

Assignment #2: Tetris AI – Report

By Omer Guven

Contents

<u>Iteration 1</u>	2
<u>Iteration 2</u>	4
<u>Iteration 3</u>	6
<u>Iteration 4</u>	11
<u>Iteration 5</u>	15
<u>Results Of Iteration 1-5 Graph Representation</u>	17
<u>Iteration 6</u>	18
<u>Iteration 7</u>	22
<u>Iteration 8</u>	46
<u>Final Iteration</u>	47
<u>Final Tetris AI Code</u>	49
<u>Conclusion</u>	53

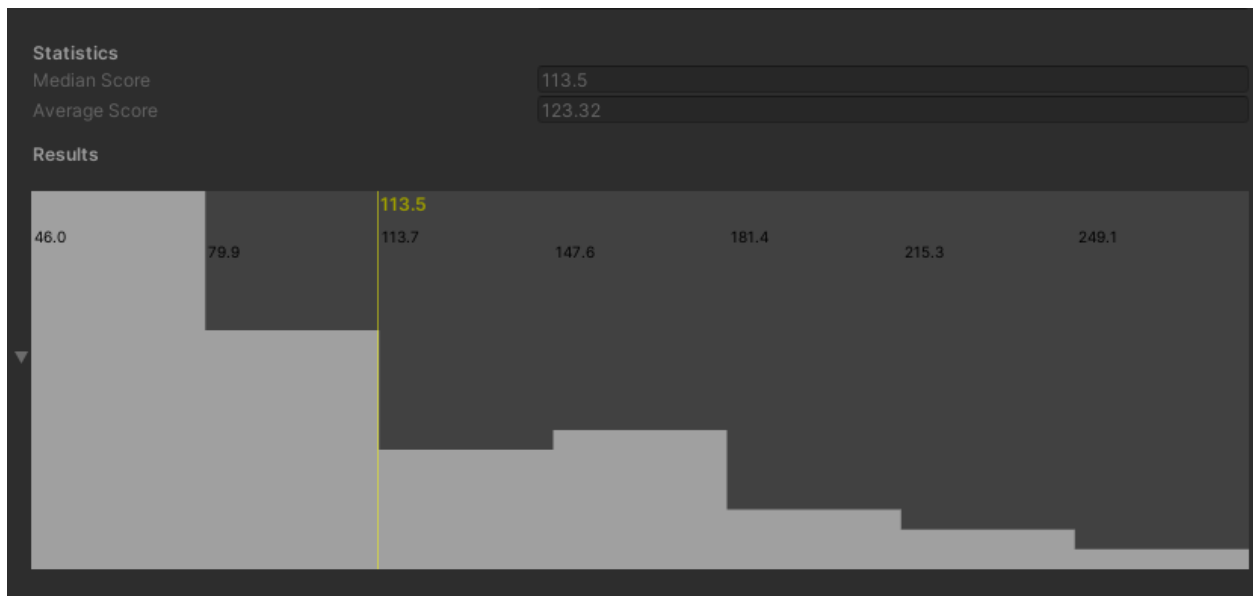
Iteration 1

With this project I started off with two simple factors that will count towards the heuristic and cost of the move. I decided to check the number of empty cells on the grid and the maximum height within the game.

```
public float Heuristic(Move move)
{
    int maxHeight = 0;
    int emptyCells = 0;
    // Simulates the effect of the move on the board
    TetrisState state = Tetris.SimulateMove(move);
    //find number of empty cells
    for (int x = 0; x < Tetris.Size.x; x++)
    {
        for (int y = 0; y < state.GetColumnHeight(x); y++)
        {
            if (state.IsEmpty(x, y))
            {
                emptyCells++;
            }
        }
        //find maxheight
        int height = state.GetColumnHeight(x);
        if (height > maxHeight)
        {
            maxHeight = height;
        }
    }
    //calculate cost
    float cost = 2 * emptyCells + ((float)1.5 * maxHeight);
    return cost;
}
```

Then I calculated the cost of each move by giving them a random weight to be multiplied by. Using utility theory, I decided the empty cells factor was more important than the max height factor. Which influenced me to give the empty cells a larger value to be multiplied by than max height. Then the Tetris AI would check all possible moves and have a cost to each move and take the move with the lowest cost and play that move.

```
public override int ChooseMove(Move[] moves)
{
    //take lowest cost move
    return moves.IndexOfMin(move => Heuristic(move));
}
```



(Results Of Iteration 1 Form 50 Simulations)

With the code from iteration 1 I did a test of 50 simulations and got an average score of 113.5 and a median score of 123.32.

Iteration 2

For iteration 2 I decided to improve my code and add more factors that would influence the positions the pieces would be placed within and change some of the values the factors are being multiplied by to calculate the cost.

First, I added a factor that would calculate the distance of the piece that is going to be placed from the middle point of the grid. Then reusing utility theory, I decided to change the values that the factors are being multiplied to larger values and decided that the empty cells should have the most influence on the cost, so I gave it the largest value to be multiplied by and then gave the distance from middle factor the second largest value to be multiplied by then finally gave the max height a value to be multiplied by. The reason behind this was having empty spaces would make it harder to clear the row so by giving it the largest value to be multiplied by it would cause there to be less empty spaces on the grid and then having more pieces on one side is more beneficial since it can put the bad pieces that are not needed on to one side and work around those pieces and finally by having max height it would stop one side from being extremely tall to the point where it is impossible to stop the game from ending.

```
public float Heuristic(Move move)
{
    //initialise variables
    int maxHeight = 0;
    int emptyCells = 0;
    int distance = 0;
    // Simulates the effect of the move on the board
    TetrisState state = Tetris.SimulateMove(move);

    //nested for loop to count number of empty cells
    for (int x = 0; x < Tetris.Size.x; x++)
    {
        for (int y = 0; y < state.GetColumnHeight(x); y++)
        {
            if (state.IsEmpty(x, y))
            {
```

```

        emptyCells++;
    }
}
int height = state.GetColumnHeight(x);
if (height > maxHeight)
{
    maxHeight = height;
}
//find distance from x to mid
if (x >= Tetris.Size.x / 2)
{
    distance = x - Tetris.Size.x / 2;
}

}
//calculate cost
float cost = 7 * emptyCells + ((float)3 * maxHeight) + 4 * distance;
return cost;

```



(Results Of Iteration 2 From 50 Simulations)

In iteration 2 I got a median score of 111.5 and an average score of 109.24 which is a decrease in both median score and average score compared to iteration 1. This could have been due to

the new factor that I decided to add to the heuristic or due to the values the factors are being multiplied by hence causing a negative overall impact in the average score and median score compared to iteration 1's results.

Iteration 3

During this iteration I decided to add more factors that would influence the cost of the move and removed the distance from the middle factor since it was one of the reasons that the Tetris AI from iteration 2 performed badly compared to the Tetris AI from iteration 1. After that, using utility theory I altered some of the values that would be multiplied with the factors which will calculate the cost of a move.

Then I decided to add a hole's factor which checks the number of holes that a Tetris block would create by checking the number of holes from one of the bottom corners to the other side on the x axis up to the column height of a piece on the board. Also, I checked if the piece that is being placed is creating pillars by counting the number of empty spaces around it and saved it to int counter. The reason I decided to check for pillars is whilst I was running my AI to see how it was performing, I realised that some pieces created pillars which only certain blocks can be placed within to clear the row, this caused the AI to perform bad since those pillar blocks cause it to die sooner hence, I added a factor to the heuristic which checks if a piece is going to create pillars since pillars were bad for the AI's performance. Then I decided to check for bumpiness since the bumpier it gets the harder it would be to place certain pieces on the board hence by adding this as a factor it should help with the performance of the AI. I checked it by finding the difference between column height before move simulation and after move simulation. Finally, I altered some of the values that are used to multiply the factors using utility theory and a similar method to the one I used from iteration 1 and 2 where I would assess which factor is more important and assign the higher value to that factor.

```
public float Heuristic(Move move)
{
    //initialise variables
    int maxHeight = 0;
    int holes = 0;
    int max = 0;
    int emptyCells = 0;
    int min = Tetris.State.GetMaxColumnHeight();
```

```

// Simulates the effect of the move on the board
TetrisState state = Tetris.SimulateMove(move);

//calculate max height
for (int x = 0; x < Tetris.Size.x; x++)
{
    if (state.GetColumnHeight(x) > max)
    {
        max = state.GetColumnHeight(x);
    }
    int height = state.GetColumnHeight(x);
    if (height > maxHeight)
    {
        maxHeight = height;
    }
}

//calculate number of empty cells and minimum height
for (int x = 0; x < Tetris.Size.x; x++)
{
    if (min > state.GetColumnHeight(x))
    {
        min = state.GetColumnHeight(x);
    }

    for (int y = 0; y < state.GetColumnHeight(x); y++)
    {
        if (state.IsEmpty(x, y))
        {
            emptyCells++;
        }
    }
}

//calculate number of empty holes
for (int j = 0; j < Tetris.Size.x; j++)
{

```

```

for (int y = 0; y < state.GetColumnHeight(j); y++)
{

    if (state.IsEmpty(j, y))
    {
        holes++;
    }

}
}

```

//calculate bumpiness

```

int bumpiness = state.GetMaxColumnHeight() - min;
int counter = 0;

```

//calculates the number of spaces between the piece that is being placed indicating that it is causing a pillar.

```

for (int x = 0; x < Tetris.Size.x; x++)
{
    for (int i = Tetris.Size.y - 1; i >= 0; i--)
    {
        if (state.IsFull(x, i))
            break;

        if (state.IsFull(x - 1, i) && state.IsFull(x + 1, i))
        {
            counter++;
        }

    }
}

```

//calculates maximum height difference of board and the highest piece on the board
maxHeight = maxHeight - Tetris.State.GetMaxColumnHeight();

//float cost = (holes * 4)+ 3*emptyCells + ((float)1.5 * maxHeight) + bumpiness * 2 + counter * 5;


```

float cost = (holes * 4) + 1 * emptyCells + ((float)1.5 * maxHeight) + bumpiness * 2 +
counter * (float)2.5;
return cost;
}

```

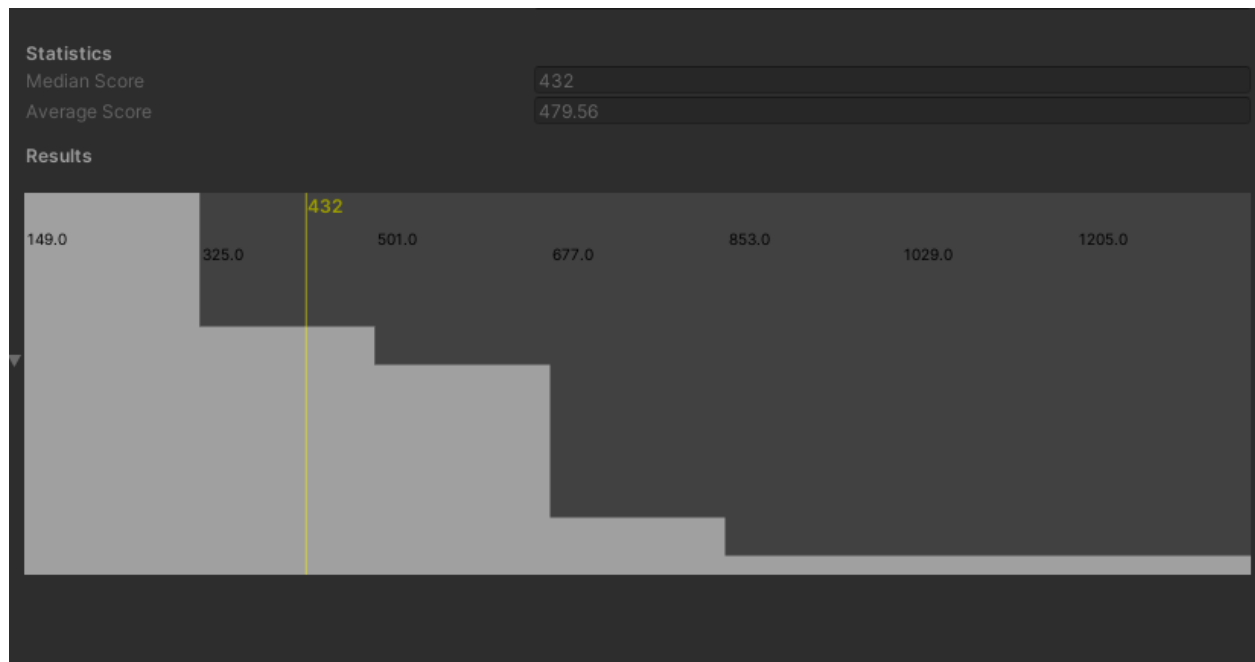
The first 50 simulations were done using $\text{float cost} = (\text{holes} * 4) + 3 * \text{emptyCells} + ((\text{float})1.5 * \text{maxHeight}) + \text{bumpiness} * 2 + \text{counter} * 5$; to calculate the cost.



(Results Of Iteration 3 Part 1 From 50 Simulations)

The results of 50 simulations show a median score of 218.5 and an average score of 222.78 which is an increase compared to iteration 1 and 2 showing that the changes in iteration 3 were beneficial since the AI performed better within the 50 simulations compared to the AI in iteration 1 and 2. The results also show consistency as the average and median scores are very similar suggesting that the AI is consistent with the scores it is producing.

