UNIVERSITY OF LIMERICK

Ms. Pac-Man Controller

CS4096 Artificial intelligence For Games

Authors:

Name:	**David Cherry**	**ID: 13056212**
Name:	**Cathaoir Agnew**	**ID: 16171659**
Name:	**Conor O'Donovan**	**ID: 9730761**

Table of Contents

Table of Figures

1. Introduction

Group projects tend to be a little trickier during pandemics. Working remotely from each other without any shared lab time for dissecting a problem, teasing out potential solutions with each other, developing a rapport within the group or gauging each other's skill sets can unearth new challenges. Discord is a VOIP and instant messaging platform with data distribution and screen sharing facilities. It is not ideal software for the task at hand, but proved a practical solution to our logistical situation.

Ms Pac-Man is a simple game. A hungry Ms Pac-Man needs to eat all the pills on a level in order to proceed to the next, while four ghosts give chase and take one of her lives if they make contact with Ms Pac-Man. There are four Power Pills, each located in a corner of the map, which when eaten by Ms Pac-Man render the ghosts edible, whereupon they begin to flee. You gain 10 points for each regular pill eaten, 50 points for each power pill and 200 points for the first ghost eaten (a score which doubles for each ghost eaten within the same power pill duration). While this all may appear quite straightforward, beneath the surface, the random element of the gameplay makes Ms Pac-Man a tricky controller to automate and create the ultimate pill and ghost eating machine.

The purpose of this project was to develop an AI controller for the Ms Pac-Man character. Our approach to the task was to investigate the existing code, understand how the agent and the four ghosts operated, how each reacted to changes in states in gameplay and establish how we could manipulate these behaviours and the data provided by the code during runtime in order to develop and implement our strategies.The Ms Pac-Man code we are working with for this project utilises individual classes as the behaviour controllers for the ghosts and Pac-Man agents. We started by exploring the "StarterPacMan" and "StarterGhosts" classes which give us an indication of how AI Ms Pac-Man and the ghosts behave. For every frame of gameplay these classes assess the state of the game, and based on certain conditions, will execute a behaviour for the characters. The classes call upon various methods and functions in order to gain information about the game, and utilises this information to determine the behaviour of the agents. These functions include, but are not limited to, the ability to determine the distance towards a target in euclidean distance (straight line distance), Manhattan distance (essentially the change in x coordinates plus the change in y coordinates) and path distance (the actual length of the path through the maze to the target), as well as other stats of interest like the number of Power Pills or regular pills left on the map.

Having explored the code and a number of techniques which we could use to implement our strategies, all of which is discussed in the following section, we decided upon a hierarchical set of strategies which our agent would employ based on the conditions of the game as they presented themselves. Our intent was not necessarily to gain an extremely high score, but rather to develop an agent that would consistently perform well using strategies we would be able to fully implement within the timeframe allotted for the project. We reasoned that one outrageously high score could be achieved by fluke, but consistent or improved average scores would point to intelligent implementation of the code and a validation of our chosen strategies

2. Design

In this section, we will briefly discuss the various techniques that were considered for this project as well as the characteristic strengths and weaknesses of these techniques. From these strengths and weaknesses the chosen method will be highlighted and discussed in further detail relevant to the project itself.

2.1 Potential Techniques

The decision of which technique to use in the controller is an important one as it will dictate limits of the capabilities of the controller within the game. A smarter control system would perform better over all, but would also require a greater skill set from the programmer, generally speaking. Each technique also has their own strengths and weaknesses that will be further investigated and discussed here.

2.1.1 Monte Carlo Tree Search

The Monte Carlos Tree Search (MCTS) is a heuristic search algorithm that is used in decision making. The core loop of the algorithm operation can be broken down to four steps; Selection, Expansion, Simulation and finally Backpropagation. In the selection phase a node is decided upon which to be expanded, usually a node with unexpanded children nodes. In the expansion phase the selected node is expanded to represent any valid move from the game position, defined by the parent node. In the simulation phase the effects of taking random moves are calculated in order to produce a value estimate of that move. Finally in the Backpropagation phase, the value is added to the terminal node as well as all parent nodes until it reaches the “root” of the tree. This in turn finds a best set of moves for a given set of frames based on allowed depth of the tree.

One of the strengths of this method is that in-depth analysis of the game isn’t strictly necessary as long as you can supply the code with allowable moves and game-end conditions. Also, the monte carlos tree tends to develop asymmetrically, favouring branches with more promising results to further expand. This technique can also be halted at any time and can simply return its “best bet” given the time it was allowed to calculate.

In terms of weaknesses, the monte carlos tree search method will tend to overlook more risky moves in favour of more initially promising results. This can lead to a local minima regression style problem, that is to say, the move it chooses might be better in the short run, but overall worse in the long run.

2.1.2 Finite State Machine

The Finite State Machine (FSM) is an abstract representation of an interconnected set of states which are switched between based on the change of conditions outlined. Each state has a set of actions, objectives, events or behaviour associated with it and is only switched with another state if the condition for the transition into another state is fulfilled. An FSM is defined by three main components; States, Transitions and Actions. The State stores information about the task, for example, the flee state or the hunt state. The Transitions are the specific conditions required to move from one state to another. The Actions are the behaviour that is followed while in a specific state.

FSM are generally speaking simple to design, implement, visualize and debug. This is because states can be individually evaluated and edited without changing other states. This can lead to situations where certain states can never be reached, however that is down to the design process of the programmer rather than a fault of FSM in general.

FSM’s can be extremely difficult to design on a large scale however and can commonly become what is commonly known as a spaghetti state machine. FSM’s are also extremely inflexible and undynamic, that is to say they will only do the actions listed when the specified conditions are met for each state. There is no dynamic response to unpredicted behaviours. Every scenario must be explicitly coded for. FSM’s can also only be in one state at a time and as a result of this, behaviour at any given moment is entirely dominated by the state that is currently active. An example of this in Ms Pac-Man is that when fleeing from a ghost and given a choice of an empty path or a path full of pills, the choice to go down the path full of pills is given no extra weight in the decision process. The actions at that time are solely based on trying to escape the ghost.

