Snake AI: Quantitative comparison of algorithms

CS7IS2 Project (2021/2022)

Chinmay Rane, Pratush Pandita, Omkar Pramod Padir, Udisha Chaudhary, Pratik Pathak

ranec@tcd.ie, panditap@tcd.ie, padiro@tcd.ie, chaudhau@tcd.ie, pathakp@tcd.ie

Abstract. The snake game serves as a benchmark for testing the performance of several AI algorithms developed for complex environments. This brings to our motivation behind picking the game. We evaluate the performance of four popular AI algorithms on a 8×8 grid and compare it with a strong baseline. We show that when the number of moves is not an issue, the baseline (Hamiltonain cycle) yields the best results. For reinforcement learning, the performance gradually saturates to the point that increasing the episodes provide no significant benefit. Alternatively, if we need a balance between the number of steps taken and the maximum score achieved, we recommend A* with forward checking algorithm since it finds the shortest path and aids the snake in surviving for longer duration.

Keywords: Snake, Path finding, Breadth first search, A* search, forward checking, Genetic algorithm, Reinforcement learning, Hamilton cycles

1 Introduction

Snake is a classic action game developed in 1976 which comprises of two elements: the snake and its food. Both the elements are put in a 2D confined space. The goal of the snake is to constantly consume food without hitting the boundary or its own body. At every step, the snake has to choose one action among: (i) STRAIGHT, (ii) LEFT, and (iii) RIGHT. The food coordinates remain constant till the time the snake consumes it. Then, it gets randomly generated in the open space within the 2D grid. The moment the snake eats the food, its length increases by one along with the game score. Note that initially, the snake begins with a length of two. The game ends when the snake either collides with its own body or the boundary. It wins the game when its body covers the entire grid with no free spaces remaining.

The motivation behind picking the snake game for evaluating AI algorithms is its relatively straight-forward implementation and rules. The game serves as a benchmark for testing the performance of several AI algorithms developed for complex environments. The game is a classic example of markov decision process

and therefore the knowledge gained by algorithms can be transferred to other related problems. In this paper, we evaluate the performance of four different family of AI algorithms: uninformed search, informed search, genetic algorithm and reinforcement learning. We assess the algorithms based on the number of steps taken, the number of food eaten and time taken to run each algorithm. We used the implementation of the snake environment provided by the OpenAI Gym library[1].

The rest of the paper is structured as follows. First, we conduct a literature review in Section 2, followed by problem definition and algorithm discussion in Section 3. In Section 4, we evaluate the performance of the four algorithms and compare them to the baseline approach. Section 5 discusses the open questions in the field, and future directions. Finally, we conclude the paper in Section 6. The recording of our presentation can be found in the link below[2].

2 Related Work

Appaji [1] compared the performance of several search-based algorithms including Breadth First Search (BFS), Depth First Search (DFS), Best First search, A* search, and Hamiltonian search against a human agent. DFS took the longest time to score 75 points followed by Hamiltonian search and then the human agent. Out of all the search algorithms the author tested, A* search emerged to be a clear winner in terms of achieving the maximum score in the least time. In fact, it even succeeded in surpassing the human performance. The reason is, it uses the Manhattan distance as heuristic function, thus helping the agent take the shortest route to the food.

Applying evolutionary algorithms to automate the snake movement is not uncommon. Works [2], [8] use genetic algorithms to generate optimal movements in every scenario. These algorithms teach the snake to survive rather than find shortest path to food and then die due to absence of path. Yeh et al. [2] incorporates this in a smart way. They compute a weighted sum of four movement-related functions and pick the action which lead to maximum score. They use an evolutionary algorithm to optimize the weights of the linear model. Their algorithm makes informed decisions in case of a known environment but have difficulty navigating complex unknown environments. This is expected because they use a lot of information about the environment to choose an action at every step and therefore doesn't work well in unknown environments.

