Snek Game Final Report

April 9, 2020

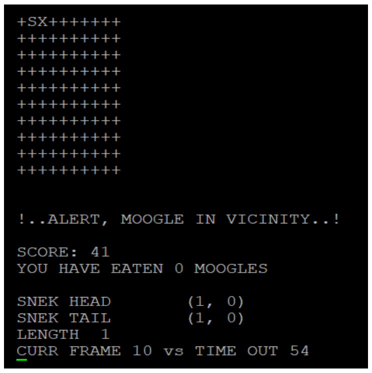
Hshmat Sahak

1005903710

Introduction:

Your name is Voldy and you wish to navigate a snake (set of connected cells) named “Magini” around a square playing field without it [1]colliding with the edges of the grid and [2]colliding with itself.

Magini initially assumes a position of (0,0) and length 1 (sample playing field and coordinate convention shown below). Magini has an “inertia” - it will move undisturbed in some direction until instructed otherwise. To command magini, the API must be able to accept a sequence of moves and execute them. Each move must specify an axis {AXIS_X, AXIS_Y} and one of four directions {Up, Down, Left, Right}



```
+SX++++++++
++++++++++
++++++++++
++++++++++
++++++++++
++++++++++
++++++++++
++++++++++
++++++++++
++++++++++


!..ALERT, MOOGLE IN VICINITY..!

SCORE: 41
YOU HAVE EATEN 0 MOOGLES

SNEK HEAD        (1, 0)
SNEK TAIL        (1, 0)
LENGTH  1
CURR FRAME 10 vs TIME OUT 54
```

Figure 1:Sample API Playing Field

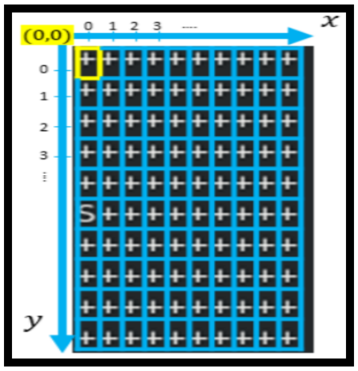
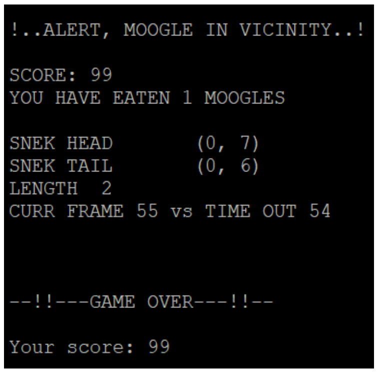


Figure 2: Coordinate System Convention



```
!..ALERT, MOOGLE IN VICINITY..!

SCORE: 99
YOU HAVE EATEN 1 MOOGLES

SNEK HEAD        (0, 7)
SNEK TAIL        (0, 6)
LENGTH  2
CURR FRAME 55 vs TIME OUT 54



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

Your score: 99
```

Figure 3: Sample End-Game State

Points are awarded as follows: one point every time the head moves one cell(LIFE_SCORE), 20 points for eating a *Moogle* and 60 points for eating a *Hurry Pooter*. There can only be one Moogle in the playing field at a time; after consuming one, Magini grows in length by 1 and the chance of a target appearing exists(10% chance of Moogle, 1% chance of Hurry Pooter).

Failure to satisfy [1] or [2] results in losing the game, and Magini is said to die. The game also ends when a moogle appears and Magini does not reach it within a TIMEOUT(Figure 3). This is related mathematically to the perimeter by the global variables BOARD_SIZE and CYCLE_ALLOWANCE as (int)((4 * BOARD_SIZE – 4) * CYCLE_ALLOWANCE).

Task

The task assigned is to design and implement an algorithm to maximize the number of points Magini acquires before the game ends. The efficacy of the algorithm will be assessed quantitatively by running 1000 trials of the algorithm across the default BOARD SIZE of 10. The algorithm will also be assessed for its effectiveness across varying board sizes.

Several methods/algorithms exist that can be incorporated when maximizing points acquired:

- Dijkstra's Algorithm finds the shortest path between the head of the snake and the food. This ensures we reach the food in time, but does not ensure maximum possible score.
- The Hamiltonian Cycle will identify a path that passes through each vertex exactly once. This can be used to ensure that the snake reaches the food, but does not ensure it does so within the TIME_OUT. For larger cycle allowances, restricting cells to be visited by the head only once will give relatively low scores.
- Genetic AI is an idea that requires learning outside the scope of ESC190. Essentially, we can train an AI to learn how to play the game most effectively. I possess little knowledge of this field; thus, it was scoped out.

The algorithm designed for maximizing points is called Randomized Path Finding Algorithm(RPFA), which involves a stack of game states intended to allow the AI to compute the snake's position a number of moves ahead. While this is not always the maximum allowable number of moves, it is clearly better than a shortest path search, and will intuitively yield a better runtime than longest path algorithms.

In the rest of the report, we will examine the objectives for the piece of software, the detailed framework behind the design, the algorithm's results with respect to each of our metrics and future work. We will also present the bottlenecks in the current algorithm and how they may be addressed.

Objectives:

Objective 1[O1]: Maximize score

Metric: Minimum, Average and Range of Scores across 1000 trials on the default BOARD_SIZE

Justification: Direct computation of a mean score will give us an expected value for our algorithm's performance. Minimum score will give us the least score our algorithm is guaranteed to accomplish. The range will give us a sense of the reliability of the algorithm

Objective 2[O2]: Smaller Runtime

Metric: Maximum and Average runtimes across 1000 trials on the default BOARDSIZE

Justification: Direct computation of mean runtime will give us an expected value for our algorithm's runtime. Maximum runtime will give us the longest time our algorithm will take(worst case runtime).

Objective 3[O3]: Smaller Space Complexity

Metric: Tight asymptotic bound in worst case

Justification: An efficient algorithm uses less storage space in memory than other solutions. We prefer a smaller space complexity so as to not use up a lot of resources in computer memory

Objective 4[O4]: Flexibility

Metric: Range of board sizes that the algorithm works for, and upper bound.

Justification: A lower-level objective is to make the algorithm flexible for multiple playing field sizes. This was selected as an augmentation to the existing API.

Objective 5[O5]: Readability

Metric: Can be measured as the fraction of functions implemented that are commented.

Justification: The code must be readable to allow myself and markers to understand my algorithm and the purpose of each function.

Detailed Framework.

Data Structure

The data structure implemented was a stack. From the study of graphs, we are aware that DFS is used in finding paths of any length; so we are motivated by its flexibility to find paths of any length for the snake[O4]. Since a DFS is guaranteed to find a path, and that path is the first it encounters, we expect it to yield high scores[O1] small runtimes[O2].