2.1.3 Artificial Neural Network

Artificial Neural Networks were briefly considered for this project but quickly dismissed due to the complexity of initiating one. The advantage of an artificial neural network is that when given enough training time, the controller could perform exceptionally well at Ms Pac-Man. This advantage is completely offset by the disadvantages associated with implementing one. These disadvantages include the complexity of initialising it as previously stated as well as the training and tweaking time required to attain a performing controller.

2.2 Chosen Technique, FSM

The controller technique chosen for this project was the Finite State Machine method. The reason we opted for this method is that it is relatively simple to design and code compared to the other methods. Ms Pac-Man itself also has a limited number of variables to contend with, thus requiring fewer states to be written, covering almost every scenario. For these reasons the weaknesses associated with FSM's are limited in regards to Ms Pac-Man, as the number of states would be low and thus avoid the spaghetti state machine mentioned before. The main goal of this project is to improve upon the existing sample controllers which were provided with the code. The key performance indicators for advancement were measured using three main parameters which indicated improvement. These parameters were the final score, the time survived and the level reached. The Diagram below (Figure 1) shows the outline of the states which were designed to attain this improvement. This diagram was achieved as the culmination of test code and observation of the visual game to determine improvements.

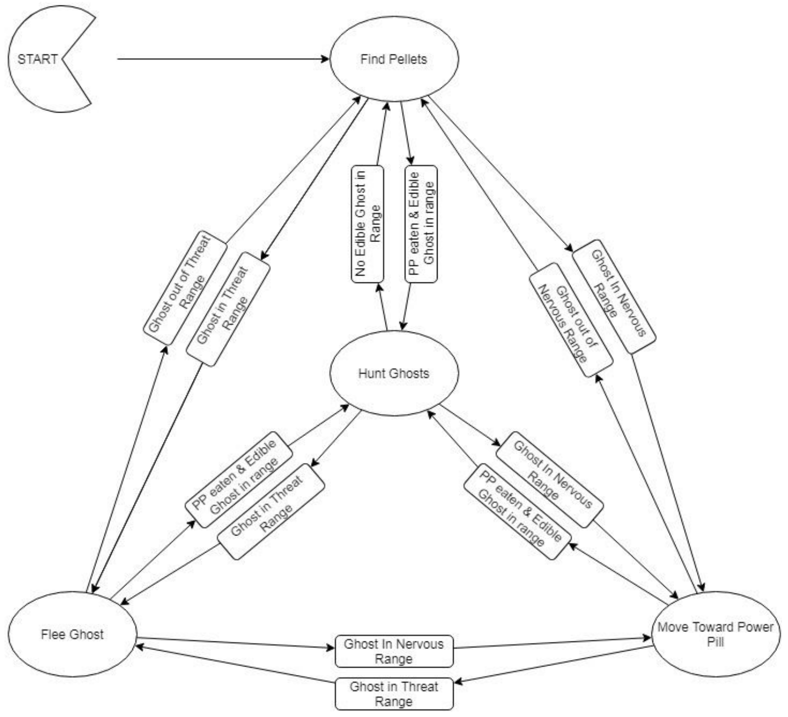

Figure 1: Finite State Machine Flow Diagram.

2.3 Details of States Utilized

The various states that were defined are detailed below along with the conditions required to transition into them along with the behaviour each will output.

2.3.1 Flee State

In this state, the Pac-Man agent will flee from the nearest ghost. This state is transitioned to when a non-edible ghost enters the immediate vicinity of the Pac-Man agent, approximately two or three blocks of space. When this state is active, the Pac-Man agent will locate the nearest ghost, and find the next move that will move it away from that ghost. Due to the small radius of this state's conditional transition, this state is only active when a ghost has nearly caught up with the Pac-Man agent. This allows the agent to take more risky paths and has resulted in a higher overall score during testing than a larger flee radius.

2.3.2 Nervous State

In this state, the Pac-Man agent will locate the nearest power pill, and begin to move towards it. This state is transitioned to when a non-edible ghost is within a middle sized radius of the Pac-Man agent, and an uneaten power pill remains still in the maze. This state cannot be active if there are no more power pills available on the current level. The purpose of this state is to ensure with relative confidence that the Pac-Man agent has access to a power pill while they are available if the Pac-Man agent is beginning to be threatened by the ghosts. It also allows the agent free reign of the maze while the ghosts are edible which effectively allows pacman to escape from being cornered.

2.3.3 Hunt State

In this state the Pac-Man agent will move toward the nearest edible ghost. This state is transitioned to when an edible ghost is within a very small hunt radius, approximately one or two blocks of space. This radius was chosen to allow the Pac-Man agent to turn and eat edible ghosts that were very close, but not necessarily along the path to the nearest regular pill. It was found during testing that the larger the hunt radius was set, the further the Pac-Man agent had to move in order to catch up with the edible ghost. This often resulted in pacman catching up to the ghost too late and effectively caused the Pac-Man agent to get itself cornered by the now non-edible ghosts.

2.3.4 Find Pills State

This state is the default state, where there are no threats to pacman in its immediate or mid range vicinity. Under this behaviour, the Pac-Man agent will locate the nearest uneaten pill and move towards it. This state is transitioned to when the conditions for no other state are met.

3. Implementation

In this section of the report we will discuss the technical aspects involved in generating functional code to operate as a successfully working Ms. Pacman controller. This includes general details regarding the I.D.E. used, how the group work was performed, what and how consensus decisions were reached, as well as a more detailed view of the code in general.

3.2 Software

The I.D.E. that was used to run the provided Ms. Pacman code was Eclipse, version 4.17.0. Eclipse was selected as it was the software recommended by the authors of the project. The Java version utilised by Eclipse was version 14.0.2.

3.3 Design Decisions

The decision to attempt an FSM style controller was made by the group after weighing the various pros and cons of each technique. It was found that the relative simplicity of a FSM would lend itself to a work from home environment by allowing each state to be worked upon individually, allowing a new state to be easily added to the code in later stages of development, or removed if necessary without fundamentally breaking the functionality of the rest of the code. The method the group chose was to individually work on a controller each and later compare the best bits from each controller and utilise the code pieces which performed best for an overlapping function.

