Report Computational Intelligence

Lorenzo Greco s3061612

Index

Note:

Labs has been made in collaboration with:

- Luca Barbato s320213
- Giuseppe Roberto Allegra s305063

QUIXO has not been developed together but strategy has been discussed.

Lab 1

During this laboratory, we conducted experiments on heuristic functions h.

We initially considered a solution based on remaining coverage, which was calculated as the difference between the total number of elements in the set and the number of elements already covered.

In an attempt to optimize it, we arrived at the same conclusion as heuristic function h1, already present in Professor Squillero's repository.

Since heuristic function h2, which optimized h1, was already present in the professor's repository, we attempted to further optimize h1 by questioning whether rounding using ceil was causing the loss of useful information for PriorityQueue ordering.

We, therefore, sought information on the implementation of the PriorityQueue and discovered that the priority doesn't necessarily have to be an integer.

Our hypothesis was that for large values of *PROBLEM_SIZE* and *NUM_SETS*, rounding was actually making the algorithm worse, as sets with different coverages were being rounded to the same integer and therefore not optimally sorted. However, after running various tests, we realized that the ceil operation is indeed functional to the algorithm, and our hypothesis was incorrect because a fractional value wouldn't be suitable as a cost estimate because a fraction of a set in a solution is not valid, since the ceil operation represents the minimum number of sets required to cover the missing elements, ensuring that the estimate is optimistic.

Lab 2

Genome & Evolutionary Strategy

Regarding the genome, not wanting to rely on the information that the best moves are those with NimSum != 0 (because within the context of evolutionary algorithms, the optimal algorithm is theoretically unknown), we have conceived a genome consisting of three parameters.

The first parameter 'preference' indicates the probability of following or not following the strategy proposed by the genome.

The second parameter 'use_lower_half' indicates the genome's strategy. We identify the min and max of NimSum for the moves and calculate the average value. Therefore, based on this parameter, we consider as possible moves either the lower or upper half of this set.

The third parameter represents the 'fitness,' i.e., the proficiency of the individual in the game.

It is certainly not the optimal strategy, but, on the other hand, the optimal strategy already exists. Thus, we wanted to create a strategy that is random but still logical.

Parent Selection

The parent selection, as with many evolutionary algorithms, occurs through a tournament where the winner is chosen based on the 'fitness' parameter of their genome. The player is rewarded with one point for each won game.

Crossover

The two parents have a weighted probability in terms of their fitness to transmit or not transmit a specific genomic parameter to their child.

Mutation

Mutation occurs based on a mutation rate, typically very low. An attempt is made on each parameter of the genome independently. If the mutation occurs, the new parameter is always chosen randomly.

Simulation

Finally, the simulation is executed. Players are trained against the optimal strategy, and the parameters are configurable.

At the end of the simulation, the best player from the last generation is selected. This best player has the honor of challenging, in a round of 1000 games, other strategies, including a rematch against the optimal strategy.

Final Considerations

Our strategy is far from being optimal, and being pseudo-random, it has a limited range of improvement.

Conducting numerous tests and varying simulation parameters, both with small and large numbers, we observed that it always starts with a win rate in the first generation of about 35%, evolving to a maximum of around 42%. There is an improvement. It improves very quickly, especially in the first five generations, reaching its maximum value and then stabilizing around that value.

Thinking that it might be a local optimum, we tried increasing the mutation rate to prevent it from settling on a percentage. Unfortunately, it didn't help, rather, increasing the mutation rate emphasizes the random component of the strategy, leading to a loss of evolutionary progress made up to that point.

In the final matches, the elected best player has a positive win rate against Gabriele's strategy, while winning about 46% of the time against the pure random strategy. Against the optimal strategy, the win rate always hovers around 43%.

We attach an example of a simulation where you can see the slight improvement.

Simulation:

Generation 1: They won 1801 games in 5000 games
The percentage of won games of this generation is: 36.02%

Generation 2:
They won 1901 games in 5000 games
The percentage of won games of this generation is: 38.02%

Generation 3:
They won 2014 games in 5000 games
The percentage of won games of this generation is: 40.28%

Generation 4:
They won 2044 games in 5000 games
The percentage of won games of this generation is: 40.88%

Generation 5:
They won 1930 games in 5000 games
The percentage of won games of this generation is: 38.6%

Generation 6:
They won 1991 games in 5000 games
The percentage of won games of this generation is: 39.82%

Generation 7:
They won 2032 games in 5000 games
The percentage of won games of this generation is: 40.64%

Generation 8:
They won 2048 games in 5000 games
The percentage of won games of this generation is: 40.96%

Generation 9:
They won 2035 games in 5000 games
The percentage of won games of this generation is: 40.70%

Generation 10:
They won 2056 games in 5000 games
The percentage of won games of this generation is: 41.12%

Best Player vs Gabriele Strategy:
They won 514 games in 1000 games
The percentage of won games is: 51.4%

Best Player vs Pure Random Strategy:
They won 469 games in 1000 games
The percentage of won games is: 46.9%

Best Player vs Optimal: The Rematch!
They won 406 games in 1000 games
The percentage of won games is: 40.6%

As you can see, it quickly surpassed the 40% winning threshold, only to experience a relapse. Unfortunately, in this simulation, it did not exceed a 41% win rate. However, the best player performed very well against Gabriele's strategy and the random strategy, achieving 'only' a 40.6% win rate in the rematch against the optimal strategy. During some tests, we recorded values even higher than 45%. What a pity, maybe it was tired from all the previous games...

