Thursday, April 9th, 2020

SNEK REPORT

Xiyue Zhang, 1005839416

Introduction

The Snek project centers around a game played within a board. There exists a snake (Magini) who is constantly in inertial motion, as well as 2 possible stationary targets (Hurry Pooter and Moogles). Magini's goal is to maximize targets eaten (aka. points) before bumping into herself or a wall of the board, while also eating Moogles within a time limit. To accomplish this, a given API has been augmented to and an algorithm, using graphs, will be implemented to optimize the points that Magini can acquire before the game ends. The algorithm will be run 100+ times to obtain a dataset of final scores across trials. An additional augmentation will be implemented to demonstrate quantitatively, by running 50+ trials, the lower bound on time limits for eating Moogles that the algorithm can still work for beyond the initial set time limit.

Objectives

Objective 1: The algorithm prevents Magini from hitting herself or the edge of the board unless the head becomes trapped. As Magini grows in length, it becomes harder for her to move around the board without hitting herself. Eventually, Magini will grow too long to be able to maneuver within the constrained space. Thus, the objective is to have the algorithm avoid choosing paths that would cause collisions with herself or a wall, allowing more points to be earned.

Metric: The algorithm's efficiency of avoiding trapping Magini will be measured by looking at the average, highest, and lowest scores across 100+ trials. This is a fair measurement as the scores directly correlate to the length of the snek when the game ended. Thus, a higher score would mean the algorithm was more successful at avoiding trapping Magini.

Objective 2: "Eat" Moogles within minimum time possible. If a Moogle is not eaten within the time constraint, then the game ends. By minimizing the time it takes for Magini's head to reach the Moogle, it allows for a greater chance for the Moogle to be eaten in time and for the game to continue.

Metric: The efficiency of the algorithm at eating Moogles quickly will be measured by the percentage of deaths in the 100+ trials resulting from the time constraint. A smaller percentage of deaths by time constraint is preferred.

Objective 3: Minimize the lower bound of cycle allowance for which the algorithm still performs the same as with the original cycle allowance. An additional goal of the Snek project is to make the algorithm more adaptable to different time limits for when Magini must eat a Moogle. Initially the cycle allowance is set at 1.5, but allowing for a lower threshold of cycle allowance is preferable as it demonstrates algorithm quality and flexibility.

Metric: 50+ trials for various cycle allowances lower than 1.5 will be run to quantitatively find the lower bound for which the algorithm can still capture a comparable number of Moogles to the initial cycle allowance of 1.5. The average and standard deviation will be taken from the trials. The smallest cycle allowance whose average, with standard deviation, is within that of cycle allowance = 1.5 will be taken as the lower bound for when the algorithm still works comparably.

Objective 4: Minimize the time and space complexity of the algorithm. Reducing the time complexity will allow large numbers of trials to be run quicker. Reducing the space complexity reduces the amount of memory needed to run the algorithm, which allows for more trials to be run in extreme cases.

Metric: As the algorithm is centered around the utilization of a graph-search algorithm, the time complexity will be taken as only the time complexity of the graph-search algorithm. Similarly, the space complexity will be measured by the space complexity of the graph-search algorithm as well as the amount of auxiliary space the API modifications use.

Detailed Framework

The language used for this project is Python 3. The primary data structure is graphs and the algorithm is based around breadth-first search, to find a path from the head of Magini to Moogles.

High-Level Overview of Solution

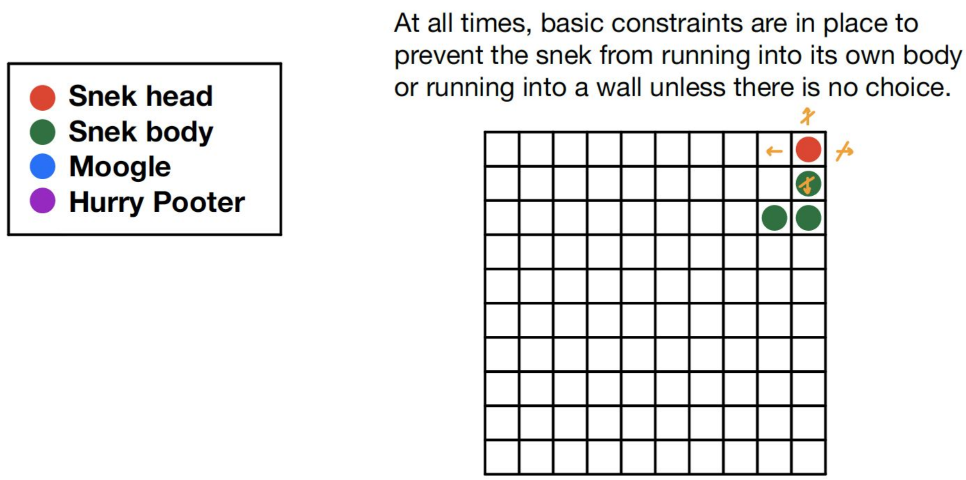


Fig. 1: High-level overview of solution (Pt. 1)

