Politecnico di Torino

COMPUTATIONAL INTELLIGENCE

2023/2024

BARBATO LUCA s320213

DISCLAIMER

LABs have been made in collaboration with Lorenzo Greco and Giuseppe Roberto Allegra.
QUIXO has not been developed together but strategy has been discussed.

TABLE OF CONTENTS

LAB 01 - A*

README

SUBMISSION: 26 OCTOBER 2023 at 1.51 PM

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 02 - NIM

README
SUBMISSION: 17 NOVEMBER 2023 at 8.41 PM

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.

```
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...

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.

REVIEW TO FEKRI FARISAN

SUBMISSION: 24 NOVEMBER 2023 at 11.46 PM

Hello there,
First of all I wanna thank you for sharing your code with us.
You've added a reasonable amount of comments to the code, which is a good thing.
The algorithm you provided seems fine but I would suggest you some changes regarding different aspects of the code.
I suggest changing the line

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

to

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

for better understanding. This reflects a limit-case because it's not typical for both your algorithm and the opponent to remove only one object per turn in each round.
Regarding the fitness function, I geniunely appriciate the change of strategy stated by

```
ply = strategy[i%3](nim)
```

because that ensures a rotational pattern for the oppent's strategy selection.
I also like a lot this part

```
if player == 0:
 score = score + 10
else:
 score = score + num_moves*0.01
```

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

```
for i in range(18):
```

maybe into

```
NUM_ROUNDS = 18
for i in range(NUM_ROUNDS)
```

alwayss for better comprehension.
I also would have suggested you to increase this number but it would be probabily useless because I see a “main problem” I think you should take into consideration:
I notice that your algorithm’s **moves are determined prior to the beginning of the game**, as stated by the following code

```
for _ in range(NUM_MOVES):
 row = randint(0, NUM_ROWS-1)
 i.genotype.extend([(row, min(randint(1, row * 2 + 1), UPPERBOUND_K))])
```

without considering the current state of the game or the moves executed by your opponent. These moves are not adjusted based on these parameters during each turn.
So, basically, your algorithm just randomly makes moves without considering what’s happening in the game or what the opponent is doing.
With that said, I think you’ve done a good job, although there’s room for improvement.
Good luck for the next labs!

REVIEW TO IACOBONE NICOLÒ
SUBMISSION: 24 NOVEMBER 2023 at 11.46 PM

Hello Nicolò,
First of all let me thank you for the exhaustive ReadMe and for having commented the most of the code.
Your efforts in documenting the code are greatly appreciated.
You’ve applied all the standard techniques of evolutionary algorithms, but you’ve tried to represent the problem in a different way.
As I understood the **CustomHashTable** class is designed to serve as a data structure for storing scores associated with hash keys representing game states (the function **generate_hash_key** creates a unique hash key for the given game state).
This way of re-thinking the game is innovative and I would say quite impressive.
While the fitness score is determined only by the number of wins, the evaluation remains constistent due to the reasonable number of games played as stated in

