CS3243 Project Report | Group 04 | Arnold Christopher Koroa A0092101Y, Dawin Widjaja A0075097W, Ivan Reinaldo A0075113R, Muhammad Omer Iqbal A0074933Y, Samantha Wong Shin Nee A0088359L

Strategy

We had implemented several strategies for this project. We will provide a comparative analysis of them.

Non-Utility Based Selection
Our first strategy attempted to mimic the human process of choosing where to place pieces, with no state enumeration. It involved simply fitting pieces if a fit[1] could be found. If more than one fit could be found, the fit that resulted in a lower overall height would be selected - or if both resulted in the same maximum height, the selection with the smallest difference in height between adjacent columns was selected. We soon realized that there were many instances when no perfect fit could be found - and while it is possible to logically encode the decision process of deciding where to put the piece based on a ranked set of priorities on the various parameters of the board - we decided it was less feasible to code than a utility-based approach. As is, this strategy clears a range of 0 - 12 rows.

Utility Feature Based Selection
Our second strategy involved assigning weights to the features listed to calculate a single score for each enumerated NextState - the NextState with the best score was chosen.

Implementation-wise, we had a new class called NextState that extended State to help us calculate the various properties of each possible enumerated NextState, given a certain orientation and column placement of an incoming block. The Features class kept track of what features were being used to calculate the score of a given state.

We describe eight different feature sets[2] and how they performed relatively to one another. Some of the feature sets may be considered to be variations of a main feature set. Feature Sets 2 and 3 are modifications on Feature Set 1, to test the effect of including delta (differences between current and future state) and squared properties on ROWS_CLEARED, MAX_HEIGHT and COVERED_GAPS.

Feature Set 4 is nearly the same as Feature Set 1, but with a few crucial changes. Feature Set 4 no longer has the MAX_HEIGHT feature. We added the IS_SUICIDE feature to prevent the player from “committing suicide”, losing when it could have made another keep-alive move. Two other features we added were ROW_TRANSITIONS and COL_TRANSITIONS[3]. These calculated the number of “changes” between occupied and unoccupied cells by row and column respectively - in essence they counted the number of COVERED_GAPS (with a multiplication factor of 4), as well as the height difference in adjacent columns.

Feature Set 5 - 8 are modifications on Feature Set 4.

Features Used in Feature Set 4:
1. Normalizing Constant
2. Number of Rows Cleared in the NextState
3. Number of Covered Gaps in the NextState
4. Average Height of Occupied Cells in NextState
5. Summation of the Absolute Height Difference between Adjacent Columns in the NextState
6. Summation of the Squared Height Difference between Adjacent Columns
7. Number of Row Transitions (ROW_TRANSITIONS)
8. Number of Column Transitions (COL_TRANSITIONS)
9. IS_SUICIDE

Finally this strategy was augmented with a genetic generation-based learning algorithm (GA), implemented in the class Breeder, that updated the weights of the various features based on the scores of their parents. We used this GA iteratively on other GA-derived weights to fine-tune our weights from random seeds.

Experimental Results

---

[1] A fit is defined as leaving no “holes”, or covered gaps in the resulting board.

[2] Please refer to the Appendix for a full list of the various features in each feature set.

[3] Row transitions and column transitions are an idea we derived from this reference: https://www2.informatik.uni-erlangen.de/publication/download/mic.pdf