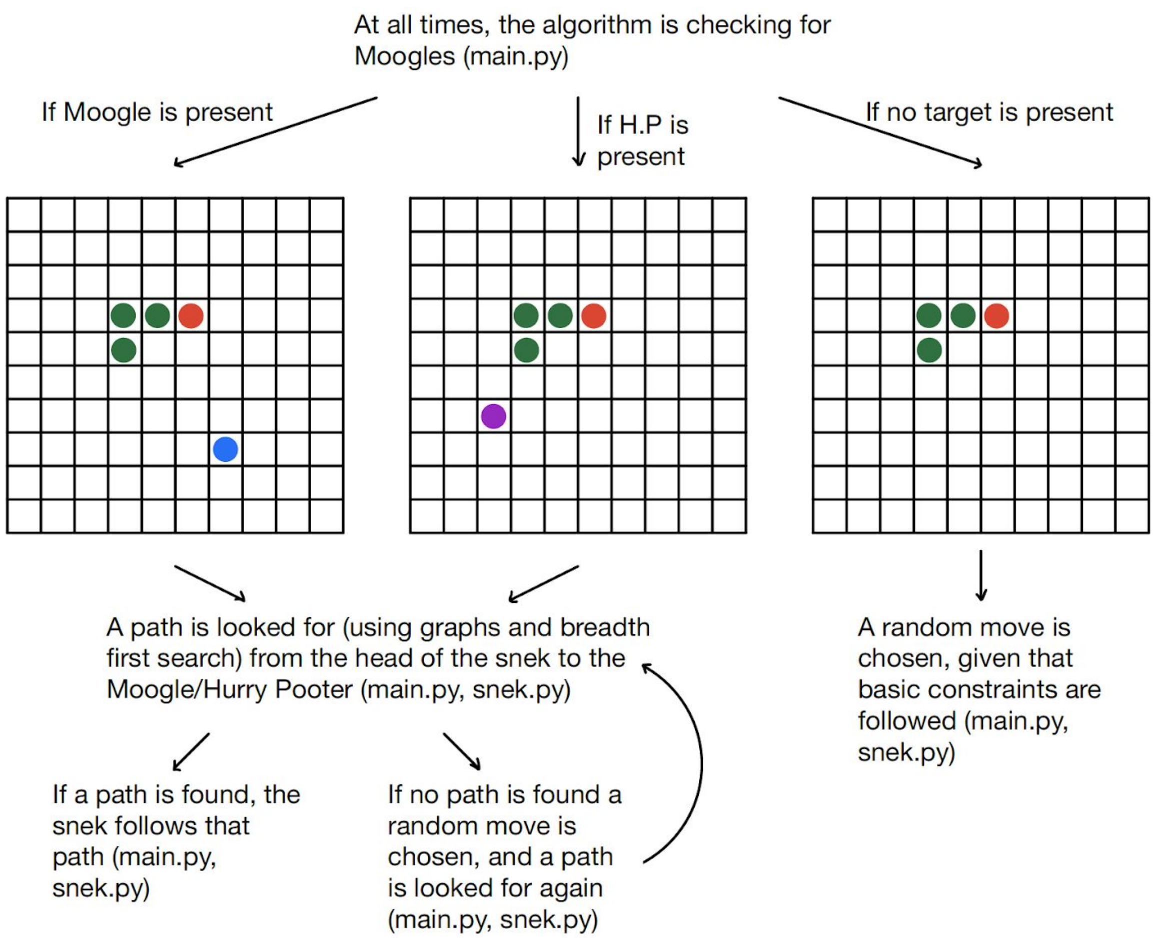


Fig. 2: High-level overview of solution (Pt. 2)

Modifications to Given API

The API was modified to accept and execute a sequence of moves in the game.

snek_api.h

- Line 55, *typedef struct Move:* data structure is declared

snek_api.c

- Line 238, *int get_moogle_flag():* identifies when a Moogle is present on the board
- Line 242, *Move* create_step(int axis, int direction, Move* next):* initializes a step and acts as a helper function for building a sequence of steps
- Line 286, *int move_steps(Move *steps, GameBoard *gameBoard):* moves Magini through a sequence of steps
- Line 298, *void free_steps(Move* steps):* frees allocated memory

Python Files

main.py: follows closely with the high-level overview of the solution illustrated in Fig. 1 and 2.

- Line 24, *if(get_moogle_flag(board)):* seeks to identify where the Moogle is on the board if a Moogle has been spotted
- Line 32, *g = create_graph(board):* creates the graph that will be used to find a path to the Moogle
- Line 35, *maps = breadth_first_search(g, start, head):* finds a path from Magini's head to the Moogle, if there exists one
- Line 37, *if(maps != None):* creates a sequence of steps that moves Magini to the Moogle. Calls the API
- Line 46, *else:* if no path is found, a random move is implemented, as long as the move does not violate any constraints that cause a self-collision or a collision into a wall

- Line 60, *else:* if no Moogle is found, a random move is also implemented, with the same considerations as the case in Line 46

snek.py: focuses on identifying where Magini should move next. Deals with "random" moves, creating graphs, and implementing the breadth-first search algorithm.