After this I decided to alter some of the values within the cost to see the impact it would have on the results, so I used $\text{float cost} = (\text{holes} * 4) + 1 * \text{emptyCells} + ((\text{float})1.5 * \text{maxHeight}) + \text{bumpiness} * 2 + \text{counter} * (\text{float})2.5$; and done a simulation of 50 for the new values that are going to influence the factors. I decided to alter the value that is being multiplied with the counter since I believed it should not have a great effect on the placement of the pieces and changed the value that empty cells are being multiplied by to see how the AI would perform.



(Results Of Iteration 3 Part 2 From 50 Simulations)

The results of 50 from the new cost calculation shows that the changes allowed the AI to perform better hence proving my theory that counter should not have had the largest impact on the placement of the Tetris pieces, since the results from this test shows the median score as 432 and the average score as 479.56 which is a significant increase compared to the results from iteration 1 and 2 and first part of iteration 3 before the calculation for the cost was altered. Hence meaning that the Tetris AI now is performing better than before and is still consistent with its results since the median score and average score are still similar.

Iteration 4

During iteration 4 I decided to make my code neater and more readable by creating new functions that would return the output of each factor, for example the empty cells function would return the number of empty cells. Then I used the past values that I multiplied the factors with as inspiration and used utility theory to assess which factors should have the most impact on the AI's decision making which allowed me to assign the factors values to be multiplied by.

```
//calculate and return number of empty cells
public float EmptyCells(Move move)
{
    int emptyCells = 0;
    int min = Tetris.State.GetMaxColumnHeight();
    // Simulates the effect of the move on the board
    TetrisState state = Tetris.SimulateMove(move);
    for (int x = 0; x < Tetris.Size.x; x++)
    {
        if (min > state.GetColumnHeight(x))
        {
            min = state.GetColumnHeight(x);
        }

        for (int y = 0; y < state.GetColumnHeight(x); y++)
        {
            if (state.IsEmpty(x, y))
            {
                emptyCells++;
            }
        }
    }
    return emptyCells;
}

//calculate and return bumpiness by finding the difference between column height before
move simulation and after move simulation
```

```

public float bumpiness(Move move)
{
    int min = Tetris.State.GetMaxColumnHeight();
    // Simulates the effect of the move on the board
    TetrisState state = Tetris.SimulateMove(move);
    int bumpiness = state.GetMaxColumnHeight() - min;
    return bumpiness;
}

//calculate and return max height
public float maxHeight(Move move)
{
    int maxHeight = 0;
    int max = 0;
    // Simulates the effect of the move on the board
    TetrisState state = Tetris.SimulateMove(move);
    for (int x = 0; x < Tetris.Size.x; x++)
    {
        if (state.GetColumnHeight(x) > max)
        {
            max = state.GetColumnHeight(x);
        }
        int height = state.GetColumnHeight(x);
        if (height > maxHeight)
        {
            maxHeight = height;
        }
    }
    maxHeight = maxHeight - Tetris.State.GetMaxColumnHeight();
    return maxHeight;
}

//calculate and return number of holes the Tetris block has created
public float Holes(Move move)
{
    int holes = 0;
    // Simulates the effect of the move on the board

```

```

TetrisState state = Tetris.SimulateMove(move);
for (int j = 0; j < Tetris.Size.x; j++)
{
    for (int y = 0; y < state.GetColumnHeight(j); y++)
    {

        if (state.IsEmpty(j, y))
        {
            holes++;
        }
    }
}
return holes;
}

```

//calculates the number of spaces between the piece that is being placed indicating that it is causing a pillar.

```

public float pillar(Move move)
{
    // Simulates the effect of the move on the board
    TetrisState state = Tetris.SimulateMove(move);
    int counter = 0;

    for (int x = 0; x < Tetris.Size.x; x++)
    {
        for (int i = Tetris.Size.y - 1; i >= 0; i--)
        {
            if (state.IsFull(x, i))
                break;

            if (state.IsFull(x - 1, i) && state.IsFull(x + 1, i))
            {
                counter++;
            }
        }
    }
}

```

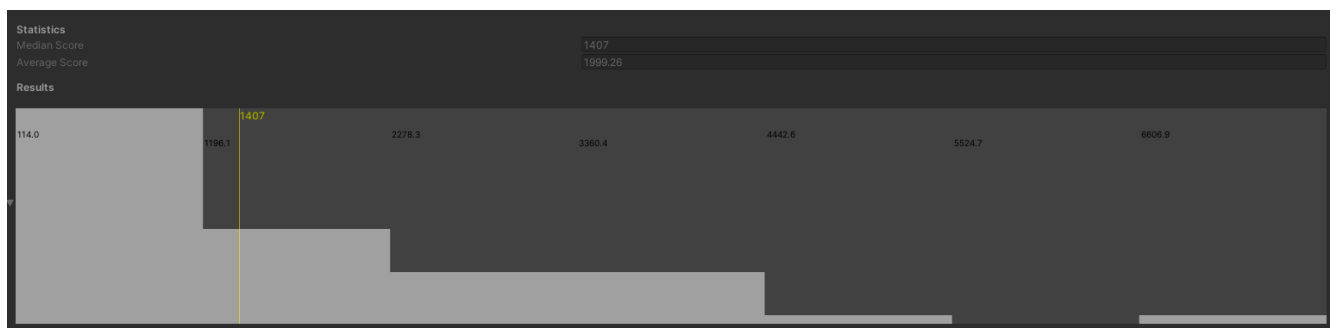
```

    return counter;
}

public float Heuristic(Move move)
{
    float Pillar = pillar(move);
    float holes = Holes(move);
    float emptyCells = EmptyCells(move);
    float MaxHeight = maxHeight(move);
    float Bumpiness = bumpiness(move);

    //calculate cost of move
    float cost = (holes * 4) + 3 * emptyCells + ((float)2 * MaxHeight) + Bumpiness * 3 + Pillar *
(float)2.5;
    return cost;
}

```



(Results Of Iteration 4 Form 50 Simulations)

The results of 50 simulations for iteration 4 were 1407 as the median score and 1999.26 as the average score showing that the changes made during this iteration had a significantly good impact on the performance of the Tetris AI since when you compare it to the results of the AIs from past iterations this has a significantly higher output on both median score and average score suggesting that it is the better performing AI and that the changes made were beneficial. However, the AI has a larger spread in score as its maximum score was 6606.9 and minimum was 114.0 but the difference between the average score and median is not large meaning that the AI is still performing consistently but the difference between average score and median has increased.

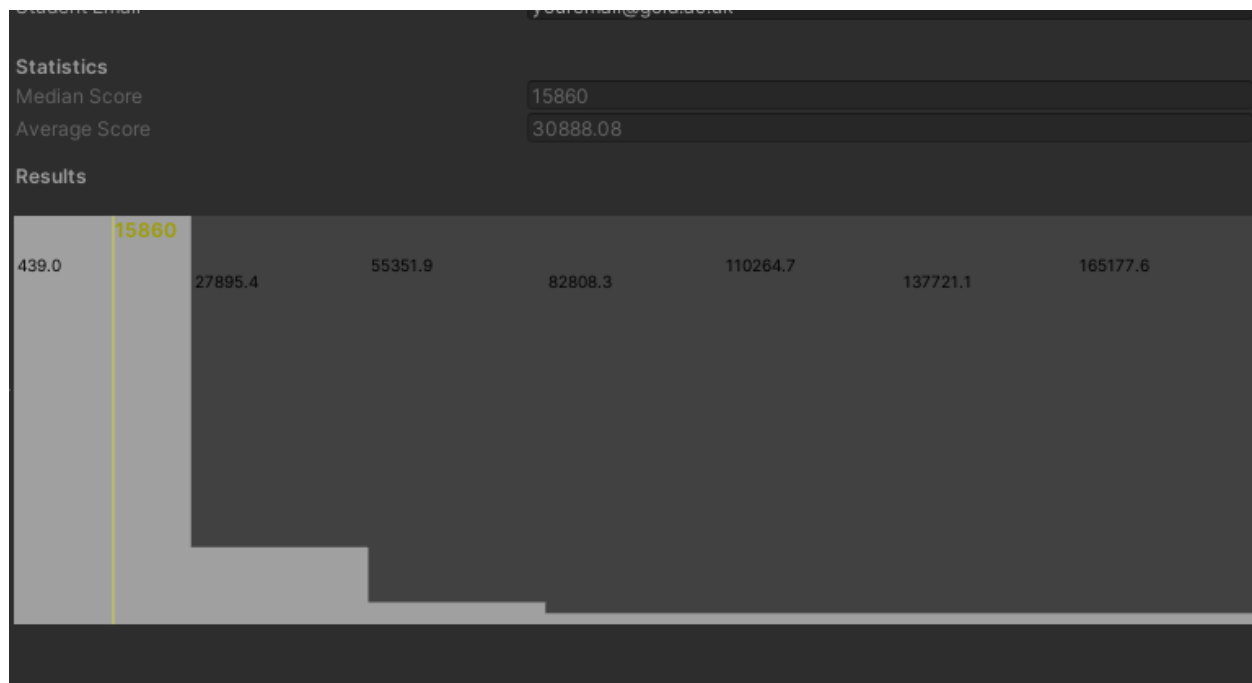
Iteration 5

During this iteration I decided to change the values that the factors were being multiplied by when calculating the cost. The reason for this is I wanted to see whether the Tetris AI's score can get higher without adding more factors to the cost calculation. Also by doing this if I was to improve the Tetris AI's score drastically it would show potential that these factors can allow the Tetris AI to perform better if it had the correct values/weight to multiply the factors with meaning that using evolutionary algorithm would be beneficial since the weights/values that the factors are multiplied by were given using utility theory and assessing which factor is more important, then assigning it the larger weight/value to specific factors so they can be multiplied. So, by using evolutionary algorithm it can give the best values to multiply the factors with as it would constantly improve on the last best values that it achieved which could potentially allow the Tetris AI to do better than before and have a good rate of growth causing its performance to increase as well.

```
public float Heuristic(Move move)
{
    float Pillar = pillar(move);
    float holes = Holes(move);
    float emptyCells = EmptyCells(move);
    float MaxHeight = maxHeight(move);
    float Bumpiness = bumpiness(move);

    //calculate cost of move
    float cost = (holes * 5) + 3 * emptyCells + ((float)2 * MaxHeight) + Bumpiness * 3 + Pillar *
(float)2.5;
    return cost;
}
```

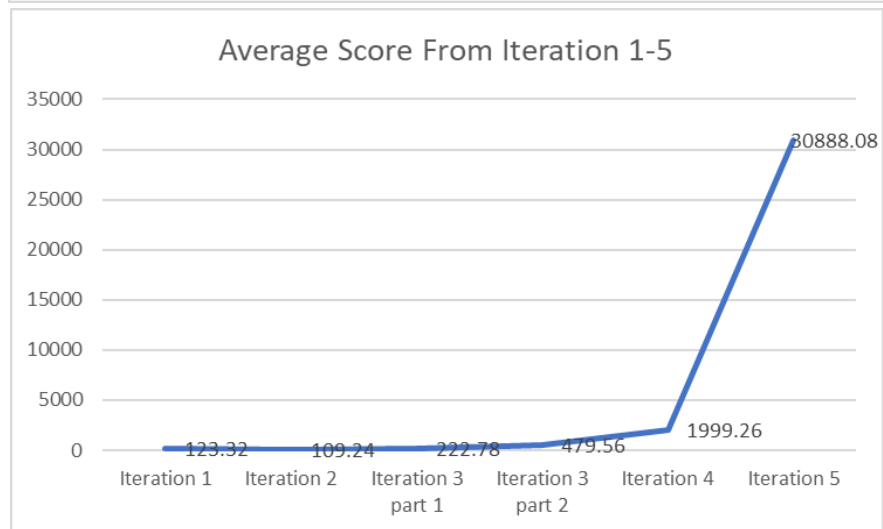
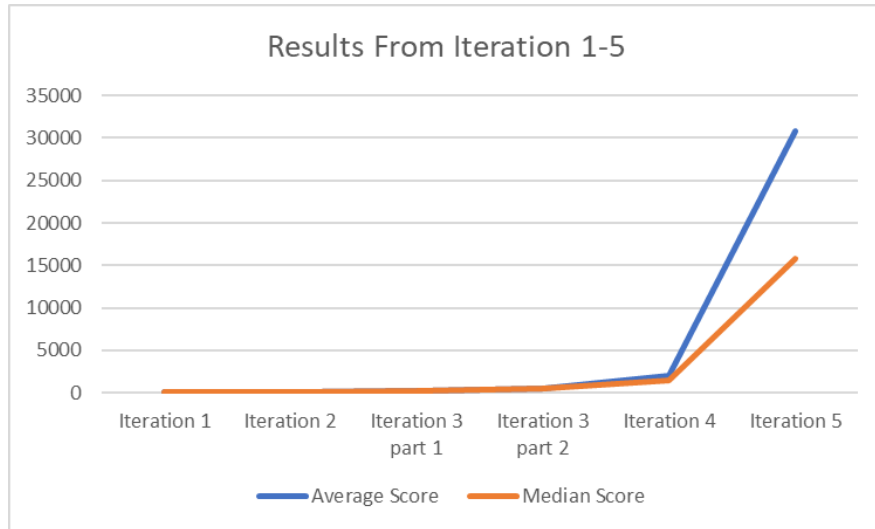
In the code above all that was changed was the values that are multiplied with the factors to calculate the moves cost. I tried a couple different sets of values but the one above within the code performed the best when I ran the code as it was getting the higher scores consistently compared to the other values that I used in the cost calculation. Then I proceeded to run 50 simulations on the Tetris AI.

