CMP 304 Unit 1 Assessment

AI Comparison

Cameron-Stewart Smart

1901578

Table of Contents

Introduction

During this assessment two AI's were explored: the Rule Based System AI algorithm and the Finite State Machine AI algorithm.

This was completed using a base project that would usually be played by a human, a game of Snake. This beloved game is a grid with 4 walls that holds a snake. The snake must traverse left, right, up and down the grid in the pursuit of fruit. The greedy snake will continue to eat fruit, getting larger every time they consume, until they are too large to fit in the box. The snake can move, eat fruit, hit a wall or hit itself. The last two would cause the game to end.

The first AI implemented was the Rule Based System. This was used first as it allows for a limited amount of actions to be converted into rules and for the AI to decide which suits the current situation best and since the snake only has 4 directions of movement this maps over to a Rule Based System quite well.

The second AI implemented was a Finite State Machine. This maps the snake into multiple states. A switch is then used to respond depending on which state the snake is in. This should work well for this problem as each move the snake takes will either be clear; meaning there are no obstacles or blocked; meaning the snake has to work around obstacles to get to their target, the fruit.

Some of metrics used to compare these will be statistics such as moves; the amount of left, right, up and down moves the snake makes and score; the amount of fruit eaten by the snake. The AI's will also be compared on how 'well' they place snake compared to how a human would play.

Methodology

Base Snake Program

During development a GitHub repository was created. Frequent commits were made in order to track the progress of the project and document development. This can be accessed here: GitHub Link.

To compare the two AI's first a game of snake had to be implemented as demonstrated below, this is not in huge detail as the AI implementation's importance is greater. Firstly the functions that allow text to be drawn on the console were implemented (*Figure 1*).

```
void initialiseDisplay() // This sets up the display to have text displayed on it
{
   COORD console_size = { 110, 30 }; // Sets size of console
   hConsole = CreateFile(TEXT("CONOUT$"), GENERIC_WRITE | GENERIC_READ,

      FILE_SHARE_READ | FILE_SHARE_WRITE,
      0L, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0L);

   SetConsoleScreenBufferSize(hConsole, console_size);
   GetConsoleScreenBufferInfo(hConsole, &consoleInfo);
}

void draw(int x, int y, string s) // For drawing anything on the screen (position x, position y, string output)
{
   COORD cursorPos; // Creates a variable to hold the coordinates

   cursorPos.X = x;
   cursorPos.Y = y + scrollNumber;

   SetConsoleCursorPosition(hConsole, cursorPos);

   cout << s;
}

void setColor(int fcolor, int bcolor = 0) // Allows the colour of the text being printed to be set
{
   SetConsoleTextAttribute(hConsole, (WORD)((bcolor << 4) |

      fcolor));
}

void clearScreen(void) // Clears the display by manually overwriting with spaces
{
   scrollNumber = 0;
   setColor(15, 0); // Selects colour to reset with (white foreground and black background)

   for (int index = 0; index <= 30; index++) { // Goes through all the lines and wipes them
      draw(0, scrollNumber, "                                                    \n");
   }
}
```

Figure 1: initialiseDisplay, draw, setColor and clearScreen functions.

These functions would allow the game to have a simple user interface that updates for each move of the snake and to display the current score. The next step was to create the structures to hold the data:

```
int moveDir = 0; // 1 = right, 2 = down, 3= left, 4 = up
int scrollNumber = 0;
int snaketrix[20][20]; // Like matrix, but a snake

// Why a matrix?

// -> a matrix allows for a 2 dimenstional grid structure to be easily represented. This can be accessed
// using the coordinates and will hold the contents of each space. Eg. if the space is snake, empty or fruit.

struct coordinates { int x, y; }; // This struct is used anytime coordinates on the snaketrix are needed
std::deque<coordinates> body; // A double ended queue is used to hold the snake positions

// Why a deque?

// -> A deque allows for both ends of the queue to be read and popped. A deque works best for snake
// as when the snake moves a new position is added for where the head is moving at the back of the
// queue and the end of the snake can be popped from the front of the queue as the snake leaves
// that space.
```

Figure 2: set up of data types.

As shown above (*Figure 2*) a 2 dimensional array matrix was used to hold the status of each position in the grid and a double ended queue used to hold the position of each body part of the snake. The next step was to then get the input from the human user. This is one of the two main functions later on adapted to the make game playable by an AI.

Another function was then created to output the bounds of the grid that the game takes place in (*Figure 3*).

```
void drawSnakeFrame() // Draws the box that the snake is confined to
{
   stringstream ss; // Creates a string stream variable to allow characters to be converted into strings for output

   setColor(1); // Blue
   draw((5), (1), "##########     #########");
   setColor(2); // Green
   draw((15), (1), "SNAKE");
   setColor(5); // Purple
   draw((5), (2), "########################");
   setColor(1); // Blue
   draw((5), (2), "#");
   draw((28), (2), "#");

   for (int i = 0; i < 10; i++) { // Loop for center section
      setColor(1); // Blue
      draw((5), (3 + i), "#");
      setColor(5); // Purple
      draw((6), (3 + i), "#                    #");
      setColor(1); // Blue
      draw((28), (3 + i), "#");
   }

   setColor(5); // Purple
   draw((5), (13), "########################");
   setColor(1); // Blue
   draw((5), (13), "#");
   draw((28), (13), "#");
   draw((5), (14), "########################");
   draw((0), (15), "                                          ");

}
```

Figure 3: drawSnakeFrame function.