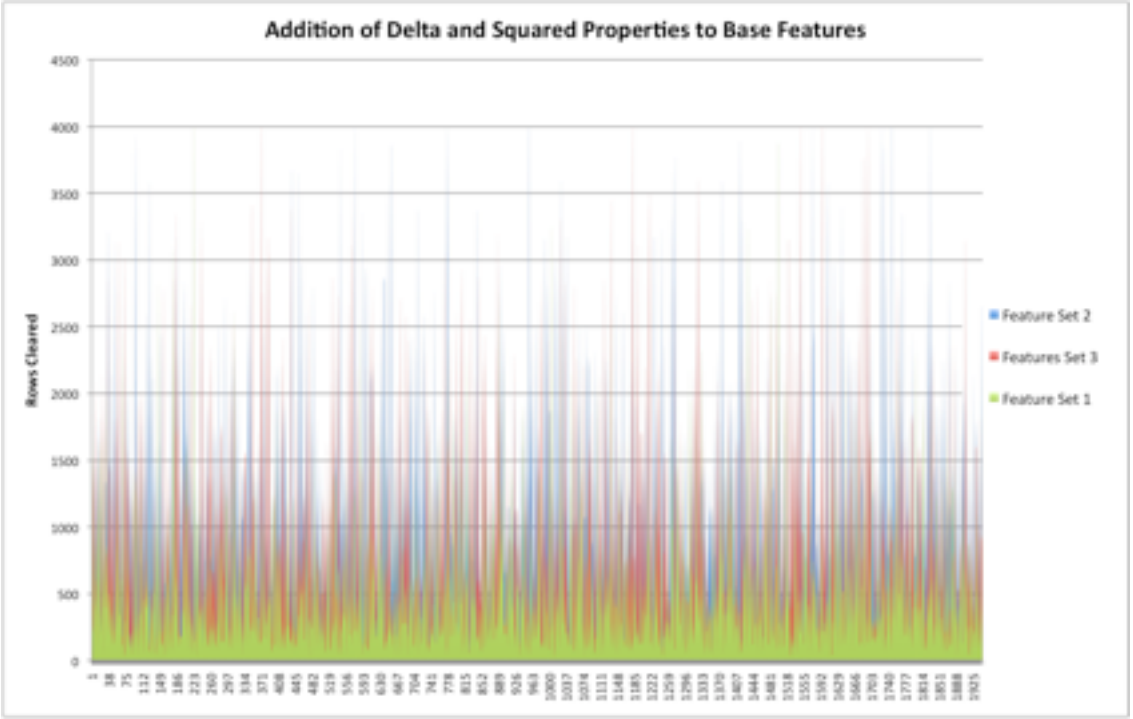

Figure 1. Chart of Performance of Feature Sets 1, 2, and 3 on >1500 fixed series of inputs.

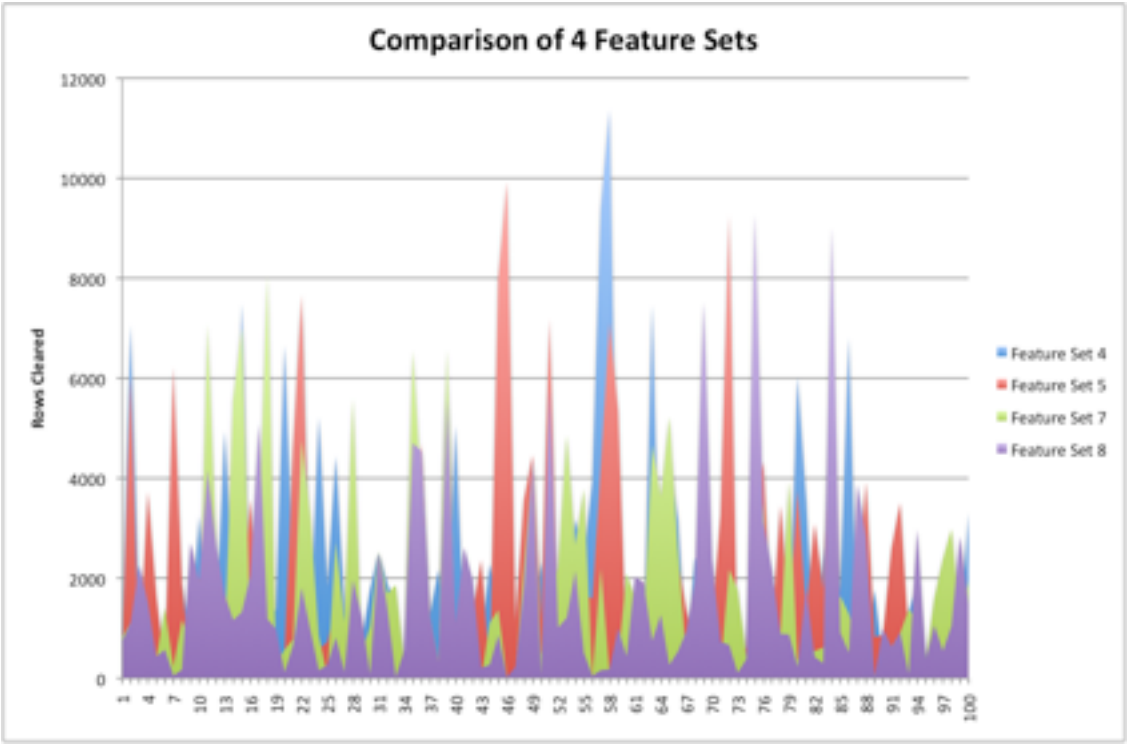

Figure 2. Chart of Performance of Feature Sets 4, 5, 7, 8 on 100 fixed series of inputs.

