Nacka Gymnasium

Gymnasiearbete

Jämförelse mellan en genetisk algoritm och mänskliga spelares resultat i Tetris

Prövning av vilka värden på mutationssteget och mutationssannolikheten som ger de mest högkvalitativa individerna

Pontus Söderlund

Handledare

Henrik Sparrman

3 oktober 2019

Abstract

The following report will describe the construction of a bare-bones Tetris clone as well as a genetic algorithm capable of playing said clone. The the first of the two goals of this report is to answer to what extent a genetic algorithm is able to compete with a human. The second is to answer what values for the mutation rate and mutation step produces the highest quality solution. The results of the study is that the genetic algorithm's scores are at least 88% better than the average human players score and 11% better than the best.

Innehåll

1 Inledning **3**

1.1 Bakgrund 3

1.1.1 Tetris 3

1.1.2 Genetiska Algoritmer 3

1.2 Mål och Frågeställning 6

1.3 Metod 7

1.4 Avgränsningar 7

2 Genomförande **8**

2.1 Programstruktur 8

2.2 Konstruktion av Tetris-klon 9

2.2.1 Modell 9

2.2.2 Presentatör 10

2.2.3 Kontrollant 10

2.3 Klasser och metoder av intresse för Tetris 12

2.3.1 Klassen `MyGameModel` 12

2.3.2 Klassen `Board` 13

2.4 Konstruktion av den genetiska algoritm 15

2.5 Klasser och metoder av intresse för den genetiska algoritmen . . 16

2.5.1 Klassen `Genetic Algorithm` 17

3 Resultat **19**

4 Diskussion **22**

4.1 Resultatet 22

4.2 Begränsningar 22

4.3 Slutsatts 23

4 Literatur **23**

5 Bilagor **25**

5.1 Övrigt 25

5.2 Kod 26

5.2.1 `Board` 26

5.2.2 `Coordinator` 43

5.2.3 `GameView` 51

5.2.4 `Generation` 57

5.2.5 `GeneticAlgorithm` 61

5.2.6 `GeneticAlgorithmView` 69

5.2.7 `Genome` 74

5.2.8 `Model` 79

5.2.9 `Move` 80

5.2.10 `MyGameMode` 82

1. Inledning

1.1 Bakgrund

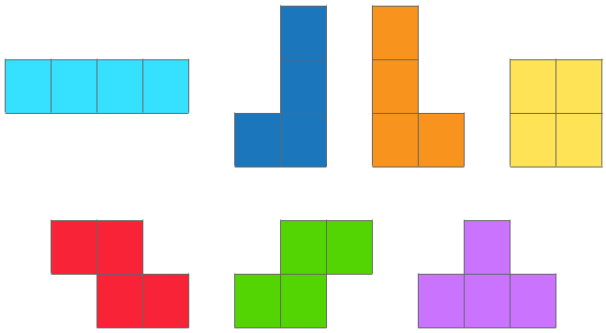
1.1.1 Tetris

Tetris är ett datorspel där målet är att få så höga poäng som möjligt genom att kombinera fallande tetrominoer. Det släpptes 1984 av den ryske speldesignern Alekséj Leonídovitj Pázjitnov och det blev snabbt en del av populärkulturen. Spelet har givits ut i flera olika versioner av olika företag och magasinet Electronic Gaming Monthly har utsett Tetris till "Det Bästa Spelet Någonsin" [Wik19a].

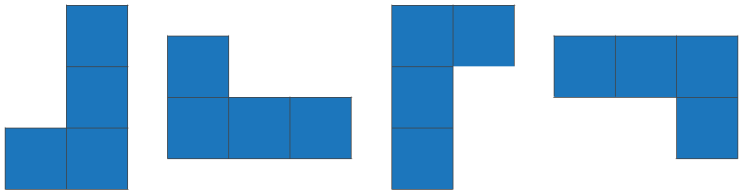
De mest populära versionerna av Tetris spelas på ett rutnät som är 10 rutor brett och 20 högt med sju olika enkelsidiga tetrominoer (fortsättningsvis även kallat delar). En enkelsidig tetromno är en kombination av fyra kvadratiska rutor, se figur 1.1 (s. 4). När spelet startar finns det en slumpmässigt vald tetromino i spelbrädets överkant som sedan börjar förflyttas, eller falla, nedåt ett steg i taget. Detta sker efter ett förutbestämt intervall som, i vissa versioner, beror på hur länge spelaren har kört. Detta är den så kallade aktiva delen. Spelaren kan kontrollera den aktiva delen genom att förflytta den i sidled, nedåt eller genom att rotera den. Vid rotation roteras den aktiva delen 90°medurs, se figur 1.2 (s. 4). Den aktiva delen slutar vara aktiv om den inte längre kan falla nästa gång. För att en del ska kunna förflyttas eller rotera krävs det att de rutorna som tetrominons rutor kommer att hamna i inte innehåller någon icke-aktiv ruta av en del, se figur 1.3 (s. 4). När biten har fallit färdigt och av aktiverats får spelaren poäng baserat på hur många rader som har rensats. En rad rensas om den är fylld med tetrominoer så att den bildar en fylld rad, se figur 1.4 (s. 5). Om det händer blir alla rutor på raden tomma och förflyttas högst upp på brädet. Spelaren får då poäng baserat på hur många rader som rensats. Spelaren får även poäng vare gång den aktiva tetrominon faller eller flyttas nedåt.

1.1.2 Genetiska Algoritmer

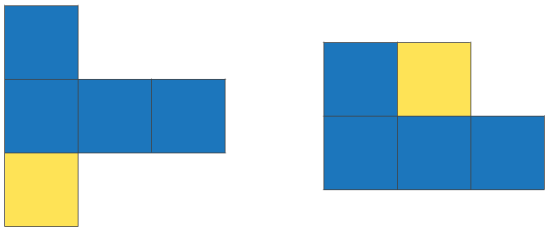
En genetisk algoritm är en form av optimiseringsalgoritm inspirerad av naturlig selektion. Genetiska algoritmer används framförallt för att hitta lösningar på problem när det finns otillräckligt med information eller om det är okänt



Figur 1.1: De sju olika tetrominoerna, I, J, L, O, Z, S och T.



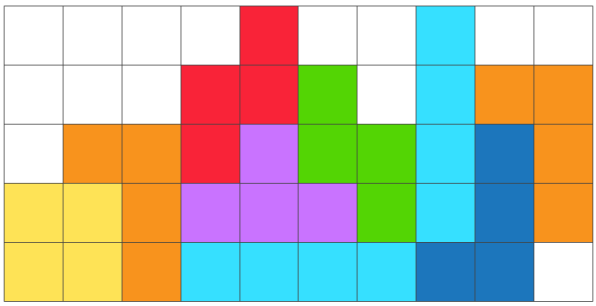
Figur 1.2: De fyra rotationstadierna (1-4) för en J-tetromino.



Figur 1.3: I den högre bilden kan J-delen inte falla men rotera. I den vänstra kan J-delen inte rotera men falla.

