BOTTOM, Move.LEFT])
        self.last_move[4][0] = random.choice([Move.TOP, Move.RIGHT])
        self.last_move[4][4] = random.choice([Move.TOP, Move.LEFT])

        # when we on the border but not in the corner, we can move in 3 directions
        for i in range(1,4):
                self.move_proba[0][i] = [0, 1/3, 1/3, 1/3] # we can't move to the top
                self.move_proba[4][i] = [1/3, 0, 1/3, 1/3] # we can't move to the bottom
                self.move_proba[i][0] = [1/3, 1/3, 0, 1/3] # we can't move to the left
                self.move_proba[i][4] = [1/3, 1/3, 1/3, 0] # we can't move to the right

                self.last_move[0][i] = random.choice([Move.BOTTOM, Move.LEFT, Move.RIGHT])
                self.last_move[4][i] = random.choice([Move.TOP, Move.LEFT, Move.RIGHT])
                self.last_move[i][0] = random.choice([Move.TOP, Move.BOTTOM, Move.RIGHT])
                self.last_move[i][4] = random.choice([Move.TOP, Move.BOTTOM, Move.LEFT])



    def init_random(self):
        """ This function initialize every attributs randomly"""
        self.pos_proba = [[0 for _ in self.col] for _ in self.row] # proba of each position
        self.picks = [[0 for i in self.row] for j in self.col] # Number of picks for each position
        self.ind = [[(i,j) for i in self.row] for j in self.col] # coordinates of each position
        self.last_move = [[0 for i in self.row] for j in self.col] # last move of each position
        self.move_proba = [[[0 for _ in range(4)] for _ in self.col] for _ in self.row] # proba of TOP, BOTTOM, LEFT, RIGHT for each position

        for i in range(5):
            self.pos_proba[0][i] = random.random()
            self.pos_proba[4][i] = random.random()
            self.pos_proba[i][0] = random.random()
            self.pos_proba[i][4] = random.random()

        self.last_move[0][0] = random.choice([Move.BOTTOM, Move.RIGHT])
        self.last_move[0][4] =  random.choice([Move.BOTTOM, Move.LEFT])
        self.last_move[4][0] = random.choice([Move.TOP, Move.RIGHT])
        self.last_move[4][4] = random.choice([Move.TOP, Move.LEFT])

        self.move_proba[0][0] = [0,random.random(),0,random.random()]
        self.move_proba[0][4] = [0,random.random(),random.random(),0]
        self.move_proba[4][0] = [random.random(),0,0,random.random()]
        self.move_proba[4][4] = [random.random(),0,random.random(),0]

        for i in range(1,4):

            self.last_move[0][i] = random.choice([Move.BOTTOM, Move.LEFT, Move.RIGHT])
            self.last_move[4][i] = random.choice([Move.TOP, Move.LEFT, Move.RIGHT])
            self.last_move[i][0] = random.choice([Move.TOP, Move.BOTTOM, Move.RIGHT])
```

```
            self.last_move[i][4] = random.choice([Move.TOP, Move.BOTTOM, Move.LEFT])

            self.move_proba[0][i] = [0, random.random(), random.random(), random.random()] # we can't move to the top
            self.move_proba[4][i] = [random.random(), 0, random.random(), random.random()] # we can't move to the bottom
            self.move_proba[i][0] = [random.random(), random.random(), 0, random.random()] # we can't move to the left
            self.move_proba[i][4] = [random.random(), random.random(), random.random(), 0] # we can't move to the right


    def clear_picks(self):
        """This functions serves to clean all the picks (ie. set it back to 0)"""
        for i in self.row:
            for j in self.col:
                self.picks[i][j] = 0



    def training(self):
        """ This function serves for the training of the weights"""
        self.init_proba()
        self.epsilon=1
        win_count = 0
        for _ in  tqdm(range(self.training_size)):
            g = Game()
            player1 = self
            player2 = RandomPlayer()
            winner = g.play(player1, player2)
            self.update_weights(winner)
            rew = self.reward_function(winner)
            self.update_epsilon(rew)
            if winner == 0:
                win_count+=1

        self.save_weights()
        return (win_count/self.training_size)*100


    def update_weights(self, winner):
        """ This function modify the proba in order to take into account the reward after the game"""
        if winner == 0:
            los = 1
        else :
            los = 0
        for i in self.row:
            for j in self.col:
                if  self.picks[i][j] >= 1: # if the piece was used
                    reward = ((los - self.pos_proba[i][j])*self.lr)/self.picks[i][j]
                    self.pos_proba[i][j] = self.pos_proba[i][j] + reward
                    index_moves = self.moves.index(self.last_move[i][j])
                    self.move_proba[i][j][index_moves] = self.move_proba[i][j][index_moves] + reward
                    self.picks[i][j]= 0 # we put it back to 0
        return reward