An example of this style was seen at early stages when the group had one controller file working from java case-switch style statements and another controller working from if-else style statements. It was found that due to the varying skill levels within the group, it was better to use the if-else statements due to ease of readability.

3.4 Code Details

The majority of the code was tested as it was written, by running the various state functions and observing not only the chosen key performance indicators (score, time and level), but by also observing the behaviour of the Pac-Man agent during gameplay. This method often led to new ideas on how to deal with specific circumstances and new functions coded in general terms to apply to those specific circumstances. In the sections below we will examine the conditions required to transition into each state and the actions that those states produced more closely. We will also look at some visual aid code that was included for observation purposes.

3.4.1 Range Line code

This code is not a State that affected gameplay like the rest of the code mentioned below. It was specifically written to help identify instances when the Pac-Man agent transitioned from one state to another, during gameplay. Because the states were directly affected by the range of the nearest ghost, by drawing lines from Ms Pac-Man to the ghosts and changing the color of the line based on three preset ranges, the observer was able to distinguish which state the agent was currently acting on for any given frame. The lines were green if the ghosts were too far away to be of any concern for Ms Pac-Man, orange if the ghosts entered the “nervous” range and red if the ghosts were within the threat range which caused Ms Pac-Man to flee from the ghost directly.

3.4.2 Flee Ghosts Code

This section of code causes Ms Pac-Man to immediately flee from the nearest ghost. The Transition condition requirements were as follows:

- The ghost was not edible.
- The ghost was not in the lair.
- The ghost was within the minimum distance of the Pac-Man agent

The action carried out in this state was to move away from the nearest ghost, with no other considerations for direction as escape takes priority. The minimum distance for this code was lowered compared to the starterPacMan controller in order to allow Ms Pac-Man to take more risky paths, getting closer to the ghosts in the process. For the ghost conditions, each ghost agent was checked individually.

3.4.3 Nervous Code

This section of code caused Ms Pac-Man to move toward a Power pill if one was available on the map and if a ghost was within a larger sized radius than the previously mentioned minimum distance. The Transition condition requirements were as follows:

- The ghost was not edible.
- The ghost was not in the lair.
- There are uneaten power pills available on the map.
- The ghost was within the large radius.

The action carried out in this state was to move toward the nearest power pill. This served multiple functions. First, it began Ms Pac-Man’s move away from ghosts, but still with a direction in mind. It helped avoid getting cornered by two ghosts in the early stages of the game, particularly in Map B. It allowed Ms Pac-Man to eat the targeted power pill and thus move freely for a set amount of time while the ghosts were edible. Finally it helped Ms Pac-Man take one quarter of the map at a time while the ghosts fled to the opposite corners, thus drastically increasing the survival rate of Ms Pac-Man. For the ghost conditions, each ghost agent was checked individually.

3.4.4 Hunt Ghosts Code

This section of code causes Ms Pac-Man to chase an edible ghost that is nearby. The Transition condition requirements were as follows:

- The ghost was edible
- The ghost was within a hunt radius

The action carried out in this state was to move towards the nearest edible ghost. Initially this code was not included as we wanted to ignore the ghosts entirely while they were in the frightened state and allow Ms Pac-Man to focus on eating the remaining pills on the map. This was because it was observed that if Ms Pac-Man chased an edible ghost halfway across the game map, it would often end up being surrounded and caught by the ghosts once the edible timer had ended. However it was also observed during testing that oftentimes an edible ghost would pass extremely close to Ms. Pacman, and that if Ms. Pacman had turned to eat that ghost, it would not have lost much time eating the remaining pills on the map but would have increased the level score by quite a large margin. To accommodate this the hunt radius was set to be extremely low, and effectively meant that Ms Pac-Man would only turn to pursue ghosts that were extremely close and not on the path Ms Pac-Man had been following.

3.4.5 Find Pills Code

This section of code caused Ms Pac-Man to move towards the nearest uneaten regular pill. This was the default behaviour for the controller and as such had no explicit conditions. It was Transitioned into if no other State Transition conditions were met. This effectively meant that the following acted as the conditions for Transitioning into this state;

- No non-edible ghost was inside the minimum distance.
- No non-edible ghost was inside the large radius distance while power pills were present.
- No edible ghost was inside the hunt distance.

The action carried out in this state was to move towards the nearest regular pill. The inclusion of this default behaviour meant that Ms Pac-Man was always in a State which dictated some behaviour. Because the game moved onto the next map when all the pills were eaten, this state always had a target destination for the action to act on as at least one pill is present on every active map..

3.4.6 Ambush Code (unused)

This section of code should have caused Ms Pac-Man to “idle” near a power pill until a ghost came in range. It was ultimately commented out from our final code as it would not function as we wanted it to. The Transition condition requirements were as follows:

- There are uneaten power pills available on the map.
- A power pill was within a small radius.
- No non-edible ghost was inside a large radius.

The action carried out in this state was to move away from the power pill. The intended effect of this code was to save the power pills until ghosts were nearby before eating the power pill. This would increase the probability of eating edible ghosts since they were nearer, thus increasing the average score. What actually happened from this code was that Ms Pac-Man would get stuck in a loop where she would move away from the power pill until it was outside the small radius and then turn around and attempt to go for the nearest regular pill which was usually at the far side of the power pill. And because the ghosts fled when Ms Pac-Man was within a distance of 15 from a power pill, they would stay away from Ms. Pacman while she was stuck in this loop. For this reason, this particular piece of code was not used.

3.4.7 Executor file changes

The Executor file was the main file that was run to operate the code. From this file the game could be run visually once or “behind the scenes” multiple times. The first alteration to this file was to change the numTrails variable which controlled the number of games run for the runExperiment function from twenty to one thousand games. This was done to specifically ensure the statistical significance of our results, and that the average score, time and levels we were observing were closer to the true average expected values for the game.

The second modification to this file was to the runExperiments function itself. This alteration was for simple data gathering. Originally the function would only return the average score of the number of games run. After the alterations, the average score, high score, low score, time survived and level reached were returned. This data was copied to an excel file and that file was used to import the data to a python notebook for further analysis utilising pandas dataframes.