Reinforcement Learning (RL) algorithms have been pervasively adopted and applied to a variety of games following the breakthrough in combining deep learning with RL pioneered by DeepMind to play Atari games [3], [5]. Works [3], [7] use a deep reinforcement learning algorithm along with experience replay to

[1] https://github.com/grantsrb/Gym-Snake
[2] Presentation recording

optimally train the neural network. The experience replay strategy reduces the correlation of sampled experiences during the network training. This helps the agent increase environment exploration.

Another problem often faced by RL community is the sparsity of rewards in a large grid leading to slow and ineffective learning. Cai et al. [7] addresses the issue by introducing a reward mechanism dependant on distance to food to generate immediate rewards, thus avoiding sparseness. Lim et al. [4] proposes the use of curriculum learning to accelerate the training of AI agents. The idea is to start with a small grid and increase the grid complexity gradually. The author shows that the approach works better than naive approach for large grid sizes.

Based on the literature survey, we implement four algorithms - uniformed search, informed search using forward checking, genetic algorithm and reinforcement learning. The specifics of these algorithms are detailed in the section below.

3 Problem Definition and Algorithm

3.1 BFS Search

We used the implementation of BFS as detailed in [1]. Breadth-first search algorithm is ideal because it always returns the minimum number of actions needed to get to the goal. BFS does this by using a first-in-first-out queue as the fringe to keep track of states to be explored next in the state space graph. Since FIFO technique pops out the first element that was pushed into the queue, shallowest states are explored first and then repeats the same for their successors until the goal state which is the food is reached. However BFS is an uninformed search algorithm implying that it explores every state it comes across and therefore it exhaustively searches the state space graph layer by layer.

For any given state, all 4 actions (UP, DOWN, RIGHT, LEFT) are applied to move to the new states also called successors which are pushed to the queue. It is the FIFO nature of the queue that decides the order of exploring the next states in order to explore those states that were pushed first, otherwise it would not be optimal. A working of BFS is shown in Fig.1. BFS stops working when it either hits itself or the grid walls.

3.2 A* search with Forward Checking

For the implementation of the A* algorithm with forward checking we referred to the work of [9]. The idea behind forward checking is that the snake simply follows its tail in the absence of path to the food. It was observed that as the snake's body grows after eating food several times, it eventually traps inside its own body and bites itself. Any simple A* implementation would not solve the problem of the snake ending up biting itself because just getting the food is

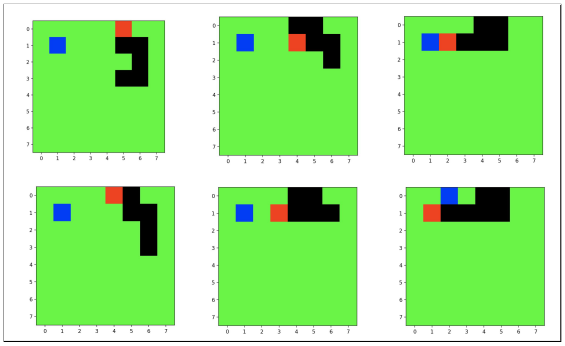

Fig. 1. Breadth-First Search

not enough, the snake needs to be aware of the surroundings. One technique to resolve this includes a kind of forward checking which makes the snake aware of the consequences of eating the food beforehand. So, an additional functionality called 'follow its tail' [3] is added according to which if there is a certain path to the food, the snake will be made to follow that path only if there exists a path that can take it towards the end of it's tail afterwards. Unless this condition is satisfied it will not follow food but only it's tail instead. Even though there would not be a path to the food, the snake still has a goal to stay alive and also avoid hitting the grid wall by following it's tail.

Pseudocode of forward checking

1. Using A* check whether there is a path to the food.
2. If a path exists :
   (a) Generate a virtual snake which will be an exact copy of the original snake.
   (b) Make the virtual snake follow the path to get to the food.
   (c) After eating the food, check whether there is a path to its tail.
   (d) If such a path exists then make the original snake follow the path that virtual snake discovered to the food. If the path does not exist then go to step 3
3. Make the original snake follow it's tail.

