University of South Wales
Prifysgol De Cymru

Faculty Of Computing and Engineering Sciences

Assessment Cover Sheet 2020-21

Module Code:	Module Title:	Module Team:
CS3S667	Artificial Intelligence for Game Developers	Mike Reddy
Assessment Title:		**Assessment No.:**
(Re)Creating the Classics		1
Date Set:	**Submission Date:**	**Return Date:**
28-Sep-2020 23:59	06-Nov-2020 23:59	04-Dec-2020 23:59

IT IS YOUR RESPONSIBILITY TO KEEP RECORDS OF ALL WORK SUBMITTED.

Marking and Assessment

This assignment will be marked out of **100%**.

This assignment contributes to **50%** of the total module marks.

Learning Outcomes to be assessed

As specified in the validated module descriptor https://icis.southwales.ac.uk

- 1) Understand the theory that underpins, and the pragmatic difficulties associated with, thedevelopment of a working AI game system
- 2) Evaluate the relative effectiveness of different approaches to AI for a given problem

Awarded mark is only provisional: subject to change and / or confirmation by the Assessment Board.

Assessment Task

Assessment Task:

Over the progress of the autumn term you are required to implement a number of core AI algorithms as exercises; these are A* search and Finite State Machines (FSMs). You are required to select an appropriate technique(s) to achieve one of the three [3] scenarios listed below. You should then complete a technical report which discusses your selection. This technical report will:

1) Provide a description and discussion of each of the techniques you have selected. You should ensure this discussion includes:

   (a) Relevant evaluation of the technique in terms of tasks to which it is suited, or is perhaps definitely not suited, this should be placed in the context of the given scenarios.

   (b) Factors you have identified which might influence its selection for a task (such as complexity/computational demands).

2) Include annotated source code (assuming the implementations are minimal; where a full implementation is included consider providing the code in an appendix with annotated snippets being used in the body of the report to illustrate operation). The intention is to demonstrate your successful implementation of the abstract concepts involved.

3) Supply evidence of the operation of the implementation, such as experimental results and screen shots (you do not need to implement the scenario, merely demonstrate the technique).

4) Clearly indicates how the strengths of the technique in general terms (point 1 above) apply to the specific scenarios. This will require you to present your understanding of the requirements of the scenario.

5) Documents any implementation details specific to the scenario, such as data representation.

6) Use appropriate diagrams and figures to support your explanation.

Scenarios: The FREE Python book "Code the Classics 1" (available from https://wireframe.raspberrypi.org/books/code-the-classics1 - look for the free option) and source code (available from https://github.com/Wireframe-Magazine/Code-the-Classics via GitHub) implements simple games inspired by classic arcade games - Pong, Frogger, BubbleBobble. Centipede, and Sensible Soccer - using Python and the PyGame library. All source code for these games is available. Pong would be too simple to do as more than a formative tutorial exercise, but the following scenarios are available for you to creata a player AI to create:

1) Bunny (aka Frogger, or more recently Crossy Road) - create an AI to go the furthest

2) Myriapod (aka Centipede) - create an AI to get the high score <- was meant to be Caverns :-(

2b) Caverns (aka Bubble Bobble) - create an AI to get the high score <- Added in, but not removed Myriapod

3) Soccer (aka Sensible Soccer) - create an AI to beat (or not be too badly beaten by) thepre-existing opponent AI

You will need to install Python version 3.8.5 https://www.python.org/downloads/ and PyGame https://www.pygame.org/wiki/GettingStarted

Group working: While this is ultimately an individual assignment, by selecting specific scenarios from above, you will be self-selecting a group of like-minded (similarly cursed) colleagues. There scope for group working will consist of formative discussion as to how to approach considering an AI to solve the games from a player's point of view. Implementation details must be individually

created, but general strategies, and some "gentle competition" between the self-selected group members will be encouraged.