3.5 Design to Implementation

Our original design process involved a lot of trial and error, seeing what worked and what did not. An example of this was the ambush code. If we could have written that code to a functional state it would have been added to the flow diagram, but unfortunately we could not. The overall process however did conform nicely to a handful of clearly defined states that the controller would act upon given the specific transition conditions for each state. One note that is important to reference here is that any state could transition to any other state depending on the conditions in game. At no point was a state “locked out” by any other until a tertiary state was reached first.

4. Analysis

Each of the example Ms Pac Man controllers, along with our custom controller, were run for 1000 experiments using the runExperiment method. We opted to use 1000 runs to ensure statistical significance from the recorded results. We decided to record and use 3 performance indicators from the Ms Pac-Man game to get a feel for how each controller performed. The 3 performance indicators we determined would indicate progress were score, time survived and level reached. It is important to note that no single performance indicator can demonstrate how well each controller is performing. An example here will portray the ambiguity - a controller scored 5000 points but only made it to level 1 vs a controller who scored 3000 but made it to level 3. Which controller would we consider to have performed better? We adopted a simple solution. As our code does not exploit any of the Starter Ghosts traits, such as hovering around a powerpill forcing the ghosts to flee and thus artificially boosting the time survived metric, we simply decided to use all 3 performance indicators, such that the higher the performance indicator the better performing the controller was.

4.1 Score

When considering score, we opted to use the median as our metric of interest. The below density plot of the scores from the custom controller shows that the data from this controller has some slight skewness to the right. So median will be used, as outliers may distort the mean and may lead to misleading results. The median will give us a more accurate reflection of the average score per run. Standard deviation is just the square root of the variance. The Standard deviation tells you how spread out the data is, and is used for computing confidence intervals.

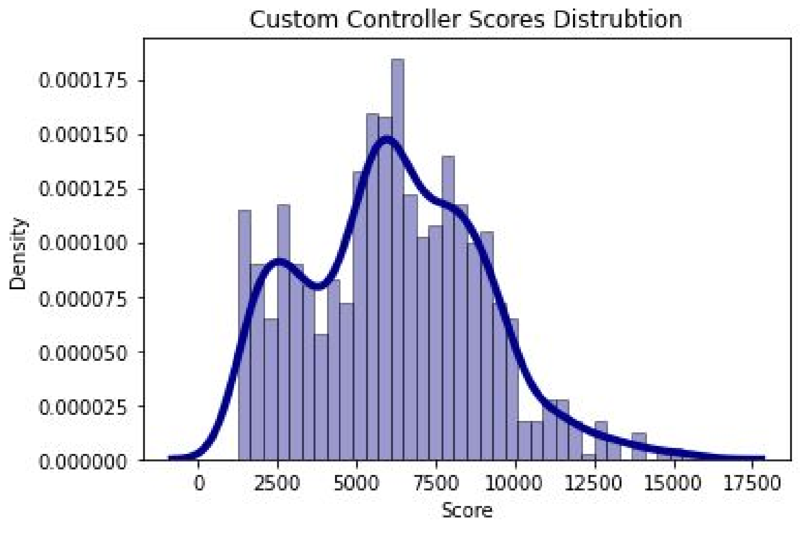

Figure 2: Custom Controller Density Plot of Scores.

Below is a table of the 1000 experiment results for each controller. The custom controller had the best performance on average of each of the 5 controllers. The custom controller had a max score of 15700 and a median score of 6130 [5958.53, 6301.46]. As we programmed the controller to only eat ghosts within a short radius, some of the variance in scores may be explained by the Pac-Man agent fleeing from the 4 close ghosts, eating a powerpill and then eating these now rounded up edible ghosts. However, on average our controller outperformed the other 4 example controllers.

Score

Controller	**Min**	**Max**	**Std Dev.**	**Median 95% CI**	**Mean 95% CI**
Custom	1270	15700	2766.47	6130 [5958.53, 6301.46]	6182.88 [6011.41, 6354.34]
Starter	920	9840	1686.31	2670[2565.48, 2774.51]	3328.08 [3223.56, 3432.59]
Random	50	540	69.23	180 [175.70, 184.29]	195.05 [190.75, 199.34]
Random Non Reverse	210	4480	570.59	1050 [1014.63, 1085.36]	1178.41 [1143.04, 1213.77]
Nearest Pill	1350	7230	570.59	2080 [2044.58, 2115.41]	2376.58 [2341.16, 2411.99]
N = 1000					

Figure 3: Statistics of Score per Controller Table.

The below box plot graphically illustrates the above information. We can see our custom controller has the maximum and median score of each of the 5 controllers.

