Snek: Report

Eeman Salman

Table of Contents

1.0 - Introduction

1.1 - A Little Background

You, Voldy, are one of the most notorious sorcerers in the Sorcerer's Spot. Those who know you know that you despise non-magical folk, known as moogles. One day, being quarantined into boredom, you and your pet snake, Magini decide to play a little game with the moogles. A visual representation of this game has been made for your henchmen to enjoy during isolation in the comfort of their homes.

1.2 - Understanding the Game

The project that we programmed describes a snake (Magini) that moves around the screen and has the goal of approaching a moogle. Once she interacts with the moogle, her score increases by 1, and Magini grows larger by one cell size. The game comes to an end when Magini either covers the entire board, bumps into a wall or bumps into herself. In order to effectively demonstrate this game, an algorithm has been implemented to automate the snake's motion and maximize the number of points that Magini can acquire before the game ends.

1.3 - Background Research

After researching possible ways to optimize the snake's movements, we came across the Hamiltonian technique [6.1.6]. This technique allows for all the nodes to be visited once, and allows the snake to reach all the moogles without hitting itself. Our team decided not to implement this technique as it only works for a specific ratio between cycle allowance and board size and we aimed to create a more universal algorithm. Furthermore, it has a runtime complexity of O(n!) which does not coincide with one of the objectives highlighted in this report [2.2].

1.4 - Report Overview

This report will evaluate the implementation of the game through its objectives and criteria. It will also delve into the justification for the chosen programming language and data structures. Finally, it will talk about future work that can improve the program as well as get rid of any potential bottlenecks.

2.0 - Objectives

2.1 - Different Board Sizes

The algorithm to be implemented should be able to maximize the score with varying board sizes.

Metric: A lower bound and upper bound for the board size will be found and the range of size will be calculated by the following formula:

$$Range = Upper\ Bound - Lower\ Bound$$

Constraint: The largest board size to be tested is 20 due to the time constraint of the project.

Criteria: A higher score on the scale seen in Table 1 will be considered a better result.

Table 1: Score of the Range of Board Sizes

Score	1	2	3	4	4
Range	1	2-5	6-10	11-15	16-19

2.2 - Time Complexity

The data structure implemented should be designed to work at a minimum time complexity. The following operations must be considered when designing the game: changing the direction of the snake as well as ensuring that the snake is prevented from hitting itself or an edge.

Metric: This will be measured using Big-O Notation [6.1.1] (see Table 2).

Criteria: A lower run-time on the scale is a better result.

Table 2: Big-O Notation for Typical Runtime Complexities

1	2	3	4	5	6	7
O(1)	O(logn)	O(n)	O(nlogn)	$O(n^2)$	$O(2^n)$	O(n!)

2.3 - Maximizing Points

The game should be designed by an algorithm that maximizes the possible number of points that Magini can get before the game ends. This means considering the targets as well as ensuring Magini does not bump into a wall or herself. This also means the maximum number of points that can be acquired is when the size of Magini is equivalent to the size of the board.

Metric: This will be measured by the average final score of the game after 100 trials have been run and recorded.

Constraint: This will only be measured for a board size of 10

Criteria: A higher average score is considered a better result.

Metric: This will be measured by the average final length of Magini at the end of the game after 100 trials have been run and recorded.

Constraint: This will only be measured for a board size of 10

Criteria: A higher average length is considered a better result.

3.0 - Detailed Framework

3.1 - C Programming Language

The project was made in the C programming language as the program offers various mechanisms and different coding preferences that we as a team make it beneficial to use. C also has a rich library, with numerous useful built-in functions as well as dynamic memory allocation [6.1.2]. Furthermore, as a team, we have a better understanding of C than Python, making it more convenient for us as well.

3.2 - Implemented Data Structure

The snake game was implemented using linked lists. Linked lists are a type of data structure that contains nodes that store information within the node as well as a reference to the next node in the list [6.1.4]. Moreover, the linked list will be used to implement a queue. A linked list is a useful data structure to use for this game as it can store the series of moves that the snake must do in order to eat the moogles as well as stay alive (see Figure 1). Furthermore, using queues allows for the program to traverse through these moves in FIFO (first-in, first-out) basis so that the queue only holds the moves that have not yet been carried out (see Figure 2). The implementation of this algorithm in C was through the use of structs and pointers which were used for the initialization, insertion, and deletion of elements in the linked list, resulting in the time complexity of O(1) [6.1.5].

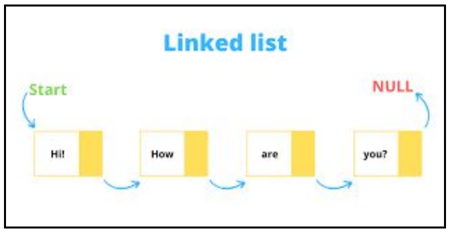


Figure 1: Visual Representation of Linked List

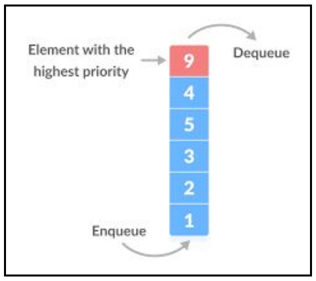


Figure 2: Visual Representation of Queues

3.3 - The Algorithm

The algorithm works by taking a linked-list approach and queuing the appropriate instructions to have the snake approach the target in the lowest amount of moves possible. Depending on the instructions queued and the axis at which the snake is moving, a dequeue of the first instruction is performed, following a dequeue of the second instruction if the snake is in the same x or y-axis as the target. When there is no target on the board the snake moves according to the preset moves it is given without the algorithm. A visual representation of this linked list implementation can be seen in Figure 3.

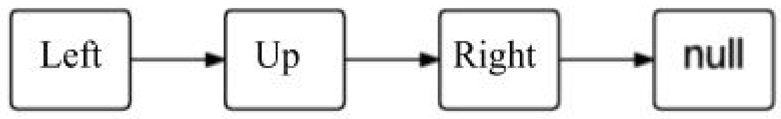


Figure 3 - Linked List Application of Algorithm Visual

There are 2 main conditions that the snake can come across when making its journey to the target. The snake may run into itself, or the target may be behind the snake. If one of these conditions are met, the algorithm calls another function to deduce which direction is the best choice for the snake and queues the appropriate instructions whilst deleting the instructions to get to the target. These instructions are continuously updated until the condition is not met and the target can now be approached safely for the snake.