- Line 76, *class SquareGrid:* is the major data structure used to implement the path searching algorithm. It contains necessary information on the board, such as the boundaries and the position of Magini, as well as helper functions that facilitate path finding.
- Line 103, *class Queue:* collects the possible cells to explore in breadth-first search
- Line 138, *random_move:* is a function that returns a random move, based on the conditions surrounding Magini's head at the moment. It also checks for the move that will lead Magini's head furthest from her tail if more than one move is possible.
- Line 193, *create_graph(board):* is a function that uses the information from the board to create a graph
- Line 206, *breadth_first_search(graph, start, head):* is a function that is the breadth-first search algorithm which identifies and returns a path that could be found between Magini's head and a Moogle
- Line 224, *calculate_steps(maps, head, start):* is a function that translates the path information into a sequence of steps that gets executed

Identifying Range of Cycle Allowances

To demonstrate that the algorithm "works" for a range of cycle allowances, testing and data analysis will be done as stated in Objective 3, Metric. It is important to note this solution prioritizes getting Magini to the Moogles over preventing Magini from trapping herself, thus indicating that few high-level changes will need to be made, outside of testing different graph searches (Future Work), to further increase the range of cycle allowances that this solution would "work" in.

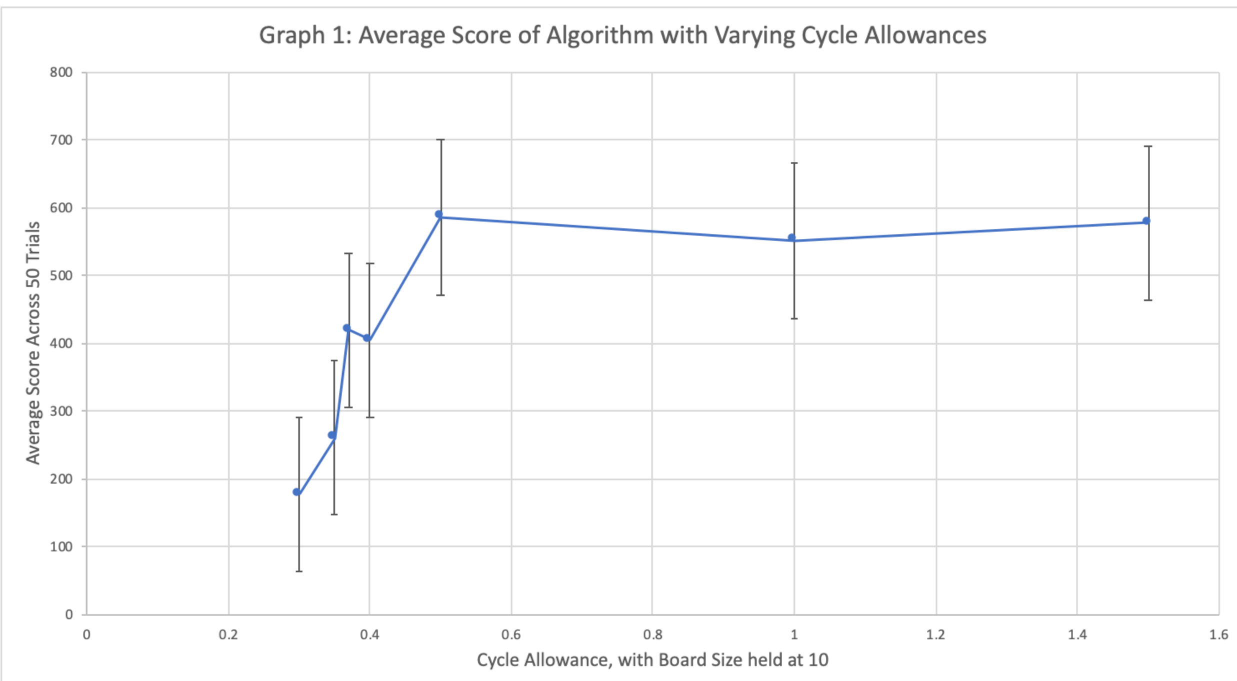
Results

Objective 1: after running 200 trials with cycle allowance 1.5 and board size 10,

- The lowest score: 28
- The average score: 577.235
- The highest score: 1247

Objective 2: after running 200 trials with cycle allowance 1.5 and board size 10, 58 trials ended from Magini not eating the Moogle within the time constraint. Therefore, the percentage of death by time constraint is 29%.

Objective 3: After running 50 trials each for cycle allowances of 0.3, 0.35, 0.37, 0.4, 0.5, 1.0, and 1.5, the average scores and standard deviation were recorded and graphed below:



As can be seen on Graph 1, 0.37 is the lowest tested cycle allowance that the algorithm still performs comparably in, as outlined by the metric for Objective 3.

Objective 4:

- The time complexity of a breadth-first search is O(|V|+|E|), where V is the number of vertices and E is the number of edges. Thus, it is dependent on the size of the board.
- The space complexity of a breadth-first search is O(|V|) which is the cardinality of the set of vertices. Again, this is dependent on the size of the board.
- The auxiliary space the API modifications uses is O(n), with n being the number of *Move*s (the data structure) being used.

