Comparison of Genetic Algorithm and Monte-Carlo implementation on Ms Pacman

COMP4105 Designing Intelligent Agents

Jarrad Foley, 10338223

Abstract

This paper compares the viability of using a Genetic Algorithm and Monte-Carlo policy within a game space to play the game Ms Pacman and comparing how they differ. The method implemented are a Standard Genetic Algorithm using Roulette Wheel Selection for parent selection and a Monte-Carlo state-action value algorithm with a soft policy to ensure all states have a non-zero probability of being used. These methods are used within the OpenAI Gymnasium on the Ms Pacman environment using a deterministic implementation of the game. The Genetic Algorithm made a final score of 10441 and the Monte-Carlo Policy got a final score 1860. The Genetic Algorithm works best for a high score, eating blue ghosts, and surviving the longest. The Monte-Carlo algorithm was good for a non-deterministic environment. The reward function needs to be worked on more to properly show the ability of the behaviours in the Monte-Carlo algorithm.

Introduction

In the field of machine learning there have been many different algorithms developed to encompass a range of different situations from image processing to pathfinding. This project is aimed around implementing a machine learning algorithm into the game space Ms Pacman provided by OpenAI Gymnasium [1]. While there are many ways to go about solving Ms Pacman, this project focuses on Genetic Algorithm and Monte-Carlo implementation within a deterministic Ms Pacman simulation. The Genetic Algorithm involves creating a population of agents initialised with a list of random instructions to be executed, evaluated, and then changed and improved through roulette wheel parent selection, genome crossover, and mutation for the next generation. The Monte-Carlo algorithm involves creating a state-machine to decide on actions for given states and assigning a reward for that action and averaging the different rewards for however many times it visits that state.

Relevant Research

For relevant research on these topics I first found a very helpful and informative paper on the concepts of Genetic Algorithms [2]. This provided general structure of how to create genetic algorithms and gave me a good idea of to how it would work in practice.

Whilst looking into Monte-Carlo soft policy algorithms I had found a research paper proposing to use a Monte-Carlo Tree Search policy to avoid pincer moves from the ghosts [3]. This paper is what gave me the idea of creating a graph of points and direction of the game board and to use a branching breadth first search algorithm to find the closest location of the ghosts and pills.

The next piece of relevant research I found was on the “Reactive control of Ms. Pac Man using information retrieval based on Genetic Programming” [4]. This had many interesting ideas of what to try such as defining safe spots for Pacman and defining escape actions which I used as an idea of evading the ghosts.

On top of this I also looked at a paper titled “The Comparison of Three Algorithms in Shortest Path Issue” [5] when considering what shortest path algorithm to use, it showed. I settled on using the breadth first search algorithm as it worked well for my problem and there was an easy implementation for it. [6]

Following from that I also looked back at the first semester work [7] which had a work through of a lot of reinforcement learning ideas and pseudocode for them, including the Monte-Carlo soft policy which I used.

The final notable paper I read was titled “Evolving diverse Ms. Pac-Man playing agents using genetic programming” [8]. This has some great ideas for behaviours that can be programmed into an evolutionary program such as checking how long it would take Pacman to get to a power up pill and if any ghost near by could reach either Pacman or the power pill first then it would deem it unsafe to attempt to go for it.

Methodology

Environment

For the environment of this project, I used the OpenAI Gymnasium with the MsPacmanDeterministic-v4 environment. This environment creates the entire Ms Pacman game.

The Pacman and Ms Pacman environments at each time step returns the following variables: observation, reward, terminated, truncated, info.

I utilise only the first 5 actions for the algorithms, noop, up, right, left, and down as this represents all of the directions that I believe to be necessary and avoiding unnecessary additional actions where the difference is only seemingly needed for when using a joystick.

I used the deterministic version of the game so to make the ghosts do the same actions each time. This is because when trying to train a Genetic Algorithm on an environment that has enemies which display randomness, a high score on that attempt may not be an accurate representation as when ran another time could show any value between 0 and even higher than the previous score it got without the instructions changing.

In order to do the following algorithms I needed to get the positioning of the objects on the board. To do this I had to utilise both the RGB observations and the RAM observation settings provided by OpenAI Gymnasium.

I used the RAM observation setting to gather the coordinates for each of the ghosts, Ms Pacman, and the cherry, and the number of lives.

I also used the RGB observation to create a smaller map by observing how many pixels roughly make up each tile and discretising it from being a 210x160 pixel printout of pixels to being a 20x15 pixel map where every 8 pixels is a new entry in the map. This allows me to have a much smaller state size allowing for easier generalisations and less amounts of times to run to fill in a good movement for each state.