Peer Review:

Done:

To Alessandro De Marco (s317626):

Hello Alessandro,
I find your idea of characterizing the genome very clever. By leveraging nimsum as a spectrum of possible choices, your strategy, regardless of the number of wins per generation (which, however, increases significantly over generations), has great evolutionary potential, as evidenced by the collected data, where the population initially randomly targeting all nimsum values tends to concentrate in a specific range as it evolves.
I also appreciate providing two methods of parent selection because it provides a more in-depth view of the problem, highlighting aspects that often take a back seat, such as convergence speed and population diversity.
Regarding the crossover function, I appreciate the idea of having a bias that influences the result (with the check if total_fitness > 0).
The mutation is standard but efficient.
Overall, the code is well-structured and very readable. The efforts to plot all the results to make them clear are commendable, and the report is simple but effective, containing everything one needs to know and providing an excellent guide to understanding your code.
Well done, and good luck with the upcoming labs!

To Antonio Ferrigno (s316467):

Hi Antonio,
Your evolutionary strategy consists of choosing moves based on those that lead to a safe condition, exploiting knowledge of the ideal nimsum. If this is not possible or we are already in a safe condition, you call a function that chooses the move based on the individual's genome.
In my opinion, what is not good about this approach is that the choice is based on the best possible move at the moment.
This is incorrect because in evolutionary strategies, in theory, the best move or the optimal strategy is not known. This is precisely why evolutionary strategies are used because the goal is to approach the optimal without knowing it, through the evolution of generations.
In your case, the strategy is the optimal one with the exception of when it is not possible, and then the genome intervenes, making a pseudo-random choice. Overall, it is therefore an optimal strategy that leaves little room for evolution.
Furthermore, during the population evaluation phase, they play against themselves. In a more realistic context, this would lead to little evolution in the species and a local minimum.
That being said, I appreciate the idea of valuing quick wins.
It is also commendable the effort to plot various graphs and statistics.
Overall, the code, regardless of its correctness or not, is well-readable.
Good luck with the next labs!

Received:

From Riccardo Renda (s310383):

The adopted strategy appears to be valid, plus also the implementation of mutation and crossover function are interesting.
The code is well-commented and also the README is very clear providing many statistics about the results.
I have only one concern regarding the creation of a new generation: it seems that after a child is generated, it is immediately subjected to mutation. This could potentially lead to the loss of good results from crossover since these operations are not mutually exclusive.
Eventually increasing the number of generations could yield better results.

My comment on the review:

Thank you for your review! Yes, you are right, after a child is generated, it is immediately subjected to mutation, but the mutation itself occurs with its mutation rate that is very low, so rarely it affects the new individual. Anyway, thanks for the suggestions (I tried with more generations, but unfortunately, the win rate did not increase).

Lab 3

Genome & Evolutionary Strategy

The code implements an evolutionary algorithm to solve an optimization problem.

In an effort to minimize fitness calls, we aimed to avoid unnecessary extra calls that didn't introduce significant benefits in the search for the best fitness.

The genome represents the genetic makeup of an individual in the population. In our context, an individual has a genome composed of a sequence of binary values (0 or 1).

The population of individuals evolves through successive generations. In each generation, individuals are evaluated based on a fitness function, and the best ones are selected as elite. The remaining individuals are chosen through random tournaments and undergo crossover and mutation, introducing genetic variations, contributing to genetic diversity, and discovering potentially better solutions.

At the end of the evolutionary process, the algorithm returns the individual with the highest fitness from the final population as the optimal or approximate solution to the problem.

Parent Selection

Parent Selection occurs in the following way:

Elite Selection: A certain number of elite individuals are selected based on their fitness performance. Elite individuals are those with the highest fitness performances and are kept unchanged in the next generation without undergoing crossover or mutation.

Parent Selection for Crossover and Mutation: To complete the new generation, two components of the elite are chosen in turn as parents and undergo crossover and mutation.

Therefore, parent selection is a combination of elite selection, which preserves the best individuals, and random choices among elites, which introduce randomness in parent selection for genetic diversity.

Crossover

The *crossover* function takes two parents, *parent1* and *parent2*, and generates a child by combining the information from the parents.

The point where "crossover" occurs is randomly chosen between the second and the last element of the genome *(len(parent1) - 1)*. Therefore, *crossover_point* represents the position (index) in the genome where the crossover between the two parents will occur. Up to the crossover point, the genome of *parent1* is copied, and after that, the genome of *parent2* is copied.

This helps maintain genetic diversity among individuals in the population during evolution.

Mutation

The *mutate* function takes an individual represented as a sequence of binary values (0 or 1) and applies mutation.

For each bit in the individual, there is a probability *(MUTATION_RATE)* that the bit will be flipped: from 0 to 1 or vice versa.