    def reward_function(self, winner):
        """ This function serves as a reward/fitness function that evolves after each game (it stays between 0 and 1)"""
        self.nb_games +=1
        if winner == 0:
            self.wins +=1

        return self.wins/self.nb_games

    def update_epsilon(self, reward):
```

```
        """ This function adapt epsilon the exploration parameter based on the reward
    obtained"""
        if reward > self.last_reward : # Decrease epsilon for better performance
            epsilon = self.epsilon - self.decay_rate
        else :
            epsilon = self.epsilon - self.decay_rate

        epsilon = max(0.1, epsilon)
        self.epsilon = min (1, epsilon)




    def save_weights(self):
        """ This function saves the weights after training"""

        fw = open('proba_pos', 'wb')
        pickle.dump(self.pos_proba, fw)
        fw.close()
        fw = open('proba_move', 'wb')
        pickle.dump(self.move_proba, fw)
        fw.close()


    def load_weights(self, file1, file2):
            """ This function use file to initialize weights"""
            self.init_proba()
            self.epsilon = 0
            fr = open(file1, 'rb')
            self.pos_proba = pickle.load(fr)
            fr.close()

            fr = open(file2, 'rb')
            self.move_proba = pickle.load(fr)
            fr.close()
```

So as we can see we choose the best strategy depending on the probabilities of the weight matrix. At each game during the training phase we update the probabilities according to the result of the last game that is the input of the reward function. At the beginning we have to initialize the probabilities and to do so, two options were designed (init_random and init_proba). The first options initialize the weights randomly while the second initialize weights with the idea that some piece and move are more probable to be good. I actually use mainly the second one as it requires less training to yield good results (and training is very long). We also have functions to save and load the weights, it very practical as with those we don't have to train the agent each time we just load the weights and move matrix and we are good, this means a smaller computational time.

7.1.2 MinMax

As a second player I decided to implement the MinMax strategy. The objective of MinMax is to determine the optimal move for a player by considering the potential outcomes of each possible move and minimizing the maximum potential loss.

```
class MinmaxPlayer(Player):
    def __init__(self, depth) -> None:
        super().__init__()
        self.depth = depth
        self.board_size = 5

```

```
    def all_possible_moves(self, player_id: int, board) -> list[list[tuple[int, int], Move]]:
            '''This function returns all the possible moves that can be done by a player in a given board'''

            all_possible_moves = []
            for row in range(5):
                for col in range(5):
                    available = False
                    border = False
                    # Check border
                    if (row == 0 or row == 4 or col == 0 or col == 4):
                        border = True
                    # Check availability
                    if (board[(row, col)] == player_id or board[(row, col)] == -1):
                        available = True

                    if border and available:
                        if (row == 0 and col == 0):
                            all_possible_moves.append([(row, col), Move.BOTTOM])
                            all_possible_moves.append([(row, col), Move.RIGHT])
                        elif (row == 4 and col == 0):
                            all_possible_moves.append([(row, col), Move.BOTTOM])
                            all_possible_moves.append([(row, col), Move.LEFT])
                        elif (row == 0 and col == 4):
                            all_possible_moves.append([(row, col), Move.TOP])
                            all_possible_moves.append([(row, col), Move.RIGHT])
                        elif (row == 4 and col == 4):
                            all_possible_moves.append([(row, col), Move.TOP])
                            all_possible_moves.append([(row, col), Move.LEFT])
                        elif (col == 0):
                            all_possible_moves.append([(row, col), Move.BOTTOM])
                            all_possible_moves.append([(row, col), Move.LEFT])
                            all_possible_moves.append([(row, col), Move.RIGHT])
                        elif (col== 4):
                            all_possible_moves.append([(row, col), Move.TOP])
                            all_possible_moves.append([(row, col), Move.LEFT])
                            all_possible_moves.append([(row, col), Move.RIGHT])
                        elif (row == 0):
                            all_possible_moves.append([(row, col), Move.TOP])
                            all_possible_moves.append([(row, col), Move.BOTTOM])
                            all_possible_moves.append([(row, col), Move.RIGHT])
                        elif (row == 4):
                            all_possible_moves.append([(row, col), Move.TOP])
                            all_possible_moves.append([(row, col), Move.BOTTOM])
                            all_possible_moves.append([(row, col), Move.LEFT])
                        else:
                            all_possible_moves.append([(row, col), Move.TOP])
                            all_possible_moves.append([(row, col), Move.BOTTOM])
                            all_possible_moves.append([(row, col), Move.LEFT])
                            all_possible_moves.append([(row, col), Move.RIGHT])

            return all_possible_moves



    def heuristic(self, game: "BetterGame") -> int:
        """ This function is designed to return the heuristic value of the current board."""