Whether the goal is to find a path to food or tail, search is need to be performed for which A* is selected. A* is an improvement from BFS because it is an informed search. Information is in the form of heuristics plus cost of the path. Heuristic h(n) is a function that returns how far the state is from the goal. Manhattan distance is adopted as the heuristics here. It calculates the absolute difference between two states in the grid which is basically the minimum distance between two states and thus it helps in maintaining admissibility since it will never overestimate the actual cost to reach the goal state. Cost of the path g(n)

[3] Tail escape implementation

is the number of pixels used to reach a state. In our implementation each pixel carries a cost of 1. This kind of informed search not only returns the shortest path but also prevents exhaustively exploring every neighbouring state.

As the problem is a state space graph search, heuristics have to maintain consistency as well. Consistency means the heuristic difference between two states is less than or equal to the step cost between the two states. Having manhattan distance as heuristics and cost of each step as 1, consistency is ensured because manhattan always returns the shortest distance between two states so that the heuristics never overestimate the step cost.

To implement A* search algorithm priority queue is used as the fringe where each state is pushed along with a priority which is h(n) + g(n). Priority queue pops out the state that has minimum overall cost and therefore only the states that are closer to the goal are explored first rather than exhaustively exploring every neighbouring state for example in BFS. A working representation of forward checking using A* algorithm is illustrated in Fig. 2.

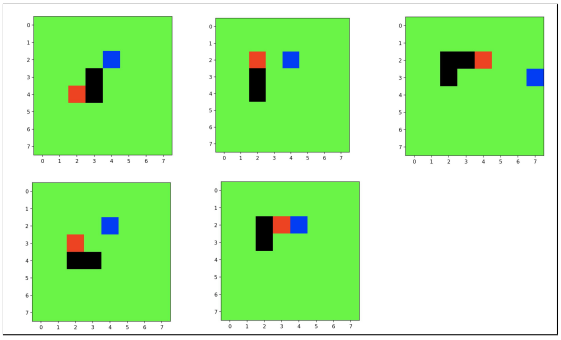

Fig. 2. A* with forward checking

3.3 Q-Learning

For the implementation of reinforcement learning we refer to the work of [6]. The author performs q-learning and defines a set of five features as the snake's state (state space of 128). Now, in the next paragraph, we briefly describe the algorithm.

Reinforcement learning is a family of algorithms where a problem is solved by using information of states 's' and actions 'a' that can be performed at a particular state. Q-learning is a form of active Reinforcement Learning method, where the agent gets experience from the environment by performing different actions at different states. Each state action pair results in a reward that is stored in a tabular structure known as the Q-Table. The values in this table are

calculated and updated with the help of bellman equation given by:

$$Q_{new}(s,a) = (1-\alpha)[Q_{prev}(s,a)] + \alpha[R(s,a,s') + \gamma \max_{a'} Q(s',a')] \quad (1)$$

In equation 1 the Q-value of a state action pair (s,a) depends on: $Q_{prev}(s,a)$, the previous Q-value of the state action pair. $R(s,a,s')$: Reward obtained after performing action a on state s. $\max_{a'} Q(s',a')$: Maximum Q-value that can be obtained from next step s' after checking all the actions possible from s'. Alpha (α) is the learning rate parameter between $(0 < \alpha \leq 1)$ that determines how fast these q-values will be updated. γ: Gamma is the discount rate applied to help the agent choose between immediate vs future rewards. The decision of exploration or exploitation is taken based on the value of parameter epsilon $(0 \leq \epsilon \leq 1)$ also known as exploration rate. Value 0 represents always exploiting (use previous Qvalues) whereas 1 represents continuously exploring new actions.