Results of Described Operations in Project Specification Handout

- The expected score of this solution will be taken as the mean score from 200 trials tested with cycle allowance 1.5 and board size 10, rounded to the nearest integer: 577
- The variance will be taken as the standard deviation: 258.04
- The minimum guaranteed score:
    - Theoretically, should be 60. A cycle allowance of 1.5 is sufficient to move the snek to anywhere on the board and the snek would need to be at least length 4 in order to trap itself. Since each Moogle is worth 20 points, 20 x 3 gives 60 as the lowest theoretical score
    - Experimentally, after 200 trials, was 28. This indicates that there still exist bugs in the program.
- The smallest cycle allowance, demonstrated quantitatively through Results, Objective 3: 0.37

Future Work / Conclusion

Out of the 2 ways that the game could end, either from self collision / with a wall or not eating Moogles in time, only the latter was addressed in depth within this algorithm. There are constraints in place that prevent Magini from choosing a step that would cause collision, but the constraint extends only to the immediate step. There isn’t deep consideration for future steps to ensure that once a step is taken, the next step isn’t limited to the point of the game ending.

Future work would address the need to implement more constraints that would consider not just the immediate step, but also future steps. As there is a constant trade-off between wanting to take the longest path (to avoid self-collision) and taking the shortest path (to eat Moogles quickly), next steps will need to be taken in consideration of both objectives.

Next Steps

1. Researching and testing different graph-searching algorithms, such as depth-first and Dijkstra's algorithm, keeping in mind efficiency at finding the quickest path and time and space complexity.
2. Successfully implementing the Hamitonian cycle to fully address the objective of avoiding self collision and collision with walls.
3. Adapting the Hamitonian cycle to include shortcuts (identified by a chosen graph-searching algorithm from Step 1) when appropriate which would allow for Magini to get to Moogles quicker.

Conclusion

Reaching Moogles quickly (Objective 2) was prioritized and for the majority of the trials, Magini ate the Moogle on time. Avoiding self-collisions and collisions with walls (Objective 1) was also considered, with basic safeguards put in place to avoid immediate collision unless necessary. The average score of 577 indicates that Magini was usually able to survive several frames before collision, which caused 71% of game-overs. Time and space complexity (Objective 4) was not specifically optimized, with time complexity being considered in Next Steps.

Appendix

snek_api.h

```c
/**
AUTHOR: SAIMA ALI
LATEST WORKING VERSION
FEBRUARY 2ND, 2020
ESC190H1S PROJECT
SNAKE API
 **/

#include <stdlib.h>
#include <stdio.h>

#define CYCLE_ALLOWANCE 1.5 // change this to expand tests
#define BOARD_SIZE 10 // change this to expand tests

#define LIFE_SCORE 1 //score awarded for simply staying alive one frame

#define AXIS_X -1
#define AXIS_Y 1

#define UP -1
#define DOWN 1
#define LEFT -1
#define RIGHT 1

#define AXIS_INIT AXIS_Y
#define DIR_INIT DOWN

#define x 0
#define y 1

#define MOOGLE_POINT 20
#define HARRY_MULTIPLIER 3

int CURR_FRAME;
int SCORE;
int MOOGLE_FLAG;

typedef struct SnekBlock{
	int coord[2];
	struct SnekBlock* next;
} SnekBlock;
```

```c

typedef struct Snek{
    struct SnekBlock* head;
    struct SnekBlock* tail;
    int length;
} Snek;

typedef struct GameBoard {
    int cell_value[BOARD_SIZE][BOARD_SIZE];
    int occupancy[BOARD_SIZE][BOARD_SIZE];
    struct Snek* snek;
} GameBoard;

typedef struct Move {
    int axis;
    int direction;
    struct Move* next;
} Move;


GameBoard *init_board(void);
Snek *init_snek(int a, int b);
int hits_edge(int axis, int direction,  GameBoard *gameBoard);
int hits_self(int axis, int direction,  GameBoard *gameBoard);
int is_failure_state(int axis, int direction,  GameBoard *gameBoard);
int advance_frame(int axis, int direction,  GameBoard *gameBoard);
void end_game(GameBoard **board);
void show_board(GameBoard* gameBoard);
Move* create_step(int axis, int direction, Move* next);
int get_score(void);
int move_steps(Move *steps, GameBoard *gameBoard);
int get_moogle_flag(void);
void free_steps(Move *steps);
```

