Playing the Snake Video Game Using an Original Predictive Algorithm

Adam N. Glustein – 1006068425

Daniel Asadi – 1006018925

ESC190 – Winter 2020

April 8th, 2020

Division of Engineering Science
UNIVERSITY OF TORONTO

Abstract

This report outlines the implementation, optimization and performance of a new algorithm for playing the classic video game Snake. The predictive, greedy heuristic (PGH) algorithm was designed to maximize average game score while being versatile with different board sizes and cycle allowances. The algorithm works by using a variant of depth-first search to predict n moves ahead, ensuring that any given move will keep the Snake alive for at least n more frames. Initially, the algorithm is analyzed with constant values of n. Secondly, it is analyzed with variable values, $n(L)$, that increase with the snake's length. Finally, it is subjected to two independent variables with $n(N, K) = \max\{N, KL\}$ to exploit the early-game advantages of a constant n and the late-game advantages of a variable n. After selecting optimal values of N and K, the PGH is tested in 500 games. The performance effects of different board sizes and cycle allowances are also explored. Further improvements include optimizing runtime, since the current system runs in exponential time [$O(b^n)$, $b \in \mathbb{R}$] with respect to n.

1 Introduction

The Snake video game genre is one of the oldest and most famous of its kind. The concept originated in 1976 and exploded to stardom in 1998 after it was preloaded on Nokia cell phones.[1] In the game of Snake, the user controls a dot on the screen that represents the head of their snake. Players can only move up, down, right or left. The body of the snake follows behind and grows when the head eats a target. The goal of the game is to grow the snake as large as possible by capturing targets, and the game ends when the snake collides with the game board's boundary or hits itself.

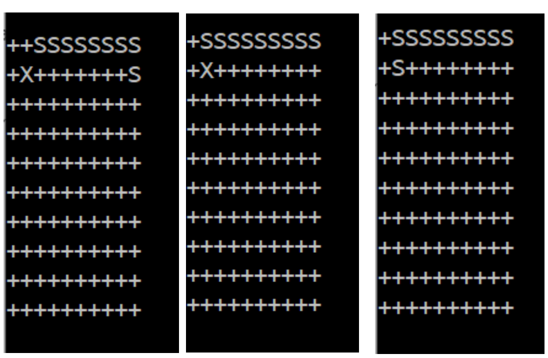


Fig. 1: A typical Snake game, showing the snake eat a target in three moves.

Solving the Snake game has become a common problem in the fields of artificial intelligence and algorithms. With a plethora of published strategies ranging from Hamiltonian cycles to deep neural networks,[2] we wanted to create an original algorithm from scratch to play the game rather than implement an existing solution. The algorithm employed, named PGH for its predictive, greedy and heuristic nature, looks ahead to ensure the snake's move does not lead into a *trap*: a situation where the snake will inevitably lose. This report provides a technical description of the algorithm and its associated data structures. It also answers three guiding questions:

1. What is the optimal number of moves to look ahead that ensures sufficient performance but runs in a reasonable amount of time?
2. How do the game board size and cycle allowance (maximum number of moves to capture each target) affect performance?
3. What are the PGH algorithm's limitations and how can these be improved?

Since target placement in Snake is a stochastic process, extensive testing on both the final algorithm and intermediate versions are conducted to give accurate representations of their abilities.

2 Objectives and Metrics

The major objective is for the Snake to maximize its score, which will be measured in the average number of targets acquired. This performance metric was chosen over the provided API's 'score' for three reasons. Firstly, it is the standard and widely used measurement of score in the game of Snake, which allows this algorithm to be compared against other previously published strategies. Secondly, the target count is more reflective of actual performance since the random allocation of bonuses included in the API's score skews results. Lastly, we wanted our algorithm to operate efficiently; since the API's scoring system rewards taking as many moves as possible, it is once again irreflective of our algorithm's intended performance.

The efficiency of the algorithm will be measured by average game time in seconds. Each game runs continuously with no sleep period between moves. This metric was chosen because the time taken per game reflects the speed at which the algorithm decides moves. Additionally, averaging out many trials (generally 100) provides a more accurate description of efficiency than single test sets. Although the exact game time measured is specific to the computer the code was tested on, the overall trend of how time increases with n is a function of the algorithm itself.

Lastly, the algorithm was designed to work within a wide range of board sizes and cycle allowances. The metric for board size versatility is the score factor, defined as the ratio of the final score to the board size. It should remain consistent across different sizes. Cycle allowance versatility will be measured by the lowest functional cycle allowance: the smallest value of CA which maintains a mean score within one standard deviation of that using the standard cycle allowance (CA = 1.5).

