CS221 Final Project: AI Agent for Snake feat. Mice

Vincent Chen {vschen}, John Kamalu {jkamalu}, Scott Kazmierowicz {scottkaz}

1 Introduction

In the original "Snake" game, a snake attempts to consume stationary "food" items without bumping into itself or a wall. In this project, we attempt to solve a modified version of Snake, in which the "food" items are "mice" with the ability to move.

2 Task Definition

2.1 Problem Statement

The fundamental problem in this modification of the Snake game is for the snake to eat as many mice in as few steps as possible. The maximum goal for this configuration of the snake game as used in this project is 30 mice.

Theoretically, it would be possible to achieve a maximum score by following a set path that sweeps across the game board systematically. To discourage this behavior, we have implemented a penalty for each step that the Snake takes.

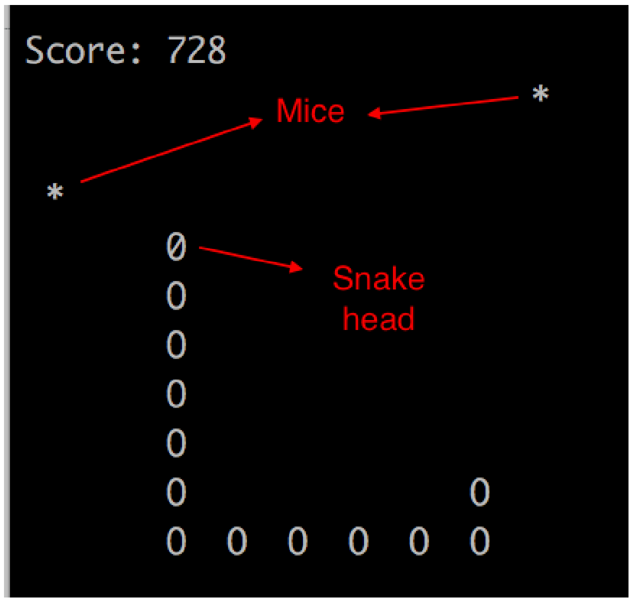


Figure 1: Graphics displaying a running instance of the modified Snake implementation

2.2 Gamestate

To define our problem, we will represent the world as a grid in which there exists a state for each 'tick' in time. The agent will only have access to the given game state at each point in time. For each of these states, the agent will attempt to take the optimal action.

The state of the world is updated after each tick (t) unit, which represents time. A state will be defined as the list of absolute mice locations and the absolute snake body positions at a given time. Each location is represented by (x, y) coordinates,

$$state_t = \left\{ \begin{array}{r} [(mx_1, my_1), (mx_2, my_2) \dots (mx_n, my_n)], \\ [(sx_1, sy_1), (sx_2, sy_2) \dots (sx_l, sy_l)] \end{array} \right\} \tag{1}$$

where (mx_i, my_i) are the coordinates for mouse i given n total mice, and (sx_i, sy_i) is the body position of the snake of length l

Given a particular state, our agent will choose an optimal action from the following choices:

$$actions = \{\text{north}, \text{south}, \text{east}, \text{east}\} \tag{2}$$

2.3 Input output behavior

In this example state, there exist three mice and one snake of length 3. Consider a potential action that arises from it:

$$state_8 = \left\{ \begin{array}{c} [(3,5),(2,2),(7,9)], \\ [(2,1),(2,2),(3,2)] \end{array} \right\} \rightarrow action = south \tag{3}$$

2.4 Configuration

The following configuration variables were used in our implementation of the game.

`MOVE_PENALTY` = 1, the score penalty every time the Snake takes a step
`MICE_TO_WIN` = 30, the number of mice the Snake must eat to "Win"
`MOUSE_REWARD_MULTIPLIER` = 10, the multiplier for how score is incremented for each mouse that is eaten
`DIMENSIONS` = (10, 10), the dimensions (width, height) of the board
`NUM_MICE` = 2, the number of mice on the board during each state
`SNAKE_SPEED` = 2, the factor by which the snake is faster than the mice

3 Evaluation metrics

We define the end states as `Win`, when the snake eats 30 mice, or `Lose`, when the snake has no valid next action. The snake has no possible next action when the head of the snake is boxed in in all directions (N, S, E, W) by other snake tiles or by the edge of the game board.

