Project Final Report
Snake Solving Algorithm

Course: ESC190H1
Instructors: Prof. Ali, Prof. Lucasius
Date: Apr. 9th, 2020

Authors:
Jingzhou Liu (1005906405)
Ritvik Singh (1005834554)

1. Introduction

The purpose of the Snek project is to develop an algorithm for an AI that maximizes the number of points it can score in a variant of the classic video game Snake. The algorithm aims to generate a sequence of desired moves[1] for a snake called Magini on a 10 × 10 game board. On the board, one target (either a Moogle or a Hurry Pooter) will spawn at a random location for Magini to eat[2]. However, Magini must eat that target before TIMEOUT[3], which is a function that sets the number of moves Magini can make after the appearance of the target. TIMEOUT resets whenever Magini eats the target while Magini increases its length by one. Magini scores one point for every move it makes, twenty points for eating a Moogle, and sixty points for eating a Hurry Pooter. The game ends when Magini's head moves into itself, moves into the edge, or Magini runs out of moves due to TIMEOUT. Thus, the algorithm seeks to maximize the points Magini can acquire before the game ends with the augmentation to adjust the board game size.

2. Describing the Objectives

<table>
<tr><td colspan="2">Objective 1: The algorithm should maximize the number of points Magini scores</td></tr>
<tr><td>Metric 1: Number of points Magini scores</td><td>Criteria 1: More points scored is better</td></tr>
<tr><td colspan="2">Explanation 1:
We used this metric to assess the performance of the algorithm when we were creating and refining the program for the 10x10 board size. However, we did not use the number of points as a metric to assess the efficacy of the algorithm across varying board sizes. Instead, metric 2 is utilized.</td></tr>
<tr><td>Metric 2: Number of targets Magini eats</td><td>Criteria 2: More targets eaten is better</td></tr>
<tr><td colspan="2">Explanation 2:
While the goal of this project is to maximize the score, the number of targets Magini eats is used to compare the efficacy of the algorithm for different board sizes. This was done for two reasons: the number of targets eaten has a direct correlation with score, and it is far more difficult to normalize the score collected over various board sizes. When looking at the</td></tr>
</table>

[1] In the context of this report, a move is considered to be a change in the position of Magini's head.
[2] A target is considered to be eaten when the position of Magini's head overlaps the position of the target.
[3] Formula for timeout (According to the given API): TIMEOUT = ((BOARD_SIZE × 4) - 4) × 1.5

number of targets eaten, it is easy to normalize the data because the maximum number of targets that can be eaten is BOARD_SIZE^2 -1, which is a strict upper bound. The score, on the other hand, has no upper bound as Magini could theoretically continue to move around the board forever if the target never spawns. With no upper bound on the score, it is impossible to normalize the scores, which makes it difficult to compare across different board sizes.

Objective 2: The algorithm should be functional for more than one board size	
Metric 3: Number of various board sizes on which Magini can collect an average of BOARD_SIZE^2 / 2 targets	**Criteria 3:** More functional board sizes is better
Explanation 3: The algorithm should be functional for more than one board size because we chose to implement the first augmentation option, which is to find a lower and upper bound on the board size that the algorithm is functional on. The algorithm is considered *functional* iff Magini consumes at least half of the size of the game board, which means the percentage of number of targets consumed over BOARD_SIZE^2 should be above 50%. The reason for half of the board size as opposed to the entire board size is that while the size increases quadratically with respect to a change in the dimension, the actual timeout only increases linearly. As such, larger board sizes can have cases where a target spawns and it is physically impossible for Magini to eat it before the timeout.	

3. Delineating the Detailed Framework

This section will discuss the details regarding the framework, the concepts, and the implementation of the Snek AI.

3.1 Language and Modifications to the Provided API

The language chosen for this project is C. The only modification that was made to the API is that in the *show_board* function of *sneck_API.c*, we changed the character that represents the snake's head and the snake's tail from 'S' to 'H' and 'T' respectively (Fig. 1). This was done to provide clarity to the location of the snake's head on the board during debugging and testing.

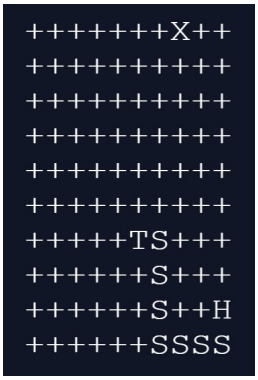


Fig. 1: Snek in Action

- "H" represents the head of Magini
- "T" represents the tail of Magini
- "S" represents the body of Magini
- "+" represents an empty location
- "X" represents the target

3.2 High-Level Conceptual Introduction to the Algorithm

In essence, the algorithm generates a Hamiltonian circuit that the snake follows; however, in the case where a target spawns, the snake then follows a short path[4] that is *guaranteed* to be *safe* and the snake returns to its Hamiltonian path when the target is eaten. This algorithm attempts to ensure Magini stays alive for as long as possible by consuming as many targets as it can; this is because the more targets the snake consumes, the higher it will score, which optimizes Metric 1 and Metric 2 of Objective 1.

