Final Report – Buccellato Federico s309075

Laboratory 1: Set cover problem using A* algorithm

In my A* algorithm implementation, I utilize a single heuristic function: *len(new_state.taken)*

This heuristic function calculates the number of states that have been taken, providing an assessment of the distance between the initial position and the current position.

To complete the evaluation function f(n), I combine this heuristic with the distance from the current state to the goal state: *len(new_state.taken) + distance(new_state)*

This combined sum serves as the evaluation function f(n), guiding the A* algorithm in its exploration for the optimal path.

Halloween Challenge

For the Halloween challenge, I implemented an algorithm for set cover optimization.

Tweak Function:

This function performs a mutation operation on the current state. It randomly selects a set and toggles its inclusion in the state. Acceptance or rejection of the new state is based on whether its fitness is better than the current state.

The algorithm stops when it finds a solution that covers all points or reaches the maximum number of iterations.

The high lambda value aims to explore and tweak for better solutions, stopping early if a valid solution is found. This approach is intended to yield good results quickly.

Result:

```
num_points: 100, density: 0.3, taken sets: 12
num_points: 100, density: 0.7, taken sets: 5
num_points: 1000, density: 0.3, taken sets: 19
num_points: 1000, density: 0.7, taken sets: 6
num_points: 5000, density: 0.3, taken sets: 21
num_points: 5000, density: 0.7, taken sets: 7
```

Laboratory 2: Nim game using EA

To address the Nim game challenge, I initially opted for an approach based on a genetic algorithm. This algorithm generated new offsprings list by mutating the current game state. The goal was to create an "adaptive" player capable of selecting for each turn, among various possible generated children, the one with the highest fitness. However, this approach showed some limitations during peer reviews.

Peer reviews received:

As professor told us today during lessons i'll try to be as much concise as I can.

- The code is well organized, but some comments would have been appreciated. In the fitness function the type hinting is wrong.
- The proposed strategy likely only works for a Nim game of 5 rows (because of the hard-coded values), and does not provide a solution for the k version.
- In order to make the final result even more accurate, your strategy shouldn't always be the first moving.
- The proposed solution it's not mutating a starting solution, but it's actually generating lambda random solutions.
- You should try to create a starting solution, mutate it in some way, apply a recombination operation(optional step) and then choosing the best one.

Hi Federico!
First of all well done for the lab 2 about Nim, i really appreciate your code style, it was very clear to read and comprehend.
I find very good your definition of an agent by defining a class for it.
However i have some advice to improve your code:

- I suggest to add more comments to your code to explain in more details your implementations, also with a short description in the README file
- About the expert agent my advice is to enhance your current optimal strategy considering the game situation end up in a position with only one row of size 2 or more and at this point the nim sum is not equal to zero so the best move is to reduce this to a size of 0 or 1 and leaving an odd number of rows with size 1, from which all the moves are constrained. A possible implementation could be this one:
- For the fitness function i would suggest to use also as measure the number of match won against the expert agent, maybe combining that to your current fitness implementation giving a different weight to each one and try different weights combination to find the best balance
- You can also try to use recombination instead of only mutation, combining them and use a parameter to choose each step which one to use
- As final advice, before using the adaptive strategy against the expert agent ,you can make a test to try different parameter combination in order to identify the best ones

I hope that my suggestion will help you and good luck for the next labs!

Positive Aspects

Modularity and Readability:

- Use of well-defined functions and classes improves code organization.
- The Individual class encapsulates relevant information

Evolutionary Strategy Implementation:

- The implementation of the (1,λ) Evolutionary Strategy is clear and follows a standard structure.

Overall the lab seems well implemented 😄

Negative Aspects

Evaluation:

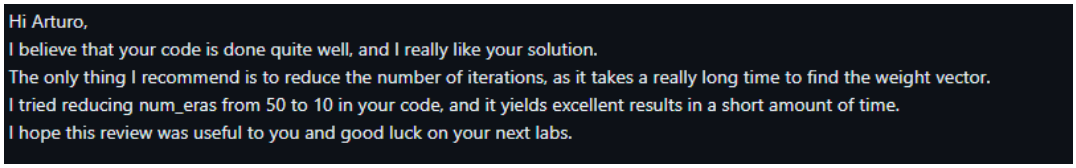
- The agent using adaptive strategies is always player 0, giving it and advantage. Starting player could be random for a more fair evaluation.