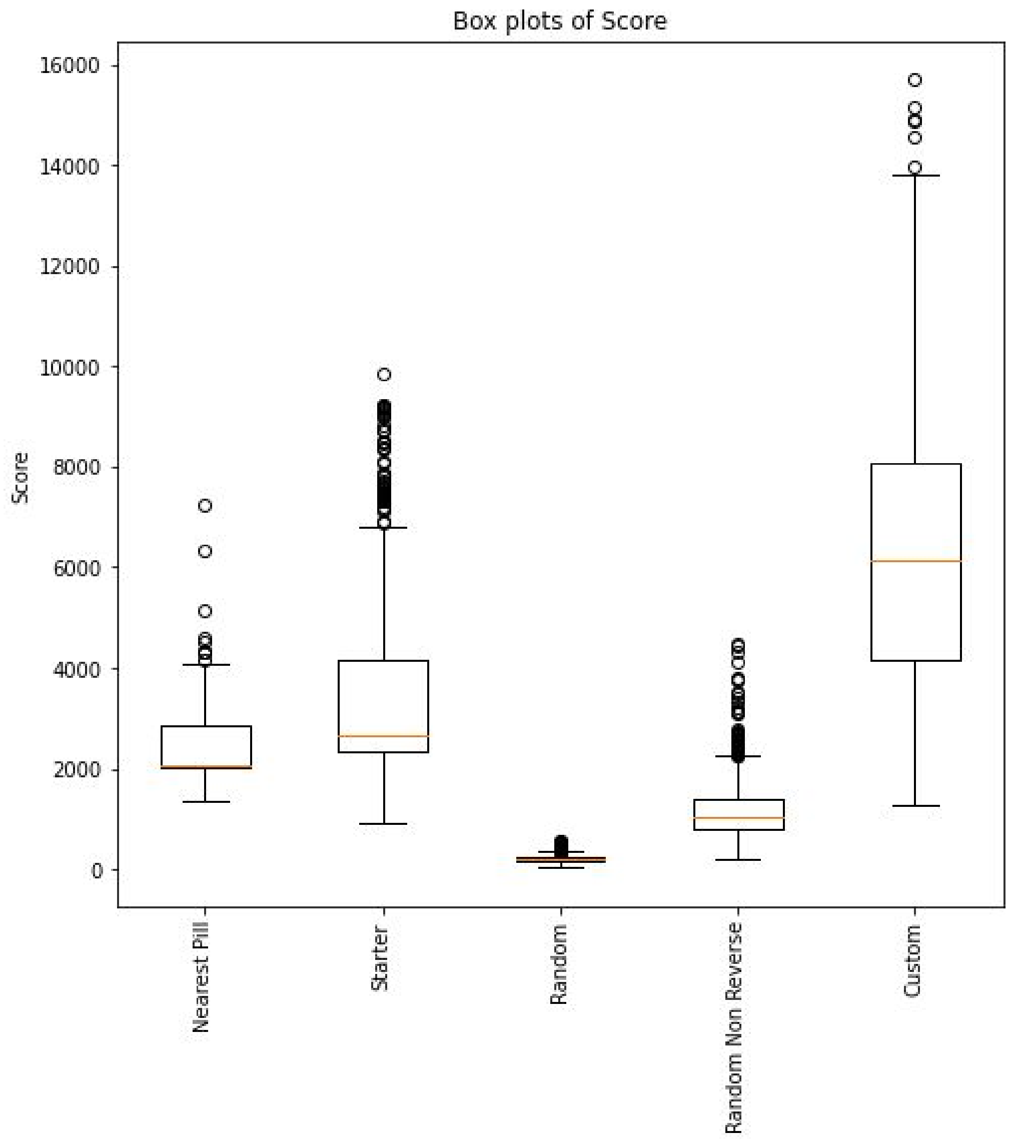

Figure 4: Box Plot of Scores per Controller.

4.2 Time Survived

When considering time, we opted to use the median as our metric of interest. Again, the density plot of the time survived below from the custom controller shows that the data has some slight skewness to the right. So median will be used, as outliers may distort the mean and may lead to misleading results. The median will give us a more accurate reflection of the average time survived per run. Standard deviation is just the square root of the variance. The Standard deviation tells you how spread out the data is, and is used for computing confidence intervals.

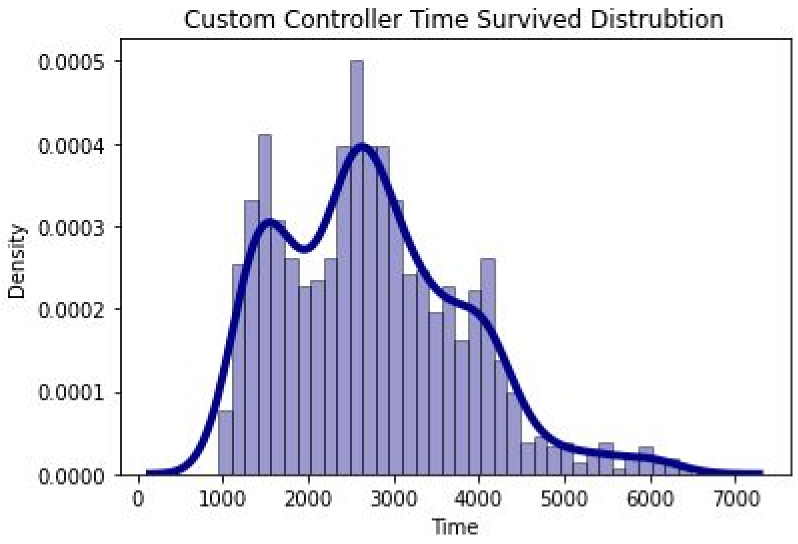

Figure 5: Custom Controller Density Plot of Time Survived.

Below is a table of the 1000 experiment results for each controller. The custom controller had the longest time survived on average for each of the 5 controllers. The custom controller had a max time survival of 6479 and a median time survived of 2637 [2571.06, 2702.93]. Due to the nature of the randomness in Ms Pac-Man, a high variance is expected in time survived.

Time Survived

Controller	**Min**	**Max**	**Std Dev.**	**Median 95% CI**	**Mean 95% CI**
Custom	953	6479	1063.83	2637 [2571.06, 2702.93]	2740.73 [2674.80, 2806.67]
Starter	486	2258	256.69	804.5 [788.59, 820.40]	848.33 [832.42, 864.24]
Random	371	1102	82.36	468.0 [462.89, 473.10]	487.66 [482.55, 492.76]
Random Non Reverse	291	2859	369.96	863.0 [840.06, 885.93]	906.61 [883.68, 929.54]
Nearest Pill	712	3085	165.52	1056.0 [1045.74, 1066.25]	1134.10 [1123.84, 1144.36]
N = 1000					

Figure 6: Statistics of Time Survived per Controller Table.

The below box plot graphically illustrates the above information from the table. We can see our custom controller has the maximum and median of time survived of each of the 5 controllers.