snek_api.c

```c
/** snake API for C **/
// need to add the elongating of snek after nomming
#include "snek_api.h"
#include <string.h>
#include <time.h>

//extern int CURR_FRAME;
//extern int SCORE;
//extern int MOOGLE_FLAG;
int MOOGLES_EATEN = 0;
int TIME_OUT = ((BOARD_SIZE * 4) - 4) * CYCLE_ALLOWANCE;

GameBoard* init_board(){
    srand(time(0));
    GameBoard* gameBoard = (GameBoard*)(malloc(sizeof(GameBoard)));
    for (int i = 0; i < BOARD_SIZE; i++){
        for (int j = 0; j < BOARD_SIZE; j++){
            gameBoard->cell_value[i][j] = 0;
            gameBoard->occupancy[i][j] = 0;
        }
    }
    gameBoard->occupancy[0][0] = 1; //snake initialized
    gameBoard->snek = init_snek(0, 0);
    CURR_FRAME = 0;
    SCORE = 0;
    MOOGLE_FLAG = 0;

    return gameBoard;
}

Snek* init_snek(int a, int b){
    Snek* snek = (Snek *)(malloc(sizeof(Snek)));

    snek->head = (SnekBlock *)malloc(sizeof(SnekBlock));
    snek->head->coord[x] = a;
    snek->head->coord[y] = b;

    snek->tail = (SnekBlock *)malloc(sizeof(SnekBlock));
    snek->tail->coord[x] = a;
    snek->tail->coord[y] = b;

    snek->tail->next = NULL;
    snek->head->next = snek->tail;

    snek->length = 1;

```

```
    return snek;
}

int hits_edge(int axis, int direction, GameBoard* gameBoard){
    if (((axis == AXIS_Y) && ((direction == UP && gameBoard->snek->head->coord[y] + UP < 0) || (direction == DOWN \
    && gameBoard->snek->head->coord[y] + DOWN > BOARD_SIZE - 1)))
       || (axis == AXIS_X && ((direction == LEFT && gameBoard->snek->head->coord[x] + LEFT < 0) || (direction == RIGHT \
       && gameBoard->snek->head->coord[x] + RIGHT > BOARD_SIZE-1))))
    {
        return 1;
    } else {
        return 0;
    }

}


int hits_self(int axis, int direction, GameBoard *gameBoard){
    int new_x, new_y;
    if (axis == AXIS_X){
        new_x = gameBoard->snek->head->coord[x] + direction;
        new_y = gameBoard->snek->head->coord[y];
    } else if (axis == AXIS_Y){
        new_x = gameBoard->snek->head->coord[x];
        new_y = gameBoard->snek->head->coord[y] + direction;
    }
    if ((gameBoard->snek->length != 1) &&
        (new_y == gameBoard->snek->tail->coord[y] && new_x == gameBoard->snek->tail->coord[x]))
    {
        return 0; //not hit self, this is the tail which will shortly be moving out of the way
    } else {
        return gameBoard->occupancy[new_y][new_x]; //1 if occupied
    }
}

int time_out(){
    return (MOOGLE_FLAG == 1 && CURR_FRAME > TIME_OUT);
}

int is_failure_state(int axis, int direction, GameBoard *gameBoard){
    return (hits_self(axis, direction, gameBoard) || hits_edge(axis, direction, gameBoard) || time_out());
}
```

```c

void populate_moogles(GameBoard *gameBoard){
    if (MOOGLE_FLAG == 0){
        int r1 = rand() % BOARD_SIZE;
        int r2 = rand() % BOARD_SIZE;

        int r3 = rand() % (BOARD_SIZE * 10);
        if (r3 == 0){
            gameBoard->cell_value[r1][r2] = MOOGLE_POINT * HARRY_MULTIPLIER;
            MOOGLE_FLAG = 1;
        } else if (r3 < BOARD_SIZE){
            gameBoard->cell_value[r1][r2] = MOOGLE_POINT;
            MOOGLE_FLAG = 1;
        }
    }
}

void eat_moogle(GameBoard* gameBoard, int head_x, int head_y) {
    SCORE = SCORE + gameBoard->cell_value[head_y][head_x];
    gameBoard->cell_value[head_y][head_x] = 0;

    gameBoard->snek->length ++;
    MOOGLES_EATEN ++;
    MOOGLE_FLAG = 0;
    CURR_FRAME = 0;
}

int advance_frame(int axis, int direction, GameBoard *gameBoard){
    if (is_failure_state(axis, direction, gameBoard)){
        return 0;
    } else {
        // update the occupancy grid and the snake coordinates
        int head_x, head_y;
        // figure out where the head should now be
        if (axis == AXIS_X) {
            head_x = gameBoard->snek->head->coord[x] + direction;
            head_y = gameBoard->snek->head->coord[y];
        } else if (axis == AXIS_Y){
            head_x = gameBoard->snek->head->coord[x];
            head_y = gameBoard->snek->head->coord[y] + direction;
        }
        else{
            head_x = 0;
            head_y = 0;
        }
```

```c

    int tail_x = gameBoard->snek->tail->coord[x];
    int tail_y = gameBoard->snek->tail->coord[y];

    // update the occupancy grid for the head
    gameBoard->occupancy[head_y][head_x] = 1;