Notes and Hints: In the scheduled workshop sessions there will be opportunities for students to discuss and obtain formative feedback on the achievement of the workshop tasks. Students are encouraged to consider how this feedback can be used to improve their submission for this assessment. It may appear that the report requires more than the recommended word count, but you can and should try to cut to the chase on each element; avoid unnecessary detail and divergences and remember the ABC of report writing: Accuracy, Brevity and Clarity. You must however demonstrate achievement of the learning outcomes Use the Grading criteria to guide your writing.

Marking Scheme

	Fail (0/29)	Narrow Fail (30/39)	3rd Class / Pass (40/49)	Lower 2nd Class / Pass (50/59)	Upper 2nd Class / Merit (60/69)	1st Class / Distinction (70/100)
Scenario Analysis (20%)	☐ No evidence that an understanding of the characteristics of the scenario was reached ☐ The scenario was not adequately explained	☐ Some indication that the characteristics of the scenario were understood, but failing to identify these correctly ☐ The scenario is poorly understood or explained	☐ The significant aspects of the scenario are identified in the context of the assessment ☐ Investigation of the scenario may exhibit a few errors	☐ The significant characteristics of the scenario have been identified ☐ A few of the subtleties have been identified and discussed	☐ The characteristics of the scenario in the context of the assessment are identified ☐ Some of the subtleties are also identified and discussed	☐ The characteristics of the scenario have been identified and generalised ☐ The scenarios is compared to other cases and comparisons drawn in relation to the context of implementing AI solutions
Scenario Solution (40%)	☐ The selected technique is inappropriate ☐ The selected technique is described incorrectly ☐ Explanation is missing, or contains many significant errors	☐ A technique which is not wholly inappropriate is selected ☐ Explanation of the technique is lacking ☐ Explanation for the selection may have significant errors	☐ A technique is selected which is adequate though may not be the most appropriate ☐ An explanation for the selection of the technique is given ☐ The explanation of the technique is over simplified or has significant errors	☐ An appropriate technique is selected that had been supported to a degree ☐ The technique is clearly defined ☐ The explanation has few if any significant errors	☐ A technique is selected supported by a well-reasoned explanation which shows an understanding of the technique and its applicability ☐ The technique is defined with understanding of implementation issues ☐ The explanation has only minor errors	☐ A technique is selected supported by a wellreasoned explanation which shows an understanding of the technique, and places this selection into a wider context, perhaps by comparisons or consideration of practical issues ☐ The technique is well defined ☐ The explanation has no errors
Scenario Implementation (40%)	☐ No evidence that a working implementation was completed ☐ No evidence of testing has been provided	☐ Some aspects of the core functionality are in place ☐ Evidence that the technique has been implemented and tested to some degree	☐ The major aspects of the core functionality are in place ☐ Evidence that the implementation has been tested	☐ Code snippets and test evidence show a successful implementation of the core functionality of the technique ☐ Some evidence of formal testing is present	☐ Code snippets show a successful implementation which address more than just core functionality ☐ Evidence is provided of a range of tests	☐ The code snippets submitted show some sophistication ☐ Clear evidence is provided that the implementation has been tested ☐ Tests illuminate suitable applications for the technique
Global:	Scenario Analysis Scenario Solution Scenario Implementation Summary and Future Work					

Artificial Intelligence for Game Developers (16079892)

The aims of this report are to show the differences of a* algorithms and finite state machines when trying to create AI for video games. It will also go into detail about how these methods actually work along with examples of implementation of these methods into a game called “Bunner” (Based on video games like Crossy road and Frogger).

Contents

What are A* algorithms and Finite state machines?

A algorithms*

A* algorithms are modified from Dijkstra’s algorithm and one of the most used techniques in path finding. A* search works by measuring the distance or value from the result node to each other node available to use. When given a starting node, it will measure the distance/value of each node connected to the starting node one by one until all connections have been measured. The node with the smallest distance/value between the current (in this context the starting node) plus the smallest distance left to go to the end node will be moved to the top of the queue. The top node will become the current node and this process will continue until the result is finally reached. The key modification between Dijkstra and A* is the fact that the node is measured not just in distance/value from the current, but also the distance/value from the result. This means if multiple nodes had the same distance from the current node, A* will still find the absolute best result, whereas Dijkstra’s algorithm may end up taking path that has more distance/value from the result.