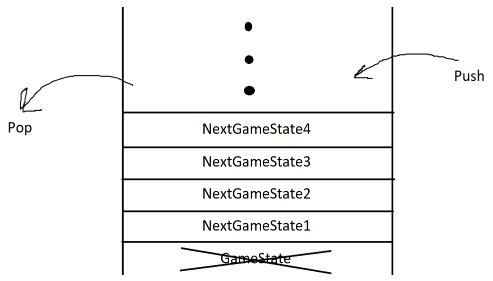


Figure 4: The game stack

RPFA is similar to DFS in graphs-exploring if a sequence of moves leads to the target. Only when one search fails will we backtrack and look for other paths. For example, in the figure to the left, the initial game state has been popped and the 4 game states obtained by moving one of {left, right, up, down} have been pushed. In the next move, next gamestate 4 will be popped.

Language Selected:

Python was selected for various reasons. First, it allowed us to take advantage of object-oriented programming; classes are used to produce a gamestate object, a stack object and an object for storing computer moves. Second, using Python enabled usage of built-in list methods. As our stack was implemented as a Python list, stack operations such as pushing and popping were made very easy. Finally, using Python scoped out checking for a clean valgrind.

<u>Software Implementation and Modifications to the API:</u>

The algorithm designed depends on a <u>stack of game states</u> that allow a depth first search to find (1) a random path to the target with distance less than the TIMEOUT if the target exists and (2) any viable path of a variable length if a target doesn't exist. The first case is addressed by the method *goforn (*board, path_length)* and the second by the method *getfirstn (*board, path_length)*, both in Operate_Stack.py.

The class definition for the stack is found in Stack_Moves.py, and is similar to the class definition for any ordinary stack, except it has a method for finding the *depth* of the stack from any element; that is, the number of moves required to reach that gamestate from starting position.

The purpose of a GameState object is to store playing field information at every move. Specifically, it stores a copy of the pointer to the game board, whether or not the snake has reached the target at that move, the axis & direction to get to that move and a variable to store the previous game state. In order to store multiple game boards at once, the API was modified so that the global variables CURR_FRAME, SCORE, MOOGLE_FLAG MOOGLES_EATEN are now instance variables of the GameBoard class.

The depth first search for *getfirstn (board, path_length)* is as shown in Figure 5. The DFS for *goforn* is similar to getfirstn with the additional requirement that the food must be in reachable distance; i.e, TIMEOUT-depth from nextgamestate < distance to food for all iterations.

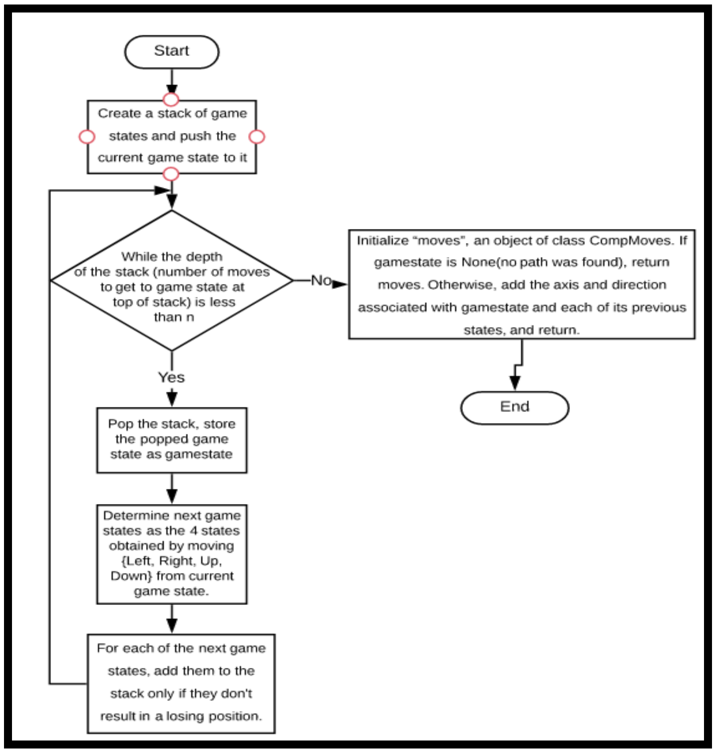

Start
Create a stack of game states and push the current game state to it
While the depth of the stack (number of moves to get to game state at top of stack) is less than n
No
Initialize "moves", an object of class CompMoves. If gamestate is None(no path was found), return moves. Otherwise, add the axis and direction associated with gamestate and each of its previous states, and return.
End
Yes
Pop the stack, store the popped game state as gamestate
Determine next game states as the 4 states obtained by moving {Left, Right, Up, Down} from current game state.
For each of the next game states, add them to the stack only if they don't result in a losing position.

Figure 5: Flowchart for getfirstn in Operate_Stack.py

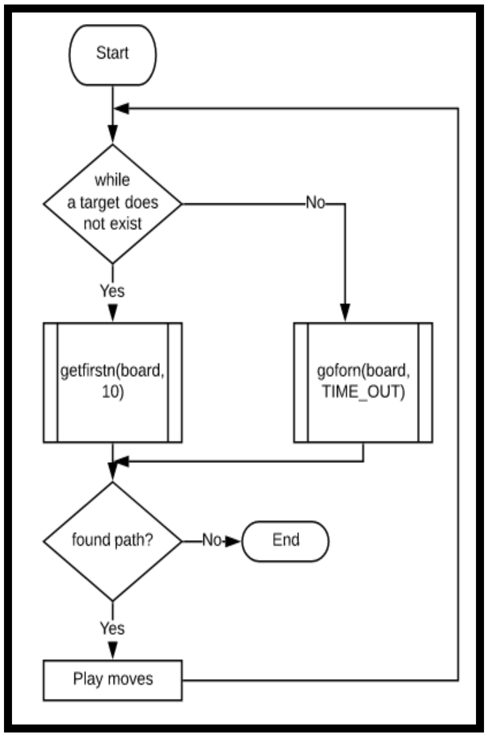


Figure 6: Flowchart for run_dfs in main.py

Finally, run_dfs() is used to run our algorithm on any values of BOARD_SIZE and CYCLE_ ALLOWANCE. It uses the two functions getfirstn and goforn as shown above(Figure 6).

Diagram Illustrating High-Level Overview of Solution:
The GameBoard, Snek and SnekBlock structs are left largely unchanged, with modifications to GameBoard to include local variables previously included as global variablesAppendix B). The GameState object is an augmentation to the original API, and is shown below.

