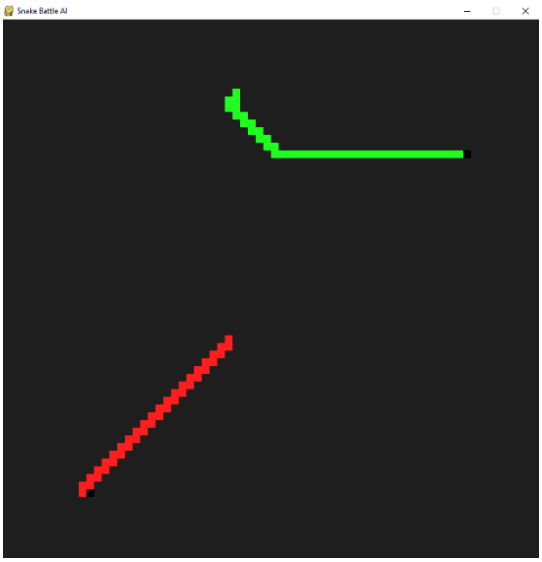
2 Player Snake AI

CSE 150 Extra Credit Report

Nikhilesh Sankaranarayanan (A91081886)

Abstract
This paper focuses on the implementation of an AI player for a modified version of the classic 'Snake' game. This is implemented using a simple depth-1 "cowardly" search AI, and then improved upon with a depth-3 adversarial search AI. The focus is on improving the win% of the player against other AI over multiple simulations.



1. Introduction

This version of Snake has been modified so that

- There is no 'candy' to eat
- There are 2 snakes on the board instead of one
- Each snake grows by length 1 each turn, and the tail end is fixed

Rules

The goal of the game is to survive for as long as possible without dying. A snake can die by bumping into the boundaries, the other snake, or itself. If one snake dies, the other snake is automatically the winner. If both snakes' heads bump into each other, the game ends in a Draw. Snakes start at a random location somewhere on the board, and start by facing upward.

2. Motivation

The idea for this game comes from my submission to the Webroot Hackathon conducted as UC San Diego in Winter 2018. The Snake game engine has been modified to resemble those conditions as closely as possible. Since that submission was made before learning about more advanced AI strategies in CSE150, it is an ideal candidate to improve upon and analyze the performance of these improvements. The primary motivation was the success of the 2048 AI from CSE150, which gave me the idea to try the same strategy on the 2-player Snake game.

3. The Game Engine

The game engine has been taken from the Snake Battle [1] project of GitHub user *Scriptim*. It has been modified to fit the original hackathon environment as closely as possible. The AI strategies work by reading in the current state of the board, and submitting the optimal move to make on each tick of the game loop. Experiments are run on multiple games by executing the entire game file, and storing the result of which player won. The code can be found at [2].

4. Strategies and Implementation

Random_ai

An AI Snake that just moves in a random path. Makes a random move each step and usually self-destructs in a few turns.

Simple_ai

I implemented a simple AI Player that always moves in a diagonal path toward the upper right corner. This is a good baseline against which to test the performance of the other AI.

Original_ai

This AI is a Depth-1 search that looks at the current board position, and attempts to move in the direction of least interference i.e. it runs away from walls and the other snake, and attempts to head toward the largest open space. This most closely resembles the original submission from the hackathon submission. Note that this AI does not consider the opponent's possible movements at all.

Adverse_ai

This improved AI runs a Depth-3 adversarial search that considers all 4 possible moves of the current (max) player, 4 possible moves of the opponent (min-player) and subsequent 4 final moves of the current player. This leads to 4 + 16 + 64 = **84** total game states to be considered.

---

[1] https://github.com/Scriptim/Snake-Battle

[2] https://github.com/Demonikki/Snake-Battle

This builds off of the Original_ai, but once one move in taken, 4 states for the opponent's move are created. The score is generated for all of those states and stored. Then, for each of **those** 4 moves, the player's head position is moved into 4 more states, and the scores for those are calculated. The final score for the original 4 states is calculated as

Max (1st level scores) + Min (2nd level scores) + Max (3rd level scores)

Therefore, this calculates the first move that contains the path that will minimize the opponent's score and maximize the player's score. Note that it does not necessarily **follow** this path – it only returns the first step that should be taken along that path. Subsequent steps are calculated from scratch again.

