CS 541: ARTIFICIAL INTELLIGENCE, SPRING 2022 GROUP ASSIGNMENT

USING AI FOR SNAKE GAME

Team Members:
Lasya Pendyala(921284620) - Worked on building the snake game interface using python and implemented Deep Q-Learning algorithm and logging the results for analysis.
Soumya Thoutam(954664146) - Worked on requirement analysis, reviewing game interface and implementing genetic algorithms and logging the results for analysis.
Anvitha Pasumarthi(956236565) - Worked on implementing Deterministic algorithms and logging the results for analysis.
Surya Majji(949699702) - Worked on implementing Brute force method of deterministic algorithms and logging the results for analysis.

0.Abstract

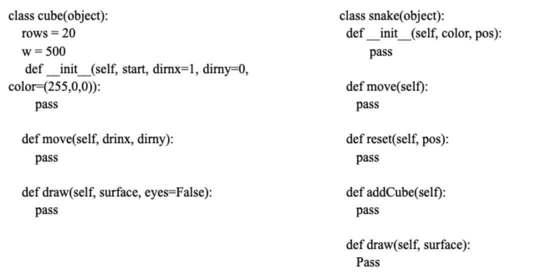
In this project we built a bot that can learn to play the snake game. It is a single player grid based game, the objective is to grow the snake as long as possible by collecting the food without colliding with itself or the wall. As people, we can develop different strategies for achieving an outstanding score but to train a bot we need to encode the strategies we intuitively come up with. In our project we implemented and analyzed algorithms from the extremely naive methods to the more complex ones, to maximize the game score. Firstly, we focused on deterministic algorithms to maximize the score. These algorithms will just encode the decision making logic into algorithm i.e., same output for a given input e.g Starting with a random approach(this is the least successful and naive algorithm which doesn't even use the information of location of food and length of grid) then we improved this by using shortest path and did some experiments to check for collisions. We ended deterministic approaches with a brute force method(Hamiltonian cycles). Moving forward we used Deep Q-Learning which is a specific type of Deep Reinforcement Learning. Before going to this approach we also gave genetic algorithm a try, which has been popularly used. In this the population of snakes will slowly learn how to play the game.

1.Introduction

The most popular and well-known game, 'The Snake,' was invented in the 1970s and is being played by many people today. Nokia phones were the first to feature this game. This game laid the groundwork for future mobile phone games. By pressing the arrow keys, the player controls a snake that must move around the screen while eating food. The tail increases one unit for every food particle consumed. The objective is to consume as many food particles as possible avoiding colliding with a wall or the snake's ever-expanding tail. It's a classic programming task to create an AI bot that can play Snake. The snake is bound in a two-dimensional board. With its head in front and its body behind it, the snake approaches the predetermined course. When the snake smacks against the playground walls or the snake body itself, the game is over. Non-ML techniques, genetic algorithms, and reinforcement learning are the three broad groups of approaches to an AI Snake agent. Using the intelligence gathered from the provided algorithms and training in the training environment, this AI bot will advance in the game.

2.Game Design

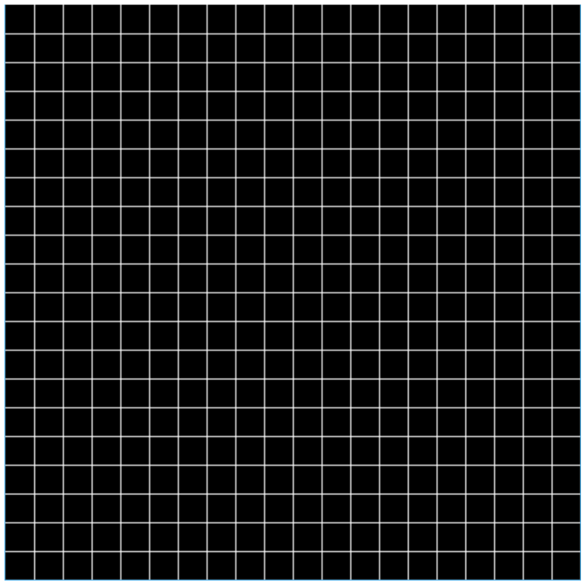
This series will show how we used python and pygame to make the popular Snake Game. To represent the play space and move things about the grid, we have utilized a typical grid system. We have used 2 main classes - Snake and Cube in our main structure of snake game UI.



```
class cube(object):
   rows = 20
   w = 500
    def __init__(self, start, dirnx=1, dirny=0,
color=(255,0,0)):
      pass

   def move(self, drinx, dirny):
      pass

   def draw(self, surface, eyes=False):
      pass
```

```
class snake(object):
  def __init__(self, color, pos):
      pass

  def move(self):
    pass

  def reset(self, pos):
    pass

  def addCube(self):
    pass

  def draw(self, surface):
    Pass
```