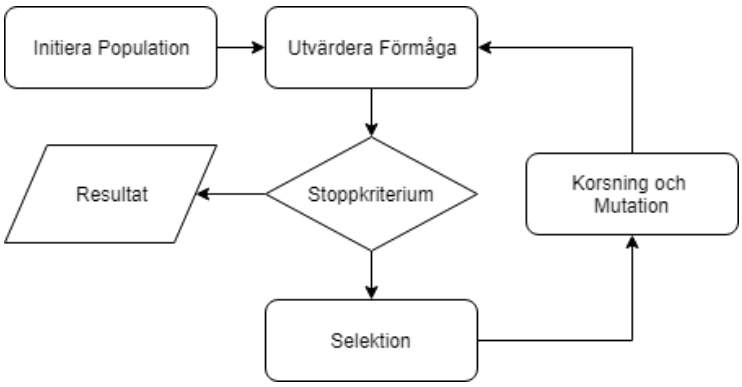
hur en optimal lösning ser ut. Den amerikanska rymdfartsorganisationen NASA använde en genetisk algoritm när de skulle designa en antenn till rymdfarkosten ST5 då antennen hade väldigt specifika krav. [Wik19b]

Idéen om att använda evolution för att hitta lösningar på komplexa problem har funnits under en lång tid. Konceptet ”genetisk algoritm” uppstod på 60-talet av professorn John Holland och i sin nuvarande form i slutet på 80-talet. Idag används genetiska algoritmer för att uppskatta lösningar på problem där konventionella metoder kan vara svåra att tillämpa. De används också där det är svårt för människor att designa en lösning men där det finns ett antal fördefinierade parametrar som går att mäta. Vanliga användningsområden är där optimeringar är inblandade. T.ex. så har NASAs Evolvable Systems Group använt sig av genetiska algoritmer för att designa satellitantenner.



Figur 1.4: I detta fall skulle raden näst längst ned rensas då den är fylld med tetromino rutor

En genetisk algoritm baseras som sagt på naturlig selektion. För att en genetisk algoritm ska fungera krävs det individer med ett antal heuristiska drag, metoder för selektion, korsning och mutation samt ett stoppkriterium. De heuristiska dragen är inte alltid nödvändiga men används nästan alltid då det gör beräkningarna enklare. I fallet Tetris skulle de heuristiska dragen kunna vara vikter som avgör vilket drag som skall spelas. T.ex. skulle en vikt kunna vara hur många hål som finns på brädet om den flyttar tetrominon i någon riktning. Vikterna är nästan aldrig heltal och det är vikterna som den genetiska algoritmen tar fram. Stoppkriterierna kan se olika ut beroende på lösningen ska användas till men vanliga kriterier är tid, godtagbar lösning, antal generationer och liknande.

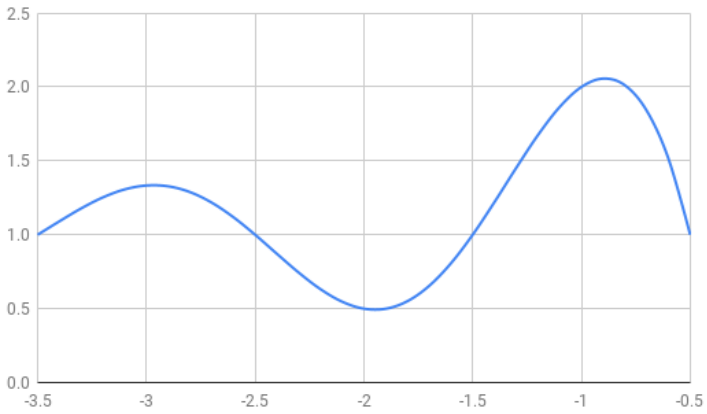


Figur 1.5: Schematiskt diagram av en genetisk algoritm

Figur 1.5 (s. 5) visar de olika stegen i en genetisk algoritm. Det första som händer är att en befolkning initialiseras. Oftast har de den första generationens individer slumpmässigt valda vikter (heuristiska drag) då det i de alldra flesta fallen är okänt vilka värden som ger bra resultat. Därefter utvärderas alla individerna mot utvärderingsfunktionen. Funktionen mäter hur bra individernas resultat är. Om stoppkriteret då är uppfyllt presenteras den bästa individen och dess lösning som resultat, om inte fortsätter algoritmen. Nästa steg är korsning och mutation. I detta stadie genereras en ny generation av individer som baseras på den föregående generationen. För att skapa en ny ge-

neration av s.k. ”barn” från den föregående generationen av s.k. ”föräldrar” krävs en urvalsmetod som väljer ut de bästa individerna från föräldrarna och låter dem ”paras”. Nya föräldrar väljes till varje barn av urvalsmetoden. Denna process pågår tills generationen har tillräckligt många individer. Efter det finns det en sannolikhet att varje barns vikter muteras för att förhindra att populationen kommer till en s.k. platå för tidigt. En platå i det här fallet är när individerna inte längre förbättras, de har blivit så bra de kan bli under förutsättningarna. Efter att korsning och mutations-steget har skett börjar processen om igen.

En begränsning med genetiska algoritmer är att lösningen som hittas inte alltid är den optimala. Inom matematiken finns det så kallade lokala och globala max-värden. Som kan ses i figur 1.6 är båda topparna lokala max-värden, men båda är inte nödvändigtvis globala max-värden. Det globala max-värdet är det största värdet inom defenitionsmängden. Både $(-3.5, 1)$ och $(-0.9, 2.1)$ lokala max-värden men $(-0.9, 2.1)$ är det globala maxvärdet. I det här arbetet kommer inspirationen för strukturen av den genetiska algoritmen samt Tetris från Siraj Ravals YouTube video ”How to Make an Evolutionary Tetris AI”[Sir17] där han går igenom och förklar hur Idrees Hassans program ”TetNet”[Idr17] fungerar. Hassans projekt är dock skrivet i JavaScript och i detta arbete kommer Java att användas vilket resulterar i en annorlunda struktur med klasser.



Figur 1.6: Lokala och globala max-värden.

1.2 Mål och Frågeställning

Detta arbete kommer besvara följande punkter:

- Hur förhåller sig en genetisk algoritms resultat i Tetris till en mänsklig spelares resultat?
- Vilken kombination av värden på parametrarna mutationssteg och mutationssannolikhet ger bäst resultat?

1.3 Metod

För att ta reda på det ovan nämnda behöver nedanstående utföras:

1. Skapa en Tetris-klon
2. Skapa en genetisk algoritm
3. Undersöka de olika parametrarnas påverkan på resultatet
4. Skapa ett dataset från mänskliga spelare
5. Jämföra den genetiska algoritmens resultat mot de mänskliga spelarnas resultat.

Testerna kommer att utföras med 100 generationer med 100 individer i varje generation. Den bästas poäng sparas. Detta kommer att upprepas 10 gånger för varje sett av parametrar. Att gör det 10 gånger ger en bättre bild av vilken kombination av parametrar som är den bästa. 21 värden kommer att testas för varje parameter. Värdena kommer att börja på 0 och öka med 0.05 tills det att de når 1, då det motsvarar 100% mutationssannolikhet. Detta sker för båda parametrarna. Varje individ kommer maximalt att göra 1000 drag (förflyttningar eller rotationer) Medelvärdet av de olika parametrarna kommer sedan jämföras för att se vilken kombination som ger det mest högkvalitativa resultatet. Ovan nämnda scenario skulle betyda att $100 \cdot 100 \cdot 10 \cdot 20$ individer skulle behöva utvärderas. Produkten av de talen är $40\,000\,000$. Ponera att hela programmet ska köra i åtta timmar. Det skulle betyda att 1400 individer skulle behöva utvärderas varje sekund.