Objective	**Metric**	
Strong performance	Average number of targets acquired (score)	
Efficient decision-making	Average time per game (s)	
Versatility across board sizes and cycle allowances (CA)	Board size	Score factor consistency
	Cycle allow.	Lowest functional cycle allowance

Table 1: Summary of the objectives and associated metrics for the project.

3 Detailed Framework of the Project

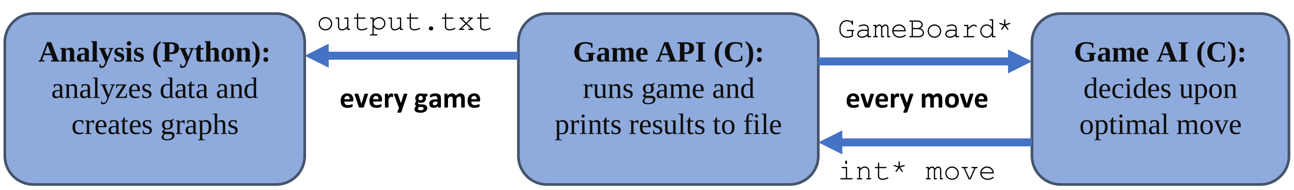


Fig. 2: The general file structure of the project, including their linking data structures.

3.1 The Game API

The game API, which runs the game and provides a graphical interface, is written in C and uses various structures to store in-game data. A frame's data is contained inside a `GameBoard` which provides all input information to the algorithm. The API was augmented to receive individual moves from the algorithm in the form of an `int*` containing the axis and direction. During testing, the `end_game` function was altered to print the final score to a file rather than the screen.

3.2 The PGH Algorithm

The PGH algorithm, which decides upon the next move, is the brain of the program. It is written in C because C is a compiled language, making it faster than interpreted languages such as

Python and therefore supporting the efficiency objective. Also, the dynamic memory allocation in C made coding easier since most functions rely on pointers to alter data.

The `n_predictor` helper function checks to see if a given move will lead to a *trap:* a situation where, after up to *n* moves, the snake will inevitably lose. To do this, it uses a depth-first search variant where each available move from the starting position is expanded recursively.

1. If any path results in an unavailable move, the function returns 0 (first base case).
2. If any path reaches *n* moves, the function returns the number of available moves from that position (second base case).
3. Elsewise, `n_predictor` returns the sum of `n_predictor(LEFT)`, `n_predictor(RIGHT)`, `n_predictor(UP)` and `n_predictor(DOWN)` **from** that game state looking *n*-1 moves ahead (recursive case).

Each time a new move is checked, the game state is updated by `update_gameboard`. Since the number of states after *n* moves grows exponentially, `n_predictor`'s runtime is $O(b^n)$, $b \in \mathbb{R}$.

The `survival` function interacts directly with the API and returns the next move by checking three ordered cases. First, it checks any moves towards the target by calling `n_predictor` on each. If `n_predictor` returns a nonzero value then the move is safe and returned. If no move towards the target is safe then `survival` checks all moves to see if *any* pass `n_predictor`; if so, that move is returned. If all moves are guaranteed to end the game in *n* frames then the algorithm returns an available move; if no moves are available the snake moves up to its death.

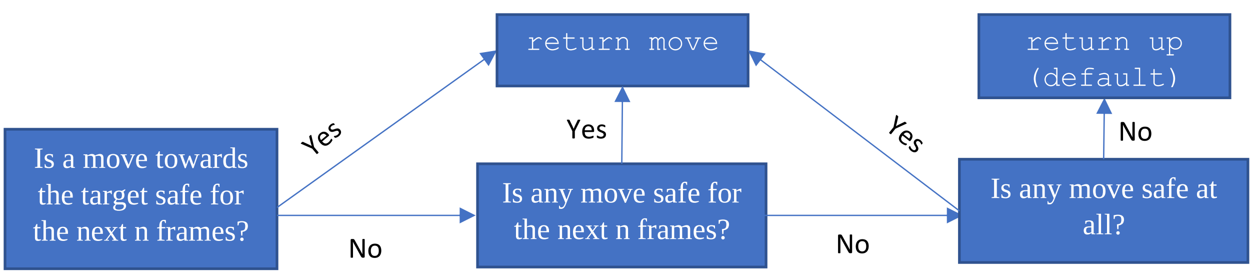


Fig. 3: Decision-making process of the `survival` function that chooses each move.