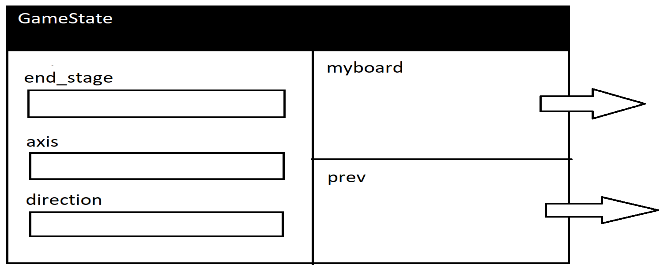


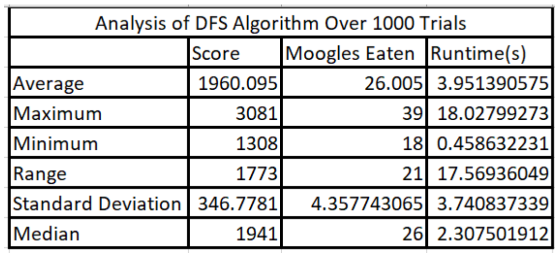
Figure 7: Illustration of GameState. Pointers both point to GameBoard Objects

Results:
[Readability]: All code(100%) has been commented and docstrings provided for Python functions and class definitions. Consistent indentation has been followed.
[Space Complexity]: The stack maintains a size less than or equal to 3n at all times, where n is the path length. This is due to our restriction on adding to the stack only if the snake is within reachable distance. The space complexity is O(n), which is easily manageable by the CPU.
[RunTime]:
This is determined by running 1000 trials(see below). However, we may examine the worst-case runtime of our algorithm when finding a path of length n. This will provide grounds for a modification to the code, as we describe shortly.
In the worst case, when no path of length n takes you to the target, we must explore every gametate. As there are n moves(n = TIMEOUT), and a maximum of 3 “nextgamestates” at each position, the runtime for searching all paths and finding it doesn't exist is O(3^n).
[Maximize Score]: Average score of 1960 with default board size. See results and discussion below.
[Flexibility with Different Board Sizes]: RPFA seems to work reasonably well for board sizes close to 10 and smaller. For board sizes greater than 10, a revised Hamiltonian or other algorithm is speculated to result in higher average scores. See results and discussion below.

Algorithm Performance with Default Board Size

We proceed now to present several graphs demonstrating the efficacy of the DFS algorithm across a default board size of 10. In particular, we analyze and comment about the algorithm's performance with respect to the score, number of targets consumed, and runtime over 1000 trials. Relevant statistics are also summarized in the chart below.

Table 1: Statistical analysis of Algorithm Performance



Analysis of DFS Algorithm Over 1000 Trials			
	Score	Moogles Eaten	Runtime(s)
Average	1960.095	26.005	3.951390575
Maximum	3081	39	18.02799273
Minimum	1308	18	0.458632231
Range	1773	21	17.56936049
Standard Deviation	346.7781	4.357743065	3.740837339
Median	1941	26	2.307501912

<u>[Maximize Score]:</u>

The average score achieved over 200 trials is ~1960. Through comparisons with other teams, this is quite good. The minimum score obtained was 1308. So, our algorithm is essentially guaranteed to achieve scores higher than 1000.

The algorithm is not too reliable however; the range is greater than the minimum score, so we can expect to see more than double the worst case. Furthermore, the standard deviation is over 300, so the data points are quite spread out(see Figure 8). This can be credited to the randomized nature of the DFS algorithm in selecting a path.

<u>[Runtime]:</u>

The average runtime is ~3.95 seconds, which is good. It allowed for fast computations of score, and even the maximum runtime was just 18 seconds. Note that by the nature of the DFS algorithm looking more than 50 moves ahead, the search space is very large(exponential, see Figure 12); to resolve the issue with respect to this metric, a counter variable has been introduced that breaks the search after reaching 50000.

Figure 8: Algorithm Score Performance

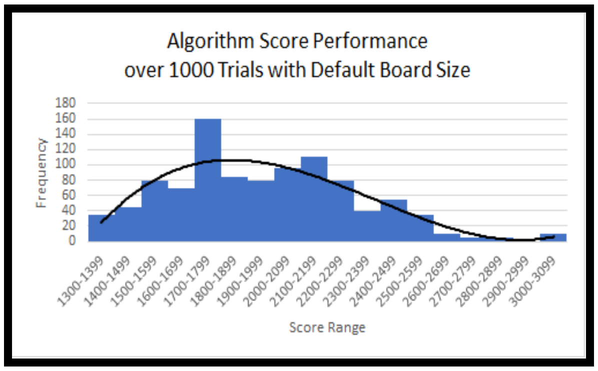
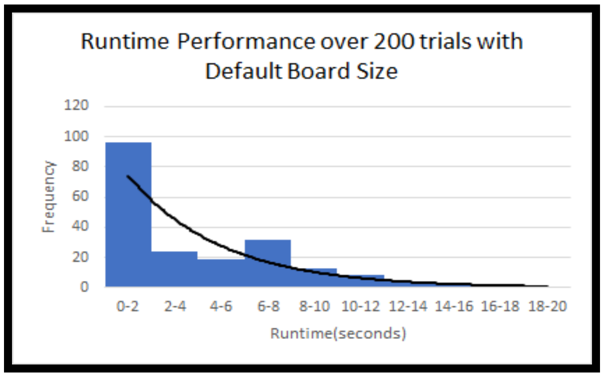


Figure 9: Algorithm Runtime Performance



Algorithm Performance with Varying Board Sizes

[Flexibility]: To assess RPFA's efficacy with varying board sizes, average score and target consumption of 200 trials were recorded and plotted for board sizes of range 5 to 15.

Figure 10: Board Size vs Average Score

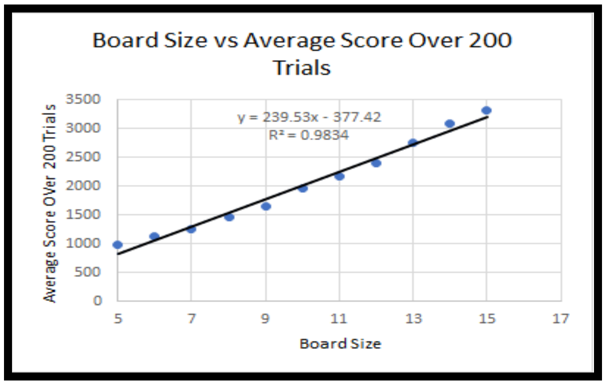
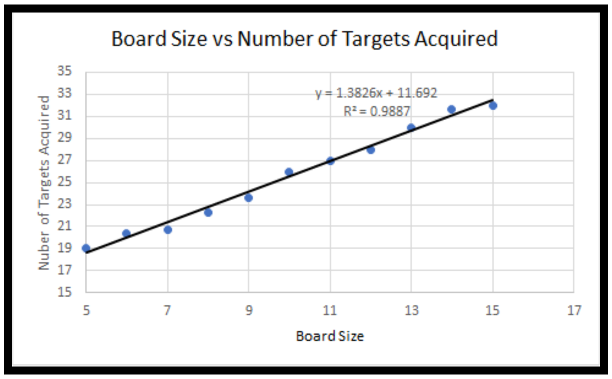


Figure 11: Board Size vs Number of Targets Acquired



Average scores and targets acquired generally increase with increasing board sizes. However, this relationship seems to be linear for both ($R^2 = 0.9834$, $R^2=0.9887$). In addition, average Moogles eaten increases slightly with board size(due to a "freer" board area, see Figure 13), but remains around 20-30 for all board sizes. So, we claim that higher scores are caused more so by higher TIMEOUT values(i.e; more moves per capture) than by actual increased board area. This serves as a major limitation and the cause again is an exponential search space that results in exponential time complexity. We have reason to believe our algorithm is not too flexible for large board sizes(>10) where time complexity becomes exponential.