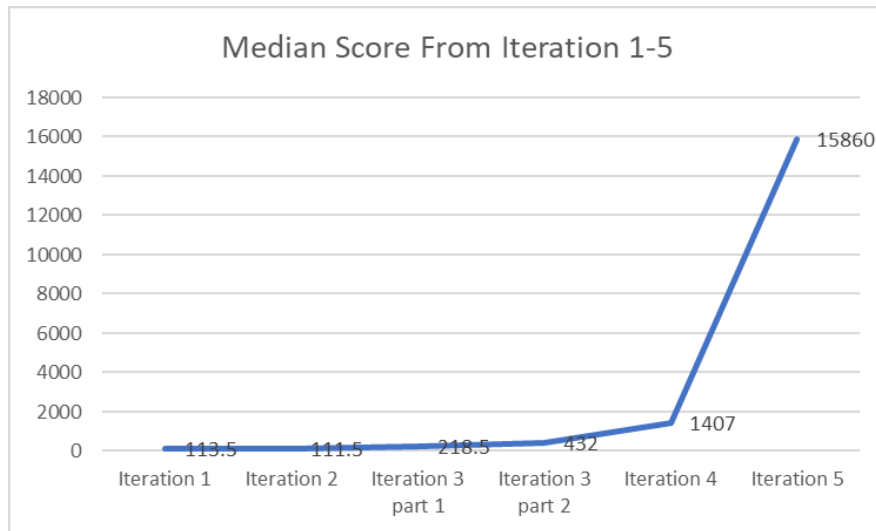


(Results Of Iteration 4 Form 50 Simulations)

The results from 50 simulations returned a median score of 15860 and an average score of 30888.08 which the median is 14453 more than the Tetris AI from iteration 4 and the average is 28888.82 more than the Tetris AI from iteration 4 suggesting that changes to the values that are used to multiply the factors with has had a beneficial impact on the Tetris AI's performance. This also helps me decide that these set of factors may be sufficient to get even better scores if an evolutionary algorithm was performed on it because the only change between this and the cost calculation in iteration 6 was the multiplier for holes which was 4 and within this iteration, it is 5. However, there is a significant difference between the average score and median suggesting that its performance is not consistent as the average is nearly double the median which may cause a problem. But if the evolution algorithm is used on these factors, it may be able to give the best genome that gives consistent scores making the median and average to be similar. In the next iteration I will be trying to implement an evolution algorithm to my work.

Results Of Iterations 1 - 5 Graph Representation





The graphs above show that the average score and median have been similar until iteration 5 where they are no longer similar which could have been due to the values the factors are being multiplied by. It also shows that with the right values to multiply the factors with the AI can substantially better.

Iteration 6

After seeing how well the Tetris AI performed in iteration 5, I decided that using an evolutionary algorithm would be sufficient and beneficial since it was doing 30888.08 average score with a median score of 15860 from the random numbers that I assigned after using utility theory and assessing which factor is the most important and assigning it the larger value. To implement the evolutionary algorithm, I created two new files, one called TetrisEvolution_oguve001 and the other called TetrisWorld_oguve001. The code for both files can be found below.

Code Within TetrisEvolution_oguve001.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using AlanZucconi.AI.Evo;

public class TetrisEvolution_oguve001 : EvolutionSystem<ArrayGenome>
{
}
}
```

Code Within TetrisWorld_oguve001.cs

```
using UnityEngine;
using AlanZucconi.Tetris;
using AlanZucconi.AI.Evo;

public class TetrisWorld_oguve001 :
    MonoBehaviour,
    IWorld<ArrayGenome>,
    IGenomeFactory<ArrayGenome>
{
    public TetrisGame Tetris;
    public TetrisAI_oguve001 AI;
    private ArrayGenome Genome;

    public void SetGenome(ArrayGenome genome)
    {
        AI = ScriptableObject.CreateInstance<TetrisAI_oguve001>();
        AI.x = genome.Params[0] * 10f + 10f;
        AI.c = genome.Params[1] * 10f + 10f;
        AI.v = genome.Params[2] * 10f + 10f;
        AI.b = genome.Params[3] * 10f + 10f;
        AI.n = genome.Params[4] * 10f + 10f;

        Tetris.TetrisAI = AI;
        Genome = genome;
    }

    public ArrayGenome GetGenome()
    {
        return Genome;
    }

    public void StartSimulation()
```

```

{
    Tetris.StartGame();
}

public bool IsDone()
{
    return !Tetris.Running;
}

public float GetScore()
{
    return Tetris.Turn;
}

public new ArrayGenome Instantiate()
{
    ArrayGenome genome = new ArrayGenome(5);
    genome.InitialiseRandom();
    return genome;
}

public void ResetSimulation()
{
    AI = ScriptableObject.CreateInstance<TetrisAI_oguve001>();
    Tetris.TetrisAI = AI;
}
}

```

After the files were created, I had to change some parts in my Tetris AI file to get the evolutionary algorithm to work . Below is the code that was added and altered within my Tetris AI file to have the evolutionary algorithm to work.

```

//values the evolution algorithm assigns are saved in these variables
public float x;
public float c;
public float v;
public float b;

```

```
public float n;
```

```
//calculate heuristic cost
public float Heuristic(Move move)
{
    float Pillar = pillar(move);
    float holes = Holes(move);
    float emptyCells = EmptyCells(move);
    float MaxHeight = maxHeight(move);
    float Bumpiness = bumpiness(move);

    //values the evolution algorithm made are used here to calculate the cost.
    float cost = (holes * x) + c * emptyCells + ((float)v * MaxHeight) + Bumpiness * b + Pillar *
(float)n;
    return cost;
}
```

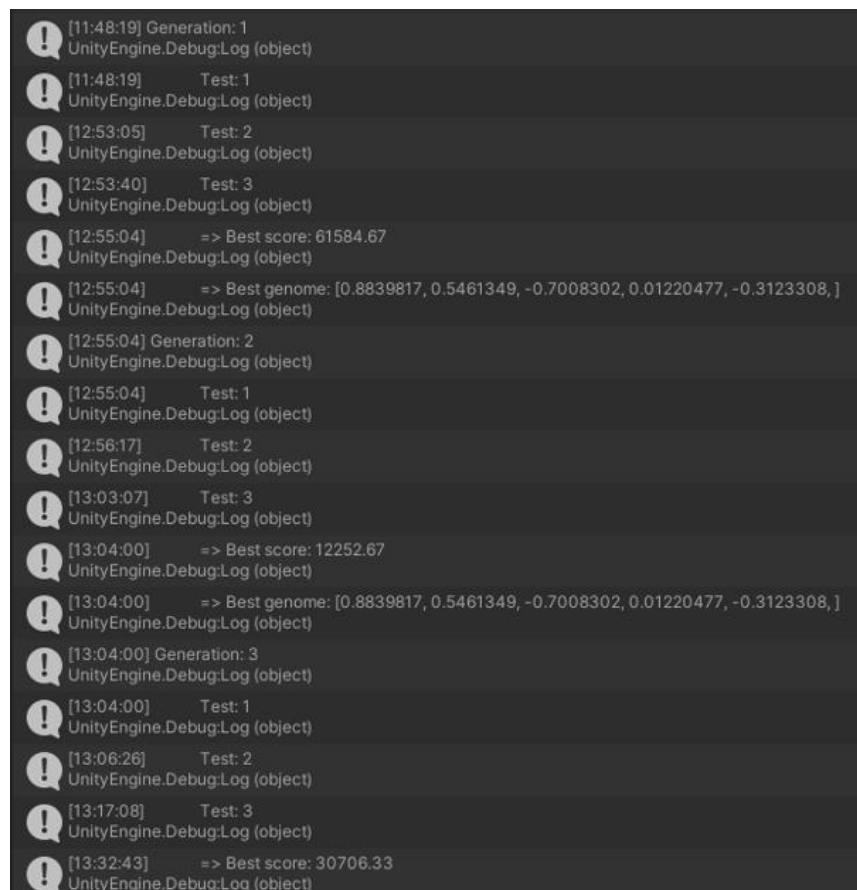
Finally, Within unity I created ten new Tetris game instances that will take the parameters the evolution algorithm assigns it and would run the code using those new weights/values to multiply the factors with so it can calculate the cost of the moves.



(Ten Instances Of TetrisGame For Evolution Algorithm To Use)

Iteration 7

During iteration 7 I started to run the evolutionary algorithm. I decided to add no first genome so it can do the evolution naturally. I also gave it a survival rate of 0.5, 2 mutations and 3 tests per genome. I had the evolution algorithm running and stopped my evolution algorithm from running manually and saved the best scores genome. Then the next time I ran it I added the best scores genome from the last time I ran it as the first genome and had it run again. I have done this multiple times and took screenshots of it running. However, I lost my first set of results when I first ran the evolution algorithm. You can see the results from the algorithm below. Next, I decided to create graphs in excel for each time I ran the evolution algorithm. Then finally I created a final graph that has all the best scores of each generation to get a visual representation of how it was performing.



```
[11:48:19] Generation: 1
UnityEngine.Debug:Log (object)
[11:48:19] Test: 1
UnityEngine.Debug:Log (object)
[12:53:05] Test: 2
UnityEngine.Debug:Log (object)
[12:53:40] Test: 3
UnityEngine.Debug:Log (object)
[12:55:04] => Best score: 61584.67
UnityEngine.Debug:Log (object)
[12:55:04] => Best genome: [0.8839817, 0.5461349, -0.7008302, 0.01220477, -0.3123308,]
UnityEngine.Debug:Log (object)
[12:55:04] Generation: 2
UnityEngine.Debug:Log (object)
[12:55:04] Test: 1
UnityEngine.Debug:Log (object)
[12:56:17] Test: 2
UnityEngine.Debug:Log (object)
[13:03:07] Test: 3
UnityEngine.Debug:Log (object)
[13:04:00] => Best score: 12252.67
UnityEngine.Debug:Log (object)
[13:04:00] => Best genome: [0.8839817, 0.5461349, -0.7008302, 0.01220477, -0.3123308,]
UnityEngine.Debug:Log (object)
[13:04:00] Generation: 3
UnityEngine.Debug:Log (object)
[13:04:00] Test: 1
UnityEngine.Debug:Log (object)
[13:06:26] Test: 2
UnityEngine.Debug:Log (object)
[13:17:08] Test: 3
UnityEngine.Debug:Log (object)
[13:32:43] => Best score: 30706.33
UnityEngine.Debug:Log (object)
```

```

[13:32:43] => Best score: 30706.33
UnityEngine.Debug:Log (object)
[13:32:43] => Best genome: [0.8839817, 0.5461349, -0.7008302, 0.01220477, -0.3123308, ]
UnityEngine.Debug:Log (object)
[13:32:43] Generation: 4
UnityEngine.Debug:Log (object)
[13:32:43] Test: 1
UnityEngine.Debug:Log (object)
[13:58:56] Test: 2
UnityEngine.Debug:Log (object)
[14:01:45] Test: 3
UnityEngine.Debug:Log (object)
[14:40:44] => Best score: 68339.66
UnityEngine.Debug:Log (object)
[14:40:44] => Best genome: [0.8839817, 0.4584129, -0.7008302, 0.01220477, -0.3123308, ]
UnityEngine.Debug:Log (object)
[14:40:44] Generation: 5
UnityEngine.Debug:Log (object)
[14:40:44] Test: 1
UnityEngine.Debug:Log (object)
[14:47:47] Test: 2
UnityEngine.Debug:Log (object)

```

```

[15:22:59] Test: 3
UnityEngine.Debug:Log (object)
[15:53:20] => Best score: 56201.67
UnityEngine.Debug:Log (object)
[15:53:20] => Best genome: [0.8839817, 0.1402875, -0.7008302, 0.01220477, -0.3123308, ]
UnityEngine.Debug:Log (object)
[15:53:20] Generation: 6
UnityEngine.Debug:Log (object)