In every game, there is a loop known as the "game loop." Until the game is exited, this loop will continue to run. It is primarily responsible for detecting events and invoking functions and procedures in response to them. We coded our game loop in the main(). For our game we have used a 20x20 grid. We used the drawGrid() function to draw the grid.



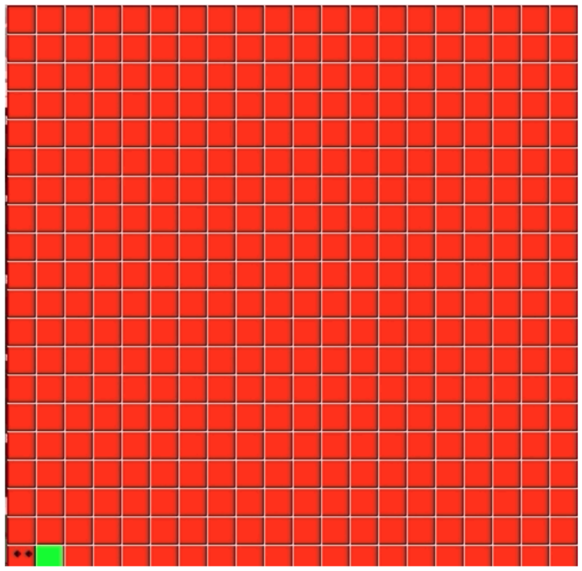
Our snake object has been made out of a list of cubes that depict the snake's body. These cubes will be stored in a list named body, which will be a class variable. Turning the snake is the most difficult component of this game. We need to remember where and in which direction the snake has turned. This is why, whenever the snake takes a turn, we add the head's location to a turn dictionary, the value of which indicates the direction the snake turned. We used the move() method, this method will first check for any event. Now that we can move our snake around the screen we need something for it to eat or collect. We will add a new cube to the end of our snake every time we clash with one of these things. This item will be referred to as a snack or food.
First, we created a position for our "food." This will be accomplished within the randomSnack() function. Next task is to find out the collisions. We lose the game when our snake object collides with itself.

2.1. Game Rules

The rules of the game to keep snake alive and achieve good score are:

1. Snake should not collide with the wall.
2. Snake should not collide with itself.
3. Snake should collect maximum food or snacks on its way.

In our game we may get complex situations where the snake achieves the highest score and can't move because of its length.



3. Techniques and their implementation

Here we demonstrate the algorithms that will be used to train the AI bot to make the best decisions possible during the game.

3.1. Deterministic Algorithms or Non-ML Techniques

This section focuses on deterministic score maximization algorithms. For any given input, deterministic algorithms will always provide the same output. In other words, we embed decision-making logic (e.g., don't turn your body, take the quickest path to the snack) into the algorithm. For each algorithm, We'll provide some intuition, issues, gameplay, and data collected from different runs. Though these techniques help to reach the best scores, there is no machine learning involved i.e., The algorithm must be manually encoded. It's helpful if you're familiar with graph theory. For huge game boards, it may be slow.

3.1.1. Random Approach

The simplest, and least successful, technique for an AI to play the game is to use a completely randomized algorithm. The AI moves a valid cell at random from a snake location (a list of cells occupied by the snake). This concept fails to make use of all of the available data (the location of the food or snack), and the gaming appears to be, well, random.

The games usually end with a score of 1 in the majority of cases. It will occasionally end with a score of 2.