Genetic Algorithm

For the Genetic Algorithm, I followed a standard methodology for the genetic algorithm described by the following pseudo code found in the “Genetic algorithms: concepts and applications” paper [2]:

```
Standard Genetic Algorithm ()
{
	// start with an initial time
	t := 0;
	// initialize a usually random population of individuals
	initpopulation P (t);
	//evaluate fitness of all initial individuals of population
	evaluate P (t);
	// test for termination criterion (time, fitness, etc.)
	while not done do
		// increase the time counter
		t := t+ 1;
		// select a sub-population for offspring production
		Ir := selectparents P (t);
		//recombine the ”genes” of selected parents
		recombine P (t);
		// perturb the mated population stochastically
		mutate P (t);
		//evaluate it’s new fitness
```

```
            evaluate P (t);
            // select the survivors from actual fitness
            P := survive P,P‘ (t);
      od
}
```

To represent this algorithm within the program I had three fundamental classes: Brain, Agent, and Population.

The brain class to initialise the random instructions of the first generation, a method of cloning the current brain for the next generation, and a mutation function to mutate the next generations brains after each run.

The agent class initialises the brain and size of the list of instructions the brain creates, a function to create offspring for the next generation and a return of fitness value for that agent.

The population class is the main section. It contains the evolutionary components of the algorithm such as the roulette wheel parent selection functions for choosing which parents are to create offspring for the next generation, the genome combination function to create children with an amount of genes from either parent, and the creating the next generation function.

During the parent selection I did a slight modification. I added the best parent from the previous generation to always move to the next generation along with a second copy of it which goes through the genome crossover and mutation. This is so that irrelevant of what happens there will either be an improvement or it wont change. The rest of the population are selected through the roulette wheel selection.

When training I created a population with a size of 50 and ran the algorithm for 1000 generations taking the best agent from each population to move to the next generation, a mutated version of the best agent, and then used the roulette wheel algorithm to choose the remainder of the population. Then applied one point genome crossover on the population during natural selection.

Monte Carlo Soft policy

The Monte-Carlo soft policy algorithm I used I got from the lecture notes of my first semesters PHYS4035 Machine Learning in Science Part 1 [7] which provided the following pseudocode:

```
Data: Soft policy π
Result: Estimated state-action-value function Q(s, a) ≈ qπ(s, a).
Initialise a state-action-value-function Q(s, a) ∈ R, arbitrarily, for all s ∈ S;
Initialise an empty list, Returns(s, a) of sampled returns for all s ∈ S;
for i ← 1 to Nmax do
      Sample a trajectory, ω, using π: ω = (S0, A0, R0 . . ., ST , AT −1, RT −1);
      G ← 0;
      for t in T − 1, T − 2, . . ., 0 do
            G ← γG + Rt;
            Append G to Returns(St, At);
            Q(St, At) ← average(Returns(St, At));
      end
end
```

The implementation of this I used from a prior piece of work [9] that I and my group partner developed as it would essentially be the same functions I would create in the same way and when they’re also in the same structure I deemed it unnecessary to rewrite.

States

The Monte-Carlo algorithm utilises the discretised map I created to be used as the state space for the state machine.

For the states I store the information of Ms Pacman's X and Y values and the X and Y values of the closest ghost to Ms Pacman. The Y coordinates are of value 15 and X coordinates are of value 21. The total number of possible states is 99,225.

Actions/Behaviours

For this algorithm I have implemented three behaviours that the agent can choose to do to determine which action to choose: run away from the ghosts, collect pills, and chase the ghosts.

The basis behind how these behaviours are implemented is using a breadth first search algorithm [6]. This algorithm used a graph of all the different coordinates on the discretised map and the directions from those coordinates the agent can move to. For this to work I produced two maps, a ghost position map, and a pill position map. The ghost position map shows the discretised ghost positions and sets the value to be 3, 4, 5, and 6 for the orange, blue, pink, and red ghost respectively, the walls being set as value 1, and the movable areas are set to 0. The pill map follows the same scheme with the walls being set to 1 and the pills on the map are set to 2.

The breadth first algorithm checks the current position of the agent and goes out in each direction one branch depth at a time until it reaches it's the end condition, in this case it's the closest ghost/pill.

In the behaviour of running away from the ghosts it will find the closest ghost and choose the first direction that isn't on the shortest path towards the ghost to get away. When pill collecting or chasing the ghosts it will follow the shortest path to get to them.

