Predator-Prey Using Genetic Programming

Muhammed Bilal
Department of Computer Science
Brock University
St. Catharines, Canada
bb18hb@brocku.ca

Hamza Sidat
Department of Computer Science
Brock University
St. Catharines, Canada
hs18so@brocku.ca

***Abstract*—This research project aims to modify the artificial ant into a predator-prey application using Genetic Programming in DEAP. The methodology involves creating a predator-prey environment, defining the behaviors (language) of predators and prey, as well as outputting trace files to visualize the predator's path. The fitness goal is that the predator eats as many prey as possible in the simulation time (steps) allowed. Our findings indicate that the predators learn to catch the prey efficiently as generations increase. These results suggest that genetic programming greatly contributes to the field of machine learning when it comes to predator-prey dynamics.**

I. Introduction

Genetic Programming (GP) is a type of evolutionary algorithm inspired by biological evolution to find solutions to problems. It works by creating a population of possible solutions and then evolving them over generations using operations like mutation and crossover. DEAP is a python-based powerful library that allows us to implement genetic programming using a wide variety of built-in functions. GP and DEAP can help create a dynamic model of predator-prey interactions, allowing us to explore different strategies and behaviours in a controlled environment.

The Predator-Prey problem involves modifying the artificial ant into a predator-prey application. There is one predator, and multiple prey. The fitness goal is that the predator eats as many prey as possible in the simulation time allowed. Prey are essentially 'dumb' pre-programmed food that move around at random.

In this project, our model represents predators and prey as individuals within a population. Each individual's behavior is determined by a set of languages, which are evolved over generations using genetic programming. We use DEAP's built-in selection, crossover, and mutation operators to evolve these behaviors over generations, with the goal of exploring the dynamics of predator-prey interactions in a simulated environment, and to investigate how different behaviors manifest over time.

II. Problem Description

In this problem, we aim to develop a program that reads a text file representing a map of the simulated environment where the predator-prey interactions will take place. In this file, 'S' denotes the predator's location, '#' represents food, and '.' signifies empty cells. Our model represents predators and prey as individuals within a population. Each individual's behavior is determined by a set of languages, which are evolved over generations using genetic programming. We use DEAP's built-in selection, crossover, and mutation operators to evolve these behaviors over generations, with the goal of exploring the dynamics of predator-prey interactions in a simulated environment, and to investigate how different behaviors manifest over time.

To achieve this, we will start by generating random individuals or ants, each representing a tree-like structure. These individuals, also known as expressions, will represent a population. Initially, these individuals are basically ants, performing random actions on the map that do not necessarily catch all the prey.

Through the process of evolutionary computation, our program will iteratively refine these individual predators on a generation-by-generation basis. Each generation involves selecting individuals from the current population, applying genetic operators such as crossover and mutation to create offspring, and evaluating the fitness of these offspring by simulating their behavior on the map and counting the number of prey they successfully consume.

An individual's fitness corresponds to the number of prey it consumes. The more prey a predator captures, the higher its fitness, indicating greater success in this simulation. This fitness function guides the evolutionary process, favoring those individuals who consume the most prey.

By iterating this evolutionary process across multiple generations, our program aims to evolve predators that maximize prey consumption. Essentially, we're demonstrating the abilities of a predator-prey simulation.

To evaluate the performance of our program, we will conduct multiple experiments, each consisting of 10 runs with different random seeds. We will explore a few parameter variations, focusing on crossover and mutation rates, as well as comparing the fittest predators' performance on a new simulated map. These investigations will improve our understanding of how different parameter configurations influence the precision of our predator-prey approach.

III. GP Parameters Tables

In this section, we present a comprehensive overview of the genetic programming (GP) parameters used in our experiments. These parameters play a crucial role in shaping

the behavior of the evolutionary algorithm and ultimately influence the quality of the solutions obtained.

TABLE I: Default GP Parameters

Parameters	Values
Population Size	400
Crossover Probability	90%
Mutation Probability	10%
Generations	70
Number of Elites	1
Tournament Size	3
Min Tree Size (Init.)	1
Max Tree Size (Init)	2
Max Tree Size	33
# of Runs	10

TABLE II: Parameter Experimentation Table

Table Combination	Parameters		
	Crossover	***Mutation***	***Elitism***
1	90%	10%	1
2	100%	0%	1

[a]Different combination of parameters used

A. Meanings and Explanations of parameters

- Crossover: Crossover facilitates exploration by generating diversity in the population. It allows for the exchange of genetic material between individuals, potentially creating offspring with beneficial combinations of traits that were not present in the parent solutions. This diversity helps prevent premature convergence to sub-optimal solutions by exploring different regions of the search space.
- Mutation: Mutations adds an element of randomness to the individuals, utilizing a bit of exploration as well as exploitation to the problem. It achieves this by introducing a bit of new genetic material to the individual, potentially causing unexpected results. Excessive mutation, however, can also hinder the convergence of the evolutionary process.
- Elitism: Elitism promotes exploitation by preserving the best solutions throughout the evolutionary process. By ensuring that the best individuals survive from one generation to the next, elitism helps maintain or improve the overall quality of solutions over time by preserving some of the fittest individuals.