Depending on the size of the list and the conditions mentioned above, the dequeue of the snake follows certain if statements that make it dequeue appropriately. Once the instructions have been dequeued, the snake returns to an idle state that prevents it from running into itself and the wall until another target is spawned on the board.

4.0 - Results

This section evaluates the game with respect to the operations outlined in the project handout [6.1.3] as well as the objectives defined in the objectives section of the report [2.0].

4.1 - Operations

One of the operations outlined in the project handout was having the program accept a sequence of moves and execute them in the game. As seen in our code in Appendix B [6.2.4], the function *insert_direction* can accept a move based on direction and axis and add it into a linked list which could then be executed if called in the *main.c* file.

The second operation mentioned in the handout was ensuring that the program runs Valgrind clean if this project was implemented in C. This execution can be found in Appendix B [6.2.2].

The third operation outlined maximizing the points for the default board size which was 10. In order for this to be maximized, the snake must capture as many moogles as it can before it hits itself, hits the edge, or is as large as the size of the board. As seen in the video linked in Appendix B [6.2.1], the snake was able to increase in length until there was no way to get to the target due to the size of the snake and its location on the grid.

The fourth operation expected for the algorithm to be run at least 100 times to show its efficacy. The scores from these trials were stored in a text file [6.2.3.1]. Based on these scores, the expected score was 1012 (taken from averaging all scores from the 100 trials). The minimum score the algorithm is guaranteed to accomplish is 373 based on these trials.

For the fifth operation, the algorithm created was tested with varying board sizes. As highlighted in the objectives as well, the range of board sizes determined the success of the algorithm implementation. The board size was changed in the *snek_api.h* file to test the varying sizes [6.2.4.1]. This implementation can be seen in the video demonstration highlighted in Appendix B [6.2.1].

4.2 - Objectives

Similar to the fifth operation, our first objective explored the varying board sizes for our algorithm. Our algorithm worked for a range of 19, giving it a score of 5 on the scale highlighted below. This score was the best achievable score on the scale, which means the algorithm exceeded expectations in terms of this objective.

Score	1	2	3	4	5
Range	1	2-5	6-10	11-15	16-19

In terms of time complexity, our code got a score of 1 on the scale shown below. This result was due to our implementation of linked lists as discussed in our detailed framework. This score was the best achievable score on the scale, which means the algorithm exceeded expectations in terms of this objective.

1	2	3	4	5	6	7
O(1)	O(logn)	O(n)	O(nlogn)	O(n^2)	O(2^n)	O(n!)

Our average score after running 100 trials was 1012, with our lowest score being 373 and our highest being 1766 [6.2.3.1]. This average score met the expectations of the project but could be improved if further work was done on the algorithm. The final length of the snake was also stored in a separate text file [6.2.3.2]. The average length of the snake was 24 with the minimum being 10 and the maximum is 35. This average length met the expectations of the project, spanning 35% of the board, but could be improved if further work was done on the algorithm.

5.0 - Conclusion

5.1 - Limitations

Overall, the performance of our game met all our objectives, but there was still space for improvement. If future work was to be done on the project, the interface for this project should be improved, so that it is easier for the henchmen to view. The range of board sizes should also be further tested for boards greater than 20. Currently, the snake dies when it takes a direction in which, after a couple more moves, it finds itself trapped within its body. For future work, the snake should be programmed to anticipate the best move to take for the snake so that it does not get stuck inside itself and dies. An addition to the algorithm can be implemented in which the snake can check to see if it will get blocked when it goes down a certain path and will try to find a different path if there is any blockage.

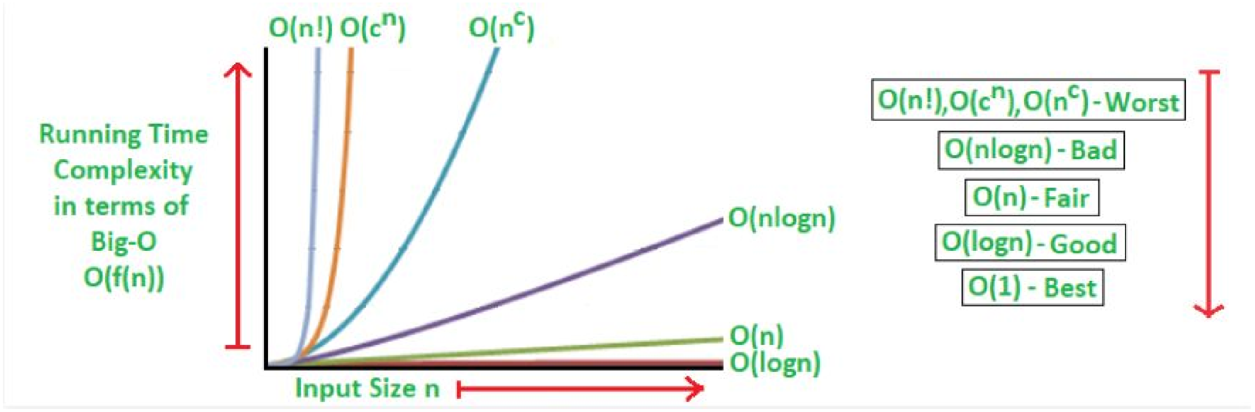
5.2 - Final Thoughts

After evaluating the performance of the game through the operations highlighted in the handout [6.1.3] and the objectives stated above, the algorithm implemented into the game was overall successful. The snake was able to successfully maneuver its way around the board, finding a path to the moogle while moving around its body and ensuring it does not hit the edge. It was also able to reset itself around the edge of the board when no moogle was seen in the vicinity. This algorithm overall exceeded expectations in all the objectives highlighted in the objectives section of the report.

