Report Computational Intelligence

Antonio De Cinque, s303503

February 7th, 2023

Contents

1 Introduction

As a student in the **Computational Intelligence** course, I found myself constantly engaged and intrigued by the material being covered. The course delved deep into the various techniques and approaches used to create intelligent systems, and provided a hands-on understanding of how these methods can be applied to solve real-world problems. I was particularly fascinated by the intricacies of **evolutionary algorithms** applied to games. I enjoyed learning about how these techniques can be used to optimize and improve the performance of intelligent systems on such a wide range of activities.

Throughout the course, I took detailed notes, completed various labs and projects, and provided reviews of my classmates' work. In this report, I will share my insights and experiences from the course, highlighting the key concepts and techniques I learned, as well as the projects and labs I completed. My course notes, lab write-ups, and reviews will provide an in-depth look at the material covered in the course, and demonstrate my understanding of the subject matter.

One of the **highlights** of the course was the **final project**, where my team and I had the opportunity to apply the concepts and techniques we learned to solve a challenging problem. The project, which involved developing a series of algorithms to solve the game of Quarto, required us to think critically and creatively, and to work collaboratively as a team. The end result has been a collaboration that not only showcased our understanding of **computational intelligence**, but also our ability to solve real-world problems using advanced techniques.

Overall, the course has been an **enriching experience** for me, and I have come away with a deeper understanding and appreciation of the field of computational intelligence. Through this report, I will have the opportunity to share my insights and experiences with my professor and classmates, and to demonstrate the knowledge and skills I have gained during the course. I **enjoyed the course** and would recommend it to anyone interested in learning about this field.

1.1 How this report is structured

For each laboratory that I did during the course, I will highlight 4 subsections:

- **Task** - The problem that has been assigned
- **Implementation** - Once I understand the requirements of a task, I start working on its implementation. I try to consider the best approach, write

clean and efficient code, and test it to ensure it meets the desired requirements

- **Received Reviews** - Receiving constructive feedback from others helps me identify areas for improvement and provided me with a fresh perspective. I am grateful for the time and effort my colleagues take to review my code and provide me with valuable insights.
- **Given Reviews** - Just as I appreciate receiving reviews of my code, I also take the opportunity to review the code of my colleagues. I believe that reviewing others' work is a great way to learn.

All the code in this report can be found in my GitHub repo: `https://github.com/antoniodecinque99/computational_intelligence42`

1.2 A special thank

> “I promise you that I will learn from my mistakes”
>
> *Fix You, song by Coldplay*

Most of all, I want to take a moment to thank all the colleagues who took their time to review my code.

As a future software developer, having a colleague review my code has been incredibly important to me, as it gives me the opportunity to learn from my mistakes.

I value very much the constructive criticism and feedback that my colleagues provided. They often catch mistakes that I might have overlooked and offer suggestions for improvement.

When a colleague points out an issue in my code, I take it as an opportunity to deepen my understanding of the topic and find ways to avoid making the same mistake again. I find it incredibly rewarding to see how I will apply what I learn to future projects and improve as a long-life learner.

Finally, I want to say thanks to the Professor Giovanni Squillero for fostering such a collaborative and learning-oriented environment in this course. Your approachable and humorous demeanor, and your dedication to our growth and development is greatly appreciated, and shall not be forgotten.

2 Laboratory 1 - Set Covering Problem

2.1 Task

Given a number **N** and some lists of integers P = $(L_0, L_1, L_2, ..., Ln)$, determine, if possible, S = $(L_{s0}, L_{s1}, L_{s2}, ..., Lsn)$ such that each number between **0** and **N - 1** appears in at least one list

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

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

2.2 Implementation

`https://github.com/antoniodecinque99/computational_intelligence42/tree/main/lab1`

This solution has been developed in collaboration with Andrea D'Attila (303339).

For the solution a **greedy algorithm** has been adopted. Here the full code, which is going to be explained for each part.

```python
import logging
import copy

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

    if (new_elements == 0):
        return float('inf')

    res = common_elements/new_elements
    return res

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

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

    starting_x = lists.pop(0)

    sol = list()
    sol.append(starting_x)

    flat_sol = list(starting_x)
```

```
    nodes = 1

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

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

        covered |= set(most_promising_x)

    w = len(flat_sol)

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

The results obtained by this greedy algorithm could be used to implement an optimized version of the breadth-first search algorithm: a node should not be processed if its cost is lower than the cost obtained with the greedy solution, because, for sure, it leads to a worse solution. Although the greedy algorithm does not always find an optimal solution, it does find a lower bound for it!

In order to explain the algorithm, we will consider an example of problem with $N = 10$ and seed $= 42$.

```
import random

def problem(N, seed = 42):
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in
        ↪  range(random.randint(N // 5, N // 2))))
        for n in range(random.randint(N, N * 5))
    ]
```

The above mentioned code generates a list like this:

```
[[0, 4], [1, 2, 3], [9, 6], [0, 1], [8, 9, 3], [8, 3], [0, 3, 4, 7,
9], [4, 5, 6], [1, 3, 5], [1, 6], [0, 9, 4, 5], [8, 1, 6], [9, 3, 5],
[0, 3], [1, 3, 6], [2, 5, 7], [1, 3, 4, 9], [8, 2, 3], [3, 4, 5, 6,
8], [0, 3], [1, 3, 4, 6], [3, 6, 7], [2, 3, 4], [9, 6], [8, 2, 3, 7],
```

```
[0, 1], [9, 2, 6], [6], [8, 0, 4, 1], [1, 4, 5, 6], [0, 4, 7], [8,
1, 4], [2, 5], [9, 5], [0, 1, 3, 4, 5], [9, 3], [1, 7], [8, 2], [8,
2, 7], [8, 9, 3, 6], [4, 5, 6], [8, 1, 3, 7], [0, 5], [0, 9, 3], [0,
3], [0, 5], [8, 3], [8, 2, 3, 7], [1, 3, 6, 7], [5, 6]]
```

After generating the problem, the algorithm will sort the lists by length, in descending order (i.e. from longest to shortest).

As a first element of the solution, the algorithm will take the longest list.

```python
def my_greedy(N):
    goal = set(range(N))

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

    starting_x = lists.pop(0)

    sol = list()
    sol.append(starting_x)

    flat_sol = list(starting_x)
    nodes = 1
    (...)
```

```
[[0, 3, 4, 7, 9], [3, 4, 5, 6, 8], [0, 1, 3, 4, 5], [0, 9, 4, 5], [1,
3, 4, 9], [1, 3, 4, 6], [8, 2, 3, 7], [8, 0, 4, 1], [1, 4, 5, 6], [8,
9, 3, 6], [8, 1, 3, 7], [8, 2, 3, 7], [1, 3, 6, 7], [1, 2, 3], [8,
9, 3], [4, 5, 6], [1, 3, 5], [8, 1, 6], [9, 3, 5], [1, 3, 6], [2, 5,
7], [8, 2, 3], [3, 6, 7], [2, 3, 4], [9, 2, 6], [0, 4, 7], [8, 1, 4],
[8, 2, 7], [4, 5, 6], [0, 9, 3], [0, 4], [9, 6], [0, 1], [8, 3], [1,
6], [0, 3], [0, 3], [9, 6], [0, 1], [2, 5], [9, 5], [9, 3], [1, 7],
[8, 2], [0, 5], [0, 3], [0, 5], [8, 3], [5, 6], [6]]
```

At this point the solution is `sol = [0, 3, 4, 7, 9]`. From now on, the algorithm will look for the most promising lists to add.

This is done by using the following heuristic function `h(x)`.

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

    if (new_elements == 0):
        return float('inf')

    res = common_elements/new_elements
    return res
```

For each list `x` in the problem set we compute:

- common elements between `x` and `sol`
- new elements that are present in `x` but not in `sol`

The cost of adding such element will be:

$$cost(x) = \frac{common \quad elements}{new \quad elements}$$

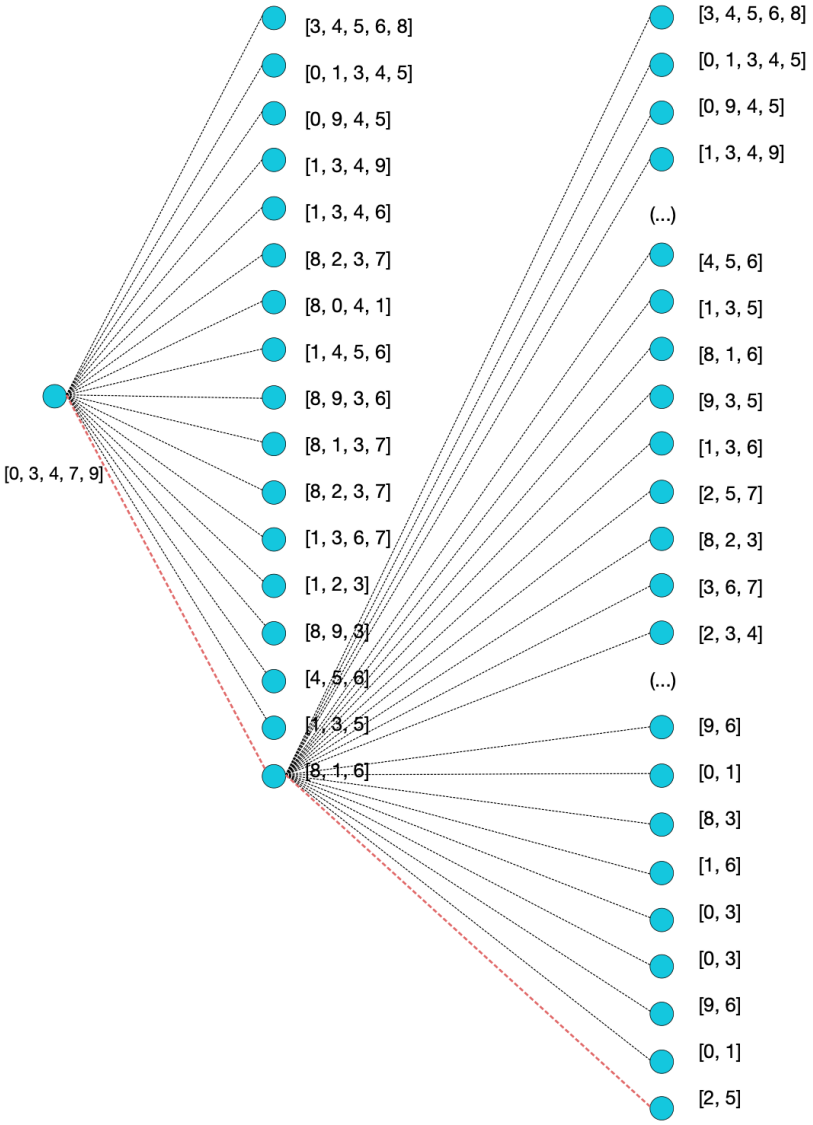
Then, the `x` corresponding to the minimum cost will be added to the solution set. In case of two elements that have same cost, the **longest one** is taken (i.e. the first one that appears in the ordered list of lists).

This is the case because the min function will take the first occurrence of the minimum. Given that the list of lists is ordered according to length, the first list will be taken.

```
def my_greedy(N):
    goal = set(range(N))

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

    starting_x = lists.pop(0)

    sol = list()
    sol.append(starting_x)

    flat_sol = list(starting_x)
    nodes = 1

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

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

        covered |= set(most_promising_x)

    w = len(flat_sol)
```

In the example provided with `N = 10` and `seed = 42`, the greedy algorithm will give an optimal solution, with minimum `w = 10` (i.e. 10 elements contained in total):

```
[[0, 3, 4, 7, 9], [8, 1, 6], [2, 5]]
```



```python
logging.getLogger().setLevel(logging.INFO)
logging.getLogger().setLevel(logging.INFO)

for N in [5, 10, 20, 100, 500, 1000]:
    my_greedy(N)
```

This is the output for $N \in [5, 10, 20, 100, 500, 1000]$

- `Greedy solution for N=5 :  w=5 (bloat=0%) - optimal`
- `Greedy solution for N=10 :  w=10 (bloat=0%) - optimal`
- `Greedy solution for N=20 :  w=24 (bloat=20%)`
- `Greedy solution for N=100 :  w=182 (bloat=82%)`
- `Greedy solution for N=500 :  w=1262 (bloat=152%)`

2.3 Received Reviews

Wow

`https://github.com/antoniodecinque99/computational_intelligence42/issues/1`

I looked at your code and I am really surprised. You've adopted a really good way of solving the problem, and it's certainly an inspiration to me as well. The way of finding the cost for the various steps is perfect and I think it also works very well and it shows by the fact that the solutions with N=5 and N=10 are very good. The process also seems quite scalable since you managed to find solutions even for high values of N (where you find the solutions better than mine).