```
NUMBER_OF_FITNESS_GAMES = 200
```

So far, I think you’ve done an excellent job!
Keep working on it and good luck for the next labs!

LAB 03

README
SUBMISSION: 4 DECEMBER 2023 at 12.07 AM

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

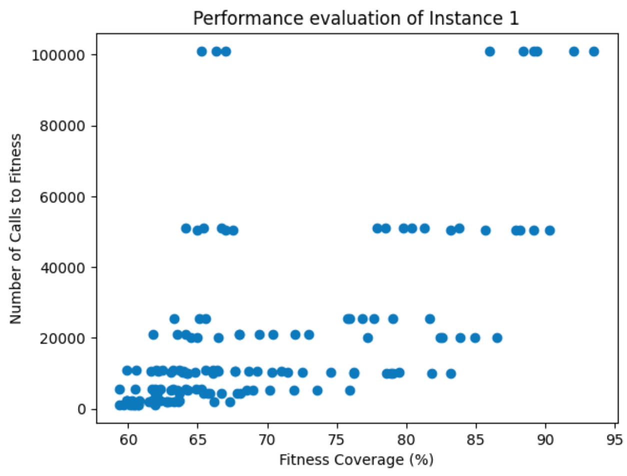
The **mutate** function takes an individual represented as a sequence of binary values (0 or 1) and applies mutation.
For each bit in the individual, there is a probability (MUTATION_RATE) that the bit will be flipped: from 0 to 1 or vice versa.

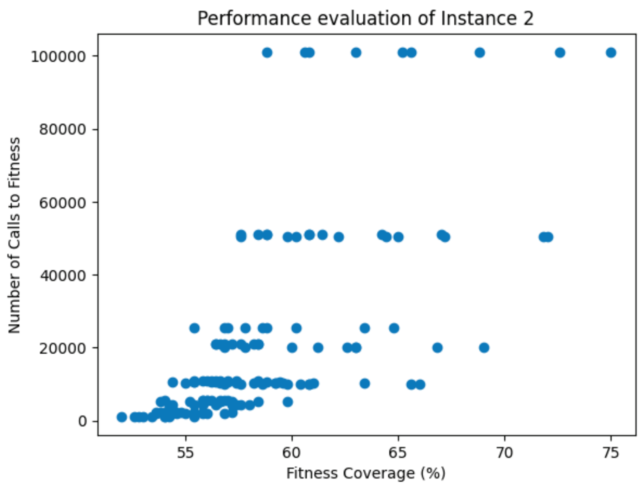
Simulation

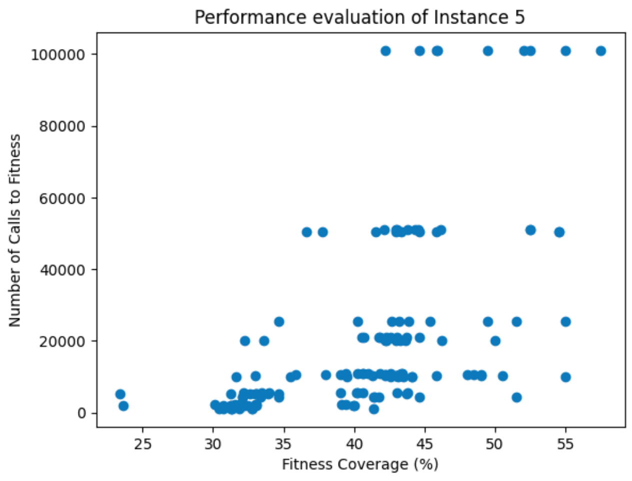
The simulation tests the possible configurations of parameters (**POPULATION_SIZE**, **MUTATION_RATE**, **GENERATIONS**, **ELITISM_PERCENTAGE**) to search for the optimal setup that yields the best solution for each problem.
Refer to the file output.md for the results printed during the execution of the algorithm (available on github.com).

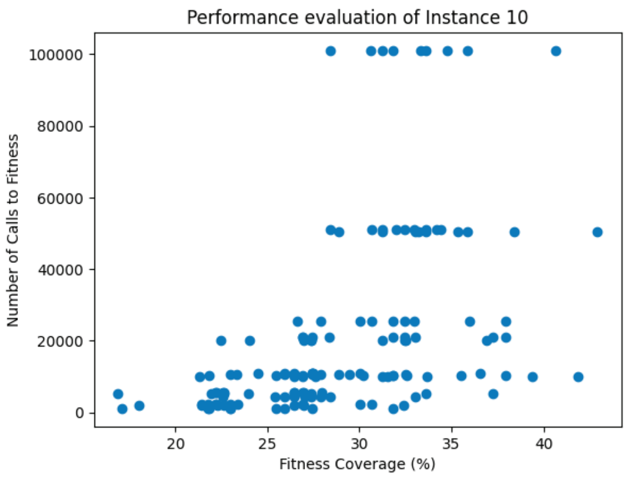
Graphical Performance Analysis

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.

REVIEW TO DARDANELLO LEONARDO

SUBMISSION: 10 DECEMBER 2023 at 8.28 PM

Hi there,
Thanks for sharing your code with us.
You implemented a Genetic Algorithm (GA) with Crossover, Selection, Mutation and an Extinction mechanism.
The main problem I see is related to the documentation.
In my opinion you should modify it by adding more comments and a way more detailed README in order to explain all the tests you made, how the code works, and your overall process. What led you to choose these parameters?
I like the idea of the extinction mechanism for bringing diversity to the population when there's no progress after a certain number of generations. However, it seems like the implementation is not effective due to a lack of significant improvements or none at all.
Since the problem was about fitness calls I think you should print the number of fitness calls for better insights.
Also, instead of printing coverage in each generation, maybe you should focus on summarizing different tests with various parameters to show trends.