The next function implemented draws the starting grid (*Figure 4*).

```
void drawGrid() { // This draws the contents of the snake game by checking content of snaketrix
   for (int outer = 0; outer < 20; outer++) {
      for (int inner = 0; inner < 10; inner++) {
         if (snaketrix[outer][inner] == 0) { // Empty space. Even empty spaces must be redrawn
// each move because the snake could have been here and moved
            setColor(1); // "Blue"
            draw((outer+7), (inner+3), " ");
         }
         else if (snaketrix[outer][inner] == 1) { // Snake
            setColor(2); // Green
            draw((outer+7), (inner+3), "S");
         }
         else if (snaketrix[outer][inner] == 3) { // Fruit
            setColor(5); // Purple
            draw((outer + 7), (inner + 3), "O");
         }
      }
   }

   draw((0), (15), "               "); // This moves the cursor to bottom after drawing
}
```

Figure 4: drawGrid function.

Another function was added initialise the grid as either empty or containing the start of the snake (*Figure 5*).

```
void initialiseGrid() { // Sets up the snaketrix
   for (int outer = 0; outer < 20; outer++) { // Fills whole snaketrix with empty spaces
      for (int inner = 0; inner < 20; inner++) {
         snaketrix[outer][inner] = 0;
      }
   }

   srand(time(NULL)); // Sets random

   int startX = rand() % 18; // Starting x decided
   int startY = rand() % 9; // Starting y decided

   snaketrix[startX][startY] = 1; // Creates snake head part 1 here

   coordinates headXY; // Sets head position as space above
   headXY.x = startX;
   headXY.y = startY;
   body.push_back(headXY);

   snaketrix[startX+1][startY] = 1; // Creates first part of body one to the right
   headXY.x = startX+1;
   body.push_back(headXY);
}
```

Figure 5: initialiseGrid function.

A function was also required to add new fruit when the old fruit is eaten (*Figure 6*).

```
int newFruit() { // Adds a new fruit to snaketrix
   srand(time(NULL));
   int fruitSet = 0;
   int endCheck = 0;

   while (fruitSet == 0) { // Uses while loop to detect when a new space is found
      int startX = (rand() % 20); // Chooses random space for new fruit
      int startY = (rand() % 10);
      endCheck++; // Increases the amount of spaces checked

      if (endCheck == 200) { // If all but one spaces are full of snake or fruit then the game is won
         return 1; // Returns end game code
      }

      if ((snaketrix[startX][startY] != 1) and (snaketrix[startX][startY] != 3)) { // Checks if space is fruit or snake
         snaketrix[startX][startY] = 3; // If space is empty then fruit is added
         setColor(5); // Purple
         draw((startX + 7), (startY + 3), "O"); // Draws fruit
         fruitSet = 1; // Ends while loop
      }
      return 0;
   }
}
```

Figure 6: newFruit function.

The most complicated part of the base program was moving the snake as the matrix had to be updated, the new position had to be checked and then the end of the snake had to be updated. This all then had to be reflected on the image of the grid in the console as shown below (only one of four cases is shown as they are all identical but adapted depending on how the snake moved) (*Figure 7*).

```
int moveSnake(int moveDir) { // This is the second main element of the game, where the snake is moved.

   coordinates headXY; // Coordinates to hold current head of snake

   switch (moveDir) { //This switches what code to use base on chosen move direction

   case 1:
      headXY = body.back(); // Sets current position of head to the body item at back of queue (front of snake)
      headXY.x = headXY.x + 1; // Sets new position of head right one space
      body.push_back(headXY); // Pushes new head position onto queue
      setColor(2); // Green
      draw((headXY.x + 7), (headXY.y + 3), "S"); // Draws new snake head

      if (snaketrix[headXY.x][headXY.y] == 1) { // Checks the space of the new head is not already snake
         return 1; // Returns end game code
      }

      if (snaketrix[headXY.x][headXY.y] == 3) { // Checks if space of the new head contains fruit
         snaketrix[headXY.x][headXY.y] = 1; // Sets fruit space to now contain snake
         return 3; // Returns fruit hit code
      }

      // This code runs when the new space is empty

      snaketrix[headXY.x][headXY.y] = 1; // Sets new space to be snake

      headXY = body.front(); // Changes head holder to back of snake in order to draw over space and pop front of queue
      setColor(3);
      draw((headXY.x + 7), (headXY.y + 3), " "); // Draws empty space over old tail of snake
      snaketrix[headXY.x][headXY.y] = 0; // Sets position of old tail in snaketrix to empty
      body.pop_front(); // Pops old tail from queue

      break;
[......]OTHER CASES[......]
      }

   headXY = body.back(); // Sets holder to head of snake to check if in bounds of game area

   if ((headXY.x >= 20 or headXY.x < 0) or (headXY.y >= 10 or headXY.y < 0)) { // Checks snake has not hit wall
      return 1; // Returns game over code
   }

   return 0;
```