Example of A*

Red = start/Finish

Blue = one possible path

Green = another possible path

Black = unused paths

{1, 1, 1, 1}

{1, 2, 1, 1}

{1, 1, 2, 1}

{1, 1 1, 1}

Example of Dijkstra

Red = start/Finish

Blue = one possible path

Green = another possible path

Black = unused paths

Orange = Longer paths but still allowed through Dijkstra's algorithm. A* avoids these unnecessary extra steps.

{1, 1, 1, 1}

{1, 2, 1, 1}

{1, 1, 2, 1}

{1, 1 1, 1}

Finite State Machines

Finite state machines can be used for path finding, but can also be used in many other ways for AI, such as AI behaviours. A finite state machine works by going through a list and checking off each individual point until it comes to a point that has an argument that works. A traffic light is a great example of finite state machine. When the light is green, and no other lights are on you go. When the light is red, and no other lights are on you stop etc. These are not very complicated states, but they don't need to be. As long as the states have clear rules as to when they should and should not be used, as well as what they should do when used. Then the system will work perfectly fine.

Its worth noting, that one technique is not better than the other. Also, more than one technique can be used in the same game. In fact, there are multiple times in the "bunner" game where both techniques could be used for the same result or where one makes more sense than the other. The context of what needs to be done dictates which technique would be best as will be shown throughout this report.

"Bunner" Code explanations

Update function start

```
def update(self):
    # print(self.x)
    current_row = None
    check_row = False
    next_row = None

    logCheck = self.MOVE_DISTANCE
    for row in game.rows:
        if check_row:
            next_row = row
            break
        if row.y == self.y:
            current_row = row
            check_row = True
    rowCheck = type(next_row).__name__
```

The first set of changes are that I added "*check_row*" and "*next_row*" to the "*Bunner*" class "*update*" function "*check_row"* and "*next_row"* are used to find the row above the AI. Its does this by using the for loop shown.
First it gets "game.rows" and checks to see if the "*check_row"* Boolean is set to "True". It wont be to start so it then skips over to "*if row.y == self.y:*" which is being used to check if the "*row"* in "*game.rows*" Y position. It then checks to see if the AI's Y position is the same. If so the AI now knows that row in particular is the current row it is stood on so it sets it to "*current_row*". Now that the current row is found the code loops again but with "*check_row"* set to true. This new row must be the one above the current so it is set to "*next_row*" before breaking the loop.
Finally "*rowCheck*" is being used to find the name/type that "*next_row*" has been assigned e.g. grass, rail, road.

Next_row start

```
if next_row:
    next_state, next_obj_y_offset = next_row.check_collision(self.x)
    #grass = isinstance(current_row.index,Grass)
    #dirt = isinstance(current_row.index,Dirt)
    #water = isinstance(current_row.index,Water)
    #road = isinstance(current_row.index,Road)
    #pavement = isinstance(current_row.index,Pavement)
    #rail = isinstance(current_row.index,Rail)
    #print(next_row.index)
    if next_state == PlayerState.ALIVE:
```

Here the code is checking to see if "*next_row"* has been defined/has a value. If so it then creates the attributes "*next_state*" and "*next_obj_y_offset*". These are very useful especially "*next_state*". With this the ai can now see what state the ai will be in if it moves into the next row. These states being "ALIVE", "SPLAT", "SPLASH" and "EAGLE". It then begins and if statement checking to see if the "*next_state*" for the AI will be "ALIVE" so it wont accidently run in front of cars or into water for example.