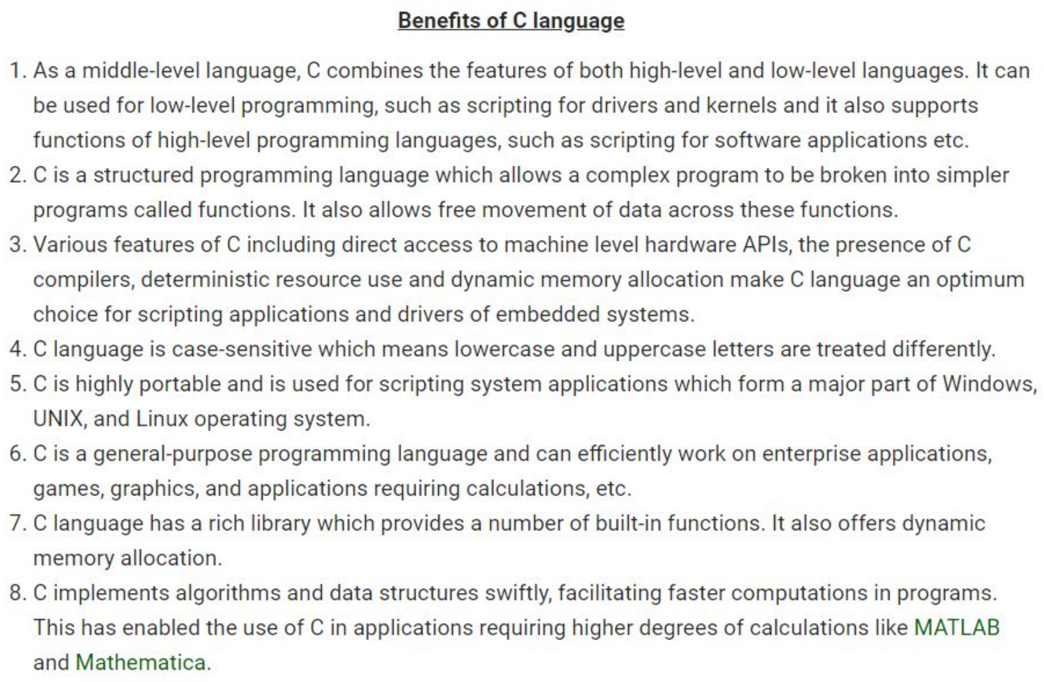
6.0 - Appendices

6.1 - Appendix A: Source Extracts

6.1.1. Soumyadeep Debnath, (2018, June 8). Analysis of Algorithms: Big-O analysis. Retrieved from https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/



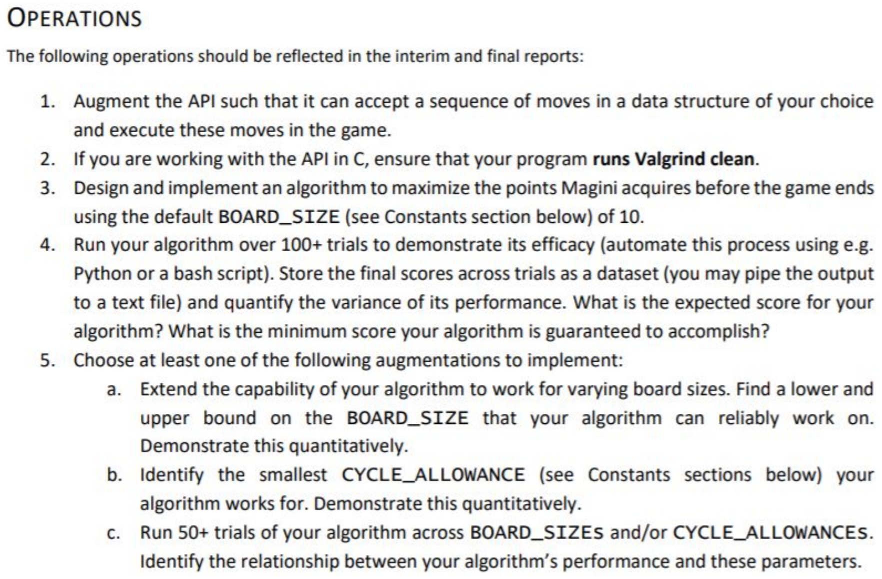
6.1.2. Benefits of C language over other programming languages. (2019, April 1). Retrieved from https://www.geeksforgeeks.org/benefits-c-language-programming-languages/



Benefits of C language

1. As a middle-level language, C combines the features of both high-level and low-level languages. It can be used for low-level programming, such as scripting for drivers and kernels and it also supports functions of high-level programming languages, such as scripting for software applications etc.
2. C is a structured programming language which allows a complex program to be broken into simpler programs called functions. It also allows free movement of data across these functions.
3. Various features of C including direct access to machine level hardware APIs, the presence of C compilers, deterministic resource use and dynamic memory allocation make C language an optimum choice for scripting applications and drivers of embedded systems.
4. C language is case-sensitive which means lowercase and uppercase letters are treated differently.
5. C is highly portable and is used for scripting system applications which form a major part of Windows, UNIX, and Linux operating system.
6. C is a general-purpose programming language and can efficiently work on enterprise applications, games, graphics, and applications requiring calculations, etc.
7. C language has a rich library which provides a number of built-in functions. It also offers dynamic memory allocation.
8. C implements algorithms and data structures swiftly, facilitating faster computations in programs. This has enabled the use of C in applications requiring higher degrees of calculations like MATLAB and Mathematica.