The idea of starting with the list with the largest number of elements I think is a good one given your choice of weight. What's more, the code is written in a few lines and this certainly helps to understand the process that leads you to find the solution, perhaps at the limit I would have added a couple more comments. I also tried running your code together with the debugger and indeed the times are very short. You don't always find the optimal solution, but I think it is the best compromise between speed and optimal.

I think the one you implemented is the best greedy solution I can think of. The weight should be what allows us to get as close as possible to the optimum.

Luca Marcellino, ***s292950***

Simple, clear and precise

`https://github.com/antoniodecinque99/computational_intelligence42/issues/2`

I had also thought of implementing a function which would make it possible to choose the list which would add the least cost to the solution and that is exactly what had to be done. The way to choose the h(x) function is very smart but also the way to find the numerator and the denominator and you are even avoiding errors like divisions by zero.

Very striking solution even if the coded algorithm does not seek to find the right solution (like Dijkstra for example). This way of resolving the problem is better because as the professor says in the course we will not be able to find the exact solution or it will take too much time.

Great job to both of you 10/10 :)

StanickLG, s307340

Well done!

`https://github.com/antoniodecinque99/computational_intelligence42/issues/3`

Your solution is one of the cleanest and most readable I've seen so far (at least between the ones I saw), there are no comments in the code but it's so clear that they're not needed and anyway the README is already perfectly clear and exhaustive. The cost function is very "simple", but at the same time very efficient.

One little thing I may say, but it's not fundamental in your case, is the reduction of the search space removing duplicates between the initial list of lists, me and my team didn't think about it before seeing the Professor doing it in his solution. It could reduce (if efficiently done) a little bit the computational time in finding the min between the results of the application of the cost function to the list of lists, avoiding to repeat it multiple times for the same candidate list, but your algorithm is already very fast and finds a very good solution up to N=1000 (w=2878 is really good considering the computational time and the number of explored nodes).

Starting from N=20 it doesn't find the optimal solution (w=23), but as I said it's so well working up to N=1000 that I think the computational time and the possibility of finding a solution is a good compromise.

vmask25

Peer Review

https://github.com/antoniodecinque99/computational_intelligence42/issues/4

At First Sight: Presentation and Readability

The README file is well done and self-explanatory. It helps understand rapidly the idea behind the algorithm used, and even how the heuristic function works. You might also have explained why you choose that specific heuristic function.

The absence of comments and a slighty complex code syntax slow down a bit the comprehension of the code.

Algorithmic Approach and Code Analysis

The algorithm you used is a Greedy, and it uses a good heuristic function (considering the results). After the first lines of initialization, we found the while-loop. Here you take the best list (found using heursitic function h), which is the list with the lowest cost. Then you remove it from the main list of the problem.

You add this list to flat_sol through .extend. The variable flat_sol seems to be a single list of integers, containing only the elements of the lists which compose the current solution sol; it is used to efficiently compute the number of integers w in the solution, after the while loop ended.

Then you update the flag covered by adding those new integers gained in the current cycle, through the |= between the sets.

The algorithm is quite interesting, because it implements a good heuristic function, and it manages to find solutions also for N = 50 and higher; it also manages to keep a "quite low" bloat (how far the solution is from the optimal one) considering that is a greedy algorithm.

Personal Suggestion:

It would be interesting trying with some other heuristic function, to understand if there exsist better ones to use. Maybe you can implement a system to explore not only the next step, but maybe the next m steps, starting from a finite bunch of k best candidate (i.e at cycle i=6 you select the best k=3 lists using the function h, and then you try exploring m=4 steps further starting from each of these 3 best candidates. Then you can select the best between those 3 further explorations). You can do it randomly, at fixed n cycle or at each cycle. It would be interesting analyzing obtained results.

SamuelePino

2.4 Given Reviews

“ https://github.com/angelicaferlin/computational_intelligence_2022/issues/2

Your search function does not return the optimal result for small values of N, such as N = 5. I also tried to formulate problems with different seeds, and it does not seen to work either.

I think that the problem lies in the lines of code in which you organise the state space during the search:

```
1 def count_unique_elem_opt(current_state, l):
2 """Returns the number of unique elements between the
  ↪  lists also regarding duplicates of numbers."""
3
4     return (len(set(l).difference(set(current_state)))*
      ↪  (len(set(l)) == len(l)))
5
6 (...)
7
8 state_space = sorted(state_space, key=lambda l:
  ↪  count_unique_elem_opt(current_state, l))
```

First off, as it has been stated in another review, if a list has even a single duplicated element, everything becomes 0, without a distinction between repetitions and missed elements.

Secondly, I suggest you to re-write this function, because it is written in a way difficult to understand.

As a cost function, you could try my idea: a function that returns the number of common elements / the number of new elements.

And in case of having two elements with same cost, take the longer list. ”

Me

“ https://github.com/FlavioPatti/Computational-Intelligence_2022-23/issues/2

You basically adapted the same Dijkstra algorithm that was shown at lecture. It is in line with the requests and behaves the way it is supposed to: it reaches the optimal solution in reasonable time if N is small.

For big values of N we would never be able to find the exact solution or it would take too much time. Therefore, next time you can think of:

- First applying a Greedy algorithm to find an upper bound of optimal solutions

- And then use your Dijkstra algorithm to narrow even more the space of search. You had my same idea for calculating the costs, and I think it is the best way for such evaluation. I also think that the formulation of the state as a subset of the indexes of the list of lists is well thought-out.

Given that the code is very similar to the ones explained by the professor, it has been fairly easy to understand. But, if someone would read it without having followed the course it would take more time for them. Therefore, the README should have been more verbose. ”

Me

INTERESTING FACT: When I reviewed Flavio's code, I did not know him at the time. A few months later, we randomly ended up in the same team for developing the final project. Not only did we get a productive collaboration, but also a good friendship.

3 Laboratory 2 - Set Covering Problem with GA approach

https://github.com/antoniodecinque99/computational_intelligence42/tree/main/lab2

3.1 Task

Given a number **N** and some lists of integers P = $(L_0, L_1, L_2, ..., Ln)$, determine, if possible, S = $(L_{s0}, L_{s1}, L_{s2}, ..., Lsn)$ such that each number between **0** and **N - 1** appears in at least one list

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

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

Try to use a Genetic Algorithm approach

3.2 Implementation

https://github.com/antoniodecinque99/computational_intelligence42/tree/main/lab2

This solution has been developed in collaboration with Andrea D'Attila (303339).

Here the full code, which is going to be explained for each part.

After running the GA using different parameters, we have chosen the ones which gave us the best results in a reasonable time.

We used `POPULATION_SIZE = 1500` and `OFFSPRING_SIZE = 1000`. In our Genetic Algorithm we used both mutation and crossover: for each generation we generate mutated offsprings with `MUTATION_RATE = 0.3`.

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

POPULATION_SIZE = 1500
OFFSPRING_SIZE = 1000
NUM_GENERATIONS = 200
MUTATION_RATE = 0.3
```

Starting from the lists generated by problem using seed = 42, we removed duplicated lists, which are useless in order to find a solution.

```
def distinct(list):
   result = []
   for l in list:
      if l not in result:
         result.append(l)
   return result
```

3.2.1 Genome

The data structure which contains the genome and the fitness is the following:

```
    Individual = namedtuple("Individual", ["genome", "fitness"])
```

The first step of the algorithm consists in generating the population. We have used a **binary representation**:

- 0 in the i-th position of the genome means that the i-th list of the problem has not been included
- 1 in the i-th position of the genome means that the i-th list of the problem has been included

To generate the population, we created genomes containing all 0s, which means that we start our algorithm having coverage and weight equal to 0. We also tried to initialize a random population, rather than a blank one, but soon realized that this leads to worse results

The algorithm will be tested with different values of N.

```
for N in [5, 10, 20, 100, 500, 1000]:
    population = list()
    lists = problem(N, 42)
    lists = distinct(lists)

    # randomic population, which leads to worse results
    # for genome in [tuple([random.choice([1, 0]) for _ in
    ↪  range(len(lists))]) for _ in range(POPULATION_SIZE)]:
    #       population.append(Individual(genome,
    ↪  compute_fitness(genome, lists)))

    for genome in [tuple([0 for _ in range(len(lists))]) for _ in
    ↪  range(POPULATION_SIZE)]:
```

```
        population.append(Individual(genome,
        ↪  compute_fitness(genome, lists)))

    my_genetic_algorithm(N, lists, population)
```

Then, in next generations, we will try to add some 1s randomically and see the evolution.

3.2.2 Fitness function

Genomes are evaluated using a fitness function. In this case, we have assigned to each genome a tuple `(c,-w)`, in which:

- the first value represent the `coverage` of that genome, which corresponds to the number of different values covered by that particulare genome
- the second value represent the `weight` of that genome, which is the total number of values included in that genome, considering duplicates too

We have added a minus to the weight, so that when comparing two genomes, the best is the one with the higher coverage, but if two genomes have the same coverage, the best is the one with the lower weight.

```
def coverage(genome, lists):
    # How many numbers are covered
    values = set()
    for i in range(len(genome)):
        if genome[i] == 1:
            values |= set(lists[i])
    return len(values)

def weight(genome, lists):
    # Weight of the solution
    return sum(genome[i]*len(lists[i]) for i in range(len(lists)))

def compute_fitness(genome, lists):
    return (coverage(genome, lists), -weight(genome, lists))
```

3.2.3 Crossover and mutation

For crossover and mutation we used the standard functions.

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

def cross_over(g1, g2):
    cut = random.randint(0, len(g1))
    return g1[:cut] + g2[cut:]

def mutation(g):
    point = random.randint(0, len(g) - 1)
    return g[:point] + (1 - g[point],) + g[point + 1 :]
```

3.2.4 Genetic Algorithm

```python
def my_genetic_algorithm(N, lists, population):
for generation in range(NUM_GENERATIONS):
    offspring = list()
    for i in range(OFFSPRING_SIZE):
        if random.random() < MUTATION_RATE:      # MUTATION
            p = tournament(population)           # promising genome
            o = mutation(p.genome)               # genome (mutated)
        else:                                    # CROSSOVER
            p1 = tournament(population)          # promising genome1
            p2 = tournament(population)          # promising genome2
            o = cross_over(p1.genome, p2.genome)

        f = compute_fitness(o, lists)
        offspring.append(Individual(o, f))


    population += offspring
    population = sorted(population, key=lambda i: i.fitness,
    ↪  reverse=True)[:POPULATION_SIZE]

    best_so_far = population[0]
    if best_so_far.fitness[0] == -best_so_far.fitness[1] and
    ↪  best_so_far.fitness[0] == N: # earlier stopping condition
        break

print(f"N = {N} : weight: {-best_so_far.fitness[1]}, bloat:
↪  {(-best_so_far.fitness[1]-N)/N*100}, #generations:
↪  {generation+1}")
```

Little optimization in the end: if we obtain an optimal solution (`coverage == weight == N`) before reaching the maximum number of generations, we can stop the algorithm.

3.2.5 Results

This is the output for $N \in [5, 10, 20, 100, 500, 1000]$ As you can notice, the results are similar to the greedy algorithm developed in the previous Laboratory!

- `N = 5 : weight: 5, bloat: 0.0, #generations: 3 - optimal`
- `N = 10 : weight: 10, bloat: 0.0, #generations: 7 - optimal`
- `N = 20 : weight: 24, bloat: 20, #generations: 200`
- `N = 100 : weight: 202, bloat: 102, #generations: 200`
- `N = 500 : weight: 1610, bloat: 222, #generations: 200`
- `N = 1000 : weight: 3600, bloat: 260, #generations: 200`

For time constraints, we managed to have 200 generations. I suspect that this algorithm could reach the best solution also with higher N, if given more generations!

3.3 Received Reviews

Clear and concise, very well written

`https://github.com/antoniodecinque99/computational_intelligence42/issues/5`

Hi Antonio! I chose your code for my review because I was impressed with your solution, so I wanted to know all the details of how you achieved it.

First of all, good job on the readme, which explains everything you used very clearly, and also gives me an idea of what to look for in the code.