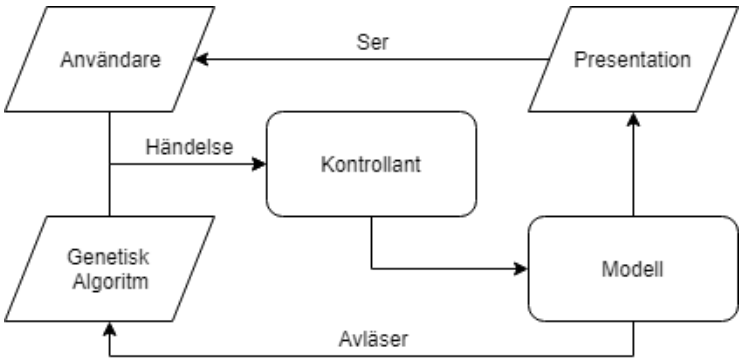
1.4 Avgränsningar

- De värdena som kommer att undersökas är endast mutationssanolikheten och mutationssteget. Mutationssteget kommer endast att studeras mellan 0 och 1 även om värden större än det är möjligt.
- De heuristiska dragen kommer att bestå av fem så kallade vikter.
  - Antal rensade rader
  - Den högsta kolumnens höjd
  - Summan av höjden för alla kolumner
  - Antal hål på brädet
  - Hur jämnt brädet är
- Normalt har genetiska algoritmer 100 eller 1000-tals individer för att få en så stor genpol, och så bra resultat som möjligt. I detta arbete kommer endast ett 100-tal individer att användas som mest då det annars skulle behövas mer beräkningskapacitet. Varje individ kommer maximalt att göra 1000 drag då fler skulle vara allt för processorkrävande.

2. Genomförande

2.1 Programstruktur

För att kunna testa och utvärdera den genetiska algoritmen behövs det en Tetris-klon och en genetisk algoritm som kan spela Tetris. Applikationen konstrueras med hjälp av arkitekturmönstret Model-View Controller som övergripande består av tre delar, en modell (Model) vars uppgift är att modellera applikationen, en presentatör (View) som presenterar modellen, och en kontrollant (Controller) som förmedlar händelser (t.ex. knapptryckningar) till modellen. Programmet struktureras på detta sätt för att skapa en klar separation mellan vad som är vad. Det är t.ex. inte relevant för presentatören att veta vad som händer i modellen, dess jobb är endast att visa hur modellen ser ut. En annan fördel med att använda MVC är att det inte behövs två separata program för att för att en användare ska kunna spela Tetris och en för att den genetiska algoritmen. Istället finns det en kontrollant som som kan konvertera och förmedla båda in strömmarna till modellen (se figur 2.1 (s. 8)). Klassen `Coordinator` fungerar som huvudklass för hela programmet och kontrollant åt `MyGameModel` samt `Genetic Algorithm`. En bi-funktion är att ha konstanter åt de andra klasserna för att underlätta förändring och testning.



Figur 2.1: Schematisk diagram över hur programmet är strukturerat för både användare och den genetiska algoritmen.

2.2 Konstruktion av Tetris-klon

Tetris delen av programmet består i huvudsak av klasserna `MyGameModel`, `Board` och `GameView`. `MyGameModel` modellerar tillsammans med `Board` hur spelet ser ut och `MyGameModel` tar emot styrsignaler från `Coordinator`. `GameView` visar brädet, poäng och dylikt.

2.2.1 Modell

Modellen, `MyGameModel`, består i huvudsak av ett bräde av klassen `Board` samt andra variabler och metoder inom huvudkategorierna "bräde", "hjälp", "hämtare" (Getters) och "sättare" (Setters). Metoderna inom kategorin "bräde" manipulerar brädet, t.ex. genom att flytta den aktiva tetrominon eller generera en ny tetromino. "Hjälp"-metoderna är metoder som kan bl.a. klona instansen av modellen och återställa modellen till sitt grundtillstånd. "Hämtare" och "sättare" metodernas uppgift är att ge andra klasser tillgång till att ändra och hämta datan från de privata instansvariablerna i modellen. Rörelse (rotation, i sidled och nedåt) sker när metoden `movement()` annropas. Metoden annropar i sin tur metoden `move()` i i klassen `Board`.

Klassen `Board` består av en tvådimensionell lista av heltal (20 och 10 element långa) som representerar spelbrädet. Detta då ett vanligt Tetris spelbräde är uppbyggt 20 rader och 10 kolumner. Varje element i listan representerar en ruta. De sju olika terominoer motsvarar en siffra, 1-7, och varje ruta i tetrominon d.v.s. en "T" tetromino motsvarar id 7, "J", 2, etc, se tabell 2.1 (s. 9). För att kunna särskilja de aktiva från de icke-aktiva delarna har de aktiva delarna ett negativt värde och de icke-aktiva ett positivt värde. Istället för numret 7 för en passiv ruta skulle en aktiv ruta då få -7. Detta är nödvändigt då endast den aktiva delen ska flyttas och inte de andra delarna. Brädet roterar med hjälp av matriser och det är då inte nödvändigt att kunna identifiera de enskilda rutorna i den aktiva delen. All form av rörelse sker, som ovan nämnt via `move()`. Metoden annropar undersöker först om det går att göra den förfrågade förflyttningen och gör sedan den om det är möjligt. (Se Tabell 2.1 (s. 9) och figur 2.2 (s. 10)).

Del	Id	Aktiv Id
Tom	0	-
I	1	-1
J	2	-2
L	3	-3
O	4	-4
S	5	-5
Z	6	-6
T	7	-7

Tabell 2.1: Tetrominoer och deras motsvarande id för passiva och aktiva rutor

0	0	0	0	0	0
0	3	3	0	-71	0
4	4	3	-72	-73	-74
4	4	3	0	0	0

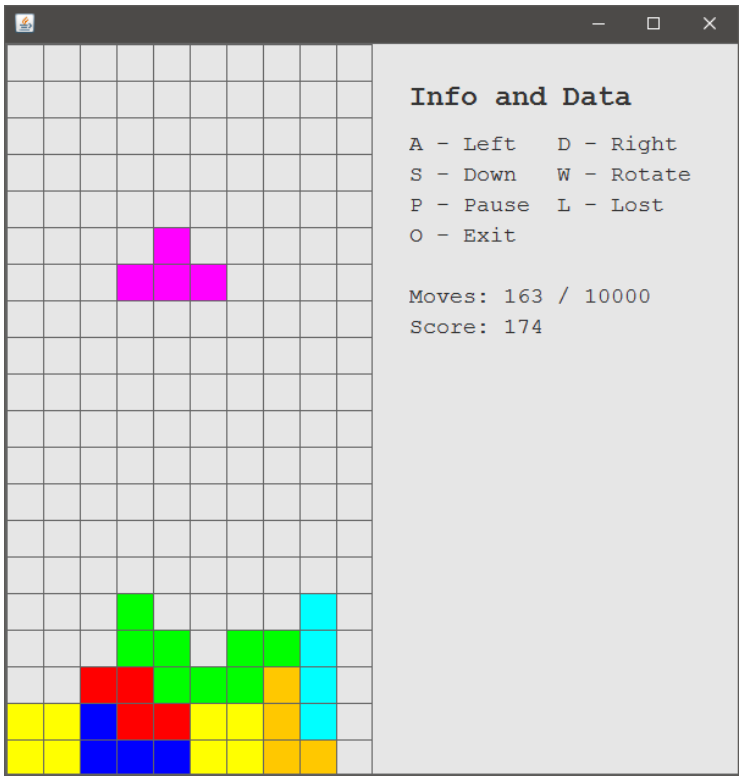
Figur 2.2: Exempel av hur det ser ut för aktiva och icke-aktiva delar.

2.2.2 Presentatör

Presentatören är det visuella som användaren ser och syftet med den är att ge en överblick över spelet. Den gör detta genom att dels visa hur brädet ser ut men också genom att visa poäng och antal förflyttningar och rotationer (drag). I programmet representeras Tetris-delens presentatör av klassen `GameView` som i huvudsak består av två inre klasser, `DisplayPanel` och `InfoPanel`. `DisplayPanel` har som uppgift att rita ut Tetris-brädet och `InfoPanel` innehåller visar poäng och antal drag, se figur 2.3 (s. 11). Anledningen till att det är en klass som innehåller två inre klasser är för att det ska gå att stänga av uppdatering av brädet visuellt när den genetiska algoritmen kör. I övrigt innehåller `GameView` metoden `propertyChange()` som annropas när användaren flyttar den aktiva delen eller den faller. Metoden anropar i sin tur `repaint()` i `DisplayPanel` och `update()` i `InfoPanel` som ritar om och uppdaterar informationen som visas i respektive klass.