This was all called by the main function. This program is easily adaptable for AI's as it is broken down into smaller functions and has one main function that reflects a move of the snake that can easily have an AI algorithm added into (*Figure 8)*.

```
int main() {

   int dir = 1;
   int result;
   int moves = 0;
   int score = 0;

   initialiseDisplay(); // Initialises display
   clearScreen(); // Clears any text on screen
   drawSnakeFrame(); // Draws frame for snake
   initialiseGrid(); // Sets up the snaketrix
   drawGrid(); // Draws main game contents
   newFruit(); // Adds starting fruit onto snaketrix and draws

   while (1) { // Loops until game ends
      dir = moveLoop(); // Gets direction from AI or user
      result = moveSnake(dir); // Moves snake
      moves++; // Increments move counter

      if (result == 1) { // If game over code is returned from snake move then game over
         break; // End game
      }

      if (result == 3) { // If fruit is hit by the snake increase score and add new fruit
         int endCheck = newFruit(); // New fruit
         if (endCheck == 1) { // If game over code is returned from new fruit (snaketrix full)
            return 0; // End game
         }
         score++; // Increment score counter
      }
      updateScore(to_string(score), to_string(moves)); // Reprints score
   }

   endScreen(to_string(score), to_string(moves)); // Displays end screen once game loop is broken

   return 0; // End
}
```

Figure 8: main function.

The functions are all prototyped in game.h and interact with each other through the source file by passing data back to the main function and using global variables to store the snaketrix. Below you can see the contents of game.h where each function is prototyped (*Figure 9*).

```
void initialiseDisplay();
void draw(int x, int y, string s);
void setColor(int fcolor, int bcolor);
void clearScreen(void);
void drawSnakeFrame();
void initialiseGrid();
void drawGrid();
int moveLoop();
int moveSnake(int moveDir);
int newFruit();
void endScreen(string score, string moves);
void updateScore(string score, string moves);
```

Figure 9: game.h.

AI 1: Rule Based System

The first AI that was implemented was the Rule Based System. In order to implement the AI a new function was created. This held the rule based structure in the form of if statements. Since the Rule Based System is a simple one the algorithm consists of a loop that finds the target position and 8 if statements that take the current surrounding spacers of the snaketrix into account and returns which direction the snake should move.

Firstly the function creates two new variables to hold the position of the target and the head of the snake. The target position is then initialised with 0 and the snake head variable with the back item in the snake body queue. The function then loops through the snaketrix to find the fruit and the position is set as the target as shown below (*Figure 10*).

```
coordinates targetXY;
coordinates ruleHeadXY;

targetXY.x = 0;
targetXY.y = 0;

ruleHeadXY = body.back();

for (int outer = 0; outer < 20; outer++) { // Lopps through snaketrix to find fruit
   for (int inner = 0; inner < 20; inner++) {
      if (snaketrix[outer][inner] == 3) {
         targetXY.x = outer;
         targetXY.y = inner;
         break;
      }
   }
}
```

Figure 10: Start of Rule Based System function.

The second part of the function is the collection of if statements that decides which direction the snake should move in and then returns it to the main function (*Figure 11)*.

```
if ((targetXY.x < ruleHeadXY.x) and (snaketrix[(ruleHeadXY.x)-1][ruleHeadXY.y] != 1)) {
// If target is left and left is not snake go left
   return 3;
} else if ((targetXY.x > ruleHeadXY.x) and (snaketrix[(ruleHeadXY.x) + 1][ruleHeadXY.y] != 1)) {
// If target is up and up is not snake go up
   return 1;
} else if ((targetXY.y < ruleHeadXY.y) and (snaketrix[ruleHeadXY.x][(ruleHeadXY.y) - 1] != 1)) {
// If target is down and down is not snake go down
   return 4;
} else if ((targetXY.y > ruleHeadXY.y) and (snaketrix[ruleHeadXY.x][(ruleHeadXY.y) + 1] != 1)) {
// If target is right and right is not snake go right
   return 2;
} else if (snaketrix[(ruleHeadXY.x) - 1][ruleHeadXY.y] == 2) { // If space left is empty go left
   return 3;
} else if (snaketrix[(ruleHeadXY.x) + 1][ruleHeadXY.y] == 2) { // If space right is empty go right
   return 1;
} else if (snaketrix[ruleHeadXY.x][(ruleHeadXY.y) - 1] == 2) { // If space up if empty go up
   return 4;
} else if (snaketrix[ruleHeadXY.x][(ruleHeadXY.y) + 1] == 2) { // If space down is empty go down
   return 2;
} else {
   return 5;
}
```

Figure 11: Collection of if statements in the Rule Based System.

The first four if statements check the position to the left, up, down and right to see if the the target position is in their direction and checks that the position is not a snake. Then the second four if statements check left, right, up and down to see if they are empty and moves in that direction.

AI 2: Finite State Machine

The second AI implemented was a finite state machine. The finite state machine is much more structured then the rule based system. Where the rule based system is just a list of if statements looking for a condition to be met the finite state machine is more of a hierarchal tree using classes and subclasses to decide how the snake will move. In the same manor as the Rule Based System a function was created to take information from the snaketrix and decide how to react then return the direction to move in. The finite state machine is made of three parts. First a class called SnakeState is created. This holds the state of the snake as either clear or blocked (*Figure 12*).

```
enum class SnakeStates { clear, blocked }; // Class that holds the state of the snake
class SnakeSurroundings { // Class that holds the state of surrounding spaces
public:
   bool up;
   bool upRight;
   bool right;
   bool downRight;
   bool down;
   bool downLeft;
   bool left;
   bool upLeft;
};
```

Figure 12: SnakeStates and SnakeSurroundings classes.