3.3.1 Implementation details Q-learning implementation works in two stages. First we train the agent by exploring different possible states and performing actions from those states and continuously update the Q table for given state action combinations. In the second stage we test and evaluate the performance of the agent by using the Q table values recorded in the training stage.

Parameters used:

- $\alpha = 0.5$ to balance tradeoff between previous and new learned q-values.
- $\gamma = 0.9$ agent tries to take a short path to reach food.
- $\epsilon = 0.1$ probability of exploring new actions randomly.

Rewards used:

- Food reward (100): Increases q-value so that agent learns to eat food.
- Death reward (-1000): Decreases q-value when the snake dies, so it forces agent to not die.
- Living reward (0): No penalty for staying alive. (Indirectly handled by γ).

Features. In an ideal implementation of q-learning all the possible information i.e., entire grid with the snake, food and empty positions should be used as the state. But this state information becomes too large due to several possible combinations. It is infeasible to explore these many states and record its q-values appropriately. So, to make things easier the state has been reduced to 10 different parameters, eight of which take only binary values 0 & 1. This helps to reduce state combinations while also retaining important information that will be useful in making decisions.

The state of the game can be given by different combinations of features[4]. Fig. 3 above compares different features used as states while training the q-learning agent. We can see that when just coordinates of the snake are used (green) the

[4] Deep reinforcement learning on snake

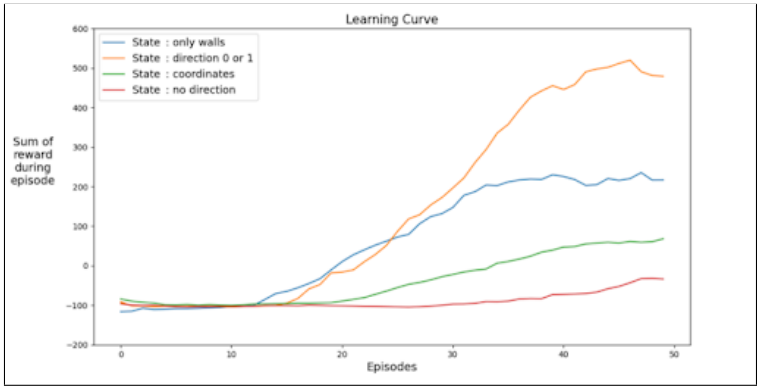

Fig. 3. Learning with different feature sets

learning is pretty slow. After adding information about walls & obstacles (blue), the learning improves. Best learning performance is achieved when the direction information (orange) is also added to above two feature types. This helps the agent to learn more in fewer episodes. Therefore, the feature set corresponding to orange line is used for the agent training and is detailed in Table 1.

Table 1. Feature set used for agent training.

Features	Values
Is food direction up?	[0,1]
Is food direction down?	[0,1]
Is the food direction right?	[0,1]
Is the food direction left?	[0,1]
Is the block above an obstacle?	[0,1]
Is the block below an obstacle?	[0,1]
Is the block to the right an obstacle?	[0,1]
Is the block to the left an obstacle?	[0,1]
Direction of movement of snake	UP, DOWN, RIGHT, LEFT [0,1,2,3]
Manhattan distance from snake head to food	1,2,3,...

3.4 Genetic Algorithm

Evolutionary algorithms are a heuristic-based method to solve problems inspired by the concepts involved in biological evolution. One such algorithm that reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation is the genetic algorithm. We refer to the work of [2] for implementation of the algorithm.

Initially, we randomly generate a population of candidate solutions called chromosomes and this is the first generation using which the evolution happens. The next step is calculate the fitness of each chromosome in the population.