Calculating Score

Each AI aims to maximize the score of the board. The score is a measure of how far away from obstacles the player's head position is. Moving closer toward the opponent's tail, or the boundaries reduces the score and the likelihood of moving in that direction. Therefore, the score is calculated as the Sum of the squares of distances from obstacles. I square the distances to weigh more heavily toward directions that have more open space (i.e. run away from obstacles).

For the opponent (min player), we calculate the score the same way and pick the move that moves it closer to obstacles (lower score).

5. Results

Now for the spicy part. As a baseline, I first ensure that the Simple_ai performs as expected. I put it to the test against itself for 100 simulations. The results are completely as expected:

Player 1	Player 2	Win % of P1	Run Time
Simple_ai	Simple_ai	51%	4.87s

Next we consider performance of Original_ai against both the Simple_ai and a Pure Random Player.

Player 1	Player 2	Win % of P1	Run Time
Original_ai	Simple_ai	12%	3.15s

What's that? My AI performs abysmally! It was then that I realized my reward calculation needed to be improved. By not considering the snake's own tail, I ran into the problem of my Snake murdering itself by running into its tail (I thought this was an interesting case to observe). Upon updating it to keep the Snake away from all possible obstacles, we get the following results:

Against Diagonal Player (Simple_ai)

Trial 1 (100 simulations)

Player 1	Player 2	Win % of P1	Run Time
Original_ai	Simple_ai	71%	8.65s

Trial 2 (100 simulations)

Player 1	Player 2	Win % of P1	Run Time
Original_ai	Simple_ai	74%	8.96s

Trial 3 (500 simulations)

Player 1	Player 2	Win % of P1	#Wins P1	#Draws	Run Time
Original_ai	Simple_ai	75%	373	1	36.68s

Trial 4 (1000 simulations)

Player 1	Player 2	Win % of P1	#Wins P1	#Draws	Run Time
Original_ai	Simple_ai	71%	710	7	46.517s

Against Pure Random Player

Trial 1 (100 simulations)

Player 1	Player 2	Win % of P1	Run Time
Original_ai	Random_ai	99%	2.14s

Trial 2 (500 simulations)

Player 1	Player 2	Win % of P1	#Wins P1	#Draws	Run Time
Original_ai	Random_ai	99%	494	1	8.33s

Against Itself (Original_ai)

Trial 1 (100 simulations)

Player 1	Player 2	Win % of P1	Run Time
Original_ai	Original_ai	55%	8.24s

The results are slightly more favorable, with the Original_ai winning most of the time against the Simple_ai. The Original_ai blows the Random_ai out of the water. The result of Original_ai against itself is uninteresting as it depends purely on favorable starting position as both AIs perform similar actions.

The runtime performance is also quite reasonable. You can verify these results yourself by running the `experiments.py` file with appropriate parameters. To change the AI used, modify the AI objects created in `snakebattle.py` under the comment `#Create AI Objects`.

Now we run our 3 AI strategies against the improved version – the **Adverse_ai**, that runs Depth-3 Adversarial Search to find the path that will give the highest reward among all possible moves.

See **Strategies** for a detailed explanation of the working of the **Adverse_ai**.
Here are the results from my experiments on this AI.

Against Pure Random Player

Trial 1 (100 simulations)

Player 1	Player 2	Win % of P1	Run Time
Adverse_ai	Random_ai	100%	2.92s

Against Diagonal Player (Simple_ai)

Trial 1 (100 simulations)

Player 1	Player 2	Win % of P1	Run Time
Adverse_ai	Simple_ai	75%	10.93

Trial 2 (500 simulations)

Player 1	Player 2	Win % of P1	Run Time
Adverse_ai	Simple_ai	69%	49.84

Results are similar enough to the **Original_ai**.
Now, the **ULTIMATE** test – against my **Original_ai**. Let's see how much it improved.