Figure 12: Average Runtime vs Board Size

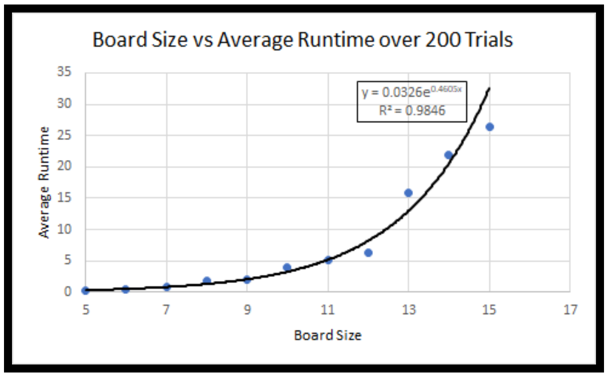
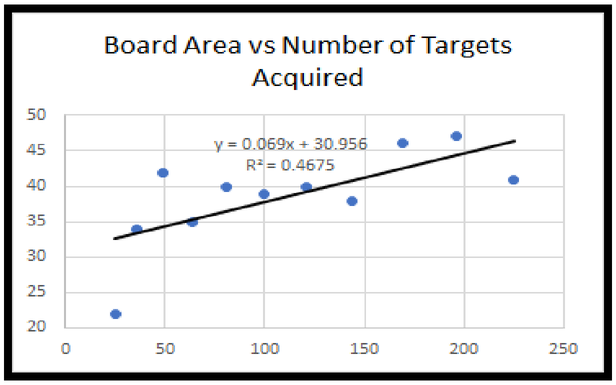


Figure 13: Board Area vs Number of Target Acquire



These results are also supported by Figure 12 and Figure 13. For larger board sizes, particularly when BOARD_SIZE $\geq$11, runtime grows rapidly. This is because higher board sizes result in higher TIME_OUT values, so we must search a longer path each time new food is spawn. When the snake is short; it has more space to move around. These simpler cases result in fewer stack operations. However, as the snake grows in length, it will eventually be large relative to the playing field(modelled as board area). These result in more stack operations as the likelihood of the snake colliding with itself or becoming trapped increases. From Figure 13, it is clear that a snake is considered long on a 5 by 5 board if over length 22, and on a 15*15 board if over length 41. Considering an increase in board area by a factor of 9 results in less than double the number of Moogles acquired indicates our algorithm is only ideal for small board sizes.

<u>Conclusion and Future Work:</u>

For board sizes less than or equal to 10, RPFA gives good score, and easily manageable runtime and space complexity. Experimentally, our algorithm gives unimpressive scores for board sizes greater than 10. We conclude that RPFA is efficient for smaller board sizes, but inefficient for larger ones.

To improve our algorithm for all board sizes, we address two major bottlenecks to our design:

1) For large board sizes, our restriction on the counter variable (see Appendix A, *goforn* in Operate_Stack.py) ensures a bound on runtime each time we search for TIME_OUT moves, but largely restricts score and moogles eaten, as it will conclude our search early.
2) The randomized nature of our depth-first search also restricts the maximum score we can get in any capture of a target. There currently exists no computer logic to optimize the order in which game states are pushed to the game stack. In particular, they are inserted in the order {down, up, right, left}, which results in a predictable side-to-side motion of the snake, moving up or down the board.

[Bottleneck 1]

To solve this, I would make the snake perform walks of 10, which it can compute easily. It is important to note that all walks must not reach a state where the snake acquires the food. After each walk of 10, I would use Dijkstra's Algorithm to confirm that a path exists between the snake and the target, and continue this process. If a path doesn't exist, I simply search for another one. When I am finally within a distance of 10 from the target, I would run the RPFA to find a path. This way, we do n/10 searches of depth 10, so time complexity is max{O((n/10)3^10), O(n^4)} = O(n^4), changing an exponential runtime to a power function.

———

[Bottleneck 2]

The idea above addresses this too. If we perform a randomized search only when we are 10 cells away from a target, then we are guaranteed a total life score of TIME_OUT-10 every time new food is spawn.