3.3 Testing and Data Analysis

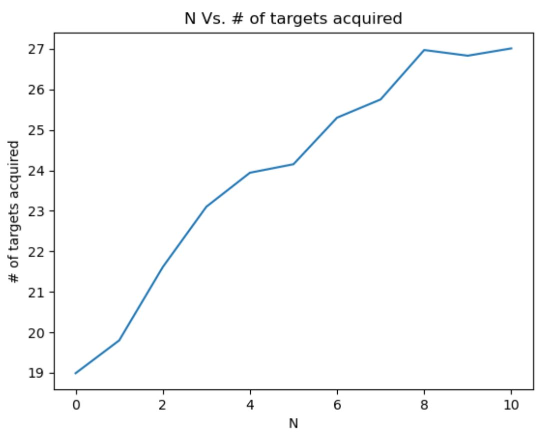
The `test_run.c` code was used to automatically run multiple games (batches up to 500) and print the results to a file. An appropriate sleep delay was employed after each game to ensure sufficient reset time for the random variables. The code also timed the games and printed the average game time to the output file. Data analysis and graph creation were done in Python using the Matplotlib library. The SciPy and NumPy libraries were also used for scientific calculations such as standard deviation. The Python code read game scores from the text files outputted by the automated testing function.

4 Algorithm Optimization

Extensive testing was conducted to choose an optimal value of *n* that balances performance and runtime. Each data point displayed in the results below is the average of 100 games.

4.1 Constant n-Value

The algorithm was initially tested with a set *n* value, $n = N$. Although runtime consistently grew exponentially, performance plateaued at $N = 8$ around 27 targets per game.



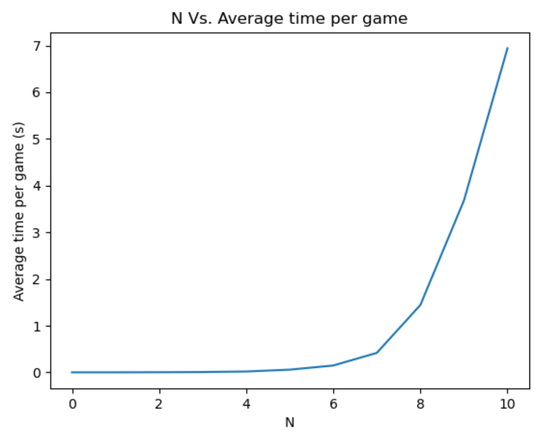


Fig. 4, 5: Performance and runtime data for constant *n* values at 100 trials per *N*.

4.2 Dynamic n-Value

Since the snake does not need to plan its moves as carefully earlier in the game than it does later, the algorithm was tested with a dynamic *n* value that was a function of the snake's length *L*. A linear function was used with $n(L) = \text{ceil}\{KL\}$, where *K* is a value between 0 and 0.3. Advantages of the dynamic *n* included improved late-game performance, demonstrated by some extremely high-scoring games. However, there were also many "duds" where the snake lost early due to *n(L)* being small.

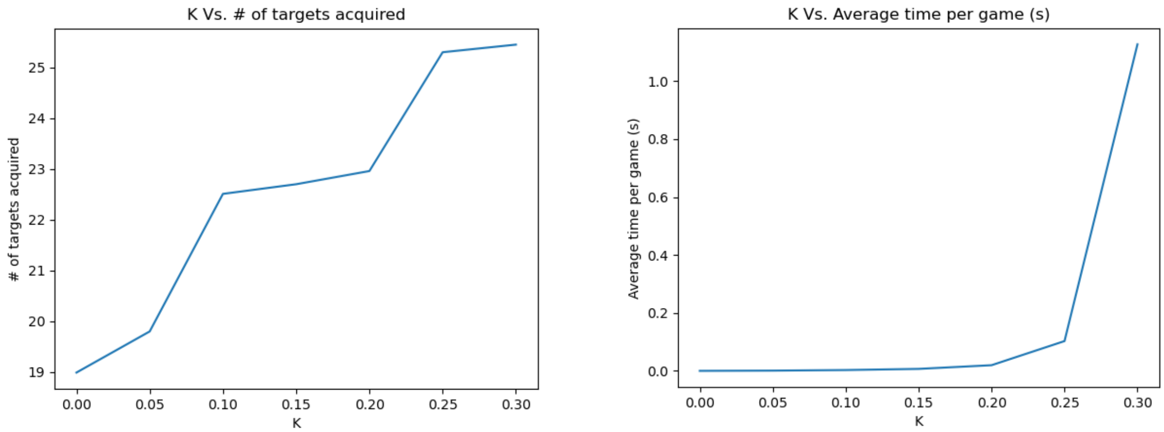


Fig. 6, 7: Performance and runtime data for dynamic *n* values where n = ceil$\{KL\}$.