Grass row + hedge detection

```
if rowCheck == "Grass":
    print("grass")
    self.stepOnLog = True

    if next_row.collide(self.x, 5):
        for hedge in next_row.children:

            #print('Self Y: ' + str(self.x) + ' Hedge Y: ' + str(hedge.x))
            if self.screenCheck == False:
                self.input_queue.append(0)
                self.input_queue.append(1)
                #print ("test 1")
                if self.x >= 410:
                    #print ("test 2")
                    self.screenCheck = True
                    self.input_queue.append(0)
                    self.input_queue.append(3)
                    break
                else:
                    break

            elif self.screenCheck == True:
                self.input_queue.append(0)
                self.input_queue.append(3)
                #print ("test 3")
                if self.x <= 60:
                    #print ("test 4")
                    self.cc = False
                    self.screenCheck = False
                    break
                else:
                    break
    else:
        self.input_queue.clear()
        self.input_queue.append(0)
```

First it checks to see if the row above is “Grass” if so it prints for testing purposes and sets “*self.stepOnLog*” to True. “*self.stepOnLog”* will be explained later in the water section. Next the ai checks to see if it is colliding with anything above it. If so it enters a for loop where it calls “hedge” from the children inside “*next_row*”. It then checks to see if “*self.screenCheck”* is false, if so it adds the movements “up” and “right” (0, 1) to the movement queue (“*self.input_queue”*). Next it checks to see if the AI’s x position is greater than or equal to 410 (almost the right edge of the screen) if so “*screenCheck*” gets set to True and the ai adds the movements “up” and left to the queue, it then breaks the for loop. Now that “*screenCheck”* is True the next loop in update will instead add the movements “up” and “left” before checking to see if the ai is at the left side of the screen (<=60). If this is the case “*screenCheck*” now equal false and the for-loop breaks. Since this method fills the input queue very quickly the else statement that pairs with the hedge collide check will clear the input queue and then add forward movement once.
Essentially this code will move up one and right one when it hits a hedge. When it hits the right of the screen it changes up one and right one to up one and left one which will eventually find the gap

within the maze at which point it clears the now unneeded inputs in the input queue and moves on ahead.

The maze is a great example of how both A* and Finite state machines can both do the same job to varying degrees of success. The example above is a finite state machine and comes with the drawbacks of it will always go right first. Even if the hedge gap is ever so slightly left of the AI it will first run all the way to the right and then back all the way to the left which can be time consuming. A* would be better approach in my opinion and with more time one I would have tried to implement. A* would ensure no time loss and instead the AI would pass the maze with ease and with no changing obstacles like cars or logs I think A* would have been a better choice.

Row Checks

```
elif rowCheck == "Dirt":
    self.stepOnLog = True
    #if self.stepOnDirt == False:
       # self.input_queue.append(0)
        #self.stepOnDirt = True
    self.input_queue.append(0)
    print("Dirt")
elif rowCheck == "Road":
    self.stepOnLog = True
    self.input_queue.append(0)
    print("Road")

elif rowCheck == "Pavement":
    self.stepOnLog = True
    #if self.stepOnPavement == False:
     #   self.input_queue.append(0)
      #  self.stepOnPavement = True
    self.input_queue.append(0)
    print("Pavement")

elif rowCheck == "Rail":
    self.stepOnLog = True
    self.input_queue.append(0)
    print("Rail")
```

These else if checks all do the same thing. They find the row type, “Dirt”, “Road”, “Pavement” and “Rail”. Next the set “stepOnLog” to True. They input a forward movement. Then they print their row type for testing purposes.

As you can see both “Dirt” and “Pavement” have some code commented out. This will be shown in more detail later in the report when I show some failed ideas and bugs.

Most of these row checks don’t need any real code. Dirt and Pavement are just empty, and Rail could only be improved a very small amount. Both A* and Finite work just as well here.

Water Row + Log detection

```
elif rowCheck == "Water":
        self.stepOnLog = False
        self.input_queue.clear()
        self.input_queue.append(0)
elif next_state == PlayerState.SPLASH:
    if self.stepOnLog == True:
        for log in next_row.children:
            self.input_queue.clear()
            if self.x < log.x:
                print("waterTest2")
                self.input_queue.append(1)
                break

            elif self.x > log.x:
                print("waterTest3")
                self.input_queue.append(3)
                break
```

When the AI gets to the waters edge this code tells it to find a nearby log, run towards it and then hop onto it. Starting with "*rowCheck"* the ai checks to see if the next row is water, if so, it sets "*stepOnLog*" to false. Next it clears the input queue before adding forward movement to the input queue.