3.1.2. Shortest Path

This technique has a quirk that arises from the sequence in which we search for the squares' distances from the food. Two things to think of while implementing this algorithm are, First, the snake will not move in the same direction as its body. Second, when selecting which way to travel, the snake examines the blocks in the following order: RIGHT, LEFT, UP, DOWN. Sometimes say if the food is on its RIGHT but the snake will only look in the LEFT and DOWN directions. Because LEFT and DOWN are the same distance from the food, the algorithm must select between the two directions. The algorithm may be skewed as a result of this decision. The algorithm chooses the first direction it sees in this version, which is LEFT in this example. A similar predicament emerges after the snake chooses to move left. When the snake examines the blocks RIGHT, LEFT, UP, and DOWN, it realizes that it can only go LEFT, UP, or DOWN. Because all three paths have the same *manhattan distance* from the fruit, the algorithm must make an arbitrary decision once again. The snake moves left in this version because it chooses the first object it sees. This cycle continues until the snake slams into the wall.

3.1.3. Hamiltonian cycles

We can develop a bot that is guaranteed to score a perfect game if algorithm runtime isn't an issue. Create a cyclic path that passes through all of the squares on the board without crossing itself (this is known as a Hamiltonian Cycle), and then keep following it until the snake's head is as long as the entire path. This method will always work, but it is tedious and wastes a lot of moves. It will take N^2 food to develop a tail long enough to fill the board in a NxN grid. If the food emerges at random, the snake will need to travel through half of the currently open squares to reach the food from its current position, which will take roughly $N^2/2$ moves at the start of the game. Given that this number lowers as the snake becomes longer, we estimate that the snake will require $N^4/4$ moves on average to win the game using this method. For a 20x20 board, this equates to around 40,000 movements.

3.2. Genetic Algorithm

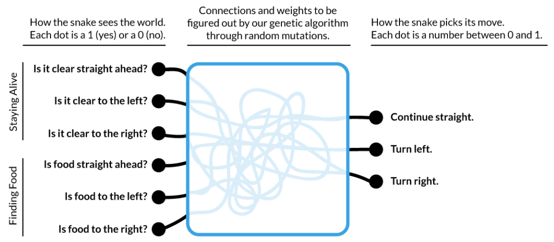
A genetic algorithm is a meta-heuristic inspired by natural selection and part of the larger category of evolutionary algorithms (EA). These algorithms are frequently used to find high-quality solutions to optimization and search problems by replicating natural selection, which states that only organisms that can adapt to changes in their environment will survive, reproduce, and pass on to the next generation. Each generation is made up of a population of individuals, in this case neural networks, each of which has its unique weights and hence represents a different solution. The snake's distance from obstacles in the four primary directions could be an input (up, down, left, right). An action, such as turning left or right, would be the output. Considering the analogy between genetic algorithms and natural selection, we can relate the weights of each individual to the genetic structure of a chromosome, which mutates generation by generation in order to give birth to better individuals. Generating a population of randomly produced chromosomes, each of which is a bit string containing only 0 or 1 values. These bit sequences are then decoded methodically into the parameters that are being learned, such as the weights for the neural network in this project. Fitness Calculations, Selection, Crossover, and Mutation are the four processes that the population goes through to replicate natural selection.

Brief description of Fitness Calculations, Selection, Crossover, and Mutation:

Fitness Calculation allows each chromosome in the population to play the game, and then assigns a score to each chromosome based on a specified fitness function and the fitness function determines how well a certain chromosome performed for the agent. In the selection process after all of a population's fitness ratings have been obtained, two chromosomes are (typically) picked for crossover in a "roulette-wheel" way. In other words, better fitness chromosomes are more likely to be chosen for crossover. Crossover is the process of producing new chromosomes that will be utilized in the next generation's population. To keep the population from reaching a local maximum, mutation is required. To make a kid chromosome, some pieces from each of the two chromosomes chosen in selection are combined. Each bit on each new kid chromosome has a possibility of being "mutated," that is, switched from 0 to 1 or 1 to 0.

An high level generalization of the workflow of the algorithm can be summarized in the following 6 steps:

1. Set the structure of the network that represents an individual.
2. Fill the weight matrices of the population of individuals with random numbers between -1 and 1.
3. Set the fitness function, which is a function that allows you to calculate each individual's performance.
4. Play a game for each person in the group and calculate their fitness function score.
5. Select a few top people from the population and use crossover and mutation techniques to develop the next generation from these top individuals. The algorithm works by exploiting these parents to produce a new offspring that outperforms each of them, culminating in a generation that is better suited to their surroundings.
6. Return to step 4 and repeat the process until the stopping criteria are not met.