Moreover if the coverage doesn't change after generation 11, as stated in the output provided, you should consider stopping further generations: continuing may increase total fitness calls without meaningful improvements.
Good luck for the next labs! :)

REVIEW TO IANNIELLI ANGELO

SUBMISSION: 10 DECEMBER 2023 at 11.26 PM

Hello there,
Thanks for sharing your code with the community.
While I appreciate your effort to use the 'islands' concept in your algorithm, I think you had some trouble understanding the exercise requirements.
First of all, the README doesn't explain the whole process you went through to write the code or why you chose certain parameters.
Also, I don't see any attempt to test different parameters, and there's no mention of it in the README, so I assume it wasn't done (but I might be wrong).
The output is too long and I found it hard to read.

Plus, I don’t see the point of plotting generations along with their fitness values. The main goal of the lab wasn’t just to maximize fitness coverage but to find a balance between the total fitness calls and fitness coverage. So, if you already have 99.0% fitness in generation 393, why continue for hundreads of generations with an extra 30k fitness calls for just a 1% improvement?
You also didn’t test different problem instances (1, 2, 5, 10) as the exercise required; you only tested instance 1.
That being said, I think you could have done a better job but it still is a solution!
Good luck for the next labs! :)

LAB 04 - TIC TAC TOE

README
SUBMISSION: 24 DECEMBER 2023 at 10.37 PM

Objectives

We assumed that X always moves first. This information can be useful as a key for interpretation and understanding of some experiments.
Starting from the professor's strategy we tried to improve it, testing how artificial intelligence behaved in different scenarios.

Training Phase (for both X and O)

Two different dictionaries were used, one for AI using X and the other one for the AI using O, as stated here:

```
# Creation of 2 dictionaries for storing X AI and O AI players training
# Accessible by value_dictionary["x"] and value_dictionary["o"]
value_dictionary = {"x": defaultdict(float), "o": defaultdict(float)}
```

During the training phase our AI plays against an opponent using a random strategy: it means the opponent will choose a random move from the available ones.
We made AI train for 1_000_000 games to be able to have an optimal dictionary capable of suggesting the best move in every situation, covering all possible scenarios that could happen. For a simple game like tic-tac-toe we can assume to have visited all possible states multiple times.
After a huge amount of tests conducted during the development we found that the best results were achieved using, during the training phase, **epsilon = 0.003** and as **Reward Policy**:

Player	Win	Lose	Draw
X	+1	-1	0
O	+3	-6	+2

AI using X (first to move)

After implementing the training phase, we wondered how good this player actually was.
On average, when tested in a series of 10_000 games against a player making random moves, performances were excellent:

- **Wins: 9908 (99.08%)**
- **Loses: 0 (0.00%)**
- **Draws: 92 (0.92%)**

But winning doesn't necessarily mean making the best moves. So, to be sure that it didn't make any mistakes and always chooses the best move, we tested move by move the various states of a lot of games.
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.

While it might appear obvious, it's not always the case. With a low number of games during training, there's the possibility that not all potential move combinations were explored, leading to the selection of non-optimal moves while still achieving victory. We encountered this issue with a training phase of 100_000 games, prompting us to increase the number to 1_000_000 for a more comprehensive exploration.

AI using O (second to move)

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 phase (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. So 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.
Initially the training phase (for O) was the same as for X. The 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 it play against AI using X with optimal moves (the one we just trained before!): 10_000 test games were conducted and they ended all with a lose. This experiment highlighted how O did not know the strategy to draw against a perfect X. This is because, in the 1,000,000 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.
That's why we decided to change its Reward Policy during the training phase. As stated above, we did a huge amount of tests and we chose the parameters that gave us the best statistics.
As a result, on average, when tested in a series of 10_000 games against a player making random moves, performances were pretty good:

- **Wins: 8779 (87.79%)**
- **Loses: 42 (0.42%)**
- **Draws: 1179 (11.79%)**

We believe that by changing the values assigned by the reward function, it is possible to achieve a perfect gameplay.

AI using X against AI using O

While we already knew AI using X was making best moves, fights against the two AIs confirmed that the Reward Policy used for AI playing with O during the training phase is very effective. When AIs were tested in a series of 10_000 games against each other the outcome was as follows:

- AI usign X Wins: 0 (0.00%)
- AI usign O Wins: 0 (0.00%)
- Draws: 10000 (100.00%)

It's important to note that this result doesn't have a random component; it is consistent and seems to remains the same with each execution.

Do you want to challenge the AIs?

After finishing our experiments, we thought it could be fun having the possibility to challenge 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!

REVIEW TO MATASSA PAOLA

SUBMISSION: 6 JANUARY 2024 at 8.59 PM

Hello Paola, I hope everything is okay.
First of all, thanks for sharing your implementation of Q-learning for playing TicTacToe!
The Q-learning implementation in the MyPlayer class is a good example of reinforcement learning. It follows the standard Q-learning update formula and uses an exploration-exploitation trade-off with an epsilon-greedy strategy.