For this, the chromosomes perform some actions in the real world environment based on the goal and we evaluate the fitness of each candidate based on a fitness function appropriate to the domain of the task. Once we have the fitness value, we choose the best candidates to create a pool of mating parents for the next generation. This parent population form the basis for producing offsprings of the next generation using operations such as crossover and mutation. During crossover, two parents are randomly selected from the mating pool and their genetic information is swapped to produce a child with the same size as the parents. Once we have a child population created from mating pool, we introduce some new genetic material in the population through mutation. This means we randomly change the information in the childrens hoping to discover a more fit population than parents in the process. This step could be compared to the exploration step in the Qlearning algorithm. The resulting population represents the next generation of childrens and replace the old parents based on their fitness value.

One important factor to consider is the termination criteria which can be based on the number of generations evolved. Another alternative is to determine a desired fitness level for the population and stop once this fitness has been reached. The resulting population will be the most fittest generation and will be returned as overall best candidate solutions.

3.4.1 Implementation details Genetic algorithm is implemented[5] by following the evolution process described earlier. In this section we go through the important design consideration involved in each step of the algorithm. Let's first look at the candidate of solutions[6] possible for the snake game. The snake can move into four directions - UP (0), DOWN (1), LEFT (2), RIGHT (3) and we also have a fifth candidate (4) that decides the move based on the distance of the snake head to food. Based on these candidates and the population size we generate an initial population randomly.

Parameters used:

- The number of chromosomes in each generation (Population size) = 100
- The number of evolutionary generations (Generations) = 100
- The number of best individuals chosen for mating (Mating pool size) = 12

The most critical part of the algorithm is the fitness function used to determine the fitness value of chromosomes during each generation of evolution. This affects the selection of mating population such that the probability of the fittest individuals to survive to the next generation is high. This will also help to eventually determine the fitness of the last generation of the population for terminate criteria.

Fitness function used:

[5] Genetic algorithm + Neural network for snake game
[6] Solving snake game using only genetic algorithm

- Food reward = 500
- Death penalty = -1000
- Snake length at the end of the game = 5000 * len

Pseudocode of genetic algorithm

1. Generate initial population using parameters described above.
2. For each generation:
   (a) Calculate the fitness of the population by playing the snake game.
   (b) Sort the population by fitness value of each individual and select the best candidates for the mating pool.
   (c) Generate the new population by using crossover and mutation.
3. Return the fittest population as best performing solution.

3.5 Hamiltonian Cycle (Baseline)

Hamiltonian path is a brute-force strategy where the snake visits each state of the graph exactly once and Hamiltonian cycle is a special kind of Hamilton path that has the start node connected to the end node. Due to this cyclic nature, the snake follows the same path each time and reaches the goal eventually without hitting its own body. As mentioned above it is a brute-force strategy which follows the same path every time and hence it is suitable as a baseline algorithm. A working representation of Hamiltonian Cycle is shown in Fig. 4. It can be seen that no matter where the food is, the snake always traverses the same entire path and thus it is not so efficient when the size of the snake is too small because of all the extra moves.

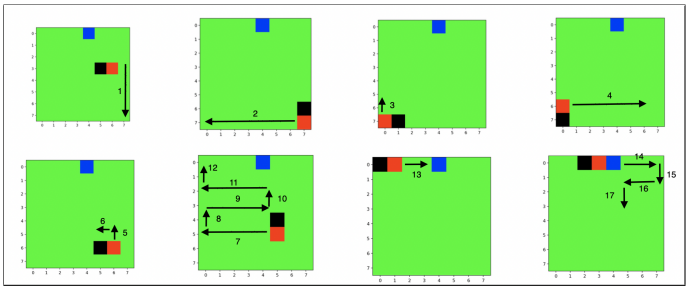

Fig. 4. Hamiltonian Cycle

4 Experimental Results

For evaluation, 100 test runs were performed for each algorithm in order to consider variances due random generation of food as well as different starting position of the snake. Here one test run corresponds to the entire gameplay until

the snake dies. The grid was limited to 8x8 space for all the evaluations. The metrics chosen for evaluation are the count of food consumed in each run and average number of steps taken to consume each food particle.