2.2.3 Kontrollant

Som tidigare nämnt fungerar `Coordinator` som kontrollant till spelet. De enda händelserna som `Coordinator` behöver förmedla är knapptryckningar och händelser från timern *`updateTimer`*. Knapptryckningarna och deras händelser defineras i metoden `addKeyBindings()` som anropar metoden `addKeyBinding()`. I `addKeyBindings()` bestäms det vilka knappar som ska göra vad medan `addKeyBinding()` ser till så att de faktiskt implementeras. Anledning till att `KeyBindings` används istället för `KeyEvent` är så att spelaren alltid ska kunna kontrollera den aktiva tetrominon oavsett vilken panel som har fokus. Nedan är metoden `keyInput` som annropas om någon av knapparna W, A, S eller D blir nedtryckta. Metoden annropar då, om antal drag inte överskrider gränsen, `movement()` i `MyGameModel`. `keyInput` metodens syfte är att samla upp relevanta signaler för rörelse. Anledningen till att den är strukturerad med ett `switch-case` är för att det ska vara lättare att eventuellt lägga till fler knapptryckningar i framtiden.



Figur 2.3: `DisplayPanel` utgör den vänstra sektionen, `InfoPanel` den högra och båda tillsammans `GameView`

```
private void keyInput(String input) {
    if (updateTimer.isRunning()) {
        switch (input) {
            case "rotate":
            case "left":
            case "down":
            case "right":
                if (model.getNumberOfMoves() > Coordinator.MOVELIMIT) {
                    model.lost();
                } else {
                    model.movement(input, false);
                }
                break;
        }
    }
}
```

2.3 Klasser och metoder av intresse för Tetris

2.3.1 Klassen `MyGameModel`

`MyGameModel` kontrollerar logiken bakom brädet och skickar vidare relevant data till `Board`.

`movement()` är metoden som anropas av `Coordinator` då antingen användaren eller den genetiska algoritmen vill flytta eller rotera en del. Metoden anropar i sin tur metoden `move()` i `Board`. `firePropertyChange()` notifierar presentatören om brädet har ändrats. Variabeln ***thinking*** håller reda på om brädet ska ritas om när den genetiska algoritmen testar möjliga drag. Anledningen till att metoden endast anropar metoder och inte ändrar brädet själv är dels p.g.a. det är lättare att få en översiktlig blick över hur programmet är strukturerat men också p.g.a. det då är enklare att felsöka programmet och det blir en större tydlighet i vad som gör vad.

```
public void movement(String direction, boolean thinking) {
    int[][] old = board.getBoard().clone();

    if (board.move(direction)) {
        numberOfMoves++;
        if (direction.equals("down")) {
            score++;
        }
    }
    if (!thinking) {
        pcs.firePropertyChange("movement", old, board.getBoard());
    } else {
        pcs.firePropertyChange("thinkingMovement", old,
             board.getBoard());
    }
}
```

`deselectPiece()` går igenom hela brädet och om någon ruta är aktivt konverterar metoden den rutan till en passivt

```
public void deselectPiece() {
    int b[][] = board.getBoard();
    for (int i = 0; i < Board.NOCOLLUMNS; i++) {
        for (int j = 0; j < Board.NOROWS; j++) {
            if (board.getTile(i, j) < 0) {
                int v = board.getTile(i, j);
                v *= -1;
                v /= 10;
                board.setTile(i, j, v);
            }
        }
    }
}
```

`getNextIdAndRemove()` är metoden som tillsammans med `getNextPieceId()` bestämmer vilken del som kommer att genereras. Antingen genereras ett slumpmässigt tal och returneras eller så avläses nästa element i en lista,*`numberList`*, och tas bort. Listan har skapat genom att en in-ström av klassen `Scanner` har avläst nummer från en textfil med pseudo-slumpmässigt genererade nummer. Vid arbete med genetiska algoritmer är det en fördel om algoritmen kan utvärdera sig mot samma datasett hela tiden, därav möjligheten att använda förutbestämda nummer.

```
public int getNextIdAndRemove() {
    if (generateRandomNumber) {
        nextPieceId = (int) (1 + Math.random() * 7);
        return nextPieceId;
    } else {
        int i = numberList.get(0);
        numberList.remove(0);
        return i;
    }
}
```

2.3.2 Klassen `Board`

`getNumberOfHoles()`, `getTotalHeight()`, `getNumberOfLinesCleared()`, och `getBumpiness()` är metoder som används av den genetiska algoritmen för att skapa en överblick över hur spelbrädet ser ut. `getBumpiness()` t.ex. fungerar adderar absolutbeloppet av alla kolumner som ligger brevid varandra. Andra metoder av intresse är metoderna för att flytta och rotera den aktiva delen. Metoderna för att flytta delen, som `moveLeft()` (se nedan), fungerar genom att flytta alla rutor som har ett negativt värde (d.v.s. de som är en del av den aktiva delen) ett steg åt vänster, höger eller nedåt.

```
public void moveLeft() {
    for (int i = 0; i < NOCOLLUMNS; i++) {
        for (int j = 0; j < NOROWS; j++) {
            if (getTile(i, j) < 0) {
                setTile(i - 1, j, getTile(i, j));
                setTile(i, j, 0);
            }
        }
    }
}
```

Rotation av den aktiva delen är mer omständligt. `rotate()` skapar först en 2×2, 3×3, eller en 4×4 matris (beroende på del) som den fyller med alla rutor från det övre vänstra hörnet av delen. Därefter skapar den en ny matris som den kopierar alla aktiva delarna till en ny matris, *`rotatedPiece`*. Därefter roterar den *`rotatedPiece`* och tar bort alla tomma rader och kolumner. Sedan undersöker metoden om den kan ersätta den nuvarande delen med en roterad del utan att det skapar konflilkt med de andra rutorna. Om det inte skpar en

konflikt tas alla aktiva delarna bort från brädet och *rotatedPiece* appliceras.

```
public boolean rotate() {
    int[][] matrix;
    int dim;
    switch (getPieceId()) {
        case -1:
            dim = 4;
            break;
        case -4:
            dim = 2;
            break;
        default:
            dim = 3;
            break;
    }

    matrix = new int[dim][dim];
    int xMin = NOCOLLUMNS;
    int yMin = NOROWS;

    //Gets the coordinates of the topleftmost corner of the matrix.
    for (int x = 0; x < NOCOLLUMNS; x++) {
        for (int y = 0; y < NOROWS; y++) {
            if (getTile(x, y) < 0) {
                if (xMin > x) {
                    xMin = x;
                }
                if (yMin > y) {
                    yMin = y;
                }
            }
        }
    }

    //Assigns values to the matrix
    for (int x = 0; x < dim; x++) {
        for (int y = 0; y < dim; y++) {
            try {
                matrix[y][x] = getTile(x + xMin, y + yMin);
            } catch (ArrayIndexOutOfBoundsException e) {
                return false;
            }
        }
    }

    //Creates a matrix with only the piece and removes it from the
        base matrix
    int[][] rotatedPiece = new int[dim][dim];
    for (int x = 0; x < dim; x++) {
        for (int y = 0; y < dim; y++) {
            if (matrix[y][x] < 0) {
                rotatedPiece[y][x] = matrix[y][x];
                matrix[y][x] = 0;
```

```
            } else {
                rotatedPiece[y][x] = 0;
            }
        }
    }

    //Rotates the rotatedPiece array 90 degrees right.
    rotatedPiece = rotateRight(rotatedPiece);

    //Removes empty columns
    rotatedPiece = removeEmptyRowsAndCols(rotatedPiece);

    for (int y = 0; y < rotatedPiece.length; y++) {
        for (int x = 0; x < rotatedPiece[0].length; x++) {
            if (matrix[y][x] > 0 && rotatedPiece[y][x] < 0) {
                return false;
            } else {
                matrix[y][x] = rotatedPiece[y][x];
            }
        }
    }

    //Removes the active pieces from the board.
    for (int x = 0; x < NOCOLLUMNS; x++) {
        for (int y = 0; y < NOROWS; y++) {
            if (getTile(x, y) < 0) {
                setTile(x, y, 0);
            }
        }
    }

    //Applies the matrix to the board
    for (int y = 0; y < rotatedPiece.length; y++) {
        for (int x = 0; x < rotatedPiece[0].length; x++) {
            setTile(xMin + x, yMin + y, rotatedPiece[y][x]);
        }
    }

    return true;
}
```