We want the snake to win in the least number of steps possible. Our score will capture all of these parameters by increasing by a positive reward (+`NUM_MICE_EATEN`) each time a mouse is eaten and decreasing by a smaller negative penalty every time a step is taken.

We recognized that at later points in the game, we would be penalizing the Snake for taking a large number of steps simply to navigate around itself. In other words, we would be penalizing the snake for being further along in the game. As a result, we scale the score linearly with the agent's progress.

Ultimately, we will use the average score over `n` games as the primary evaluation metric. The win rate will serve as an auxiliary metric.

4 Infrastructure

Originally, we had implemented a framework to handle search problems with heuristics (as discussed in 5.1. After moving to a primarily game-playing implementation, we needed to majorly overhaul the framework so that it could handle GameStates that were accessible by multiple game-playing agents.

In order to implement this, we drew inspiration from the PacMan assignment [3] with the following requirements in mind:

1. Robust testing framework to run many trials with graphics enabled (for better understanding of our features) or quiet mode (for quick run-time and data collection)
2. Separate controller rules for Snake and Mice agents
3. Ability to swap and use different agents

Ultimately, we spent a fairly significant portion of our time working on this revised infrastructure to build upon for future features and extensions. The infrastructure we have built is capable while simple and flexible.

The following are brief descriptions of each of the primary files in our infrastructure:

- `game.py`:
  - Handles basic input including argument parsing for different agents and multiple runs.
  - Pulls in configuration variables from `config.py`.
  - Contains testing framework to run multiple games and compute statistics for error analysis.
  - Includes 'GameState' and 'GameRules' classes to define general functionality of Snake game.
- `config.py`: Defines configuration variables for the game, including default setup dimensions, scoring options, win conditions, and graphics options as defined in 2.4.
- `mouseRules.py` and `snakeRules.py`: Defines state and action behavior for the snake and mice agents.
- `util.py`: Contains all helper functions for heuristics.
- `agents.py`: Contains implementations of various agents, including evaluation functions A, B, and C.

5 Modeling

5.1 Heuristic-based search

When we began, we considered a search-based model for solving this problem. However, we realized that because the state changes at every time step, the search space will also increase at every time step for every state. In an attempt to curb the computation time, we researched ways to implement iterative deepening or heuristic search algorithms such that we could still perform the computation in a reasonable time.

Ultimately, we realized that this was not a scalable approach because the search space would grow exponentially. At each new time step, there is an entirely new set of states to consider. In other words, there is an entirely new problem every time a step is taken.

5.2 Minimum-depth Expectimax agent

The Expectimax algorithm, which works well when the non-snake agents follow random policies, has the ability to consider all possible states up to a given depth and make an informed decision based on that state information. While the search algorithm attempted to find complete paths to win the game, Expectimax allows informed decisions based on future states up to a certain depth. This approach is much more reasonable computationally.

Let's define the Snake game with mice:

- Players = $\{a_0, \ldots, a_n\}$, where a_0 is the Snake and $a_1, \ldots, a_n$ are the n mice agents.
- State `s` = Snake body positions and mice positions
- `Action(s)`: set of legal moves that for Agent a_i, constrained by factors like wall location, Snake body positions, and other mice locations
- `isEnd(s)`: when the Snake has lost by colliding with itself or a wall, or when the Snake has won by eating K mice, where K is the maximum defined for a winning state

- `Utility(s)`: the score at each state, as determined by the number of steps taken and the number of mice eaten

The following recurrence is defined for the maximum value, or expected utility, at the game state beginning with a defined depth `d` and decrementing until it reaches 0:

$$V_{max,opp}(s,d) = \left\{ \begin{array}{lr} \texttt{Utility(s)}, & \texttt{isEnd()} \\ \texttt{Eval(s)}, & d = 0 \\ max_{a \in Actions(s)} V_{max,opp}(succ(s,a,),d), & Player(s) = a_0 \\ \sum_{a \in Actions(s)} \pi_{opp}(s,a) V_{max,opp}(succ(s,a,),d), & Player(s) = a_1 \dots a_n \\ \sum_{a \in Actions(s)} \pi_{opp}(s,a) V_{max,opp}(succ(s,a,),d-1), & Player(s) = a_n \end{array} \right\} \tag{4}$$