Appendix A: Code

Operate_Stack.py

```
# import necessary files
from Stack_Moves import *
from Game_State import *
from Comp_Moves import *
from time import sleep
from snek import *

# getfirstn accepts a board and an integer n, and returns a random sequence of valid moves
def getfirstn (board, n):
    '''
    getfirstn will be used to generate moves for the snake when no food is present. As there is no concern
about optimizing score in this stage of the game,
    the snake will obtain the first valid path of length n using a depth-first search algorithm
    '''
    count = 0
    stackofgamestates = StackMoves()
    gamestate = GameState(board)
    stackofgamestates.push(gamestate)
    while stackofgamestates.depth() <= n:
        gamestate = stackofgamestates.pop()
        if gamestate is None:
            break

        nextgamestate1 = gen_next_gamestate(1,1,gamestate, 0)
        nextgamestate2 = gen_next_gamestate(1,-1, gamestate, 0)
        nextgamestate3 = gen_next_gamestate(-1,1, gamestate, 0)
        nextgamestate4 = gen_next_gamestate(-1,-1, gamestate, 0)

        stackofgamestates.push(nextgamestate1)
        stackofgamestates.push(nextgamestate2)
        stackofgamestates.push(nextgamestate3)
```

```
            stackofgamestates.push(nextgamestate4)

            count+=1

    moves = Comp_moves()
    if gamestate is None:
        return moves

    tempstate = stackofgamestates.stack[-1]
    while tempstate != None and tempstate.prev != None:
        moves.insert_move(tempstate.axis, tempstate.direction)
        tempstate = tempstate.prev
    return moves

#goforn accepts a pointer to a board(converted form c) and an integer n, and returns a sequence of
computer moves of length n or less that lead to a MOOGLE
def goforn (board, n):
    '''
    This function finds the first available path to a moogle using a heuristic DFS algorithm.
    The stack stores all gamestates. In each iteration of the while loop, the most recently added state is
explored, and neighbouring gamestates are added.
    If all immediate moves lead to a failing state, that gamestate is popped from the stack and the next
gamestate at the top is explored.
    '''
    stackofgamestates = StackMoves()
    gamestate = GameState(board)
    stackofgamestates.push_all(gamestate)
    count = 0;

    while(True):
        if count > 25000:
            break;
        gamestate = stackofgamestates.pop()
        if gamestate is None:
            break
        if gamestate.end_stage == 1:
            sample = getfirstn (gamestate.myboard, 10)
            if sample.num_moves == 10:
                break
            else:
                continue

        nextgamestate1 = gen_next_gamestate(1,1,gamestate, 1)
        if ((nextgamestate1 is not None and distance(nextgamestate1, board)
<n-stackofgamestates.depthfrom(gamestate)) or (nextgamestate1 is not None and
nextgamestate1.end_stage == 1)):
            stackofgamestates.push_all(nextgamestate1)

        nextgamestate2 = gen_next_gamestate(1,-1, gamestate,1)
        if ((nextgamestate2 is not None and distance(nextgamestate2, board) <
(n-stackofgamestates.depthfrom(gamestate))) or (nextgamestate2 is not None and
nextgamestate2.end_stage == 1)):
            stackofgamestates.push_all(nextgamestate2)

        nextgamestate3 = gen_next_gamestate(-1,1, gamestate,1)
        if ((nextgamestate3 is not None and distance(nextgamestate3, board) <(
n-stackofgamestates.depthfrom(gamestate))) or (nextgamestate3 is not None and
nextgamestate3.end_stage == 1)):
            stackofgamestates.push_all(nextgamestate3)

        nextgamestate4 = gen_next_gamestate(-1,-1,gamestate,1)
        if ((nextgamestate4 is not None and distance(nextgamestate4, board)
<(n-stackofgamestates.depthfrom(gamestate))) or (nextgamestate4 is not None and
nextgamestate4.end_stage == 1)):
            stackofgamestates.push_all(nextgamestate4)
        count+=1

    moves = Comp_moves()
    while gamestate != None and gamestate.prev != None:
        moves.insert_move(gamestate.axis, gamestate.direction)
        gamestate = gamestate.prev
    return moves

# distance accepts a gamestate and a board pointer, and returns distance from snek head in "nextgamestate"
to moogle position as it appears in board
def distance(gamestate, board):
    '''
    This function is useful as it sets a condition for adding game states to our stack.
    We will only add game states to our stack if it is mathematically possible to reach the food from that
position, as determined by the minimum ditance        calculated below
    '''
    (moogle_x, moogle_y) = find_moogle(board)
    (curr_x, curr_y) = (gamestate.myboard[0].snek[0].head[0].coord[x],
gamestate.myboard[0].snek[0].head[0].coord[y])
    return abs(moogle_x-curr_x) + abs(moogle_y-curr_y)

# find_moogle accepts a board pointer and returns position of moogle
def find_moogle (board):
    '''
    find_moogle is used in distance() function to determine the moogle position before calculating distance
to it
    '''
    for i in range (0, BOARD_SIZE):
        for j in range (0, BOARD_SIZE):
            if board[0].cell_value[i][j] > 1:
                return (j, i)
    return (0,0)
```

Snek.py

```
'''
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.
'''
from ctypes import *
BOARD_SIZE = 10
CYCLE_ALLOWANCE = 1.5
TIME_OUT = ((BOARD_SIZE * 4) - 4) * CYCLE_ALLOWANCE

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

AXIS_X = -1
AXIS_Y = 1

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

AXIS_INIT = AXIS_X
DIR_INIT = RIGHT
```

```
# ---------------------------------

# 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), ('nextblock', POINTER(SnekBlock))]

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)), ('CURR_FRAME', c_int), ('SCORE', c_int),
('MOOGLE_FLAG', c_int), ('MOOGLES_EATEN', c_int)]

        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


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

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, [])
init_snek = wrap_func(snek_lib, 'init_snek', POINTER(Snek), [c_int, c_int])
hits_edge = wrap_func(snek_lib, 'hits_edge', c_int, [c_int, c_int, POINTER(GameBoard)])
hits_self = wrap_func(snek_lib, 'hits_self', c_int, [c_int, c_int, POINTER(GameBoard)])
is_failure_state = wrap_func(snek_lib, 'is_failure_state', c_int, [c_int, c_int, POINTER(GameBoard)])
time_out = wrap_func(snek_lib, 'time_out', None, [])
init_block = wrap_func(snek_lib, 'init_block', POINTER(SnekBlock), [c_int, c_int])
sleeep = wrap_func(snek_lib, 'sleeep', None, [])
#get_state = wrap_func(snek_lib, 'get_state', POINTER(GameState), [POINTER(GameBoard)])
#get_next_gamestate = wrap_func(snek_lib, 'get_next_gamestate', POINTER(GameState), [c_int, c_int,
POINTER(GameBoard), POINTER(GameState)])
```

Stack_Moves.py

```
'''
Stack Class

instance variables: stack
stack- implemented as a python list, to take advantage of append(), len() and pop() methods
'''
class StackMoves:
    #Constructor- initialize to empty list
    def __init__(self):
        self.stack = []

    # obtain size of stack using built-in len method
    def size(self):
        return len(self.stack)

    # return True if stack is empty, otherwise, return Flase
    def is_empty(self):
        if self.size() == 0:
            return True
        return False

    # push elemment to stack using python's built-in append method, add only if snake in gamestate is yet to
reach moogle
    def push(self, element):
        if element is not None and element.end_stage == -1:
            self.stack.append(element)

    # push element to stack using python's built-in append method, add element so long as it's not None
    def push_all (self, element):
        if element is not None:
            self.stack.append(element)
    # uses pyhton's built-in pop method to remove and return gamestate at the top of the stack
    def pop(self):
        if self.is_empty():
            return None
        return self.stack.pop()

    # return most recent element pushed to the stack, under the condition that it has not been visited before
    def top(self):
        '''
        This function is used when determining which gamestate to explore next, found in getfirstn() method
of operate_stack file
        If no unvisited element is found, returns None
        '''
        if self.is_empty():
            return None
        count = self.size()-1
        curr = self.stack[count]
        while count>0 and curr.visited:
            count-=1
            curr = self.stack[count]
        if curr.visited:
            return None
        return curr

    # return distance from starting gamestate to top of stack, where distance is number of moves required to
reach gamestate from the starting game board.
    def depth(self):
        if self.is_empty():
            return 0
        element = self.stack[-1]
        count = 1
        while element.prev is not None:
            count+=1
            element = element.prev
        return count

    # return depth from a passed-in gamestate to starting game board, where depth is the number of moves
that were required to reach the gamestate              # from the starting position
    def depthfrom(self, element):
        count = 0
        temp = element
        while temp.prev is not None:
            count+=1
            temp = temp.prev
        return count
```