During pill collection, when the agents discretised x and y coordinates reach the coordinates of a pill indicated by the value being 2, that coordinate will then be set to 0 as to allow the behaviour to find the next coordinate with a pill on it.

Rather than manually writing my own breadth first search algorithm, I found an online solution which is quite concise [10].

Reward

I used the following reward function:

$$R = 10a + 50b - c - 2d + 10e$$

Where:

$$a = \ Ghost\ Distance$$

$$b \ = \ Score$$

$$c = \begin{cases} c = 500 \ \ where\ a \ < \ 2 \\ c = 300 \ \ where\ a \ < \ 3 \\ c = 100 \ \ where\ a \ < \ 4 \end{cases}$$

$$d \ = \ Pills\ Left$$

$$e \ = \ Lives$$

I designed it this way to try to promote the use of the pill collecting behaviour more than the rest unless the ghosts get too close to the agent which it then would switch to the ghost evasion behaviour and run away.

Challenges

I had the most challenges with the Monte-Carlo algorithm.

The main challenge I was faced with was figuring out how to get information from the observation from OpenAI Gymnasium. Many of the games available on OpenAI Gymnasium have a fully worked out observation section which explains player and opponent positions, angles, anything relevant to the game field, however, Pacman and Ms Pacman variants didn't have this comprehensive list. There were only two options for observations with Pacman and Ms Pacman, RGB and RAM. The RGB gave a 2-dimensional array of RGB values showing every pixel on the game screen whereas RAM gave a 1-dimensional array or information. I had to

figure what each of these values was meant by manually printing out the observations into a CSV file for roughly the first 200 frames of the game and setting the action of Ms Pacman to be moving left then upwards. From there I colour coded the CSV file so that if the current cell is larger than the previous cell it would be coloured green, and if it is smaller than the previous cell it would be coloured green. This helped me to decern whether the object is moving in a positive or negative direction from its previous state.

By doing this and watching what Ms Pacman does given the action of go left and upwards I could figure out which frame it changes direction and match that up with the CSV to find an entry which is moving negatively (indicating left movement) until that frame and then doesn't change whilst also seeing where another entry changes from not moving to moving negatively (indicating upwards movement). I then repeated this process of watching the ghosts for their movement of the red moving in a certain path and the other ghosts moving in the starting square in a predictable pattern I figured out which output from the RAM observation the ghosts represented by. This showed that the RAM observation produces all the X coordinate values of Ms Pacman and the ghosts first and then does the Y values.

Another problem I encountered was that when Ms Pacman would go through the tunnel to appear on the opposite time, the graph set up for directions would crash because the map did not extend that far so the program wouldn't know where to put it. It would constantly error if it got too close to the tunnel, so I had to set up the graph to not allow the agent to go through the tunnel. This ended up having a detrimental effect to the agents as there was a few times in which it could have used the tunnels to get away from the ghosts and avoid being boxed in.

A big challenge was trying to get a reward function that would positively encourage the agent to do a beneficial action such as running away from the ghosts when they're too close and then changing to collecting pills when it is save.

Observations

The Monte-Carlo Algorithm with all behaviours clears around the game board relatively well, eating a few ghosts as it does it.

The Genetic Algorithm works particularly well when it comes to eating the ghosts when they go blue. In general, the placement of the ghosts when the agent collects the pill to turn the ghosts blue allows it to eat all the ghosts each time, excluding the final time where it only gets one.

Deterministic Experiments

I use four different agents for the experiments. The first being the Genetic algorithm and the other three are three Monte-Carlo agents. The differences are the behaviours programmed in them. There is a Monte-Carlo that only does the pill collecting behaviour, one that only does the behaviour of running away from the ghosts, and the other has all three behaviours implemented and uses the state-machine to choose the best behaviour to use at each time step.

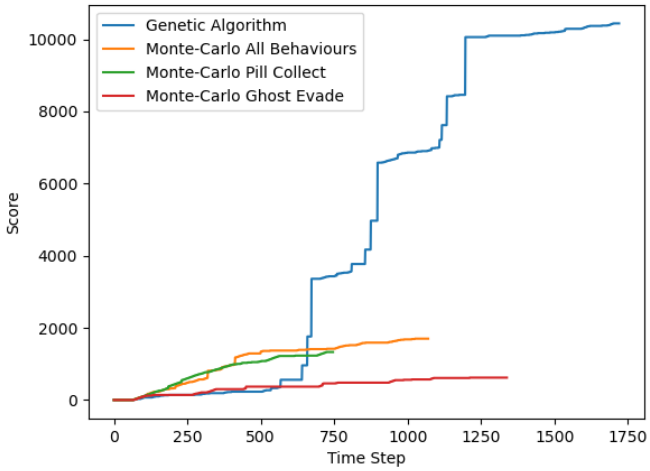
Number of pills eaten

Algorithm	**Pills Collected**
Genetic Algorithm	112
Monte-Carlo All Behaviours	114
Monte-Carlo Pill Collect	126
Monte-Carlo Ghost Evade	54

The best agent for collecting the number of pills is using only the pill collection behaviour although the Genetic Algorithm and Monte-Carlo algorithm with all behaviours wasn't far behind.