I found it interesting how you select your initial population by having a genome with all 0's, therefore relying mostly on mutation to develop the first few generations. With this, I think you might be able to achieve even better solutions (they're still very good as is!) by having a higher mutation chance in the first few generations, maybe by having it as a function that decreases exponentially so it settles rather quickly. I am also curious as to how you chose this particular mutation chance of 0.3.

Did you experiment with more crossover options? I found with my code that one-point crossover wasn't the best function for my purposes, so maybe if you have the time it could be worth trying different approaches to the function.

Your code is clear and concise, very well written, and I didn't find any issues with it. It seems that your choices of parameters, fitness and crossover led you to a great solution, so I want to congratulate you on that! I am sure it took a lot of work and experimenting, so it's always nice to see when it pays off.

Have a nice day and keep up the good work ;)

”

Federico Boscolo, ***feurode46***

3.4 Given Reviews

“ **Very clean and modular. Very easy to understand**

`https://github.com/feurode46/computational-intelligence-2022/issues/5`

First of all, I like the way you structured your code.

Secondly, I also like the fact that you used a different type of crossover. Normally I have seen crossover functions in which you cut genomes in random points and then you put them together. In your case, instead, you take random parts of genomes, and then you put them together.

```
1 def cross_over(self, g1, g2):
2     g3 = list()
3     g3.extend([random.choice([g1[i], g2[i]]) for i in
      ↪  range(self.problem_size)])
4     return tuple(g3)
```

Here are a few suggestions:

Try to omit the duplicates, to deal with a reduced scale of the problem (reducing computation and time). With a high value of N, your code explodes, and you get very high bloat. I do not really understand the reason why, but I think it may be because of the crossover function. Have you tried using the “classical” one that we saw in lecture? Or a combination of both?

I think your code was very clean and modular. Very easy to understand. I would be curious to see the code with the (mu, lambda) evolution strategy that you didn't post!

”

Me

A few points

https://github.com/FabioSofer/Computational_Intelligence2022/issues/2

The code is very clear, and also the readme file is easy to understand.

A few points:

- I do not understand well your fitness function. In this way, you condense the fitness in a single value and you lose the priority of the parameters that contribute to fitness. Also, for N = 500, you get a negative value. Is it a mistake?

  I suggest you instead to return a tuple, in which you have the count of numbers covered and the opposite of the weight of the solution, both to maximise. You can add a minus to the weight, so that when comparing two genomes, the best is the one with the higher coverage. If two genomes have the same coverage, the best should be the one with the lower weight.

```python
def fitnessFunc(genome):
    # Compute in parallel the coverage and the length of
    ↪  the genome solution
    tuple_set = set()
    genome_lenght=0
    for index,i in enumerate(genome):
        if i==1:
            tuple_set |= set(PROBLEM_SPACE[index])
            genome_lenght+=len(PROBLEM_SPACE[index])
    genome_coverage=len(tuple_set)
    # Adding a penalty if not valid (coverage < 100%)
    malus = 0
    if genome_coverage != N[GOAL_N]:
        malus = genome_lenght
    return K*genome_coverage - genome_lenght - malus
```

- Have you tried to make offspring size higher? I think it would generate more "randomness" and you can improve more in less time.

Good job anyways!

Me

4 Laboratory 3 - Policy Search for Nim game

https://github.com/antoniodecinque99/computational_intelligence42/tree/main/lab3

4.1 Tasks and template code

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

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

- Task 3.1: An agent using fixed rules based on **nim-sum** (i.e., an **expert system**)
- Task 3.2: An agent using evolved rules
- Task 3.3: An agent using minmax
- Task 3.4: An agent using reinforcement learning

For reference, I am writing here the template code for the the `Nim` and `Nimply` classes.

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

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

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

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

    def assign_rows(self, rows):
        self._rows = list(rows)
        return self

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

```
21
22      @property
23      def k(self) -> int:
24          return self._k
25
26      def nimming(self, ply: Nimply) -> None:
27          row, num_objects = ply
28          assert self._rows[row] >= num_objects
29          assert self._k is None or num_objects <= self._k
30          self._rows[row] -= num_objects
31
32      def game_over(self) -> bool:
33          return sum(self._rows) == 0
34
35  def active_rows(state: Nim) -> int:
36      return sum(o > 0 for o in state.rows)
37
```

Then we have the functions that compute the current state (starting from current data)...

```
1   def cook_status(state: Nim) -> dict:
2       cooked = dict()
3       cooked["possible_moves"] = [
4           (r, o) for r, c in enumerate(state.rows) for o in range(1,
            ↪  c + 1) if state.k is None or o <= state.k
5       ]
6       cooked["active_rows_number"] = sum(o > 0 for o in state.rows)
7       cooked["shortest_row"] = min((x for x in enumerate(state.rows)
        ↪  if x[1] > 0), key=lambda y: y[1])[0]
8       cooked["longest_row"] = max((x for x in enumerate(state.rows)),
        ↪  key=lambda y: y[1])[0]
9       cooked["nim_sum"] = nim_sum(state)
10
11      brute_force = list()
12      for m in cooked["possible_moves"]:
13          tmp = deepcopy(state)
14          tmp.nimming(m)
15          brute_force.append((m, nim_sum(tmp)))
16      cooked["brute_force"] = brute_force
17
18      return cooked
```

...and the expert / random strategies (used for evaluation).

```
1   def nim_sum(state: Nim) -> int:
2       *_, result = accumulate(state.rows, xor)
3       return result
```

```

def pure_random(state: Nim) -> Nimply:
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    num_objects = random.randint(1, state.rows[row])
    return Nimply(row, num_objects)
```

```
NUM_MATCHES = 100
NIM_SIZE = 4

def evaluate(strategy1: Callable, strategy2: Callable, nim_size=NIM_SIZE) -> float:
    opponent = (strategy1, strategy2)
    won = 0

    for m in range(NUM_MATCHES):
        nim = Nim(nim_size)
        player = 0
        while nim:
            ply = opponent[player](nim)
            nim.nimming(ply)
            player = 1 - player
        if player == 1:
            won += 1
    return won / NUM_MATCHES
```

4.2 Task 3.1 - Fixed Rules

For the first task, I have defined some fixed rules and evaluated their performances against a random player. They are ordered according to performance against random player.

1. Pick minimum amount of matches in the highest row
2. Pick maximum amount of matches in the highest row
3. Pick maximum amount of matches in the lowest row $\approx$ Pick minimum amount of matches in the lowest row

```
def pick_minimum_from_highest_row(state: Nim) -> Nimply:
    """Pick always the minimum possible number of the highest row"""
    possible_moves = [(r, o) for r, c in enumerate(state.rows)
                      for o in range(1, c + 1)]
```

```
    return Nimply(*max(possible_moves, key=lambda m: (-m[0],
    ↪  m[1])))

evaluate(pick_minimum_from_highest_row, pure_random)
# 80% average win rate
```

```
def pick_maximum_from_highest_row(state: Nim) -> Nimply:
    """Pick always the maximum possible number of the highest
    ↪  row"""
    possible_moves = [(r, o) for r, c in enumerate(state.rows)
                      for o in range(1, c + 1)]
    return Nimply(*max(possible_moves, key=lambda m: (m[0], m[1])))

evaluate(pick_maximum_from_highest_row, pure_random)
# 65% average win rate
```

```
def pick_minimum_from_lowest_row(state: Nim) -> Nimply:
    """Pick always the minimum possible number of the lowest row"""
    possible_moves = [(r, o) for r, c in enumerate(state.rows)
                      for o in range(1, c + 1)]
    return Nimply(*max(possible_moves, key=lambda m: (-m[0],
    ↪  -m[1])))


def pick_maximum_from_lowest_row(state: Nim) -> Nimply:
    """Pick always the maximum possible number of the lowest row"""
    possible_moves = [(r, o) for r, c in enumerate(state.rows)
                      for o in range(1, c + 1)]
    return Nimply(*max(possible_moves, key=lambda m: (m[0],
    ↪  -m[1])))

evaluate(pick_maximum_from_lowest_row, pure_random)
evaluate(pick_minimum_from_lowest_row, pure_random)
# 45% average win rate for both
```

Intuitively, it makes sense that picking the minimum from the highest row is better.

Essentially, it is generally convenient to have more matches available, so that it is more likely for the agent to take the last match (and therefore win).

To further investigate this hypothesis, I tried to combine all these rules two by two in a `count_and_decide` function:

- If the active rows number is even, use the first chosen rule
- If the active rows number is odd, use the second chosen rule

Here are the best performing versions of `count_and_decide` function:

```
def count_and_decide(state: Nim) -> Nimply:
    if active_rows(state) % 2 != 0:
        return pick_maximum_from_highest_row(state)
    else:
        return pick_minimum_from_lowest_row(state)
evaluate(count_and_decide, pure_random)
# 85% average win rate
```

```
def count_and_decide(state: Nim) -> Nimply:
    if active_rows(state) % 2 != 0:
        return pick_maximum_from_highest_row(state)
    else:
        return pick_maximum_from_lowest_row(state)
evaluate(count_and_decide, pure_random)
# 85% average win rate
```

```
def count_and_decide(state: Nim) -> Nimply:
    if active_rows(state) % 2 != 0:
        return pick_maximum_from_highest_row(state)
    else:
        return pick_minimum_from_highest_row(state)
ate(count_and_decide, pure_random)
# 70% average win rate
```

```
def count_and_decide(state: Nim) -> Nimply:
    if active_rows(state) % 2 == 0:
        return pick_minimum_from_lowest_row(state)
    else:
        return pick_minimum_from_highest_row(state)
evaluate(count_and_decide, pure_random)
# 85% average win rate
```

```
def count_and_decide(state: Nim) -> Nimply:
    if active_rows(state) % 2 == 0:
        return pick_maximum_from_lowest_row(state)
    else:
        return pick_minimum_from_highest_row(state)
evaluate(count_and_decide, pure_random)
# 85% average win rate
```

After examining these examples, a new conclusion dawned on me. The best perfoming versions of `count_and_decide` had one thing in common: push the game to have a particular final configuration. We would want, in the end, to have the a configuration in which we have only two rows, composed of one match each:

| <— opponent is obliged to take just one match!

| <—

This can be achieved by having a modified `count_and_decide` function:

- When the agent plays first:
  - If the active rows are even, chose a random row and pick all but one element
  - If the active rows are odd, chose a random row and pick all elements
- When the agent plays second, do the opposite:
  - If the active rows are odd, chose a random row and pick all but one element
  - If the active rows are even, chose a random row and pick all elements

```python
def pick_all_elements(state: Nim) -> Nimply:
    row = random.choice([r for r, c in enumerate(state.rows)
    ↪  if c > 0])
    return Nimply(row, state.rows[row])

def pick_all_but_one_elements(state: Nim) -> Nimply:
    row = random.choice([r for r, c in enumerate(state.rows)
    ↪  if c > 0])
    return Nimply(row, state.rows[row] - 1)
```

To my great suprise, the `count_and_decide` rule seemed to beat every time the random and expert players! I asked my reviewers for suggestions on this, and got some feedback.

```python
#Agent playing first
def count_and_decide(state: Nim) -> Nimply:
    if active_rows(state) % 2 == 0:
        return pick_all_but_one_elements(state)
    else:
        return pick_all_elements(state)

print(evaluate(count_and_decide, pure_random))      # 100% win rate
print(evaluate(count_and_decide, optimal_strategy)) # 100% win rate
```

```python
#Agent playing second
def count_and_decide(state: Nim) -> Nimply:
    if active_rows(state) % 2 != 0:
```

```
        return pick_all_but_one_elements(state)
    else:
        return pick_all_elements(state)

print(evaluate(pure_random, count_and_decide))       # 100% win rate
print(evaluate(optimal_strategy, count_and_decide)) # 100% win rate
```

4.3 Task 3.2 - Evolved rules

For this task, I reused the same structure of evolutionary algorithm that I wrote in the second laboratory. The strategy depends on a probability `p`, which is effectively the genome. This is the genetic algorithm stucture:

```
Individual = namedtuple("Individual", ["genome", "fitness"])
POPULATION_SIZE = 50
NUM_GENERATIONS = 100
OFFSPRING_SIZE = 300
```

```
def my_genetic_algorithm(population, strategy):
    for generation in range(NUM_GENERATIONS):
        offspring = list()
        for i in range(OFFSPRING_SIZE):
            if random.random() < 0.2:
                p = tournament(population)
                o = mutation()
            else:
                # promising genome 1
                p1 = tournament(population)
                # promising genome 2
                p2 = tournament(population)
                o = crossover(p1.genome, p2.genome)
            f = compute_fitness(o, strategy)
            offspring.append(Individual(o, f))

        population += offspring
        population = sorted(population, key=lambda i: i.fitness,
        ↪  reverse=True)[
            :POPULATION_SIZE]

        best_so_far = population[0]
        if (generation % 5 == 0):
            print(
                f"Generation #{generation}\tGENOME (Probability):
                ↪  {best_so_far.genome}\tFITNESS:
                ↪  {best_so_far.fitness}")
```