Comp_Moves.py

```
'''
Comp_moves class

Class with instance variables num_moves and top.
num_moves- number of moves computer will play
top- move to be played first

This allows the API to accept a sequence of moves and stores it as a class
'''

class Comp_moves:
    # Constructor
    def __init__(self):
        self.num_moves = 0
        self.top = None

    # Add computer move to be executed last
    def push_move (axis, direction):
        move = Comp_move(axis, direction)
        if self.top is None:
            self.top = move
            self.num_moves +=1
            return
        curr = self.top
        while curr.next_move != NULL:
            curr = curr.next_move
        curr.next_move = move
        self.num_moves += 1

    # Add computer move to be executed first
    def insert_move(self, axis, direction):
        move = Comp_move(axis, direction)
        if self.top is None:
            self.top = move
            self.num_moves += 1
            return
        move.next_move = self.top
        self.top = move
        self.num_moves += 1
'''
Comp_move class

Class with instance variables axis, direction, and next_move
axis- axis in which snake must travel(x,y)
direction- direction of travel (Up, Down, Left, Right)
next_move- the next move to be executed by the snake

This class stores information regarding a single move. It will be used to give the snake instructions on
where to move next
'''
class Comp_move:
    def __init__(self, axis, direction):
        self.axis = axis
        self.direction = direction
        self.next_move = None
```

Game_State.py

```
# import snek file
from snek import *
import random
# define MACROS for game being at a finished or unfinished state
END = 1
NOT_DONE = -1

'''
GameState class

instance variables: myboard, end_stage, axis, direction, visited, prev
myboard- pointer to board containing all relevant board information
end_stage- 1 if game at finished position(successfully ate MOOGLE), -1 if not done(yet to eat MOOGLE)
axis- specify motion as either horizontal or vertical direction to reach gamestate
direction- 1 for down, right; -1 for up, left
visited- required in stack implementation, to determine if gamestate has been explored or not
prev- required in stack implementation, stores previous gamestate (to get to current state)

This class stores information on a potential game position. When initialized, it stores a copy of the playing
board, and in our stack implementation, possible gamestates will be stored as stack elements to determine a
path for the snake to the food
'''
class GameState:
  #Constructor
  def __init__(self, board):
    self.myboard = copy_board(board)
    self.end_stage = -1
    self.axis = AXIS_INIT
    self.direction = DIR_INIT
    #self.visited = False
    self.prev = None
```

```
# copy_board returns a pointer to a GameBoard struct(converted from c)
def copy_board(board):
    '''
    accepts a pointer to a board and returns a pointer to a copy of the board
    when initializing gamestate, we want to create a copy pointer to the board to use for computing future
moves
    '''
    tempboard = init_board()
    tempboard[0].cell_value = board[0].cell_value
    tempboard[0].occupancy = board[0].occupancy
    tempboard[0].snek = copy_snake(board[0].snek)
    tempboard[0].CURR_FRAME = board[0].CURR_FRAME
    tempboard[0].SCORE = board[0].SCORE
    tempboard[0].MOOGLE_FLAG = board[0].MOOGLE_FLAG
    tempboard[0].MOOGLES_EATEN = board[0].MOOGLES_EATEN
    return tempboard

#copy_snake returns a pointer to a snek struct(converted from c)
def copy_snake(snake):
    '''
    When copying gameboard, we need to copy snake pointer as a separate unit
    snake is not an immutable type, so we will copy snake body and return pointer to achieve effects of
deepcopy
    '''
    copy = init_snek(snake[0].head[0].coord[x], snake[0].head[0].coord[y])
    snekblock = snake[0].head[0].nextblock
    while(copy[0].length < snake[0].length):
        if copy[0].length == 1:
            copy[0].tail[0].coord[x] = snekblock[0].coord[x]
            copy[0].tail[0].coord[y] = snekblock[0].coord[y]
        else:
            copy[0].tail[0].nextblock = init_block(snekblock[0].coord[x], snekblock[0].coord[y])
            copy[0].tail = copy[0].tail[0].nextblock
        copy[0].length+=1
        snekblock = snekblock[0].nextblock
    return copy

# gen_next_gamestate accepts a gamestate, an axis and a direction, and returns a new game state that is the result of the snake in the
# curent gamestate moving in the specified axis and direction
def gen_next_gamestate (axis, direction, curr_state, moogle_val):
    '''
    This function is used to determine what to add into our game stack in each iteration of the while loop found in operate_stack()
    We will use this method to determine whether moving to the right, left, up, or down is legal and include or ignore them accordingly
    when updating our stack
    '''
    if (is_failure_state(axis, direction, curr_state.myboard)):
        return None
    #random.seed()
    next_state = GameState(curr_state.myboard)
    next_state.axis = axis
    next_state.direction = direction
    next_state.prev = curr_state

    advance_frame(axis, direction, next_state.myboard)
    if moogle_val == 0 and next_state.myboard[0].MOOGLE_FLAG == 1:
        next_state.myboard[0].MOOGLE_FLAG = 0
        for i in range (BOARD_SIZE):
                for j in range (BOARD_SIZE):
                        next_state.myboard[0].cell_value[i][j] = 0

    if next_state.myboard[0].SCORE - curr_state.myboard[0].SCORE > 1:
        next_state.end_stage = 1
    return next_state
```