2.4 Konstruktion av den genetiska algoritm

Som tidigare nämnt fungerar en genetisk algoritm genom att i första steget initialisera en population av individer. Det andra steget är att utvärderar individernas lämplighet. Om någon av individerna är tillräckligt lämplig avslutas programmet. Algoritmen består av klassen `GeneticAlgorithm` som i sin tur består av huvudsak av generationen *`currentGeneration`* av klassen `Generation`. Klassen `Generation` fungerar som en lista med metoder för att sortera och lägga till individer av klassen `Genome`. `Genome` beskriver individer-

na och består av vikterna (de heuristiska dragen) samt id, lämplighets, generation och antal drag gjorda.

Som figur 1.5 (s. 5) visar finns det fem stadier i en genetisk algoritm, initialisering, utvärdering, stop villkor, selektion och mutation och korsning.

1. **Initialisering.** Initialiseringen hanteras av metoden `initalization()` som skapar den den första generationen och ser till att algoritmen kör tills stoppkriteriet är uppfyllt.
2. **Utvärdering.** Utvärderingen representeras av metoden `loopThroughGeneration()`. Metoden går igenom alla individer i generationen och utvärderar dem mot Tetris.
3. **Stoppkriterium.** Villkoret är i detta fall att algoritmen ska köra tills det att rätt antal generationer har kört. Villkoret beskrivs av metoden `stopCriteriaIsFulfilled()`.
4. **Selektion.** Selektionen sker i metoden `evolve()` och är relativt simpel då den endast väljer ut en individ från den bättre halvan av generationen. Den bästa individen i den föregående generationen ingår även i den nya generationen för att resultatet inte ska kunna bli sämre efter den nya generationen.
5. **Mutation och korsning.** Mutation och korsning sker i två steg. Först skapas den nya generationen med hjälp av korsning. Den bästa individen från föregående generation läggs också till men korsas eller muteras inte. Individerna som bildar den nya generationen kan delas in i fyra grupper, den bästa från föregående generation, korsningar, helt nyskapade och mutationer av den bästa. I snitt 75% av de nya individerna är korsningar, 12,5% helt nyskapade och 12,5% mutationer av den bästa. Anledningen till att det skapas nya är för att genpolen kan bli för homogen annars vilket kan resultera i att utvecklingen når en platå för tidigt. Mutationerna av den bästa läggs till då de förhoppningsvis kan vara bättre än den bästa.
   (a) **Korsning.** I korsning skapas det en ny individ som har 50% sannolikhet att ärva varje vikt från sin förälder.
   (b) **Mutation.** När ett genom muteras finns det en sannolikhet, mutationssannolikheten, att vikten muteras för varje vikt. Om vikten (som är ett heuristiskt drag) ska muteras adderas eller subtraheras ett slumpmässigt tal mellan -1 och 1 multiplicerat med mutationssteget.

2.5 Klasser och metoder av intresse för den genetiska algoritmen

Den genetiska algoritmen består av klasserna `Genetic Algorithm`, `Generation`, `Genome` och `Move`. Dock är `Generation`, `Genome`, och `Move` av mindre intresse då de endast lagrar data och inte gör några saker själv. Kort sagt lagrar

`Generation` alla individer, av klassen `Genome`. `Genome` innehåller varje individs lämplighet och dess vikter för de olika heuristiska dragen. `Move` lagrar ett visst drags poäng och dess kombination av drag.

2.5.1 Klassen `Genetic Algorithm`

Metoden `loopThroughGeneration()`

Metoden går igenom alla genomer i generationen och låter dem köra Tetris tills anntingen de a, når gränsen för hur många drag de kan göra, eller b, förlorar. I båda fallen får genomet som körde ett lämplighetspoäng (fitness) som motsvarar spelets poäng när antingen fall a eller b inträffar. Sedan återställs brädet och poängen och nästa genom upprepar proceduren.

```
private void loopThroughGeneration() {
    for (Genome g : currentGeneration.getGenomes()) {
        modelState.reset();

        while (getCurrentGenome().getMovesTaken() < MOVE_LIMIT) {
            int pieceId = getCurrentState().getNextIdAndRemove();
            if (getCurrentState().getBoard().canGenerateNew-
            Piece(pieceId)) {
                getCurrentState().getBoard().generatePiece(pieceId);
                makeNextMove();
                try {
                    Thread.sleep(delay);
                } catch (InterruptedException ex) {
                    System.out.println(ex);
                }
            } else {
                break;
            }
        }

        g.setFitness(modelState.getScore());
        currentGenomeId++;
        pcs.firePropertyChange("nextGenome", currentGenomeId - 1,
             currentGenomeId);
    }
}
```

Metoden `evolve()`

Metodens syfte är att utveckla den nuvarande generationen till nästa steg. Det görs genom att först skapa kopior av den nuvarande generationen. Detta görs två gånger för att dels lägga till i listan ***`previousGenerations`*** för att kunna analysera de olika generationernas individers lämplighet. Den andra, ***`modGen`*** används för att t.ex. ta bort den sämre halvan av befolkningen för att lättare kunna göra ett urval av lämpliga individer.

Därefter återställs den bästa individen i den föregående generationen så att den ska kunna delta i nästa generation. Sedan återställs *currentGeneration* och den bästa individen, *fittest*, läggs till.

Sedan tas de sämsta individerna bort och nya genereras och läggs till den nya generationen. Sättet som den nya generationen genereras på (se ovan) är inte optimalt då gränserna för hur många individer som ska genereras på vilket sätt är arbitärt framtagna. Dock borde de värdena ha någon större inverkan på resultatet då det är lika för alla generationer.

```
private void evolve() {
    Generation previousGen = currentGeneration.clone();
    Generation modGen = currentGeneration.clone();
    previousGenerations.add(previousGen);

    Genome fittest = currentGeneration.getFittest().clone();
    fittest.setId(fittest.getNewId(0, currentGenerationId + 1));
    fittest.setFitness(-1);
    fittest.setMovesTaken(0);

    currentGenerationId++;
    currentGeneration = new Generation(currentGenerationId);
    currentGeneration.addGenome(fittest);
    currentGenomeId = 0;

    //Remove the worst genomes
    for (int i = 0; i < POPULATION_LIMIT / 2; i++) {
        modGen.removeGenome(i);
    }

    //Creates new Genomes.
    int n = currentGeneration.size() - 1;
    while (currentGeneration.size() < POPULATION_LIMIT) {
        n++;
        if (Math.random() < 0.75) {
            Genome parent1 = modGen.getRandomGenome();
            Genome parent2 = modGen.getRandomGenome();
            currentGeneration.addGenome(makeChild(n, parent1,
                parent2));
        } else {
            System.out.println("evolve");
            if (Math.random() < 0.5) {
                currentGeneration.addGenome(new
                    Genome(currentGenerationId, n));
            } else {
                Genome g = fittest.clone();
                g.setId(g.getNewId(currentGenerationId, n));
                currentGeneration.addGenome(mutateGenome(g));
            }
        }
    }
    pcs.firePropertyChange("nextGeneration", previousGen,
        currentGeneration);
}
```

3. Resultat

Efter tester framkom det att cirka 26 individer per sekund kan utvärderas. Det skulle då ta cirka 430 timmar att utvärdera 40 000 000 individer. Därför har mutationssannolikheten och mutationsstegets intervall ökat från 0.05 till 0.1 då endast 10 000 000 individer behöver bearbetas då. Antal individer per generation minskas även från 100 till 10 och antal generationer till 50. Då behöver endast 500 000 individer utvärderas, vilket tar ca. 5.3 timmar.