The second part of the finite state machine is the set of if statements that look at the snaketrix and decides what the current state of the snake is. If all of the spaces around the snake in the snaketrix are empty then the snakes status is set as clear. If any of the spaces have an obstacle such as the edge of the grid or another part of the snake then the status of the snake is set as blocked (*Figure 13*).

```
if ((snaketrix[(finiteHeadXY.x)][(finiteHeadXY.y) - 1] == 2) or (snaketrix[(finiteHeadXY.x)][(finiteHeadXY.y) - 1] == 3)) {
// Checks if space up is clear
      snake_surroundings.up = false; // Sets space up as clear in class
   }
   if ((snaketrix[(finiteHeadXY.x) + 1][(finiteHeadXY.y) - 1] == 2) or (snaketrix[(finiteHeadXY.x) + 1][(finiteHeadXY.y) - 1] == 3)) {
// Checks if space up and right is clear
      snake_surroundings.upRight = false; // Sets space up and right as clear in class
   }
   if ((snaketrix[(finiteHeadXY.x) + 1][(finiteHeadXY.y)] == 2) or (snaketrix[(finiteHeadXY.x) + 1][(finiteHeadXY.y)] == 3)) {
// Checks if space right is clear
      snake_surroundings.right = false; // Sets space right as clear in class
   }
   if ((snaketrix[(finiteHeadXY.x) + 1][(finiteHeadXY.y) + 1] == 2) or (snaketrix[(finiteHeadXY.x) + 1][(finiteHeadXY.y) + 1] == 3)) {
// Checks if space down and right is clear
      snake_surroundings.downRight = false; // Sets space down and right as clear in class
   }
   if ((snaketrix[(finiteHeadXY.x)][(finiteHeadXY.y) + 1] == 2) or (snaketrix[(finiteHeadXY.x)][(finiteHeadXY.y) + 1] == 3)) {
// Checks if space down is clear
      snake_surroundings.down = false; // Sets space down as clear in class
   }
   if ((snaketrix[(finiteHeadXY.x) - 1][(finiteHeadXY.y) + 1] == 2) or (snaketrix[(finiteHeadXY.x) - 1][(finiteHeadXY.y) + 1] == 3)) {
// Checks if space down and left is clear
      snake_surroundings.downLeft = false; // Sets space down and left as clear in class
   }
   if ((snaketrix[(finiteHeadXY.x) - 1][(finiteHeadXY.y)] == 2) or (snaketrix[(finiteHeadXY.x) - 1][(finiteHeadXY.y)] == 3)) {
// Checks if space left is clear
      snake_surroundings.left = false; // Sets space left as clear in class
   }
   if ((snaketrix[(finiteHeadXY.x) - 1][(finiteHeadXY.y) - 1] == 2) or (snaketrix[(finiteHeadXY.x) - 1][(finiteHeadXY.y) - 1] == 3)) {
// Checks if space up and left is clear
      snake_surroundings.upLeft = false; // Sets space up and left as clear in class
   }
```

Figure 13: If statements checking status of each direction in snaketrix.

The third part is a switch that takes in the snakes current status and responds accordingly (*Figure 14*). If the snakes status is clear then it moves on space in the direction of the target (the fruit). If the snakes status is blocked then it uses if statements to check how to respond. Each nest of if statements in the code checks the direction of the target, then checks if that direction is clear. If that direction is clear it moves in that direction. If that direction is not clear then it works its way around the obstacle.

```
if (targetXY.x < finiteHeadXY.x) { // If target is left
   if (snake_surroundings.left == false) { // Checks if left is clear
      return 3;
   } else if (snake_surroundings.up == false) { // If left is not clear and up is clear
      return 4;
   } else if (snake_surroundings.down == false) { // If left and up are not clear and down is clear
      return 2;
   } else if (snake_surroundings.right == false) { // If left, up and down are not clear and right is clear
      return 1;
   }
}
[REPEATED FOR ALL FOUR DIRECTIONS]
```

Figure 14: If statements deciding which direction to move.

Results

AI 1: Rule Based System

The first AI created was the rule based system. This system works well initially. It manages to move in the direction of the fruit and hone in on it, collect in and move on to the next one. The AI starts to struggle when it has to negotiate avoiding itself. The snake struggles to get around itself and struggles to see into the future due to how simple it is. Eventually the snake seems to always find itself stuck in a square and when none of its if statements hit eventually it eats itself as shown below (*Figure 15*).

