CT421 Artificial Intelligence
Project 2: Iterated Prisoners Dilemma

Evan Murphy | Student ID: 21306306

Code available at: https://github.com/xEvanM/ct421-ai-ga-ipd

Part 1: Evolution Against Fixed Strategies

1.1 What is a genetic algorithm?

A genetic algorithm is an algorithm that evolves, based on the foundations of natural selection and genetics. In Computer Science, genetic algorithms are often used for search and optimisation type problems, yielding high quality solutions.

A genetic algorithm begins with an initial population, and the process of natural selection throughout generations allows species to 'evolve', simulating "survival of the fittest" with the idea that the final generation will provide your optimal solution.[1]

1.2 Introduction to the Iterated Prisoners Dilemma

The prisoners dilemma is a fundamental experiment of game theory. Two rational agents can choose to cooperate with each other, or defect.[2]

In simple terms, you can imagine two individuals arrested and held in separate interrogation rooms, each with a choice to cooperate with the officers or defect; if both cooperate, they each receive a reduced sentence, but if one cooperates while the other defects, the defector goes free while the cooperator gets the harshest penalty, and if both defect, they both receive a sentence that is worse than mutual cooperation yet not as severe as the penalty for being the sole cooperator, as shown in the payoff matrix below.

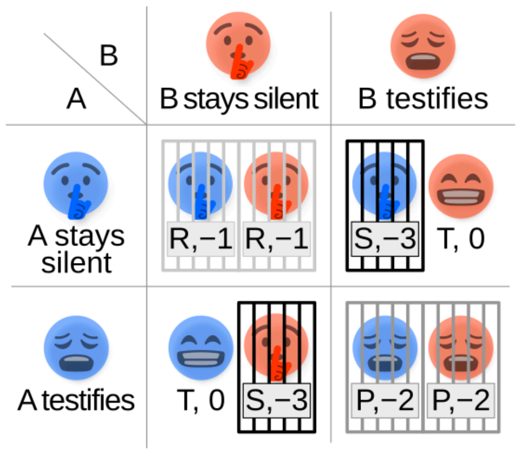


Fig 1: Payoff matrix [3]

The iterated prisoners dilemma is an extension of this, in which the dilemma is played out multiple times. This allows players to choose strategies that reward cooperation, or punish defection over time. This provides an interesting experiment for this project, where we evaluate a genetic algorithm's performance against a number of known, fixed strategies.

1.3 Code Outline

1.3.1 Strategies

Each strategy is implemented as a Python class, with a function 'make_move' that dictates how the strategy makes it moves. Here are the strategies I have implemented as opponents to the strategy evolved by my genetic algorithm. "C" and "D" refer to "cooperate" and "defect" respectively.

- AlwaysCooperate: This strategy cooperates without any conditions.
- AlwaysDefect: This strategy defects without any conditions.
- Alternator: Alternates between "C" and "D" by tracking it's own last move.
- TitForTat: Begins with 'C" and copies the opponents last move.
- GrimTrigger: Cooperates until opponent defects, then defects until the end of the game.
- SuspiciousTitForTat: Begins with "D" and copies the opponents last move.
- Pavlov: Starts by cooperating, then repeats the previous move if the opponent matched it's move, otherwise it switches based on the previous outcome.

Each strategy, or *genome* that is generated by the genetic algorithm, using memory length of 1 for clarity, is represented as some combination of C and D such as: ['C', 'D', 'C', 'D', 'D']. This represents *(Initial Move), (D, D), (D, C), (C, D), and (C, C)*, where '(D, D)' indicates that the player and the opponent both defected, for example, as I am keeping a history of both player and opponent moves.

1.3.2 Prisoner's Dilemma Functions

There are a number of functions that simulate the iterated prisoners dilemma. These involve keeping a move history, making decisions, and calculating the payoff based on a payoff matrix.

- Play_ipd: This simulated a match between two strategies for N rounds, updating the scores and maintaining a move history based on the memory_length.
- Decide_move: This picks a move based on the strategy of the player. It supports both strategy class instances and genome-based strategies evolved by the genetic algorithm by adjusting the game history to the players perspective (flipping the moves), and then looking up the move.
- Strategy_lookup: This is an extension of decide_move used for genome-based decision making. Recent rounds in memory are converted to a binary string that provides an index for our genome-based strategy to make decisions.