Resultatet av den första undersökningen där individerna fick köra i 50 generationer var visade ett resultat där skillnaden mellan den bästa och den sämsta kombinationen av parametrar var cirka 2.4%. Därför beslutades det att två undersökningar till skulle utföras, en med 25 generationer per par av parametrar och en med 10. Resultaten av den första undersökningen (50 generationer) redovisas i tabell 5.2 (s. 25), den andra (25 generationer) i tabell 5.3 (s. 26), och den tredje (10 generationer) i tabell 5.4 (s. 26). Datan från undersökningarna och de mänskliga spelarna finns sammanställd i tabell 3.1 (s. 20).

Tabell 3.2 (s. 20) visar medelvärdet för mutationssannolikheten för alla mutationssteg. Tabellen visar att medelvärdet är högre om mutationssanolikheten är större.

Frågeställningarna som skulle besvaras var ”Hur förhåller sig en genetisk algoritms resultat i tetris till en mänsklig spelares resultat?” och ”Vilken kombination av värden på parametrarna mutationssteget och mutationssannolikhete ger bäst resultat?”. Svaret på den första frågan är att beroende på hur många generationer som simuleras är så algorimens resultat cirka 88 till 102% bättre än en mänsklig spelares om medelvärdena jämförs och 11 till 18% om de bästa värdena jämförs. Svaret på den andra frågan verkar vara att högre mutationssannolikheten har en större påverkan än vad mutattionssteget har. Tabell 3.3 (s. 21) och 3.2 (s. 20) visar att värdet på mutationssteget påverkar resultatet mindre (mellan 0,20% och 0,41% mellan de högsta och lägsta värdena) än mutationssannolikheten (mellan 1,4% och 2,3% mellan de högsta och lägsta värdena)

	Spelare	50 generationer	25 generationer	10 generationer
Minsta värde	238	5948	5708	5542
Största värde	5133	6091	5938	5683
Procentuell skillnad	2057%	2.4%	4.0%	2.5%
Absolut skillnad	4895	143	230	141
Medelvärde	3004	6058	5893	5654
Standard-avvikelse	1586	13.4	44.1	30.7
Relativ skillnad (Medel)	100	202	196	188
Relativ skillnad (Bästa)	100	118	116	111

Tabell 3.1: Jämförelse mellan de mänskliga spelarnas och de genetiska algorimernas resultat.

	50	25	10
0.0	5957	5801	5601
0.1	5979	5837	5604
0.2	5989	5859	5650
0.3	6010	5881	5659
0.4	6034	5898	5661
0.5	6045	5906	5668
0.6	6053	5921	5675
0.7	6062	5926	5680
0.8	6069	5929	5671
0.9	6075	5929	5670
1.0	6082	5933	5667
Skillnad (max, min)	2,1%	2,3%	1,4%

Tabell 3.2: Jämförelse mellan de olika mutationssanolikheternera. Värdena i tabellen är medelvärde av en rad, d.v.s. den mutationssanolikheten angiven i den här tabellens första kolumn, och alla mutationssteg. De tre kolumnerna är de olika undersökningarna.

	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	Skillnad (max, min)
50	6029	6028	6028	6028	6029	6030	6032	6034	6036	6039	6040	0,20%
25	5879	5884	5886	5889	5890	5894	5895	5897	5901	5902	5903	0,41%
10	5669	5647	5648	5651	5656	5654	5655	5655	5656	5658	5658	0,39%

Tabell 3.3: Jämförelse mellan de olika mutationssteget. Värdena i tabellen är medelvärde av en kolumn, d.v.s. det mutationssteget angivit i den här tabellens övre rad, och alla mutationssanolikheter. De tre raderna är de olika undersökningarna.

4. Diskussion

4.1 Resultatet

Resultaten från alla undersökningar visar samma sak, den genetiska algoritmen är bättre på Tetris än de mänskliga spelarna, oavsett vilken kombination av parametrar som används. Algoritmen producerar, som förväntat, sämre individer om färre generationer simuleras. Genom att studera datan verkar det som att mutattionssteget bidrar i en större utsträckning till ett högre resultat. Även mutationsssannolikheten påverkar resultatet men i mindre utsträckning. Trots att det finns en observerbar skillnad i alla testerna är skillnaden mellan de högsta och lägsta värdena i testerna endast mellan $2,4\%$ och $4,0\%$. Det är inte en speciellt signifikant skillnad även om den visar på att det finns en skillnad.

Även om skillnaden mellan de olika genetiska algoritmerna inte är stora är skillnaden mellan algorimen och spelarna det. Då medelvärdet av spelarnas resultat var 3004 och mellan 6058 och 5654. Den genetiska algoritmen är helt klart överlägsen (88 till 102% bättre) de mänskliga spelarna när medelvärdena jämförs. Om de bästa resultaten jämförs ser det bättre ut för människorna då det endast skiljer sig 11 till 18% mellan de bästa resultaten.

4.2 Begränsningar

För att förbättra resultatet ytterligare med de avgränsningar som finns skulle en möjlighet vara att optimera algoritmen. Det enklaste och förmodligen mest effektiva sättet skulle vara att begränsa algoritmen från att utvärdera individer som är alltför lika varandra. Individernas vikter består av datatypen `double` som har 16 decimaler. Under några av testerna verkade det som att endast några decimaler spelade någon roll för hur individen presterade. Det betyder att istället för att utvärdera alla i princip identiska individer skulle de kunna viktas istället. Det skulle tillåta att fler individer per generation och flera generationer skulle kunna utvärderas, vilket skulle ge ett bättre resultat.

En annan sak som skulle bättra resultatets kvalité är att testa med fler och bättre valda värden för mutationssteget, T.ex. Skulle värdena kunna följt en exponentiell utveckling för att se vilken storleksordning det optimala värdet ligger i. Efter att ha tagit reda på det skulle fler undersökningar inom det

bästa intervallet kunna göras för att på så sätt hitta ett bra resultat.

På tal om optimeringar är både algoritmen och spelet redan optimerade till viss grad, t.ex. roterar spelet delarna på ett sådant sätt att vissa sätt att spela på inte är möjligt. Algorimen undersöker inte heller alla möjliga sätt att placera de olika delarna på då det skulle ta alldeles för lång tid.

I denna rapport tränades algoritmen endast mot samma värden hela tiden då det inte finns något annat sätt att få konsekventa resultat hela tiden. Detta är ett s.k. statiskt datasett. Genetiska algorimer har överlag svårt att klara av s.k. dynamiska datasett (d.v.s. när datan ändras. T.ex. när bitarna inte kommer i samma ordnign varje gång). Lösningen som kan ha hittats i ett statiskt datasett kan vara irrelevant för alla andra datasett. Detta sågs till viss grad då en del tester faktiskt gjordes med dynamiska datasett men då mer än 1000 drag skulle behöva göras för att få ett acceptabelt resultat utfördes alla undersökningar med ett statiskt datasett.