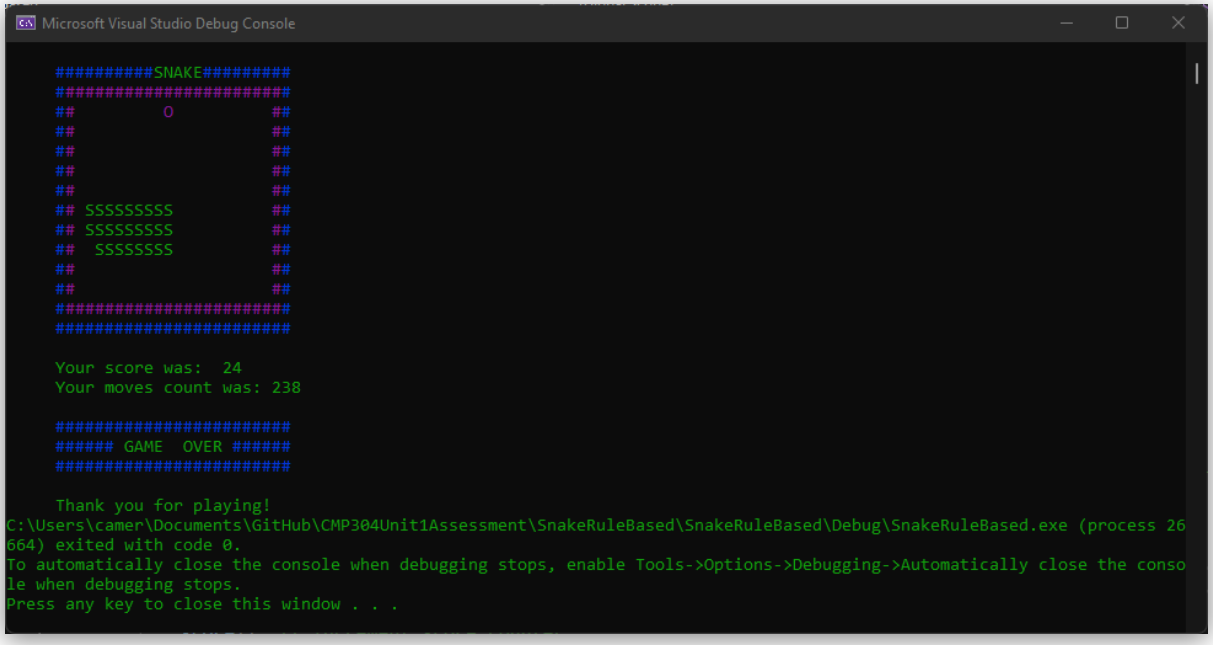


Figure 15: Rule based system snake hitting itself.

Please see the appendix for a full set of results containing the score, moves and end game trigger of each test that was run (*Appendix: Results*).

AI 2: Finite State Machine

The second AI implemented, the finite state machine has a few improvements over the rule based system. The finite state machine is better at over coming obstacles as it is smarter. The finite state machine is fully aware of its surroundings before is decides which direction to move in. Since the finite state machine uses classes it has two states: clear or blocked. This allows it to know if it has to work around obstacles when moving. If it does have to work around obstacles it is much smarter in the way it does it since it checks if the space is an obstacle then works around it where the rule based system just goes in another direction. The finite state machine does have a strange issue where it gets confused and hits the wall of the game. Many attempts have been made to fix this issue but It happens in about every one of three attempts as shown below (*Figure 16*).

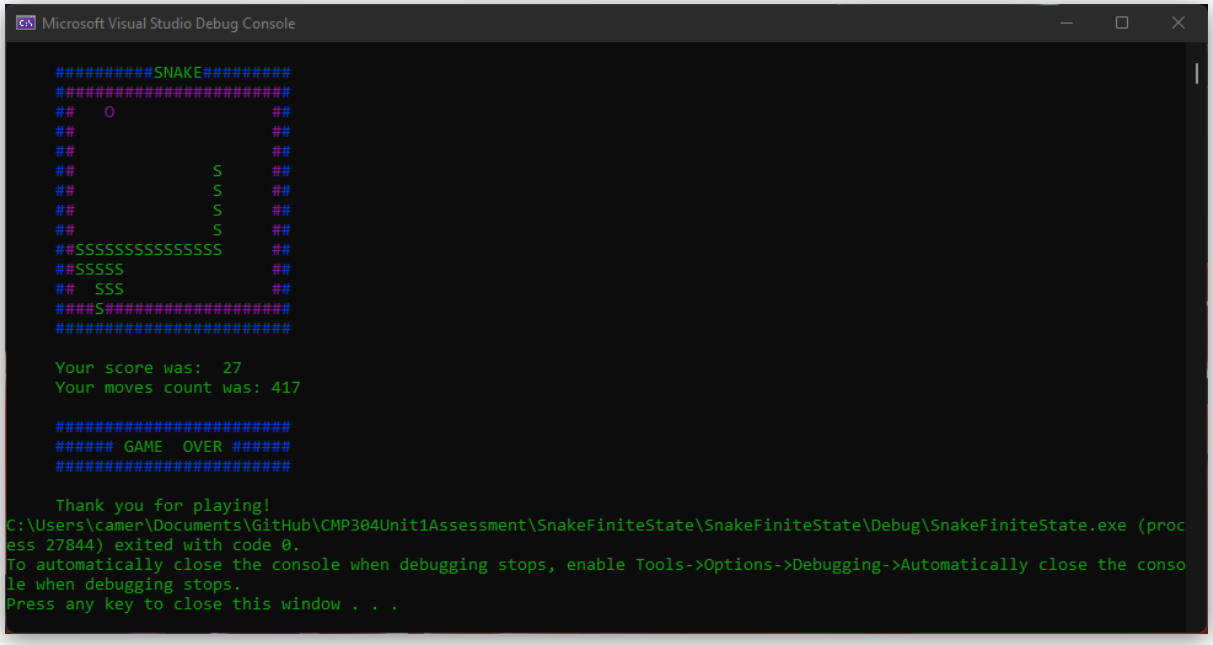


Figure 15: Finite state machine snake hitting wall.

Please see the appendix for a full set of results containing the score, moves and end game trigger of each test that was run (*Appendix: Results*).