- Payoff: This calculates the scores for each player when called in each round, the scores as defined in the fixed payoff matrix.

An interesting point is how the genome size is calculated as '*genome_size = 2 ** (2 * memory_length)* + *1*', which covers all possible combinations of both players' moves over the memory period, with the extra 1 added for a default or offset value.

1.3.3 Genetic Algorithm Functions

These functions are used to evolve strategies over generations. They involve elements such as initialising a population, evaluating fitness, as well as crossover, mutation, and selection.

Crossover involves 'merging' genetic material from two parents into a single offspring. Mutation iterates through the genes in a genome and has a chance of flipping these genes from "C" to "D", or "D" to "C", which maintains diversity. Tournament selection is used to select parents, which multiple genomes are selected at random and only the fittest is preserved, maintaining diversity whilst keeping the strongest candidates.

My fitness function normalises scores to a range 1 to 5 which is easier to interpret, as 1 represents the poorest outcome whilst 5 represents the optimal. This made sense to me as it related directly to the payoff matrix used in the IPD. It also keeps scores standardised for experiments with a varying number of rounds which makes parameter tuning easier as we can see how changing parameter X impacts the fitness, without needing to make additional considerations.

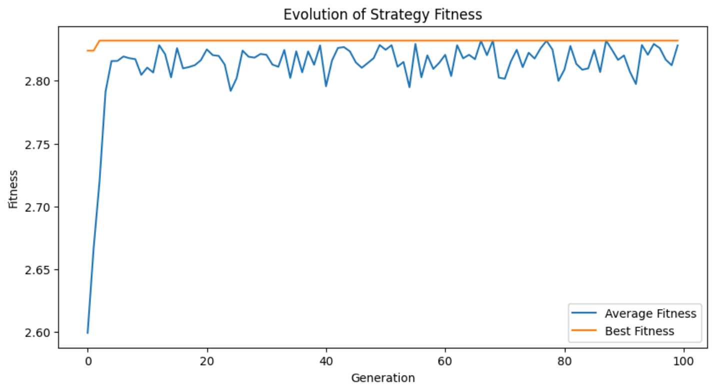
The genetic algorithm function orchestrates the aforementioned functions, and tracks the best and average fitness values throughout N generations so that they can be graphed and analysed as seen in section 1.4.

1.4 Experiments and Results

I ran a number of experiments to deepen my understanding of genetic algorithms themselves, as well as the iterated prisoners dilemma. Unless otherwise specified, these experiments were ran with a population of 50 over 100 generations. There were a lot of possible interesting experiments to run, so I've only highlighted a few here.

1.4.1 Experiment 1: GA v All Opponents (Memory = 1)

My first experiment involved running my genetic algorithm against all opponents with a mutation rate of 0.4 and a crossover rate of 0.6. These rates were kept relatively high to maintain a level of diversity in a small space. I first ran the experiment with a memory_length of 1.

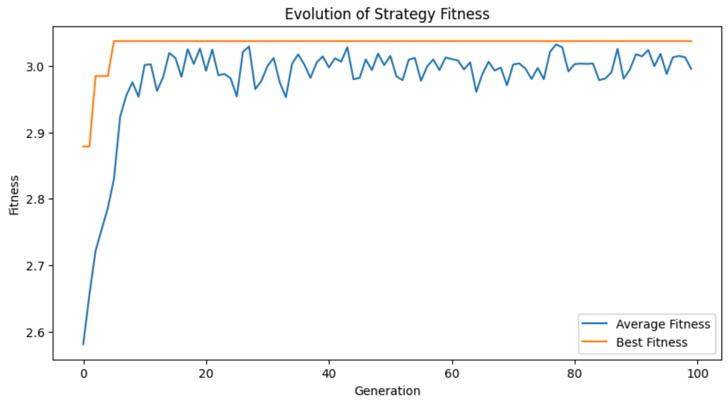


This resulted in the best strategy being found in the third generation, and a best evolved strategy genome of ['C', 'D', 'C', 'D', 'D']. The optimal fitness found was 2.832.