    if (gameBoard->snek->length > 1) { //make new head
        SnekBlock *newBlock = (SnekBlock *)malloc(sizeof(SnekBlock));
        newBlock->coord[x] = gameBoard->snek->head->coord[x];
        newBlock->coord[y] = gameBoard->snek->head->coord[y];
        newBlock->next = gameBoard->snek->head->next;

        gameBoard->snek->head->coord[x] = head_x;
        gameBoard->snek->head->coord[y] = head_y;
        gameBoard->snek->head->next = newBlock;

        if (gameBoard->cell_value[head_y][head_x] > 0){  //eat something
            eat_moogle(gameBoard, head_x, head_y);
        } else { //did not eat
            //delete the tail
            gameBoard->occupancy[tail_y][tail_x] = 0;
            SnekBlock *currBlock = gameBoard->snek->head;
            while (currBlock->next != gameBoard->snek->tail){
                currBlock = currBlock->next;
            } //currBlock->next points to tail

            currBlock->next = NULL;
            free(gameBoard->snek->tail);
            gameBoard->snek->tail = currBlock;
        }

    } else if ((gameBoard->snek->length == 1) && gameBoard->cell_value[head_y][head_x] == 0){ // change both head and tail coords, head is tail
        gameBoard->occupancy[tail_y][tail_x] = 0;
        gameBoard->snek->head->coord[x] = head_x;
        gameBoard->snek->head->coord[y] = head_y;
        gameBoard->snek->tail->coord[x] = head_x;
        gameBoard->snek->tail->coord[y] = head_y;

    } else { //snake is length 1 and eats something
        eat_moogle(gameBoard, head_x, head_y);
        gameBoard->snek->head->coord[x] = head_x;
        gameBoard->snek->head->coord[y] = head_y;
    }
```

```c

        // update the score and board
        SCORE = SCORE + LIFE_SCORE;
        if (MOOGLE_FLAG == 1){
            CURR_FRAME ++;
        }

        // populate moogles
        populate_moogles(gameBoard);
        return 1;
    }
}

void show_board(GameBoard* gameBoard) {
    //fprintf(stdout, "\033"); // clear terminal ANSI code
    //fprintf(stdout, "\033"); // reset cursor position

    char blank =    43;
    char snek =     83;
    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
                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");

    }

    fprintf(stdout, "\n\n");
```

```
    if (MOOGLE_FLAG == 1){
        fprintf(stdout, "!..ALERT, MOOGLE IN VICINITY..!\n\n");
    }
    fprintf(stdout, "SCORE: %d\n", SCORE);
    fprintf(stdout, "YOU HAVE EATEN %d MOOGLES\n\n", MOOGLES_EATEN);

    fprintf(stdout, "SNEK HEAD\t(%d, %d)\n", gameBoard->snek->head->coord[x], gameBoard->snek->head->coord[y]);
    fprintf(stdout, "SNEK TAIL\t(%d, %d)\n", gameBoard->snek->tail->coord[x], gameBoard->snek->tail->coord[y]);
    fprintf(stdout, "LENGTH \t%d\n", gameBoard->snek->length);
    fprintf(stdout, "CURR FRAME %d vs TIME OUT %d\n", CURR_FRAME, TIME_OUT);


    fflush(stdout);
}

int get_score() {
    return SCORE;
}

// added
int get_moogle_flag(){
    return MOOGLE_FLAG;
}

Move* create_step(int axis, int direction, Move* next){
    Move* step = (Move*)malloc(sizeof(Move));
    step->axis = axis;
    step->direction = direction;
    step->next = next;
    return step;
}
```

```c
void end_game(GameBoard **board){
    // fprintf(stdout, "\033");
    // fprintf(stdout, "\033");

    // ---------- prints out score; careful to delete after use ----------
    FILE *fp = fopen("output.txt", "a");
    if (fp == NULL){
        fprintf(stdout, "No such file\n");
    }
    else{
        fprintf(fp, "%d\n", SCORE);
    }
    fclose(fp);
    // -------------------------------------------------------------------

    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
    SnekBlock **snekHead = &((*board)->snek->head);
    SnekBlock *curr;
    SnekBlock *prev = NULL;
    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);
}
```

```c

int move_steps(Move *steps, GameBoard *gameBoard){
    while (steps != NULL){
        int con = advance_frame(steps->axis, steps->direction, gameBoard);
        // in case something goes wrong
        if (con == 0){
            return con;
        }
        steps = steps->next;
    }
    return 1;
}

void free_steps(Move* steps){
    Move* curr;
    while(steps != NULL){
        curr = steps;
        steps = steps->next;
        free(curr); // have to test Valgrind on this
    }
}
```