6.1.3. As highlighted in the project handout

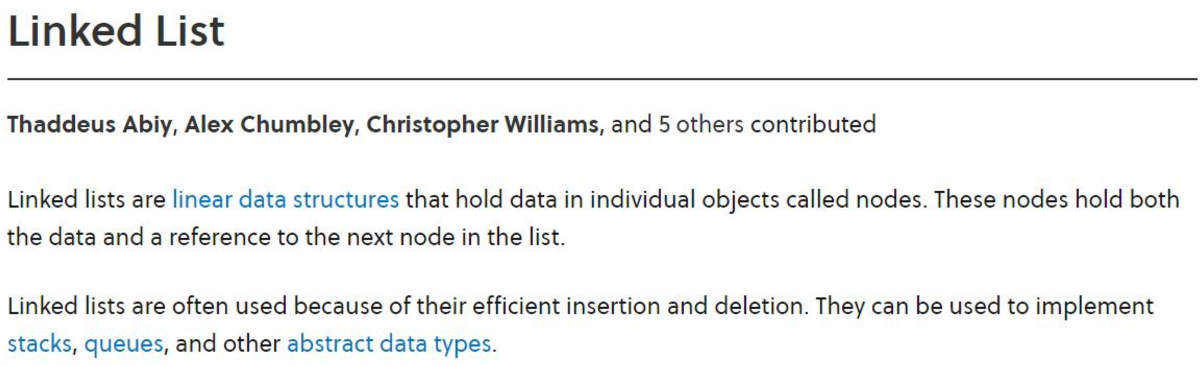


OPERATIONS

The following operations should be reflected in the interim and final reports:

1. Augment the API such that it can accept a sequence of moves in a data structure of your choice and execute these moves in the game.
2. If you are working with the API in C, ensure that your program **runs Valgrind clean**.
3. Design and implement an algorithm to maximize the points Magini acquires before the game ends using the default BOARD_SIZE (see Constants section below) of 10.
4. Run your algorithm over 100+ trials to demonstrate its efficacy (automate this process using e.g. Python or a bash script). Store the final scores across trials as a dataset (you may pipe the output to a text file) and quantify the variance of its performance. What is the expected score for your algorithm? What is the minimum score your algorithm is guaranteed to accomplish?
5. Choose at least one of the following augmentations to implement:
   a. Extend the capability of your algorithm to work for varying board sizes. Find a lower and upper bound on the BOARD_SIZE that your algorithm can reliably work on. Demonstrate this quantitatively.
   b. Identify the smallest CYCLE_ALLOWANCE (see Constants sections below) your algorithm works for. Demonstrate this quantitatively.
   c. Run 50+ trials of your algorithm across BOARD_SIZEs and/or CYCLE_ALLOWANCEs. Identify the relationship between your algorithm's performance and these parameters.

6.1.4. Linked List. (n.d.). Retrieved from https://brilliant.org/wiki/linked-lists/



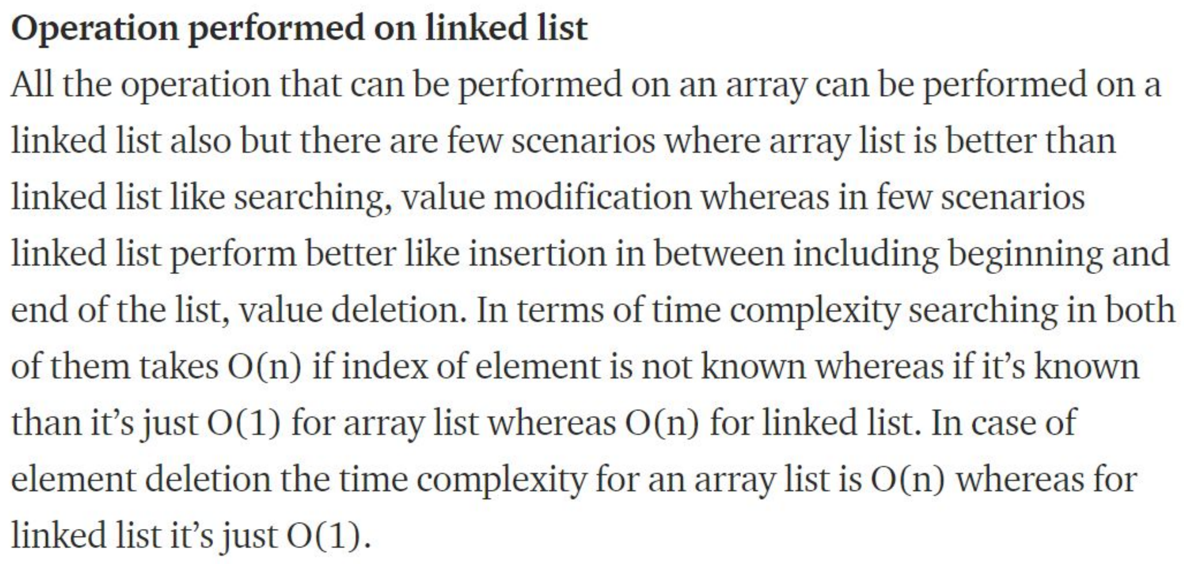
Linked List

Thaddeus Abiy, **Alex Chumbley**, **Christopher Williams**, and 5 others contributed

Linked lists are linear data structures that hold data in individual objects called nodes. These nodes hold both the data and a reference to the next node in the list.

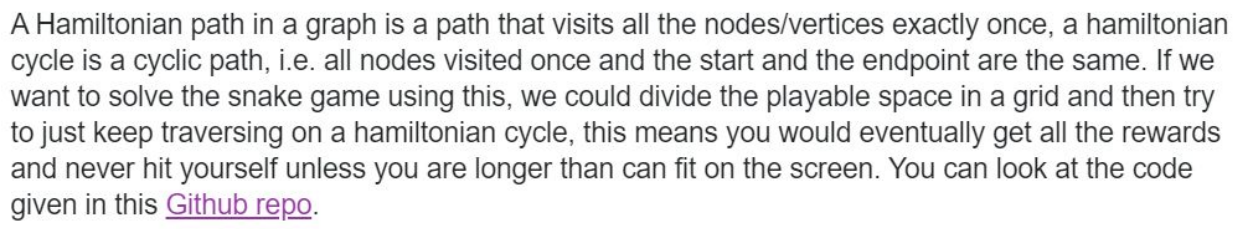
Linked lists are often used because of their efficient insertion and deletion. They can be used to implement stacks, queues, and other abstract data types.

6.1.5. Rai, P. (2017, May 27). Know your data structure - Linked List. Retrieved from https://medium.com/@pankaj.rai16/know-your-data-structure-linked-list-4b00fcfbda93


Operation performed on linked list

All the operation that can be performed on an array can be performed on a linked list also but there are few scenarios where array list is better than linked list like searching, value modification whereas in few scenarios linked list perform better like insertion in between including beginning and end of the list, value deletion. In terms of time complexity searching in both of them takes O(n) if index of element is not known whereas if it's known than it's just O(1) for array list whereas O(n) for linked list. In case of element deletion the time complexity for an array list is O(n) whereas for linked list it's just O(1).