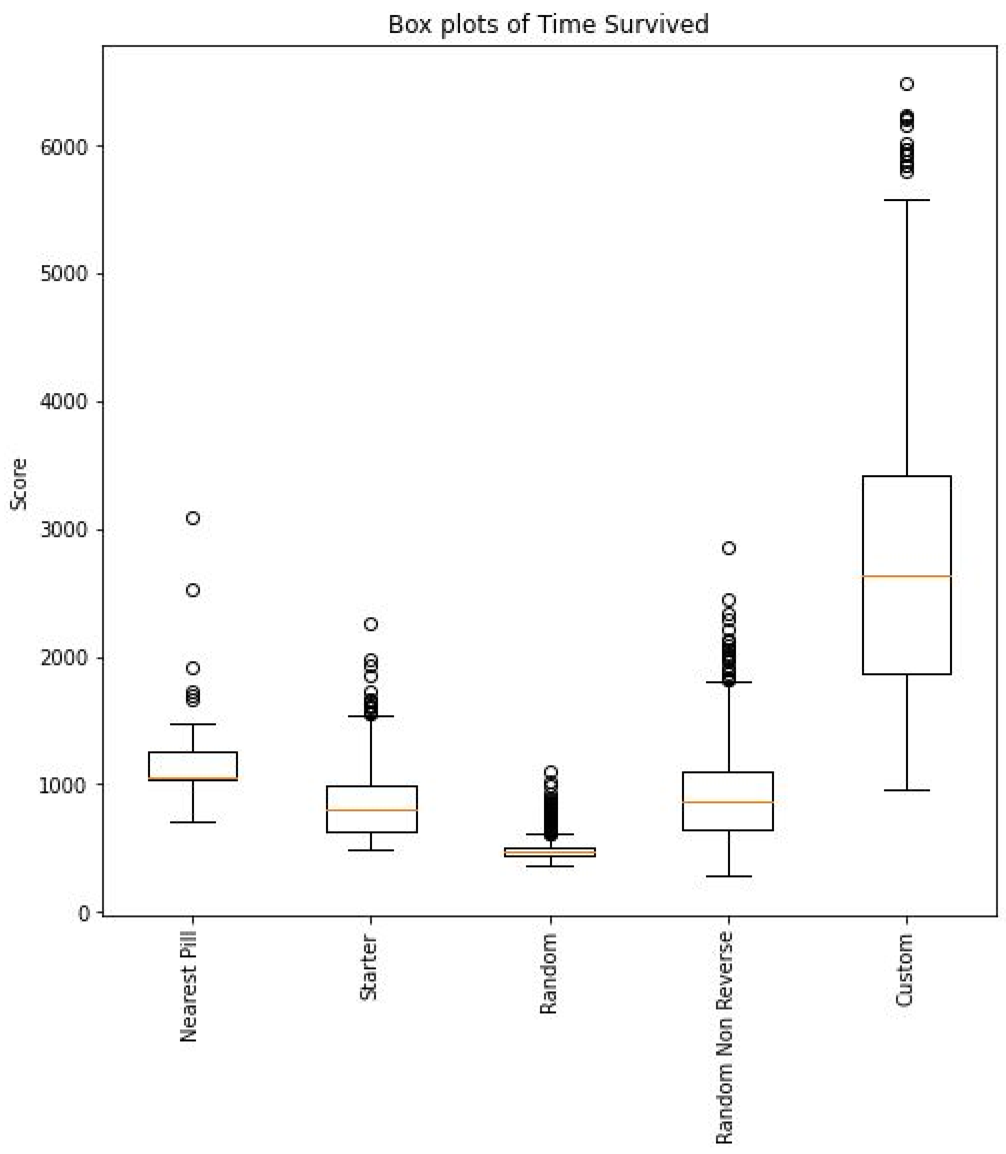

Figure 7: Box Plot of Time Survived per Controller.

4.3 Level Reached

When considering level reached, as this is a discrete variable, our metric of interest is the mode of level reached, i.e. the most frequent level reached per controller in the experiments. The below table shows the counts of each level reached for each of the 5 controllers. Our custom controller had a mode of level 2, while the remaining controllers had a mode of level 1. The custom controller also had a bigger range, from level 1 to level 5, and has more counts across the first three levels in comparison to the other controllers.

Level Reached

Controller	**Level 1**	**Level 2**	**Level 3**	**Level 4**	**Level 5**
Custom	258	435	269	37	1
Starter	999	1	-	-	-
Random	1000	-	-	-	-
Random Non reverse	1000	-	-	-	-
Nearest Pill	687	311	2	-	-
N = 1000					

Figure 8: Statistics of Level Reached per Controller.

The bar plot below graphically illustrates the above information from the table. We can see our custom controller tends to progress further in the levels in comparison to the other controllers.