main.py

```python
from snek import *
from time import sleep

if __name__ == "__main__":
    # run 100 trials
    for trial in range(0, 100):
        # ptr to board
        board = init_board()

        play_on = 1
        show_board(board)
        axis = AXIS_INIT
        direction = DIR_INIT
        head_coord = [0, 0]
        tail_coord = [0, 0]
        moogle_coord = [0, 0]
        while (play_on):
            # indexing at 0 dereferences the pointer
            head_coord[x],head_coord[y] = board[0].snek[0].head[0].coord[x], \
                                          board[0].snek[0].head[0].coord[y]
            tail_coord[x],tail_coord[y] = board[0].snek[0].tail[0].coord[x], \
                                          board[0].snek[0].tail[0].coord[y]
            # looks for Moogle
            if(get_moogle_flag(board)):
                for i in range(0, BOARD_SIZE):
                    for j in range(0, BOARD_SIZE):
                        # not differentiating between Harry Potter and Moogle
                        if board[0].cell_value[i][j] > 1:
                            moogle_coord[x] = j
                            moogle_coord[y] = i
                # find paths to Moogle/Harry Potter
                g = create_graph(board)
                start = (moogle_coord[x], moogle_coord[y])
                head = (head_coord[x], head_coord[y])
                maps = breadth_first_search(g, start, head)
                # if path is found
                if (maps != None):
                    step1 = calculate_steps(maps, head, start)
                    if(step1 != None):
                        play_on = move_steps(step1, board)
                        free_steps(step1)
                    # theoretically will not go here:
                    else:
                        play_on = 0
```

```python
                # if no path is found
                else:
                    any_step = random_move(head_coord, tail_coord, board)
                    if any_step == 0:
                        play_on = advance_frame(AXIS_X, LEFT, board)
                    elif any_step == 1:
                        play_on = advance_frame(AXIS_X, RIGHT, board)
                    elif any_step == 2:
                        play_on = advance_frame(AXIS_Y, UP, board)
                    elif any_step == 3:
                        play_on = advance_frame(AXIS_Y, DOWN, board)
                    # nowhere to go. suicide
                    else:
                        play_on = advance_frame(AXIS_X, RIGHT, board)
            # if no moogle on board
            else:
                random_move_step = random_move(head_coord, tail_coord, board)
                if random_move_step == 0:
                    play_on = advance_frame(AXIS_X, LEFT, board)
                elif random_move_step == 1:
                    play_on = advance_frame(AXIS_X, RIGHT, board)
                elif random_move_step == 2:
                    play_on = advance_frame(AXIS_Y, UP, board)
                elif random_move_step == 3:
                    play_on = advance_frame(AXIS_Y, DOWN, board)
                # nowhere to go. suicide
                else:
                    play_on = advance_frame(AXIS_X, RIGHT, board)
            show_board(board)
            sleep(0.05)
        #pass by reference to clean memory
        end_game(byref(board))
    print("The 100 trials are finished")
```