Analysis and Discussion

The players from Feature Sets 4-8 performed an order of magnitude better than the players from Feature Sets 1-3. This suggests the features of ROW_TRANSITIONS AND COL_TRANSITIONS are crucial to the utility-based agent. In a way, they are mimicking the “fit” of a piece, the better the fit of a piece to the given contour of occupied cells, the smaller the number of transitions. Amongst Feature Sets 1-3, the sets with the DELTA features appeared to be able to get higher scores. The SQUARED features do not appear to provide added benefit.

It may also suggest that MAX_HEIGHT is a feature that is detrimental for a high-performance agent. While maximum height seems intuitively important, sometimes it is better for the player to stack higher than create more holes. It is however, still important to keep a handle on the total height of the occupied cells, so leaving

in AVG_HEIGHT is necessary (we had another player with no height feature at all, and it performed with a mean of 140 rows cleared).

Our best performing player is Feature Set 4 with the weights -9.601434011245342E-6, 6.220389129888194E-8, -7.25722989951555E-7, 3.87295004028969E-8, -7.113950335019432E-8, -2.128328195173797E-8, -9.528753577395118E-7, -2.057322654518275E-8, 0.21522552000444517 respectively. As expected, the score is positively correlated with ROWS_CLEARED, but surprisingly, also positively correlated with AVG_HEIGHT. The heaviest weighted features are the COVERED_GAPS and ROW_TRANSITIONS. This corresponds with our intuitive understanding of trying not to create holes and fitting pieces well.

It is possible that we did not find the best set of weights possible for each of the Feature Sets. For example, Feature Set 7 and 8 are extensions of Feature Set 4, and we expected them to do better with the additional features of BLOCKADE, which counts the number of occupied cells above a hole, and ROWS_WITH_GAPS, which counts the number of rows containing holes (this feature would penalize a state that creates more rows with holes, given the same number of holes created, something that is not covered by COVERED_GAPS or ROW_TRANSITIONS). We had been monitoring games where the end state involved a game board with a single hole in each row.

The more features we use, the higher dimensional space we are in. Whilst the GA is a good way for us to fine tune our weights (it leads to 200-300% improvements from our first few hand-coded weights), it only allows us to move along the n-dimensional object between our starting points, given n features. If we seed the weights too close to each other, we could end up at a local, rather than global maximum. With a larger n, it naturally becomes harder to traverse the entire n-dimensional space - there is a combinatorial explosion on the possible sets of weights.

The current GA only compares two players in every iteration. We can design an improved GA that compares and combines more than two players at each iteration. Furthermore, the current implementation has a fixed probability of whether the better player's weights or the worse player's weights is used. It might be better if the probability of using a certain player's weight value for the next generation is proportional to the performance of that player as compared to the other players in the current iteration.

It can be said that there is almost no correlation between the same sequence of pieces and how well a player performs. Some players perform very well on a certain input - another player that performs better on average, could perform less well on the same sequence of pieces. Sometimes it is also possible for a player to have a different performance on the same sequence of inputs - we believe this is due to the loss of precision on the weights that cause certain decisions in the sequence to be different - leading to a different result. The utility-based agent can appear “non-deterministic” in this respect, as compared to a model-based or case-by-case decision-based agent.

We also tested our player on pathological inputs, for example, all ‘L’s. Surprisingly, it does slightly better on the ‘L’ then symmetric-L, clearing about 120 rows in the former, and only half of that, 60 rows, in the latter. It does most poorly on the ‘T’, managing about 20 rows. For the rest of the shapes, it is able to clear rows infinitely (the ‘Z’s and symmetric-Z build to some k number of rows, where k is 5 or 11, then finds a pattern that is able to stack the pieces and clear rows infinitely - occasionally they clear only 2 rows though).

The reason for a large variation in the number of rows cleared, by any player, is the frequency of occurrence of some pathological sequences. Certain sequences cause the build up of k rows. When ROWS/k of those sequences occur, the player loses. If these sequences appear in close succession, the player loses quickly, if they appear widely spaced, the player can clear many rows in the interim.