The strategy will use `p` as a treshold:

- Choose a random row and pick all but one element as strategy, with probability `p`
- Otherwise, choose a random row and pick all elements

```
def evolution(evolvable_strategy):
    population = list()
    for _ in range(POPULATION_SIZE):
        genome = str(random.random())
        population.append(Individual(
            genome, compute_fitness(genome, evolvable_strategy)))

    my_genetic_algorithm(population, evolvable_strategy)

def make_strategy(genome: str) -> Callable:
    def evolvable(state: Nim) -> Nimply:

        if random.random() < float(genome):
            ply = pick_all_elements(state)
        else:
            ply = pick_all_but_one_elements(state)
        return ply

    return evolvable

evolution(make_strategy)
```

The fitness function consists in the winning rate against the optimal strategy. The crossover consists in the mean of the probabilities (genomes) of two chosen genomes. The mutation consists in changing such value completely.

```
def compute_fitness(genome, strategy):
    return evaluate(strategy(genome), optimal_strategy)

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

def crossover(g_1, g_2):
    g_c = (float(g_1) + float(g_2))/2
    return str(g_c)

def mutation():
    return str(random.random())

```

The solution reached immediately a high fitness value and then stopped growing: the typical early convergence problem! At first I was not aware of the problem, but a review kindly cleared it out for me.

```
Generation #0 GENOME (Probability):  0.3916160795171203 FITNESS: 0.33
Generation #5 GENOME (Probability):  0.40491509329877645 FITNESS: 0.38
Generation #10 GENOME (Probability):  0.5178968736576 FITNESS: 0.4
Generation #15 GENOME (Probability):  0.5178968736576 FITNESS: 0.4
Generation #20 GENOME (Probability):  0.5178968736576 FITNESS: 0.4
Generation #25 GENOME (Probability):  0.4614059834781882 FITNESS: 0.41
Generation #30 GENOME (Probability):  0.4614059834781882 FITNESS: 0.41
Generation #35 GENOME (Probability):  0.4614059834781882 FITNESS: 0.41
Generation #40 GENOME (Probability):  0.4614059834781882 FITNESS: 0.41
Generation #45 GENOME (Probability):  0.4614059834781882 FITNESS: 0.41
Generation #50 GENOME (Probability):  0.4614059834781882 FITNESS: 0.41
```

The problem could be solved by using a $\mu + \lambda$ evolutionary algorithm.

This is a type of evolutionary computation that uses a combination of both μ (parent) and λ (offspring) individuals in a population to generate new offspring in each iteration. It is a form of genetic algorithm where the μ individuals represent the best individuals in the population and λ individuals represent the newly generated offspring. The algorithm continuously updates the population with the best individuals until a satisfactory solution is found.

Here a pseudo code implementation of it

Algorithm 1 $\mu + \lambda$ Evolutionary Algorithm

$\mathbf{P} = \{\mathbf{x}_1, \ldots, \mathbf{x}_\mu\} \leftarrow$ Initial population
repeat
  Evaluate fitness of individuals in $\mathbf{P}$ using $h(\mathbf{x})$
  $\mathbf{P}' = \{\mathbf{x}'_1, \ldots, \mathbf{x}'_\mu\} \leftarrow$ Select μ best individuals from $\mathbf{P}$
  $\mathbf{O} = \{\mathbf{y}_1, \ldots, \mathbf{y}_\lambda\} \leftarrow$ Generate λ offspring from $\mathbf{P}'$
  Evaluate fitness of individuals in $\mathbf{O}$
  $\mathbf{P} \leftarrow$ Combine $\mathbf{P}'$ and $\mathbf{O}$ and select μ best individuals
until stopping criterion met

4.4 Task 3.3 - MinMax agent

For the minmax problem, I developed a version of Minmax with Alpha-Beta Pruning, as it was more computionally efficient and lighter. Here the full implementation, that will be explained in each part. This code has been inspired by the pseudocode provided on Wikipedia (`https://en.wikipedia.org/wiki/Minimax#Pseudocode`). The explanation of the algorithm is a mixture of my personal notes from the course and the above mentioned Wikipedia article.

```python
import math

def minmax(state: Nim, maximizing_player: bool, alpha=-1, beta=1):
    if not state:
        return -1 if maximizing_player else 1

    data = cook_status(state)
    possible_next_states = []

    for ply in data['possible_moves']:
        tmp_state = deepcopy(state)
        tmp_state.nimming(ply)
        possible_next_states.append(tmp_state)

    if maximizing_player:
        best_val = -math.inf

        for next_state in possible_next_states:
            val = minmax(next_state, not maximizing_player, alpha,
            ↪  beta)
            best_val = max(best_val, val)
            alpha = max(alpha, best_val)

            if beta <= alpha:
                break
        return best_val
    else:
        best_val = math.inf
        next_state = deepcopy(state)
        ply = optimal_strategy(next_state)
        next_state.nimming(ply)

        val = minmax(next_state, not maximizing_player, alpha,
        ↪  beta)
        best_val = min(best_val, val)
        beta = min(beta, best_val)

        return best_val

```

```python
def minmax_strategy(state: Nim) -> Nimply:
    data = cook_status(state)

    for ply in data['possible_moves']:
        tmp_state = deepcopy(state)
        tmp_state.nimming(ply)

        score = minmax(tmp_state, maximizing_player = True)
        if score > 0:
            break
    return ply
```

The Minimax algorithm is a recursive method for choosing the next move in a two-player game. Each game position is assigned a value based on its favorability to a player, computed by a position evaluation function.

The player chooses the move that maximizes the minimum value of the opponent's possible moves. The value of each position is the following:

- A win for player A is assigned a value of $+\mathbf{1}$
- A win for player B is assigned a value of **-1**

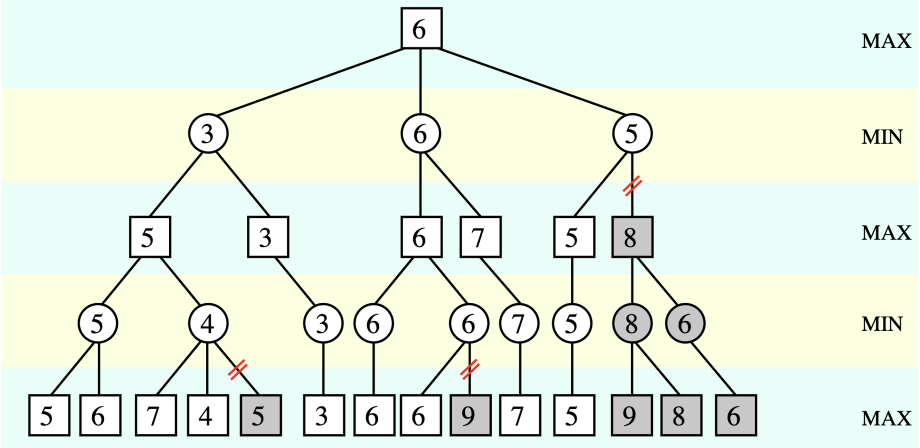
Player A is called the **maximizing player**, and player B is called the **minimizing player**, hence the name minimax algorithm. The goal is to assign a positive or negative infinity value to a position that represents a win or loss. For complex games like chess or go, this is not always feasible, and positions are given finite values as an estimate of the chances of winning. This is because it is not computationally feasible to look ahead to the end of the game, except **towards the end**.

The Minimax algorithm can be described as exploring the different game moves (nodes) in a game tree. The average number of moves (children) available at each node is called the effective branching factor. The number of nodes to be explored generally increases quickly as the number of moves (plies) increases, leading to a large number of nodes to be explored in games such as chess.

To improve the performance of the Minimax algorithm, without affecting its results, the following techniques can be used:

- **Alpha-Beta Pruning:** This technique helps to reduce the number of nodes explored in the game tree, making the algorithm more efficient.
- **Heuristic Pruning:** This technique uses different heuristics to further reduce the number of nodes explored, but not all heuristics give the same results as the unpruned search.

In this case, I have adopted alpha–beta pruning. The **benefit** of alpha–beta pruning lies in the fact that branches of the search tree can be **eliminated**. This way, the search time can be limited to the 'more promising' subtree, and a deeper search can be performed in the same time. This optimization reduces the effective depth to slightly more than half that of simple minimax if the nodes are evaluated in an optimal or near optimal order (best choice for side on move ordered first at each node).



The figure was taken from `https://upload.wikimedia.org/wikipedia/commons/thumb/9/91/AB_pruning.svg/2560px-AB_pruning.svg.png`, and it is very explanatory to understand this improvement. Suppose that moves are evaluated from left to right. The subtrees that are grayed out do not have to be explored because they have already been determined to be either equivalent to or worse than another subtree, and therefore will not affect the final result.

For each move, the minmax algorithm is called. The algorithm will return either 1 or -1, in case the agent wins or not. If the agent wins, then the search is stopped, otherwise other possible moves will be explored.

```python
def minmax_strategy(state: Nim) -> Nimply:
    data = cook_status(state)

    for ply in data['possible_moves']:
        tmp_state = deepcopy(state)
        tmp_state.nimming(ply)

        score = minmax(tmp_state, maximizing_player = True)
        if score > 0:
            break
    return ply
```

The function starts by checking if the current state is a terminal state, meaning there are no more moves left to make. If this is the case, the function returns -1 if it's the maximizing player's turn, or 1 if it's the minimizing player's turn.

```
def minmax(state: Nim, maximizing_player: bool, alpha=-1, beta=1):
    if not state:
        return 1 if maximizing_player else -1

    (...)
```

If the current state is not a terminal state, the function generates all possible next states by calling the `cook_status` function and nimming method of the Nim class to generate the next state for each possible move. The nimming method changes the state of the game to represent the move made.

```
def minmax(state: Nim, maximizing_player: bool, alpha=-1, beta=1):
    (...)
    data = cook_status(state)
        possible_next_states = []

        for ply in data['possible_moves']:
            tmp_state = deepcopy(state)
            tmp_state.nimming(ply)
            possible_next_states.append(tmp_state)
    (...)
```

The function then uses a loop to iterate through all possible next states and calculates the score for each state by calling the minmax function recursively with the new state and the opposite player. The `alpha` and `beta` parameters are used to prune the search tree, improving the efficiency of the algorithm.

The alpha and beta parameters are used to keep track of the best move found so far by the maximizing player (`alpha`) and the best move found so far by the minimizing player (`beta`).

```
def minmax(state: Nim, maximizing_player: bool, alpha=-1, beta=1):
    (...)
    if maximizing_player:
        best_val = -math.inf

        for next_state in possible_next_states:
            val = minmax(next_state, not maximizing_player, alpha,
            ↪  beta)
            best_val = max(best_val, val)
            alpha = max(alpha, best_val)

            if beta <= alpha:
                break
```

```
        return best_val
    else:
        best_val = math.inf
        next_state = deepcopy(state)
        ply = optimal_strategy(next_state)
        next_state.nimming(ply)

        val = minmax(next_state, not maximizing_player, alpha,
        ↪  beta)
        best_val = min(best_val, val)
        beta = min(beta, best_val)

        return best_val
```

If, at any point in the search, `beta` (the best move found so far by the minimizing player) is less than or equal to `alpha` (the best move found so far by the maximizing player), it means that the maximizing player has already found a move that guarantees a win, regardless of what the minimizing player does. In this case, the search can be stopped in the current branch, as it will not affect the final result.

The break statement in the line `if beta <= alpha:  break` stops the search in the current branch and returns to the previous level of the search tree, effectively pruning the tree and improving the efficiency of the algorithm.

By evaluating against the **optimal strategy**, I got a strange behavior:

- if nim size is 4, I never beat the opponent
- If nim size is 5, I always beat the opponent

A reviewer (`s295103`) kindly cleared it out for me. They suggested that this behavior happened because of the **Horizon effect**.