A Hamiltonian circuit, in the context of this project, is a path that goes through every cell on the board exactly once and ends where it started. This ideal Hamiltonian circuit is only possible on an even dimensioned board (Fig. 2). However, in the event that the dimension of the board is odd, a modified Hamiltonian circuit can be generated. This circuit goes through every cell except for one (Fig. 3.1), and if Magini needs to go into that cell, a different Hamiltonian circuit is generated to include that cell (Fig. 3.2). By following a Hamiltonian circuit indicated, we can generate a *safe* path that Magini can always follow and guarantees that it will eat all the Moogles possible.

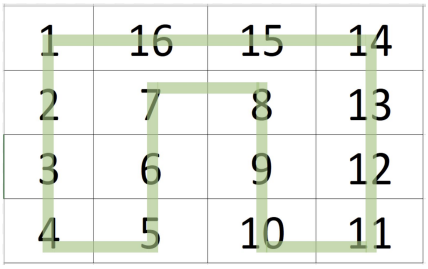


Fig 2: An Even Ham. Path

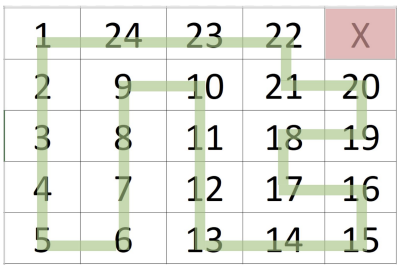


Fig 3.1: An Odd Ham. Path

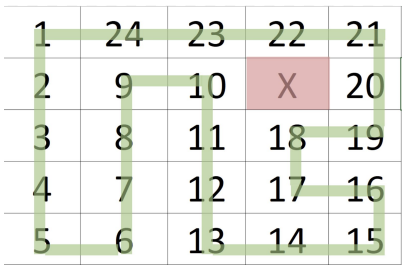


Fig 3.2: An Odd Ham. Path Modified

[4] A short path in this context means a path that deviates from a normal Hamiltonian path and results in fewer moves to reach the target.

However, because of TIMEOUT, the algorithm needs to be able to take shortcuts. We decided against implementing a standard shortest path finding algorithm such as Dijikstra's or A* because while those algorithms are able to avoid the body of the snake, it cannot calculate whether a certain path will eventually result in the snake trapping itself. Instead, we drew some inspiration from John Tapsell's article on a Snake AI [1]. The revelation is that in order for the snake to never trap or eat itself, its next move must allow the snake to immediately return to a predefined Hamiltonian path if the snake needs to. If this condition is enforced for every move, the snake always has a guaranteed safe path to resort to since Magini will not die if it follows a Hamiltonian path.

This can be visualized by transforming a Hamiltonian Path into its one-dimensional equivalent (Fig. 4.1). In Figure 4.2, the snake's head could potentially move to Square 6, 8, 10, 12; however, the only moves that satisfy the aforementioned condition are Square 10 and 12 since Square 6 and 8 would result in the snake going into itself if it were to immediately follow the Hamiltonian path. Notice that in Figure 4.1, the allowable moves (Square 10 and 12) are both outside the tail and the head (they are both < 2 and > 9), but the non-allowable moves (Square 6 and 8) are both in between the tail and the head. This is true in general, so as long as the move for the snake is not in between the tail and head, the snake is safe.

Fig. 4.1: One-Dimensional View of Hamiltonian Path (Purple Represents Allowable Squares)