It misses a few pills on its way because of a small problem I had to work around when creating the small game map it counts the agent going into a tile and collecting the pill when it just slightly goes into it and misses the pill instead meaning it doesn't go back to get them as it thinks it already has.

Highest scoring



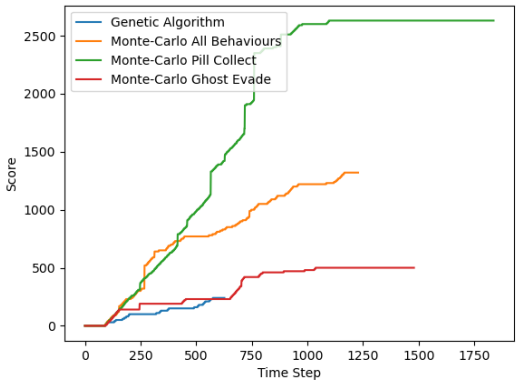
As you can see on the graph, the Genetic Algorithm outperforms the Monte-Carlo algorithm by a significant margin. This is due to the genetic algorithm eating all the ghosts when the turn blue for the first three power up pills it eats as well as collecting 112 pills at the same time and the cherry.

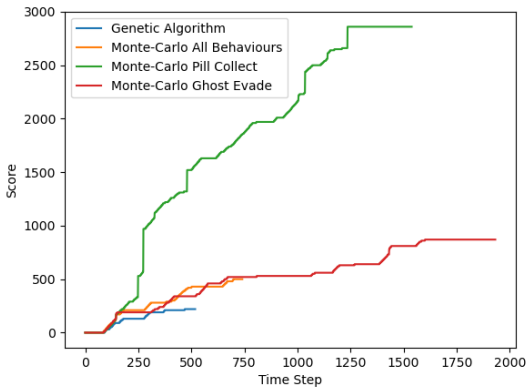
Survivability

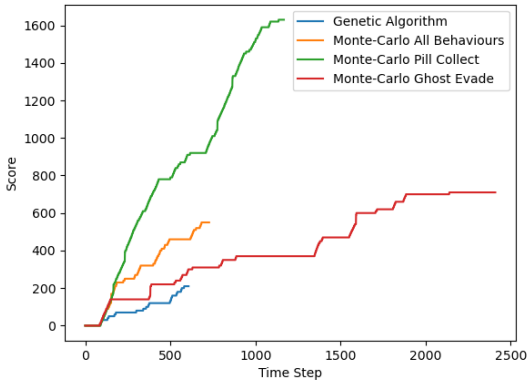
Algorithm	**Time steps survived**
Genetic Algorithm	1720
Monte-Carlo All Behaviours	896
Monte-Carlo Pill Collect	746
Monte-Carlo Ghost Evade	1338

The Genetic algorithm performed significantly better than all of the other algorithms surviving roughly 23% better than the next highest algorithm.

Non-Deterministic





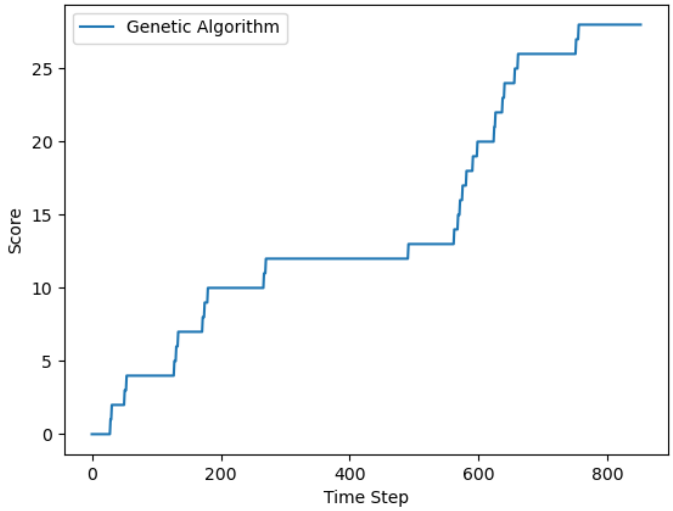


I ran the non-deterministic environment on the four models three times each. These show that the pill collection behaviour each time was vastly superior in score as it was able to collect most of the pills each time and a few times ate the ghosts when they were blue.