Simulation

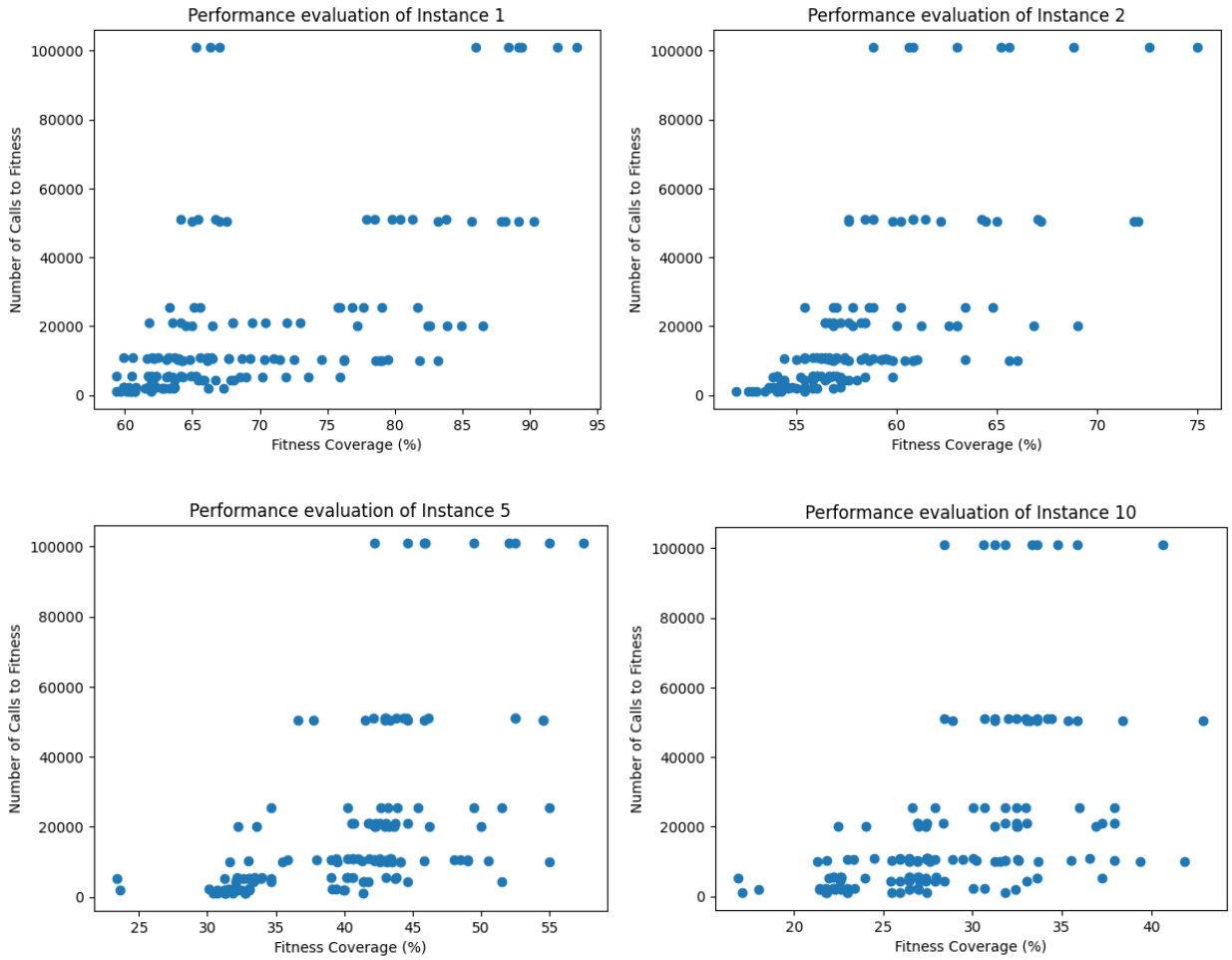
The simulation tests the possible configurations of parameters *(POPULATION_SIZE,MUTATION_RATE, GENERATIONS, GENERATIONS)* to search for the optimal setup that yields the best solution for each problem.

Graphical Performance Analysis

As you can see from the output, what influences the number of fitness calls, besides the evolutionary strategy used, are only the population size and the number of generations. The parameters *mutation_rate* and *elitism_percentage* affect the best fitness.

A higher number of population and generations corresponds to a higher best fitness, but not always.

Each point on the graph represents a combination of parameters (population size, generations, mutation rate, elitism percentage). Points in the bottom right represent the best combinations; in the top left, the worst. From numerous tests, it emerged that the best values for the mutation rate and elitism percentage parameters are around 0.001 and 0.1.



Final Considerations

After various tests with different evolutionary strategies that aimed to minimize fitness calls, we opted for this algorithm. Furthermore, reducing fitness calls too much seemed impractical without compromising the success of the best fitness too much.

As regards the number of fitness calls, since it's not always true that a greater number of population and generations necessarily corresponds to a higher best fitness, it can be said that a greater number of fitness calls does not necessarily correspond to a greater best fitness. This implies that achieving high best fitness values is possible even with a relatively low number of fitness calls.

As regards the problems, increasing the instance of the problem worsens the best fitness; since the maximum values of population and generations are not reached, the number of fitness calls decreases as the instance of the problem increases.

Peer Review:

Done:

To Claudio Savelli (s317680):

Hello Claudio,

First of all, I find it slightly challenging to provide a peer-review due to the amount of work involved.
Therefore, I will try to proceed in an organized manner.