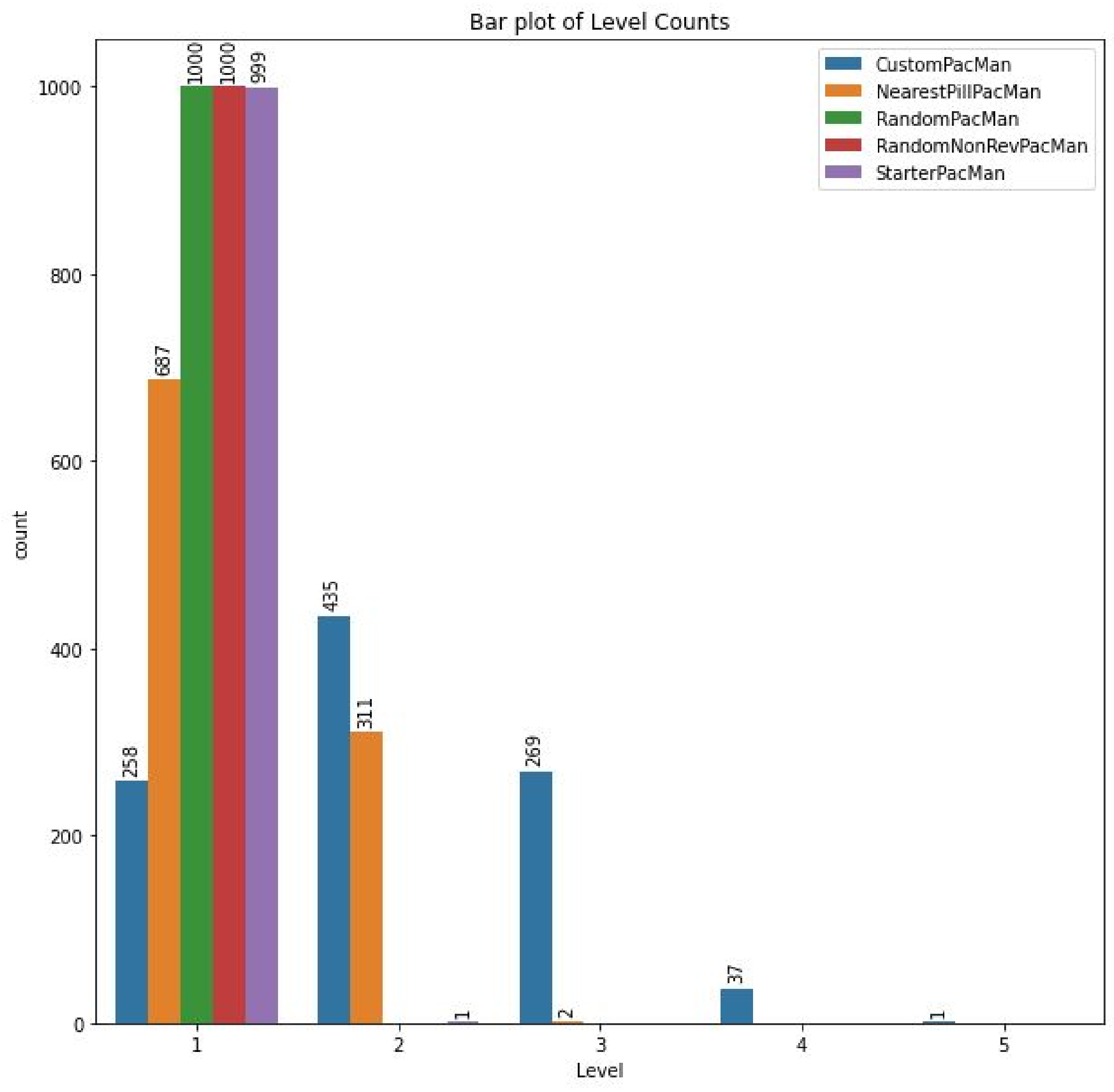

Figure 9: Bar Plot of Counts of Level Reached per Controller.

For a more clear visualization, below is the individual bar plot for our custom controller level counts. We can see level 2 is the most frequent level our controller reaches.

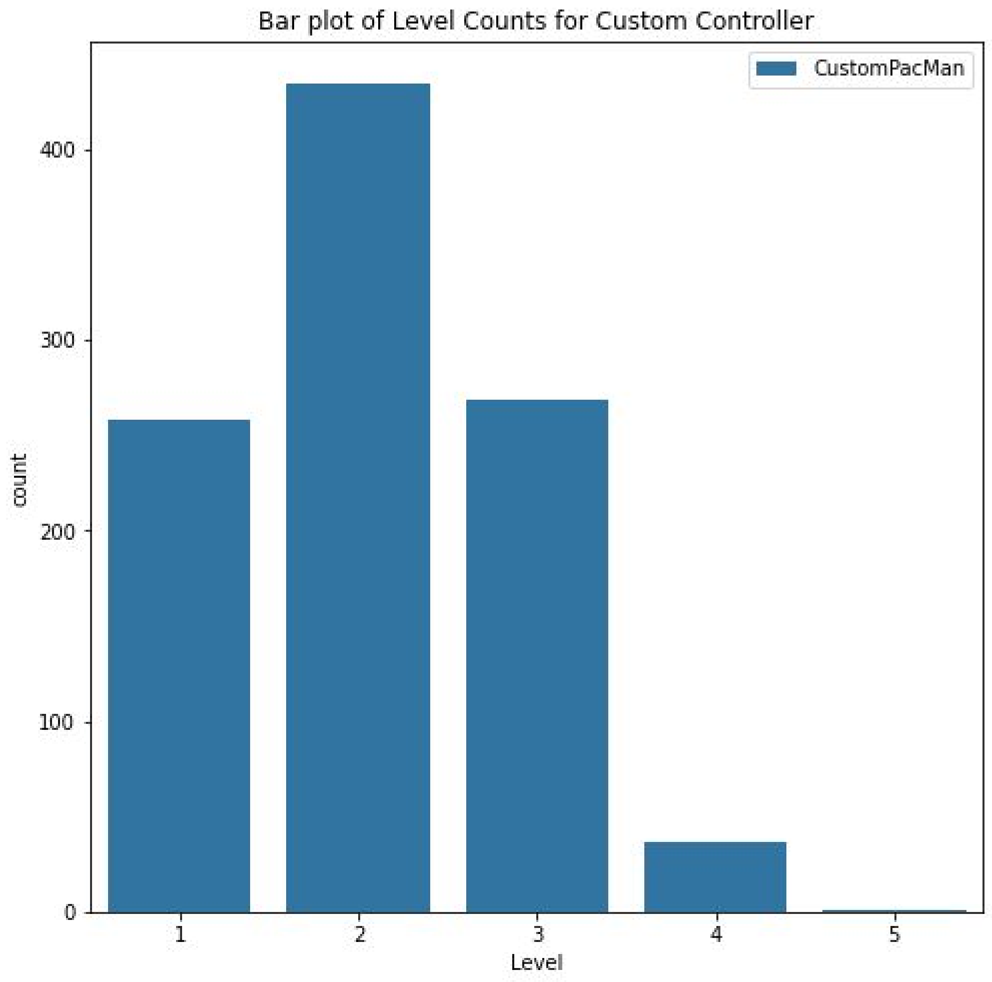

Figure 10: Bar Plot of Counts of Level Reached for Custom Controller.

Correlation between Performance Indicators

It was interesting just to have a quick look at the correlation of each of the key performance variables to see how they correlated with another. Below are some plots to show the correlation of each of the variables. There is a strong positive correlation between each of the performance indicators, which is expected. As the longer you live, the more you can increase your score and progress through each of the levels. It also makes sense that the more levels you progress through there would be more opportunity to score more points.