5.3 Minimum-depth Minimax agent

In addition to the expecimtax agent, we implemented a Minimax agent that had the capability to play with adversarial mice. In our implementation, the mice's policy was to move in the direction that would take it furthest from the Snake. This game is defined the same as the Exectimax above in 5.2

$$V_{max,min}(s,d) = \left\{ \begin{array}{lr} \texttt{Utility(s)}, & \texttt{isEnd()} \\ \texttt{Eval(s)}, & d = 0 \\ max_{a \in Actions(s)} V_{max,min}(succ(s,a,),d), & Player(s) = a_0 \\ min_{a \in Actions(s)} V_{max,min}(succ(s,a,),d), & Player(s) = a_1 \dots a_n \\ min_{a \in Actions(s)} V_{max,min}(succ(s,a,),d-1) & Player(s) = a_n \end{array} \right\} \tag{5}$$

Finally, to prune unnecessary branches of the search tree, we implemented a Minimax agent that made use of alpha-beta pruning, which was intended to run more efficiently than the vanilla Minimax agent.

6 Approach

6.1 Baseline

Our baseline is a greedy agent in which the snake takes the valid action that moves it into the available tile which has the smallest Manhattan Distance to the nearest mouse. This means that the snake will not move into itself on purpose, as moving into itself in not a valid move. As said above, the snake will die when it has no valid moves because it has been boxed in by the snake body or the edges of the game board. This baseline is good at pursuing mice, but gets itself stuck once it is sufficiently large.

The baseline achieved an average score of 653.42 over 50 trials, with a 0% win rate.

6.2 Oracle

The oracle makes the same action decisions as the baseline, except that the oracle is allowed to move into a tile that is already occupied by a section of the snake. In other words, the oracle can cross itself, allowing it to simply move towards the nearest mouse at all times. Thus it always wins (gets to the desired number of mice), and does so without having to work around its own body.

This agent is still penalized for each step that it takes, but it shows an upper bound for winning the game by consuming K mice in the least number of steps.

The oracle achieved an average score of 4016.92 over 50 trials, with a 100% win rate.

6.3 Challenges

1. **Clustering**: The Snake should go to areas with clustered mice, not simply the closets mouse.
2. **Trapped Snake**: The Snake should avoid/escape situations in which it is trapped. [2]

3. **Approaching Mice**: Because the mice are moving randomly, the Snake will make erratic actions in attempts to catch mice on approach. This causes the Snake to form convoluted knots, making it more likely to bump into itself.
4. **Performance**: With the depth-limited agents, computational efficiency was a big consideration because the state space grew exponentially.

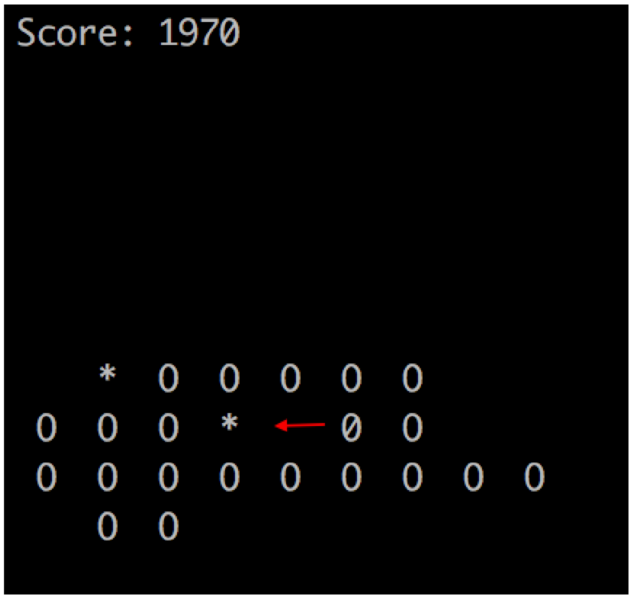


Figure 2: A scenario that depicts the impending death of the agent as it attempts to greedily eat a mouse without considering its current state.

6.4 Evaluation Functions

We combined our heuristics in three different ways, creating three different evaluation functions. They each perform differently with respect to win rate, average score (these first two are highly correlated), and time to compute. A explanation of each function is below. Of course, there is an infinite number of possible weightings of our heuristics; we chose this representative trio to show the performance of our different heuristics working together.

6.4.1 A

Our first evaluation function is similar to the baseline, with the straight-line heuristic added in. This function is simple and fast, as it only has to caluclate score, distance to the closest mouse, and number of snake tiles emerging in a straight line behind the head. The inclusion of the straight line heuristic encourages the snake to keep its shape larger and simpler, naively avoiding knots and traps.