The best fitness score of 2.832 against a set of diverse opponents indicates that the evolved strategy performs reasonably well against cooperative, defecting, and reactive behaving opponents. With a short memory_length of 1, there are only 32 possible genomes, so high mutation (0.4) and crossover (0.6) rates help explore this small search space quickly, leading to convergence by the third generation.

1.4.2 Experiment 2: GA v All Opponents (Memory = 2)

This experiment involved keeping all other aspects the same, but increasing the memory length from '1' to '2'. This represents a growth in possible genomes from 32 to 131,027.

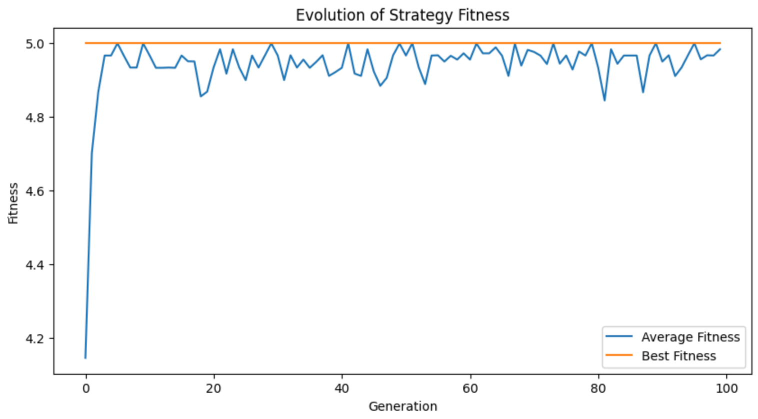


In this instance, we achieve a higher fitness of 3.038 after 6 generations, compared to the fitness of 2.832 with memory length 1. Despite the large increase in the number of possible strategies (from 32 to 131,072 with memory_length 2), the algorithm converged in 6 generations. I believe that this is because the opponent strategies remain relatively simple, resulting in a smoother fitness landscape. As such the high mutation and crossover rates lead to quickly exploration and identification of effective strategies that perform well against the fixed set of opponents.

A quick experiment increasing memory from 2 to 3 yielded a fitness of 3.089, a very small improvement, after 28 generations.

1.4.3 Experiment 3: GA v Always Cooperate

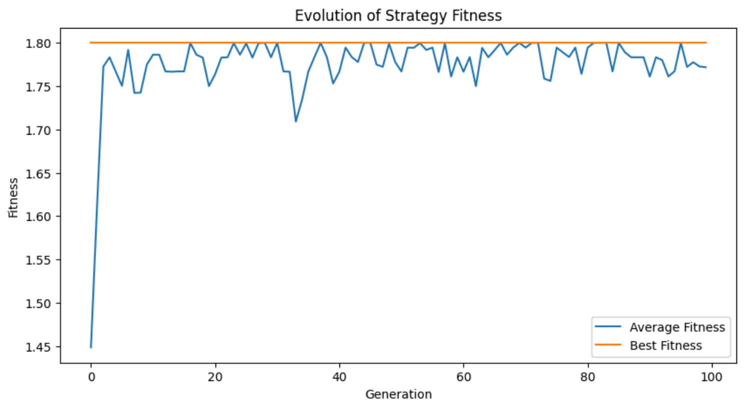
This experiment was conducted with memory length 2, and no changes to other parameters, expect the only opponent we are playing against is 'AlwaysCooperate'.



We achieve optimal fitness at generation 1. This is entirely unsurprising as with a population size of 50, we just need a strategy that always defects in response to 'C' which will almost always exist in the initial population.

1.4.4 Experiment 4: GA v Always Defect

I conducted the same experiment as outlined above, with the sole opponent of 'AlwaysDefect' instead of 'AlwaysCooperate'.