The Minimax algorithm can only search a limited portion of the game tree, typically a few moves deep, due to feasibility constraints. This means that my algorithm, when only searching four moves ahead, may make a mistake, but the error will not be visible because the algorithm does not search far enough to detect it. The search depth has to be limited, but this can result in an incorrect outcome if a crucial change exists **just beyond** the search horizon.

4.5 Task 3.4 - Q-Learning agent

For this task I used a Q-Learning approach. I have first seen this approach from a colleague, which can be found here: `https://github.com/bred91/Computational_Intelligence_2022-2023/tree/main/lab3`.

I quickly found out that he got this beautiful implementation from the repo `https://github.com/abelmariam/nimPy`, which I took as example too!

The explanation of the algorithm is a mixture of my personal notes from the course and the Wikipedia article `https://en.wikipedia.org/wiki/Q-learning`

```python
import numpy as np

class QL_Agent():
    q = {}
    previous_state = previous_action = None
    WIN_REWARD, LOSS_REWARD = 1, -1

    def __init__(self, state, k = None, epsilon = 1, learning_rate = 1, discount_factor = 1) -> None:
        # q is a function f: State x Action -> R
        # and is internally represented as a Map.

        # alpha is the learning rate and determines
        # to what extent the newly acquired information
        # will override the old information

        # gamma is the discount rate
        # and determines the importance of future rewards

        # epsilon serves as the exploration rate
        # and determines the probability that the agent,
        # in the learning process,
        # will randomly select an action

        self.epsilon = epsilon
        # epsilon -> the higher epsilon, the more random I act
        self.learning_rate = learning_rate
        # alpha -> the higher alpha, the more I replace "q"
        self.discount_factor = discount_factor
        # gamma -> the higher gamma, the more I favor long-term reward
        # as I get closer and closer to the deadline,
        # my preference for near-term reward should increase,
        # which means my gamma should decrease.

```

```
    def makeKey(self, state):
        possActions = list(self.getActions(state))
        someAction = possActions[0]

        # generating Q Table
        if (tuple(state), someAction) not in self.q:
            for i in possActions:
                self.q[(tuple(state), i)] = np.random.uniform(0.0,
                ↪  0.01)

    def is_terminal(self, state):
        '''returns True if the state is terminal'''
        return sum(state) == 0

    def getActions(self, state):
        '''returns a list of possible actions for a given state'''
        if self.is_terminal(state):
            return [None]

        possible_actions = []
        for row, num_objects in enumerate(state):
            for remaining in range(num_objects):
                possible_actions.append((row, num_objects -
                ↪  remaining))
        return possible_actions


    def policy(self, state):
        '''Policy
        This function takes a state and chooses the action for that
↪  state that will lead to the maximum reward'''
        possActions = list(self.getActions(state))

        if np.random.random() < self.epsilon:
            # Highest reward -> Low exploration rate
            q_values = [self.q[(tuple(state),i)] for i in
            ↪  possActions]
            return possActions[np.argmax(q_values)]
        else:
            # Random -> High exploration rate
            chosen_action_idx = np.random.randint(0,
            ↪  len(possActions))
            return possActions[chosen_action_idx]

    # Updates the Q-table as specified by the standard Q-learning
    ↪  algorithm
    def update_q(self, state):
        if self.is_terminal(state):
```

```
            self.q[(tuple(self.previous_state),
            ↪  self.previous_action)] += \
                self.learning_rate * (self.LOSS_REWARD -
                ↪  self.q[(tuple(self.previous_state),
                ↪  self.previous_action)])

            current_action = self.previous_state =
            ↪  self.previous_action = None
        else:
            self.makeKey(state)
            current_action = self.policy(state)

            if self.previous_action is not None:
                next_state = state.copy()
                next_state[current_action[0]] -= current_action[1]

                reward = self.WIN_REWARD if
                ↪  self.is_terminal(next_state) else 0
                maxQ = max(self.q[(tuple(state), a)] for a in
                ↪  self.getActions(state))
                self.q[(tuple(self.previous_state),
                ↪  self.previous_action)] += \
                    self.learning_rate * (reward +
                    ↪  self.discount_factor * maxQ - \
                        self.q[(tuple(self.previous_state),
                        ↪  self.previous_action)])

            self.previous_state, self.previous_action =
            ↪  tuple(state),current_action
        return current_action
```

```
def Q_play(opponent_strategy: Callable, nim_dim = 4):
    losses = 0
    wins = 0
    nGames = 10000

    for i in np.arange(nGames):
        currState = Nim(nim_dim)      # Reset game
        agent = QL_Agent(currState)   # Reset Agent

        while True:
            # Opponent plays
            opponent_play = opponent_strategy(currState)
            currState.nimming(opponent_play)

            action_p1 = agent.update_q(currState._rows)

            if(action_p1 is not None):
```

```
                currStateᐧnimming(Nimply(action_p1[0],
                ↪  action_p1[1]))

            if action_p1 is None:
                # Player can't do any actions -> LOSS
                losses += 1
                break
            elif currStateᐧgame_over():
                # Player reached gameover state -> WIN
                wins += 1
                break

    print(f"Games: {nGames} Wins: {wins} Losses: {losses} =>
    ↪  winrate: {wins/(wins+losses)}")
```

The core of the Q-Learning algorithm is based on the **Bellman equation**, which is used to update the estimated value of a given state-action pair. The equation calculates a weighted average of the current estimate and the new information, updating the estimate in the process. The goal of the algorithm is to find the maximum expected reward for a given state-action pair, which represents the optimal policy for the agent.

The **Bellman equation** provides a way to iteratively update this estimate by taking into account the current reward and the estimated rewards of all possible next states.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{current value}}\Big)}^{\text{temporal difference}}$$

- $Q(s_t, a_t)$ is the current estimate of the maximum future reward for taking action a_t in state s_t.
- α is the learning rate, a value between 0 and 1 that determines the rate at which the algorithm updates its estimate.
- r_t is the immediate reward received after taking action a_t in state s_t.
- γ is the discount factor, a value between 0 and 1 that determines the importance of future rewards relative to the current reward.
- $\max Q(s_{t+1}, a)$ is the maximum estimated future reward for all possible actions a in the next state s_{t+1}.

The function `getActions` first checks if the given state is terminal, in which case it returns a list with only one action: None. Otherwise, it generates all possible combinations of `(row, remaining number of objects in the row)` from the given state, and returns a list of these combinations as the possible actions.

```
def getActions(self, state):
    '''returns a list of possible actions for a given state'''
    if self.is_terminal(state):
        return [None]

    possible_actions = []
    for row, num_objects in enumerate(state):
        for remaining in range(num_objects):
            possible_actions.append((row, num_objects - remaining))
    return possible_actions
```

The function `makeKey` generates a Q table for this reinforcement learning. It starts by getting a list of possible actions from the state passed as an argument (by calling the `getActions` function). Then, it sets a default value for the Q-value of each action by initializing it randomly using a uniform distribution ranging from 0 to 0.01. The Q-value is stored in a dictionary `self.q` with the key being a tuple consisting of the state and the action, exactly like the definition of $Q(s_t, a_t)$.

```
def makeKey(self, state):
    possActions = list(self.getActions(state))
    someAction = possActions[0]

    # generating Q Table
    if (tuple(state), someAction) not in self.q:
        for i in possActions:
            self.q[(tuple(state), i)] = np.random.uniform(0.0,
            ↪  0.01)
```

The function `policy` function is used to choose the next action that the agent should take given the current state of the environment.

- Let `possActions` be a list of all possible actions in the current state.
- Generate a random number r between 0 and 1.
- If $r < \epsilon$, meaning the exploration rate is high:
  - Compute the Q-value for each action in `possActions`, and store the results in a list `q_values`.
  - Return the action with the highest Q-value, given by `possActions[np.argmax(q_values)]`.

- If $r \geq \epsilon$, meaning the exploration rate is low:
  - Choose a random index `chosen_action_idx` from 0 to the length of `possActions-1`.
  - Return the action associated with `chosen_action_idx`, given by `possActions[chosen_action_idx]`.

```
def policy(self, state):
    '''Policy
    This function takes a state and chooses the action for that
    ↪ state that will lead to the maximum reward'''
    possActions = list(self.getActions(state))

    if np.random.random() < self.epsilon:
        # Highest reward -> Low exploration rate
        q_values = [self.q[(tuple(state),i)] for i in possActions]
        return possActions[np.argmax(q_values)]
    else:
        # Random -> High exploration rate
        chosen_action_idx = np.random.randint(0, len(possActions))
        return possActions[chosen_action_idx]
```

In this way, the function balances exploration and exploitation: with high exploration rate, it will take random actions to potentially discover new, high-reward states; with low exploration rate, it will choose the action with the highest estimated reward based on the current Q-table.

Now, here is the main part of the algorithm, the `q_update` function.

```
# Updates the Q-table as specified by the standard Q-learning
↪ algorithm
def update_q(self, state):
    if self.is_terminal(state):
        self.q[(tuple(self.previous_state), self.previous_action)]
        ↪ += \
            self.learning_rate * (self.LOSS_REWARD -
            ↪ self.q[(tuple(self.previous_state),
            ↪ self.previous_action)])

        current_action = self.previous_state = self.previous_action
        ↪ = None
    else:
        self.makeKey(state)
        current_action = self.policy(state)

        if self.previous_action is not None:
            next_state = state.copy()
```

```
            next_state[current_action[0]] -= current_action[1]

            reward = self.WIN_REWARD if
            ↪  self.is_terminal(next_state) else 0
            maxQ = max(self.q[(tuple(state), a)] for a in
            ↪  self.getActions(state))
            self.q[(tuple(self.previous_state),
            ↪  self.previous_action)] += \
                self.learning_rate * (reward + self.discount_factor
                ↪  * maxQ - \
                    self.q[(tuple(self.previous_state),
                    ↪  self.previous_action)])

        self.previous_state, self.previous_action =
        ↪  tuple(state),current_action
    return current_action
```

If we are in the terminal state, we simply apply a special case of the Bellman formula:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot (r_{loss} - Q(s_t, a_t))$$

Otherwise, we apply the Bellman formula that was outlines earlier:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot (r_{win} + \gamma \cdot \max Q(s_{t+1}, a) - Q(s_t, a_t))$$

I tested the algorithm against random and expert players. In both cases the agent was capable to learn how to beat the opponent and win a good rate of matches (97% average win rate).

4.6 Received Reviews

Peer Review

`https://github.com/antoniodecinque99/computational_intelligence42/issues/6`

Task 3.1 Fixed Rules

For this task I want to congratulate with you because did a really great work. Indeed you tried a lot of different ideas and you evaluated all. I think that I don't have other idea to suggest you.

Task 3.2 Evolvable Rules

You find something interesting and I didn't think a something like what you code. Moreover because I never thought to a genetic algorithm with only one gene. Maybe you can try to implement something with different genes where in each is present a probability associated to different fixed rules. In this way you can implement different crossover and mutations.

Luca Marcellino, `s292950`

I love the trial-and-error way

`https://github.com/antoniodecinque99/computational_intelligence42/issues/7`

Task 1

I love the trial-and-error way in which you developed your fixed-rules solution. I am curious how the final version of `count_and_decide` would perform against the optimal strategy.

Task 2

Your solution reached immediately a high fitness value and then stopped growing: the typical early convergence problem. This can be easily solved by increasing the population size and using a (mu, lambda) EA with lambda $>$ 5mu*. If you want more exploration than that, you could also implement some quick diversity promotion strategy like extinction.

Task 3

Great implementation, nothing to say. The strange phenomenon relative to the number of heaps might be due to the horizon effect: try to implement a Monte Carlo tree search when the minmax algorithm reaches interesting nodes.

Task 4

Given how the evaluate function works, the first player to move always use the strategy passed as the first argument. Thus, if your

RL agent always does the best move, the `optimal_strategy` agent can't do anything but choose a random action and will perform similarly to the `pure_random` agent. Congrats, your RL agent learned the optimal strategy! ”

s295103

“ **Peer Review**

`https://github.com/antoniodecinque99/computational_intelligence42/issues/8`

- Your rules are interesting, you could also train your rules changing the k value, you are taking in account only the case when k is None
- My rules are similar to yours, maybe you could also compare it with other strategies like gabriele or some other basic rules
- I think you could improve task 2 weighting different rules, It will evolve to get the best probabilities of each rules. For me I see both picj function as the same IF ELSE rules, maybe adding some randomness rules, some Other Simple rules and weighting them you will get better results for your fitness. Your evolved rules are not evolving very well compared to your results in the task 1
- Your minmax is quite efficient and the Q-learning is very impressive, I would have a deeper look at your code to understand what I was not able to do
- Well coded and well commented code ”

tonatiu92

4.7 Given Reviews

My reviews were formalized at the time of the report, as I already had notes for them.