A final comment is about the rate of row clearance, defined as the number of rows cleared over the number of pieces read in. Naturally, the higher the rate, the better the player. We find that when a player has cleared upwards of 50 rows, its rate of row clearance is >0.3. Upwards of 80 rows, >0.33. The maximum rate reached by any of our players is 0.399, and it is a rate that can be reached by even our poorer feature sets. The number does make sense - there are 10 columns, and the median width of the pieces is 3. If the number of columns increased, we would expect the rate to be lower.

Concluding Statements

We proved that tetris is not all about what it looks like: fit in the pieces and make sure that we don’t pile them up too high. This might be the basic strategy at first glance, but more parameters are required to encode this knowledge in a utility-based agent.

Appendix

I.
These are the statistics of our best performing feature set (Feature Set 4):
*Score is number of rows cleared, Sequence is number of pieces read in, Rate is Score/Sequence

Mean score: 2051.39
Max score: 11359.0
Min score: 116.0
Median score: 1314.0
Standard Deviation: 2172.203967840038
Sequence Mean score: 5170.77
Sequence Max score: 28440.0
Sequence Min score: 331.0
Sequence Median score: 3329.0
Min rate: 0.3504531722054381
Max rate: 0.3994022503516174
Median rate: 0.39475271411338964

II.
Features Used in Feature Set 1:
1. Normalizing Constant (CONSTANT)
2. Number of Rows Cleared in the NextState (ROWS_CLEARED)
3. Maximum Height of NextState (MAX_HEIGHT)
4. Average Height of Occupied Cells in NextState (AVG_HEIGHT)
5. Number of Covered Gaps in the NextState (COVERED_GAPS)
6. Summation of the Absolute Height Difference between Adjacent Columns in the NextState
7. Summation of the Squared Height Difference between Adjacent Columns

Features Used in Feature Set 2 (all the features in Feature Set 1, including):
1. Difference between the Maximum Height of a possible NextState and the current given State (DELTA_MAX_HEIGHT)
2. Difference between the Number of Covered Gaps of a possible NextState and the current given State (DELTA_COVERED_GAPS)
3. Difference of (6) of a possible NextState and the current given State

Features Used in Feature Set 3 (all the features in Feature Set 2, including):
1. Number of Rows Cleared in the NextState squared
2. Maximum Height of NextState squared
3. Number of Covered Gaps in the NextState squared

Features Used in Feature Set 5 (all the features in Feature Set 4, except):
1. IS_SUICIDE

Features Used in Feature Set 6 (all the features in Feature Set 4, including):
1. Number of Rows Cleared in the NextState squared
2. Maximum Height of NextState squared
3. Number of Covered Gaps in the NextState squared

Features Used in Feature Set 7 (all the features in Feature Set 4, including):
1. Number of Occupied Cells Above an Unoccupied Cell in NextState (BLOCKADE)

Features Used in Feature Set 8 (all the features in Feature Set 7, including):
1. Number of Rows with Covered Gaps (ROWS_WITH_GAPS)