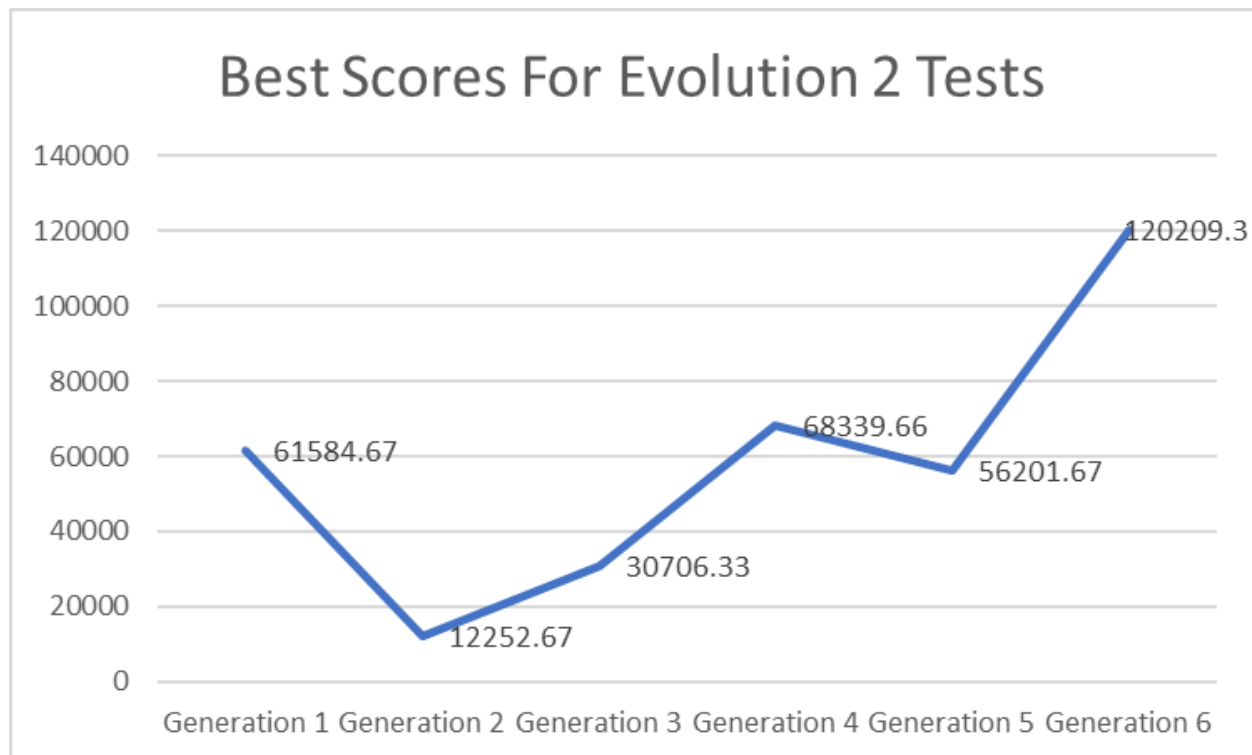
```

```

[15:53:20] Generation: 6
UnityEngine.Debug:Log (object)
[15:53:20] Test: 1
UnityEngine.Debug:Log (object)
[16:05:31] Test: 2
UnityEngine.Debug:Log (object)
[16:34:16] Test: 3
UnityEngine.Debug:Log (object)
[17:38:08] => Best score: 120209.3
UnityEngine.Debug:Log (object)
[17:38:08] => Best genome: [0.8839817, 0.1402875, -0.7008302, -0.08404892, -0.3123308, ]
UnityEngine.Debug:Log (object)
[17:38:08] Generation: 7
UnityEngine.Debug:Log (object)
[17:38:08] Test: 1
UnityEngine.Debug:Log (object)
[18:47:18] Test: 2
UnityEngine.Debug:Log (object)

```

(Results Of 2nd Time Running Evolution Algorithm)



(Results Of 2nd Evolution Run On Graph)

```
[12:40:11] Generation: 1
UnityEngine.Debug:Log (object)

[12:40:11]      Test: 1
UnityEngine.Debug:Log (object)

[13:09:07]      Test: 2
UnityEngine.Debug:Log (object)

[13:18:17]      Test: 3
UnityEngine.Debug:Log (object)

[13:26:43]      => Best score: 130041.3
UnityEngine.Debug:Log (object)

[13:26:43]      => Best genome: [0.8839817, 0.1402875, -0.7008302, -0.08404892, -0.3123308, ]
UnityEngine.Debug:Log (object)

[13:26:43] Generation: 2
UnityEngine.Debug:Log (object)

[13:26:43]      Test: 1
UnityEngine.Debug:Log (object)

[14:01:26]      Test: 2
UnityEngine.Debug:Log (object)

[14:11:52]      Test: 3
UnityEngine.Debug:Log (object)

[14:52:19]      => Best score: 302349.3
UnityEngine.Debug:Log (object)

[14:52:19]      => Best genome: [0.8387001, 0.1402875, -0.7008302, -0.08404892, -0.3123308, ]
UnityEngine.Debug:Log (object)
```


! [14:52:19] Generation: 3
UnityEngine.Debug:Log (object)

! [14:52:19] Test: 1
UnityEngine.Debug:Log (object)

! [15:04:59] Test: 2
UnityEngine.Debug:Log (object)

! [15:13:05] Test: 3
UnityEngine.Debug:Log (object)

! [15:29:59] => Best score: 77755
UnityEngine.Debug:Log (object)

! [15:29:59] => Best genome: [0.8839817, 0.1402875, -0.7008302, -0.06873323, -0.3123308,]
UnityEngine.Debug:Log (object)

! [15:29:59] Generation: 4
UnityEngine.Debug:Log (object)

! [15:29:59] Test: 1
UnityEngine.Debug:Log (object)

! [15:59:04] Test: 2
UnityEngine.Debug:Log (object)

! [16:46:14] Test: 3
UnityEngine.Debug:Log (object)

! [17:31:17] => Best score: 230398
UnityEngine.Debug:Log (object)

! [17:31:17] => Best genome: [0.8839817, 0.1402875, -0.7008302, -0.06873323, -0.3123308,]
UnityEngine.Debug:Log (object)

! [17:31:17] Generation: 5
UnityEngine.Debug:Log (object)

! [17:31:17] Test: 1
UnityEngine.Debug:Log (object)

! [17:44:20] Test: 2
UnityEngine.Debug:Log (object)

! [17:54:33] Test: 3
UnityEngine.Debug:Log (object)

! [18:10:17] => Best score: 99763
UnityEngine.Debug:Log (object)

! [18:10:17] => Best genome: [0.8839817, 0.1183702, -0.7259094, -0.01862722, -0.2787018,]
UnityEngine.Debug:Log (object)

! [18:10:17] Generation: 6
UnityEngine.Debug:Log (object)

! [18:10:17] Test: 1
UnityEngine.Debug:Log (object)

! [18:19:16] Test: 2
UnityEngine.Debug:Log (object)

! [18:46:04] Test: 3
UnityEngine.Debug:Log (object)

! [19:00:49] => Best score: 118898.3
UnityEngine.Debug:Log (object)

! [19:00:49] => Best genome: [0.8839817, 0.1159481, -0.6504662, -0.06873323, -0.3123308,]
UnityEngine.Debug:Log (object)

[19:00:49] Generation: 7
UnityEngine.Debug:Log (object)

[19:00:49] Test: 1
UnityEngine.Debug:Log (object)

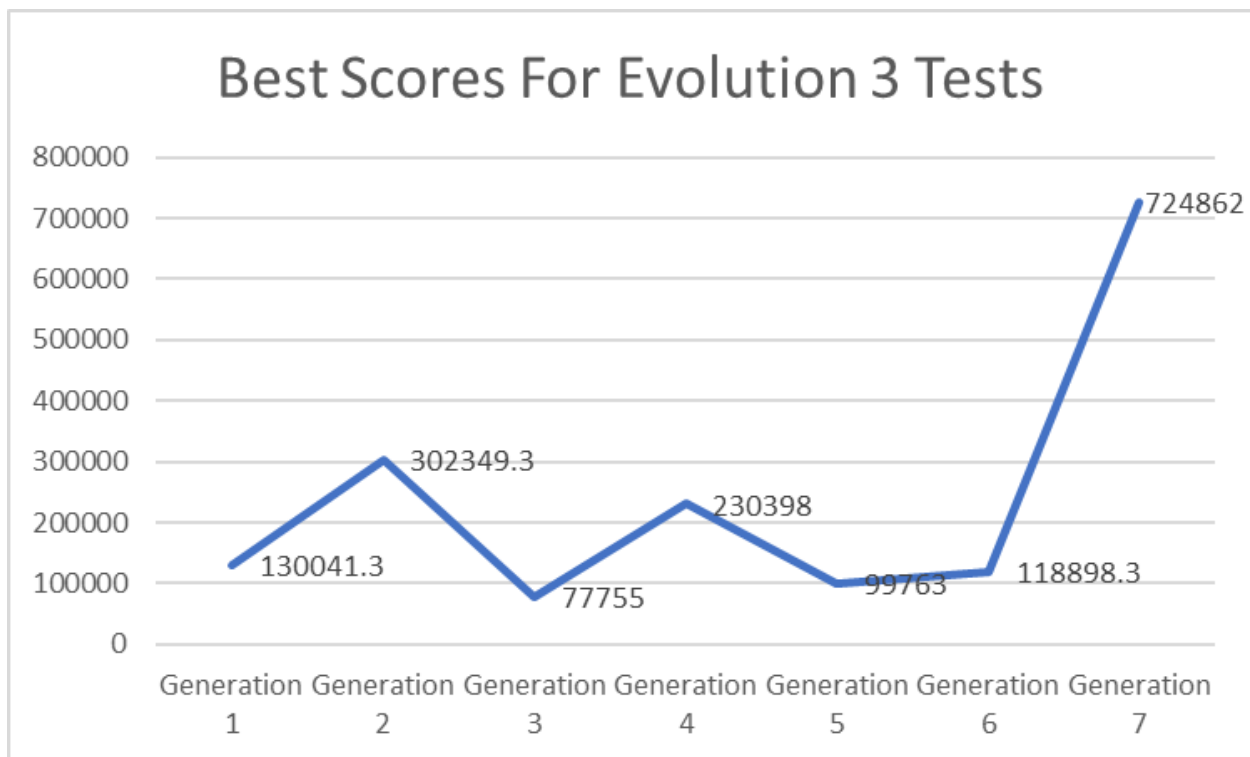
[20:56:04] Test: 2
UnityEngine.Debug:Log (object)

[21:22:16] Test: 3
UnityEngine.Debug:Log (object)

[21:54:16] => Best score: 724862
UnityEngine.Debug:Log (object)

[21:54:16] => Best genome: [0.8839817, 0.1183702, -0.7259094, -0.08404892, -0.2237923,]
UnityEngine.Debug:Log (object)

(Results Of 3rd Time Running Evolution Algorithm)



(Results Of 3rd Evolution Run On Graph)

```

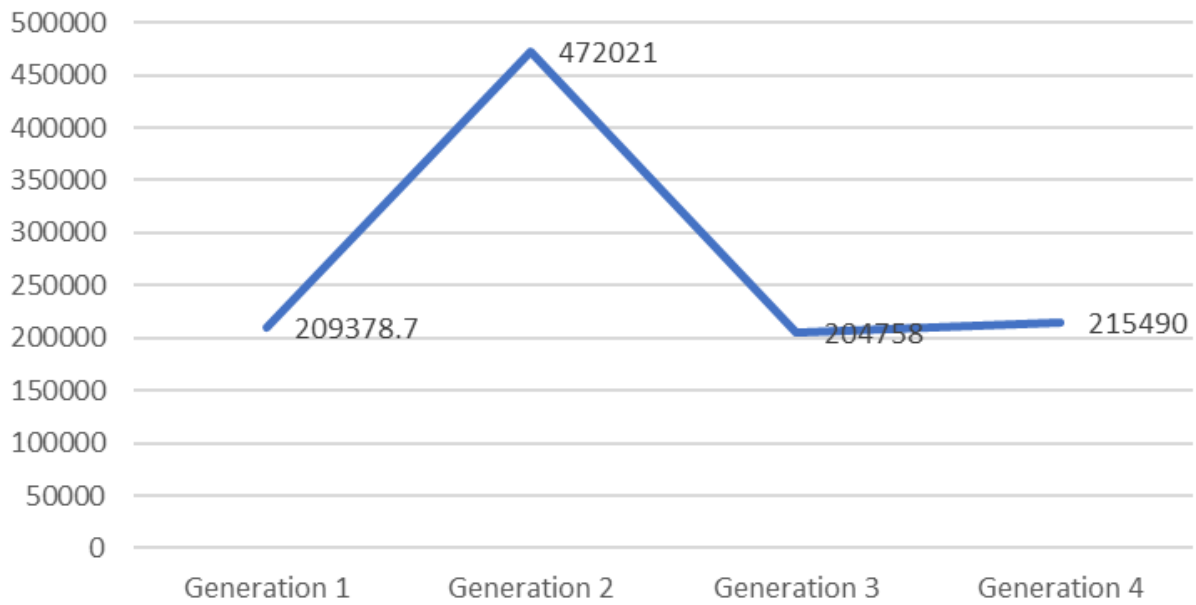
! [14:49:45] Generation: 1
UnityEngine.Debug:Log (object)
! [14:49:45] Test: 1
UnityEngine.Debug:Log (object)
! [15:17:55] Test: 2
UnityEngine.Debug:Log (object)
! [15:32:46] Test: 3
UnityEngine.Debug:Log (object)
! [15:54:42] => Best score: 209378.7
UnityEngine.Debug:Log (object)
! [15:54:42] => Best genome: [0.8839817, 0.134547, -0.7259094, -0.08404892, -0.2237923, ]
UnityEngine.Debug:Log (object)
! [15:54:42] Generation: 2
UnityEngine.Debug:Log (object)
! [15:54:42] Test: 1
UnityEngine.Debug:Log (object)
! [16:24:36] Test: 2
UnityEngine.Debug:Log (object)
! [18:03:08] Test: 3
UnityEngine.Debug:Log (object)
! [18:07:59] => Best score: 472021
UnityEngine.Debug:Log (object)
! [18:07:59] => Best genome: [0.8839817, 0.134547, -0.7259094, -0.08404892, -0.2237923, ]
UnityEngine.Debug:Log (object)
! [18:07:59] Generation: 3
UnityEngine.Debug:Log (object)
! [18:07:59] Test: 1
UnityEngine.Debug:Log (object)
! [18:49:03] Test: 2
UnityEngine.Debug:Log (object)
! [19:17:32] Test: 3
UnityEngine.Debug:Log (object)
! [19:34:26] => Best score: 204758
UnityEngine.Debug:Log (object)