This evaluation was performed on a Windows 10 laptop with configurations of CPU: Intel Core i7-6700 HQ, RAM: 16 GB DDR3 2400 mhz, without any GPU based acceleration. Code was executed in python 3.6 environment on PyCharm IDE.

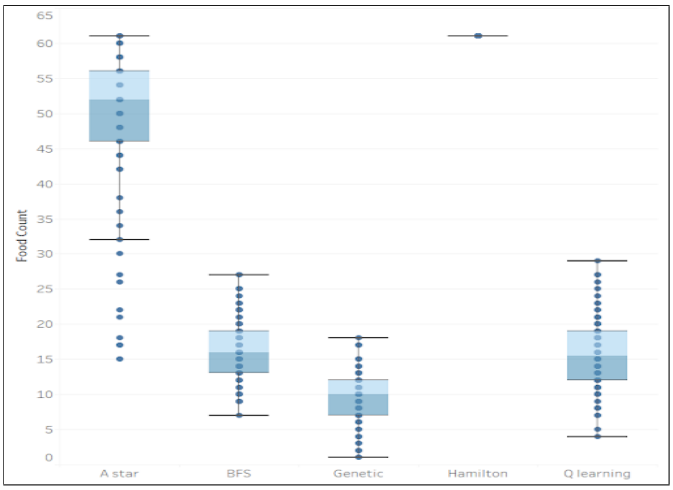

Fig. 5. Count of food eaten by snake in every algorithm

4.1 Comparison based on Food Consumption

As seen in Fig. 5 the baseline agent, hamiltonian path is the best performer when just the food count is considered. This is because the snake is actually trying to cover all the grid points in a cycle and the food will always be in one of the grid points. Thus we can see that the scores are extremely high and also constant at around 61. The next best performer is A star algorithm which uses tail escape mechanism as additional information in order to escape death which increases its scores.

A simple BFS works well with a median score of 16. One important thing to note is that, in these search algorithms the snake agent is allowed only those actions which do not cause it to hit the boundary or its own body. This is not the case for Q learning and Genetic algorithms. In these algorithms we have allowed all four actions and the agent is expected to learn on its own which actions will lead to its death. This is one of the reasons why Q learning and Genetic scores might seem to be lower, but they are performing well considering the actions they take are not pre-filtered.

4.2 Comparison based on steps taken to consume food

Another important metric for comparison is the average amount of steps taken by the agent to consume one food particle. A better agent will take a smaller number of steps to reach food. In Fig. 6, we can clearly see that in this case the baseline hamiltonian agent is performing the worst as it is never trying to get the shortest path. A star algorithm due to the forward checking mechanism sometimes chooses a safer path which causes more steps per food consumed. The Genetic algorithm is also decent as the fitness function includes the amount of steps required for the food consumption. The Q learning algorithm tries to find a shorter path because a discount factor of 0.9 has been used while training. BFS also takes a short path since it explores the states in a breadth-first fashion.

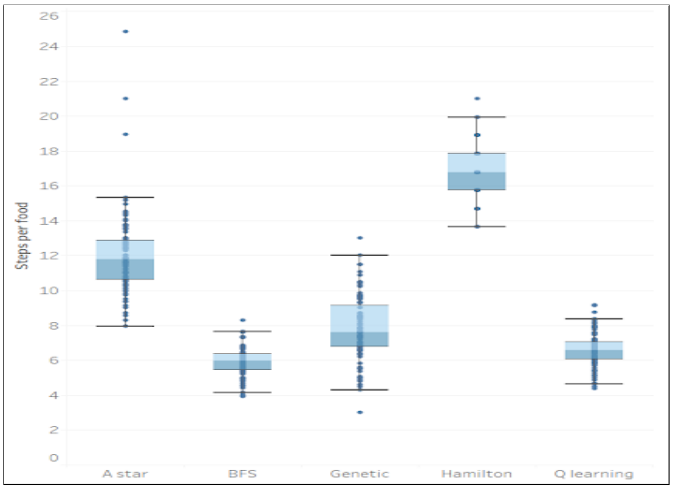

Fig. 6. Number of moves for each food particle consumed

Thus we can say that the baseline is only good if the steps are not being considered. A star offers a good balance between food consumption and steps taken. Q learning, genetic and BFS have smaller number of steps but the food count is also low.