In addition, you conducted two additional experiments (neural_network and valhalla) that are not included in the readme. Since they are not covered in the readme, I was unsure whether to evaluate them. Due to a lack of knowledge in neural networks and as they are not part of the lab assignment, I did not assess it (even if I had wanted to, I wouldn't have been able to at this time). Regarding the 'Valhalla' strategy, I read, studied, understood, and appreciated it, but it will not be the subject of this peer-review as it is not included in the readme.

I greatly appreciate providing different strategies to compare them.
In this case, you used three different strategies: a basic one, one with islands, and one with segregation in the islands.

As if that weren't enough, even in the basic one, you made an effort to provide multiple parent selection methods, mutation functions, and crossover functions. A in-depth study of this strategy, as you did, would have been enough to meet the requirements of the lab, but you went beyond; congratulations!
All these functions (parent selection, mutation, crossover) are well-written, efficient, and also very clever. Again, well done!

I don't know where you got the idea of islands and segregation; it took me a while to understand how they work, and I must say these are really interesting strategies! In this regard, I would have preferred more

comments or sections of text explaining the idea behind these strategy rather than leaving all the effort at the discretion of the reader.

Other very interesting ideas you applied include dynamic mutation probability, with the respective diversity threshold, and the 'cooldown_time'. If I were to talk specifically about each one, I would never finish writing, so I'll just say I found them really clever additions!
Oh, I almost forgot, 'memoization': I really appreciate its implementation because it shows the effort to reduce the number of fitness calls, which was the lab's task, so nice job!

To be honest, I would have preferred a more in-depth analysis of the results, rather than just a table with all the results. Additionally, more explanations about the strategies used and how they function would have been beneficial. It might have been useful to test additional parameter configurations to evaluate further optimizations and better understand the algorithm's behavior in different conditions. Summing up and drawing conclusions about the strategies would have improved code understanding as well.

Overall, the code is well-structured and readable, but not so easy to understand, especially if you are not very comfortable with the strategies used. The results are clear, but further analysis and conclusions about them would not have hurt.

Well done, and good luck with the upcoming labs!

To Giacomo Fantino (s310624):

Hi Giacomo,
First of all congratulations on your work!
Overall, your work is very clear, organized, and readable. I appreciate the effort you put into proposing different strategies for comparison: a simple evolutionary algorithm, one with diversity in parent selection, one with adaptability, and one with both diversity and adaptability.
Specifically, the mutation function, parent selection, and crossover functions are simple but non-trivial, and not too difficult to understand; but adding a few more comments on these functions might have improved overall readability and comprehension.
You focused extensively on comparing different strategies rather than explaining the strategies themselves and attempting to reduce fitness calls with a strategy.
The results are well presented and commented on, and the conclusions drawn are accurate and clear.
I appreciate the fact that you tested the algorithms over a high number of generations.
It would have been beneficial to have some additional analysis on how the different parameters influence the results, rather than keeping them static during the experiments.
In conclusion, I think you've done a good job. It's evident that you invested a lot of time and effort, and the result is satisfactory and aligned with the requirements of the assignment.
Good luck with the next labs!

Received:

From Riccardo Renda (s310383):

Starting with the structure and clarity of the code, I think your notebook is elegant and easy to follow. Thanks to the numerous markdowns, comments and the extremely explanatory README the reader can immediately understand the main steps of the Lab and how they were processed. The only note I would like to make is regarding the presentation of the results, where I would advise you to make the best solution obtained with each instance of the problem more visible (You have indeed highlighted it within the output file, but it is 2000+ lines). Moreover, even the graph, which could be very interesting to analyse, partially loses its usefulness if which combination of parameters each point goes back to isn't traced. Having said that, I really compliment you on the overall clarity!

As far as the code is concerned, I seem to have found a dissimilarity between what is written in the README and the notebook. In fact, within the code, it seems to me that no tournament method (taking a random subset of individuals each time and choosing from those the best one) is used to select the parents for each generation, but simply from the best (elite), two parents are randomly and iteratively sampled and generated from those an offspring. This method I imagine could easily incur a local minimum, considering that at each generation potentially the elite could vary very little.

Also, for the reason just mentioned, as a possible improvement that would probably drastically decrease the number of fitness calls used I would suggest you try using memoisation, i.e. saving locally the results obtained by evaluating the fitness function on a particular gene, not having to call it again when you need to review it. Furthermore, it might be useful to add a stopping criterion, to stop the generation in case for many consecutive iterations the best solution(s) does not change.

Another aspect that aroused my interest was the mutation function. I found your very exploratory approach very interesting, considering that with the maximum 'mutation_rate' value about 10% of the solution changes for each offspring. Most likely the best hyperparameter was the lowest one precisely because even once you got close to the solution you were mutating your gene too much, moving away from the minimum and thus continuing to choose elite elements. I would advise you to try a hybrid approach, keeping the mutation rate high during the first generations (exploration), and then gradually lowering it in the final ones (exploitation)!

Overall, I think you did a great job generating a clear, formative and effective solution. Reading your README made me curious about the other approaches you tried before this one, you could have mentioned them! Thank you for the interesting insights! Until next time!

Lab 4

N.B.

I don't know if it's an official rule of the game, but we assumed that X always moves first. This information can be useful as a key for interpretation and understanding of some experiments.

Objectives

For the resolution of this laboratory, we started from the professor's strategy and tried to improve it, testing how artificial intelligence behaved in different scenarios.