IV. GP Language

TABLE III: FUNCTION SET

Function	Arity	Example
if_food_ahead	2	if_food_ahead(x, y)
if_food_behind	2	if_food_behind(x, y)
prog2	2	prog2(x, y)
prog3	3	prog3(x, y, z)

TABLE IV: TERMINAL SET

Terminal	Action
move_forward	Moves predator forward in its direction
move_backward	Moves predator backward in its direction
turn_left	direction to face left
turn_right	direction to face right

A. Meanings and Explanations of GP Language

- if_food_ahead: This function calls 'sense_food' which detects whether a prey is located in front of the predator or not.
- if_food_behind: This function calls 'sense_food' which detects whether a prey is located behind the predator or not.
- prog2: This function performs any sequence of 2 functions or terminals.
- prog3: This function performs any sequence of 3 functions or terminals.

V. Fitness Evaluation

The fitness function serves as the guidance of the evolutionary algorithm towards the goal, which in our case is to evolve a predator to be able to consume as many prey as possible. Our algorithm in DEAP does this in a very simple way:

Here, a number of things are occurring: We are transforming the tree expression in a callable function. The tree expression represents the movement and actions the predator took throughout the 2D-array map. For each individual predator, a fitness function is evaluated by simulating the path of the predator onto the generated map. The fitness of that predator is then evaluated based on the number of prey they have consumed. A predator consumes a prey when it enters the same position cell as the prey.

A. Formula

The fitness formula can be represented as:

$$\text{Fitness(ind)} = PreyEaten \tag{1}$$

Through the fitness formula stated above, it is clear that the fitness of each individual depends on the count of prey they have eaten. The higher their prey consumption, the fitter the individual predator will be.

VI. Experiments

The independent variables that we experimented with are all labelled in Table II. It consists of different combinations of crossover & mutation rates, all with elitism involved. The experiments resulted in 2 graphs being produced. Furthermore, we conducted experiments of the predator-prey evolution with and without walls. The experiments without the walls consisted of the predator passing through the end of the 2D map and reaching the opposite end of the same map, whereas, experiments consisting of actual walls, prevents the predator and preys from passing through them, simulating a more realistic approach.

For our experiments, we are using **raw fitness** to represent the fitness of each individual and to visualize the fitness using graphs, where, larger the fitness value, better the solution.

A. 90% Crossover, 10% Mutation, With Elitism

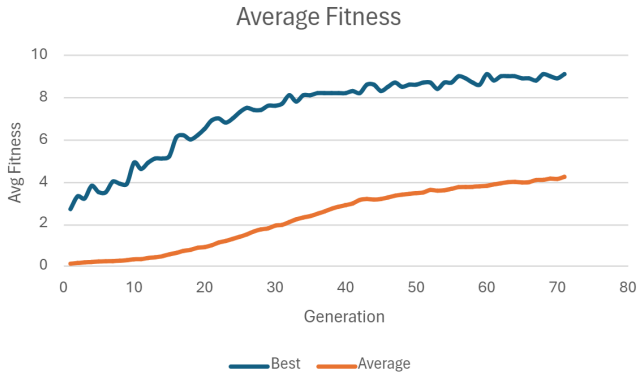

Fig. 1: Best and Average Fitness With Elitism using 90% Crossover and 10% Mutation

Through the figure above, the fitness of the predators are clearly improving on average, as seen through the convergence of the fitness values as generations pass.

B. 100% Crossover, 0% Mutation, With Elitism

As per the plot in Figure 2, the plot looks somewhat similar the one in Figure 1, depicting similar fitness values and convergence.

C. 90 % Crossover, 10% Mutation, with Walls

For the plot of the simulation where walls are introduced in the playground (Figure 3), we noticed significant decrement in the average fitness for both, best and average fitness of all runs.