4.3 Multivariable Optimization

To combine the early-game safety of constant *n* values and the late-game benefit of dynamic values, we let $n = \max\{N, KL\}$ where *n* is the number of moves predicted ahead, *N* is a constant value and *K* is as defined in 4.2. Optimizing the surface in **Fig. 8,** $N = 8$ and $K = 0.25$ gave improved performance (28.0 targets/game) at a reduced time cost (1.2 s/game) comparative to any test case in 4.1 and 4.2. These are the chosen values for the final algorithm.

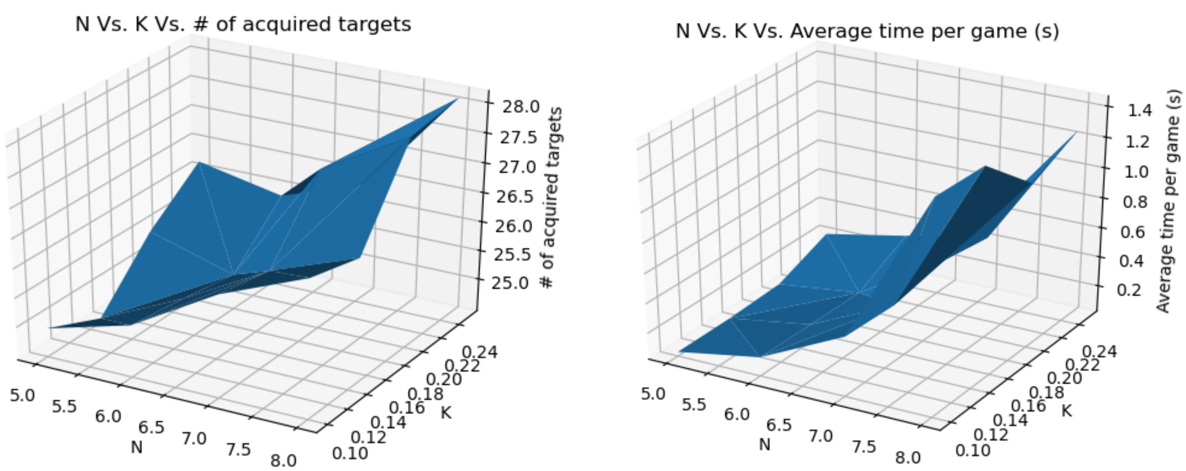


Fig. 8, 9: Multivariable performance and runtime data (16 test cases, 100 trials per case).

5 Final Algorithm Results

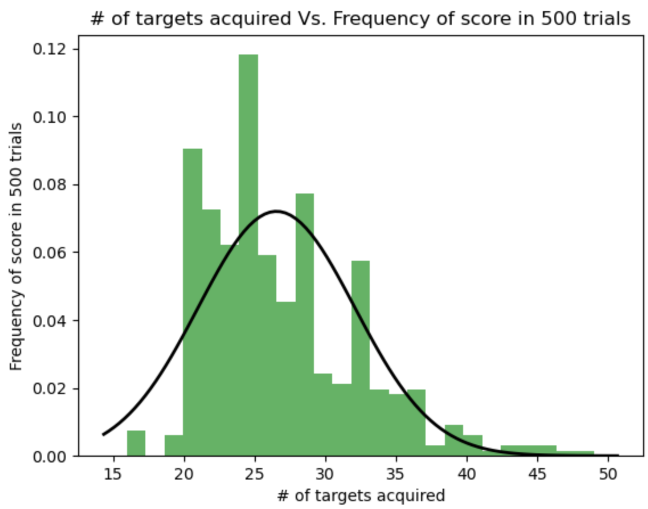


Fig. 10: Final algorithm performance in 500 trials.

After the optimization in section 4, the chosen n value is $n = \max\{8, 0.25L\}$. Testing the algorithm 500 times gave a positively skewed distribution with $\mu = 27.1$, $\sigma = 5.7$ and a range of 16-50. The mean runtime remained consistent at 1.2 s/game.

Although the algorithm does not perform as well as we hoped, it still outperforms some existing strategies such as shortest/longest paths and Monte Carlo tree search.[2] Additionally, the runtime is very efficient at 1.2 s/game. A video demonstrating the algorithm's performance is attached in Appendix B, as well as proof the algorithm runs `valgrind` clean.