Experiments conducted

AI moves first (X)

As we started, we focused on implementing the training model into the algorithm of an AI-controlled player.

The underlying idea is that a player should be able to find the best move for each position. The algorithm we implemented leverages the knowledge of all states acquired during the training phase to find the best move for each state.

Of course, the effectiveness of this solution depends on the success of the training session. With a sufficient number of games during the training phase, for a simple game like tic-tac-toe, we can assume to have visited all possible states multiple times, thus having an optimal dictionary capable of suggesting the best move in every situation.

The training games use random moves to try to explore as many states and as many times as possible.

After implementing this algorithm, we wondered how good this player actually was. Then we put it to the test in a series of 1000 games against a player who moves randomly, obtaining excellent results:

Statistics against the random player over 1000 games (Using X):
Wins: 995
Loss: 0
Draws: 5

From these results, we can assume that the AI has reached the highest possible level and (with the X) plays perfectly.

But winning doesn't necessarily mean making the best moves.

So, to be sure that it didn't make mistakes, we implemented the function that prints move by move the various states of a single game.

The tests performed, showed that every time the AI had the opportunity to win, it closed the game, confirming our hypothesis that it had now reached perfection.

It may seem obvious, but it's not. With a low number of games during training, it could happen that, not exploring all possible combinations of moves sufficiently, it chose non-optimal moves while still winning. This

happened to us with a number of games in the training phase equal to 100.000, prompting us to opt for a greater number of games, equal to 500.000 .

AI moves second (O)

After achieving this result, we wondered how the AI would behave if it had to use O and move second.

Initially, we adapted the algorithm that selects intelligent moves to find the best moves for O.

Instead of choosing the move that led us to the best state for X, we selected moves that led us to the worst state, therefore more advantageous for O.

Although it played quite decently, it tied and lost a bit too much. Starting second, when X plays well, if you are skilled, you can manage to draw. Therefore, we were not concerned about the number of draws, but there were too many losses for it to be considered a good AI.

In addition, the training (for X) stored and updated the state value based on how advantageous it was for X, also considering the difference between the final result of the game and the value of that state up to that point.

Therefore, it didn't fit perfectly to our case.

Then we decided to create a training program exclusively for O, with a dictionary that learns the best moves in every situation and that could, at least, draw even against the best opponents.

Training for O: Phase 1

The training principle was the same as for X.

The initial results were not very satisfying. We managed to reduce the losses a bit, but the AI was still far from being a very good player.

The major weakness come out when we tried to make this AI play against the one that plays perfectly with X.

The 1000 test games ended with a score of 1000 to 0 for X.

This experiment highlighted how O did not know the strategy to draw against a perfect X. This is because, in the 500,000 randomly played training games, the times the computer randomly plays the best moves for X and O's best moves to draw are too few, or not highly valued enough, for O to learn and memorize the optimal defensive strategy in its dictionary.

Thus we started the phase 2 of training for O...

Training for O: Phase 2

If Mohammed will not go to the mountain, the mountain must come to Mohammed...

Since O, during the training phase, did not have the opportunity to learn the defensive strategy against the perfect X, we designed a phase 2 of the training where we exclusively trained our AI against the perfect X, taking care to record the results and update the relevant states in O's dictionary.

During this second training phase, the AI, moving randomly during these games, had the opportunity to draw and discover the ideal defensive strategy against X and memorize it.

To ensure that once it drew, it wouldn't forget how it had done it, we introduced a second reward function, different from the one used during phase 1, which would reward draws more significantly.

Passed this second training phase, O challenged X in search of its revenge.

This second time, things went a bit differently...

Statistics AI vs AI:
Wins X: 0
Loss X: 0
Draws: 1000

O managed to draw all the games, demonstrating that it had learned the defensive strategy against X's best moves, and therefore, the training was a success.

Having overcome this obstacle, there still remained the issue of losses. Ideally, it should be able to draw every game, insted of having consistently a 12% rate of losses.

Also in this case, we made sure to verify that in games won by O, it used the best moves. Therefore, we implemented the function that prints move by move the various states of a single game.

Then we checked that it knew the best moves.

It reamained to understand why it continued to lose some games.

We gave ourselves the explanation that during the phase 1 of training, it couldn't explore all possible states in dept or couldn't face enough those few games it lost to memorize the counter-moves.

Thus we designed the third phase of the training.

Training for O: Phase 3

Following the same approach as in phase 2, we saved the games that O lost in data structures and then replaying them during the third phase of training to update its dictionary more accurately with the best moves for each state.

We implemented the new reward function and the function to play against predefined sequences of moves. It is impossible to make O replay the exact same games it lost since O moves randomly. However, over a large number of games, it is likely to replay similar or identical games, hoping to win them and memorize the winning moves.

Then we stared the third phase of training.

As a result, the losses decreased to approximately 4%, but they were still present.

Therefore, the third phase of training did not achieve the desired goal.

P.S.

Afterwards, we conducted further tests, focusing our efforts on the training parameters, like *epsilon* and the *reward* in case of victory, draw, or defeat.

Surprisingly, by trying various values of epsilon and slightly adjusting the rewards, we achieved the lowest recorded result: 0.3% losses.