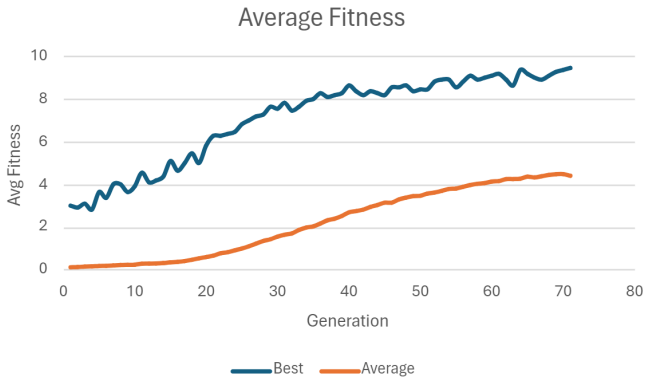

Fig. 2: Best and Average Fitness With Elitism using 100% Crossover and 0% Mutation

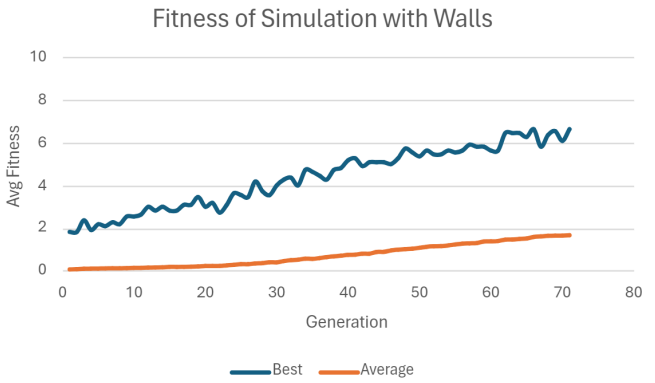

Fig. 3: Best and Average Fitness with Walls

VII. Results

Through our results, it is shown that the average runs involving the parameters of 100% crossover and 0% mutation (shown in Figure 2) ended up performing an overall fitter solution than using the parameters of 90% crossover and 10% mutation (shown in Figure 1). This difference can be seen where the average best solution from Figure 2 ends up reaching a higher fitness value by the end of the generations than that of Figure 1.

As per our observations from multiple runs using different hyper parameters, we found that predator was following specific patterns as seen in the trace array printed in the end the run. Secondly, in our comparative experiment, we noticed huge difference in the change of these patterns. We compared the behaviour of the predator in a field with and without the wall through Figures 4 and 5. We noticed that in Figure 5, the predator was moving randomly at first, and then, it was only following the path along the wall, which also resulted in a lower fitness score because it adapted the behaviour for chasing preys near the wall only, resulting in the consumption of lesser prey overall.

As we can see in Figures 4, the predator in playground without wall behaves better than the one without the wall. This is because if does not adapt the static nature of the locations of the preys, which stops when they encounters the wall, leveraging predators to eat only nearby the wall. Furthermore, the predator, being able to pass through walls and get to the other side, encourages greater exploration of the map, thus increasing chances of encountering and consuming more prey, resulting in an overall higher fitness.

VIII. Conclusion

Through our experiments above, we observed that the fitness for same simulation, with varying hyper parameters, resulted in similar fitness curves. However, when we changed the simulation environment by introducing wall to the playground, we noticed significant decrement in the overall fitness of runs. Additionally, as per the trace matrix printed, we observed that the predator followed the path of wall after first few moves. The given work can be improved by introducing an additional

Fig. 4: The Predator's trace without using walls

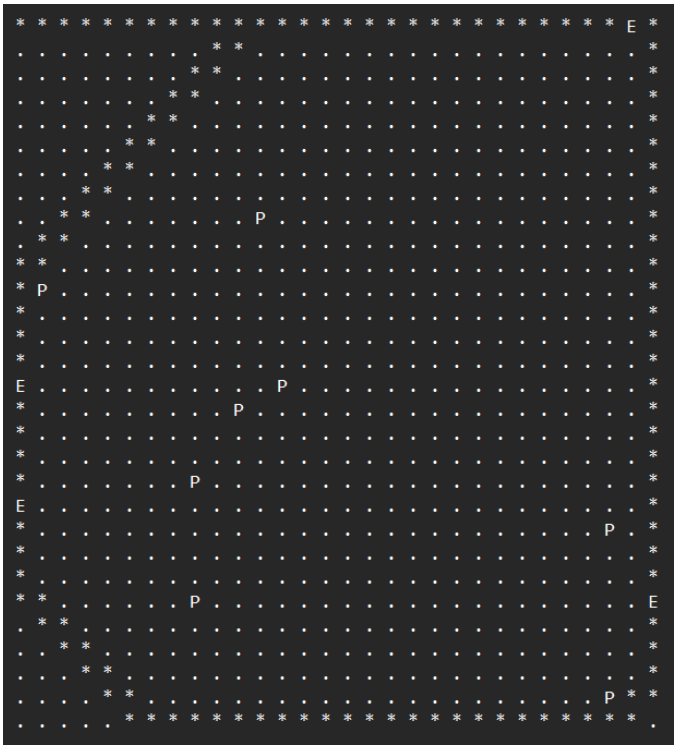

Fig. 5: The Predator's trace using walls

terminal sets and function sets, such as if_wall_ahead. Furthermore, unlike our current behaviour of preys which stop at the wall, we can introduce new movements for them, when they encounter the wall causing them to move around and away from it.