6 Varying Board Sizes and Cycle Allowances

To analyze the versatility objective, the algorithm was tested in 50 trial batches with different board sizes and cycle allowances. The score factor remained relatively constant (range 2.5-3.3) across board sizes, showing that the algorithm works consistently. The substantial amount of variation in **Fig. 11** can be attributed to randomness in the game. The lowest functional cycle allowance, using the mean and standard deviation at CA = 1.5 in Section 5, is approximately 0.6. This offers a huge range of cycle allowances that will lead to satisfactory performance. The variation past CA = 1.5 in **Fig. 12** is likely due to randomness. Overall, the algorithm is extremely versatile and well-adapted to different board sizes and cycle allowances.

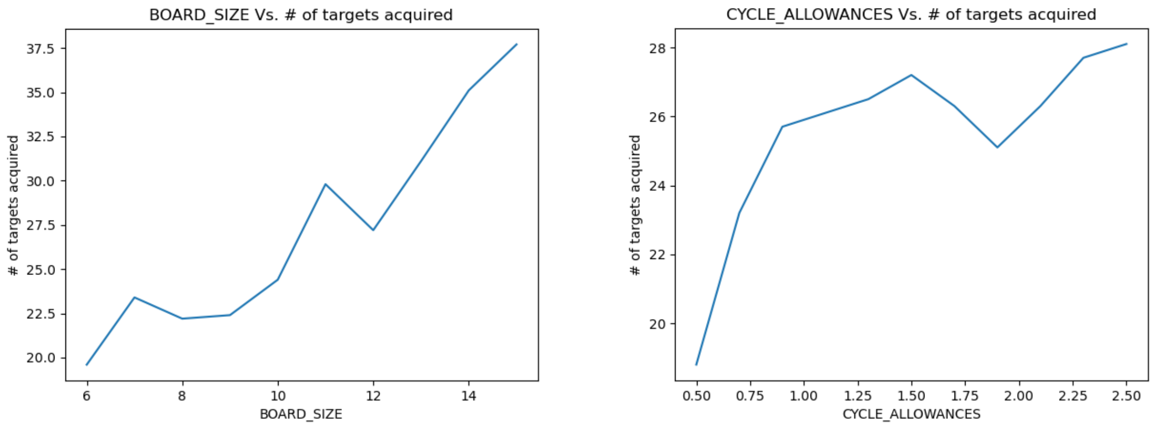


Fig. 11, 12: The effect of board size and cycle allowance on performance. The board size tests were run with a constant CA = 1.5 and the cycle allowance tests were run on a 10x10 board.

7 Future Work and Conclusion

The major limitation of the PGH algorithm is its exponential time complexity which restricts *n* to small values. The most promising improvement to the algorithm would be to make it run more efficiently, optimally in polynomial time. Using the dynamic programming principle of memoization, the program could store already-analyzed game states in a cache. Then, each move could avoid fully expanding all nodes from the start position by checking the cache for already-analyzed situations. This would transfer the exponential growth from time complexity to space complexity, which may be more viable depending on the *n* value and specific computer.

An even easier way to reduce the naivety of the algorithm is to prune the decision tree. Since `n_predictor` only needs to find one viable path to allow a move, the function could terminate once this path is found rather than continue to search. The function structure would need to be changed from recursive to a loop. While the worst-case runtime will remain exponential, this heuristic measure will greatly improve average runtime since, with large *n* values, the algorithm would usually check only a few paths rather than millions. This would enable larger *n* values which would translate to improved performance.

In conclusion, the PGH algorithm is an original solution to the Snake game problem which provides versatility across board sizes and cycle allowances. While initially promising, future improvements must be made to optimize its runtime for it to exit its current performance plateau.

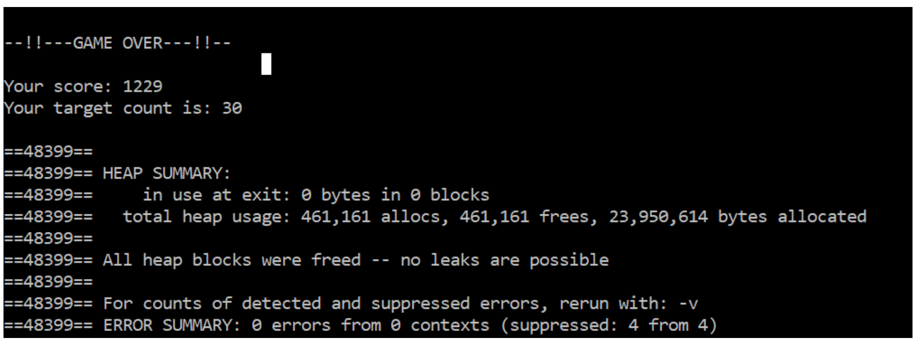
Appendix A: References

[1] Snake through the ages. (2019, March 11). Retrieved April 9, 2020, from https://community.phones.nokia.com/discussion/44549/snake-through-the-ages.