Overall, the code looks great to me (even if I'd suggest moving the training process of the MyPlayer class into a separate method for modularity improvements) and is well-commented. However, in my opinion, the README could be more exhaustive.
Also, in your README, you state that: "Initially I trained MyPlayer against RandomPlayer, achieving a win rate of approximately 60%, then I used the RandomPlayerTwo for the training, enabling me to elevate my victories to 80% thanks to a faster convergence." but testing your algorithm incrementing the number of games played in the training phase (from 20_000 to at least 100_000), even if trained against RandomPlayer, leads to more consistent and less random results across multiple runs and is better for achieving a performance boost (reaching easily 80-90% of wins).
Another thing you could have tried is training two different agents capable of playing, one with X (as first) and one with O (as second), with different dictionaries and different custom reward policies for each agent instead of training it playing one time as first and one time as second.
Maybe also developing an interface for playing against the AI you trained would have been great!
Good luck for the exam!

REVIEW TO MASETTA DIMITRI
SUBMISSION: 6 JANUARY 2024 at 11.17 PM

Hi Dimitri,
First of all, thanks for sharing your work with the community.
Your implementation of the Q-learning algorithm for playing TicTacToe uses the MAGIC board (as I also did). It is pretty clean and easily understandable, even if, in my opinion, there are too few comments. Additionally, it is missing one of the most important sections to understand the whole process you've been through: the README!
Talking about the code, I would have appreciated the creation of a function for the training phase to improve modularity, instead of putting the code as it comes in the algorithm.
Another thing I noticed and I have to point out is that the AI doesn't play with "O", neither it is trained for doing it.
Plus, I tested your code, and as the graphic you provided states, epochs = 1_000_000 are too many since epochs = 500_000 would have given you the exact same results in half of the time needed to compute the solution.
In addition, some comments like "Evaluate state: +1 first player wins" actually do not reflect what the code does. Maybe you tried different reward policies and just didn't remember to change comments?
But if you did, why is there no explanation behind the use of +10/-100/0 as rewards, instead, for example, simply +1/-1/0? Results don't seem to change too much.
Also, I think it would have been great to see an interface for playing against the trained AI.
By the way, overall, I think you did a pretty good job with this lab, gaining nice results for the AI playing with X.
I'm looking forward to seeing your solution for the QUIXO game! Good luck!

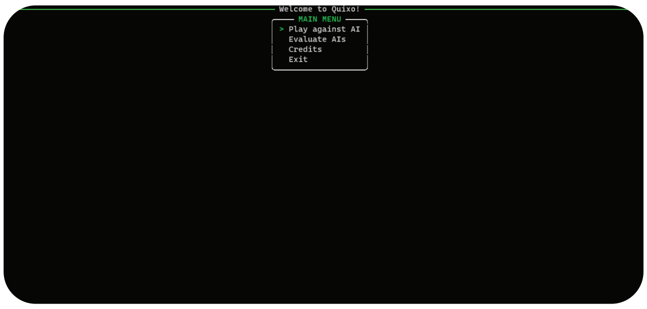
QUIXO

IF YOU'RE CURRENTLY USING A BRIGHT-THEMED TERMINAL, SWITCHING TO A DARK THEME MAY ENHANCE YOUR USER EXPERIENCE.

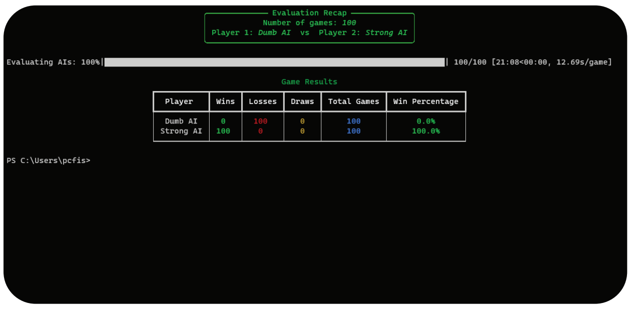
README
SUBMISSION: 19 FEBRUARY 2024 at 2.00 AM

Introduction & relevant aspects

Assuming we are all familiar with this turbocharged version of Tic-Tac-Toe, let's not spend time on explanations and dive straight into the discussion.
Let's start by clarifying what I mean for Draw because in the actual Quixo game, only two outcomes are available: Win or Lose. A Draw is determined when the total number of moves in the game exceeds 60 or a repeated pattern (the last sequence of 10 moves) is found in the game's move history. This has been done in an optic to avoid stales and consequently an infinite loop when making certain AIs playing against each other.
I have attempted to keep many of the original functions supplied intact to facilitate faster code comprehension: I only did some refactoring to the code provided and changed something in the play function mainly to address the Draw state checks and to make possible the interactive interface to work properly without making unecessary wrappers.
A custom menu was developed, granting users the ability to play against the AIs (also displaying the move history), and to evaluate any of them against a Random player or against each other as well.