"elif next_state == PlayerState.SPLASH:" is an else if not to "*row_check"* but to "*next_state == PlayerState.ALIVE*". This means that when the AI detects that the row ahead of it will cause it to enter the "SPLASH" state it will run this code. It starts by checking if "*stepOnLog*" is True at which point it finds log (child) inside "*next_row*" and enters a for loop. It then clears the input queue to help ensure each loop won't add more and more inputs until it becomes too much. The AI will then check to see whether the log is right or left of it and head in that direction one time before breaking the loop. The reason "*stepOnLog*" is used in all row checks is to ensure that this code can be turned of once it makes the first step otherwise the ai will run into the water as the log moves. However, we need to use this code for the start of every river so only the water row turns it to False whilst all other rows re-enable it back to being True.

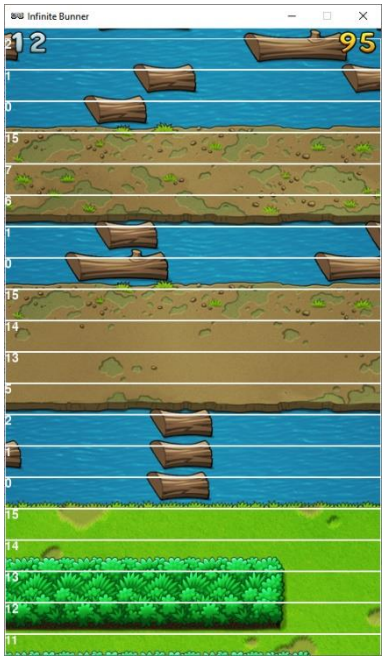
A* and finite state machines both work perfectly fine for log detection. A* would have to account for the constant updating of the log and player however which may cause some issues when first implementing however, if done correctly the ai could walk more freely on the logs whereas in my finite state machine the AI stands still and waits for the log to cross its path since walking on the log itself was much harder to detect and control. The ability to walk on the logs doesn't make a huge difference since each row of logs moves opposite each other. So, either way the AI is going to get to the log it will just be ever so slightly slower to do so.

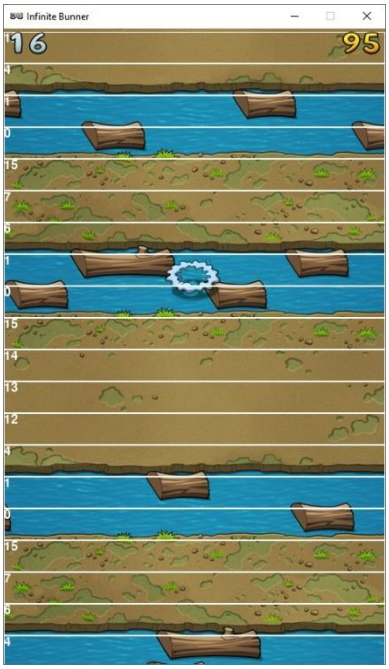
Tests, bugs and missing features

Tests

These are the high scores the AI manged to get after being run 20 times in a row. Along with some example screenshots. The letters show how the AI lost. W = water, C = car, E = eagle, T = train.

1. 15 - c
2. 13 - c
3. 38 - w
4. 14 - c
5. 94 - c
6. 20 - c
7. 4 - c
8. 42 - w
9. 16 - w
10. 1 - c
11. 10 - c
12. 1 - w
13. 45 - c
14. 12 - w
15. 7 - c
16. 67 - c
17. 1 - c
18. 11 - c
19. 23 - c
20. 49 - c