6.4.2 B

Our second function takes the weights of the first and then adds in the Number of Blocked Adjacent Tiles heuristic with a negative weight. This evaluation function is very powerful, as it combines the simplicity of A with a simple heuristic that more intentionally avoids traps and knots. The snake is encouraged to go toward mice, but not when doing so would surround its head with too many snake tiles.

6.4.3 C

This last function includes all of the ”useful” heuristics (excluding the ones marked unused below). This approach is complicated and slow, and relies on a constant weighting of many non-independent features. It includes heuristics which each improve upon the baseline in isolation, but their combination is a bit inefficient and noisy.

7 Features

All of the features which we considered for our evaluation function (in our Expectimax and Minimax snake agents) fall into one of two categories: (1) heuristics which encourage the snake to achieve a higher score and (2) heuristics which discourage the snake from dying and self-endangerment.

7.1 Encouragement

- **Score**: This heuristic encourages actions wherein the snake consumes a mouse and increases the score.
- **Distance to Closest Mouse**: This heuristic encourages the snake to move towards the nearest mouse m by weighting against the euclidean distance between m and the snake head.
- **Distance to Farthest Mouse (Unused)**: This heuristic encourages the snake to move towards the farthest mouse m by weighting against the distance between m and the snake head. The intuition behind this heuristic is that by traversing the game board to the farthest mouse, there is a higher likelihood the snake will cross other mice. This heuristic is unsuccessful, because as the snake approaches the farthest mouse m, m is reassigned.

7.2 Discouragement

- **Length of Straight Snake Segment**: This heuristic discourages the snake from changing direction by positively weighting the length of the longest straight segment extending back from the snake head. The intuition behind this heuristic is that by minimizing contortion in the snake body we can maximize the total available, maneuverable space.
- **Number of Legal Actions**: This heuristic discourages the snake from moving into successor states with relatively fewer executable legal actions. Because the maximum depth our Expectimax and Minimax agents can explore is so constrained by the games large state space, this heuristic weights against dead-end situations where further movement, though possible, is deterministically terminal.
- **Rectangle Area of Snake**: This heuristic gives a positive weight to the rectangular area defined by the row and column ranges spanned by the snake. By encouraging the snake to span greater lengths of the game board, we preference game states with less entanglement in the snake body.
- **Area Blocked by Snake**: This heuristic discourages the snake from taking actions which result in an increased number of unreachable tiles – tiles surrounded and blocked off by the snake body. The intuition behind this heuristic is that by minimizing knotting in the snake we can optimize for the total available, maneuverable space.
- **Number of Blocked Adjacent Tiles**: This heuristic discourages the snake from moving into successor states in which the snake head is surrounded by relatively fewer unoccupied/illegal tiles. By keeping the space around the snake head free, we minimize the possibility that the snake head is in a restricted and compromising position.

- **Number of Turns in Snake (Unused)**: This heuristic, like Length of Straight Snake Segment, discourages the snake from changing direction, weighting against the total number of turns present in the snake body. The functional purpose of this heuristic is to minimize the possibility that the snake body develop any pockets. This heuristic's experimental performance was poor.

8 Results and Error Analysis

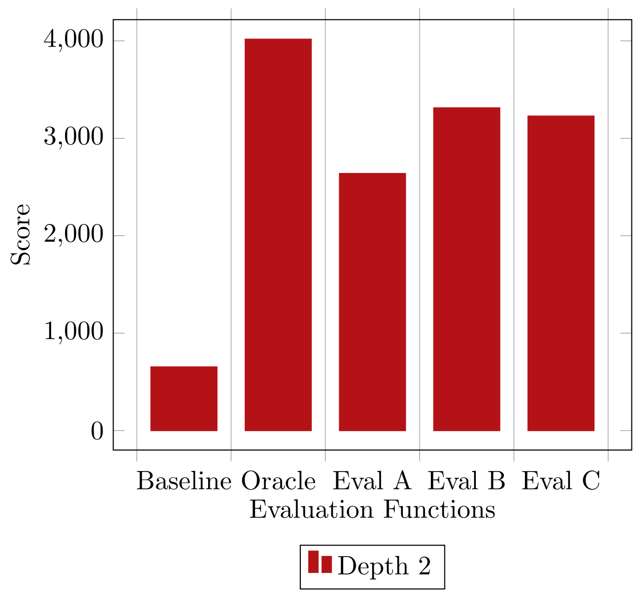
8.1 Results

The following results were computing as the averages of 50 trials with each of the following agents:

Agent	Evaluation Function	Depth	Average Score	Win Rate (%)	Average Time Per Game (s)
Baseline (greedy)	-	-	653.42	0	0.12
Oracle	-	-	4016.92	100	0.47
Expectimax	A	1	2535.96	16	0.83
Expectimax	B	1	3184.68	44	1.88
Expectimax	C	1	3227.84	40	3.95
Expectimax	A	2	2638.68	18	24.21
Expectimax	B	2	3312.4	32	24.64
Expectimax	C	2	3227.84	40	109.57
Minimax	A	1	1643.28	8	0.59
Minimax	B	1	2339.76	28	1.03
Minimax	C	1	2518.36	20	2.20
Minimax	A	2	1686.24	8	14.54
Minimax	B	2	2414.08	28	23.91
Minimax	C	2	2972.54	16	82.05
Alpha Beta	A	1	1729.92	4	0.36
Alpha Beta	B	1	2460.52	32	0.77
Alpha Beta	C	1	2231.83	20	1.66
Alpha Beta	A	2	2580.12	16	8.29
Alpha Beta	B	2	2598.56	24	11.15
Alpha Beta	C	2	2965.88	44	36.09

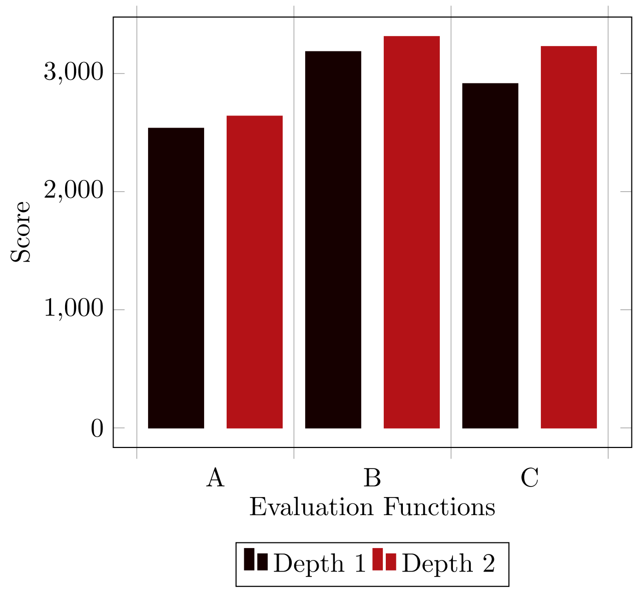
8.2 Graphs

8.2.1 Average Score for Greedy, Oracle, Eval Functions for Expectimax (Depth = 2)



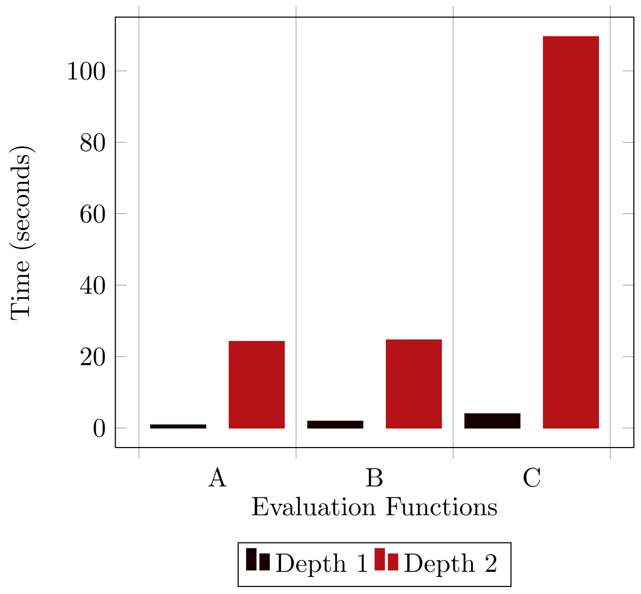
As mentioned, the Greedy agent acts as the performance baseline and the Oracle agent acts as the performance ceiling. The Eval Functions for the Expectimax agent are, in ascending order of performance, configurations A, C, and B. We believe that evaluation function C underperforms even though it has more features because the it overfits the problem. We might be applying too many features such that we longer generalize for more specific situations.