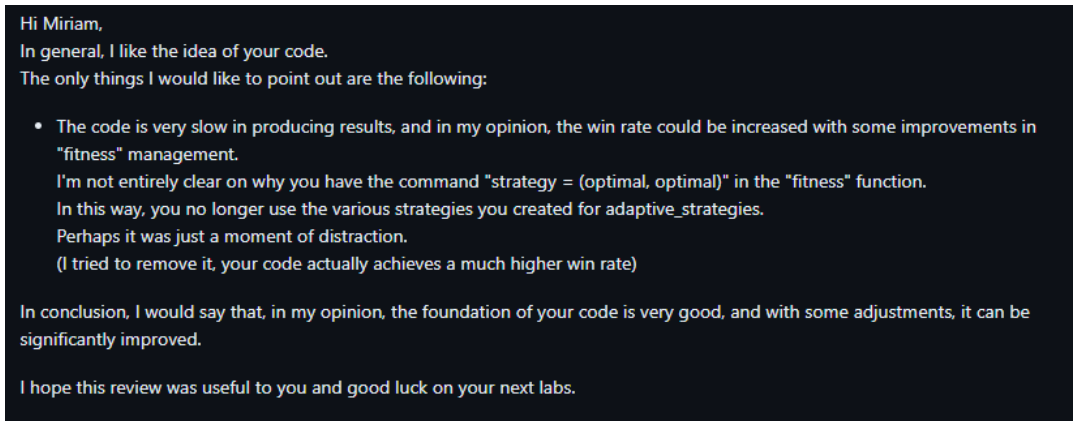
Subsequently, in an attempt to refine the gameplay strategy, I further developed the code into a second file called "lab_2_after review." In this new implementation, I introduced various functions representing different strategies to tackle the Nim game. Later, I designed the "Individual" class, which contained a vector of weights, each associated with a specific strategy.

Through a training process spanning 200 generations, the genetic algorithm aimed to identify the optimal weight vector that could maximize the overall fitness. The choice of moves was made in a weighted manner, proportional to the weights associated with different strategies.

Finally, after identifying the weight vector that maximized fitness, I allowed the individual to play again, considering it as a trained player. This approach, combining evolved strategies and optimized weights, aimed to achieve a higher level of performance in the context of the Nim game.

Peer reviews done by me:


Hi Arturo,
I believe that your code is done quite well, and I really like your solution.
The only thing I recommend is to reduce the number of iterations, as it takes a really long time to find the weight vector.
I tried reducing num_eras from 50 to 10 in your code, and it yields excellent results in a short amount of time.
I hope this review was useful to you and good luck on your next labs.


Hi Miriam,
In general, I like the idea of your code.
The only things I would like to point out are the following:

- The code is very slow in producing results, and in my opinion, the win rate could be increased with some improvements in "fitness" management.
  I'm not entirely clear on why you have the command "strategy = (optimal, optimal)" in the "fitness" function.
  In this way, you no longer use the various strategies you created for adaptive_strategies.
  Perhaps it was just a moment of distraction.
  (I tried to remove it, your code actually achieves a much higher win rate)

In conclusion, I would say that, in my opinion, the foundation of your code is very good, and with some adjustments, it can be significantly improved.

I hope this review was useful to you and good luck on your next labs.

Laboratory 9: Problem Instances 1, 2, 5, 10 on a 1000-Loci Genome using E.A.

To address this problem, two different approaches were explored. The first involved employing a genetic algorithm, while the second utilized the island model.

Genetic Algorithm

The first solution involves employing an evolutionary algorithm.

When selecting a parent, a choice is made from a pool of random individuals in the population, favouring the one with the highest fitness.

In the process of mutating an individual, a mutation occurs by flipping one of the bits in the individual's genotype.

Moving on to the One-Cut Crossover operation, offspring is generated through a crossover between two parents. This involves selecting a random cut point and combining parts of the parents' genotypes.

Island Model

The second approach incorporates the island model.