[2] Surma, G. (2018, September 23). Slitherin - Solving the Classic Game of Snake with AI (Part 1: Domain Specific). Retrieved April 9, 2020, from https://towardsdatascience.com/slitherin-solving-the-classic-game-of-snake-with-ai-part-1-domain-specific-solvers-d1f5a5ccd635.

Appendix B: Proof of Functionality

1. Valgrind screenshot.



```
--!!---GAME OVER---!!--

Your score: 1229
Your target count is: 30

==48399==
==48399== HEAP SUMMARY:
==48399==     in use at exit: 0 bytes in 0 blocks
==48399==   total heap usage: 461,161 allocs, 461,161 frees, 23,950,614 bytes allocated
==48399==
==48399== All heap blocks were freed -- no leaks are possible
==48399==
==48399== For counts of detected and suppressed errors, rerun with: -v
==48399== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

2. Video of algorithm

https://drive.google.com/file/d/1zZZlxC1n3dighUMqOkmA03F4nnoIte5w/view?usp=sharing

Appendix C: Code

1. test_run.c

```c
#include "game_AI.c"
#include <unistd.h>
#include <time.h>
#include <math.h>
#define MAX(n1, n2) n1 > n2 ? n1 : n2

void play_game_to_file(FILE* f) {
    // printf("starting\n");
    //board initialized, struct has pointer to snek
    GameBoard* board = init_board();
    // show_board(board);

    int axis = AXIS_INIT;
    int direction = DIR_INIT;

    int play_on = 1;

    while (play_on){
        int n = MAX(8, ceil(0.25*board->snek->length));
        // this controls how many moves are predicted ahead
        int* move = survival(board, n);
        // show_board(board);
        play_on = advance_frame(move[0], move[1], board);
        free(move);
        /* printf("Going ");

        if (move[0] == x){
            if (move[1] == RIGHT){
                printf("RIGHT");
            } else {
                printf("LEFT");
            }
        } else {
            if (move[1] == UP){
                printf("UP");
            } else {
                printf("DOWN");
            }
        } printf("\n");
        // usleep(184220); */
    }
    end_game(&board, f);
}

void print_100_games_to_file(char* fn) {
    int delay;
    FILE* f  = fopen(fn, "w");
    int start = clock();
    for (int i = 0; i < 100; i++)
    {
        play_game_to_file(f);
        delay = 1000000;
        usleep(delay); // to ensure randomness
    }
    int end = clock();
    char* time = calloc(10, sizeof(char));
    sprintf(time, "%f", ((double)(end-start))/(CLOCKS_PER_SEC*100)-delay/2000000); // needs tuning when you change sleep time
    fputs("Average game time: ", f); fputs(time, f);
    free(time);
    fclose(f);
}

int main(){
    print_100_games_to_file("output.txt");
    return 0;
}
```

2. game_AI.h

```c
#include "snek_api_augmented.h"

int* find_target(GameBoard* gb);
int coord_equal(int* c1, int* c2);
int move_available(GameBoard* gb, int* next_square);
int* next_position(int* coord, int* move);
int target_next(int* head, int* target);
GameBoard* update_gameboard(GameBoard* gb, int* move, int* target);
void delete_gameboard(GameBoard* gb);
int n_predictor(int n, GameBoard* gb, int* move, int* target);
int* survival(GameBoard* gb, int n);

```

3. game_AI.c

```c
#include "game_AI.h"

int* find_target(GameBoard* gb) {
    for (int i = 0; i < BOARD_SIZE; i++) {
        for (int j = 0; j < BOARD_SIZE; j++) {
            if (gb->cell_value[i][j] != 0) {
                int* target = calloc(2, sizeof(int));
                target[0] = j;
                target[1] = i;
                return target;
            }
        }
    }
    return NULL;
}

int coord_equal(int* c1, int* c2) {
    if (c1[0] == c2[0] && c1[1] == c2[1]) return 1;
    return 0;
}

