Lab 2:

To: Manuel escobar s307729

The code is well-organized, and you've applied good practices like using comments and modularization. The strategies for the Nim game and the game loop are clear. However, there are areas for improvement:

1. While you added a basic test case at the end, consider expanding your tests, especially for the evolutionary strategy.

2. Using the negative `nim\_sum` function to calculate fitness might be problematic. The optimal function uses `nim\_sum = 0`, so by using the negative, you unintentionally follow the optimal strategy, rendering the evolutionary approach ineffective.

3. When mutating the population, you replace parents with children. The logic of the operation is good but It could be beneficial to move the line of code creating children into a different function to enhance readability.

In summary, your code is functional, but addressing these points would make it more versatile and readable.

To: Alessio Cappello s309450

The code is well-structured, and the use of meaningful names for variables and functions makes it easy to understand. The overall logic and flow of the code are clear and easy to follow. The fitness function, which assesses the genome based on the win rate, is quite basic. To enhance its effectiveness, it might be beneficial to conduct additional trials against various strategies.

Lab 3:

To: Davide Natale s318967

Hi Davide, your code is well-organized, I appreciate the clear sections dedicated to individuals, populations, mutation, recombination, and the two types of evolutionary algorithms.

The mutation and recombination functions are well-implemented and easy to understand. I also like the flexibility added by using a list of three crossover functions with a random choice.

The implementation of the steady-state EA follows the standard structure for evolutionary algorithms. However, I couldn't find where the self-adaptive mutation probability is adjusted in the code.

In the generational EA, the inclusion of elitism by preserving the best individual in each generation is a good strategy.

The test cases for both steady-state EA and generational EA are well-documented and provide clear output. To enhance testing, I would suggest adding more cases with different parameter values such as POPULATION\_SIZE or num\_generations.

Overall, good job, and best of luck with your future labs!

To: Luca Sturaro s320062

Hi Luca, your code effectively implements a straightforward evolutionary algorithm (EA) with clear organization into functions, enhancing modularity and readability. However, consider adding a README to elucidate the algorithm's logic and document the achieved results.

The current implementation uses fixed values for parameters such as `GENERATIONS`, `POPULATION\_SIZE`, and `mutation\_rate`. To enhance the algorithm's performance, experiment with varying these parameters and explore additional crossover strategies.

Results are stored in the `results` list, a good practice. Yet, printing results for every generation, even with minor improvements, may hinder readability. Instead, consider printing the best results obtained when testing the algorithm with different parameter values.

Overall, the algorithm is functional. Well done, and good luck for the next labs!

Lab 4:

To: Andrea Galella s310166

Hi Andrea, your implementation is a reinforcement learning approach for training a tic-tac-toe player using the Q-learning algorithm. Here are some comments and suggestions to improve your code:

- Start with a brief description at the beginning of your code, explaining the script's purpose, the reinforcement learning approach applied, and any notable parameters used.

- Consider adding comments to clarify complex sections of the code, particularly those related to reinforcement learning. This will make it easier for others (or yourself) to understand the code.

- It's interesting to compare the performance of the reinforcement learning agent against a random player. I believe the player should achieve a higher win rate. When starting as 'X,' it should almost always win, while playing as 'O,' it should lose less and draw more. There's potential to increase the win rate.

- Displaying the Q-table after a single game is a good practice as it helps verify that the Q-values are being updated correctly.

- Introduce error handling for unexpected situations, like an invalid action selected by the agent.

- A more detailed readme would be beneficial, explaining the strategy employed, the roles of various functions, and the most significant parameters used.

These suggestions aim to improve the clarity, maintainability, and robustness of your code. Overall, your implementation is solid. Best of luck with your exam!

To: Donato Lanzillotti s316001

Hi Donato, I've reviewed your code and I really appreciate your strategy, especially because it closely aligns with the one I used. Your code structure is generally clear, and you've effectively used functions to modularize different parts of the code. However, it would be beneficial to include comments or docstrings to explain the purpose of each function and its parameters.

The training functions (`train\_agent\_1`, `train\_agent\_2`, `train\_agent\_2\_v2`) appear to be functioning correctly. Still, I would suggest considering the addition of a mechanism to save and load the state-action dictionaries to avoid the need for retraining each time.

I don't fully agree with assigning a higher reward for a draw than a win in the third function. Even if the agent starts playing second, it should always aim for victory. If winning is not possible, then it can settle for a draw. Adjusting the reward values could help achieve this desired behavior.

The testing functions (`policy\_vs\_random`, `random\_vs\_policy`, `policy\_vs\_policy`, `policy1\_vs\_policy2`, `human\_vs\_policy`) provide effective means to evaluate the trained agents. The results section at the end, where you test the agents against each other and a human player, is clear and informative.

I would recommend handling user input errors more gracefully in the `human\_vs\_policy` function. For instance, if the user enters a non-integer or an unavailable move, consider providing a clear error message and prompting again.

Overall, great job! Best of luck for your exam!