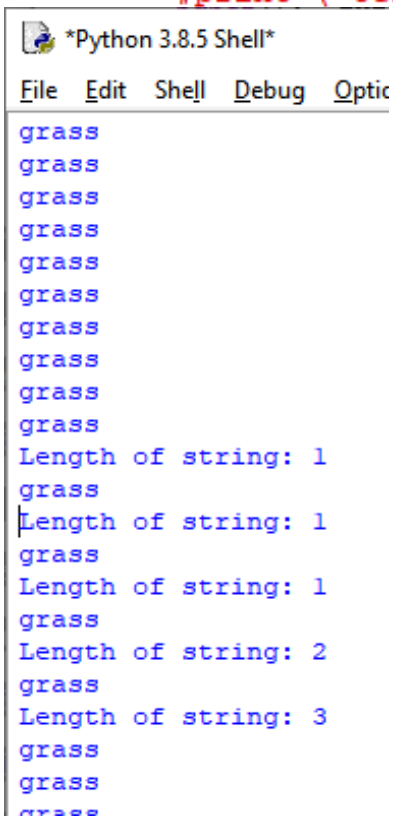


The only issues for the AI were cars, logs moving away from AI and unfortunate starts. Cars were by far the biggest obstacle due to the fact that the AI only has one tactic. Wait for the car to move, this means the AI will often get hit by a car whilst it waits for another to move out of the way. A good solution to this would be an A* algorithm that updates often so the AI can see which obstacles it needs to avoid and move accordingly. Next are the logs, specifically if they are moving away from the AI. Any other interaction with logs is perfect but the ever so slight delay between moving left/right and changing to moving up means the log will float just far enough that the AI misses. This could be solved by extending the logs collision or narrowing the AI's collision with logs to force it to move even closer to the centre before leaping. Slowing the speed of the log's movement could also help since the AI would have more time to spare before jumping. Finally, are the row 1 deaths. Sometimes the AI will move into a car or lake the second it spawns. It's often down to unfortunate timing as a log spawns just close enough to make the ai jump but just far enough that it will float away before it can make it. Another is cars spawning ever so slightly left or right of the ai first jump. The easiest way to fix this would be to fix the two problems already mentioned.

Print statements were the main use of testing. It allowed me an easy way to see what the values were compared to what they should be and if certain parts of the code were being reached correctly. Each row had its own print statement confirming that it had been reached and lots of tests were done inside the water and grass rows for hedge and log detection. Prints were a very quick and easy way to know if the code was behaving as it should and were always my first attempt at problem solving. Some examples are shown below.

```
#print('Self Y: ' + str(self.x) + ' Hedge Y: ' + str(hedge.x))
if self.screenCheck == False:
    self.input_queue.append(0)
    self.input_queue.append(1)
    #print ("test 1")
    if self.x >= 410:
        #print ("test 2")
        self.screenCheck = True
        self.input_queue.append(0)
        self.input_queue.append(3)
        break
    else:
        break

elif self.screenCheck == True:
    self.input_queue.append(0)
    self.input_queue.append(3)
    #print ("test 3")
    if self.x <= 60:
        #print ("test 4")
```




```
if self.x < log.x:
    print("waterTest2")
    self.input_queue.ap
    break

elif self.x > log.x:
    print("waterTest3")
    self.input_queue.ap
    break
```

Missing Features/bugs

As mentioned, before I was unable to implement decent code for car detection. I made some attempts with finite state machines, but a lot of issues were quickly created, and I had to stop to focus on other states such as log and hedge detection. If I had time to implement an A* algorithm I think car detection and avoidance is where it would have the most superiority over finite state machines since keeping track of so many objects going in random direction, at random speeds and in a random amount of lanes can quickly issues under finite state machines, finite amount of states. Its not impossible but it is a lot more work for something a lot less efficient than an A* algorithm that has its nodes/grid updated to match the AI and cars positions.