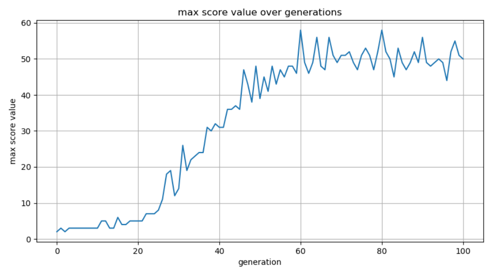

How the snake sees the world.
Each dot is a 1 (yes) or a 0 (no).
Connections and weights to be figured out by our genetic algorithm through random mutations.
How the snake picks its move.
Each dot is a number between 0 and 1.
Staying Alive
Is it clear straight ahead?
Is it clear to the left?
Is it clear to the right?
Finding Food
Is food straight ahead?
Is food to the left?
Is food to the right?
Continue straight.
Turn left.
Turn right.

The structure for input and output layers of the neural network for memorization.

We have made several copies of the neural net and let each one try to move the snake around. Those who excel have established a link between some component of the input and output data, which keeps it alive. We kept tweaking the ones that obtained the highest scores over time, and ultimately we had an AI that can play Snake.

Diving into the algorithm, the genetic algorithm was performed multiple times using various fitness factors. The initial fitness function was *((score^3)*(frame score))* . Score equals the snake's length minus 1 (because the snake always starts at length 1), and frame score is the number of frames the snake was alive. However, in order to maximize the frame score component of the fitness function, many snakes spun in circles constantly without consuming any food at first. As a result, the training was changed such that if a snake does not consume food in 50 frames, it is killed. In addition, if a snake died as a result of not eating any food for 50 frames, the frame score was reduced by 50 points to discourage the habit even further. The fitness function was then manually adjusted to ((score*2)^2)*(frame score^1.5) after 430 generations. This function was utilized to persuade the snakes to prioritize survival over obtaining the food, as they had reached a point where they were frequently hitting their own bodies while attempting to consume food. Selection was done in a conventional "roulette-wheel" method, with two chromosomes chosen with a probability proportional to fitness scores to form each kid chromosome. Single-point crossover was used to implement crossover. To make the child chromosome, a single location is randomly chosen to partition the bits between each parent chromosome. Each bit was mutated at a rate of.008 per second. It's also worth noting that the selection, crossover, and mutation processes only produced half of the chromosomes in each new population, with the other half being the half of the previous population with the best fitness ratings. The (n+n) technique is the name for this approach. Because it avoids the best chromosomes from being lost due to random crossover and mutation, the (n+n) technique significantly increased the performance of the genetic algorithm. A fascinating detail was that each weight in a chromosome utilized 8 bits, resulting in a total length of 2032 bits for a single chromosome.

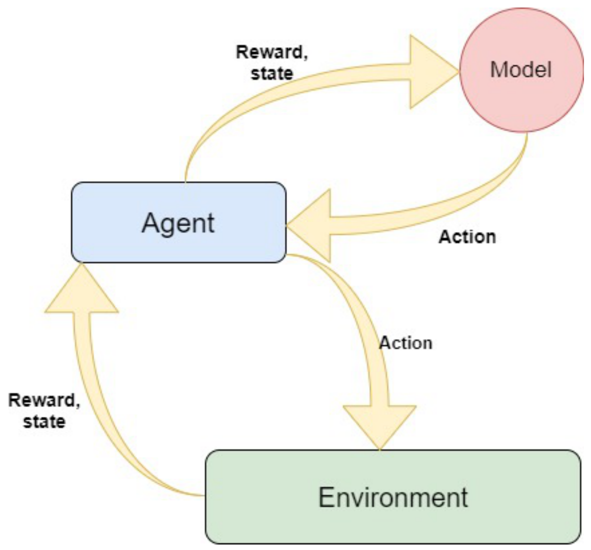
While training the agents we acquire an average fitness value of 185275 points over the last generation, an average number of deaths of 18.7, and an average score of 15.6 after running the genetic algorithm over 100 generations. By the end the snake has a fitness value of 244850 points, and a maximum score of 55.



After all observations, we can see that the generations stopped improving after generation 60. Nevertheless, a better fitness function and a more thorough representation of the status of the environment might drive the snakes' performance even higher, eventually enhancing it over generations.