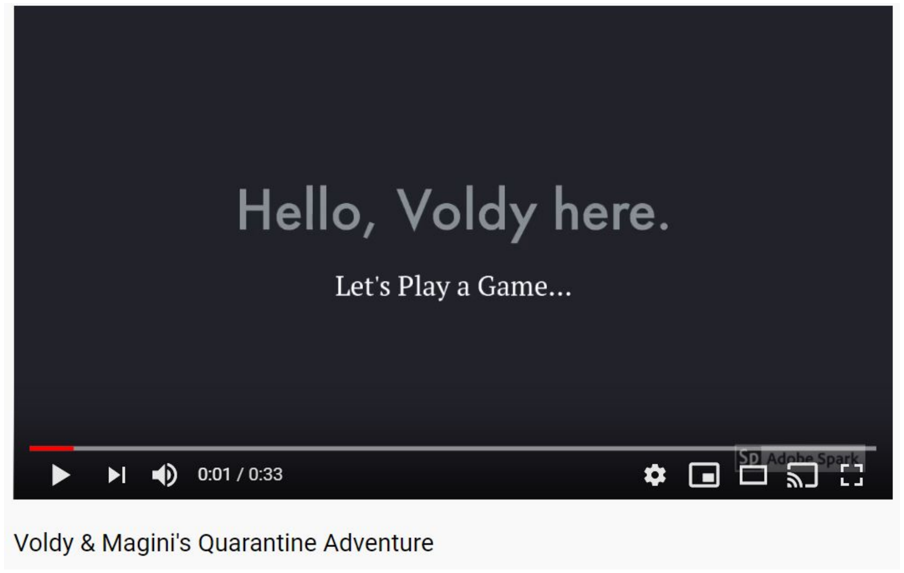
6.1.6. Kandil, A. K. A. (1969, September 1). How to solve Snake Game with a Hamiltonian graph algorithm? Retrieved from https://ai.stackexchange.com/questions/16985/how-to-solve-snake-game-with-a-hamiltonian-graph-algorithm


A Hamiltonian path in a graph is a path that visits all the nodes/vertices exactly once, a hamiltonian cycle is a cyclic path, i.e. all nodes visited once and the start and the endpoint are the same. If we want to solve the snake game using this, we could divide the playable space in a grid and then try to just keep traversing on a hamiltonian cycle, this means you would eventually get all the rewards and never hit yourself unless you are longer than can fit on the screen. You can look at the code given in this Github repo.

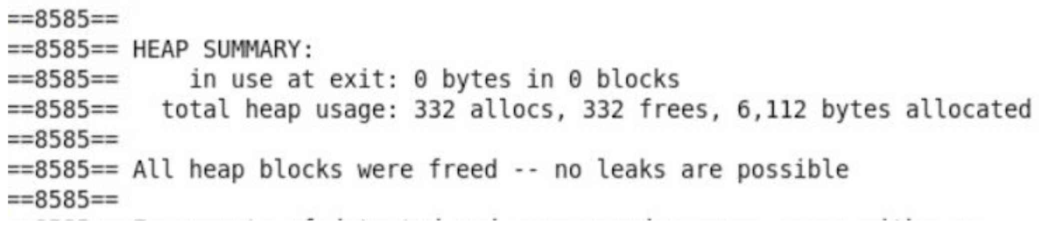
6.2 - Appendix B: Coding & Implementation

6.2.1 - Video Demonstration

https://youtu.be/7kicrRCsh8w



6.2.2 - Valgrind Clean



```
==8585==
==8585== HEAP SUMMARY:
==8585==     in use at exit: 0 bytes in 0 blocks
==8585==   total heap usage: 332 allocs, 332 frees, 6,112 bytes allocated
==8585==
==8585== All heap blocks were freed -- no leaks are possible
==8585==
```

6.2.3 - Final Results (Text Files)

6.2.3.1 - Final Score

982
1199
589
473
1042
1414
1301
1013
667
1135

846
935
982
1551
1236
942
1035
863
1168
373
820
1263
1341
831
649
1035
481
1388
962
1431
839
1766
1391
712
1361
946
527
527
908
775
1128
1087
808
731
731
1205
1323
1262
912

1002
618
1299
950
1424
1166
572
1409
1036
1276
1499
561
561
1391
854
1675
944
930
1379
876
1064
864
1229
1436
844
1161
695
1002
934
1195
1078
883
936
913
1468
849
795
1204
835

915
1157
841
936
964
1152
860
1231
1150
1292
540
540

6.2.3.2 - Final Snake Length

23
29
14
14
24
29
28
25
16
30
22
22
21
35
29
24
25
21
31
10
21
27
30
22
16

23
13
31
24
34
20
38
33
17
30
21
14
14
21
21
27
24
18
21
21
29
30
30
21
24
16
30
24
32
30
17
31
23
29
33
16
16
33
21

35
24
22
28
20
28
20
28
29
21
29
17
25
24
27
28
21
23
23
34
18
21
30
20
20
27
21
21
21
27
22
28
25
29
15
15

6.2.4 - Code

6.2.4.1 - snek_api.h file

```
#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* next;
} 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;
} 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();
```

6.2.4.2 - snek_api.c file

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

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

GameBoard* init_board(){
        CURR_FRAME=0;
        SCORE=0;
        MOOGLE_FLAG = 0;
        MOOGLES_EATEN=0;
        TIME_OUT = ((BOARD_SIZE * 4) - 4) * CYCLE_ALLOWANCE;
        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);
        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;
}
//updated
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;
        }

}

//updated
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());
}

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->occupancy[r1][r2]!=1){
                        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;
                }
                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;
                }

                // 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[2J"); // clear terminal ANSI code
//	fprintf(stdout, "\033[0;0H\n"); // 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");
		fflush(stdout);
	}