int move_available(GameBoard* gb, int* move) {
    int* next_square = next_position(gb->snek->head->coord, move);
    if (next_square[0] > BOARD_SIZE-1 || next_square[0] < 0 || next_square[1] > BOARD_SIZE-1 || next_square[1] < 0) { // out of range
        free(next_square);
        return 0;
    }
    if (gb->occupancy[next_square[1]][next_square[0]] && !coord_equal(next_square, gb->snek->tail->coord)) { // hitting itself
        free(next_square);
        return 0;
    }
    free(next_square);
    return 1;
}

int* next_position(int* coord, int* move) {
    int* next = calloc(2, sizeof(int));
    next[0] = coord[0]; next[1] = coord[1];
    next[move[0]] += move[1];
    return next;
}

int target_next(int* head, int* target) {
    if (target == NULL) return 0;
    if (head[0] == target[0] && abs(head[1]-target[1]) == 1) return 1;
    if (head[1] == target[1] && abs(head[0]-target[0]) == 1) return 1;
    return 0;
}
```

```c
GameBoard* update_gameboard(GameBoard* gb, int* move, int* target) {
    GameBoard* new_gb = malloc(sizeof(GameBoard));
    new_gb->snek = malloc(sizeof(Snek));
    new_gb->snek->head = malloc(sizeof(SnekBlock));
    new_gb->snek->length = gb->snek->length;
    // cell value is not needed since the target has been located already
    int* new_head = next_position(gb->snek->head->coord, move);
    new_gb->snek->head->coord[0] = new_head[0];
    new_gb->snek->head->coord[1] = new_head[1];
    new_gb->snek->head->next = gb->snek->head;

    // now update occupancy and also update snake tail OR length if needed
    SnekBlock* curr = new_gb->snek->head;
    int tail_next = 0;

    while (curr->next && !tail_next) { // change
        new_gb->occupancy[curr->coord[1]][curr->coord[0]] = 1;
        if (coord_equal(curr->next->coord, gb->snek->tail->coord)) tail_next = 1;
        else curr = curr->next;
    }
    if (target_next(gb->snek->head->coord, target)) {
        new_gb->occupancy[curr->next->coord[1]][curr->next->coord[0]] = 1; // adding in previous tail
        new_gb->snek->length++;
        new_gb->snek->tail = curr->next;
    }
    else {
        new_gb->snek->tail = curr; // making new tail
    }

    return new_gb;
}

void delete_gameboard(GameBoard* gb) {
    for (int i = 0; i < BOARD_SIZE; i++)
        free(gb->occupancy[i]);
    free(gb->occupancy);
    free(gb->snek->head);
    // not freeing internal nodes as those may be in the actual game snake, causes seg fault
    free(gb->snek);
    free(gb);
}

int n_predictor(int n, GameBoard* gb, int* move, int* target) {
    // The brain of the program: checks if after n moves the Snake will be trapped

    if (!move_available(gb, move)) return 0; // base case 1

    if (n == 0) { // base case 2
        int sum = 0;
        int* new_move = calloc(2, sizeof(int));
        for (int ax = 0; ax < 2; ax++) {
            for (int dir = -1; dir < 2; dir+=2) {
                new_move[0] = ax; new_move[1] = dir;
                if (move_available(gb, new_move)) sum += 1;
            }
        }
        free(new_move);
        return sum;
    }

    // move is available, so check all possible paths after said move
    GameBoard* new_gb = update_gameboard(gb, move, target);
    int non_traps = 0;
    int* new_move = calloc(2, sizeof(int));
    for (int ax = 0; ax < 2; ax++) {
        for (int dir = -1; dir < 2; dir+=2) {
            new_move[0] = ax; new_move[1] = dir;
            non_traps += n_predictor(n-1, new_gb, new_move, target); // recursive step: worst case O(4^n)
        }
    }
    free(new_move);
    delete_gameboard(new_gb); // frees memory
    return non_traps;
}
```

```c
int* survival(GameBoard* gb, int n) {
    // 1. Check to see if a move towards the target that ensures n moves is possible
    int* target = find_target(gb);
    int* head = gb->snek->head->coord;
    int target_xdir = 0;
    int target_ydir = 0;
    int* move = calloc(2, sizeof(int));

    if (target) {
        if (head[0] < target[0]) target_xdir = 1;
        else if (head[0] > target[0]) target_xdir = -1;
        if (head[1] < target[1]) target_ydir = 1;
        else if (head[1] > target[1]) target_ydir = -1;
    }

    if (target_xdir != 0) {
        move[0] = x; move[1] = target_xdir;
        if (n_predictor(n, gb, move, target) > 0) return move;
    }
    if (target_ydir != 0) {
        move[0] = y; move[1] = target_ydir;
        if (n_predictor(n, gb, move, target) > 0) return move;
    }