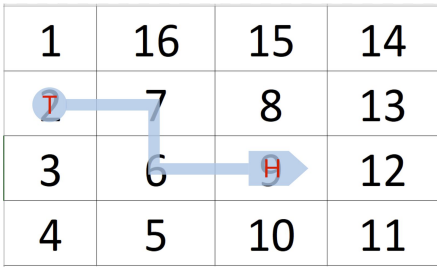


Fig. 4.2: An Example of Snake

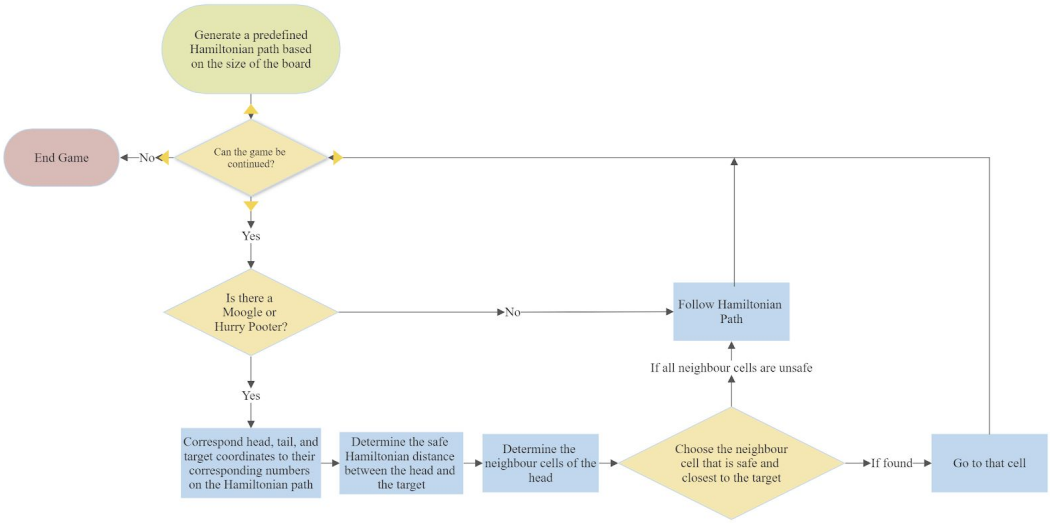
```
1 100 99 98 97 96 95 94 93 92
2 19 20 37 38 55 56 73 74 91
3 18 21 36 39 54 57 72 75 90
4 17 22 35 40 53 58 71 76 89
5 16 23 34 41 52 59 70 77 88
6 15 24 33 42 51 60 69 78 87
7 14 25 32 43 50 61 68 79 86
8 13 26 31 44 49 62 67 80 85
9 12 27 30 45 48 63 66 81 84
10 11 28 29 46 47 64 65 82 83
```

Fig. 5: Printed 10 × 10 Hamiltonian Array

3.3 Data Structure Used for Storing the Hamiltonian Circuit

A two-dimensional array of size BOARD_SIZE^2 was utilized to store the Hamiltonian circuit. The array stores integer numbers that represent the order of the cell that the snake should follow to complete a Hamiltonian path. For example, if the snake's head is on 3, it should go to the cell with 4 next if it were to follow the Hamiltonian path. Figure 5 is a printed version of the 10 × 10 Hamiltonian array.

3.4 General Flow Chart of the Algorithm



3.5 Code Walk-Through of One Case

This section explores one of the six main cases. Due to the length constraint of the report, the other cases will not be discussed. The full code is provided in Appendix B with more annotations. Consider the following scenario on a 4 × 4 board (Fig. 6.1), where 'X' represents the location of a moogle. Since a moogle is present, a short cut needs to be taken.

X	16	15	14
T	7	8	13
3	6	H	12
4	5	10	11

Fig. 6.1: One Scenario

X T 3 4 5 6 7 8 H 10 11 12 13 14 15 16

Fig. 6.2: One Scenario 1-D Equivalent

The following code determines the safe distance[5] the moogle can travel. In this case, *tailNum* = 2, *headNum* = 9, and *moogleNum* = 1.

[5] In the context of this algorithm, distance represents the number of moves it takes to get from one point to another by following a Hamiltonian path. For example, if the head is at 3 and the tail is at 10, the distance between the head and the tail is 6 because it takes 6 moves to get from 3 to 10.

```
int distTail = (headNum < tailNum) ? tailNum-headNum-1:tailNum-headNum-1+max_ham_num;
int distMoogle = (headNum < moogleNum) ? moogleNum-headNum-1:moogleNum-headNum-1+max_ham_num;
int available = distTail-3; //Subtract a buffer amount

if(distMoogle < distTail){available--;}
if(distMoogle < available){available = distMoogle;}
if(available < 0){available = 0;}
```

distTail calculates the number of cells outside of the head and the tail, which equals 8 in this case (10,11,12,13,14,15,16,1). This can be thought of as the distance the head would need to travel to reach where the tail is by following the Hamiltonian path.

distMoogle calculates the distance between the head and the moogle, which equals 7 in this case.

available is first set to *distTail* - 3. The number 3 here is a buffer. It is used to prevent the head from colliding with the tail when the head eats the moogle and grows one cell longer. The buffer value 3 is empirically determined because 3 yields the most amount of points (Metric 1). If the buffer were 1 or 2, the snake would frequently collide into itself. Buffer values of 3, 4, 5 were tested 200 times each and 3 demonstrates the best performance as exhibited below.