! [19:34:26] Generation: 4
UnityEngine.Debug:Log (object)
! [19:34:26] Test: 1
UnityEngine.Debug:Log (object)
! [19:42:58] Test: 2
UnityEngine.Debug:Log (object)
! [20:14:04] Test: 3
UnityEngine.Debug:Log (object)
! [20:57:24] => Best score: 215490
UnityEngine.Debug:Log (object)
! [20:57:24] => Best genome: [0.8839817, 0.1183702, -0.7259094, -0.08404892, -0.2237923, ]
UnityEngine.Debug:Log (object)

```

(Results Of 4th Time Running Evolution Algorithm)

Best Scores For Evolution 4 Tests



(Results Of 4th Evolution Run On Graph)

! [10:30:33] Generation: 1
UnityEngine.Debug:Log (object)

! [10:30:33] Test: 1
UnityEngine.Debug:Log (object)

! [11:12:38] Test: 2
UnityEngine.Debug:Log (object)

! [11:29:02] Test: 3
UnityEngine.Debug:Log (object)

! [12:13:26] => Best score: 214534.3
UnityEngine.Debug:Log (object)

! [12:13:26] => Best genome: [0.8839817, 0.1183702, -0.7259094, -0.08404892, -0.2237923,]
UnityEngine.Debug:Log (object)

! [12:13:26] Generation: 2
UnityEngine.Debug:Log (object)

! [12:13:26] Test: 1
UnityEngine.Debug:Log (object)

! [12:57:25] Test: 2
UnityEngine.Debug:Log (object)

! [13:18:42] Test: 3
UnityEngine.Debug:Log (object)

! [14:02:49] => Best score: 263239.7
UnityEngine.Debug:Log (object)

! [14:02:49] => Best genome: [0.8839817, 0.1183702, -0.6665653, -0.1112889, -0.2237923,]
UnityEngine.Debug:Log (object)

! [14:02:49] Generation: 3
UnityEngine.Debug:Log (object)

! [14:02:49] Test: 1
UnityEngine.Debug:Log (object)

! [14:34:29] Test: 2
UnityEngine.Debug:Log (object)

! [14:54:14] Test: 3
UnityEngine.Debug:Log (object)

! [15:03:53] => Best score: 140134
UnityEngine.Debug:Log (object)

! [15:03:53] => Best genome: [0.8839817, 0.1183702, -0.6665653, -0.1112889, -0.2237923,]
UnityEngine.Debug:Log (object)

! [15:03:53] Generation: 4
UnityEngine.Debug:Log (object)

! [15:03:53] Test: 1
UnityEngine.Debug:Log (object)

! [15:56:00] Test: 2
UnityEngine.Debug:Log (object)

! [18:27:59] Test: 3
UnityEngine.Debug:Log (object)

! [19:07:27] => Best score: 974640.7
UnityEngine.Debug:Log (object)

! [19:07:27] => Best genome: [0.8839817, 0.1183702, -0.6665653, -0.1419593, -0.2237923,]
UnityEngine.Debug:Log (object)

! [19:07:27] Generation: 5
UnityEngine.Debug:Log (object)

! [19:07:27] Test: 1
UnityEngine.Debug:Log (object)

! [19:38:58] Test: 2
UnityEngine.Debug:Log (object)

! [19:51:42] Test: 3
UnityEngine.Debug:Log (object)

! [20:00:09] => Best score: 190500
UnityEngine.Debug:Log (object)

! [20:00:09] => Best genome: [0.8839817, 0.1898533, -0.6665653, -0.1419593, -0.1262886,]
UnityEngine.Debug:Log (object)

! [20:00:09] Generation: 6
UnityEngine.Debug:Log (object)

! [20:00:09] Test: 1
UnityEngine.Debug:Log (object)

! [20:38:29] Test: 2
UnityEngine.Debug:Log (object)

! [21:56:03] Test: 3
UnityEngine.Debug:Log (object)

! [22:15:04] => Best score: 423378.3
UnityEngine.Debug:Log (object)

! [22:15:04] => Best genome: [0.8839817, 0.1898533, -0.6665653, -0.1419593, -0.1262886,]
UnityEngine.Debug:Log (object)

! [22:15:04] Generation: 7
UnityEngine.Debug:Log (object)

! [22:15:04] Test: 1
UnityEngine.Debug:Log (object)

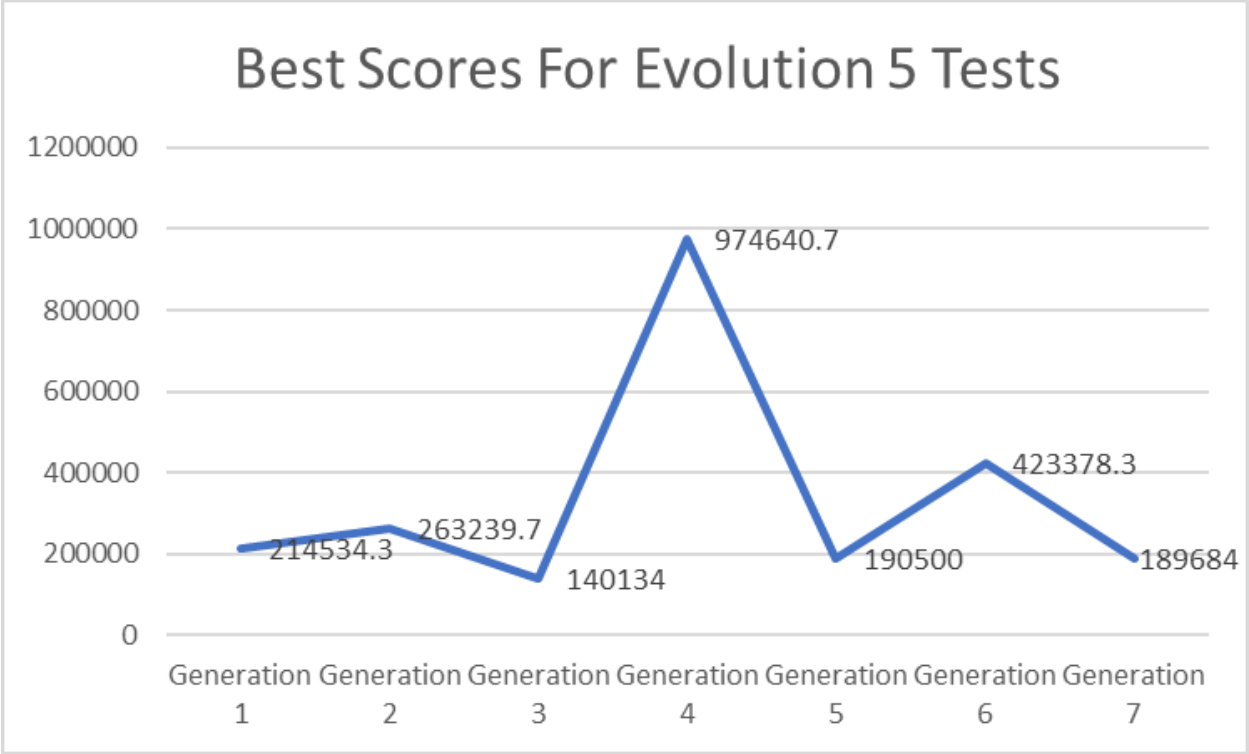
! [22:37:16] Test: 2
UnityEngine.Debug:Log (object)

! [23:17:57] Test: 3
UnityEngine.Debug:Log (object)

! [23:28:17] => Best score: 189684
UnityEngine.Debug:Log (object)

! [23:28:17] => Best genome: [0.8839817, 0.1898533, -0.6665653, -0.1195359, -0.1262886,]
UnityEngine.Debug:Log (object)

(Results Of 5th Time Running Evolution Algorithm)



(Results Of 5th Evolution Run On Graph)

! [11:58:33] Generation: 1
UnityEngine.Debug:Log (object)

! [11:58:33] Test: 1
UnityEngine.Debug:Log (object)

! [11:59:17] Test: 2
UnityEngine.Debug:Log (object)

! [11:59:51] Test: 3
UnityEngine.Debug:Log (object)

! [12:00:35] => Best score: 3105.667
UnityEngine.Debug:Log (object)

! [12:00:35] => Best genome: [0.8839817, 0.1898533, 0.6665653, -0.1195359, -0.1262886,]
UnityEngine.Debug:Log (object)

! [12:00:35] Generation: 2
UnityEngine.Debug:Log (object)

! [12:00:35] Test: 1
UnityEngine.Debug:Log (object)

! [12:01:22] Test: 2
UnityEngine.Debug:Log (object)

! [12:02:14] Test: 3
UnityEngine.Debug:Log (object)

! [12:08:03] => Best score: 40905.33
UnityEngine.Debug:Log (object)

! [12:08:03] => Best genome: [0.823316, 0.1898533, 0.6665653, -0.8547227, -0.1262886,]
UnityEngine.Debug:Log (object)

! [12:08:03] Generation: 3
UnityEngine.Debug:Log (object)

! [12:08:03] Test: 1
UnityEngine.Debug:Log (object)

! [12:09:10] Test: 2
UnityEngine.Debug:Log (object)

! [12:09:43] Test: 3
UnityEngine.Debug:Log (object)

! [12:10:24] => Best score: 8130.333
UnityEngine.Debug:Log (object)

! [12:10:24] => Best genome: [0.823316, 0.1898533, 0.6665653, -0.8547227, -0.1262886,]
UnityEngine.Debug:Log (object)

[12:10:24] Generation: 4
UnityEngine.Debug:Log (object)

[12:10:24] Test: 1
UnityEngine.Debug:Log (object)

[12:11:34] Test: 2
UnityEngine.Debug:Log (object)

[12:14:05] Test: 3
UnityEngine.Debug:Log (object)

[12:16:44] => Best score: 15699
UnityEngine.Debug:Log (object)

[12:16:44] => Best genome: [0.823316, 0.1898533, 0.6665653, -0.8547227, -0.1262886,]
UnityEngine.Debug:Log (object)

[12:16:44] Generation: 5
UnityEngine.Debug:Log (object)

[12:16:44] Test: 1
UnityEngine.Debug:Log (object)

[12:20:01] Test: 2
UnityEngine.Debug:Log (object)

[12:34:33] Test: 3
UnityEngine.Debug:Log (object)

[12:40:32] => Best score: 99445.34
UnityEngine.Debug:Log (object)

[12:40:32] => Best genome: [0.823316, 0.1898533, 0.6665653, -0.8547227, -0.1262886,]
UnityEngine.Debug:Log (object)

[12:40:32] Generation: 6
UnityEngine.Debug:Log (object)

[12:40:32] Test: 1
UnityEngine.Debug:Log (object)

[12:54:24] Test: 2
UnityEngine.Debug:Log (object)

[13:06:18] Test: 3
UnityEngine.Debug:Log (object)

[13:09:56] => Best score: 77301
UnityEngine.Debug:Log (object)

[13:09:56] => Best genome: [0.823316, 0.1898533, 0.6665653, -0.7923605, -0.1262886,]
UnityEngine.Debug:Log (object)

! [13:09:56] Generation: 7
UnityEngine.Debug:Log (object)

! [13:09:56] Test: 1
UnityEngine.Debug:Log (object)

! [13:13:38] Test: 2
UnityEngine.Debug:Log (object)

! [13:22:41] Test: 3
UnityEngine.Debug:Log (object)

! [13:30:47] => Best score: 42387
UnityEngine.Debug:Log (object)

! [13:30:47] => Best genome: [0.823316, 0.1898533, 0.6665653, -0.8547227, -0.02825101,]
UnityEngine.Debug:Log (object)

! [13:30:47] Generation: 8
UnityEngine.Debug:Log (object)

! [13:30:47] Test: 1
UnityEngine.Debug:Log (object)

! [13:37:11] Test: 2
UnityEngine.Debug:Log (object)

! [13:46:28] Test: 3
UnityEngine.Debug:Log (object)

! [13:51:19] => Best score: 45523.33
UnityEngine.Debug:Log (object)

! [13:51:19] => Best genome: [0.823316, 0.1898533, 0.5828078, -0.9377805, -0.02825101,]
UnityEngine.Debug:Log (object)

! [13:51:19] Generation: 9
UnityEngine.Debug:Log (object)

! [13:51:19] Test: 1
UnityEngine.Debug:Log (object)

! [13:56:39] Test: 2
UnityEngine.Debug:Log (object)

! [14:09:20] Test: 3
UnityEngine.Debug:Log (object)

! [14:25:38] => Best score: 98158.66
UnityEngine.Debug:Log (object)

! [14:25:38] => Best genome: [0.823316, 0.1898533, 0.5828078, -0.9377805, -0.02825101,]
UnityEngine.Debug:Log (object)

! [14:25:38] Generation: 10
UnityEngine.Debug:Log (object)

! [14:25:38] Test: 1
UnityEngine.Debug:Log (object)

! [14:45:09] Test: 2
UnityEngine.Debug:Log (object)

! [14:56:05] Test: 3
UnityEngine.Debug:Log (object)

! [15:02:32] => Best score: 81063.66
UnityEngine.Debug:Log (object)

! [15:02:32] => Best genome: [0.823316, 0.1839059, 0.6665653, -0.8547227, -0.02825101,]
UnityEngine.Debug:Log (object)

[15:02:32] Generation: 11
UnityEngine.Debug:Log (object)

[15:02:32] Test: 1
UnityEngine.Debug:Log (object)

[15:11:54] Test: 2
UnityEngine.Debug:Log (object)

[15:35:13] Test: 3
UnityEngine.Debug:Log (object)

[15:44:15] => Best score: 122368
UnityEngine.Debug:Log (object)

[15:44:15] => Best genome: [0.823316, 0.1898533, 0.5828078, -0.9377805, -0.02825101,]
UnityEngine.Debug:Log (object)

[15:44:15] Generation: 12
UnityEngine.Debug:Log (object)

[15:44:15] Test: 1
UnityEngine.Debug:Log (object)

[15:57:09] Test: 2
UnityEngine.Debug:Log (object)

[16:19:09] Test: 3
UnityEngine.Debug:Log (object)

[16:33:31] => Best score: 122837.3
UnityEngine.Debug:Log (object)

[16:33:31] => Best genome: [0.823316, 0.1839059, 0.6665653, -0.8281465, -0.02825101,]
UnityEngine.Debug:Log (object)

[16:33:31] Generation: 13
UnityEngine.Debug:Log (object)

[16:33:31] Test: 1
UnityEngine.Debug:Log (object)

[16:41:05] Test: 2
UnityEngine.Debug:Log (object)

[16:52:34] Test: 3
UnityEngine.Debug:Log (object)

[17:07:01] => Best score: 73843.34
UnityEngine.Debug:Log (object)

[17:07:01] => Best genome: [0.823316, 0.1839059, 0.2597177, -0.8281465, -0.02825101,]
UnityEngine.Debug:Log (object)

[17:07:01] Generation: 14
UnityEngine.Debug:Log (object)

[17:07:01] Test: 1
UnityEngine.Debug:Log (object)

[17:50:55] Test: 2
UnityEngine.Debug:Log (object)

[18:18:26] Test: 3
UnityEngine.Debug:Log (object)

[18:47:37] => Best score: 287172
UnityEngine.Debug:Log (object)