Against Original_ai

Trial 1 (100 simulations)

Player 1	Player 2	Win % of P1	#Wins P1	#Draws	Run Time
Adverse_ai	Original_ai	63%	63	3	11.77s

Trial 2 (250 simulations)

Player 1	Player 2	Win % of P1	#Wins P1	#Draws	Run Time
Adverse_ai	Original_ai	64%	160	5	29.76s

Trial 3 (500 simulations)

Player 1	Player 2	Win % of P1	#Wins P1	#Draws	Run Time
Adverse_ai	Original_ai	64.8%	324	8	29.23s

Trial 4 (1000 simulations)

Player 1	Player 2	Win % of P1	#Wins P1	#Draws	Run Time
Adverse_ai	Original_ai	63.8%	638	20	115.27s

Trial 4 (5000 simulations!!!)

Player 1	Player 2	Win % of P1	#Wins P1	#Draws	Run Time
Adverse_ai	Original_ai	63.8%	3091	91	584.27s

```
('Time elapsed for ', 5000, ' simulations: ', 584.8940000534058)
('Player 1 won ', 3091, 'times')
('Player 2 won ', 1861, 'times')
('Draw happened ', 91, 'times')
PS C:\Users\nikhi\Documents\Code\CSE150 Extra Credit\Snake-Battle>
```

Observations:

- Adverse_ai runs a fair bit slower than Original_ai did vs Random_ai and Simple_ai.
- Similar performance against Simple_ai as seen by Original_ai (~70%).
- Outperforms Original_ai by a decent margin (wins roughly 2/3 of the time).
- There is a rare bug where Adverse_ai crashes into the top left corner, did not fix in time. Performance will certainly improve a few % points if fixed.
- It never, ever lost a game against the Random Player in my experiments.
- A few games were thrown out due to odd failures/both players instantly dying, etc.

6. Further Improvements

Apart from pruning techniques such as Depth Pruning (currently in use), the model could implement Alpha-beta pruning. However, I believe this adversarial search does not benefit significantly from that improvement as the number of states being considered is not large. In addition, moves that end in failure anytime along their path are automatically skipped over, and are not further evaluated.

Reinforcement Learning could potentially help improve the performance of the AI player, however. Similar to the Blackjack Assignment, it could be possible to evaluate a policy like “move in the direction of largest free space”. Computing the average reward of all states in that direction could tell us if that would eventually lead us to outlast the opponent.

7. Conclusion

The 2 strategies we implemented perform reasonably well against a "dumb" AI. The original implementation (Original_ai) can also be improved significantly by performing an adversarial search, treating the Player as a Max Player and the opponent as a Min Player (Adverse_ai).

This strategy works much better in terms of winrate, although the results are slower as it is more computationally intensive. Therefore, it is limited to a Depth-3 search. The improved AI does **not** perform significantly better against the hardcoded, diagonal-moving Simple_ai, but when put against a Rational Depth-1 AI it outmaneuvers it nearly 2/3 of the time. I can attribute this to simply a tricky pattern that is hard to beat (moving diagonally across the screen takes up a lot of space).

The Improved AI does run a few seconds slower – for the same number of simulations, it is around 10-20% slower than Original_ai against the Simple_ai and Random_ai. This can potentially be solved by better pruning techniques such as Alpha-Beta pruning.

In future I would like to extend this to support Monte-Carlo Learning and compare the results against the Adverse_ai – this is a nice task for Spring Break. In addition, it would be interesting to see how Human Players fair against these AI when run at a playable speed.

8. Credits

- GitHub user **Scriptim** for their Snake Game Engine **Snake-Battle.**
- My friend and Hackathon partner **Yimeng Yang** with whom I developed the original Original_ai during the Webroot Hackathon at UC San Diego.
- **Dr. Sicun Gao** for his lectures on Adversarial Search, and the opportunity to complete this very fun report.