	Buffer Value = 3	Buffer Value = 4	Buffer Value = 5
Mean Score (n = 200)	3132.97	3069.545	3076.9

available is then compared to *distMoogle*. If *distMoogle* is lower than *available*, *available* takes on that value. *available* essentially represents the distance Magini can travel without overshooting the target and without going into a cell that is in between the tail and the head. Thus, by traveling to a cell with a distance less than *available*, the move is safe. In this case, *available* = 7.

Now, we need to consider all the neighbouring cells of the head. The goal here is to determine the neighbor with the greatest distance that is also smaller than *available*. This is because the greater the distance, the more cells it will skip and therefore closer to the target; it also cannot exceed *available* because then it will overshoot. Since all neighbours between the head and the tail are deemed unsafe as stated above (in this case, 6 and 8), they cannot be considered for the next move and are therefore incremented by BOARD_SIZE^2 to ensure they are larger than *available*. In this case, the neighbour with the largest distance yet smaller than *available* is 12, which will be the next cell the snake should move to.

```
if(coord[x] > 0 && board->occupancy[coord[y]][coord[x]-1] != 1)
  leftNum = arr[coord[x]-1][coord[y]];
if(coord[x] < BOARD_SIZE-1 && board->occupancy[coord[y]][coord[x]+1] != 1)
  rightNum = arr[coord[x]+1][coord[y]];
if(coord[y] > 0 && board->occupancy[coord[y]-1][coord[x]] != 1)
  upNum = arr[coord[x]][coord[y]-1];
if(coord[y] < BOARD_SIZE-1 && board->occupancy[coord[y]+1][coord[x]] != 1)
  downNum = arr[coord[x]][coord[y]+1];

int adjacent[] = {leftNum , rightNum, upNum, downNum};
int greatestDist = -1;

for(int i = 0; i < 4; i++) {
  int distance = (headNum < adjacent[i]) ? adjacent[i]-headNum-1 : adjacent[i]-headNum-1+max_ham_num;
  if(adjacent[i] != -1 && distance <= available && distance > greatestDist) {
    greatestDist = distance;
    greatest = adjacent[i];
  }
}
```

3.6 Time and Space Complexity of the Algorithm

The time and space complexity of the algorithm is both $O(n^2)$ where n is the board size. This is because the storage and iteration of the Hamiltonian array which maps each coordinate to one number and the number of coordinates grows quadratically with respect to the board size.

4. Exhibiting Results of the Algorithm [Video]

4.1 Results of Algorithm on Default Board Size

The algorithm was initially tested on the standard board size of 10 over 1000 trials. This was because the main objective was to maximize the score on the default board (Metric 1). The mean score that it achieved was 3139.791 and the mean number of moogles eaten was 60.69 moogles. The standard deviation for the score was 368.12 which is 11.7%. Empirically, we can see that the lowest the algorithm will score is 1935 points (Fig. 7.1). If we consider data that is within three standard deviations of the mean significant, we can conclude that 99.85% of the time, the score will be above 2035 points.

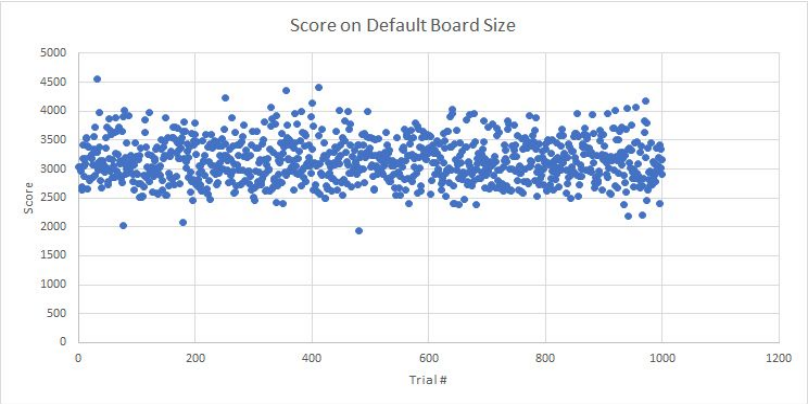


Fig. 7.1: Scores on the Default Board Size

4.2 Results of Algorithm on Variable Board Size[6]

As per the Objective 2, the algorithm was augmented to account for differing board sizes. For each of the board sizes ranging from 1 all the way to 14, 100 trials were calculated (Fig. 7.2). In accordance to Metric 3, in order to measure if the algorithm reliably worked for the board size, percentage of the mean number of targets eaten divided by the board size squared was calculated (Fig. 7.3). Higher board sizes were not tested because at 13, the percentage was already lower than 50% and as seen in the graph, it is strictly decreasing after board size 4. The lower bound on which the algorithm can reliably work is 2 and the upper bound is 12. Furthermore, as demonstrated in Figure 7.2, the score seems to increase linearly with respect to the board size at a rate of 414 points/board size (See Appendix C).