```
[18:47:37] Generation: 15
UnityEngine.Debug:Log (object)

[18:47:37]      Test: 1
UnityEngine.Debug:Log (object)

[18:59:09]      Test: 2
UnityEngine.Debug:Log (object)

[19:24:24]      Test: 3
UnityEngine.Debug:Log (object)

[19:37:16]      => Best score: 84657.66
UnityEngine.Debug:Log (object)

[19:37:16]      => Best genome: [0.7804424, 0.1898533, 0.6336566, -0.9377805, -0.02825101, ]
UnityEngine.Debug:Log (object)

[19:37:16] Generation: 16
UnityEngine.Debug:Log (object)

[19:37:16]      Test: 1
UnityEngine.Debug:Log (object)

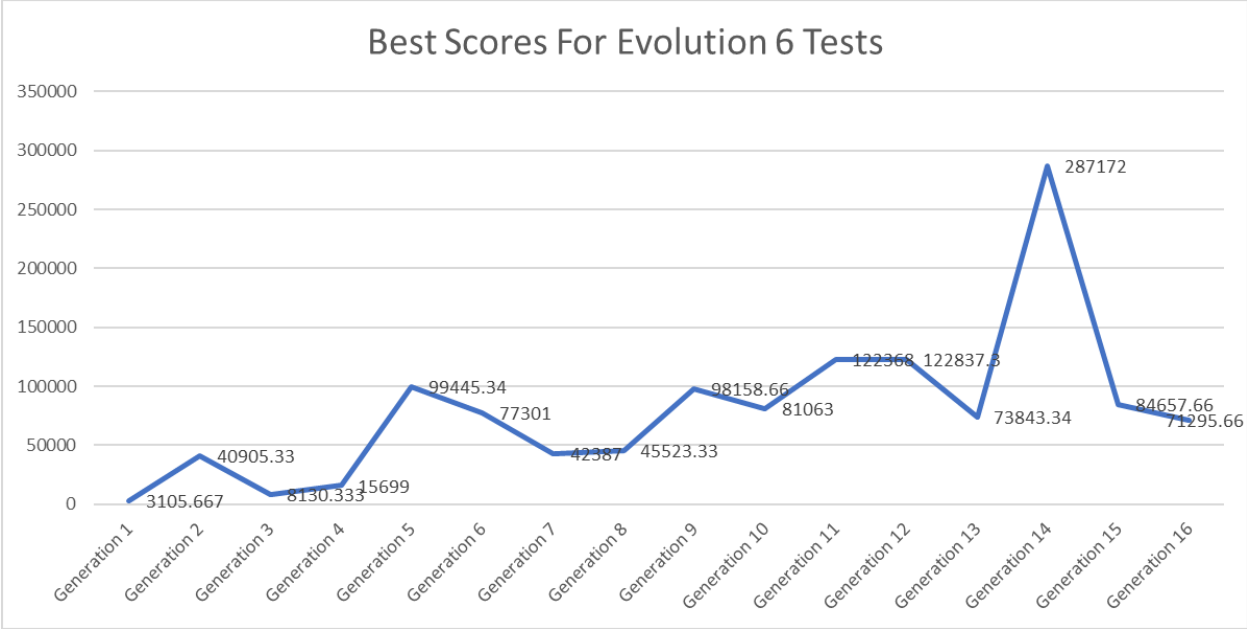
[19:52:23]      Test: 2
UnityEngine.Debug:Log (object)

[20:01:36]      Test: 3
UnityEngine.Debug:Log (object)

[20:14:26]      => Best score: 71295.66
UnityEngine.Debug:Log (object)

[20:14:26]      => Best genome: [0.823316, 0.1839059, 0.2597177, -0.8281465, -0.02825101, ]
UnityEngine.Debug:Log (object)
```

(Results Of 6th Time Running Evolution Algorithm)



(Results Of 6th Run On Graph)

[13:03:44] Generation: 1
UnityEngine.Debug:Log (object)

[13:03:44] Test: 1
UnityEngine.Debug:Log (object)

[13:26:58] Test: 2
UnityEngine.Debug:Log (object)

[14:03:25] Test: 3
UnityEngine.Debug:Log (object)

[14:20:46] => Best score: 145770
UnityEngine.Debug:Log (object)

[14:20:46] => Best genome: [0.823316, 0.1839059, 0.2597177, -0.8281465, -0.02825101,]
UnityEngine.Debug:Log (object)

[14:20:46] Generation: 2
UnityEngine.Debug:Log (object)

[14:20:46] Test: 1
UnityEngine.Debug:Log (object)

[14:56:44] Test: 2
UnityEngine.Debug:Log (object)

[15:14:14] Test: 3
UnityEngine.Debug:Log (object)

[15:50:32] => Best score: 190404
UnityEngine.Debug:Log (object)

[15:50:32] => Best genome: [0.823316, 0.1839059, 0.2597177, -0.7791536, -0.02825101,]
UnityEngine.Debug:Log (object)

[15:50:32] Generation: 3
UnityEngine.Debug:Log (object)

[15:50:32] Test: 1
UnityEngine.Debug:Log (object)

[16:57:27] Test: 2
UnityEngine.Debug:Log (object)

[17:48:09] Test: 3
UnityEngine.Debug:Log (object)

[18:48:17] => Best score: 339805.7
UnityEngine.Debug:Log (object)

[18:48:17] => Best genome: [0.823316, 0.1839059, 0.2597177, -0.7791536, -0.02825101,]
UnityEngine.Debug:Log (object)

[18:48:17] Generation: 4
UnityEngine.Debug:Log (object)

[18:48:17] Test: 1
UnityEngine.Debug:Log (object)

[19:08:26] Test: 2
UnityEngine.Debug:Log (object)

[19:47:55] Test: 3
UnityEngine.Debug:Log (object)

[20:09:17] => Best score: 179224.3
UnityEngine.Debug:Log (object)

[20:09:17] => Best genome: [0.823316, 0.2685622, 0.2597177, -0.7791536, -0.02825101,]
UnityEngine.Debug:Log (object)

[20:09:17] Generation: 5
UnityEngine.Debug:Log (object)

[20:09:17] Test: 1
UnityEngine.Debug:Log (object)

[20:46:05] Test: 2
UnityEngine.Debug:Log (object)

[21:04:40] Test: 3
UnityEngine.Debug:Log (object)

[21:32:20] => Best score: 147992.3
UnityEngine.Debug:Log (object)

[21:32:20] => Best genome: [0.823316, 0.1839059, 0.2597177, -0.9808854, -0.02825101,]
UnityEngine.Debug:Log (object)

[21:32:20] Generation: 6
UnityEngine.Debug:Log (object)

[21:32:20] Test: 1
UnityEngine.Debug:Log (object)

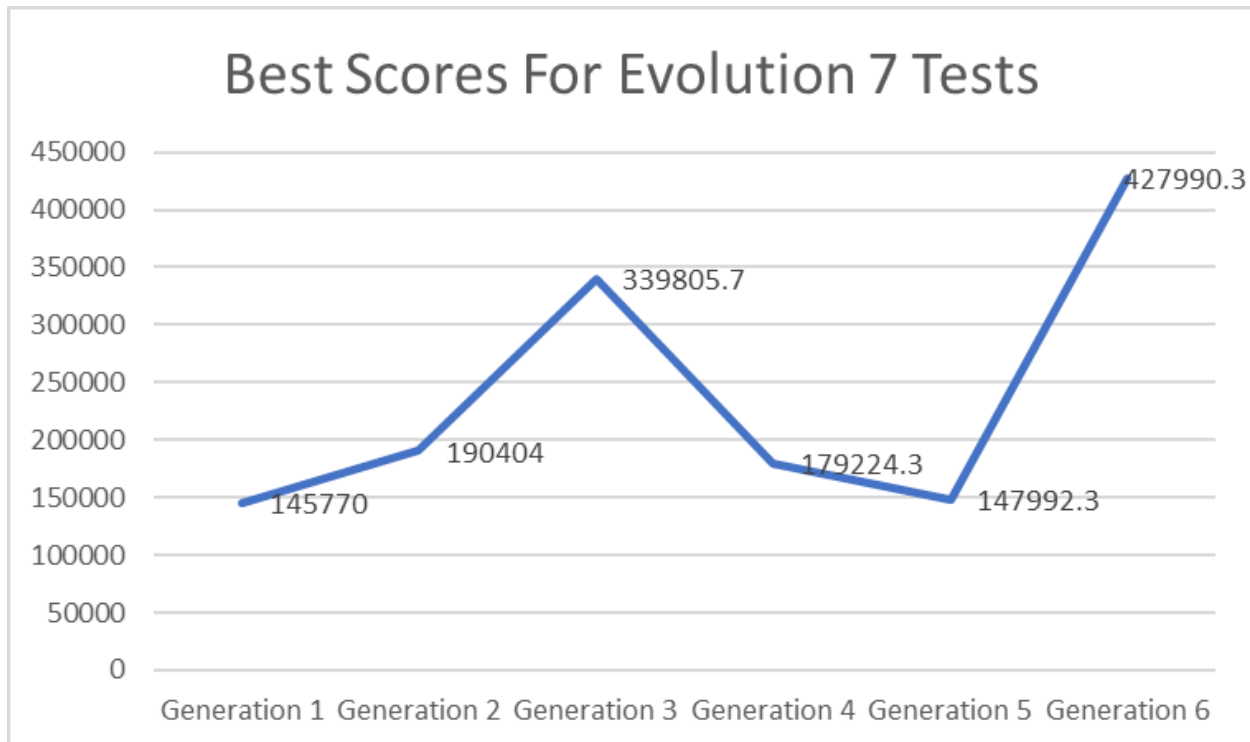
[23:27:20] Test: 2
UnityEngine.Debug:Log (object)

[00:34:37] Test: 3
UnityEngine.Debug:Log (object)

[00:53:42] => Best score: 427990.3
UnityEngine.Debug:Log (object)

[00:53:42] => Best genome: [0.823316, 0.1839059, 0.2597177, -0.9808854, -0.02825101,]
UnityEngine.Debug:Log (object)

(Results Of 7th Time Running Evolution Algorithm)



(Results Of 7th Evolution Run On Graph)

! [10:20:04] Generation: 1
UnityEngine.Debug:Log (object)

! [10:20:04] Test: 1
UnityEngine.Debug:Log (object)

! [10:58:41] Test: 2
UnityEngine.Debug:Log (object)

! [11:35:13] Test: 3
UnityEngine.Debug:Log (object)

! [12:44:03] => Best score: 374710
UnityEngine.Debug:Log (object)

! [12:44:03] => Best genome: [0.823316, 0.1839059, 0.2597177, -0.9808854, -0.02825101,]
UnityEngine.Debug:Log (object)

! [12:44:03] Generation: 2
UnityEngine.Debug:Log (object)

! [12:44:03] Test: 1
UnityEngine.Debug:Log (object)

! [13:01:19] Test: 2
UnityEngine.Debug:Log (object)

! [13:29:19] Test: 3
UnityEngine.Debug:Log (object)

! [14:19:55] => Best score: 251709.3
UnityEngine.Debug:Log (object)

! [14:19:55] => Best genome: [0.823316, 0.1839059, 0.2597177, -0.9808854, -0.04233827,]
UnityEngine.Debug:Log (object)

! [14:19:55] Generation: 3
UnityEngine.Debug:Log (object)

! [14:19:55] Test: 1
UnityEngine.Debug:Log (object)

! [14:47:13] Test: 2
UnityEngine.Debug:Log (object)

! [15:28:16] Test: 3
UnityEngine.Debug:Log (object)

! [16:34:07] => Best score: 333455
UnityEngine.Debug:Log (object)

! [16:34:07] => Best genome: [0.8568326, 0.1839059, 0.2597177, -0.9808854, -0.02825101,]
UnityEngine.Debug:Log (object)

! [16:34:07] Generation: 4
UnityEngine.Debug:Log (object)

! [16:34:07] Test: 1
UnityEngine.Debug:Log (object)

! [18:14:09] Test: 2
UnityEngine.Debug:Log (object)

! [20:26:15] Test: 3
UnityEngine.Debug:Log (object)

! [20:53:26] => Best score: 600379.7
UnityEngine.Debug:Log (object)

! [20:53:26] => Best genome: [0.83995, 0.1839059, 0.2597177, -1, -0.04233827,]
UnityEngine.Debug:Log (object)

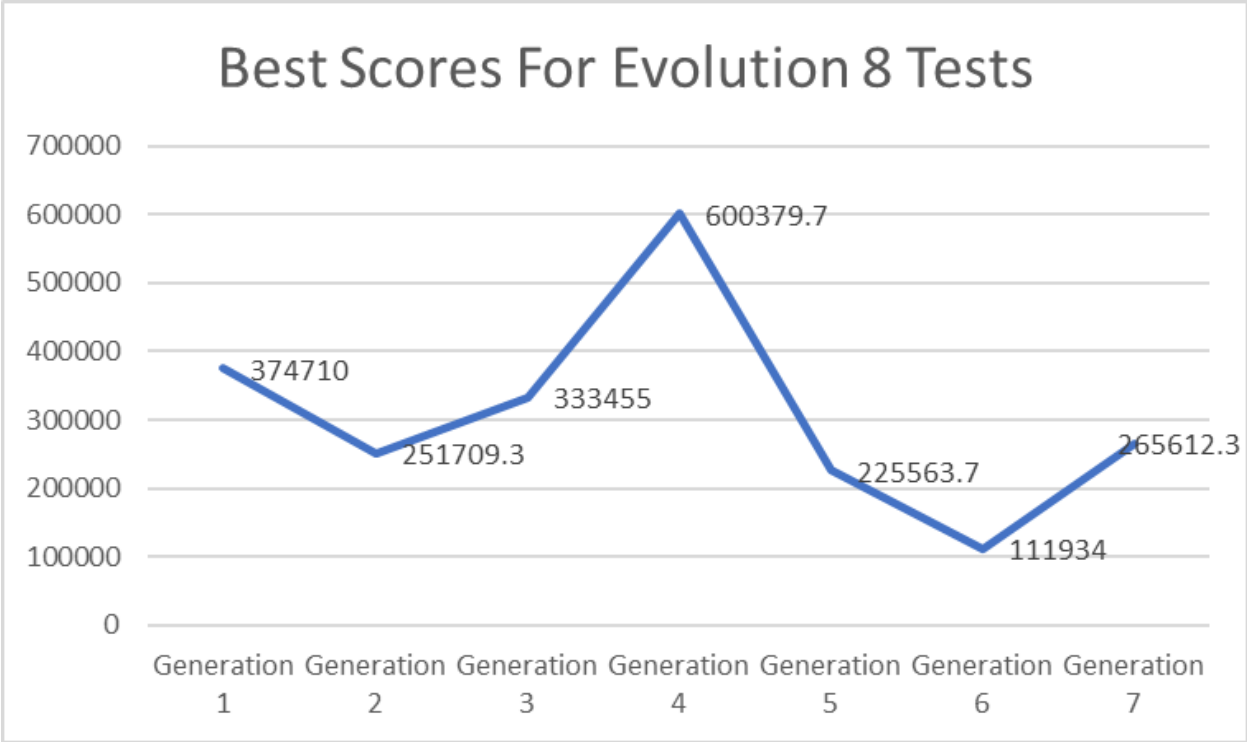
```