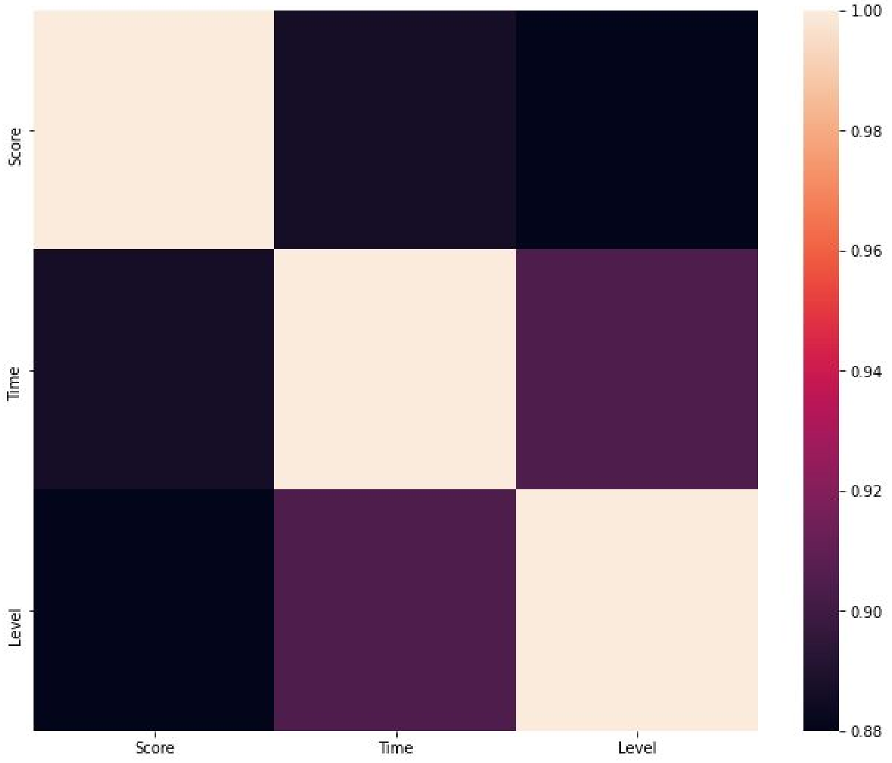

Figure 11: Correlation Matrix Plot of Performance Indicators.

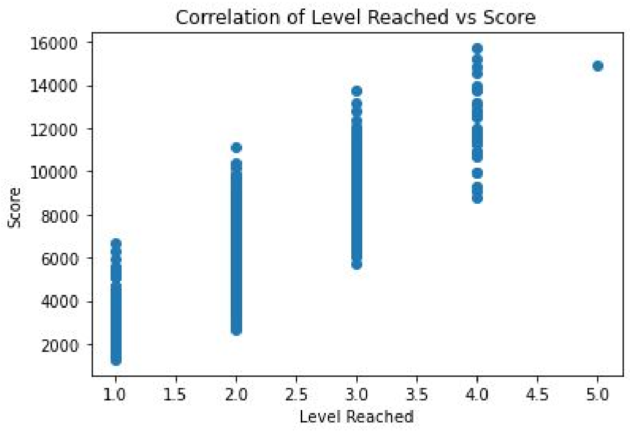

Figure 12: Scatter Plot of Level Reached vs Score.

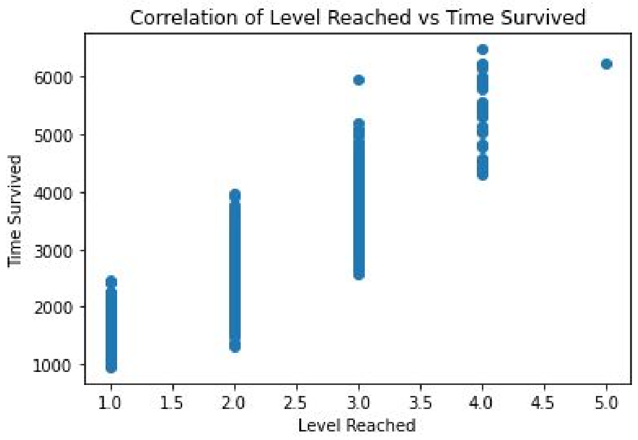

Figure 13: Scatter Plot of Level Reached vs Time Survived.

All 3 plots show a strong positive correlation. The longer you live, the more time you could gain points as well as progress through each of the levels. It also makes sense that the more levels you progress through the more likely you are to increase your score. It is important to remember correlation does not mean causation. However, we can see there is a positive relationship between each of these performance indicators.

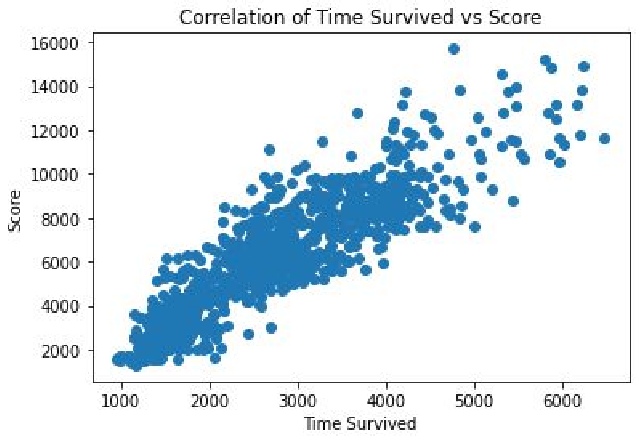

Figure 14: Scatter Plot of Time Survived vs Score.

5. Conclusion

Our controller outperforms the example controllers in all 3 performance indicators we considered to be informative. The below table shows the key metrics of interest for each controller so we can easily see the differences in controller performance. Again, across all 3 performance indicators a higher score is more desirable.

Performance Indicators

Controller	**Score (Median)**	**Time Survived (Median)**	**Level Reached (Mode)**
Custom	6130 [5958.53, 6301.46]	2637 [2571.06, 2702.93]	2
Starter	2670[2565.48, 2774.51]	804.5 [788.59, 820.40]	1
Random	180 [175.70, 184.29]	468.0 [462.89, 473.10]	1
Random Non Reverse	1050 [1014.63, 1085.36]	863.0 [840.06, 885.93]	1
Nearest Pill	2080 [2044.58, 2115.41]	1056.0 [1045.74, 1066.25]	1
N = 1000			

Figure 15: Statistics of Performance Indicators per Controller Table.