5 Future Work

The snake-game we implemented was an instance of open gym AI where an 8×8 grid had one food item and one snake. This gaming environment can be extended to a larger grid say 30×30 having multiple snakes and food items making it a multiplayer game. Moreover, as we saw in A*, the snake solves the dilemma of hitting its own body but as a consequence it gets stuck in an infinite loop

and keeps on following the tail. Some work can be done here to overcome this problem as well. One way would be to combine A* and RL in such a way that RL takes up the control after running A* for a while.

6 Conclusion

In this paper, we assessed the performance of four different AI algorithms - BFS, A* search with forward checking, Genetic algorithm and Reinforcement learning and compared them with a strong baseline - Hamiltonian cycle. We observed that A* search with forward checking performs significantly better than others considering two evaluation metrics - number of food consumed and the number of moves taken for each food particle. The baseline gives near-perfect score, however, takes the most amount of moves. Coming to RL algorithms, there's a scope for improvement by incorporating Neural Networks (NN) for estimating the Q-values. Same applies to the genetic algorithm. Finally, we believe this work would be beneficial for future researchers in adopting appropriate AI algorithms for their applications and would also accelerate the development of AI algorithms to improve upon the flaws of discussed approaches.

References

1. Appaji, Naga Sai Dattu. "Comparison of Searching Algorithms in AI Against Human Agent in the Snake Game." (2020). URL www.diva-portal.org/smash/get/diva2:1474900/FULLTEXT02
2. Yeh, Jia-Fong, Pei-Hsiu Su, Shi-Heng Huang, and Tsung-Che Chiang. "Snake game AI: Movement rating functions and evolutionary algorithm-based optimization." In 2016 Conference on Technologies and Applications of Artificial Intelligence (TAAI), pp. 256-261. IEEE, 2016. doi: 10.1109/TAAI.2016.7880166. URL https://ieeexplore.ieee.org/document/7880166
3. Wei, Zhepei, Di Wang, Ming Zhang, Ah-Hwee Tan, Chunyan Miao, and You Zhou. "Autonomous agents in Snake game via deep reinforcement learning." In 2018 IEEE International Conference on Agents (ICA), pp. 20-25. IEEE, 2018. doi: 10.1109/AGENTS.2018.8460004. URL https://ieeexplore.ieee.org/document/8460004
4. Lim, Swee Kiat. "Curriculum learning with Snake." (2020). URL https://web.stanford.edu/class/aa228/reports/2020/final16.pdf
5. Finnson, Anton, and Molno¨, Victor. "Deep Reinforcement Learning for Snake." (2019). URL www.diva-portal.org/smash/get/diva2:1342302/FULLTEXT01.pdf
6. Ma, Bowei, Meng Tang, and Jun Zhang. "Exploration of Reinforcement Learning to SNAKE." (2016). URL https://cs229.stanford.edu/proj2016spr/report/060.pdf
7. Cai, Ruikai, and Condi Zhang. "Train a snake with reinforcement learning algorithms." (2020). URL https://openreview.net/pdf?id=iu2XOJ45cxo
8. Kumar, Ankit. "Automated Snake Game with Genetic Algorithm." (2021).
9. Sharma, Shubham, Saurabh Mishra, Nachiket Deodhar, Akshay Katageri, and Parth Sagar. "Solving the classic snake game using ai." In 2019 IEEE Pune Section International Conference (PuneCon), pp. 1-4. IEEE, 2019. doi: 10.1109/PuneCon46936.2019.9105796. URL https://ieeexplore.ieee.org/document/9105796