! [20:53:26] Generation: 5
  UnityEngine.Debug:Log (object)
! [20:53:26]      Test: 1
  UnityEngine.Debug:Log (object)
! [21:11:50]      Test: 2
  UnityEngine.Debug:Log (object)
! [22:13:38]      Test: 3
  UnityEngine.Debug:Log (object)
! [23:03:42]      => Best score: 225563.7
  UnityEngine.Debug:Log (object)
! [23:03:42]      => Best genome: [0.83995, 0.1839059, 0.2597177, -1, -0.04233827,]
  UnityEngine.Debug:Log (object)
! [23:03:42] Generation: 6
  UnityEngine.Debug:Log (object)
! [23:03:42]      Test: 1
  UnityEngine.Debug:Log (object)
! [23:18:49]      Test: 2
  UnityEngine.Debug:Log (object)
! [23:44:31]      Test: 3
  UnityEngine.Debug:Log (object)
! [00:10:20]      => Best score: 111934
  UnityEngine.Debug:Log (object)
! [00:10:20]      => Best genome: [0.8568326, 0.7596295, 0.2597177, -0.9306839, -0.02825101,]
  UnityEngine.Debug:Log (object)

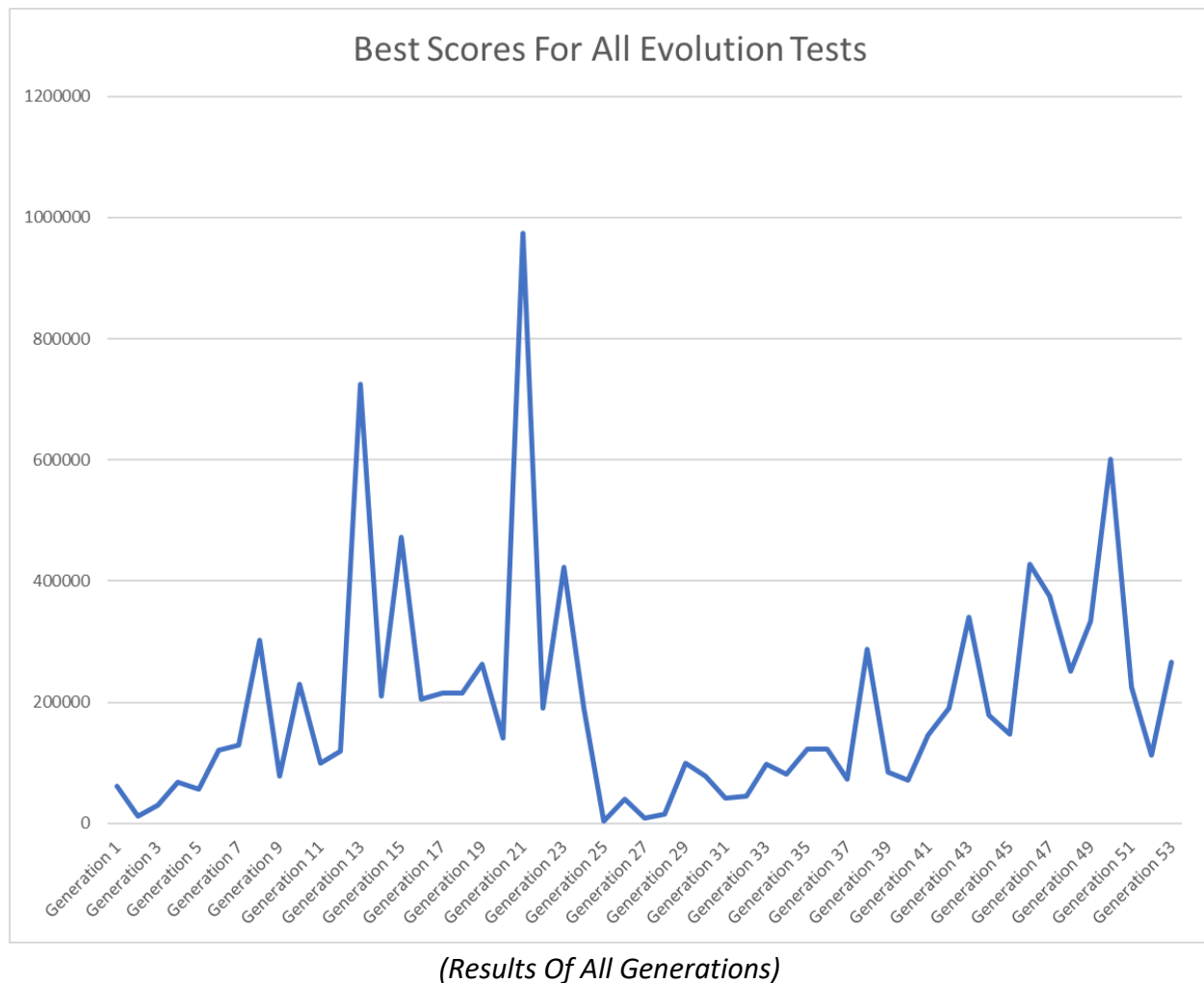
! [00:10:20] Generation: 7
  UnityEngine.Debug:Log (object)
! [00:10:20]      Test: 1
  UnityEngine.Debug:Log (object)
! [00:53:17]      Test: 2
  UnityEngine.Debug:Log (object)
! [01:52:21]      Test: 3
  UnityEngine.Debug:Log (object)
! [02:11:53]      => Best score: 265612.3
  UnityEngine.Debug:Log (object)
! [02:11:53]      => Best genome: [0.83995, 0.2411025, 0.2597177, -1, 0.02577379,]
  UnityEngine.Debug:Log (object)

```

(Results Of 8th Time Running Evolution Algorithm)



(Results Of 8th Evolution Run On Graph)

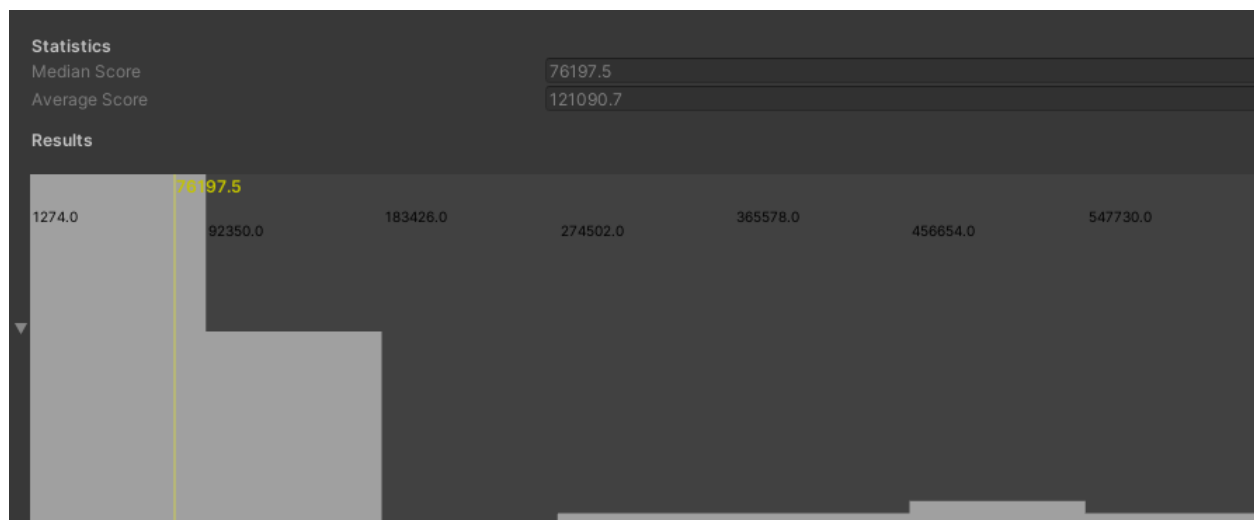


The results above show that the Tetris AI performs bad then starts to perform better and then does the same cycle again but does not perform as bad. This suggests that if the evolution algorithm was to be run for a lot longer then the performance should level off where it would be performing consistently good and producing similar scores constantly. The large spike that is close to one million could be an anomalous result where the Tetris AI in that generation got extremely lucky and performed well in the three tests that were done which is quite unlikely. Furthermore, this spike could suggest that the Tetris AI could perform at that level consistently if the values the factors are being multiplied by were the perfect values. Therefore, if I was able to run my system for a lot longer it might be able to reach that one million score consistently but due to the time limit on the coursework, I had to stop the evolution algorithm running once it hit generation 53 otherwise, I would not have enough time to complete the coursework. Finally the best score in generation 53 has a -1 in its genome which is suggesting that the factor that the -1 is being multiplied by is not a good factor for my AI since with it is performing worse

since -1 is multiplied by 10 and has an addition of 10 making it 0 meaning that the factor that is being multiplied with 0 is a bad factor which is potentially causing my Tetris AI to perform worse due to the factors existence. Therefore, by using this method and running this algorithm it has helped me figure out bad factors that contribute to my heuristic causing the Tetris AI to make bad decisions and perform badly. Additionally, this algorithm has helped me get the best genome for the Tetris AI so it can perform better compared to the method I have used during iteration 1 - 5 where I used utility theory and assess the most important factors and assign them values accordingly.

Iteration 8

During iteration 8 I have taken the best genome from my evolution algorithm which is the result from generation 53 and implemented it into my Tetris AI after multiplying each value by 10 and adding each value by 10. My cost calculation is $\text{float cost} = (\text{holes} * (\text{float})18.3995) + (\text{float})12.41025 * \text{emptyCells} + ((\text{float})12.597117 * \text{MaxHeight}) + \text{Bumpiness} * (\text{float})0 + \text{Pillar} * (\text{float})10.2577379$; This cost calculation had two typos within it where inputted the numbers incorrectly: the typos are the multiplier for empty cells where it should have been 12.411025 and max height where the multiplier should have been 12.597177. I did not realise the two mistakes within my cost calculation and subsequently ran a test of 50 simulations. Then I realised the mistakes.



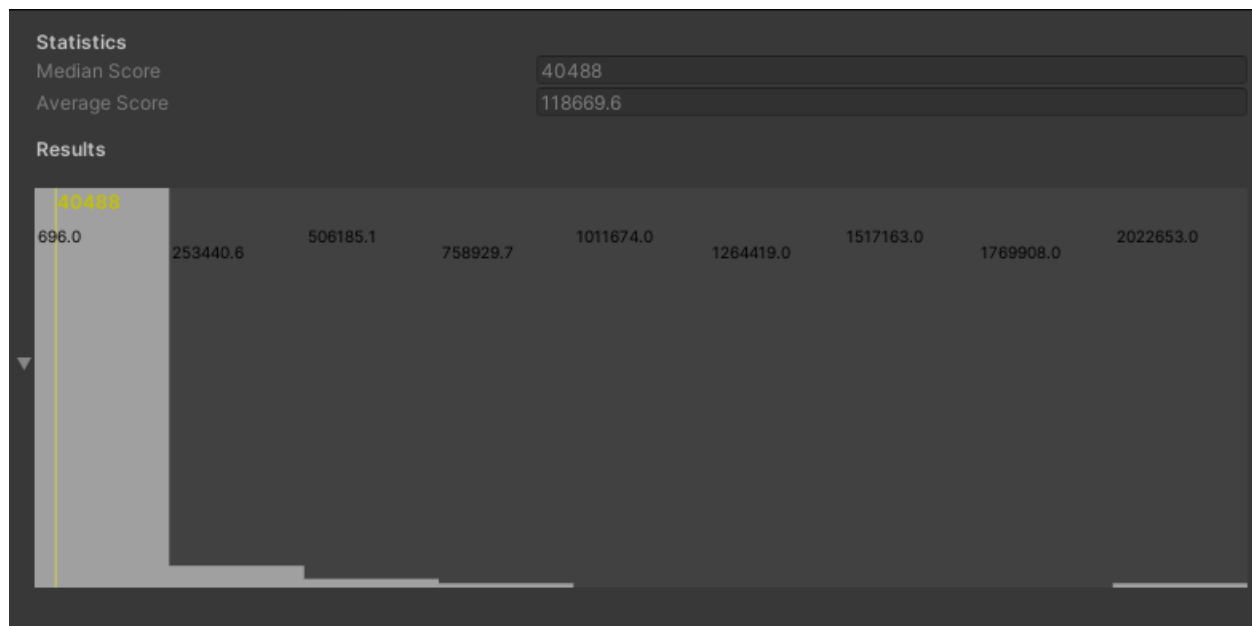
(Results of 50 Simulations With Typos)

The results above show a very good increase in the median score and average score compared to the results my AI was giving before I ran the evolution algorithm on it, suggesting that by using the evolution algorithm it has had a great impact and has allowed my median score to be

closer to my average score compared to the results in iteration 5 meaning that the evolution algorithm has helped it with the consistency of the scores it is achieving as well. However, the results may have been due to the typos within the cost calculation, so I'll be running 100 simulations for the cost calculation with the typo and without the typo to see whether it was beneficial or not.

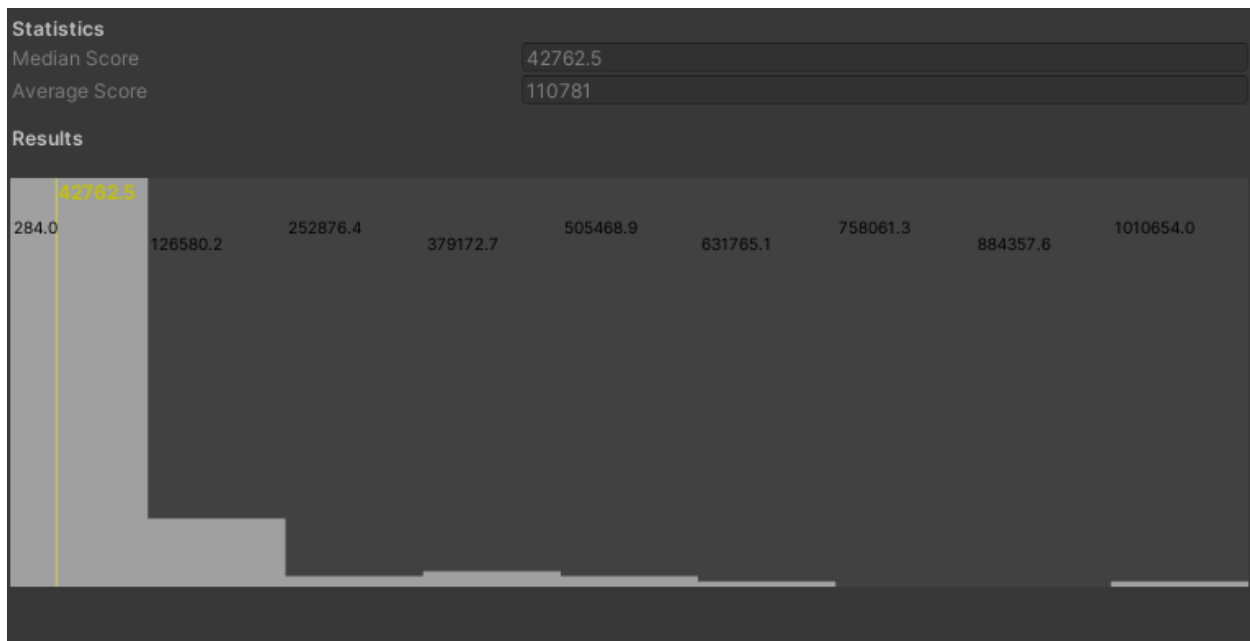
Final Iteration With Final Test Results

During my final iteration I ran two 100 simulations one for the cost calculation with the typos and the other for the correct cost calculation which had the best genome as the factor multipliers.



(Test 1 - 100 Simulation Test With Typo Within The Factor Multipliers)

My results from test 1 show a very large spread with the score the Tetris AI can achieve which goes from 696 to 2 million score. Suggesting that the typo was not beneficial since it is very inconsistent with its results. However, the extremely low and high scores could have been anomalies that caused a very negative impact on the median score and the average of the Tetris AI.



(Test 2 - 100 Simulation Test Without Typo Within The Factor Multipliers)

My results from test 2 show inconsistency with its scores as its lowest score was 284 which could be an anomaly and its highest score was over a million which could also be an anomaly but compared to the results from test 1 this has a better median suggesting that the scores it is achieving is more consistent meaning that this Tetris AI is performing better hence meaning that the AI with the cost calculation that does not have typos is the better performing AI. Also, within the results of test 2 it has achieved a higher median score than the results of test 1 but its average is lower meaning it is not able to achieve very high scores but can get more consistent scores which is better for an AI as you would want it to be consistent instead of inconsistent. Therefore, the Tetris AI from test 2 which uses the correct factor multipliers is the better performing AI.

The code for the correct cost calculation is :