> **Thank you**
>
> `https://github.com/bred91/Computational_Intelligence_2022-2023/issues/9`
>
> **Task 3.1**
>
> You implemented the first agent by using a simple set of rules that seem to work reasonably well. The agent is able to make decisions based on the number of objects in each row, as well as whether or not it is its turn. Simple, but effective! The game outputs the current game state after each turn, which made it easy for me to understand what is happening.
>
> **Task 3.2**
>
> You successfully integrated a genetic algorithm (GA) for task 3.1, using a genome that represented the 3 parameter values for triggering actions, fitness as the percentage of matches won against the random agent, and implementing mutation and crossover operators. Your approach resulted in significant improvement in the number of matches won. Good job!
>
> **Task 3.3**
>
> Despite the fact that the algorithm is not able to beat an optimal opponent, I think your code is still valid. My minmax also does not work well against the expert system, and I haven't seen an algorithm that works well with reasonable depth.
>
> **Task 3.4**
>
> I took inspiration from your code and the original repo. I want to thank you for it, because the algorithm is written in an elegant way and it made me understand Reinforcement Learning even better.
>
> *Me*

> **I know well your creativity and problem-solving skills!**
>
> `https://github.com/feurode46/computational-intelligence-2022/issues/7`
>
> **Task 3.1**
>
> This is a well thought-out implementation! It's great to see that you've come up with a strategy that is able to beat an opponent with random moves 95% of the time. Your approach of using a decision tree to guide your moves is great, and it's interesting to see how it adapts to different situations, such as the introduction of k.

We had the same idea regarding the terminal condition of the game. Your strategy is simple yet effective, and I know well your creativity and problem-solving skills!

Task 3.2

The genome and crossover design are straightforward and effective, while the fitness evaluation method of 100 games against a random player provides reliable results. You could try to implement it against an optimal player.

The results of the training show that the algorithm came to the same conclusion as the creator's initial strategy, with a 95% win rate against a random player. It seems strange that you never ever win against the optimal. I think you could try to change the possible type of moves, based on the probability. Additionally, experimenting with different population sizes, offspring sizes, and mutation rates could potentially lead to better results, but I'm sure you did it already.

Task 3.3

The implementation of Minimax with memoization and alpha-beta pruning shows effort to optimize the algorithm. I will certainly take your implementation of memoization, and use it with a library such as pickle. The issue with the implementation performing worse than the vanilla one is intriguing and I suggest further investigation.

Overall, the average win rate of 76% against a random player is a good accomplishment, but it would be interesting to see how the algorithm performs with a different heuristic / depth. Minmax doesn't seem to work well with this type of game.

Task 3.4

The results of training the Reinforcement Learning agent in this task show promising results, with a good winrate against the optimal strategy. However, the performance against a random strategy is not as good, considering the weakness of playing randomly in this game.

One potential reason for the inconsistent learning and variability in performance could be the random initialization of the G-table and the lack of a good heuristic to guide the learning. Additionally, fine-tuning more the exploration-exploitation balance and adjusting the learning rate differently may improve the stability and consistency of the agent's learning.

Another approach to consider could be combining the Reinforcement Learning algorithm with other methods, such as Monte Carlo Tree Search.

You did a very good job! ”

Me

5 The Power of Wandering: a Genetic Algorithm for Quarto

5.1 Introduction

The final stage of the course is to implement an agent for the board game **Quarto**.

To tackle this project I join a group of students, to exchange ideas and share results of trying different approaches.

The group consisted of six people

- Me
- Federico Boscolo (s294908)
- Leonardo Rolandi (s301216)
- Flavio Patti (s301104)
- Davide Aiello (s303296)
- Giuseppe Pellegrino (s303999)

We had some conference calls to try different approaches and share the results. Each part of the group was assigned a specific type of algorithm to implement (Monte Carlo search, reinforcement learning, genetic algorithms...).

With Flavio, I had to implement a genetic algorithm. The code of was public to all other members of the group (Flavio's private GitHub repo `https://github.com/FlavioPatti`). At the very beginning, we started to work together on it, but eventually I went forward on my own, and ended up rewriting most of the algorithm from scratch many times.

I like to call this algorithm "The power of Wandering". I always thought this concept, since I was little. Wandering is a magical and powerful act. It can spark creativity, bring joy, and lead to unexpected discoveries. It allows you to break free from routine, explore new perspectives and unleash the imagination. The beauty of wandering is that it can be done anywhere, at any time and it's open to interpretation.

Most colleagues around me were developing a MinMax algorithm, as it seemed to work better in this game. The genetic algorithm, instead, was not as strong,

even in beating the random player. I took it as a personal challenge, and tried to develop this idea as much as I could with the time given. The algorithm is surely not the strongest, but I like the idea behind it!

5.2 Full Algorithm

This is the full implementation, which i am going to explain part by part.

```python
from collections import namedtuple
import random
from copy import deepcopy
import quarto
import math
import numpy as np
import itertools

Individual = namedtuple("Individual", ["genome", "fitness"])
# Genome -> array of 8 elements (4 figures + 4 positions)

BOARD_SIZE = 4
GENOME_SIZE = BOARD_SIZE * 2
POPULATION_SIZE = 512
NUM_GENERATIONS = 40
OFFSPRING_SIZE = 100
TOURNAMENT_SIZE = 5
CROSSOVER_THRESHOLD = 0.4
MUTATION_THRESHOLD = 0.1

SELF_CHOOSE = 0
SELF_PLACE = 1


class GeneticAlgorithm():
    def __init__(self, current_game: quarto.Quarto):
        self.current_game = current_game
        self.iterations = 4

    def tupleToIndex(self, x, y):
        return 4 * y + x

    def indexToTuple(self, index):
        x = index % BOARD_SIZE
        y = math.floor(index / BOARD_SIZE)
        return (x,y)

    def try_place(self, x: int, y: int) -> bool:
```

```
        '''
        Verify if a piece is placeable but don't actually place it
        '''
        return not (y < 0 or x < 0 or x > 3 or y > 3 or
        ↪  self.current_game._board[y, x] >= 0)

    def place(self, x: int, y: int, piece_index: int) -> bool:
        '''
        Place piece in coordinates (x, y). Returns true on success
        '''
        if self.try_place(x, y):
            self.current_game._board[y, x] = piece_index
            self.current_game._binary_board[y, x][:] =
            ↪  self.current_game._Quarto__pieces[piece_index].binary
            return True
        return False

    def unplace(self, x: int, y: int) -> bool:
        '''
        Take away piece in coordinates (x, y). Returns true on
↪   success
        '''
        self.current_game._board[y, x] = -1
        self.current_game._binary_board[y, x][:] = np.nan
        return True

    def available_positions(self, genome: list = None):
        '''
        Lists available positions on the board, considering also
↪   the positions potentially taken by the genome
        '''
        listAvailablePositions = []

        for x in range(self.current_game.BOARD_SIDE):
            for y in range(self.current_game.BOARD_SIDE):
                if self.try_place(x, y):
                    coord = self.tupleToIndex(x, y)
                    listAvailablePositions.append(coord)
        #print(listAvailablePositions)

        if genome is not None:
            for i in range(GENOME_SIZE//2, GENOME_SIZE//2 +
            ↪  self.iterations):
                if (genome[i] in listAvailablePositions) and
                ↪  len(listAvailablePositions) > 0:
                    listAvailablePositions.remove(genome[i])

        return listAvailablePositions