    // 2. Check to see if any other moves are available that ensure at least n further moves
    for (int ax = 0; ax < 2; ax++) {
        for (int dir = -1; dir < 2; dir+=2) {
            move[0] = ax; move[1] = dir;
            if (n_predictor(n, gb, move, target) > 0) return move;
        }
    }

    // 3. If no moves ensure at least n further moves, then choose an available move.
    for (int ax = 0; ax < 2; ax++) {
        for (int dir = -1; dir < 2; dir+=2) {
            move[0] = ax; move[1] = dir;
            if (move_available(gb, move)) return move;
        }
    }
    return move; // no moves available, go up
}
```

4. snek_api_augmented.c (altered end_game)

```c
void end_game(GameBoard **board, FILE* f){
    //fprintf(stdout, "\033[2J");
    //fprintf(stdout, "\033[0;0H");
    fprintf(stdout, "\n\n\n--!!---GAME OVER---!!--\n\nYour score: %d\n\n\n\n", SCORE);
    fflush(stdout);
    // need to free all allocated memory
    // first snek

    char* score = calloc(4, sizeof(char)); // this section of code prints to file
    sprintf(score, "%d", get_moogles_eaten());
    fputs(score, f);
    fputs("\n", f);
    free(score);


    SnekBlock **snekHead = &((*board)->snek->head);
    SnekBlock *curr;
    SnekBlock *prev;
    while ((*snekHead)->next != NULL) {
        curr = *snekHead;
        while (curr->next != NULL){
            prev = curr;
            curr = curr->next;
        }
        prev->next = NULL;
        free(curr);
    }
    free(*snekHead);
    free((*board)->snek);
    free(*board);
}
```

5. plot2d.py

```
import matplotlib.pyplot as plt


def main():
    num_elements = 11
    X = [0] * num_elements
    Y = [0] * num_elements
    i = 0

    filepath = 'data/cycle.txt'
    file = open(filepath, 'r')
    arr = file.readlines()
    for line in arr:
        line = line.split()
        X[i] = float(line[0])
        Y[i] = float(line[1])
        i += 1

    print(X)
    print(Y)
    # plot
    plt.plot(X, Y)

    # change axes ranges
    # plt.xlim(0, 100)
    # plt.ylim(0, 40)

    # add title
    plt.title('CYCLE_ALLOWANCES Vs. # of targets acquired')

    # add x and y labels
    plt.xlabel('CYCLE_ALLOWANCES')
    plt.ylabel('# of targets acquired')

    # show plot
    plt.show()


if __name__ == "__main__":
    main()
```

6. plot3d.py

```
import matplotlib.pyplot as plt

num_elements = 16
X = [0] * num_elements
Y = [0] * num_elements
Z = [0] * num_elements
i = 0

filepath = 'data/avg_t3.txt'
file = open(filepath, 'r')
arr = file.readlines()
for line in arr:
    line = line.split()
    X[i] = float(line[0])
    Y[i] = float(line[1])
    Z[i] = float(line[2])
    i += 1

print(X)
print(Y)
print(Z)

# x = np.reshape(X, (1, 16))
# y = np.reshape(Y, (1, 16))
# z = np.reshape(Z, (1, 16))

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_trisurf(X, Y, Z)

ax.set_title('N Vs. K Vs. Average time per game (s)')
ax.set_xlabel('N')
ax.set_ylabel('K')
ax.set_zlabel('Average time per game (s)')

plt.show()
```

7. plot_distribution.py

```
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt


def main():
    num_elements = 500
    X = [0] * num_elements
    i = 0

    filepath = 'data/distribution500.txt'
    file = open(filepath, 'r')
    arr = file.readlines()
    for line in arr:
        line = line.split()
        X[i] = float(line[0])
        i += 1

    data = sorted(X)
    print(data)
    mu, std = norm.fit(data)

    # Plot the histogram.
    plt.hist(data, bins=25, density=True, alpha=0.6, color='b')

    # Plot the PDF.
    xmin, xmax = plt.xlim()
    x = np.linspace(xmin, xmax, 500)
    p = norm.pdf(x, mu, std)
    print(mu)
    print(std)
    plt.plot(x, p, 'k', linewidth=2)
    plt.title('# of targets acquired Vs. Frequency of score in 500 trials')
    plt.xlabel('# of targets acquired')
    plt.ylabel('Frequency of score in 500 trials')
    plt.show()


if __name__ == "__main__":
    main()
```