	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;
}

void end_game(GameBoard **board){
        //fprintf(stdout, "\033[2J");
        //fprintf(stdout, "\033[0;0H");
        fprintf(stdout, "\n\n\n--!!---GAME OVER---!!--\n\nYour score: %d\n\n\n\n", SCORE);
        fflush(stdout);
        // need to free all allocated memory
        // first snek
        SnekBlock **snekHead = &((*board)->snek->head);
        SnekBlock *curr;
        SnekBlock *prev;
        while ((*snekHead)->next != NULL) {
                curr = *snekHead;
                while (curr->next != NULL){
                        prev = curr;
                        curr = curr->next;
                }
                prev->next = NULL;
                free(curr);
        }
        free(*snekHead);
        free((*board)->snek);
        free(*board);
}
```

6.2.4.3 - Implemented Algorithm (search_func.h)

```
struct direct{
        int axis;
        int direction;
        struct direct *next;
};

int sizelist(struct direct* linked){
        struct direct* temp = linked;
        int counter = 0;

        while (temp->next){
        temp = temp->next;
        counter++;
        }
        return counter;
}

int sizelist2(struct SnekBlock* linked){
        struct SnekBlock* temp = linked;
        int counter = 0;

        while (temp->next){
        temp = temp->next;
        counter++;
        }
        return counter;
}

struct direct* intialize_linked(){
        struct direct *linked = malloc(sizeof(struct direct));

        linked -> next = NULL;
        return linked;
}

struct direct* new_direction(struct direct* linked,int direction, int axis){
        struct direct* new = malloc(sizeof(struct direct));
        if (new){
                new -> axis = axis;
                new -> direction = direction;
        }
        return new;
        free(new);
}

void insert_direction(struct direct* linked, int direction, int axis){

        struct direct*new = new_direction(linked, direction, axis);
        new -> next = linked -> next;
        linked -> next = new;

        if (axis == AXIS_Y && direction == DOWN){
                new->direction!=UP;
        }
        if (axis == AXIS_Y && direction == UP){
                new->direction!=DOWN;
        }
        if (axis == AXIS_X && direction == RIGHT){
                new->direction!=LEFT;
        }
        if (axis == AXIS_X && direction == LEFT){
                new->direction!=RIGHT;
        }

}

int dequeue_d(struct direct* link){
        struct direct* temp = link;
        while (temp->next){
                temp = temp->next;
        }
        int d = temp->direction;
        return d;
}
int dequeue_axis(struct direct* link){
        struct direct* temp;
        temp = link;
        while (temp->next){
                temp = temp->next;
        }
        int axis = temp->axis;

        return axis;
}

void free_direct(struct direct*link){
        struct direct* temp;
        struct direct* prev;

        temp = link;
        while (temp->next){
                prev=temp;
                temp = temp->next;}

        prev->next = NULL;
        free(temp);

}



int check_moogle_x(int cellvalue[BOARD_SIZE][BOARD_SIZE]){
        int x_coord = BOARD_SIZE;
        int y_coord = BOARD_SIZE;
        int coordinates[2];
        for(x_coord=0; x_coord<BOARD_SIZE; x_coord++){
                for(y_coord=0; y_coord<BOARD_SIZE; y_coord++){
                        if(cellvalue[x_coord][y_coord] == 20 || cellvalue[x_coord][y_coord] == 60){
                                return x_coord;
                break;
                }
            }
            if(cellvalue[x_coord][y_coord] == 20 || cellvalue[x_coord][y_coord] == 60){
                break;}
        }
        return x_coord;
}

int check_moogle_y(int cellvalue[BOARD_SIZE][BOARD_SIZE]){
        int x_coord = BOARD_SIZE;
        int y_coord = BOARD_SIZE;
        int coordinates[2];
        for(x_coord=0; x_coord<BOARD_SIZE; x_coord++){
            for(y_coord=0; y_coord<BOARD_SIZE; y_coord++){
                if(cellvalue[x_coord][y_coord] == 20 || cellvalue[x_coord][y_coord] == 60){
                    return y_coord;
                break;
                }
            }
            if(cellvalue[x_coord][y_coord] == 20 || cellvalue[x_coord][y_coord] == 60){
                break;}
        }
        return y_coord;
}

int check_moogle_value(int cellvalue[BOARD_SIZE][BOARD_SIZE]){
        int x_coord = 0;
        int y_coord = 0;
        int coordinates[2];
        int value = 0;
        for(x_coord=0; x_coord<BOARD_SIZE; x_coord++){
            for(y_coord=0; y_coord<BOARD_SIZE; y_coord++){
                if((cellvalue[x_coord][y_coord] == 20) || (cellvalue[x_coord][y_coord] == 60)){
                    value = cellvalue[x_coord][y_coord];
                break;
                }
            }
            if((cellvalue[x_coord][y_coord] == 20) || (cellvalue[x_coord][y_coord] == 60)){
                break;}
        }
        return value;
}

int search_snek(int coord[2], int occupancy[BOARD_SIZE][BOARD_SIZE]){
        int x_coord = 0;
        int y_coord = 0;
        int value = 0;
```

```
	for(x_coord=0; x_coord<BOARD_SIZE; x_coord++){
		for(y_coord=0; y_coord<BOARD_SIZE; y_coord++){
			if(occupancy[x_coord][y_coord]>0){
				value = 1;
			break;
			}
		}
		if(occupancy[x_coord][y_coord]>0){
			break;}
	}
	return value;
}

int which_way_x(struct Snek* snek,int occupancy[BOARD_SIZE][BOARD_SIZE], int direction){
	struct SnekBlock* temp = malloc(sizeof(struct SnekBlock));
	temp = snek->head->next;
	int value = 1;
	int same_x = -1;
	while (temp->next != NULL){
		if (temp->coord[y] == snek->head->coord[y]){
			same_x = temp->coord[x];
			break;
		}
		temp = temp->next;
	}

	if (same_x > snek->head->coord[x]){
		value = LEFT;
	}

	if (same_x < snek->head->coord[x]){
		value = RIGHT;
	}

	if (direction == UP && snek->head->coord[x] == 0)value = RIGHT;
	if (direction == UP && snek->head->coord[x] == BOARD_SIZE-1)value = LEFT;
	if (direction == DOWN && snek->head->coord[x] == 0)value = RIGHT;
	if (direction == DOWN && snek->head->coord[x] == BOARD_SIZE-1)value = LEFT;

	return value;
}


int which_way_y(struct Snek* snek,int occupancy[BOARD_SIZE][BOARD_SIZE], int direction){
	struct SnekBlock* temp = snek->head->next;
	int value = 1;
	int same_x = 1;
	while (temp->next != NULL){
```