Main.py

```
'''
Hshmat Sahak
April 9, 2020
ESC190 Project Submission
Snek Game
'''

#import necessary files
from snek import *
import time
from Comp_Moves import *
from Game_State import *
from Operate_Stack import *
from Stack_Moves import *
import random

# play_out accepts an object of class comp_moves, a board pointer and game condition (eat food or not),
and commands the snake to move accordingly
def play_out (moves, board, moogle_val):
	'''
	this method allows the program to control the snake by continuously advancing the gameboard
frame according to the parameter "moves", and displaying 	the new position to screen
	'''
	#random.seed()
	move = moves.top
	while (move is not None and board[0].MOOGLE_FLAG == moogle_val):
		advance_frame(move.axis, move.direction, board)
		show_board(board)
		move = move.next_move

#*****************************************ALGORITHM***************************
******************
```

```
def run_dfs():
    '''
    Main function.
    '''
    start_time = time.time()
    #initialize board
    board = init_board()
    # cycle through loop as long as game is not finished
    while True:
        # while no food present
        while board[0].MOOGLE_FLAG != 1:
            #obtain any sequence of 50 moves and play it out
        #sleep(1)
            the_moves = getfirstn (board, 25)
            if the_moves.num_moves == 0:
                show_board(board)
                score = board[0].SCORE
                moogles_eaten = board[0].MOOGLES_EATEN
                end_game(board)
                #print("heooo")
                #sleep(3)
                return (BOARD_SIZE, score, moogles_eaten, time.time() - start_time)
            play_out(the_moves, board, 0)
        #sleep(1)
        if board[0].MOOGLE_FLAG == 1: #once food appears
            #find (any) path to food
            #print("calculating")
            #sleep(2)
            the_moves = goforn (board, TIME_OUT)
            if the_moves.num_moves == 0: #if path not found, end game
                show_board(board)
                score = board[0].SCORE
                moogles_eaten = board[0].MOOGLES_EATEN
                end_game(board)
                #sleep(0.5)
                break # exit out of while loop
            play_out (the_moves, board, 1) # travel to food
    end_time = time.time()
    return (BOARD_SIZE, score, moogles_eaten, end_time - start_time)

#************************************************MAIN**************************
*****************
if __name__ == "__main__":
    sleeep()
    trials = int(input("Enter number of trials: "))
    file_name = 'boardsize'
    board = int(input("Enter board size: "))
    file_name = file_name + str(board) + "_output.csv"
    with open (file_name, "a") as f:
        for i in range (trials):
            (boardsize, score, moogles, runtime) = run_dfs()
            #sleep(1)
            f.write(str(boardsize) + "," + str(score) + "," + str(moogles) + ", " + str(runtime) + "\n")
```

snek_api.c

```
#include <string.h>
#include <time.h>
#include "snek_api.h"
#include <unistd.h>

int TIME_OUT = ((BOARD_SIZE * 4) - 4) * CYCLE_ALLOWANCE;

GameBoard* init_board(){
        GameBoard* gameBoard = (GameBoard*)(malloc(sizeof(GameBoard)));
        gameBoard-> CURR_FRAME = 0;
        gameBoard->SCORE = 0;
        gameBoard->MOOGLE_FLAG = 0;
        gameBoard-> MOOGLES_EATEN=0;
        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;
        gameBoard->snek = init_snek(0, 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->nextblock = NULL;
        snek->head->nextblock = snek->tail;

        snek->length = 1;

        return snek;
}

SnekBlock* init_block(int a, int b){
        SnekBlock* block = malloc(sizeof(SnekBlock));
        block->coord[x] = a;
        block->coord[y] = b;
        block->nextblock = NULL;
        return block;
}

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;
        }
        return gameBoard->occupancy[new_y][new_x];
}

int time_out(GameBoard *gameBoard){
        return (gameBoard->MOOGLE_FLAG == 1 && gameBoard->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(gameBoard));
}

void populate_moogles(GameBoard *gameBoard){
        if (gameBoard->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;
                        gameBoard->MOOGLE_FLAG = 1;
                } else if (r3 < BOARD_SIZE){
                        gameBoard->cell_value[r1][r2] = MOOGLE_POINT;
                        gameBoard->MOOGLE_FLAG = 1;
                }
        }
}

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

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

int advance_frame(int axis, int direction, GameBoard *gameBoard){
        if (is_failure_state(axis, direction, gameBoard)){
                return 0;
        } else {
                int head_x, head_y;
                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;
                }
                int tail_x = gameBoard->snek->tail->coord[x];
                int tail_y = gameBoard->snek->tail->coord[y];

                gameBoard->occupancy[head_y][head_x] = 1;
                if (gameBoard->snek->length > 1) {
                        SnekBlock *newBlock = (SnekBlock
*)malloc(sizeof(SnekBlock));
                        newBlock->coord[x] =
gameBoard->snek->head->coord[x];
                        newBlock->coord[y] =
gameBoard->snek->head->coord[y];
                        newBlock->nextblock =
gameBoard->snek->head->nextblock;

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

                        if (gameBoard->cell_value[head_y][head_x] > 0){
                                eat_moogle(gameBoard, head_x, head_y);
                        } else {
                                gameBoard->occupancy[tail_y][tail_x] = 0;
                                SnekBlock *currBlock =
gameBoard->snek->head;
                                while (currBlock->nextblock !=
gameBoard->snek->tail){
                                        currBlock = currBlock->nextblock;
                                }

                                currBlock->nextblock = 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 {
                        eat_moogle(gameBoard, head_x, head_y);
                        gameBoard->snek->head->coord[x] = head_x;
                        gameBoard->snek->head->coord[y] = head_y;
                }
                gameBoard->SCORE = gameBoard->SCORE + LIFE_SCORE;
                if (gameBoard->MOOGLE_FLAG == 1){
                        gameBoard->CURR_FRAME ++;
                }

                populate_moogles(gameBoard);
                return 1;
        }
}

void show_board(GameBoard* gameBoard) {
        fprintf(stdout, "\033[2J");
        fprintf(stdout, "\033[0;0H");

        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){
                                fprintf(stdout, "%c", snek);
                        } else if (gameBoard->cell_value[i][j] > 0) {
                                fprintf(stdout, "%c", moogle);
                        } else {
                                fprintf(stdout, "%c", blank);
                        }
                }
                fprintf(stdout, "\n");

        }

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

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

```c
        fflush(stdout);
}

int get_score(GameBoard *gameBoard) {
        return gameBoard->SCORE;
}

void end_game(GameBoard **board){
        fprintf(stdout, "\n\n\n--!!---GAME OVER---!!--\n\nYour score:
%d\n\n\n\n", (*board)->SCORE);
        fflush(stdout);
        SnekBlock **snekHead = &((*board)->snek->head);
        SnekBlock *curr;
        SnekBlock *prev;
        while ((*snekHead)->nextblock != NULL) {
                curr = *snekHead;
                while (curr->nextblock != NULL){
                        prev = curr;
                        curr = curr->nextblock;
                }
                prev->nextblock = NULL;
                free(curr);
        }
        free(*snekHead);
        free((*board)->snek);
        free(*board);
}

void sleeep(){
        srand(time(NULL));
}
```

Snek_api.h

```
#include <string.h>
#include <time.h>
#include "snek_api.h"
#include <unistd.h>

int TIME_OUT = ((BOARD_SIZE * 4) - 4) * CYCLE_ALLOWANCE;

GameBoard* init_board(){
	GameBoard* gameBoard = (GameBoard*)(malloc(sizeof(GameBoard)));
	gameBoard-> CURR_FRAME = 0;
	gameBoard->SCORE = 0;
	gameBoard->MOOGLE_FLAG = 0;
	gameBoard-> MOOGLES_EATEN=0;
	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;
	gameBoard->snek = init_snek(0, 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->nextblock = NULL;
```