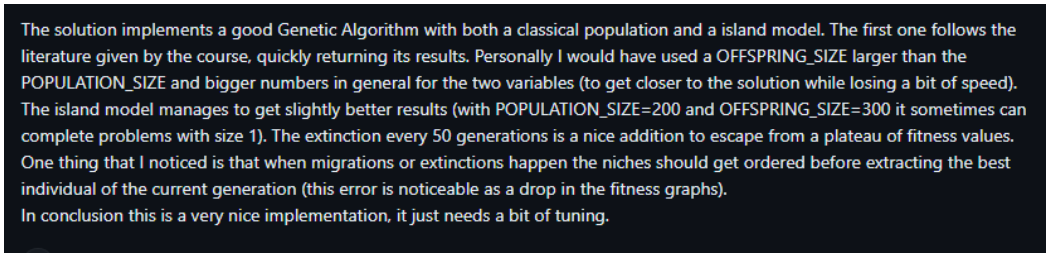
To begin, the code establishes multiple niches (subpopulations) within the initial population by rearranging and dividing individuals.

Subsequently, for each niche, new offspring are generated through mutation or uniform crossover, based on a predefined probability.

The implementation of the island model involves evolving each niche independently for a specified number of generations. After a set number of generations, individuals undergo swapping between random niches in a process referred to as migration.

In cases where the difference between the best fitness in the current niches and the previous best fitness falls below a certain threshold, the population undergoes extinction and recreation. This process entails preserving a certain percentage of the best individuals as survivors and generating new individuals to fill the remaining population.

Peer reviews received:



Hi Federico, I took a glance at your code. It's well written, comment style helps in understanding exhaustively what you're doing at each step. I also used the Island Model with migration among all the islands, but I also found it interesting being cautious about performing the migration only between two islands. The only point I want to make is to experiment with different values for the parameters since it's possible to notice a boost in the performance, but seriously, that's the only thing I can point out.
Good job! Keep going this way and good luck with the next deliveries :)

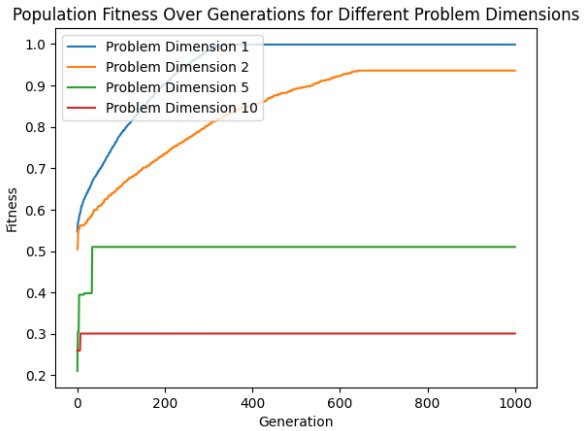
The code is well-written, and the comments in it really help with understanding. I like that you chose to implement not just an evolutionary algorithm but also an island model, using some techniques we've learned about recently, like extinction and migrants.

A suggestion I have is to try using slightly different settings for your algorithms. In my opinion, using 2000 iterations with a population size of 10 and offspring size of 60 seems a bit unbalanced. You can get a much higher max_fitness by increasing your population and offspring while reducing the number of iterations to avoid too many fitness calls.
For example, with 1500 iterations and a population size of 200, offspring size of 300, you can achieve 100% max_fitness for PROBLEM_SIZE=1 in both models you used and get close to 100% for PROBLEM_SIZE=2. In any case, performance improves considerably for all PROBLEM_SIZE, with a more extensive selection among individuals in your population, resulting in more widespread mutations and crossovers.

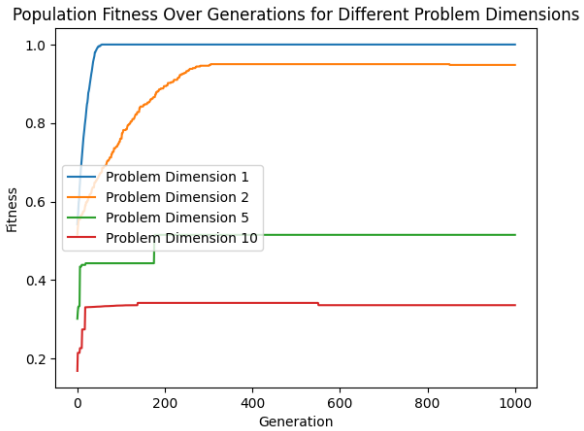
Hi Luca, i have some advice for you. I hope that this review can help you.