3.3. Reinforcement learning

Reinforcement learning is a rapidly developing and interesting topic of artificial intelligence. At its most basic level, reinforcement learning entails an agent, an environment, a set of actions that the agent can perform, and a reward function that rewards good behavior and punishes bad behavior. The agent adjusts its parameters as it explores the environment in order to maximize its own predicted return. The agent in our case i.e., The snake game is, of course, the snake. The NxN board is the environment (with many possible states of this environment depending on where the food and the snake are located). Turn left, turn right, or keep going straight are all actions. *Deep Reinforcement Learning (DRL)* is a type of reinforcement learning that combines the reinforcement learning concepts with deep neural networks. Recently, DRL has been used to create superhuman chess and Go systems, learn to play Atari games using only the pixels on the screen as input, and control robots. We applied a simplified version of the reinforcement algorithm i.e., *Deep Q-Learning* in our project to obtain the optimal scores for our AI snake bot.



3.3.1. Deep Q-Learning

Deep Q-Learning is a subset of Deep Reinforcement Learning. The neural network learns the "Q function," which takes the current state of the environment as an input and returns a vector of expected rewards for each conceivable action. Following that, the agent can choose the action that maximizes the Q function. The game then changes the surroundings and offers a reward (e.g. +10 for eating the food, -10 for hitting a wall) based on this activity. The Q function is simply approximated by a randomly initialized neural network at the start of training. So to simplify Q-Learning controls and make decisions in an environment using an MDP (Markov Decision Process). It is made up of a Q-Table that is updated on a regular basis. A Q-Table is a matrix containing a collection of states and the chance of each action succeeding. The table is updated when the agent explores the area. The optimal action to do is the one that has the greatest value.

STATE S	UP	DOWN	RIGHT	LEFT
1	0.53	0.25	-0.15	0.89
2	-0.51	0.31	0.67	0.25
3	0.85	0.87	0.26	-0.15
4	0.75	-0.32	-0.12	0.48
. . .	. . .	. . .	. . .	. . .

Example of Q table with highlighted actions of each state

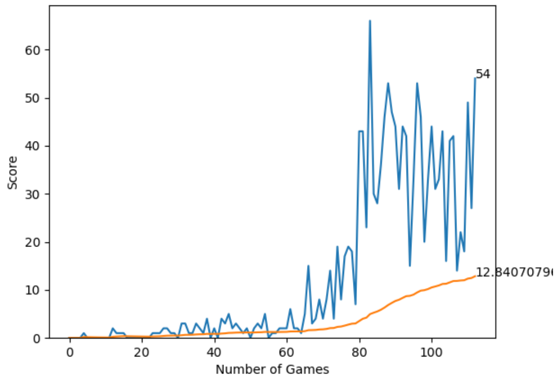
The algorithm's purpose is to use the *Bellman equation* to update the Q-values. This equation is used to guide the neural network in the proper direction by providing an approximation for Q. The Bellman Equation's output improves as the network improves.

$$Q_{new}(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma max Q'(s',a') - Q(s,a)]$$

Where Qnew(s,a) is the new value of the Q-table for the state s and action a, α is the learning rate, γ is the discount factor, R(s,a) is the reward and Q'(s‘,a‘) is the maximum predicted reward given the new state s‘ and all the possible actions a‘. A Deep Q-network is used to calculate the values Q(s,a). The network has a learning rate of 0.0005 and is made up of three dense layers with 120 neurons each, followed by three dropout layers to improve generalization and decrease overfitting. The output is a softmax layer with four neurons that outputs a probability value for each potential direction. The state s is a binary array of 12 values that represents the environment in a simple manner. It considers: if there is a hazard on the left, right, up, and down in relation to the position of the snake's head (4 values); the snake's direction (4 values); and the relative location of the food with respect to the snake's head (4 values) (4 values). As a result, the network's input layer is made up of 12 neurons that are ready to take the state s as input. The loss is defined as a minimal squared error function:

$$L = (R(s,a) + \gamma max Q'(s',a') - Q(s,a))^2$$

As a result, the training process tries to reduce the difference between the true target value and the predicted one in order to minimize the loss. The prize is calculated as follows: +10 if the snake has eaten a food; -10 if the snake has collided with a wall or with itself. Thus, the reward is defined to determine the behavior of our snake. Adding a modest reward for each step, for example, could make the snake more cautious at the risk of causing it to loop in a circle without eating food. A higher reward for consuming food, on the other hand, may make the snake more aggressive, causing it to focus on rushing toward the food, increasing the chance of colliding with the environment's barriers. By the end of the algorithm following the snake's movements, the system collects the value of the new state s' and runs a learning step to update the network's weights using the previously defined loss function, thereby updating the virtual Q-table. The system additionally computes and records the tuple formed by the old state s, the action a, the reward r, the new state s', and a boolean variable indicating whether the snake has perished or not in the replay memory during this phase. When the snake dies, it enters a learning phase in which it samples random tuples from the replay memory to allow the agent to learn from its previous actions. The Ɛ parameter set to 0 during testing to prevent the agent from performing random activities. It had a high score of 54 points in a set of 100 games.



Nevertheless, the agent has been trained using a simplified version of the Deep Q-learning technique. More training episodes and a more complete encoding of the states representing the entire environment would eventually enhance the performance and abilities of our snake.

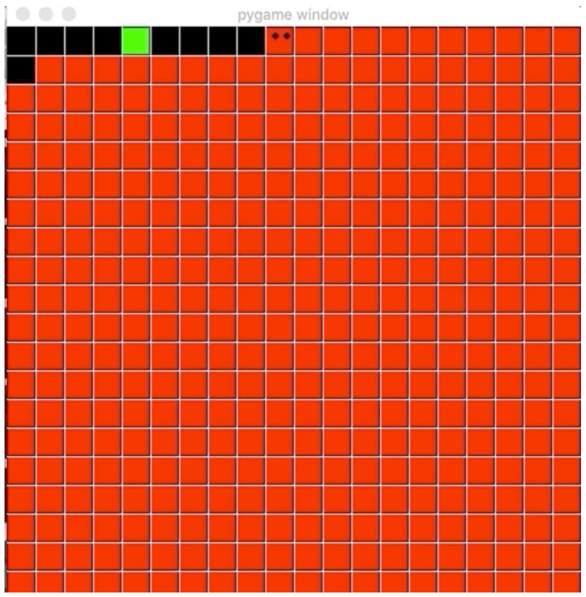
4. Exploring the Results

In this section we are going to showcase the optimal outputs we achieved using each algorithm in this project.

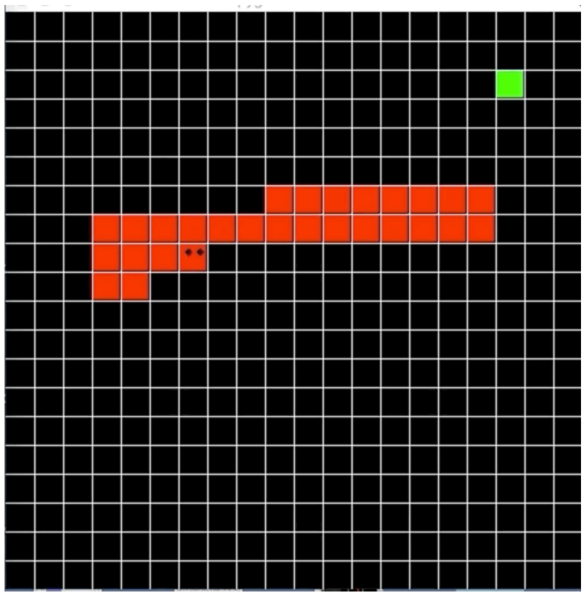
ALGORITHM	SCORES
Random Approach	1-2
Shortest Path	14-15
Hamiltonian Cycle	399
Genetic Algorithm	55
Deep Q-Learning	54