```

```
82
83      def available_pieces(self, genome: list = None):
84          '''
85          Lists available pieces, considering also the pieces
    ↪   potentially taken by the genome
86          '''
87          listAvailablePieces = list(range(16))
88
89          if genome is not None:
90              for i in range(0, self.iterations):
91                  if (genome[i] in listAvailablePieces):
92                      listAvailablePieces.remove(genome[i])
93
94          for x in range(self.current_game.BOARD_SIDE):
95              for y in range(self.current_game.BOARD_SIDE):
96                  current_piece = self.current_game._board[y,x]
97                  if current_piece != -1 and current_piece in
                    ↪  listAvailablePieces:
98                      listAvailablePieces.remove(current_piece)
99
100         return listAvailablePieces
101
102
103     def tournament(self, population,
        ↪  tournament_size=TOURNAMENT_SIZE):
104         '''
105         Parent selection - TOURNAMENT version
106         '''
107         return max(random.choices(population, k=tournament_size),
            ↪  key=lambda i: i.fitness)
108
109     def roulette_wheel_selection(self, population):
110         '''
111         Parent selection - ROULETTE WHEEL version
112         '''
113         fitness_sum = sum(individual.fitness for individual in
            ↪  population)
114         if fitness_sum == 0:
115             return self.tournament(population, TOURNAMENT_SIZE)
116
117         normalized_fitness = [individual.fitness/fitness_sum for
            ↪  individual in population]
118
119         cumulative_probabilities = [sum(normalized_fitness[:i+1])
            ↪  for i in range(len(normalized_fitness))]
120         random_num = random.random()
121         for i, prob in enumerate(cumulative_probabilities):
122             if random_num <= prob:
123                 return population[i]
```

```
124
125     def cross_over_1(self, genome_1: list, genome_2: list):
126         '''
127         Crossover between genomes. The new genome will have some
    ↪   genes from first parent, and other genes from second one.
128         '''
129         new_genome = [-1]*GENOME_SIZE
130
131         for i in range(0, self.iterations):
132             if (random.randint(0,1) > CROSSOVER_THRESHOLD):
133                 new_genome[i] = genome_1[i]
134             else:
135                 new_genome[i] = genome_2[i]
136
137         for i in range(GENOME_SIZE//2, GENOME_SIZE//2 +
        ↪   self.iterations):
138             if (random.randint(0,1) > CROSSOVER_THRESHOLD):
139                 new_genome[i] = genome_1[i]
140             else:
141                 new_genome[i] = genome_2[i]
142
143         return new_genome
144
145     def cross_over_2(self, genome_1, genome_2):
146         piece_changes, pos_exchanges = random.randint(1, 3),
        ↪   random.randint(1, 4)
147         new_genome = deepcopy(genome_1)
148
149         for i in range(1, piece_changes):
150             new_genome[i] = genome_2[i]
151         for i in range(piece_changes, 4 - piece_changes):
152             new_genome[i] = genome_2[i]
153
154         for i in range(1, pos_exchanges):
155             new_genome[4 + i] = genome_2[4 + i]
156         for i in range(pos_exchanges, 4 - pos_exchanges):
157             new_genome[4 + i] = genome_2[4 + i]
158
159         return new_genome
160
161
162     def mutation_1(self, genome):
163         """
164         In the genome, "pieces" genes are swapped among each other,
165         and "position" genes are swapped among each other too.
166         """
167         new_genome = deepcopy(genome)
168
```

```
169         if (random.randint(0,1) > MUTATION_THRESHOLD): # mutate
            ↪  pieces
170             i_1, i_2 = random.sample(range(self.iterations), 2)
171             new_genome[i_1], new_genome[i_2] = new_genome[i_2],
                ↪  new_genome[i_1]
172 
173         if (random.randint(0,1) > MUTATION_THRESHOLD): # mutate
            ↪  positions
174             i_1, i_2 = random.sample(range(GENOME_SIZE//2,
                ↪  GENOME_SIZE//2 + self.iterations), 2)
175             new_genome[i_1], new_genome[i_2] = new_genome[i_2],
                ↪  new_genome[i_1]
176 
177         return new_genome
178 
179     def mutation_2(self, genome: list):
180         new_genome = deepcopy(genome)
181 
182         for pieceIndex in range(1, self.iterations): # mutate
            ↪  pieces
183             if (random.randint(0,1) > MUTATION_THRESHOLD):
184                 available_pieces =
                    ↪  self.available_pieces(new_genome)
185                 if (len(available_pieces) > 0):
186                     new_genome[pieceIndex] =
                        ↪  (random.choice(available_pieces))
187 
188         for posIndex in range(GENOME_SIZE//2, GENOME_SIZE//2 +
            ↪  self.iterations): # mutate positions
189             if (random.randint(0,1) > MUTATION_THRESHOLD):
190                 available_positions =
                    ↪  self.available_positions(new_genome)
191                 if (len(available_positions) > 0):
192                     new_genome[posIndex] =
                        ↪  (random.choice(available_positions))
193         return new_genome
194 
195     def isWinning(self, current_piece: int, pos_index: int):
196         pos_tuple = self.indexToTuple(pos_index)
197         is_winning = False
198 
199         if (self.place(pos_tuple[0], pos_tuple[1], current_piece)):
200             is_winning = True if (self.current_game.check_winner()
                ↪  >= 0) else False
201             self.unplace(pos_tuple[0], pos_tuple[1])
202 
203         return is_winning
204 
205     def isWinnable(self, current_piece: int):
```

```
206         available_positions = self.available_positions()
207         for pos in available_positions:
208             if self.isWinning(current_piece, pos):
209                 return True
210         return False
211 
212     def computeFitness (self,genome: list, strategyType: int):
213         tot_reward = 0
214 
215         for i in range(self.iterations):
216             piece = genome[i]
217             pos_index = genome[GENOME_SIZE//2 + i]
218 
219 
220             winning_move = self.isWinning(piece, pos_index)
221             is_winnable = self.isWinnable(piece)
222 
223             # My turn
224             if i == 0:
225                 if strategyType == SELF_PLACE:
226                     # Opponent chose a piece, and I need to place
                        ↪  it now
227                     if (winning_move):
228                         # am I predicting winning move?
229                         tot_reward = 1
230                         break
231                     if (is_winnable):
232                         # is the opponent dumb enough to give me a
                            ↪  potentially winning piece?
233                         tot_reward += 0.5
234                         break
235                 elif strategyType == SELF_CHOOSE:
236                     # I need to choose a piece for the opponent
237                     # therefore, the first genome does not make
                        ↪  sense
238                     continue
239 
240             # Opponent turn
241             elif i == 1:
242                 if strategyType == SELF_CHOOSE:
243                     # I choose a piece for the opponent
244                     if (winning_move):
245                         # can the opponent make a winning move with
                            ↪  this?
246                         tot_reward = -1
247                         break
248                     if (is_winnable):
249                         # am I being dumb enough to give the
                            ↪  opponent a potentially winning piece?
```

```
250                         tot_reward -= 0.5
251                         break
252                     else:
253                         # the opponent cannot do anything with this
                            ↪ piece!
254                         tot_reward += 0.25
255                 if strategyType == SELF_PLACE:
256                     # For now this move does not make sense,
257                     # as I will be re-executing the algorithm later
                        ↪ on!
258                     continue
259
260             # Trying to look ahead
261             elif i == 2 and (winning_move or is_winnable):
262                 tot_reward += 0.25
263             elif i == 3 and (winning_move or is_winnable):
264                 tot_reward -= 0.25
265
266
267         return tot_reward
268
269     def initPopulation(self, strategyType: int, size: int =
    ↪ POPULATION_SIZE):
270         '''
271         Generate initial population. StrategyType is needed to
↪   evaluate fitness.
272         '''
273         population = []
274
275         remaining = len(self.available_pieces())
276         self.iterations = min(remaining, self.iterations)
277
278         for p in range(size):
279             genome = [-1]*GENOME_SIZE
280
281             if self.current_game._Quarto__selected_piece_index ==
                ↪ -1:
282                 genome[0] = random.randint(0,15)
283             else:
284                 genome[0] =
                    ↪ self.current_game._Quarto__selected_piece_index
285
286             for i in range(1,self.iterations):
287                 listAvailablePieces = self.available_pieces(genome)
288                 genome[i] = random.choice(listAvailablePieces)
289
290             for i in range(GENOME_SIZE//2, GENOME_SIZE//2 +
                ↪ self.iterations):
```

```
                listAvailablePositions =
                ↪  self.available_positions(genome)
                genome[i] = random.choice(listAvailablePositions)

            population.append(Individual(genome,
            ↪  self.computeFitness(genome, strategyType)))

        return population


    def my_move(self, strategyType: int):
        population = self.initPopulation(strategyType)
        if (self.iterations == 1):
            return (population[0][0][1], population[0][0][4])

        for g in range(NUM_GENERATIONS):

            offspring = list()
            for i in range(OFFSPRING_SIZE):
                if random.random() > MUTATION_THRESHOLD:
                    # mutation
                    p = self.roulette_wheel_selection(population)
                    o = self.mutation_2(p.genome)

                    f = self.computeFitness(o, strategyType)

                    if (f > p.fitness):
                        offspring.append(Individual(o, f))
                    else:
                        offspring.append(p)

                if random.random() > CROSSOVER_THRESHOLD:
                    # crossover
                    p1 = self.roulette_wheel_selection(population)
                    p2 = self.roulette_wheel_selection(population)

                    o = self.cross_over_1(p1.genome, p2.genome)
                    f = self.computeFitness(o, strategyType)

                    offspring.append(Individual(o,f))

            # append offspring to population
            population += offspring

            # code to have unique elements in population
            # given that we use "namedtuple" types we need to do it
            ↪  manually
            unique_population = []
            unique_genomes = []
```

```
337             for individual in population:
338                 if individual.genome not in unique_genomes:
339                     unique_genomes.append(individual.genome)
340                     unique_population.append(individual)
341 
342             # take most promising genomes only, according to
                ↪  fitness
343             population = sorted(unique_population, key=lambda i:
                ↪  i[1], reverse = True)[:POPULATION_SIZE]
344             #print(*population, sep="\n")
345 
346             best_fitness = population[0].fitness
347             # sometimes the algoithm gets stuck in finding only
                ↪  losing solutions.
348             # by reinitializing the population, we introduce
                ↪  randomness again
349             # almost always, the algorithm gets back up
350             if best_fitness < 0:
351                 print("yeuch")
352                 population += self.initPopulation(strategyType,
                    ↪  POPULATION_SIZE * 2)
353             population = sorted(unique_population, key=lambda i:
                ↪  i[1], reverse = True)[:POPULATION_SIZE]
354 
355         best_genome = population[0].genome
356 
357         #print(population[0])
358         piece_to_give = best_genome[1]
359         position_to_play = self.indexToTuple(best_genome[4])
360         #print((piece_to_give, position_to_play))
361         return (piece_to_give, position_to_play)
362 
363 class GeneticPlayer(quarto.Player):
364     """GA player"""
365 
366     def __init__(self, current_game: quarto.Quarto):
367         #super().__init__(quarto)
368         self.geneticAlgorithm = GeneticAlgorithm(current_game)
369 
370         self.piece_to_give = None
371         self.pos_chosen = None
372 
373     def choose_piece(self):
374         (self.piece_to_give, self.pos_chosen) =
            ↪  self.geneticAlgorithm.my_move(SELF_CHOOSE)
375         #print("GA chooses piece - ", self.piece_to_give)
376         return self.piece_to_give
377 
378     def place_piece(self):
```

```
379             (self.piece_to_give, self.pos_chosen) =
                ↪  self.geneticAlgorithm.my_move(SELF_PLACE)
380             #print("GA chooses pos - ", self.pos_chosen)
381             return self.pos_chosen
382
383
```

5.3 Genome representation and utility functions

As individual representation the algorithm uses the usual namedtuple, containing genome and fitness. The genome is a very peculiar representation. It is a list of 8 elements, where:

- the first 4 elements consist in possible pieces to be placed
- the last 4 elements consist in the respective positions, where such pieces are to be placed

Given the `GENOME_SIZE = 8`, for `i < 4`, we are going to have the piece `genome[i]`, with its corresponding position `genome[i + GENOME_SIZE//2]`.

- `genome[0]` - This is the piece that I have been given, or that I have to choose myself. Its position will be `genome[4]`
- `genome[1]` - This is the piece that I will choose, and give it to my opponet. Its position will be `genome[5]`
- `genome[2]` - This is the piece that I predict will be given to me, so that I will have to place it. Its position will be `genome[6]`
- `genome[3]` - This is the piece that I will choose if my earlier prediction is somehow right. Its position will be `genome[7]`

We are basically looking ahead of 4 possible moves. The algorithm takes into account also the terminal state, by using the variable `self.iterations`. If there are less than 4 moves left, then we will consider only `self.iterations` moves.

For making the algorithm faster and more simple, the positions are stored as integers. Therefore, there are some support functions to convert the position back into a tuple of coordinates.

```
    def tupleToIndex(self, x, y):
        return 4 * y + x

    def indexToTuple(self, index):
        x = index % BOARD_SIZE
        y = math.floor(index / BOARD_SIZE)
        return (x,y)
```

Other utils functions to check for piece / position availability are overwritten from the Quarto module.

```
    def try_place(self, x: int, y: int) -> bool:
        '''
        Verify if a piece is placeable but don't actually place it
        '''
        return not (y < 0 or x < 0 or x > 3 or y > 3 or
        ↪  self.current_game._board[y, x] >= 0)

    def place(self, x: int, y: int, piece_index: int) -> bool:
        '''
        Place piece in coordinates (x, y). Returns true on success
        '''
        if self.try_place(x, y):
            self.current_game._board[y, x] = piece_index
            self.current_game._binary_board[y, x][:] =
            ↪  self.current_game._Quarto__pieces[piece_index].binary
            return True
        return False

    def unplace(self, x: int, y: int) -> bool:
        '''
        Take away piece in coordinates (x, y). Returns true on
    ↪  success
        '''
        self.current_game._board[y, x] = -1
        self.current_game._binary_board[y, x][:] = np.nan
        return True
```

Then, we have functions that return all the available pieces / positions on the table. They are very useful in genetic operators. Indeed, they take into account also the pieces / positions of the genome itself. I wanted to avoid a "sick" genome in which we have duplicate pieces or positions!

```
    def available_positions(self, genome: list = None):
        '''
        Lists available positions on the board, considering also
    ↪  the positions potentially taken by the genome
        '''
```

```
        listAvailablePositions = []

        for x in range(self.current_game.BOARD_SIDE):
            for y in range(self.current_game.BOARD_SIDE):
                if self.try_place(x, y):
                    coord = self.tupleToIndex(x, y)
                    listAvailablePositions.append(coord)
        #print(listAvailablePositions)

        if genome is not None:
            for i in range(GENOME_SIZE//2, GENOME_SIZE//2 +
            ↪  self.iterations):
                if (genome[i] in listAvailablePositions) and
                ↪  len(listAvailablePositions) > 0:
                    listAvailablePositions.remove(genome[i])

        return listAvailablePositions


    def available_pieces(self, genome: list = None):
        '''
        Lists available pieces, considering also the pieces
↪   potentially taken by the genome
        '''
        listAvailablePieces = list(range(16))

        if genome is not None:
            for i in range(0, self.iterations):
                if (genome[i] in listAvailablePieces):
                    listAvailablePieces.remove(genome[i])

        for x in range(self.current_game.BOARD_SIDE):
            for y in range(self.current_game.BOARD_SIDE):
                current_piece = self.current_game._board[y,x]
                if current_piece != -1 and current_piece in
                ↪  listAvailablePieces:
                    listAvailablePieces.remove(current_piece)

        return listAvailablePieces
```

5.4 Genomic operators

Genomic operators in genetic algorithms are functions that modify or manipulate the genetic material of an individual in a population. They include:

- **Selection**: Choosing individuals from the population to participate in the

next generation based on their fitness.

- **Crossover**: Combining genes from two parents to create offspring with a mix of both parents' traits.
- **Mutation**: Randomly changing the value of one or more genes to introduce new variations.

These operators work together to evolve the population towards a solution to the problem being optimized. The exact implementation of the genomic operators can have a significant impact on the performance of the genetic algorithm, so choosing the right combination of operators / parameters has been important for the optimization part. After testing several parameters, I found these to be a good compromise between computation time and efficiency.

```
POPULATION_SIZE = 512
NUM_GENERATIONS = 40
OFFSPRING_SIZE = 100
TOURNAMENT_SIZE = 5
CROSSOVER_THRESHOLD = 0.4
MUTATION_THRESHOLD = 0.1
```

For the selection operator, I tried to implement a combination of two types:

- **Tournament selection**: a small random sample of individuals from the population is selected and the best one is chosen for reproduction. This process is repeated until the desired number of individuals have been selected.
- **Roulette wheel selection**: individuals are assigned a portion of the roulette wheel proportional to their fitness. the wheel is then spun and the individual at the stopping point is selected. This allows individuals with higher fitness to have a higher chance of being selected for reproduction

```
    def tournament(self, population,
    ↪  tournament_size=TOURNAMENT_SIZE):
        '''
        Parent selection - TOURNAMENT version
        '''
        return max(random.choices(population, k=tournament_size),
        ↪  key=lambda i: i.fitness)

    def roulette_wheel_selection(self, population):
        '''
        Parent selection - ROULETTE WHEEL version
        '''
```

```
        fitness_sum = sum(individual.fitness for individual in
        ↪  population)
        if fitness_sum == 0:
            return self.tournament(population, TOURNAMENT_SIZE)

        normalized_fitness = [individual.fitness/fitness_sum for
        ↪  individual in population]

        cumulative_probabilities = [sum(normalized_fitness[:i+1])
        ↪  for i in range(len(normalized_fitness))]
        random_num = random.random()
        for i, prob in enumerate(cumulative_probabilities):
            if random_num <= prob:
                return population[i]
```

The privileged method is roulette wheel selection, as it allows better individuals to be more likely to be selected as parents. Given that the fitness is initialized to 0 for neutral moves, this method cannot be applied at the beginning. In such cases, I applied a tournament selection.