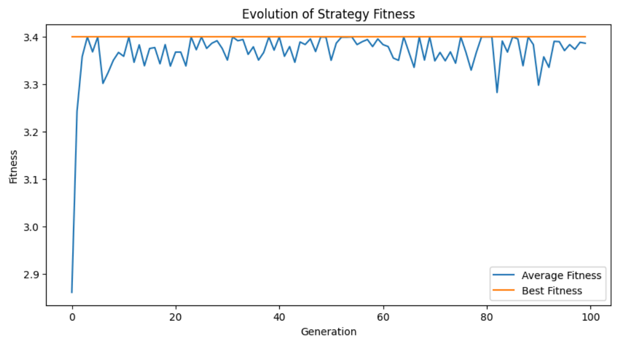


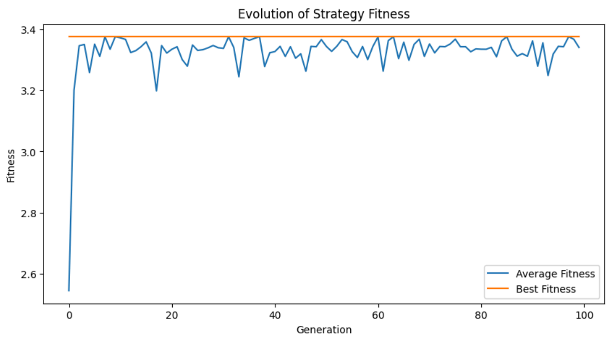
We achieve our best fitness of 1.80 at generation 1 yet again. As in the previous experiment, this too is entirely unsurprising as with a population size of 50, we just need a strategy that always defects in response to a defection, which yet again, will almost always exist in the initial population.

The fitness achieved 1.80 makes sense when you consider a payoff of 1 out of 5 in each of the 100 rounds that are involved in each generation. This gives us a score of 100/500, or 0.2, which when normalised using our formula gives us **1 + 4 * 0.2 = 1.8**.

1.4.4 Experiment 4: GA v TitForTat Variants

I conducted an experiment to compare TitForTat and SuspciousTitForTat with the same conditions as previous experiments.



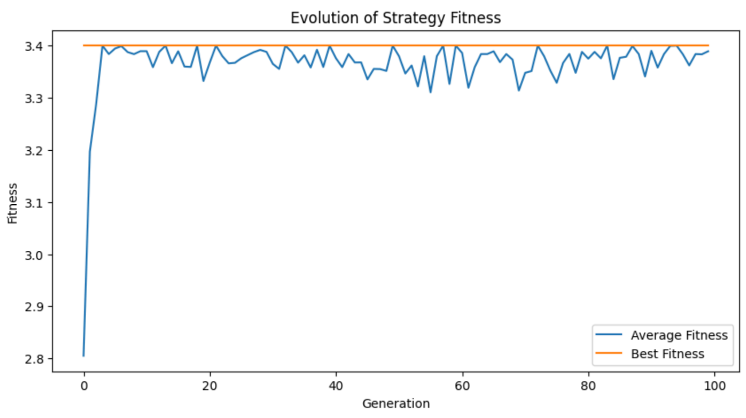


You can see above that against both components we achieve our 'best fitness' on the first generation, however, the fitness for SuspiciousTitForTat is 3.376 whilst the fitness for TitForTat is 4.0. The reason for this difference is because our maximum achievable score is reduced due to the need to defect in the first instance, earning only 1 'point' instead of the '5' we can achieve in the first round if playing against TitForTat.

The reason that we're unable to obtain a 'best fitness' of 5 is because in rounds where the opponent defects, we must also defect to get a maximum payoff. We can only achieve a score of 5 in the rounds where the opponent offers 'C' and we can play 'D'.

1.4.5 Experiment 4: GA v Pavlov

The final experiment that I will outline for this part is testing my genetic algorithm devised strategies against the Pavlov strategy.



In this experiment, the GA converged on a strategy that predominantly cooperates with Pavlov, achieving a normalized fitness of 3.4 (since mutual cooperation yields 3 points per round; 1+4×(300/500)=3.4). Pavlov's win–stay, lose–shift strategy ensures that defecting on the previous move is consistently punished, so the GA can't learn that defecting would be beneficial for the final move to achieve a slightly higher score. Consequently, here we can see that the strategies created by our GA believe that consistent cooperation leads to maximising payoff. This is why we achieve the optimal fitness in the first generation.