4.3 Slutsatts

Sammanfattningsvis är maskinerna bättre Tetris-spelare än människor, vilket inte är speciellt underligt då algoritmen i princip har obegränsat med tid att planera sitt drag under för att sedan kunna utföra det felfritt. Människorna ligger på så sätt i underläge då de varken har obegränsat med tid eller kan utföra sin plan felfritt. Det är svårt att göra några starka uttalanden om vilka värden för mutationssanolikhet och mutationssteg som är de bästa då de, a) endast skilljer sig ett fåtal procent mellan de bästa och sämsta resultaten, och b) ett något underligt beteende observerades i datan där högre mutationssteg och mutationssannolilhet ledde till bättre resultat även om mutationssannolikheten eller mutationssteget var noll.

Litteratur

[Wik19a] Wikipedia contributors. *Tetris — Wikipedia, The Free Encyclopedia*. [Online; accessed 31-Mars-2019]. 2019. URL: https://en.wikipedia.org/wiki/Tetris.

[Wik19b] Wikipedia contributors. *Tetris — Wikipedia, The Free Encyclopedia*. [Online; accessed 3-Mars-2019]. 2019. URL: https://en.wikipedia.org/wiki/Genetic_algorithm.

[Idr17] Idrees Hassan. *TetNet*. [Online; accessed 15-November-2018]. 2017. URL: https://github.com/IdreesInc/TetNet.

[Sir17] Siraj Raval. *How to Make an Evolutionary Tetris AI*. [Online; accessed 15-November-2018]. 2017. URL: https://youtu.be/xLHCMMGuNOQ.

5. Bilagor

5.1 Övrigt

	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.0											
0.1											
0.2											
0.3											
0.4											
0.5											
0.6											
0.7											
0.8											
0.9											
1.0											

Tabell 5.1: Demonstration av hur resultatet kommer att presenteras. Mutationssannolikhet nedåt, mutationssteg i sidled.

	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.0	5978	5963	5961	5955	5954	5948	5948	5948	5955	5958	5963
0.1	5966	5970	5972	5973	5983	5983	5983	5982	5987	5984	5982
0.2	5984	5986	5986	5985	5986	5987	5988	5989	5990	6000	6003
0.3	5999	5999	5998	6001	6005	6010	6012	6012	6019	6025	6032
0.4	6030	6029	6028	6030	6033	6033	6036	6038	6037	6037	6039
0.5	6038	6038	6037	6041	6040	6044	6047	6050	6051	6052	6052
0.6	6051	6050	6049	6050	6050	6051	6054	6054	6056	6058	6058
0.7	6057	6056	6056	6058	6058	6060	6065	6068	6069	6069	6069
0.8	6068	6067	6066	6065	6065	6067	6069	6070	6072	6071	6074
0.9	6072	6071	6071	6070	6071	6070	6074	6079	6080	6081	6081
1.0	6080	6079	6079	6078	6078	6078	6079	6081	6084	6090	6091

Tabell 5.2: Genomsnittlig poäng för olika mutationssteg och sannolikhet. Denna data tillhör undersökningen med 50 generationer. Mutationssannolikheten går nedåt och mutationssteget i sidled

	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.0	5708	5753	5765	5804	5793	5812	5813	5836	5850	5839	5837
0.1	5829	5831	5831	5825	5835	5836	5837	5840	5844	5848	5851
0.2	5841	5848	5848	5850	5858	5861	5864	5863	5869	5876	5873
0.3	5870	5873	5872	5874	5870	5880	5884	5885	5890	5895	5893
0.4	5892	5893	5895	5893	5896	5898	5898	5903	5904	5903	5903
0.5	5901	5902	5904	5903	5905	5905	5905	5908	5909	5912	5916
0.6	5915	5915	5915	5918	5921	5922	5923	5925	5925	5926	5928
0.7	5926	5926	5926	5925	5925	5925	5924	5923	5929	5929	5929
0.8	5929	5929	5929	5930	5930	5928	5930	5928	5927	5926	5929
0.9	5927	5927	5927	5929	5929	5929	5929	5928	5930	5929	5932
1.0	5931	5931	5931	5932	5933	5933	5933	5933	5934	5935	5938

Tabell 5.3: Genomsnittlig poäng för olika mutationssteg och sannolikhet. Denna data tillhör undersökningen med 25 generationer. Mutationssannolikheten går nedåt och mutationssteget i sidled

	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.0	5542	5542	5555	5578	5617	5593	5590	5582	5595	5591	5591
0.1	5591	5595	5576	5589	5598	5608	5613	5613	5607	5624	5628
0.2	5638	5638	5647	5644	5652	5647	5651	5652	5654	5661	5664
0.3	5663	5659	5658	5659	5658	5659	5661	5657	5658	5659	5660
0.4	5665	5660	5660	5659	5661	5660	5660	5662	5662	5662	5665
0.5	5666	5664	5665	5666	5664	5666	5668	5668	5668	5674	5676
0.6	5676	5675	5677	5678	5677	5673	5672	5671	5677	5673	5675
0.7	5677	5677	5680	5681	5682	5681	5682	5683	5682	5679	5676
0.8	5674	5671	5673	5669	5670	5673	5671	5671	5672	5672	5670
0.9	5670	5670	5671	5671	5671	5672	5672	5672	5671	5668	5666
1.0	5666	5665	5664	5664	5665	5667	5668	5669	5671	5670	5668

Tabell 5.4: Genomsnittlig poäng för olika mutationssteg och sannolikhet. Denna data tillhör undersökningen med 10 generationer. Mutationssannolikheten går nedåt och mutationssteget i sidled

5.2 Kod

5.2.1 Board

```
package Tetris;

import java.util.ArrayList;
import java.util.Arrays;

/**
 * @author Pontus Soderlund
 */
public class Board implements Cloneable {
```

Poäng	Antal drag
3192	883
3650	998
238	285
2505	905
2183	940
1891	712
310	406
3907	1001
2494	971
3165	1001
828	559
4210	1001
4091	905
5038	1001
5124	1001
2970	788
4117	1001
767	575
5133	1001
4267	814

Tabell 5.5: Spelares poäng och antal drag. Medelvärdet för poäng är 3004 och högsta är 5133. Medelvärdet för antal drag 839 och högsta är 1001 drag.

```
//**********************
// Global Variables
//**********************
//The width and height of the board
public static final int NOROWS = 20, NOCOLLUMNS = 10;
private int[][] board;

//**********************
// Constructor(s)
//**********************
public Board() {
    int[][] b = {
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

```
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};
    this.board = b;
}

public Board(int[][] board) {
    this.board = board;
}

//***********************
// Can Methods
//***********************
/**
 * Checks if a Tile with the specified Id can be generated
 *
 * @param pieceId the id of the tile to be generated
 * @return whether the tile can be created
 */
private boolean canGenerateNewPiece(int pieceId) {
    if (pieceId == 1) {      //I-Piece
        if (board[1][3] != 0 || board[1][4] != 0 || board[1][5] != 0
            || board[1][6] != 0) {
            return false;
        }
    } else if (pieceId == 2) { //J-Piece
        if (board[0][5] != 0 || board[1][5] != 0 || board[2][5] != 0
            || board[2][4] != 0) {
            return false;
        }
    } else if (pieceId == 3) { //L-Piece
        if (board[0][4] != 0 || board[1][4] != 0 || board[2][4] != 0
            || board[2][5] != 0) {
            return false;
        }
    } else if (pieceId == 4) { //O-Piece
        if (board[0][4] != 0 || board[0][5] != 0 || board[1][4] != 0
            || board[1][5] != 0) {
            return false;
        }
    } else if (pieceId == 5) { //S-Piece
        if (board[0][4] != 0 || board[1][4] != 0 || board[1][5] != 0
            || board[2][4] != 0) {
            return false;
        }
    } else if (pieceId == 6) { //Z-Piece
        if (board[0][5] != 0 || board[1][5] != 0 || board[1][4] != 0
            || board[2][4] != 0) {
            return false;
        }
```