III.
Breeder.java, the class we used to fine-tune out weights:

```
import java.util.*;

public class Breeder {
    public final static int SAMPLE_SIZE = 100;
    public final static int BREED_ITERATION = 1000;
```

```
public static double HIGH_P_THRES = 88;
public static double LOW_P_THRES = 99;
public static int numFeatures = 7;

public static Random randGen = new Random();

public static double[] getStats(double[] weights) {
	int[] scoreList = new int[SAMPLE_SIZE];
	double avgScore = 0, varScore = 0, stdDevScore = 0;

	for (int i = 0; i < SAMPLE_SIZE; i++) {
		State s = new State();
		PlayerSkeleton p = new PlayerSkeleton(weights);
		while (!s.hasLost()) {
			s.makeMove(p.pickMove(s, s.legalMoves()));
		}
		scoreList[i] = s.getRowsCleared();
		avgScore += scoreList[i];
	}

	avgScore = avgScore / ((double) SAMPLE_SIZE);
	for (int i = 0; i < SAMPLE_SIZE; i++) {
		varScore += (scoreList[i] - avgScore) * (scoreList[i] - avgScore)
				* 1.0;
	}
	varScore = varScore / SAMPLE_SIZE * 1.0;
	stdDevScore = Math.sqrt(varScore);

	return new double[] { avgScore, varScore, stdDevScore };
}

static boolean better(double[] score1, double[] score2) {
	return score1[0] > score2[0];
}

public static void main(String[] args){
	double[] wRef1 =  new double[] {
	// 1946.70,			/* CONSTANT */
	// 388.43,			/* ROWS_CLEARED */
	// -3.36,			/* MAX_HEIGHT */
	// -4.82,			/* DELTA_MAX_HEIGHT */
	// -68.40,			/* COVERED_GAPS */
	// -111.74,		/* DELTA_COVERED_GAPS */
	// -10.92,			/* AVG_HEIGHT */
	// 379.08,			/* DELTA_AVG_HEIGHT */
	// -22.02,			/* SUM_ABS_HEIGHT_DIFF */
	// -20.79			/* DELTA_SUM_ABS_HEIGHT_DIFF */

      -7.720365371906888,
		5.940915554719386,
		//4.53450876353425480,
		-0.12124613960138278,
		//0.27983173173592785,
		//-0.55714395842047314,
		-2.477321983110656,
		//-7.227672112968805,
		//-9.067154095408078454,
		-0.14518808292558213,
		//3.1585370155208734,
		//-1.257975960254059461721,
		-1.1860124403007994,
		//-0.7243385364234114,
		//-1.4348465749525040510,
		-1.103434819057849578581
		};
	double[] wHand =  new double[] {
	0,			/* CONSTANT */
	100,			/* ROWS_CLEARED */
	-5,
	-5,			/* MAX_HEIGHT */
	-5,			/* DELTA_MAX_HEIGHT */
	-10,
	-150,		/* COVERED_GAPS */
	-200,		/* DELTA_COVERED_GAPS */
	-20,
	-5,			/* AVG_HEIGHT */
	-100,			/* DELTA_AVG_HEIGHT */
	-20,
	-30,		/* SUM_ABS_HEIGHT_DIFF */
	-20			/* DELTA_SUM_ABS_HEIGHT_DIFF */
	-10,
	-5
		};
```

```
		int NGEN = 8;
		Random random = new Random();
		double[][] gen = new double[NGEN][];
		//gen[0] = new double[] {-9.770756747569378, 7.536096948211782, -0.15344104005358817, 0.35994719334166974, -3.1346459740890284, -9.165803729592603, -0.2283504662987085, 3.999204134104953, -1.5007260553728094, -0.91653199687301};
		//gen[1] = new double[] {486.675, 100.0, -3.36, -4.82, -68.4, -200.0, -5.0, -100.0, -30.0, -20.0};
		//gen[2] = new double[] {105.43253815742321, 30.83704520472885, -0.9297627843871829, -1.1841151106423722, -68.4, -200.0, -5.0, -100.0, -30.0, -20.0};
		//gen[1] = new double[] {32.48581083059224, 27.23001322384344, -0.25561010883341245, -1.1841151106423722, -68.4, -200.0, -5.0, -100.0, -30.0, -20.0};
		//gen[1] = new double[] {0.9266257041073647, 0.837484482310802, -0.005298893029129517, -0.043717150606703115, -1.5733348100669897, -5.404692237085569, -0.13511509864342033, -2.414533892953383, -1.0021306562633423, -0.6055850494278614};
		//gen[2] = new double[] {0.021596608846013502, 0.014249672906482562, -1.3661955331280978E-4, -9.534315780956981E-4, -0.03915677305256805, -0.09241272364259767, -0.0031985161670923736, -0.05915856517255117, 0.03467310044516623, -0.02260961512491643};
		//gen[3] = new double[] {3.158594275364314, 2.6475732539183303, -0.024852962134660882, 0.08716397231688221, -6.650529659289416, -19.44599315581701, -0.4861498288954254, -9.722996577908505, -2.9168989733725503, -1.9445993155817016};
		//gen[4] = new double[] {2.5019225255160737, 2.0971427744287094, -0.019686031306864884, 0.0690425824722024, -5.267884543123147, -15.403171178722655, -0.38507927946806647, -7.701585589361327, -2.310475676808397, -1.5403171178722659};

		// avg 1036 stddev 1013
		gen[0] = new double[] {
				-7.720365371906888,
				5.940915554719386,
				//0.5+random.nextGaussian()*1.2,
				-0.12124613960138278,
				//0.27983173173592785,
				//-0.5-random.nextGaussian()*1.5,
				-2.477321983110656+random.nextGaussian(),
				//-0.227672112968805-random.nextGaussian(),