Conclusion

Overall the finite state machine does a better job than the rule based system when playing snake. Both AI's lack foresight which is their main downfall but the finite state machine is smarter in the way it approaches obstacles. The rule based systems main downfall is how limited it is. The AI uses a list of 9 if statements, the first 4 checking if a nearby space is not a snake, then moving and the second 4 making the snake move and the last if ending the game. The finite state machine actually checks status of the snake, wether it is clear or blocked. If clear it moves closer to the target and if blocked if avoids obstacles. If the snake has to avoid obstacles it checks which general direction it must move in and then if the space in that direction is blocked it moves in the direction most likely to get it around the obstacle.

If more time was spent on the project an effective next step would be to implement a path finding algorithm such as A* into both of the AI's. This would give them foresight and allow them to choose a path that would avoid any obstacles and go straight for the target, allowing them to survive longer.

To test the solutions each was allowed to run through the game until it reached an end point by either hitting the side of the box or hitting itself. The score and moves were then recorded along with the reason for the game ending.

As shown in the results given in the previous section, a higher average score was achieved by the finite state machine. This is due to its better ability to get around obstacles. The finite state machine did also require more moves though. To compare the two programs the lower, middle and average quartiles were calculated from test data. Perhaps most interesting about the result of this is that the Finite state machines averages grew further apart the higher the result. For example the RBS had a Q1 of 18.60 and the FSM had a Q1 of 22.76 which is a difference of 4.16 but the RBS had a Q3 of 29.44 and the FSM had a Q3 of 35.76 which is a difference of 6.32 (*Figure 16*). So as illustrated by the graph on the next page below (*Figure 17*) the two datasets grow apart as the FSM was achieving higher results.

Comparison

	Rule Based System	Finite State Machine	Difference
Lower Quartile	18.6	22.76	4.16
Median Quartile	24.02	29.26	5.24
Upper Quartile	29.44	35.76	6.32

Figure 16: Comparison of the two AI's average results.

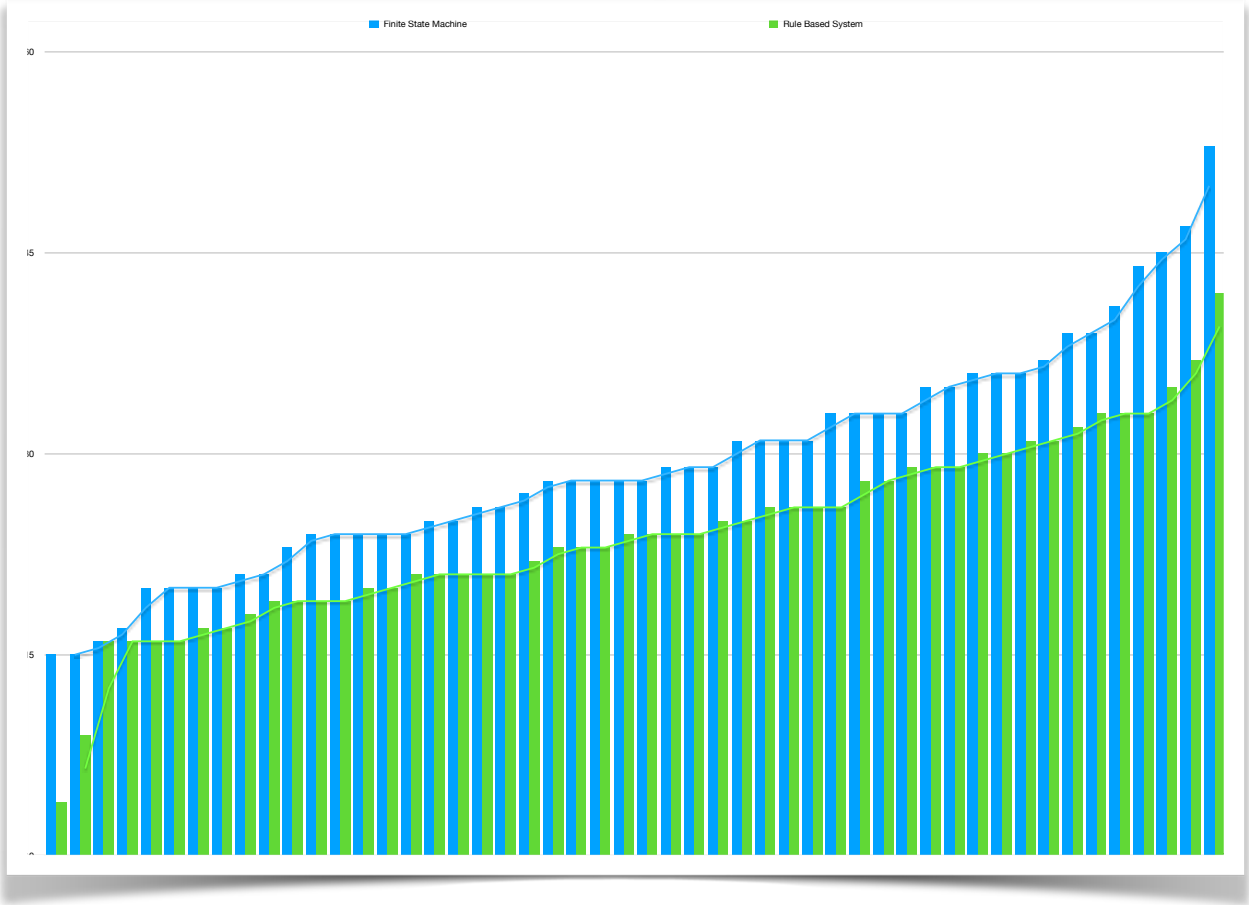


Figure 16: Graph comparison of the rule based system and the finite state machine that shows spread of data with moving average line.