For the crossover, I also tried two approaches:

- The first approach to crossover will just create a new one, by taking some pieces from the first parent, and other pieces from the second.
- The second approach to crossover will, instead, exchange the pieces and positions of both genomes, but it will keep the association `piece -> position`.

I ended up using the first one, as it seemed to me that more randomness would benefit the algorithm search.

```
    def cross_over_1(self, genome_1: list, genome_2: list):
        '''
        Crossover between genomes. The new genome will have some
↪   genes from first parent, and other genes from second one.
        '''
        new_genome = [-1]*GENOME_SIZE

        for i in range(0, self.iterations):
            if (random.randint(0,1) > CROSSOVER_THRESHOLD):
                new_genome[i] = genome_1[i]
            else:
                new_genome[i] = genome_2[i]

        for i in range(GENOME_SIZE//2, GENOME_SIZE//2 +
        ↪  self.iterations):
```

```
            if (random.randint(0,1) > CROSSOVER_THRESHOLD):
                new_genome[i] = genome_1[i]
            else:
                new_genome[i] = genome_2[i]

        return new_genome

    def cross_over_2(self, genome_1, genome_2):
        piece_changes, pos_exchanges = random.randint(1, 3),
        ↪  random.randint(1, 4)
        new_genome = deepcopy(genome_1)

        for i in range(1, piece_changes):
            new_genome[i] = genome_2[i]
        for i in range(piece_changes, 4 - piece_changes):
            new_genome[i] = genome_2[i]

        for i in range(1, pos_exchanges):
            new_genome[4 + i] = genome_2[4 + i]
        for i in range(pos_exchanges, 4 - pos_exchanges):
            new_genome[4 + i] = genome_2[4 + i]

        return new_genome
```

For the mutation, I also used two approaches:

- In the first approach, "pieces" genes can be swapped among each other, and "position" genes can be swapped among each other too.
- In the second approach, I simply take one piece or one position in the genome and change it with a random one

```
    def mutation_1(self, genome):
        """
        In the genome, "pieces" genes are swapped among each other,
        and "position" genes are swapped among each other too.
        """
        new_genome = deepcopy(genome)

        if (random.randint(0,1) > MUTATION_THRESHOLD): # mutate
        ↪  pieces
            i_1, i_2 = random.sample(range(self.iterations), 2)
            new_genome[i_1], new_genome[i_2] = new_genome[i_2],
            ↪  new_genome[i_1]

        if (random.randint(0,1) > MUTATION_THRESHOLD): # mutate
        ↪  positions
```

```
            i_1, i_2 = random.sample(range(GENOME_SIZE//2,
            ↪  GENOME_SIZE//2 + self.iterations), 2)
            new_genome[i_1], new_genome[i_2] = new_genome[i_2],
            ↪  new_genome[i_1]

        return new_genome

    def mutation_2(self, genome: list):
        new_genome = deepcopy(genome)

        for pieceIndex in range(1, self.iterations): # mutate
        ↪  pieces
            if (random.randint(0,1) > MUTATION_THRESHOLD):
                available_pieces =
                ↪  self.available_pieces(new_genome)
                if (len(available_pieces) > 0):
                    new_genome[pieceIndex] =
                    ↪  (random.choice(available_pieces))

        for posIndex in range(GENOME_SIZE//2, GENOME_SIZE//2 +
        ↪  self.iterations): # mutate positions
            if (random.randint(0,1) > MUTATION_THRESHOLD):
                available_positions =
                ↪  self.available_positions(new_genome)
                if (len(available_positions) > 0):
                    new_genome[posIndex] =
                    ↪  (random.choice(available_positions))
        return new_genome
```

5.5 Fitness function

And here it comes the most important part of the algorithm: the fitness function. A fitness function is a key component of a genetic algorithm as it defines the quality of a particular solution or candidate. It acts as an evaluation tool to assess the performance of each candidate, or genome, in a population. The fitness function plays a critical role in guiding the selection process in a genetic algorithm. Higher the fitness score of a genome, the more likely it is to be selected for further processing. The ultimate goal of a genetic algorithm is to optimize the fitness function. Hence, defining an effective fitness function is of utmost importance in genetic algorithms, as it directly affects the results of the optimization process.

I tried countless types of combinations, and I ended up with a fitness function that performs quite well.

```
    def isWinning(self, current_piece: int, pos_index: int):
        pos_tuple = self.indexToTuple(pos_index)
        is_winning = False

        if (self.place(pos_tuple[0], pos_tuple[1], current_piece)):
            is_winning = True if (self.current_game.check_winner()
            ↪  >= 0) else False
            self.unplace(pos_tuple[0], pos_tuple[1])

        return is_winning

    def isWinnable(self, current_piece: int):
        available_positions = self.available_positions()
        for pos in available_positions:
            if self.isWinning(current_piece, pos):
                return True
        return False

    def computeFitness (self,genome: list, strategyType: int):
        tot_reward = 0

        for i in range(self.iterations):
            piece = genome[i]
            pos_index = genome[GENOME_SIZE//2 + i]

            is_winning_move = self.isWinning(piece, pos_index)
            is_winnable = self.isWinnable(piece)

            # My turn
            if i == 0:
                if strategyType == SELF_PLACE:
                    # Opponent chose a piece, and I need to place
                    ↪  it now
                    if (is_winning_move):
                        # am I predicting winning move?
                        tot_reward = 1
                        break
                    if (is_winnable):
                        # is the opponent dumb enough to give me a
                        ↪  potentially winning piece?
                        tot_reward += 0.5
                        break
                elif strategyType == SELF_CHOOSE:
                    # I need to choose a piece for the opponent
                    # therefore, the first genome does not make
                    ↪  sense
                    continue

            # Opponent turn
```

```
            elif i == 1:
                if strategyType == SELF_CHOOSE:
                    # I choose a piece for the opponent
                    if (is_winning_move):
                        # can the opponent make a winning move with this?
                        tot_reward = -1
                        break
                    if (is_winnable):
                        # am I being dumb enough to give the opponent a potentially winning piece?
                        tot_reward -= 0.5
                        break
                    else:
                        # the opponent cannot do anything with this piece!
                        tot_reward += 0.25
                if strategyType == SELF_PLACE:
                    # For now this move does not make sense,
                    # as I will be re-executing the algorithm later on!
                    continue

            # Trying to look ahead
            elif i == 2 and (is_winning_move or is_winnable):
                tot_reward += 0.25
            elif i == 3 and (is_winning_move or is_winnable):
                tot_reward -= 0.25


        return tot_reward
```

Let us remind the genome structure:

- `genome[0]` - This is the piece that I have been given, or that I have to choose myself. Its position will be `genome[4]`
- `genome[1]` - This is the piece that I will choose, and give it to my opponet. Its position will be `genome[5]`
- `genome[2]` - This is the piece that I predict will be given to me, so that I will have to place it. Its position will be `genome[6]`
- `genome[3]` - This is the piece that I will choose if my earlier prediction is somehow right. Its position will be `genome[7]`

The fitness function is computed according to 2 strategies:

- If the strategy is `SELF_CHOOSE`, it means that I have to choose a piece for my opponent. It means that I have already put a piece on the board, and therefore the `genome[0]` does not bear any meaning. This gene will be therefore "deactivated", and will not contribute to the fitness value.
- If the strategy is `SELF_PLACE`, it means that a piece has been chosen from the opponent, and now I need to place it. `genome[0]` will be the piece chosen by the opponent, which I have to place.

The code is self explanatory with the comments. The maximum reward would be 1, and the minimum reward -1. There are intermediate situations, highlighted with different fractional rewards.

5.6 Genetic Algorithm choosing function

The bulk of the algorithm is the classical structure of the genetic algorithm.

```
    def my_move(self, strategyType: int):
        population = self.initPopulation(strategyType)
        if (self.iterations == 1):
            return (population[0][0][1], population[0][0][4])

        for g in range(NUM_GENERATIONS):

            offspring = list()
            for i in range(OFFSPRING_SIZE):
                if random.random() > MUTATION_THRESHOLD:
                    # mutation
                    p = self.roulette_wheel_selection(population)
                    o = self.mutation_2(p.genome)

                    f = self.computeFitness(o, strategyType)

                    if (f > p.fitness):
                        offspring.append(Individual(o, f))
                    else:
                        offspring.append(p)

                if random.random() > CROSSOVER_THRESHOLD:
                    # crossover
                    p1 = self.roulette_wheel_selection(population)
                    p2 = self.roulette_wheel_selection(population)

                    o = self.cross_over_1(p1.genome, p2.genome)
                    f = self.computeFitness(o, strategyType)

```

```
                    offspring.append(Individual(o,f))

            # append offspring to population
            population += offspring

            # code to have unique elements in population
            # given that we use "namedtuple" types we need to do it manually
            unique_population = []
            unique_genomes = []
            for individual in population:
                if individual.genome not in unique_genomes:
                    unique_genomes.append(individual.genome)
                    unique_population.append(individual)

            # take most promising genomes only, according to fitness
            population = sorted(unique_population, key=lambda i: i[1], reverse = True)[:POPULATION_SIZE]
            #print(*population, sep="\n")

            best_fitness = population[0].fitness
            # sometimes the algoithm gets stuck in finding only losing solutions.
            # by reinitializing the population, we introduce randomness again
            # almost always, the algorithm gets back up
            if best_fitness < 0:
                print("yeuch")
                population += self.initPopulation(strategyType, POPULATION_SIZE * 2)
            population = sorted(unique_population, key=lambda i: i[1], reverse = True)[:POPULATION_SIZE]

        best_genome = population[0].genome

        #print(population[0])
        piece_to_give = best_genome[1]
        position_to_play = self.indexToTuple(best_genome[4])
        #print((piece_to_give, position_to_play))
        return (piece_to_give, position_to_play)
```

In particular, we handle the case in which we have just one single move. Very rare, but still good to use less resources in this particular case.

A very particular piece that I want to highlight is the following. Sometimes the algorithm gets stuck into a negative loop, where all genomes have low fitness. In such cases, even if not common, the algorithm seems to lose. In order to avoid this, I reinitialize the population with new random elements, and include

them in the offspring. By doing this, the algorithm seems to get back up and avoid a loss.

```
    best_fitness = population[0].fitness
    # sometimes the algoithm gets stuck in finding only losing
    ↪  solutions.
    # by reinitializing the population, we introduce randomness
    ↪  again
    # almost always, the algorithm gets back up
    if best_fitness < 0:
       print("yeuch")
       population += self.initPopulation(strategyType,
       ↪  POPULATION_SIZE * 2)
       population = sorted(unique_population, key=lambda i: i[1],
       ↪  reverse = True)[:POPULATION_SIZE]
```

I tested this Genetic Algorithm against a random player. With 300 matches, it wins 99.5% of the times. I also tested the algorithm against a very strong MinMax, made by my friend Federico Boscolo (`https://github.com/feurode46/`), and it seems to lose almost all the time.

Genetic algorithms and Minimax algorithms are two different approaches altogether. Minimax is a deterministic algorithm that considers all possible moves and their outcomes to find the optimal move, whereas Genetic algorithms are heuristic and based on simulating evolution to find good solutions.

Minimax is considered stronger in games like Quarto because it can look ahead several moves and make decisions based on a comprehensive evaluation of the game state, whereas genetic algorithms may struggle to find the best move because they rely on random chance and may not take into account all relevant factors. Additionally, Minimax has proven successful in deterministic, perfect information games like chess and checkers.

In summary, Minimax is stronger in games like Quarto because it has a more systematic and comprehensive approach to decision making, while genetic algorithms have some limitations in this regard.

```
Everything = * = 42 ASCII CODE
```