				//-9.0-random.nextGaussian()*1.5,
				-0.14518808292558213,
				//3.1585370155208734+random.nextGaussian(),
				//-1.0-random.nextGaussian()*1.3,
				-1.1860124403007994+random.nextGaussian()*0.01,
				//-0.7243385364234114+random.nextGaussian()*0.01,
				//-1.0-random.nextGaussian()*1.3,
				-1.0-random.nextGaussian()*1.3};
		// 868 676
		gen[1] = new double[] {
				-6.784655270236987,
				5.236380532976169,
				//2.3+random.nextGaussian(),
				-0.10674959177950814,
				//0.24581475242154946,
				//-1.3-random.nextGaussian(),
				-1.4093078526900575,
				//-6.414398381783404,
				//-5.0-random.nextGaussian(),
				-0.12911666396094648,
				//2.755367234728835,
				//-5.0-random.nextGaussian(),
				-1.053712254027337,
				//-0.6434737625468806,
				//-3.0-random.nextGaussian(),
				-2.0-random.nextGaussian()};
		//942 906
		gen[2] = new double[] {
				-1.374125439554717+random.nextGaussian(),
				4.147737020170424+random.nextGaussian(),
				//7.0+random.nextGaussian(),
				-0.08455635164854841+random.nextGaussian(),
				//0.19470986539310936+random.nextGaussian(),
				//-1.0-random.nextGaussian(),
				-1.1233123227966717+random.nextGaussian(),
				//-5.080844958210634+random.nextGaussian(),
				//-0.5-random.nextGaussian(),
				-0.10227330952346571+random.nextGaussian(),
				//2.1825263866287106+random.nextGaussian(),
				//-0.3-random.nextGaussian(),
				-0.8346454764150536+random.nextGaussian(),
				//-0.5096955673133842+random.nextGaussian(),
				//-1.3-random.nextGaussian(),
				-1.0-random.nextGaussian()};

		gen[3] = new double[] {
				-2.1909022200706247,
				11.1749796738761684,
				//1.3650709288169485,
				-0.024301640103314634,
				//0.07964036094800452,
				//-0.23441100743272944,
				-0.6905009190278727,
		//		-2.051625370158972,
		//		-2.948565712335654,
```

```
                -0.04087007519868151,
        //      3.1585370155208734,
        //      -0.6996312021491575,
                -1.1860124403007994,
        //      -0.7243385364234114,
        //      -0.2259546444639846,
                -0.5299243737589001
        };

        gen[4] = new double[] {
                -1.949902975862856,
                7.04573190974979,
        //      1.214913126647084,
                -0.021628459691950024,
        //      0.07087992124372403,
        //      -0.2086257966151292,
                -0.6145458179348067,
        //      -1.825946579441485,
        //      -2.624223483978732,
                -0.036374366926826544,
        //      3.1585370155208734,
        //      -0.6996312021491575,
                -1.1860124403007994,
        //      -0.7243385364234114,
        //      -0.2259546444639846,
                -0.5299243737589001
        };

        gen[5] = new double[] {
                -5.3203050035735036,
                1.745056208022166,
        //      0.38347609288277873,
                -0.007248558472990012,
        //      0.1933598441151341,
        //      -0.04739478360810739,
                -0.19494091419998483,
        //      -0.5776969995408238,
        //      -2.668241265494882,
                -0.044045306374088766,
        //      3.1585370155208734,
        //      -0.6996312021491575,
                -1.1860124403007994,
        //      -0.7243385364234114,
        //      -0.2259546444639846,
                -0.5299243737589001
        };

        gen[6] = new double[] {
                -4.759072056863753,
                5.245456250233282,
        //      0.33975773956808447,
                -0.0075911928522450385,
        //      0.17295498013867727,
        //      -0.04807754772039741,
                -0.1918088182505231,
        //      -0.521970542278571,
        //      -2.440072206777856,
                -0.12820596603825316,
        //      3.1585370155208734,
        //      -0.6996312021491575,
                -1.1860124403007994,
        //      -0.7243385364234114,
        //      -0.2259546444639846,
                -0.5299243737589001
        };

        gen[7] = new double[] {
                -0.0030449018076265126,  0.008503946354535288,  -5.645284418745485E-6,  -0.008817400172507569,  9.471116309795316E-6,
-7.920136586945793E-4, -1.2018971001413466E-4
        };

        double[] w1 = gen[7], w2 = gen[0];
        double[] goodPlayer = null, badPlayer = null, goodScore = null, badScore = null;

        double bestScore = 0;
        double[] bestPlayer = null;