Moreover, with these new parameters, there was no need for the phase 2 to manage to draw against an AI-controlled X. The phase 3, with these parameters, proved to be only detrimental, worsening the performance.

At this point we suppose phases 2 and 3 may not be very useful, but they were still nice experiments from an instructive and recreational perspective.

Intelligent Trainig

As a last experiment, we tried a different strategy during the training phase.

Until now, during the training phase, games were played randomly. We wondered, whether or not, it would be a good idea to leverage the knowledge acquired up to that point already during the training phase.

Statistics against the random player over 1000 games (Using X):
Wins: 654
Loss: 277
Draws: 69

The results were a bit disappointing.
This is because, being influenced right from the start by the acquired knowledge, the AI fails to explore in dept all possible moves or tends to focus only on certain paths it believes might be winning, possibly ignoring other better paths.

Challenge the AIs!

After finishing our experiments, we thought it could be fun to challenge ourselves against the AIs we created.

They are unbeatable...

The few combinations that O fails to block are so rare and suboptimal that any human player would never play them, effectively being unable to beat the AIs (both with X and O) ever!

Peer Review:

Done:

To Thomas Baracco (s308722):

Hi Thomas,

First of all, I want to tell you that I appreciated the idea of using a q-learning algorithm to train the agents.

The code itself is very clear and readable. The q-learning algorithm does its job, and it's appropriately commented.

Perhaps I would have preferred a greater separation between the game logic and the training logic, with comments in specific sections rather than inside the code, but these are personal preferences.

You have certainly fulfilled the requirements of the lab.
Maybe to make things more interesting, you could have done an analysis on the training parameters behaviour.
For example, how many games are needed to train your agent optimally, how positive and negative rewards influence the q-table, improving or worsening the training outcome.
It would have been interesting to see such a study with the appropriate results presented and explained clearly, perhaps with your reflection on the achieved results.
At the end of training, your agent has a winning rate of 90.88% against a random player; are the remaining games losses or draws? What would happen if it played second? What if two intelligent agents faced each other?...
These questions would be other good examples of analysis that you could have done.

That being said, this doesn't take away from the fact that you've done an excellent job. You've implemented a good algorithm and achieved a satisfactory result. Congratulations!

I wish you the best for the exam, and good luck! Let's not give up now that we're almost at the finish line!

To Silvio Chito (s309619):

Hi Silvio,

I want to start by saying that I appreciate your choice of using a q-learning algorithm.

The code overall is readable and clear, well-commented where necessary, with a clear distinction between different sections and phases of the algorithm – good job!
I also appreciate the fact that there is a clear and concise README.
The decision to use a fixed probability for choosing actions at each state by exploiting the epsilon-greedy strategy makes sense and is commendable.

One feature I particularly appreciate is the reward management system. Many, including myself, have relied on simple scores based on whether the player wins, draws, or loses. I find your implementation of rewards clever and original, compliments!

The algorithm is implemented correctly, and for this reason, a more in-depth analysis of the results obtained would have been interesting.

From the code, it's evident that the agent can be trained with both X and O. I appreciate the idea, but a comparison between the two cases would have been useful.
I've already mentioned that I find the reward management system original, but is it genuinely better than a more simplified version? Your agent achieves a 71% win rate against a random player; how many games does it lose or draw? It would have been interesting to analyze these aspects as well.

In conclusion, the lab has been completed, and there is evident effort on your part to customize and improve a generic q-learning algorithm, well done!

I wish you good luck for the exam and your future exams!

Received:

From Andrea Galella (s310166):

Hi Lorenzo, I'm leaving you some comments about your implementation of the Lab 10, hoping you will find them useful.

About the training:
Overall I think you did a great job. The training of IA moves first (X) it's pretty clever and well done. The training of IA moves second (O), and your idea of implementing a training program exclusively for it, really shows you put a lot of effort in this.
To get better results in your last implementation (where you tried to train the agent without random games), I would suggest to implement a Q-learnign algorithm. In this way you can try to achieve a more suitable rappresentation of the action-value pair by using a Q-table and updating it using the Bellman equation tweaked by the learning rate and the discount factor.

About the games:
The solution shows you achieved great results both by moving first and by moving second. I liked how you used different ways to show the goodness of your solution, by allowing us to check a Random x AI match, an AI x AI match, and even to test it by ourselves.

About code readability and documentation:
Even without comments the code is very clean and easy to understand. I think you did a great job with documentation, the README.md file is complete, it doesn't just give informations about the code but also about the problem and the way you handled it step by step.

From Claudio Savelli (s317680):

Hello again! I saw your Lab10 and was very interested, so I decided to review your work!

Starting with the writing and structure of the code, I found it excellent to divide each method into its own cell with a title, making easy to understand the purpose each function had within your workflow. The only thing I have to say about this is that it would probably have been preferable to write general functions that were valid whether the agent played first or second (or both), thus avoiding writing a lot of unnecessary code and making the work less verbose. This could easily be achieved by passing the two players to intelligent_game (rather than intelligent_game_X or intelligent_game_O).

As for the experiments you carried out, I found them very interesting and really complete. I appreciated how you put the whole workflow you followed inside the README, pointing out its strengths and weaknesses, making the analysis of the work extremely more immediate and clear, so much so that it seemed as if I were working with you!