Change to Pacman

The Monte-Carlo Algorithm is unable to run on the regular version of Pacman. Due to the board size and wall positions being different, unless I remake the map, modify the parameters for the state-machine, and train from scratch to find values for the new states it can be in this will just crash the project.

The Genetic Algorithm performed very poorly on this. This is because the genetic algorithm best agent was trained specifically on the game space for Ms Pacman which uses a different map and has different ghost behaviour than regular Pacman does and so the list of instructions does not translate well to the new environment. However due it its ability to run it perform better than the Monte-Carlo algorithm which cannot run at all.



Results

The Genetic Algorithm performed better with getting a higher game score as it was able to eat a lot of the ghosts when they turned blue and survived the longest amount of time.

Monte-Carlo Policy performed better on the non-deterministic environments. This shows it has some elements of generalisation which helps it perform well on when the ghosts are doing different moves than its used to.

The Monte-Carlo Pill Collection behaviour is the best with random ghosts movement and can collect the most amount of pills on the board.

The Monte-Carlo evade ghosts behaviour didn't seem to be good at any of the tasks. It got stuck a lot and was pushed to hide in a corner for most of the game before it died.

Conclusion

The results show that the Genetic Algorithm works well when it comes to pills eaten, scoring highest, and surviving for the longest however fails when it comes to using a non-deterministic environment.

The Monte-Carlo algorithm with all behaviours is very limited by the reward function it has, if the reward function doesn't reward correctly then no matter how good the behaviours are, it won't perform how it is intended to perform.

I believe, with a better rewards function that encourages pill collection as well as avoiding the ghosts when they're close it would outperform the genetic algorithm in most aspects as it then has behaviour not reliant on the game board but on the relative positions of the other objects on the board.

Future work

Future work I would do on this is adapt the genetic algorithm int using genetic programming to generate behaviours rather than a list of instructions so that it could be more adaptive to other environments.

Another thing I would do would be to add in when the ghosts are blue into the states so that the behaviour to chase the ghosts has more of a chance to be used. I overestimated the scope of this project and the problems I faced when doing it that I couldn't implement this feature.

A future development could be to implement NEAT (Neuroevolution of augmenting topologies) into the genetic algorithm for better performance.

References

[1]"MsPacman", OpenAI Gymnasium, https://gymnasium.farama.org/environments/atari/ms_pacman/, 2022

[2] K. F. Man, K. S. Tang and S. Kwong, "Genetic algorithms: concepts and applications [in engineering design]," in IEEE Transactions on Industrial Electronics, vol. 43, no. 5, pp. 519-534, Oct. 1996, doi: 10.1109/41.538609.

[3] N. Ikehata and T. Ito, "Monte-Carlo tree search in Ms. Pac-Man," 2011 IEEE Conference on Computational Intelligence and Games (CIG'11), Seoul, Korea (South), 2011, pp. 39-46, doi: 10.1109/CIG.2011.6031987.

[4] M. F. Brandstetter and S. Ahmadi, "Reactive control of Ms. Pac Man using information retrieval based on Genetic Programming," 2012 IEEE Conference on Computational Intelligence and Games (CIG), Granada, Spain, 2012, pp. 250-256, doi: 10.1109/CIG.2012.6374163.

[5] Xiao Zhu Wang, "The Comparison of Three Algorithms in Shortest Path Issue", 2018 J. Phys.: Conf. Ser. 1087 022011

[6] Robbi Rahim "Breadth First Search Approach for Shortest Path Solution in Cartesian Area" 2018 J. Phys.: Conf. Ser. 1019 012036.

[7] J. P. Garrahan, E. Gillman and J. F. Mair, "Machine Learning in Science Part 1 (PHYS4035) Lecture Notes" Nov., 2022.

[8] Alhejali, Atif & Lucas, Simon. (2010). Evolving diverse Ms. Pac-Man playing agents using genetic programming. 1 - 6. 10.1109/UKCI.2010.5625586.

[9] J. Foley, S. McSweeney, "Drone Flight Controller, Project 9 - MLiS Part I", Jan. 2023

[10] SeasonalShot, "How to trace the path in a Breadth-First Search?", stackoverflow.com, Retrieved from https://stackoverflow.com/a/50575971 (accessed May, 2022).