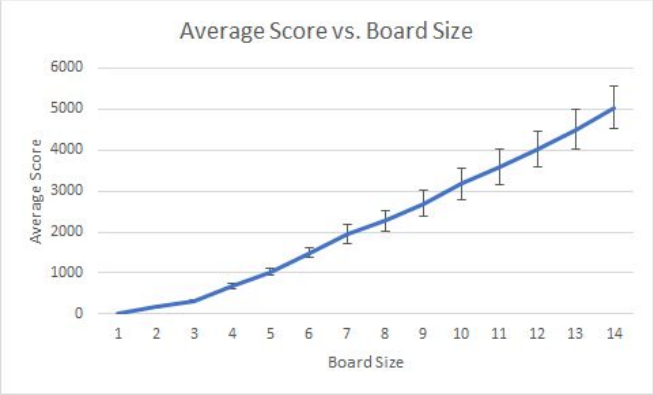


Fig. 7.2: Average Score vs. Board Size

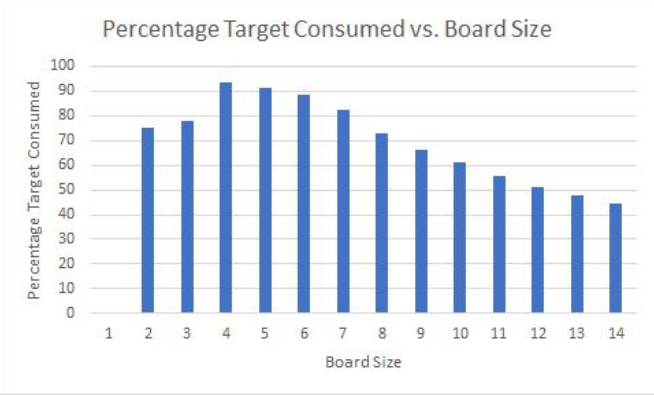


Fig 7.3: Percentage Targeted Consumed vs. Board Size

5. Future Work and Conclusion

One of the main aspects of the algorithm that can be improved is to not immediately take a short cut once a target appears. This is because 1 point is awarded for simply making a move and staying alive. Therefore, the ideal algorithm would be for Magini to stay alive but not eat the target and then calculate the distance to it such that it is able to just avoid a timeout by eating it. By doing so, Magini would take exactly the same number of moves as TIMEOUT to consume a target, hence optimizing the number of points Magini gets.

In conclusion, with respect to Objective 1, the mean number of points Magini scores on the default 10 × 10 board is 3139.791 (Metric 1) and the mean number of targets eaten was 60.69 moogles (Metric 2). 99.85% of the data will also score higher than 2035. For Objective 2, the algorithm works effectively on board size 2 to 12, which is 11 different board sizes (Metric 3).

[6] A video consisting of the algorithm working on several board sizes is provided: https://www.youtube.com/watch?v=pKwhaKGeAgs

6. Appendices

Appendix A: References:

[1] Tapsell, John. Nokia 6110 Part 3 – Algorithms [Web]. (2015). Retrieved from https://johnflux.com/2015/05/02/nokia-6110-part-3-algorithms/