- I think that is very important to write a README. It could include some clarifications about:
  - the main choses that you made in your work (e.g what algorithm you implemented)
  - some motivation about the hyperparameters that you chose (e.g mutation probability, offspring size..)
  - everything that is relevant to well understanding your code
- I immediately noticed the of_over. I think this is a form of "cheating" because you shouldn't know that fitness increases if the number of 1's in the genome increases. So, don't use it to evaluate the performance of your algorithm.
- I see that your results are obtained using a one_cut_crossover. You can try using a two_cut _crossover to increase diversity during recombination and you may get slightly better results with just this small variation. I'll leave you my implementation for the 2-cut:
- For improving you results I can advice you to try different mutation operators such as scramble_mutation, random_resetting, reverse_mutation, gaussian_mutation, swap_mutation.
  Recommended reading: https://www.geeksforgeeks.org/mutation-algorithms-for-string-manipulation-ga/
  Some other comments:
- I really liked your implementation of uniform crossover

Following the reviews I received, I decided to modify my code and create a new version located in the file "lab_9_after_review," where I attempt to address the key points of my colleagues' feedback. By doing so, I have improved my lab results to some extent.



Evolutionary algorithms



Island model

Peer reviews done by me:

Hello,

I've noticed some areas where the code could be further improved. In particular, I would like to suggest an optimization in handling population updates. Currently, extending the population in the following way, population = population[0:ELITE_SIZE] + offspring, leads to a significant decrease in the population size, dropping from 500 to 174 individuals.

Another area of improvement could be adding comments within the code itself. Adding comments in more complex sections or providing a brief description of each function can enhance code readability and facilitate understanding, especially for those reading it for the first time.

Despite these suggestions, it's important to emphasize that the evolutionary algorithm is well-structured and comprehensible. Keep up the good work!

Hello,

I have noticed that your solution faithfully follows the literature provided by the course. I find your code to be extremely well-organized and readable, and the comments significantly contribute to the understanding of the code. The evolutionary algorithm is structured effectively and appears correct. I particularly appreciated how you handle the possible termination in case of reaching the maximum fitness or constant fitness across generations.

Overall, your work is of high quality. The only suggestion I can give is to experiment with some different algorithms to assess how fitness behaves in alternative contexts.

Nevertheless, it is a job very well done!

Laboratory 10: Tic Tac Toe with reinforcement learning

In this code, I'm implementing a Tic-Tac-Toe game using a combination of the Minimax algorithm and Q-learning.

The minmax function incorporates the Minimax algorithm. Through recursion, it explores all possible states and actions, opting for the action that maximizes (for the maximizing player) or minimizes (for the minimizing player) the value of the game outcome.

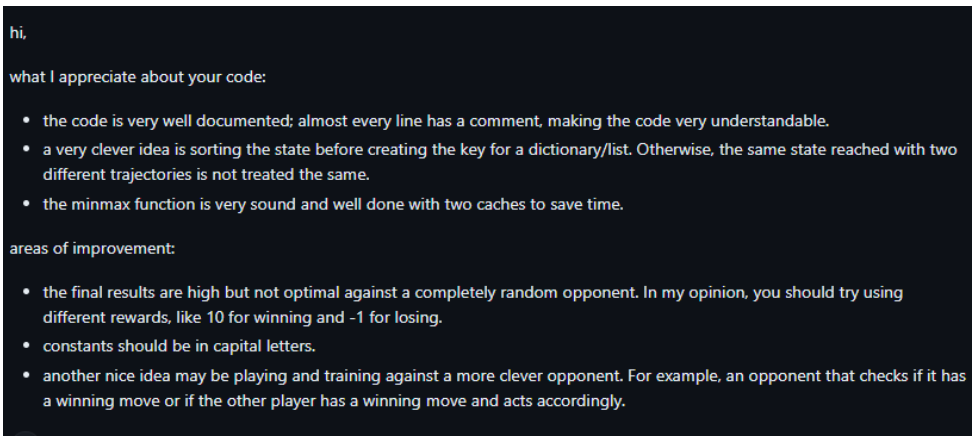
I've opted to train the Q-learning model by having it play numerous games against both a random player and a Minimax player. The Q-values dictionary is then updated using the backpropagation function. This function positively or negatively updates the various values associated with the states based on whether the sequence of moves led to a win or a loss.