Another missing feature was telling the ai to halt when the train sound is played. I implemented some code to do this which will be shown down below but the issue was I didn't have time to detect which part of the rail the AI was stood on which meant whilst the AI would halt when the train sound effect was played there was a chance it would stop on the tracks themselves. I ended up taking it out because it worked better the way it had been done before. If the train was in front of the ai the ai would stop moving and considering just how fast the train appears on screen its very rare the AI is hit by hit and instead usually doesn't get the chance to cross before the train appears. At which point it just waits for the train to leave the tracks. The code was very simple. I would create a function in the "*game*" class called "*soundcheck*" and return the attribute "*trainSound*". Inside the "*Rail*" class once the train bell sound had been played it would change "*game.trainSound*" to equal true. Finally, in the "*bunner*" class, update function, "*rowCheck"* rail. It would call the "*game*" class function and if True would loop without making a movement. I had planned to extend the loop with a counter that would gradually go down each loop until I could be sure that the train must be on the tracks because then the "*next_row*" collision detection would hold the AI in place until the train had left. As I said before though I took it out due to it being less helpful than the AI already was. With a little extra time to find a way to determine which part of the rail row the AI was stood on before enacting this code and it would have worked fine. In terms of which technique would be better for this approach I would say Finite state machine is the only real choice. The train appears so fast that A* wouldn't have any real time react to it whereas by using the trains bell sound a finite state machine can prepare itself the same way a player would which in my opinion is the better choice.

```
class Rail(Row):
    def __init__(self, predecessor, index, y)
        super().__init__("rail", index, y)

        self.predecessor = predecessor

    def update(self):
        super().update()

        # Only Rail rows with index 1 have tr
        if self.index == 1:
            # Recreate the children list, exc
            self.children = [c for c in self.

            # If on-screen, and there is curr
            if self.y < game.scroll_pos+HEIGH
                # Randomly choose a direction
                dx = choice([-20, 20])
                self.children.append(Train(dx
                game.play_sound("bell")
                game.trainSound = True
                game.play_sound("train", 2)
```

```
def soundCheck(self):
        return trainSound
```

This final feature was removed due to a bug. I originally wanted the AI to go back into the centre of the screen when on either dirt or pavement rows since the AI can't be hit by anything on these rows. Due to things like logs and mazes forcing the ai to move I figured the downtime these rows have could be used to recentre. Unfortunately, the input queue would become clogged and the AI would focus on emptying the oldest parts of the queue rather than the latest instructions. This meant the AI would start to jump right to recentre and just never stop jumping right and up until it hit a car, river or train. Since it doesn't affect the gameplay too much, I chose to scrap this idea over scrapping the hedge and log detections. The code I was using is shown underneath (some of it is commented out) and had been implemented into both pavement and dirt rows. A* could technically do this, one example being, just make the centre of the dirt and pavement tiles much cheaper in value forcing the AI to take the path but I prefer the finite state machine approach. It seems a little more natural as a player we reposition themselves for an advantage rather than just create barriers that AI isn't allowed to pass.

```
elif rowCheck == "Pavement":
    self.stepOnLog = True
    if self.stepOnPavement == False:
        self.input_queue.append(0)
        self.stepOnPavement = True
    else:
        self.input_queue.append(0)
    print("Pavement")
```

```
elif rowCheck == "Dirt":
    self.stepOnLog = True
    if self.stepOnDirt == False:
        self.input_queue.append(0)
        self.stepOnDirt = True
    else:
        self.input_queue.append(0)
    print("Dirt")
```

```
if self.state == PlayerState.ALIVE:
    #if rowCheck == "Pavement":
    #    print("Pavement")
     #   if self.x < 230 or self.x > 250:
      #      if self.x < 230:
       #         self.input_queue.append(1)
        #    if self.x > 250:
         #       self.input_queue.append(3)
        #elif self.x >= 230 and self.x <= 250:
         #   self.input_queue.append(0)
            #self.stepOnPavement = False

    #if rowCheck == "Dirt":
        #if self.x < 230 or self.x > 250:
          #  if self.x < 230:
            #    self.input_queue.append(1)
           # elif self.x > 250:
              #  self.input_queue.append(3)
       # elif self.x >= 230 and self.x <= 250:
          #  self.input_queue.append(0)
          #  self.stepOnDirt = False
       # self.input_queue.clear()
```