Appendix B: Algorithm Code

main.c:

```c
#include "snek_api.h"
#include <unistd.h>

void play_game() {
  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;
  int coord[2];
  int moogleLocX;
  int moogleLocY;

  //Generate Hamiltonian Circuit
  int arr_count = 1;
  int arr[BOARD_SIZE][BOARD_SIZE]; //Hamiltonian Array
  int max_ham_num; //The maximum value that will occur in the Hamiltonian Array

  if (BOARD_SIZE%2 == 0){//Even sized board
    max_ham_num = BOARD_SIZE*BOARD_SIZE;
    //First Column
    while(arr_count <= BOARD_SIZE) {
      arr[0][arr_count-1] = arr_count;
      arr_count++;
    }
    //Subsequent Columns
    for(int i = 1; i < BOARD_SIZE; i++) {
      if(i % 2 == 0) {
       for(int j = 1; j < BOARD_SIZE; j++)
        arr[i][j] = arr_count++;
      }else {
        for(int j = BOARD_SIZE-1; j > 0; j--)
          arr[i][j] = arr_count++;
      }
    }
    //First row
    for(int i = BOARD_SIZE-1; i > 0; i--){
      arr[i][0] = arr_count++;
    }

  }else if (BOARD_SIZE%2 == 1){//Odd sized board
    max_ham_num = BOARD_SIZE*BOARD_SIZE - 1;
    //First Column
    while(arr_count <= BOARD_SIZE) {
      arr[0][arr_count-1] = arr_count;
      arr_count++;
    }
    //Subsequent Columns Until BOARD_SIZE-2
    for(int i = 1; i < BOARD_SIZE-2; i++) {
      if(i % 2 == 0) {
        for(int j = 1; j < BOARD_SIZE; j++) {
          arr[i][j] = arr_count++;
        }
      }else {
        for(int j = BOARD_SIZE-1; j > 0; j--) {
          arr[i][j] = arr_count++;
        }
      }
    }
    //Last Two Columns
    for (int i = BOARD_SIZE-1; i > 0; i--){
      if (i % 2 == 0){
        arr[BOARD_SIZE-2][i] = arr_count++;
        arr[BOARD_SIZE-1][i] = arr_count++;
      }else{
        if (i % 2 == 1){
          arr[BOARD_SIZE-1][i] = arr_count++;
```

```
            arr[BOARD_SIZE-2][i] = arr_count++;
          }
        }
      }
      arr[BOARD_SIZE-1][0] = -1; //Initialize one inaccessible square to -1
      //First Row
      for(int i = BOARD_SIZE-2; i > 0; i--) {
        arr[i][0] = arr_count++;
      }
    }

    //Start Game
    while (play_on){
      //In the case where a moogle spawns in the inaccessible square, alter the Hamiltonian Array
      if (BOARD_SIZE%2 == 1 && (board->cell_value[0][BOARD_SIZE-1] == 20 ||board->cell_value[0][BOARD_SIZE-1] == 60)){
        arr[BOARD_SIZE-1][0] = BOARD_SIZE*BOARD_SIZE - BOARD_SIZE + 1;
        arr[BOARD_SIZE-2][1] = -1;
      }else if (BOARD_SIZE%2 == 1){
        arr[BOARD_SIZE-2][1] = BOARD_SIZE*BOARD_SIZE - BOARD_SIZE + 1;
      }

      coord[x] = board->snek->head->coord[x];
      coord[y] = board->snek->head->coord[y];
      int curr = arr[coord[y]][coord[x]];
      int found = 0;
      int newX, newY;

      int tailNum = arr[board->snek -> tail -> coord[x]][board->snek -> tail -> coord[y]]; //Associating tail coordinates to its Hamiltonian Number
      int headNum = arr[board->snek -> head -> coord[x]][board->snek -> head -> coord[y]]; //Associating head coordinates to its Hamiltonian Number
      if (headNum == -1){headNum = BOARD_SIZE*BOARD_SIZE - BOARD_SIZE +1;}
      //In the case where head is at the inaccessible square, make head take on the same value as the value before the Hamiltonian Array alteration

      int moogleNum;
      int greatest = -1;
      //Initilizing the neighbours
```

```
      int leftNum = -1;
      int rightNum = -1;
      int upNum = -1;
      int downNum = -1;

      if (MOOGLE_FLAG) { //If there is a target
        //Find Moogle's coordinates
        for(int i = 0; i < BOARD_SIZE; i++) {
          for(int j = 0; j < BOARD_SIZE; j++) {
            if(board->cell_value[i][j] > 1) {
              moogleLocX = j;
              moogleLocY = i;
            }
          }
        }
        moogleNum = arr[moogleLocX][moogleLocY]; //Find Moogle's Hamiltonian number

        int distTail = (headNum < tailNum) ? tailNum-headNum-1:tailNum-headNum-1+max_ham_num;
        //distTail calculates the number of cells that are outside of the head and the tail. distTail is the number of moves the snake takes to reach the tail by following the Hamiltonian path
        int distMoogle = (headNum < moogleNum) ? moogleNum-headNum-1:moogleNum-headNum-1+max_ham_num;
        //distMoogle is the number of moves that the snake can go to reach the Moogle by following the Hamiltonian path
        int available = distTail-3; //available is initially set to distTail - 3. 3 is a buffer.