```
                if (temp->coord[x] == snek->head->coord[x]){
                        same_x = temp->coord[y];
                        break;
                }
                temp = temp->next;
        }

        if (same_x > snek->head->coord[y]){
                value = UP;
        }

        if (same_x < snek->head->coord[y]){
                value = DOWN;
        }

        if (direction == RIGHT && snek->head->coord[y] == BOARD_SIZE-1)value = UP;
        if (direction == RIGHT && snek->head->coord[y] == 0)value = DOWN;
        if (direction == LEFT && snek->head->coord[y] == BOARD_SIZE-1)value = UP;
        if (direction == LEFT && snek->head->coord[y] == 0)value = DOWN;

        return value;
}

int go_around(struct Snek* snek, int target[2], int axis, int direction, int check_moogle_value, struct direct* linked,
int occupancy[BOARD_SIZE][BOARD_SIZE], int go_x, int go_y){

        int move_around = 0;
        int value_x;
        int value_y;

        if (direction == UP && axis == AXIS_Y &&
((((occupancy[snek->head->coord[y]-1][(snek->head->coord[x])]) == 1)) || snek->head->coord[y] == 0) && go_x
== 0 && snek->head->coord[y]-1 > -1){
                value_x = which_way_x(snek, occupancy, direction);
                insert_direction(linked, value_x, AXIS_X);
                move_around = 1;
        }

        if (direction == DOWN && axis == AXIS_Y &&
((((occupancy[snek->head->coord[y]+1][(snek->head->coord[x])]) == 1)) || snek->head->coord[y] ==
BOARD_SIZE-1) && go_x == 0 && snek->head->coord[y]+1 < BOARD_SIZE){
                value_x = which_way_x(snek, occupancy, direction);
                insert_direction(linked, value_x, AXIS_X);
                move_around = 1;
        }
```

```
	if (direction == RIGHT && axis == AXIS_X &&
((((occupancy[(snek->head->coord[y])][snek->head->coord[x]+1]) == 1)) || snek->head->coord[x] ==
BOARD_SIZE-1) && go_y == 0 && snek->head->coord[x]+1 < BOARD_SIZE){
		value_y = which_way_y(snek, occupancy, direction);
		insert_direction(linked, value_y, AXIS_Y);
		move_around = 1;
	}

	if (direction == LEFT && axis == AXIS_X &&
((((occupancy[(snek->head->coord[y])][snek->head->coord[x]-1]) == 1)) || snek->head->coord[x] == 0) && go_y
== 0 && snek->head->coord[x]-1 > -1){
		value_y = which_way_y(snek, occupancy, direction);
		insert_direction(linked, value_y, AXIS_Y);
		move_around = 1;
	}

	return move_around;
}

int real_turn_around(struct Snek* snek, int target[2], int axis, int direction, int check_moogle_value, struct direct*
linked, int occupancy[BOARD_SIZE][BOARD_SIZE], int go_x, int go_y){
	int turn_around = 0;

	if (check_moogle_value > 0){
		if (direction == UP && axis == AXIS_Y &&  target[y] > snek->head->coord[y] && target[x] ==
snek->head->coord[x]){
			int value_x = which_way_x(snek, occupancy, direction);
			insert_direction(linked, value_x, AXIS_X);
			insert_direction(linked, DOWN, AXIS_Y);
			turn_around = 1;
		}

		if (direction == DOWN && axis == AXIS_Y && target[y] < snek-> head->coord[y] && target[x]
== snek->head->coord[x]){
			int value_x = which_way_x(snek, occupancy, direction);
			insert_direction(linked, value_x, AXIS_X);
			insert_direction(linked, UP, AXIS_Y);
			turn_around = 1;
		}

		if (direction == RIGHT && axis == AXIS_X && target[x] < snek->head->coord[x] && target[y]
== snek->head->coord[y]){
			int value_y = which_way_y(snek, occupancy, direction);
			insert_direction(linked, value_y, AXIS_Y);
			insert_direction(linked, LEFT, AXIS_X);
			turn_around = 1;
		}
		if (direction == LEFT && axis == AXIS_X && target[x] > snek->head->coord[x] && target[y]
== snek->head->coord[y]){
			int value_y = which_way_y(snek, occupancy, direction);
			insert_direction(linked, value_y, AXIS_Y);
			insert_direction(linked, RIGHT, AXIS_X);
			turn_around = 1;
		}
	}
	return turn_around;
}

void calculate_route(struct Snek* snek, int target[2], int axis, int direction, int check_moogle_value, struct direct*
linked, int occupancy[BOARD_SIZE][BOARD_SIZE], int go_x, int go_y){

	int bruh = 0;

	if (check_moogle_value>1 && sizelist(linked) == 0){
		if (!(target[x] == snek->head->coord[x] || target[y] == snek->head->coord[y])){

			if (snek->head->coord[x] < target[x] && axis == 1){
				insert_direction(linked, RIGHT, AXIS_X);
			}

			if (snek->head->coord[x] > target[x] && axis == 1){
				insert_direction(linked, LEFT, AXIS_X);
			}

			if (snek->head->coord[y] < target[y] && axis == -1){
				insert_direction(linked, DOWN, AXIS_Y);
			}

			if (snek->head->coord[y] > target[y] && axis == -1){
				insert_direction(linked, UP, AXIS_Y);
			}


			if (axis == -1){

				if (snek->head->coord[x] < target[x]){
					insert_direction(linked, RIGHT, AXIS_X);
					bruh = 1;
				}

				if (snek->head->coord[x] > target[x]){
					insert_direction(linked, LEFT, AXIS_X);
					bruh = 1;
				}
			}
```