The bug I struggled with most was most definitely the input queue receiving too many inputs. It's already been mentioned throughout the report, but it caused issues on the hedge detection, log detection and dirt/pavement recentring. The issue was always the same, the code would get stuck in loops even with break and enter hundreds of inputs before the first could be acted upon. Then the Ai would attempt to enter all those inputs one by one whilst pushing all new inputs that would help steer it in the right direction to the back of the queue. In my first attempt to fix this created this code.

```
if self.timer == 0 and len(self.input_queue) > 0:
    print ('Length of string: ' + str (len(self.input_queue)))
    #inputCheck = False
    if len(self.input_queue) > 10:
        self.input_queue.pop(0)
    else:
    # Take the next input off the queue and process it
        self.handle_input(self.input_queue.pop(0))
```

This usually just uses the first and last line of the code to place the inputs into the handle_input function but with this I attempted to check if the length was larger than 10 and if so it would start popping the queue until it had gotten below 10 (I also tried >0,>1,>2,>3). It didn't work due the queue being filled so fast all the time that it just wasn't making a dent. Next I tried to add queue.pop to wherever I thought could help inside the hedge detection but once again it did little to help due to my placements and to my original code accidently continuing the hedge code even after it had passed which caused even more queue issues. I next tried to use time.sleep but it would freeze the entire game so I gave up on it pretty quickly. Then I tried to create a counter that would only run the code every couple of loops but this tsopped my code from using two inputs easily which I needed to check the maze i.e. up and left or up and right. Thnkfully I finally came up with a solution.

```
if next_row.collide(self.x, 5):
    for hedge in next_row.children:

        #print('Self Y: ' + str(self.x) + ' Hedge Y: ' + str(hedge.x))
        if self.screenCheck == False:
            self.input_queue.append(0)
            self.input_queue.append(1)
            #print ("test 1")
            if self.x >= 410:
                #print ("test 2")
                self.screenCheck = True
                self.input_queue.append(0)
                self.input_queue.append(3)
                break
            else:
                break

        elif self.screenCheck == True:
            self.input_queue.append(0)
            self.input_queue.append(3)
            #print ("test 3")
            if self.x <= 60:
                #print ("test 4")
                self.screenCheck = False
                break
            else:
                break
else:
    self.input_queue.clear()
    self.input_queue.append(0)
```

By using this method I was able to properly check when the AI was no longer in front of the hedge by using "*next_row.collide"* and the queue issue was fixed by clearing the queue the second the AI wasn't colliding. Also the 5 in "*next_row.collide*" is used to make the hedges seem wider than they are since the AI would sometime get stuck on the very corner of the maze exit. Luckily it fixed it entirely.

References

Computerphile. “A* (A Star) Search Algorithm - Computerphile.” YouTube, 14 Feb. 2017, www.youtube.com/watch?v=ySN5Wnu88nE. Accessed 6 Nov. 2020.

“Dijkstra’s Algorithm - Computerphile.” YouTube, 3 Jan. 2017, www.youtube.com/watch?v=GazC3A4OQTE. Accessed 6 Nov. 2020.

GeeksforGeeks. “A* Search Algorithm - GeeksforGeeks.” GeeksforGeeks, 7 Sept. 2018, www.geeksforgeeks.org/a-search-algorithm/.

Google. “Python Strings | Python Education.” Google for Education, developers.google.com/edu/python/strings. Accessed 6 Nov. 2020.

Payne, Trevor. “Let’s Learn Python #20 - A* Algorithm - YouTube.” Youtube, 21 Dec. 2013, www.youtube.com/watch?v=ob4faIum4kQ. Accessed 6 Nov. 2020.

Programiz. “Python Type().” Www.Programiz.com, www.programiz.com/python-programming/methods/built-in/type. Accessed 6 Nov. 2020.

Python. “8.15. Types — Names for Built-in Types — Python 2.7.18 Documentation.” Docs.Python.org, docs.python.org/2/library/types.html. Accessed 6 Nov. 2020.