        if(distMoogle < distTail){available--;} //This is saying if distMoogle < distTail, there is the possibility where after the snake eats a moogle, the head would collide with the tail. This happens when the Moogle is 1 cell away from the tail. As a preventative measure, subtract one from available.
        if(distMoogle < available){available = distMoogle;} //available is set to distMoogle if it is larger than distMoogle
        //Available is essentially the number of safe moves that the head of the snake can take by following the Hamiltonian path at this exact moment. It is <= min{distTail, distMoogle}. This is important to short cut finding because when a short cut is performed (a path that is not the Hamiltonian path), it is only safe to short cut as much as the number of moves the head can safely move by following a Hamiltonina path.
        if(available < 0){available = 0;} //If the available is below zero, this means that there is no possible short cut that is safe. In this case, the snake simply follows the regular Hamiltonian path

        //Getting the Hamiltonian numbers for the neighbours adjacent to the head
```

```
    //Getting the Hamiltonian numbers for the neighbours adjacent to the head
    if(coord[x] > 0 && board->occupancy[coord[y]][coord[x]-1] != 1)
      leftNum = arr[coord[x]-1][coord[y]];
    if(coord[x] < BOARD_SIZE-1 && board->occupancy[coord[y]][coord[x]+1] != 1)
      rightNum = arr[coord[x]+1][coord[y]];
    if(coord[y] > 0 && board->occupancy[coord[y]-1][coord[x]] != 1)
      upNum = arr[coord[x]][coord[y]-1];
    if(coord[y] < BOARD_SIZE-1 && board->occupancy[coord[y]+1][coord[x]] != 1)
      downNum = arr[coord[x]][coord[y]+1];

    int adjacent[] = {leftNum , rightNum, upNum, downNum};
    int greatestDist = -1;

    for(int i = 0; i < 4; i++) {
      int distance = (headNum < adjacent[i]) ? adjacent[i]-headNum-1 : adjacent[i]-headNum-1+max_ham_num;
      //distance is the distance between the head and its neighbours.
      if(adjacent[i] != -1 && distance <= available && distance > greatestDist) { //This determines the greatest distance that is also less than available.
      This allows the snake to select the neighbour that is the closest to the target yet it does not overshoot the target.
        greatestDist = distance;
        greatest = adjacent[i];
      }
    }

  }

  if(greatest == -1) //If no shortcut is found, follow normal Hamilatonian circuit path
    greatest = (headNum < max_ham_num) ? headNum + 1 : 1;

  //Converting Hamiltonian number into the corresponding game board coordinates
  for(newX = 0; newX < BOARD_SIZE && found == 0; newX += found == 0) {
    for(newY = 0; newY < BOARD_SIZE && found == 0; newY += found == 0) {
      if(greatest == arr[newX][newY])
        found = 1;
    }
  }
```

```

  //In the event the Hamiltonian circuit value is occupied, find an open neighbour
  if(board->occupancy[newY][newX] == 1) {
    newY = board->snek->head->coord[y];
    newX = board->snek->head->coord[x];
    if(newX > 0 && board->occupancy[newY][newX-1] != 1 && coord[x] != newX-1)
      newX--;
    else if(newX < BOARD_SIZE-1 && board->occupancy[newY][newX+1] != 1 && coord[x] != newX+1)
      newX++;
    else if(newY > 0 && board->occupancy[newY-1][newX] != 1 && coord[y] != newY-1)
      newY--;
    else if(newY < BOARD_SIZE-1 && board->occupancy[newY+1][newX] != 1 && coord[y] != newY+1)
      newY++;
  }

  //Based on the new coordinate values, assign the next axis and direction
  if(newX > coord[x]) {
    axis = AXIS_X;
    direction = RIGHT;
  }else if(newX < coord[x]) {
    direction = LEFT;
    axis = AXIS_X;
  }else if(newY > coord[y]) {
    axis = AXIS_Y;
    direction = DOWN;
  }else if(newY < coord[y]) {
    axis = AXIS_Y;
    direction = UP;
  }

  show_board(board);
  play_on = advance_frame(axis, direction, board);
```

```
    //Printing some information
    if (axis == AXIS_X){
      if (direction == RIGHT){
        printf("RIGHT");
      } else {
        printf("LEFT");
      }
    } else {
      if (direction == UP){
        printf("UP");
      } else {
        printf("DOWN");
      }
    } printf("\n");
    usleep(15000);
  }

  end_game(&board);
}

int main(){
    play_game();

    int score = SCORE;
    int numm = MOOGLES_EATEN;

    FILE *stream = fopen("performance.txt", "a");
    fprintf(stream, "%d\n", score);
    fclose(stream);

    FILE *stream1 = fopen("moogles.txt", "a");
    fprintf(stream1, "%d\n", numm);
    fclose(stream1);

    return 0;
```