The table shows all the high scores we achieved after 100 iterations

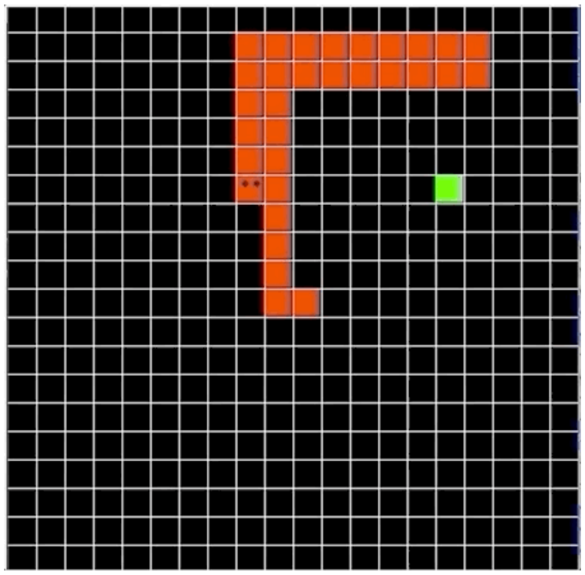
To wrap things up, we explored various deterministic techniques to create snake-playing machines. The random algorithm is completely ineffective. When compared to a random algorithm using a sophisticated shortest path method is quick and produces some really good results. The optimal algorithm is based on hamiltonian cycles, and it is certain to win (but is slow). Moving ahead, the Genetic algorithm heavily relies on chance, thus we may need to train many generations before obtaining a good one. Once the model is trained, predicting the next move is much easier with this algorithm but because mutations are unpredictable, convergence can be slow and the model's performance is determined by the inputs available to it. If the inputs simply specify whether impediments are present in the snake's local proximity, the snake will be unaware of the "big picture" and will become trapped inside its own tail. Like in the case of the genetic algorithm, in Deep Q-Learning the model's performance is determined by the number of inputs available to the network, and more inputs equal more model parameters, which means more time to train. Thus, even though non ML technique i.e., Hamiltonian cycles guaranteed to reach the best scores, there is no machine learning involved i.e., The algorithm must be manually encoded. It's only helpful if we're familiar with graph theory. For huge game boards, it may be slow. We added a few GIF's from our executions but we weren't able to capture most of the maneuvers in the GIF.

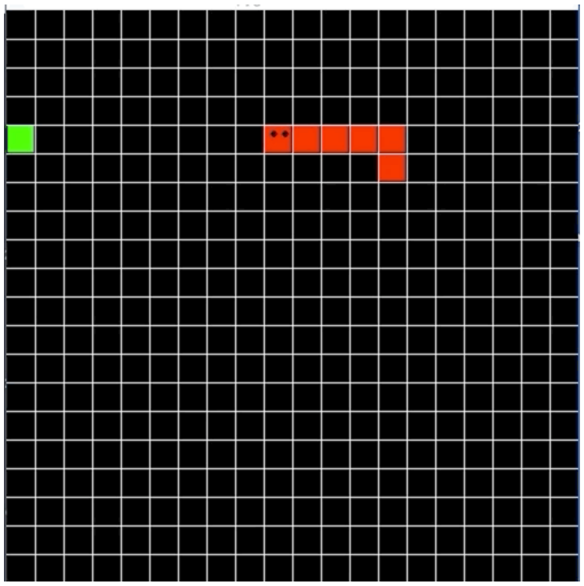


Hamiltonian cycles



Genetic Algorithm


Deep Q-Learning


Shortest Path

5. Conclusion

As a result, after studying and analyzing the concepts, we have come to the conclusion that in order for AI Bot to earn the highest possible score we can use the deterministic algorithm but it has its cons. For genetic and Deep Q-Learning algorithms efficient scores depend on the number of input and how efficient the model is trained to give the enhanced and optimal scores.

6. Future Work

Furthermore, the Deep Q-learning algorithm, when updated with features introduced in the Rainbow algorithm, such as Double and Dueling Deep Q-networks or Multi-step learning, can boost the agent's performance significantly. Also this training Bot's performance in the snake game could pave the way for future training Bots in larger games. This could be the way things go in the future for effective gaming and training.

7. References

[1] *Machine Learning for Humans, Reinforcement Learning* - *https://medium.com/machine-learning-for-humans/reinforcement-learning-6eacf258b265*

[2] *Genetic Algorithms geeksforgeeks* - *https://www.geeksforgeeks.org/genetic-algorithms*

[3] *Solving the Classic Snake Game Using AI* - *https://ieeexplore-ieee-org.proxy.lib.pdx.edu/stamp/stamp.jsp?tp=&arnumber=9105796*

[4] *Hamiltonian path* - *https://en.wikipedia.org/wiki/Hamiltonian_path*

[5] *Bellman equation* - *https://en.wikipedia.org/wiki/Bellman_equation*

[6] *Implementing snake game* - *https://www.edureka.co/blog/snake-game-with-pygame/*