8.2.2 Average Score for Eval Functions for Expectimax (Depth = 1 and 2)



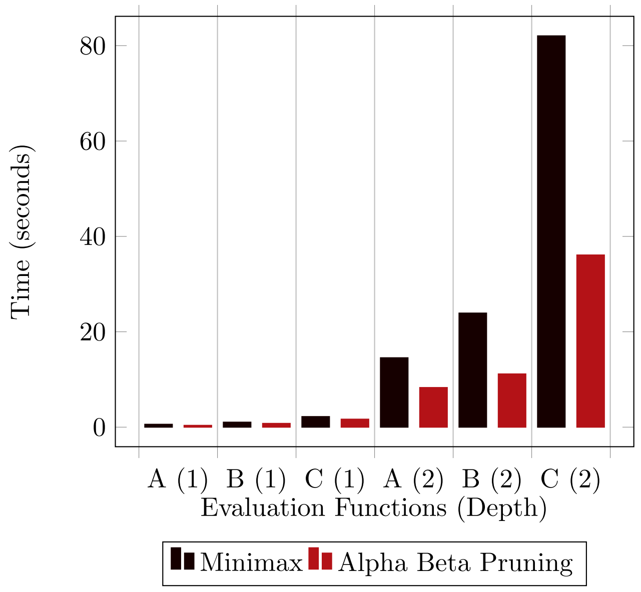
For each of the Expectimax agents, increasing the depth also increased the score. This makes sense because the agent is able to look one step deeper into the "future".

8.2.3 Average Time for Eval Functions Depth 1 and 2



With increased depth, however, we observed exponential increases in runtime. We needed to carefully balance computation time and performance. As expected, Evaluation Function C is slowest, as it has to compute the values of more features.

8.2.4 Average Time for Minimax and Alpha-Beta Pruning Depth 1 and 2



The Minimax agent, after implementing the alpha-beta pruning optimization, takes roughly half as much time on average to complete one run-through of the simulation across all configurations.

9 Literature Review to Inform Future Work

9.1 Evolutionary Algorithms, Features

The game of Snake serves as an interesting problem that can be addressed with various AI and machine learning techniques. Outside of the realm of search and game playing, we consider an approach using evolutionary algorithms.

Yeh et al. direct the movement of the snake using a controller comprised of several weighted features, which they predefine [4]. They use several EA variants – "different crossover and environmental selection operators" – to find the optimal assignment for a chromosomal representation of the weight vector. Though our models differ, we may be able to borrow some of their feature definitions in future work, three of seem particularly promising: Smoothness, Space, and Apple (their mouse equivalent).

- **Smoothness**: The authors define *smoothness* as the maximum of the set of shortest distances to all reachable values. Intuitively, this is the longest distance the snake can travel in the fewest number of turns. Encouraging actions with greater smoothness potential discourages the snake from unnecessarily and dangerously contorting itself.
- **Space**: The authors define *space* as the total number of cells reachable from the current snake position. Intuitively, this is the total space into which the snake has the ability to move given its current configuration. Encouraging actions with greater space potential discourages the snake from making myopic decisions which, although optimized for smoothness, may be fatal.

  Smoothness and *Space* are meant to complement each other.
- **Apple**: The authors define *apple* as the quotient *space / len(minimum action sequence to apple)*. Intuitively, this feature is proportional to space, and inversely proportional to length of the minimum action sequence to the apple (mouse). In this way, the snake is able to make a metered decision in the case where pursuing a mouse would mean restricting the total area available for movement.

References

[1] Previous CS221 project, Agent for the Snake Game $Khwaja, Patel, Rastogi$

[2] Implementation of A* heuristic search agent in which the Snake completes a subroutine to avoid being trapped: `https://www.youtube.com/watch?v=DnyltgX2ACo`

[3] CS221 Pacman Project source code

[4] Yeh, Jia-Fong, Pei-Hsiu Su, Shi-Heng Huang, and Tsung-Che Chiang. Snake Game AI: Movement Rating Functions and Evolutionary Algorithm-based Optimization. Thesis. National Taiwan Normal University, 2016. N.p.: n.p., n.d. ResearchGate. Web. 16 Dec. 2016.