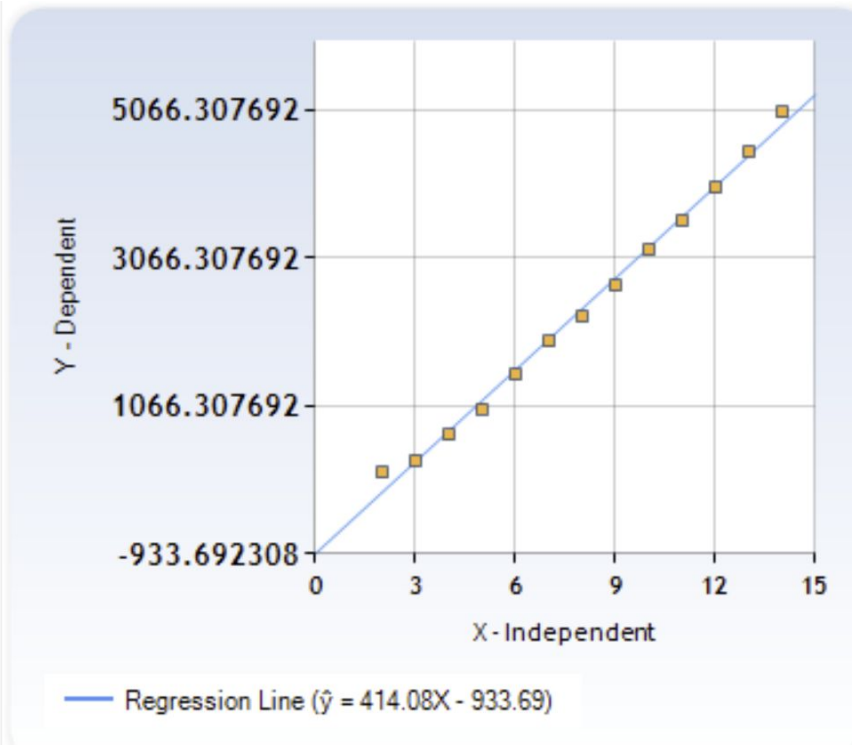
The code for the *snek_api.c* remained the same except for the following change:

```
  char blank =  43;
  char snek =   83;
  char snek_head = 72; //'H'
  char snek_tail = 84; //'C'
  int head_x = gameBoard->snek->head->coord[x];
  int head_y = gameBoard->snek->head->coord[y];
  int tail_x = gameBoard->snek->tail->coord[x];
  int tail_y = gameBoard->snek->tail->coord[y];
  char moogle = 88;

  for (int i = 0; i < BOARD_SIZE; i++){
    for (int j = 0; j < BOARD_SIZE; j++){
      if (gameBoard->occupancy[i][j] == 1){
        //snake is here
        if(i == head_y && j == head_x)
          fprintf(stdout, "%c", snek_head); //represent head as H
        else if(i == tail_y && j == tail_x)
          fprintf(stdout, "%c", snek_tail); //represent tail as T
        else
          fprintf(stdout, "%c", snek);
      } else if (gameBoard->cell_value[i][j] > 0) {
        //there be a moogle
        fprintf(stdout, "%c", moogle);
      } else {
        //nothing to see here
        fprintf(stdout, "%c", blank);
      }
    } //new line
    fprintf(stdout, "\n");

  }
```

Appendix C: Linear Regression of Score vs. Board Size



Appendix D: Valgrind Check

liuji241@remote:snek_finished

```
SSSSSSST+X
SSS++++SS+
SSSSSSSSSH
SSSSSS++SS
SSSSSS++SS
SS++SS++SS
SS++SS++SS
SS++SS++SS
SS++SS++++
++++SS++++


!..ALERT, MOOGLE IN VICINITY..!

SCORE: 3521
YOU HAVE EATEN 62 MOOGLES

SNEK HEAD       (9, 2)
SNEK TAIL       (7, 0)
LENGTH  63
CURR FRAME 55 vs TIME OUT 54



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

Your score: 3521



==41043==
==41043== HEAP SUMMARY:
==41043==     in use at exit: 0 bytes in 0 blocks
==41043==   total heap usage: 1,919 allocs, 1,919 frees, 32,608 bytes allocated
==41043==
==41043== All heap blocks were freed -- no leaks are possible
==41043==
==41043== For counts of detected and suppressed errors, rerun with: -v
==41043== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
[liuji241@remote snek_finished]$
```