```
        snek->head->nextblock = snek->tail;

        snek->length = 1;

        return snek;
}

SnekBlock* init_block(int a, int b){
        SnekBlock* block = malloc(sizeof(SnekBlock));
        block->coord[x] = a;
        block->coord[y] = b;
        block->nextblock = NULL;
        return block;
}

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;
        }
        return gameBoard->occupancy[new_y][new_x];
}

int time_out(GameBoard *gameBoard){
        return (gameBoard->MOOGLE_FLAG == 1 && gameBoard->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(gameBoard));
}

void populate_moogles(GameBoard *gameBoard){
        if (gameBoard->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;
                        gameBoard->MOOGLE_FLAG = 1;
                } else if (r3 < BOARD_SIZE){
                        gameBoard->cell_value[r1][r2] = MOOGLE_POINT;
                        gameBoard->MOOGLE_FLAG = 1;
                }
        }
}

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

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

int advance_frame(int axis, int direction, GameBoard *gameBoard){
        if (is_failure_state(axis, direction, gameBoard)){
                return 0;
        } else {
                int head_x, head_y;
                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;
                }
                int tail_x = gameBoard->snek->tail->coord[x];
                int tail_y = gameBoard->snek->tail->coord[y];

                gameBoard->occupancy[head_y][head_x] = 1;
                if (gameBoard->snek->length > 1) {
                        SnekBlock *newBlock = (SnekBlock
*)malloc(sizeof(SnekBlock));
                        newBlock->coord[x] =
gameBoard->snek->head->coord[x];
                        newBlock->coord[y] =
gameBoard->snek->head->coord[y];
                        newBlock->nextblock =
gameBoard->snek->head->nextblock;

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

                        if (gameBoard->cell_value[head_y][head_x] > 0){
                                eat_moogle(gameBoard, head_x, head_y);
                        } else {
                                gameBoard->occupancy[tail_y][tail_x] = 0;
                                SnekBlock *currBlock =
gameBoard->snek->head;
                                while (currBlock->nextblock !=
gameBoard->snek->tail){
                                        currBlock = currBlock->nextblock;
                                }

                                currBlock->nextblock = 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 {
                        eat_moogle(gameBoard, head_x, head_y);
                        gameBoard->snek->head->coord[x] = head_x;
                        gameBoard->snek->head->coord[y] = head_y;
                }
                gameBoard->SCORE = gameBoard->SCORE + LIFE_SCORE;
                if (gameBoard->MOOGLE_FLAG == 1){
                        gameBoard->CURR_FRAME ++;
                }

                populate_moogles(gameBoard);
                return 1;
        }
}

void show_board(GameBoard* gameBoard) {
        fprintf(stdout, "\033[2J");
        fprintf(stdout, "\033[0;0H");

        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){
                                fprintf(stdout, "%c", snek);
                        } else if (gameBoard->cell_value[i][j] > 0) {
                                fprintf(stdout, "%c", moogle);
                        } else {
                                fprintf(stdout, "%c", blank);
                        }
                }
                fprintf(stdout, "\n");

        }

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

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

```
        fflush(stdout);
}

int get_score(GameBoard *gameBoard) {
        return gameBoard->SCORE;
}

void end_game(GameBoard **board){
        fprintf(stdout, "\n\n\n--!!---GAME OVER---!!--\n\nYour score:
%d\n\n\n\n", (*board)->SCORE);
        fflush(stdout);
        SnekBlock **snekHead = &((*board)->snek->head);
        SnekBlock *curr;
        SnekBlock *prev;
        while ((*snekHead)->nextblock != NULL) {
                curr = *snekHead;
                while (curr->nextblock != NULL){
                        prev = curr;
                        curr = curr->nextblock;
                }
                prev->nextblock = NULL;
                free(curr);
        }
        free(*snekHead);
        free((*board)->snek);
        free(*board);
}

void sleeep(){
        srand(time(NULL));
}
```

snek_api.h

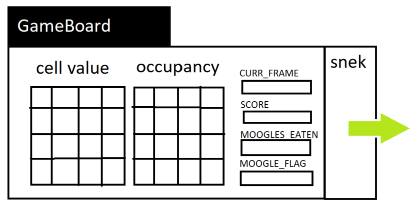
```
/**
AUTHOR: SAIMA ALI, EDIT AND REVISED BY: HSHMAT SAHAK
LATEST WORKING VERSION
FEBRUARY 2ND, 2020
ESC190H1S PROJECT
SNAKE API
 **/

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

#define CYCLE_ALLOWANCE 1.5
#define BOARD_SIZE 10

#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* nextblock;
} SnekBlock;

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;
    int CURR_FRAME;
    int SCORE;
    int MOOGLE_FLAG;
    int MOOGLES_EATEN;
}GameBoard;

GameBoard *init_board();
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);
```

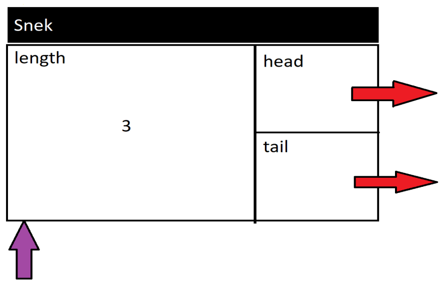
```
void end_game(GameBoard **board);
void show_board(GameBoard* gameBoard);
int get_score();
int time_out();
```

Appendix B: Illustrations for High-Level Overview of Solution (extracted form lab doc)

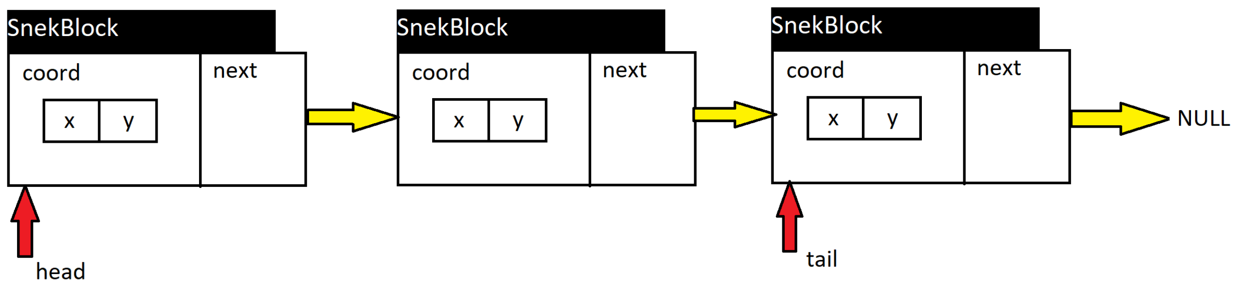
GameBoard(Modified with Local Variables)



Snek(left unchanged)



SnekBlock(left unchanged)


SnekBlock
coord
x
y
next
SnekBlock
coord
x
y
next
SnekBlock
coord
x
y
next
NULL
head
tail