```
Welcome to Quixo!
  MAIN MENU
> Play against AI
  Evaluate AIs
  Credits
  Exit
```



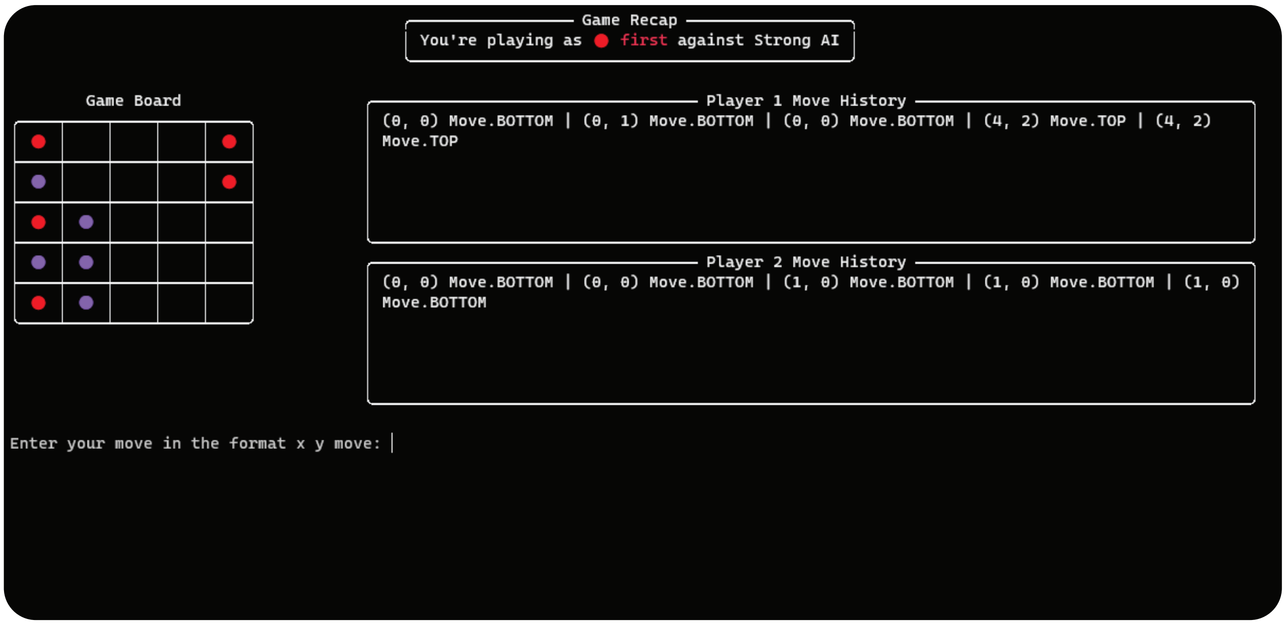
```
                 Evaluation Recap
              Number of games: 100
      Player 1: Dumb AI  vs  Player 2: Strong AI