As shown from this conclusion the finite state machine snake comes out on top despite hitting the wall in 44% of the test runs versus 0% for the rule based system. On average the FSM lasted 5.24 moves longer than the RBS so is the better implemented solution. If more time was spent on the project hopefully the wall hitting issue would have been resolved.

References

[1] Acornley, C., 2022. Lecture - Finite State Machine Code. [video] Available at: <https://mylearningspace.abertay.ac.uk/d2l/le/content/22525/viewContent/479121/View> [Accessed 24 March 2022].

[2] Acornley, C., 2022. Lecture - Rule Based Systems. [video] Available at: <https://mylearningspace.abertay.ac.uk/d2l/le/content/22525/viewContent/479123/View> [Accessed 24 March 2022].

[3] WeAreBrain Blog. 2022. Rule-based AI vs machine learning: what's the difference? - WeAreBrain Blog. [online] Available at: <https://wearebrain.com/blog/ai-data-science/rule-based-ai-vs-machine-learning-whats-the-difference/> [Accessed 26 March 2022].

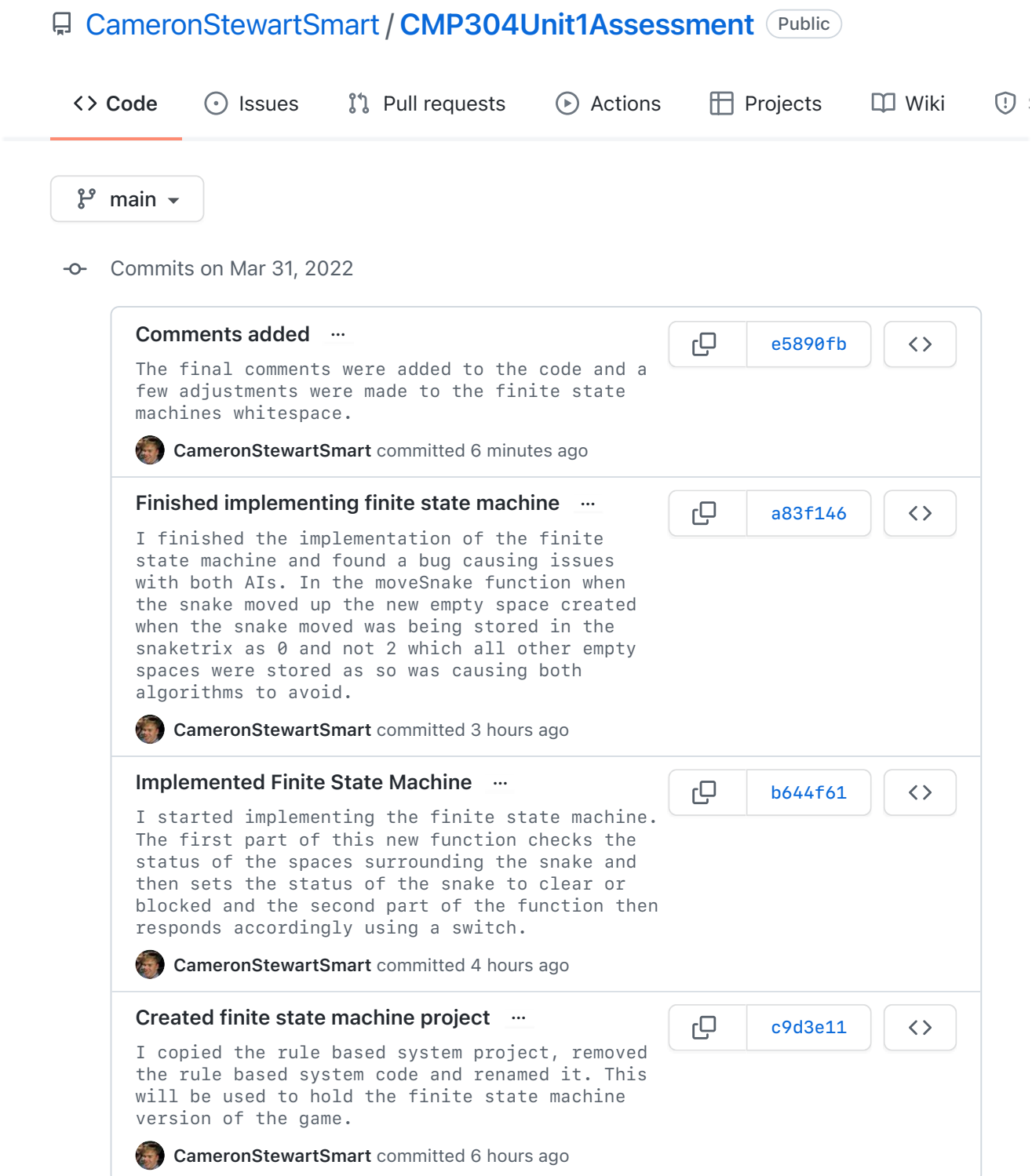
[4] En.wikipedia.org. 2022. Finite-state machine - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Finite-state_machine> [Accessed 27 March 2022].