```
				if (axis == 1){

					if (snek->head->coord[y] < target[y]){
							insert_direction(linked, DOWN, AXIS_Y);
							bruh = 1;
					}

					if (snek->head->coord[y] > target[y]){
						insert_direction(linked, UP, AXIS_Y);
						bruh = 1;
					}
				}
			} else {
				if (snek->head->coord[x] < target[x]){
					insert_direction(linked, RIGHT, AXIS_X);
					bruh = 1;
				}

				if (snek->head->coord[x] > target[x]){
					insert_direction(linked, LEFT, AXIS_X);
					bruh = 1;
				}

				if (snek->head->coord[y] < target[y]){
					insert_direction(linked, DOWN, AXIS_Y);
					bruh = 1;
				}

				if (snek->head->coord[y] > target[y]){
					insert_direction(linked, UP, AXIS_Y);
					bruh = 1;
				}
			}
		}
}
```

6.2.4.4 - main.c file

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

int turn_around;

void play_game() {
	FILE *final_score = fopen("scores.txt", "w");
	FILE *final_length = fopen("length.txt", "w");
```

```
		for (int i = 1; i<=100; i++){
			printf("TRIAL %d", i);
			GameBoard* board = init_board();
			show_board(board);

			int axis = AXIS_INIT;
			int direction = DIR_INIT;

			int play_on = 1;
			int coord[2];

			struct direct* list = intialize_linked();

			while (play_on){
				coord[x] = board->snek->head->coord[x];
				coord[y] = board->snek->head->coord[y];


				int moogle_value = check_moogle_value(board ->cell_value);
				int moogle_y = check_moogle_y(board -> cell_value);
				int moogle_x = check_moogle_x(board -> cell_value);
				int target[2] = {moogle_y, moogle_x};

				unsigned short go_x = (axis == AXIS_Y && direction == DOWN && coord[y] ==
(BOARD_SIZE - 1)) || (axis == AXIS_Y && direction == UP && coord[y] == 0);
				unsigned short go_y = (axis == AXIS_X && direction == RIGHT && coord[x] ==
(BOARD_SIZE - 1)) || (axis == AXIS_X && direction == LEFT && coord[x] == 0);

				int last_dequeue = 0;
				int move_around = 0;
				int already_found = 0;

				if (sizelist(list) == 0)calculate_route(board -> snek, target, axis, direction, moogle_value,
list, board->occupancy, go_x, go_y);

					if (sizelist(list) == 0){
						move_around = go_around(board -> snek, target, axis, direction,
moogle_value, list, board->occupancy, go_x, go_y);
						already_found = 1;
					}

					if (sizelist(list) == 1 && move_around!= 1){
						turn_around = real_turn_around(board -> snek, target, axis, direction,
moogle_value, list, board->occupancy, go_x, go_y);
						move_around = go_around(board -> snek, target, axis, direction,
moogle_value, list, board->occupancy, go_x, go_y);
```

```
                                        if (turn_around == 0 && move_around == 0) move_around =
go_around(board -> snek, target, list -> next -> axis, list -> next -> direction, moogle_value, list, board->occupancy,
go_x, go_y);
                                }

                                if (sizelist(list) == 2 && turn_around !=1 && move_around !=1){
                                        turn_around = real_turn_around(board -> snek, target, axis , direction,
moogle_value, list, board->occupancy, go_x, go_y);
                                        move_around = go_around(board -> snek, target, axis, direction,
moogle_value, list, board->occupancy, go_x, go_y);
                                        if (turn_around == 0 && move_around == 0){
                                                if(target[x] == coord[x] || target[y] == coord[y]){
                                                        move_around = go_around(board -> snek, target, list
-> next -> axis, list -> next -> direction, moogle_value, list, board->occupancy, go_x, go_y);
                                                } else {
                                                move_around = go_around(board -> snek, target, list -> next
-> next -> axis, list -> next -> next -> direction, moogle_value, list, board->occupancy, go_x, go_y);
                                                }
                                        }
                                }


                                if (sizelist(list) >= 3){
                                        if (move_around == 1 || sizelist(list) == 4){
                                                free_direct(list);
                                                free_direct(list);
                                        }

                                        if (turn_around == 1) free_direct(list);
                                }

                                if (sizelist(list) == 2){
                                        if((target[x] == coord[x] || target[y] == coord[y]) && turn_around == 0
&& move_around == 0){
                                                free_direct(list);
                                        }

                                        if((target[x] == coord[x] || target[y] == coord[y]) && turn_around ==
1){
                                                axis = dequeue_axis(list);
                                                direction = dequeue_d(list);
                                                free_direct(list);
                                        }

                                        if(move_around == 1){
                                                free_direct(list);
                                        }
```

```
                if (turn_around == 0 && move_around == 0){
                        axis = dequeue_axis(list);
                        direction = dequeue_d(list);
                }
        }

        if (sizelist(list) == 1){
                if (turn_around == 1){
                        turn_around++;
                }

                else{
                axis = dequeue_axis(list);
                direction = dequeue_d(list);
                free_direct(list);
                last_dequeue = 1;
                turn_around = 0;
                }
        }

        if (go_x && sizelist(list) == 0 && last_dequeue == 0) {
                axis = AXIS_X;
                if (coord[x] < (int)(BOARD_SIZE / 2)){
                        direction = RIGHT;
                } else {
                        direction = LEFT;
                }
        } else if (go_y && sizelist(list) == 0 && last_dequeue == 0) {
                axis =  AXIS_Y;
                if (coord[y] < (int)(BOARD_SIZE / 2)){
                        direction = DOWN;
                } else {
                        direction = UP;
                }
        }

show_board(board);
play_on = advance_frame(axis, direction, board);
printf("Going ");

if (axis == AXIS_X){
        if (direction == RIGHT){
                printf("RIGHT");
        } else {
                printf("LEFT");
        }
} else {
        if (direction == UP){
```

```
                    printf("UP");
                } else {
                    printf("DOWN");
                }
            }
            printf("\n");
            usleep(5550);
        }
        fprintf(final_score, "%d\n", SCORE);
        fprintf(final_length, "%d\n", board->snek->length);
        end_game(&board);
    }
    fclose(final_score);
    fclose(final_length);
}


int main(){
    play_game();
    return 0;
}
```