Part 2: Extension

2.1 Goals of Extension

The goal of extension is to see how some changes such as an extended genotype or the introduction of noise impact the performance of our evolved strategies against the fixed strategies we are playing against in this project.

2.2 Extending the Genotype

Extending the genotype by increasing the memory length enlarges the strategy's genome, which could allow for more nuanced behaviors.[4] I would note however that as seen in the first part of this project, the impact of increasing the memory length (as such expanding the genotype) was relatively small because the fixed opponents (e.g., AlwaysCooperate, AlwaysDefect, TitForTat, etc.) follow simple decision rules that don't fully exploit the added complexity of a longer memory.

Due to this limited benefit from genotype extension alone, I will in the next section 2.3 introduce noise so we can evaluate how noise can impact our GA.

2.3 Introduction of Noise

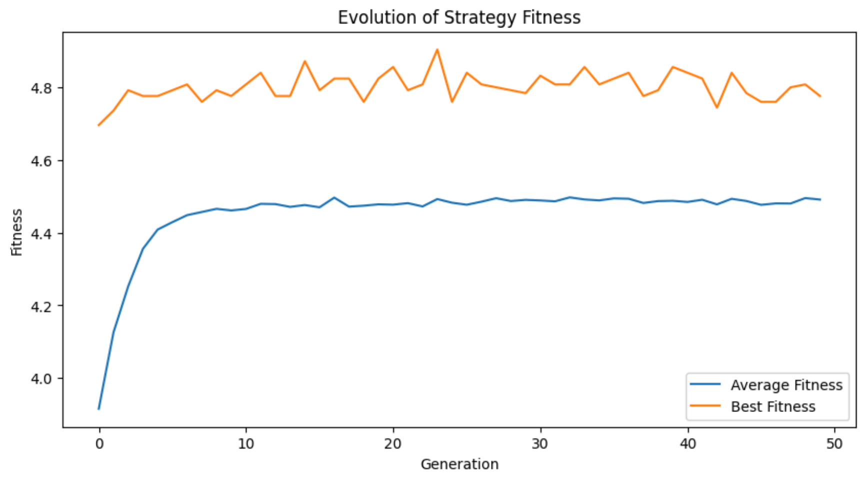
Noise, in the context of this project, refers to the making of random errors by a player in the iterated prisoner's dilemma. This should simulate real-world miscommunications that might occur, or execution mistakes that could be made.

The goal of running experiments with the introduction of noise is to test the robustness of different strategies in conditions that aren't perfect.

2.3.1 Extension Experiment 1: 2-way noise (affecting all players)

In this experiment, we apply a level of noise to both the fixed strategy opponents and the genetic algorithm’s evolved strategies. The reason to apply noise to both players in the game is because in realistic settings, noise affects all players.

I achieved this "2-way noise" by simply modifying the "play_ipd" function to have a chance (default of 0.1) to flip the move played by any player.