Evaluating AIs: 100%|████████████████████████████| 100/100 [21:08<00:00, 12.69s/game]

                   Game Results
```

Player	Wins	Losses	Draws	Total Games	Win Percentage
Dumb AI	0	100	0	100	0.0%
Strong AI	100	0	0	100	100.0%

```
PS C:\Users\pcfis>
```



```
Game Recap
You're playing as ● first against Strong AI

Game Board

Player 1 Move History
(0, 0) Move.BOTTOM | (0, 1) Move.BOTTOM | (0, 0) Move.BOTTOM | (4, 2) Move.TOP | (4, 2)
Move.TOP

Player 2 Move History
(0, 0) Move.BOTTOM | (0, 0) Move.BOTTOM | (1, 0) Move.BOTTOM | (1, 0) Move.BOTTOM | (1, 0)
Move.BOTTOM

Enter your move in the format x y move:
```

What strategy has been used?

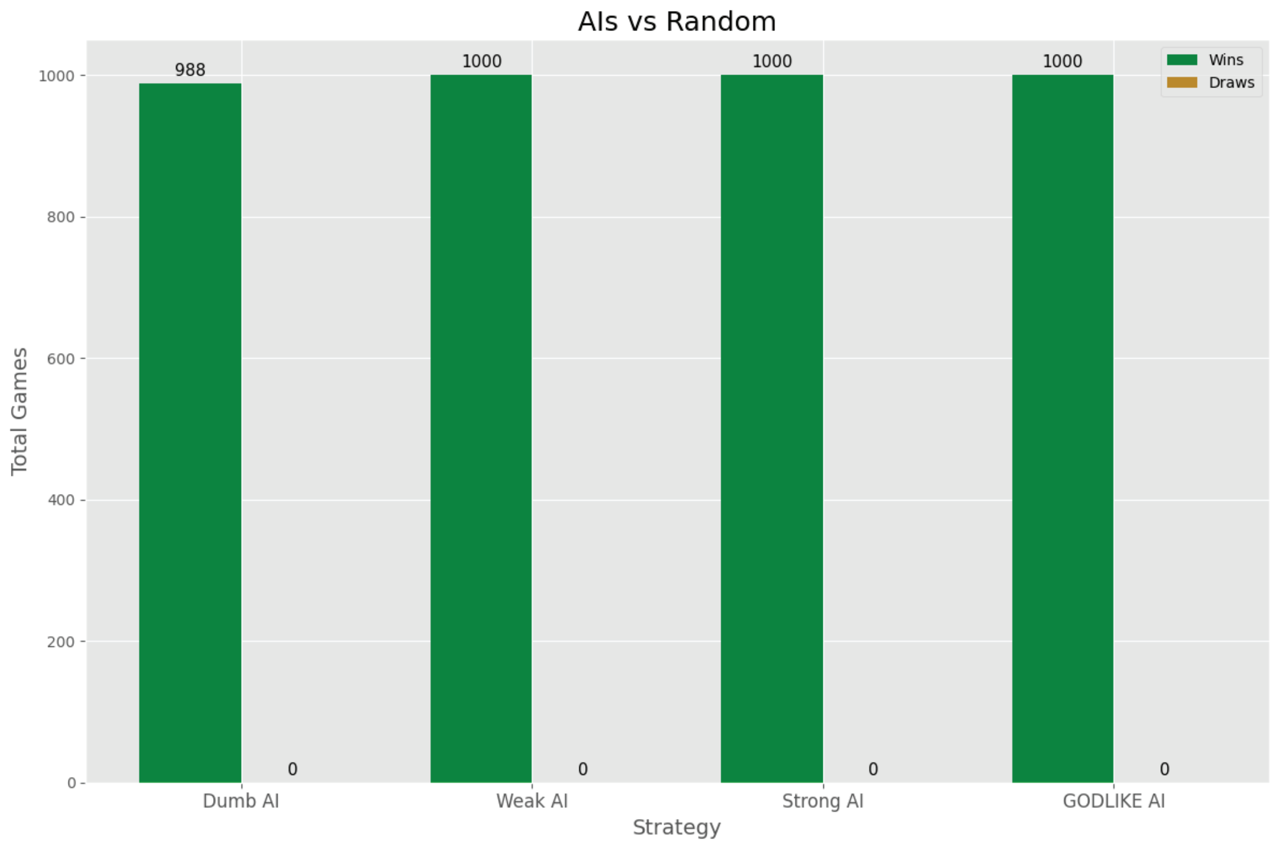
Let's now talk about the strategy.
Quixo has a well-defined state space (all board configurations) and action space (all possible moves).
Since there it a consistent number of possible states the use of RL would be challenging, expecially with limited computational resources available for training.
For this reason, I thought the most suitable option was to implement a **recursive Minimax algorithm** in order to minimize the possible loss, always considering worst-case scenarios.
Since, as I said before, there a lot of states and therefore it seems like not possible to explore them all, Minimax comes along with a **Depth-Controlled Search** and a **Custom Evaluation Function**, making it able to evaluate every state assigning a value (more specifically for every state in which I find a win/lose, I assign ±infinite and for all the other states I return an evaluation based on the number of AIs' pieces minus the number of the opponent's pieces). The logic behind it is that the more pieces you have on the board, the more likely you are to win. Of course, this is under the constraint that the AI can foresee many moves ahead and determine if the opponent has the possibility to win.
I developed four different Agents, everyone able to explore the tree with a different **maximum depth limit**:

- **Dumb AI** (max_depth = 1)
- **Weak AI** (max_depth = 2)
- **Strong AI** (max_depth = 3)
- **GODLIKE AI** (max_depth = 4)

Minimax is also paired with **Alpha-Beta Pruning**, as it significantly reduces the execution time of the algorithm.
Dumb and Weak AI exhibit imperceptible response times, while Strong AI could take up to one second to perform a move, and GODLIKE AI might require approximately 5 to 10 seconds.

What are the results?

Testing AIs against a Random Player gave amazing results: (and also some joy...)



When the AIs were tested against each other, it became evident that the Dumb AI could be easily defeated. However, in tests involving the Weak AI or higher levels, the game was more likely to end in a draw.

THE END (or the beginning...)

Every line of code has a moral and ethical implication.

Grady Booch

THANKS.

Barbato Luca s320213