[5] En.wikipedia.org. 2022. Rule-based system - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Rule-based_system> [Accessed 27 March 2022].

Appendix

GitHub Commits

GitHub Link



Commits on Mar 28, 2022

Implemented RuleBased AI Algorithm ···

I added the initial rule based algorithm in the form of a new function that can be called to return the move direction to the moveSnake function.

6479de5

 **CameronStewartSmart** committed 3 days ago

Cloned base project ···

I cloned the base project into the folder SnakeRuleBased to be adapted to use the rule based AI.

e3619d1

 **CameronStewartSmart** committed 3 days ago

Added comments ···

Added all comments to the files to explain choices I made and the logic behind the code.

a6f8ce0

 **CameronStewartSmart** committed 3 days ago

Finished base gane ···

Added in the endScreen function that displays final score and end message. The score and move count is also calculated properly now and I updated the game to only have one fruit per grid. I hope this will make it easier for the AI to progress through the game.

2d5ee59

 **CameronStewartSmart** committed 3 days ago

Commits on Mar 25, 2022

Added awareness ···

The game now knows if the snake has touched the edge, fruit or itself and knows to react accordingly. The game is almost ready for AI's to be added to play it but is having issues with displaying the end screen. The clearScreen() function does not work at all.

83be85d

 **CameronStewartSmart** committed 6 days ago

Added snake movement ···

The snake can now move but seems to move two places at once and will stop moving unless button is held down.

64e8e16

 **CameronStewartSmart** committed 7 days ago

Commits on Mar 24, 2022

Fixed crash bug ···

My laptop decided to hibernate and close all applications which corrupted game.h and the program database. These have been recreated.

 **CameronStewartSmart** committed 7 days ago

f815628

Implemented structure to store snake body ···

There is now a deque of structs that hold coordinates used for the body of the snake.

 **CameronStewartSmart** committed 7 days ago

31b1680

Implemented grid initialiser ···

The functions to initialise the grid including the starting position of the snake were added.

 **CameronStewartSmart** committed 7 days ago

3e98194

Setup project files ···

Created all necessary files for project and set up any includes needed to display text on the command line.

 **CameronStewartSmart** committed 7 days ago

6e050a7

Initial commit

 **CameronStewartSmart** committed 7 days ago

Verified

54479e2

Newer Older

Result Data

Rule Based System

Score	Moves	End
4	24	Snake ate itself
9	83	Snake ate itself
16	160	Snake ate itself
16	125	Snake ate itself
16	124	Snake ate itself
16	175	Snake ate itself
17	182	Snake ate itself
17	147	Snake ate itself
18	203	Snake ate itself
19	183	Snake ate itself
19	176	Snake ate itself
19	242	Snake ate itself
19	200	Snake ate itself
20	224	Snake ate itself
20	246	Snake ate itself
21	268	Snake ate itself
21	219	Snake ate itself
21	222	Snake ate itself
21	207	Snake ate itself
21	266	Snake ate itself
22	271	Snake ate itself
23	307	Snake ate itself
23	233	Snake ate itself
23	235	Snake ate itself
24	266	Snake ate itself
24	313	Snake ate itself
24	307	Snake ate itself
24	366	Snake ate itself
25	227	Snake ate itself
25	348	Snake ate itself
26	289	Snake ate itself
26	271	Snake ate itself
26	250	Snake ate itself
26	329	Snake ate itself
28	219	Snake ate itself
28	323	Snake ate itself
29	358	Snake ate itself
29	395	Snake ate itself
29	382	Snake ate itself
30	325	Snake ate itself
30	407	Snake ate itself
31	338	Snake ate itself
31	388	Snake ate itself
32	317	Snake ate itself
33	337	Snake ate itself
33	460	Snake ate itself
33	406	Snake ate itself
35	415	Snake ate itself
37	450	Snake ate itself
42	431	Snake ate itself

Finite State Machine

Score	Moves	End
15	116	Snake ate itself
15	204	Snake ate itself
16	187	Snake hit wall
17	232	Snake hit wall
20	222	Snake hit wall
20	213	Snake ate itself
20	183	Snake hit wall
20	283	Snake ate itself
21	198	Snake ate itself
21	296	Snake hit wall
23	214	Snake ate itself
24	251	Snake hit wall
24	267	Snake hit wall
24	262	Snake hit wall
24	284	Snake ate itself
24	271	Snake hit wall
25	325	Snake ate itself
25	281	Snake ate itself
26	259	Snake ate itself
26	606	Snake hit wall
27	272	Snake ate itself
28	344	Snake ate itself
28	342	Snake hit wall
28	306	Snake ate itself
28	701	Snake hit wall
28	367	Snake ate itself
29	301	Snake hit wall
29	248	Snake hit wall
29	403	Snake ate itself
31	336	Snake ate itself
31	496	Snake hit wall
31	449	Snake ate itself
31	192	Snake ate itself
33	332	Snake hit wall
33	387	Snake hit wall
33	570	Snake hit wall
33	589	Snake hit wall
35	419	Snake hit wall
35	434	Snake ate itself
36	555	Snake hit wall
36	417	Snake ate itself
36	522	Snake hit wall
37	443	Snake ate itself
39	493	Snake hit wall
39	409	Snake ate itself
41	520	Snake hit wall
44	579	Snake ate itself
45	808	Snake ate itself
47	410	Snake ate itself
53	653	Snake hit wall