Another plus point is that you not only limited yourself to optimising the agents against a random player but also trained them to play against optimal tic-tac-toe players, forcing a draw every time, even going beyond the laboratory objectives!

The only note I want to make about the experiments is the number of games played during the training phase. In fact, you could have considered, instead of increasing the number of training games from 100,000 to 500,000, which also means slowing down the training time, exploiting the nature of the tic-tac-toe table! In fact, for each position in the play, there can be up to 4 totally identical positions by simply rotating the board (unless the first move is 'X' in the middle). In this way you don't only decrease the number of games to be played by a factor of 3 but also ensures that you see more states in large numbers so that the value dictionary covers all possible positions more evenly.

In conclusion, I think the work is superb, perfectly covering not only the required objectives but also going beyond them, with a neat code accompanied by an excellent description. Great job!

From Alexandre Senouci (s321473):

Your code is really well organised and show each step of your research and work for this lab.
I don't know if you first implemented ia first move and next ia seconde move, or if you made those method at the same time.

Increasing the difficulty from random move to smart move make the learning rate smoother than being directly confronted to smart move.

The only negative point is maybe the time of training and the amount of state that you need to store, but making a near perfect player compensate this point.

QUIXO

This project implements an artificial intelligence based on the Minimax algorithm with Alpha-Beta pruning to play Quixo, a strategic board game.

Minimax Player

Introduction to the Minimax Algorithm:

The Minimax algorithm is a technique used in the field of artificial intelligence for finding optimal decisions in zero-sum games, such as chess, tic-tac-toe, or other board games. Its goal is to find the best move to make in a given game state.

The Minimax algorithm works by performing a layered search within the tree of possible moves, up to a certain depth. Specifically, the algorithm considers all possible moves available to the current player, then simulates the opponent's responses to each of these moves, and so on until reaching a certain depth in the tree of possible moves.

Once the maximum search depth is reached or the game is over (e.g., when one of the players wins or a draw occurs), the algorithm evaluates the final state of the game by assigning a score. This score is then propagated backward through the tree of moves to the root, using a principle of maximizing (Max) for the current player and minimizing (Min) for the opponent.

The Minimax algorithm aims to maximize the positive outcome for the current player and minimize the negative outcome for the opponent. This is achieved through a recursive evaluation of all possible game scenarios, taking into account the likelihood that the opponent chooses the best move in response to the current player's moves.

However, the Minimax algorithm can be computationally expensive as it explores the entire tree of possible moves, becoming impractical in games with a high number of possible moves or with a very large search depth. To mitigate this issue, a technique called pruning is often used, such as Alpha-Beta pruning, to cut off branches of the tree that do not affect the final outcome, thereby reducing the number of nodes to explore.

Implementation:

The Minimax algorithm is implemented in Quixo through the *MiniMaxPlayer* class, which extends the *Player* class.

This class includes 3 methods in addition to the constructor: the method to make a move *(make_move)*, the method to run the recursive algorithm *(minimax)*, and the method to evaluate a game state *(evaluate_state).*

make_move(self, game: "Game") -> tuple[tuple[int, int], Move]:

This method is called to choose the best move in the current state of the game. It performs a search of all possible valid moves for the player, and for each move, it invokes the Minimax algorithm with Alpha-Beta pruning up to the specified maximum depth and returns the best move found. If no better move is found, a random move among the possible ones is returned.

```
14      def make_move(self, game: "Game") -> tuple[tuple[int, int], Move]:
15
16          best_points = float("-inf")
17          best_move = None
18          current_player = game.get_current_player()
19          self.player = current_player
20
21          moves = get_possible_moves(game.get_board(), current_player)
22
23          alpha = float("-inf")
24          beta = float("inf")
25
26          for move in moves:
27
28              board = deepcopy(game.get_board())
29              game_cloned = Game(board, game.get_current_player())
30              game_cloned.move(move[0], move[1], current_player)
31
32              points = self.minimax(game_cloned, self.max_depth, alpha, beta, False)
33              if points > best_points:
34                  best_points = points
35                  best_move = move
36              alpha = max(alpha, best_points)
37              if beta <= alpha:
38                  break
39
40          # Return the best move, or a random move if no best move is found
41          if best_move:
42              return best_move
43          else:
44              return random.choice(moves)
```

minimax(self, game, depth, alpha, beta, findingMax):

This method implements the Minimax algorithm with alpha-beta pruning. It evaluates all possible legal moves and selects the one that maximizes the advantage of the current player or minimizes the advantage of the opponent, depending on the turn. It uses alpha-beta pruning to eliminate unpromising branches in the search for the optimal move.

```
46      def minimax(self, game, depth, alpha, beta, findingMax):
47          # Minimax algorithm with alpha-beta pruning
48          current_player = game.get_current_player()
49
50          if depth == 0 or game.check_winner() != -1:
51              # Evaluate the game if it's at max depth or there is a winner
52              return self.evaluate_state(game)
53
54          if findingMax:
55              max_evaluation = float("-inf")
56
57              #print(f'player MAXIMIZING:{current_player}')
58
59              for move in get_possible_moves(game.get_board(), current_player):
60
61                  game_cloned = game.clone()
62                  game_cloned.move(move[0], move[1], current_player)
63
64                  eval = self.minimax(game_cloned, depth - 1, alpha, beta, False)
65                  max_evaluation = max(max_evaluation, eval)
66                  alpha = max(alpha, eval)
67                  if beta <= alpha:
68                      break
69              return max_evaluation
70          else:
71              min_evaluation = float("inf")
72
73              #Switching player to minimize opponent's state
74              current_player = (self.player + 1)%2
75
76              #print(f'player minimizing:{current_player}')
77
78              for move in get_possible_moves(game.get_board(), current_player):
79
80                  game_cloned = game.clone()
81                  game_cloned.move(move[0], move[1], current_player)
82
83                  eval = self.minimax(game_cloned, depth - 1, alpha, beta, True)
84                  min_evaluation = min(min_evaluation, eval)
85                  beta = min(beta, eval)
86                  if beta <= alpha:
87                      break
88              return min_evaluation
```

evaluate_state(self, game):