The choose_action function is employed to determine the best action for a given state. It selects the optimal action based on the Q-values. If no Q-values are available for the current state, it randomly chooses an action. Otherwise, it selects the action with the maximum Q-value.

Result:

```
Percentage of wins for the Q-learning agent: 81.17%
Percentage of wins for the opponent: 2.03%
Percentage of draws: 16.80%
```

Peer reviews received:



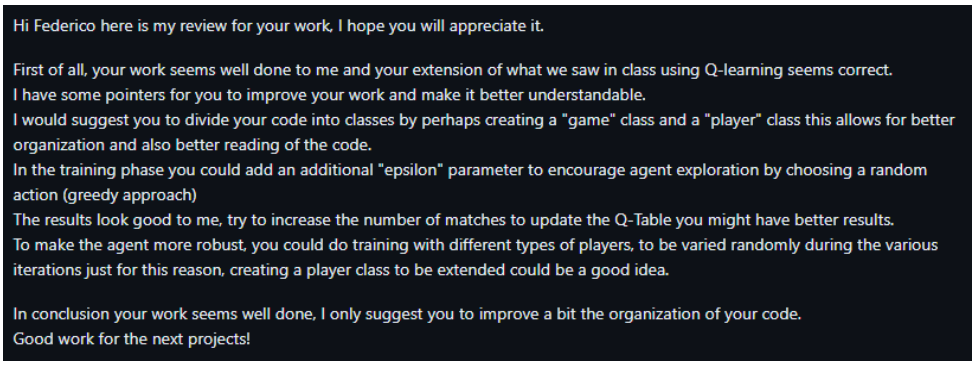
hi,

what I appreciate about your code:

- the code is very well documented; almost every line has a comment, making the code very understandable.
- a very clever idea is sorting the state before creating the key for a dictionary/list. Otherwise, the same state reached with two different trajectories is not treated the same.
- the minmax function is very sound and well done with two caches to save time.

areas of improvement:

- the final results are high but not optimal against a completely random opponent. In my opinion, you should try using different rewards, like 10 for winning and -1 for losing.
- constants should be in capital letters.
- another nice idea may be playing and training against a more clever opponent. For example, an opponent that checks if it has a winning move or if the other player has a winning move and acts accordingly.



Hi Federico here is my review for your work, I hope you will appreciate it.

First of all, your work seems well done to me and your extension of what we saw in class using Q-learning seems correct.
I have some pointers for you to improve your work and make it better understandable.
I would suggest you to divide your code into classes by perhaps creating a "game" class and a "player" class this allows for better organization and also better reading of the code.
In the training phase you could add an additional "epsilon" parameter to encourage agent exploration by choosing a random action (greedy approach)
The results look good to me, try to increase the number of matches to update the Q-Table you might have better results.
To make the agent more robust, you could do training with different types of players, to be varied randomly during the various iterations just for this reason, creating a player class to be extended could be a good idea.

In conclusion your work seems well done, I only suggest you to improve a bit the organization of your code.
Good work for the next projects!

Review added by Hossein Khodadadi (313884) and Abolfazl Javidian (314441).

The writer has used the min-max strategy to update the q_values recursively. A back-propagation function is defined to optimize the action value dictionary based on the gamma and learning rate. it's commonly used in games where two agents are in direct opposition to each other, each trying to maximize their own rewards while minimizing the opponent's rewards, which is implemented in the min-max function. The method has converged to 81.17 percent of win by 20,000 iterations, which is computationally efficient. However, there are some vague ideas in these functions:

1. In the min-max function, It seems that all the values stored in the scores list are always -1 or 1, so how it dynamically help to find the best action and the corresponding score.

2. In the q_learning function, the next_state variable is defined but apparently, it is never used.

If the procedure of learning the q_values is not too much time-consuming, a greed search on different hyper-parameters of alpha and gamma could be performed to achieve higher win rates.