REFERENCES

[1] DEAP Development Team, "DEAP: Distributed Evolutionary Algorithms in Python Documentation," Available: https://deap.readthedocs.io/en/master/, [Accessed: February 07, 2024].

[2] R. Poli, W.B. Langdon, N.F. McPhee, *A Field Guide to Genetic Programming*, ISBN 978-1-4092-0073-4, 2008.

APPENDIX

APPENDIX A: GP TREE OF THE BEST INDIVIDUAL IN A SIMULATION WITH THE WALL

```
prog3(prog2(prog3(move_backward, move_backward,
↪  prog3(move_backward,
↪  if_food_behind(prog3(move_backward,
↪  move_forward,
↪  prog2(prog3(prog3(move_forward,
↪  if_food_ahead(move_backward,
↪  prog3(if_food_behind(turn_right,
↪  move_forward), prog2(move_backward,
↪  turn_left), prog3(move_forward,
↪  move_forward, move_backward))),
↪  if_food_ahead(turn_right, move_forward)),
↪  move_forward, if_food_behind(move_forward,
↪  prog3(prog3(prog3(move_backward,
↪  move_backward, prog2(move_forward,
↪  move_forward)), prog3(move_forward,
↪  move_forward, move_backward),
↪  move_backward), move_backward,
↪  move_forward))), turn_right)),
↪  prog3(move_backward, move_backward,
↪  prog2(move_backward,
↪  prog3(if_food_ahead(move_forward,
↪  move_backward), move_forward,
↪  if_food_ahead(move_forward,
↪  prog3(move_forward, move_forward,
↪  move_forward))))))),
↪  if_food_ahead(prog3(move_forward,
↪  prog3(prog3(move_backward,
↪  if_food_ahead(move_backward,
↪  move_backward),
↪  if_food_ahead(prog3(prog3(turn_left,
↪  move_forward, move_forward),
↪  prog3(turn_right, move_backward,
↪  turn_left), turn_left), move_forward)),
↪  move_backward, turn_left), turn_left),
↪  move_forward))),
↪  if_food_ahead(if_food_behind(move_backward,
↪  prog2(if_food_ahead(move_forward,
↪  prog3(turn_left,
↪  if_food_ahead(prog3(turn_left, turn_left,
↪  move_backward), prog3(move_forward,
↪  move_forward, move_backward)), turn_left)),
↪  prog3(if_food_behind(prog2(move_forward,
↪  turn_left), prog3(turn_right,
↪  move_backward, turn_left)), move_forward,
↪  if_food_ahead(prog2(move_backward,
↪  move_forward), prog3(move_forward,
↪  turn_left, prog3(move_backward,
↪  move_forward, move_forward))))))),
↪  if_food_ahead(move_backward,
↪  if_food_behind(move_backward,
↪  prog3(turn_right, move_backward,
↪  turn_left))))), move_forward,
↪  prog3(move_forward, move_forward,
↪  prog3(move_backward, move_forward,
↪  if_food_ahead(move_backward,
↪  move_backward))))
```

APPENDIX B: GP TREE OF THE BEST INDIVIDUAL IN NORMAL SIMULATION(WITHOUT WALLS)

```
prog3(prog2(move_backward, prog3(move_backward,
↪  prog3(prog3(prog3(move_backward,
↪  prog3(move_backward,
↪  if_food_behind(turn_left, move_forward),
↪  prog3(move_backward, move_backward,
↪  move_backward)), move_backward),
↪  move_backward, move_backward),
↪  prog3(move_backward, move_backward,
↪  move_backward),
↪  if_food_ahead(prog2(move_forward,
↪  move_backward), move_backward)),
↪  move_forward)),
↪  prog3(prog2(prog2(move_backward,
↪  move_backward), prog3(move_backward,
↪  move_backward, prog3(prog3(turn_left,
↪  move_forward, move_backward), move_backward,
↪  turn_right))), move_backward,
↪  move_backward), move_backward)
```