        winner = game.check_winner()
        if winner == game.current_player_idx:
            return 1
        elif winner == 1 - game.current_player_idx:
            return -1
        else:
            return (game.best_sequence() - 2.5) / 5


```

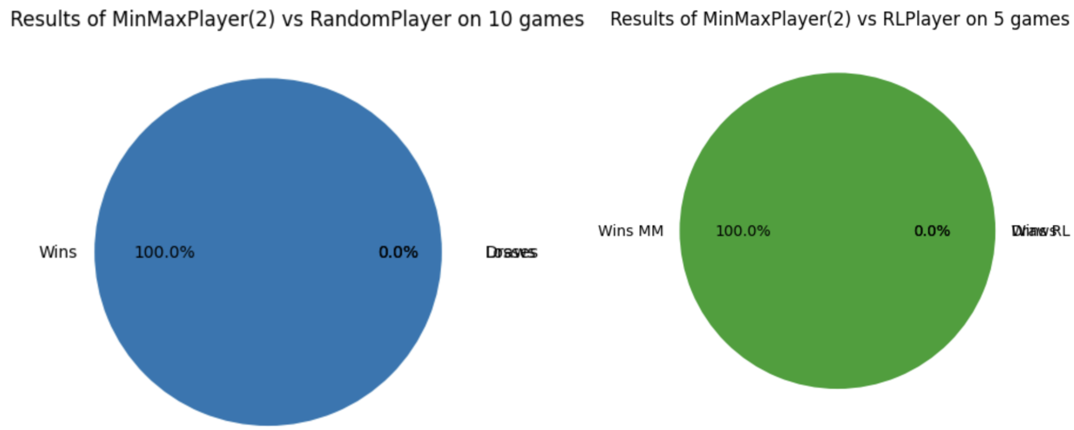
```
    def minmax(self, game: "BetterGame", depth, is_max) -> int:
        """ This function is designed to return the best move that can be done by the current player."""

        if game.check_winner() != -1 or depth >= self.depth:
            return self.heuristic(game)

        possible_moves = self.all_possible_moves(game.current_player_idx, game.get_board())

        if is_max:  # Maximizer
            best_score = float("-inf")
            for move in possible_moves:
                game_new = deepcopy(game)
                player = 1 - game_new.current_player_idx
                game_new.move(move[0], move[1], player)
                current_score = self.minmax(game_new, depth + 1, False)
                best_score = max(current_score, best_score)
            return best_score
        else:                   # Minimizer
            best_score = float("inf")
            for move in possible_moves:
                game_new = deepcopy(game)
                player = 1 - game_new.current_player_idx
                game_new.move(move[0], move[1], player)
                current_score = self.minmax(game_new, depth + 1, True)
                best_score = min(current_score, best_score)
            return best_score


    def make_move(self, game: "BetterGame") -> tuple[tuple[int, int], Move]:
        """ This function is designed to return the best move that can be done by the current player."""

        best_score = float("-inf")
        best_move = None
        possible_moves = self.all_possible_moves(game.current_player_idx, game.get_board())
        for move in possible_moves:
            game_new = deepcopy(game)
            game_new.move(move[0], move[1], game_new.current_player_idx)
            current_score = self.minmax(game_new, 0, False)
            if current_score > best_score: # we keep the move with the best minmax score
                best_score = current_score
                best_move = move

        return best_move[0], best_move[1]
```

So as we can see there is need of implementing only 4 functions. First *all_possible_moves*, that gives back every possible action that the player can do at time t. Then the heuristic function which return a number between $[-1, 1]$ depending on who is winning the game and the best sequence made by the player. Once that is done we only have to implement the *minmax* function that gives us a score to evaluate the quality of the move. And then the *make_move* function is choosing the move with the highest minmax score along all the possible moves.

7.2 Results

The RL PLayer is winning around 80 % of the time. Moreover it is very quick to play, we need a few seconds to play 3000 games.

The Minmax player is really powerful as it wins almost every time with depth $= 2$. However it is longer to play with, on average 1 minute per game. Even against the RL Player MinMax wins all the time. The results are visible in Figure 3. The only problem of Minmax is when the two players use the same strategies it doesn't work and we often find ourselves with only draws. Maybe also I could've used another heuristic as the one chosen takes into account only the longuest sequence but it is interesting to have multiple long sequence that could potentially win in order to "trap" the adversary.



(a) Minmax Player against Random Player

(b) Minmax Player against RL Player

FIGURE 3 – Statistics of Minmax player