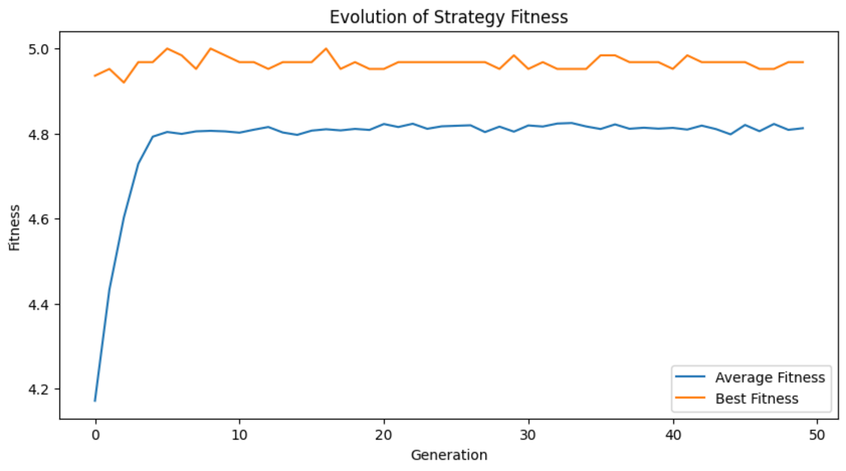


With this, when evolving our GA against an AlwaysCooperate opponent with a noise probability of 0.1, the best strategy achieved a normalized fitness of 4.904, compared to a perfect score of 5.0 without noise. This slight drop is due to the occasional mistakes that were caused by the noise, which prevent consistently optimal outcomes.

2.3.2 Extension Experiment 2: 1-way noise (affecting GA)

In this experiment, noise is introduced only to the moves made by the GA-evolved strategies whilst the fixed opponents strategies are maintained and unaffected. This provides a comparison to the above experiment in section 2.3.1.

I implemented 1-way noise by further iterating on the play_ipd function to accept the strategies are parameters, and if the strategy was not an instance of 'Strategy', therefore it was a genetic algorithm evolved strategy, we would apply noise.



Here we can see that with noise affecting *only* the GA evolved strategies, we can still achieve a perfect score of '5', which was achieved on generation 6. It might sound counterintuitive that this was achieved, however it makes sense with some further thought outlined below.

As we're playing against an 'Always Cooperate' type strategy, our goal is to always defect in order to maximise the payoff for our constantly cooperating. This means that all we need in order to achieve a perfect score is an evolved strategy that despite noise, will defect. This could be achieved if we can bypass noise entirely due to the small percentage chance noise will impact a move, or noise could even help us along the way by flipping a sub-optimal move to an optimal move!

A further quick experiment in which I increased noise to 0.5 resulted in a best fitness of 4.552, which aligns with the theory above.

2.3.3 Coping with Noise

Upon conducting some research on how to cope with noise in the Iterated Prisoner's Dilemma, I have found multiple interesting solutions[5].

Some suggestions state that noise can be coped with by making a reciprocatory strategy more generous. This would involve making a strategy such as 'TitForTat' for forgiving. This could involve cooperation despite the defecting of an opponent, which aims to help a player maintain cooperation in a noisy environment.

Another suggestion is the idea of 'contrition'[6], whereby a player acknowledges their own moves that were subjected to noise, ie. they tried to play move X, but played move Y. This self-punishing behaviour can allow a return to mutual cooperation and bring some stability back to the game, in spite of noise. This however, wouldn't work against an unforgiving opponent who punishes defection such as Grim Trigger, in which the opponent would permanently defect in response to a single defection from the player.

The final suggestion that I came across was the use of the Pavlov Strategy which adopts a methodology of "Win-Stay, Lose-Shift", meaning that in spite of noise, it can reward successful patterns of play by continuing to play these. This would be dependent on the form of noise, and isn't easy to demonstrate through my own experiments as it's possible an intended move repetition would be flipped. It does however provide a better chance than some other common strategies.

References Appendix

1. GeeksforGeeks, 2023. Genetic Algorithms. Available at: https://www.geeksforgeeks.org/genetic-algorithms/.
2. Wikipedia, 2024. Prisoner's Dilemma. Available at: https://en.wikipedia.org/wiki/Prisoner%27s_dilemma.

3. Wikipedia, 2024. Prisoner's Dilemma - Figure 1. Available at: https://upload.wikimedia.org/wikipedia/commons/thumb/2/22/Prisoners_dilemma.svg/560px-Prisoners_dilemma.svg.png.
4. Wikipedia, 2024. Chromosome (evolutionary algorithm). Available at: https://en.wikipedia.org/wiki/Chromosome_(evolutionary_algorithm).
5. Jstor, 1986. Evolution of Cooperation under Noise. Journal of Conflict Resolution, 30(1), pp. 109-128. Available at: https://www.jstor.org/stable/174327.
6. Axe, R., 1991. How to Cope with Noise in the Iterated Prisoner's Dilemma. University of Michigan. Available at: https://websites.umich.edu/~axe/research/How_to_Cope.pdf.