```
public float Heuristic(Move move)
{
    float Pillar = pillar(move);
    float holes = Holes(move);
    float emptyCells = EmptyCells(move);
    float MaxHeight = maxHeight(move);
    float Bumpiness = bumpiness(move);

    float cost = (holes * (float)18.3995) + (float)12.411025 * emptyCells + ((float)12.597177 *
```

```

MaxHeight) + Bumpiness * (float)0 + Pillar * (float)10.2577379;

    return cost;
}

```

Finally, by using an evolution algorithm for 53 generations it has improved my Tetris AI's performance drastically since it was only averaging 30888.08 and had a median of 15860 in iteration 5 where I made use of utility theory. Also, the AI's new results compared to the old results suggests that if I was to carry on running the evolution algorithm it could cause my AI to perform even better and become more consistent as well.

Final Tetris AI Code

```

using UnityEngine;
using AlanZucconi.Tetris;
using AlanZucconi.AI.Evo;
[CreateAssetMenu
(
    fileName = "TetrisAI_oguve001",
    menuName = "Tetris/2021-22/TetrisAI_oguve001"
)]
public class TetrisAI_oguve001 : TetrisAI
{
    //values the evolution algorithm assigns are saved in these variables
    //used during the evolution algorithm
    public float x;
    public float c;
    public float v;
    public float b;
    public float n;

    //counts number of empty cells on the board
    public float EmptyCells(Move move)

```



```

{
    int emptyCells = 0;
    int min = Tetris.State.GetMaxColumnHeight();
    // Simulates the effect of the move on the board
    TetrisState state = Tetris.SimulateMove(move);
    for (int x = 0; x < Tetris.Size.x; x++)
    {
        if (min > state.GetColumnHeight(x))
        {
            min = state.GetColumnHeight(x);
        }

        for (int y = 0; y < state.GetColumnHeight(x); y++)
        {
            if (state.IsEmpty(x, y))
            {
                emptyCells++;
            }
        }
    }
    return emptyCells;
}

```

//finds bumpiness by finding the difference between column height before move simulation and after move simulation

```

public float bumpiness(Move move)
{
    int min = Tetris.State.GetMaxColumnHeight();
    // Simulates the effect of the move on the board
    TetrisState state = Tetris.SimulateMove(move);
    int bumpiness = state.GetMaxColumnHeight() - min;
    return bumpiness;
}

```

//calculates and returns maximum height

```

public float maxHeight(Move move)
{

```

```

    int maxHeight = 0;
    int max = 0;
    // Simulates the effect of the move on the board
    TetrisState state = Tetris.SimulateMove(move);
    for (int x = 0; x < Tetris.Size.x; x++)
    {
        if (state.GetColumnHeight(x) > max)
        {
            max = state.GetColumnHeight(x);
        }
        int height = state.GetColumnHeight(x);
        if (height > maxHeight)
        {
            maxHeight = height;
        }
    }
    maxHeight = maxHeight - Tetris.State.GetMaxColumnHeight();
    return maxHeight;
}

//calculates and returns number of holes the tetris blocks have created
public float Holes(Move move)
{
    int holes = 0;
    // Simulates the effect of the move on the board
    TetrisState state = Tetris.SimulateMove(move);
    for (int j = 0; j < Tetris.Size.x; j++)
    {
        for (int y = 0; y < state.GetColumnHeight(j); y++)
        {
            if (state.IsEmpty(j, y))
            {
                holes++;
            }
        }
    }
}

```

```
}  
    return holes;  
}
```

//calculates and returns number of spaces between the piece that is being placed
indicating that it is causing a pillar

```
public float pillar(Move move)  
{  
    // Simulates the effect of the move on the board  
    TetrisState state = Tetris.SimulateMove(move);  
    int counter = 0;  
  
    for (int x = 0; x < Tetris.Size.x; x++)  
    {  
        for (int i = Tetris.Size.y - 1; i >= 0; i--)  
        {  
            if (state.IsFull(x, i))  
                break;  
  
            if (state.IsFull(x - 1, i) && state.IsFull(x + 1, i))  
            {  
                counter++;  
            }  
        }  
    }  
    return counter;  
}
```

//calculate cost of each move

```
public float Heuristic(Move move)  
{  
    //call all the functions  
    float Pillar = pillar(move);  
    float holes = Holes(move);  
    float emptyCells = EmptyCells(move);
```

```

float MaxHeight = maxHeight(move);
float Bumpiness = bumpiness(move);

//values the evolution algorithm made are used here to calculate the cost.
//float cost = (holes * (float)x) + (float)c * emptyCells + ((float)v * MaxHeight) +
Bumpiness * (float)b + Pillar * (float)n;

//calculate cost of the move using the factors returns and the multipliers
float cost = (holes * (float)18.3995) + (float)12.411025 * emptyCells + ((float)12.597177 *
MaxHeight) + Bumpiness * (float)0 + Pillar * (float)10.2577379;

//return the cost of the move
return cost;
}
public override int ChooseMove(Move[] moves)
{
    //perform the move with the lowest cost
    return moves.IndexOfMin(move => Heuristic(move));
}
}

```

Conclusion

Overall, I believe my Tetris AI is performing to a good standard and with more time it could have done better since I would have been able to make more changes and have the evolution algorithm run for a lot longer. The techniques I used at the beginning such as utility theory was helpful as it helped me figure out if the code for the AI needs to be worked on or the multipliers to the factors required changing. Without this technique it would have been hard to determine if the factors that influenced decision making was the problem or the factor multipliers that influenced the cost was the problem. Another technique I used was the evolution algorithm which I ran for 53 generations. It has improved my Tetris AI's performance drastically since it was only averaging 30888.08 and had a median of 15860 in 50 simulations in iteration 5 where I mainly used utility theory compared to the results now where the Tetris AI is averaging 110781 and has a median of 42762.5 showing immense improvement. One of my problems was my bumpiness factor which had a negative impact on the AI's performance. This is because whilst I

was running the evolution algorithm it started assigning the bumpiness factor a parameter of -1 which when put into the system is a 0 meaning that without the bumpiness factor the AI would perform better. Hence by running the algorithm I was able to find this out which in theory has helped me as well.

What worked well was the set of factors as it still allowed the AI to perform at a good standard where it has a high average and median. But the average is more than double the median which suggests that the AI is not getting consistently similar results and may be dying too early from time to time and living for a long time occasionally where it is close to one million points causing the average to be more than double the median. Which is one of the problems with my AI. To solve this problem, I could work more on the factors that are influencing the decision of the AI and add new factors such as stop the AI from clearing single rows constantly and have it clear multiple rows at a time causing it to create more space on the board in a single turn as it is clearing multiple rows at a time. Another way I could solve this problem is by running the generation algorithm for a lot longer until it starts to give consistent results within the output of each generation. But overall, I believe my AI is performing at a good standard and is achieving a good average and median which can be improved without changing the factors and only changing the values the factors are being multiplied.