        for(int i=0;i < BREED_ITERATION; i++){
                System.out.println("Starting generation " + i + "...");
                System.out.println("\nBest player so far of avg score: " + bestScore);
                System.out.println("stats: " + Arrays.toString(bestPlayer));
                System.out.println();
                double[] score1 = getStats(w1);
                double[] score2 = getStats(w2);
```

```
                    if(score1[0] > bestScore){
                            bestScore = score1[0];
                            bestPlayer = Arrays.copyOf(w1, numFeatures);
                    }
                    if(score2[0] > bestScore){
                            bestScore = score2[0];
                            bestPlayer = Arrays.copyOf(w2, numFeatures);;
                    }

                    boolean changed = false;
                    if(score1[0] < bestScore/5){
                            w1 = Arrays.copyOf(bestPlayer, numFeatures);
                            w2 = Arrays.copyOf(gen[randGen.nextInt(NGEN)], numFeatures);
                            changed = true;
                    }
                    if(score2[0] < bestScore/5){
                            w1 = Arrays.copyOf(gen[randGen.nextInt(NGEN)], numFeatures);
                            w2 = Arrays.copyOf(bestPlayer, numFeatures);
                            changed = true;
                    }
                    if(changed) continue;

                    if(better(score1,score2)) {
                            goodPlayer = Arrays.copyOf(w1, numFeatures);
                            goodScore = score1;
                            badPlayer = Arrays.copyOf(w2, numFeatures);
                            badScore = score2;
                    } else {
                            goodPlayer = Arrays.copyOf(w2, numFeatures);
                            goodScore = score2;
                            badPlayer = Arrays.copyOf(w1, numFeatures);
                            badScore = score1;
                    }

                    //HIGH_P_THRES = goodScore[0] / badScore[0] * 100.0;
                    //LOW_P_THRES = 100 - HIGH_P_THRES - 1;

                    for(int j=0;j<numFeatures;j++){
                            int p = randGen.nextInt(101);
                            w1[j] = p <= HIGH_P_THRES ? (goodPlayer[j] * (HIGH_P_THRES/100) + badPlayer[j] * ((100-LOW_P_THRES)/100))  :
                                            (p <= LOW_P_THRES ? (badPlayer[j] * (HIGH_P_THRES/100) + goodPlayer[j] * ((100-LOW_P_THRES)/
100)) :
                                            randGen.nextGaussian() * goodPlayer[j]);
                            p = randGen.nextInt(101);
                            w2[j] = p <= HIGH_P_THRES ? (goodPlayer[j] * (HIGH_P_THRES/100) + badPlayer[j] * ((100-LOW_P_THRES)/100)) :
                                            (p <= LOW_P_THRES ? (badPlayer[j] * (HIGH_P_THRES/100) + goodPlayer[j] * ((100-LOW_P_THRES)/
100)) :
                                                    randGen.nextGaussian() * goodPlayer[j]);
                    }

                    printStats(goodPlayer, badPlayer, goodScore, badScore);
            }

            System.out.println("Results after " + BREED_ITERATION + " generations:");
            printStats(goodPlayer, badPlayer, goodScore, badScore);

            System.out.println("Best player of avg score: " + bestScore);
            System.out.println("stats: " + Arrays.toString(bestPlayer));
    }

    static void printStats(double[] goodPlayer, double[] badPlayer, double[] goodScore, double[] badScore){
            System.out.print("GoodPlayer stats:\nweights: ");
            for(int i=0;i<numFeatures;i++){
                    System.out.print(goodPlayer[i] + ", ");
            }
            System.out.println("\nAvg Score: " + goodScore[0]);
            System.out.println("Variance: " + goodScore[1]);
            System.out.println("Std Dev: " + goodScore[2]);

            System.out.print("\nBadPlayer stats:\nweights: ");
            for(int i=0;i<numFeatures;i++){
                    System.out.print(badPlayer[i] + ", ");
            }
            System.out.println("\nAvg Score: " + badScore[0]);
            System.out.println("Variance: " + badScore[1]);
            System.out.println("Std Dev: " + badScore[2]);
            System.out.println();
    }
}
```