Peer reviews done by me:

Hi,

Your code is very neat and readable. However, I would suggest adding more detailed comments and a README to explain your choices and make your work clearer.

Regarding the code, I think it would make more sense to train it using a method other than just randomness so that the model learns more accurate combinations.

Additionally, I noticed a potential error in your code during the training phase, in this segment:

```
for steps in tqdm(range(10000)):
trajectory = random_game()
for i in range(len(trajectory) - 1):
state = trajectory[i]
next_state = trajectory[i + 1]
action = agent.choose_action(state, valid_actions=list(set(range(1, 9 + 1)) - (state.x.union(state.o))))
final_reward = state_value(next_state)
agent.update_q_value(state, action, final_reward, next_state)
```

Using the main moves from trajectory may imply that sometimes actions already present in the current state are analyzed, thereby inserting elements like:

```
(frozenset({4, 5, 6}), frozenset({8, 3}), 4)
(frozenset({4, 5, 6}), frozenset({8, 3}), 8)
etc...
```

Aside from this, it's still good code both in terms of solution and writing. With a few small improvements, you will be able to address these minor issues.

Hi,

Your code is very well-organized, and I believe the comments greatly assist in understanding it. However, I would recommend creating a README as well, where you can explain in detail what you used to provide a more comprehensive overview of your overall work.

Apart from this, I noticed the presence of some errors in your code that make the statistics reported at the end inaccurate.

Firstly, the dictionary lacks the presence of valid keys. This is because when you create self.current_state = env.make_move(action) and subsequently pass it to self.update_state_value(self.current_state, reward), you didn't realize that make_move doesn't have a return value and will always return None. This way, your values dictionary will only be a set of many (None: float) pairs.

Another significant error to note is the incorrect functioning of env.is_finished(). In fact, it often ends the game after one or two moves, so it is impossible for the game to be finished so quickly.

The rest of the code seems logically correct, and the concept of Q-learning appears to be clear. I recommend correcting these errors and then reassessing the actual statistics of the model.

Exam project: Quixo game

For this Quixo game project, I've decided to develop a player based on the MinMax algorithm. This algorithm is used to optimize the decision-making process for the player, aiming to maximize the player's chances of winning.

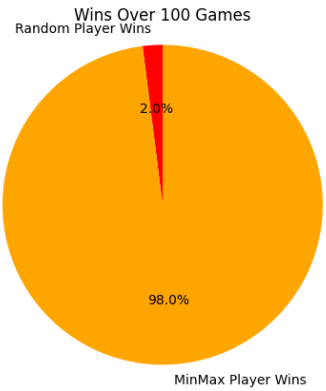
The Alpha-Beta pruning technique is also implemented to enhance the efficiency of the MinMax algorithm. It works by eliminating (or "pruning") branches in the decision tree that are not likely to influence the final decision. This allows the algorithm to focus on the most promising options, speeding up the selection of the next move.

The player is set up to evaluate the game state from its perspective, aiming to maximize its own score while minimizing the opponent's score. This involves calculating scores based on the number of consecutive pieces a player has in rows, columns, and diagonals. The goal is to identify the maximum count of consecutive pieces in these directions.

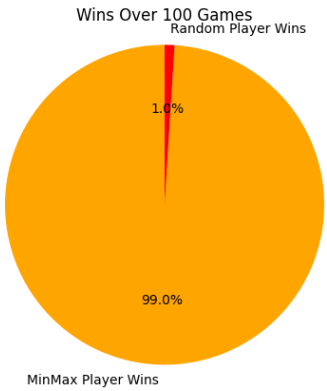
The decision-making process involves systematically exploring possible moves, creating child states for each move, and using the MinMax algorithm to evaluate the potential outcomes of these moves. The aim is to identify the most advantageous move for the current player.

Before initiating the MinMax algorithm, for each move that I examine, I first check if it leads to a victory. If such a move exists, I choose that move as it is one of the best moves. This approach speeds up the program by avoiding the need to calculate in depth all possible scenarios for each move. Essentially, it prioritizes winning moves and eliminates unnecessary computations, thereby enhancing the efficiency of the program.

Finally, the performance of this MinMax player is tested by simulating games against a player that makes moves randomly. The results of these games provide valuable insights into the effectiveness of the MinMax strategy.



Random player starts



MinMax player starts