snek.py

```python
'''
February 9, 2020
Saima Ali
Porting the Snek API in C to Python
Tested in the ESC190 VM

In terminal, run
>>> python3 main.py

If you change the board size here,
you will have to modify snek_api.h
and recompile.
'''
import sys
import collections
from ctypes import *
BOARD_SIZE = 10
INFINITY = sys.maxsize  # added

# do not modify --------------------
x = 0
y = 1

AXIS_X = -1
AXIS_Y = 1

UP = -1
DOWN = 1
LEFT = -1
RIGHT = 1

AXIS_INIT = AXIS_Y
DIR_INIT = DOWN
# ----------------------------------

# import the library
# dependant on directory structure
snek_lib = CDLL("./libsnek_py.so")

class SnekBlock(Structure):
    # has ptr to itself, need to declare fields later
    pass

SnekBlock._fields_ = [('coord', c_int * 2), ('next', POINTER(SnekBlock))]
```

```python

class Snek(Structure):
    _fields_ = [('head', POINTER(SnekBlock)), \
                ('tail', POINTER(SnekBlock)), \
                ('length', c_int)]

class GameBoard(Structure):
    _fields_ = [('cell_value', (c_int * BOARD_SIZE) * BOARD_SIZE), \
                ('occupancy', (c_int * BOARD_SIZE) * BOARD_SIZE), \
                ('snek', POINTER(Snek))]

    def __repr__(self):
        #don't need this, print(board[0]) does work though
        #left as a reference for how to access GameBoard attributes
        s = ''
        for i in range(0, BOARD_SIZE):
            for j in range(0, BOARD_SIZE):
                if self.occupancy[i][j] == 1:
                    s += 'S'
                elif self.cell_value[i][j] != 0:
                    s += 'X'
                else:
                    s += '+'
            s += '\n'
        return s

class Move(Structure):
    pass
Move._fields_ = [('axis', c_int), ('direction', c_int), ('next', POINTER(Move))]


class SquareGrid:
    def __init__(self, boardsize):
        self.width = boardsize
        self.height = boardsize
        self.walls = []

    # check if within board
    def in_bounds(self, id):
        (x, y) = id
        return 0 <= x < self.width and 0 <= y < self.height

    # check if will run into snek
    def passable(self, id):
```

```python
        return id not in self.walls

    def neighbors(self, id):
        (x, y) = id
        # possible next steps
        results = [(x + 1, y), (x, y - 1), (x - 1, y), (x, y + 1)]
        if (x + y) % 2 == 0: results.reverse()  # aesthetics
        # filter out bad possibilities
        results = filter(self.in_bounds, results)
        results = filter(self.passable, results)
        return results


# data structure used in breadth-first algorithm
class Queue:
    def __init__(self):
        self.elements = collections.deque()

    def empty(self):
        return len(self.elements) == 0

    def put(self, x):
        self.elements.append(x)

    def get(self):
        return self.elements.popleft()


def wrap_func(lib, funcname, restype, argtypes):
    ''' Referenced from
    https://dbader.org/blog/python-ctypes-tutorial-part-2
    '''
    func = lib.__getattr__(funcname)
    func.restype = restype
    func.argtypes = argtypes
    return func
```

```python

init_board = wrap_func(snek_lib, 'init_board', POINTER(GameBoard), [])
show_board = wrap_func(snek_lib, 'show_board', None, [POINTER(GameBoard)])
advance_frame = wrap_func(snek_lib, 'advance_frame', c_int, [c_int, c_int, POINTER(GameBoard)])
end_game = wrap_func(snek_lib, 'end_game', None, [POINTER(POINTER(GameBoard))])
get_score = wrap_func(snek_lib, 'get_score', c_int, [])
get_moogle_flag = wrap_func(snek_lib, 'get_moogle_flag', c_int, [])
move_steps = wrap_func(snek_lib, 'move_steps', c_int, [POINTER(Move), POINTER(GameBoard)])
create_step = wrap_func(snek_lib, 'create_step', POINTER(Move), [c_int, c_int, POINTER(Move)])
free_steps = wrap_func(snek_lib, 'free_steps', None, [POINTER(Move)])


# Functions to play game
def random_move(head, tail, board):
    ok_to_move = 0
    dist = [0, 0, 0, 0]
    temp = [0, 0]

    # going left
    temp[x] = head[x] - 1
    temp[y] = head[y]
    if(temp[x] < 0 or board[0].occupancy[temp[y]][temp[x]]) == 1:
        dist[0] = INFINITY
    else:
        dist[0] = abs(temp[x]-tail[x]) + abs(temp[y]-tail[y])
        ok_to_move = 1

    # going right
    temp[x] = head[x] + 1
    temp[y] = head[y]
    if (temp[x] == BOARD_SIZE or board[0].occupancy[temp[y]][temp[x]]) == 1:
        dist[1] = INFINITY
    else:
        dist[1] = abs(temp[x] - tail[x]) + abs(temp[y] - tail[y])
        ok_to_move = 1

    # going up
    temp[x] = head[x]
    temp[y] = head[y] - 1
    if (temp[y] < 0 or board[0].occupancy[temp[y]][temp[x]]) == 1:
        dist[2] = INFINITY
```

```
    else:
        dist[2] = abs(temp[x] - tail[x]) + abs(temp[y] - tail[y])
        ok_to_move = 1

    # going down
    temp[x] = head[x]
    temp[y] = head[y] + 1
    if (temp[y] == BOARD_SIZE or board[0].occupancy[temp[y]][temp[x]]) == 1:
        dist[3] = INFINITY
    else:
        dist[3] = abs(temp[x] - tail[x]) + abs(temp[y] - tail[y])
        ok_to_move = 1

    direction = 0
    temp_dist = 0
    if ok_to_move == 0:
        return 4  # no option but to end the game
    else:
        # check for option that will lead further away from snek tail
        for i in range(4):
            if dist[i] != INFINITY and dist[i] > temp_dist:
                direction = i
                temp_dist = dist[i]
        return direction


# creates graph for breadth_first algorithm (helper function)
def create_graph(board):
    g = SquareGrid(BOARD_SIZE)
    #create a list to include all locations accupied by snake
    wall = []
    head_coord = (board[0].snek[0].head[0].coord[x], board[0].snek[0].head[0].coord[y])
    for i in range(BOARD_SIZE):
        for j in range(BOARD_SIZE):
            if board[0].occupancy[i][j] == 1 and (j, i) != head_coord:
                wall.append((j, i))
    g.walls = wall
    return g
```

```python
def breadth_first_search(graph, start, head):
    frontier = Queue()
    frontier.put(start)
    maps = {}
    maps[start] = None

    while not frontier.empty():
        current = frontier.get()
        if current == head:
            return maps
        for next in graph.neighbors(current):
            if next not in maps:
                frontier.put(next)
                maps[next] = current
    return None


# execute path gotten from algorithm
def calculate_steps(maps, head, start):
    path = []
    path.append(head)
    x = maps.get(head)
    if x == None:
        return None
    while x != start and x != None:
        path.append(x)
        x = maps.get(x)
    if x == start:
        path.append(start)
    else:
        return None
    path.reverse()
    steps = []
    axis = AXIS_X
    direction = RIGHT

    # the last step
    if path[1][0] == path[0][0]:
        axis = AXIS_Y
        if path[1][1] > path[0][1]:
            direction = UP
        else:
            direction = DOWN
```

```
    elif path[1][0] > path[0][0]:
        axis = AXIS_X
        direction = LEFT
    else:
        axis = AXIS_X
        direction = RIGHT
    step = create_step(axis, direction, None)
    steps.append(step)

    for i in range(1, len(path)):
        if path[i] == head:
            return steps[i-1]
        else:
            if path[i+1][0] == path[i][0]:
                axis = AXIS_Y
                if path[i+1][1] > path[i][1]:
                    direction = UP
                else:
                    direction = DOWN
            elif path[i+1][0] > path[i][0]:
                axis = AXIS_X
                direction = LEFT
            else:
                axis = AXIS_X
                direction = RIGHT
            step = create_step(axis, direction, steps[i-1])
            steps.append(step)
    return None
```