This method evaluates the current state of the game and returns a score based on its convenience for the current player.

In particular, it returns 5 if the current player has won, -5 if the opponent has won, and otherwise calls the *get_intermediate_state_evaluation* function to obtain an intermediate evaluation of the game state.

get_intermediate_state_evaluation(self, game)

To evaluate an intermediate state of the game, it is necessary to obtain an estimate in points of who is winning the game. To do this, the *get_intermediate_state_evaluation* method calculates, through the *find_max_sequence* method, the maximum (not contiguous) sequence of pieces (in vertical, horizontal, and both diagonals) for both players and evaluates the state as: *max_sequence_length_player* - *max_sequence_length_opponent*.

```
67  def find_max_sequence(player, board, board_size=5):
68
69      max_sequence_length = 0
70
71      # Check rows for pieces
72      for i in range(board_size):
73          row_count = sum(1 for j in range(board_size) if board[i][j] == player)
74          max_sequence_length = max(max_sequence_length, row_count)
75
76      # Check columns for pieces
77      for j in range(board_size):
78          col_count = sum(1 for i in range(board_size) if board[i][j] == player)
79          max_sequence_length = max(max_sequence_length, col_count)
80
81      # Check main diagonal \
82      diag_count1 = sum(1 for i in range(board_size) if board[i][i] == player)
83
84      # Check secondary diagonal /
85      diag_count2 = sum(1 for i in range(board_size) if board[i][board_size - 1 - i] == player)
86
87      max_sequence_length = max(max_sequence_length, diag_count1, diag_count2)
88
89      return max_sequence_length
```

Statistics

The Minimax Player has proven to be very skilled against Random:

```
MiniMax with depth: 1
Game 100: 100%|██████████| 100/100 [00:21<00:00,  4.75 game/s]
MiniMax wins: 100
Random wins: 0
MiniMax wins percentage: 100.0%

MiniMax with depth: 2
Game 100: 100%|██████████| 100/100 [04:35<00:00,  2.75s/ game]
MiniMax wins: 100
Random wins: 0
MiniMax wins percentage: 100.0%

MiniMax with depth: 3
Game 100: 100%|██████████| 100/100 [18:12<00:00, 10.93s/ game]
MiniMax wins: 100
Random wins: 0
MiniMax wins percentage: 100.0%

MiniMax with depth: 4
Game 100: 100%|██████████| 100/100 [3:43:15<00:00, 133.95s/ game]
MiniMax wins: 100
Random wins: 0
MiniMax wins percentage: 100.0%
```

For a more accurate and truthful analysis, the Minimax Player starts the first half of matches as second and the second half as first.

Considerations

In Quixo, a Minimax Player is able to outperform a random player even with low algorithm depth like 1.

This is because even looking only a few moves ahead compared to an opponent who plays randomly allows you to ensure victory in most cases.

Most times even with a depth of 1, it manages to win 100% of the games. With depths greater than 1, it manages to win 100% of the games against Random opponents.

Bonus

I have also implemented the ability to test your skills against AI or Random opponents, thanks to the *Human_Player* class which extends the *Player* class by overriding the *make_move* function.

I didn't implement the tie case in the AI vs Random tests because Random is so bad that it's impossible to tie against it when AI plays. However, I did implement the possibility of a tie in the HumanPlayer vs AI, HumanPlayer vs Random and AI vs AI matches because in this case, the possibility exists. To prevent the algorithm from getting stuck, I inserted a move counter that, after reaching 100 moves, declares the tie.

The games use *Game2*, which is a class that extends *Game*, providing functions to clone the game state, print the board with X and O instead of 1 and 0, and play interactively against other players, both as the first and second player.

```python
class HumanPlayer(Player):
    def __init__(self) -> None:
        super().__init__()
        self.player = None

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

        current_player = game.get_current_player()
        self.player = current_player


        while True:
            try:
                move_input = input("Enter your move (format: 'COLUMN ROW direction'): ")

                if keyboard.is_pressed('esc'):
                    raise KeyboardInterrupt

                move_parts = move_input.split()
                if len(move_parts) != 3:
                    raise ValueError("Invalid input format. Please enter COLUMN, ROW, and direction separated by spaces.")

                column = int(move_parts[0])
                row = int(move_parts[1])
                direction = Move[move_parts[2].upper()]

                if not game.is_valid_move(column, row, direction, current_player):
                    print("Invalid move. Please try again.")
                    continue

                return (column, row), direction
            except ValueError as ve:
                print(ve)
            except KeyError:
                print("Invalid direction.")
```