```
    } else if (pieceId == 7) { //T-Piece
        if (board[0][4] != 0 || board[1][3] != 0 || board[1][4] != 0
             || board[1][5] != 0) {
            return false;
        }
    }
    return true;
}

/**
 * Checks if the tile with the coordinates newX, newY is occupied.
 *
 * @param newX the x-coordinate to move to
 * @param newY the y-coordinate to mote to
 * @return false if it is occupied, else true.
 */
private boolean canMoveTo(int newX, int newY) {
    return getTile(newX, newY) <= 0;
}

/**
 * Checks if it is possible to move the current piece in right, left
     or down.
 *
 * @param direction the direction in which to move.
 * @return false if it is not possible. True if it is.
 */
public boolean canMove(String direction) {
    switch (direction) {
        case "right":
            for (int i = 0; i < NOCOLLUMNS; i++) {
                for (int j = 0; j < NOROWS; j++) {
                    if (getTile(i, j) < 0) {
                        if (i < NOCOLLUMNS - 1) {
                            if (getTile(i + 1, j) > 0) {
                                return false;
                            }
                        } else if (i == 9) {
                            return false;
                        }
                    }
                }
            }
            break;
        case "left":
            for (int i = 0; i < NOCOLLUMNS; i++) {
                for (int j = 0; j < NOROWS; j++) {
                    if (getTile(i, j) < 0) {
                        if (i > 0) {
                            if (getTile(i - 1, j) > 0) {
                                return false;
                            }
                        } else if (i == 0) {
                            return false;
```

```
                    }
                }
            }
        }
        break;
    case "down":
        for (int i = NOROWS - 1; i >= 0; i--) {
            for (int j = 0; j < NOCOLLUMNS; j++) {
                if (getTile(j, i) < 0) {
                    if (i != NOROWS - 1) {
                        if (getTile(j, i + 1) > 0) {
                            return false;
                        }
                    } else if (i == NOROWS - 1) {
                        return false;
                    }
                }
            }
        }
        break;
    }
    return true;
}

//***********************
// Move Methods
//***********************
public boolean move(String direction) {
    switch (direction) {
        case "rotate":
            return rotate();
        case "right":
            if (canMove(direction)) {
                moveRight();
                return true;
            }
            break;
        case "left":
            if (canMove(direction)) {
                moveLeft();
                return true;
            }
            break;
        case "down":
            if (canMove(direction)) {
                moveDown();
                return true;
            }
            break;
    }
    return false;
}

/**
```

```
 * Rotates the current piece 90 degrees to the right
 *
 * @return false if the tile could not be rotated.
 */
private boolean rotate() {
    int[][] matrix;
    int dim;
    switch (getPieceId()) {
        case -1:
            dim = 4;
            break;
        case -4:
            dim = 2;
            break;
        default:
            dim = 3;
            break;
    }

    matrix = new int[dim][dim];
    int xMin = NOCOLLUMNS;
    int yMin = NOROWS;

    //Gets the coordinates of the topleftmost corner of the matrix.
    for (int x = 0; x < NOCOLLUMNS; x++) {
        for (int y = 0; y < NOROWS; y++) {
            if (getTile(x, y) < 0) {
                if (xMin > x) {
                    xMin = x;
                }
                if (yMin > y) {
                    yMin = y;
                }
            }
        }
    }

    //Assigns values to the matrix
    for (int x = 0; x < dim; x++) {
        for (int y = 0; y < dim; y++) {
            try {
                matrix[y][x] = getTile(x + xMin, y + yMin);
            } catch (ArrayIndexOutOfBoundsException e) {
                return false;
            }
        }
    }

    //Creates a matrix with only the piece and removes it from the
        base matrix
    int[][] rotatedPiece = new int[dim][dim];
    for (int x = 0; x < dim; x++) {
        for (int y = 0; y < dim; y++) {
            if (matrix[y][x] < 0) {
```

```
                rotatedPiece[y][x] = matrix[y][x];
                matrix[y][x] = 0;
            } else {
                rotatedPiece[y][x] = 0;
            }
        }
    }

    //Rotates the rotatedPiece array 90 degrees right.
    rotatedPiece = rotateRight(rotatedPiece);

    //Removes empty columns
    rotatedPiece = removeEmptyRowsAndCols(rotatedPiece);

    for (int y = 0; y < rotatedPiece.length; y++) {
        for (int x = 0; x < rotatedPiece[0].length; x++) {
            if (matrix[y][x] > 0 && rotatedPiece[y][x] < 0) {
                return false;
            } else {
                matrix[y][x] = rotatedPiece[y][x];
            }
        }
    }

    //Removes the active pieces from the board.
    for (int x = 0; x < NOCOLLUMNS; x++) {
        for (int y = 0; y < NOROWS; y++) {
            if (getTile(x, y) < 0) {
                setTile(x, y, 0);
            }
        }
    }

    //Applies the matrix to the board
    for (int y = 0; y < rotatedPiece.length; y++) {
        for (int x = 0; x < rotatedPiece[0].length; x++) {
            setTile(xMin + x, yMin + y, rotatedPiece[y][x]);
        }
    }

    return true;
}

/**
 * Moves the current piece one step to the right
 */
private void moveRight() {
    for (int i = NOCOLLUMNS - 1; i >= 0; i--) {
        for (int j = 0; j < NOROWS; j++) {
            if (getTile(i, j) < 0) {
                setTile(i + 1, j, getTile(i, j));
                setTile(i, j, 0);
            }
        }
```

```
    }
}

/**
 * Moves the current piece one step to the left
 */
private void moveLeft() {
    for (int i = 0; i < NOCOLLUMNS; i++) {
        for (int j = 0; j < NOROWS; j++) {
            if (getTile(i, j) < 0) {
                setTile(i - 1, j, getTile(i, j));
                setTile(i, j, 0);
            }
        }
    }
}

/**
 * Moves the current piece one step down
 */
private void moveDown() {
    for (int i = NOROWS - 1; i >= 0; i--) {    //Y
        for (int j = NOCOLLUMNS - 1; j >= 0; j--) { //X
            if (getTile(j, i) < 0) {
                setTile(j, i + 1, getTile(j, i));
                setTile(j, i, 0);
            }
        }
    }
}

//***********************
// Support Methods
//***********************
/**
 * Deselects all active pieces by setting their value, if negative
     to a positive
 * counterpart
 */
public void deselectPiece() {
    for (int i = 0; i < NOCOLLUMNS; i++) {
        for (int j = 0; j < NOROWS; j++) {
            if (getTile(i, j) < 0) {
                int v = getTile(i, j);
                setTile(i, j, Math.abs(v));
            }
        }
    }
}

/**
 * Clears the rows that are full
 *
 * @return Number of rows cleare
```

```
 */
public int clearRows() {
    int rowsCleared = 0;
    for (int i = 0; i < NOROWS; i++) {
        boolean canClearLine = true;
        for (int j = 0; j < NOCOLLUMNS; j++) {
            if (getTile(j, i) == 0) {
                canClearLine = false;
                break;
            }
        }
        if (canClearLine) {
            for (int j = 0; j < NOCOLLUMNS; j++) {
                board[i][j] = 0;
            }

            for (int j = i; j > 0; j--) {
                for (int k = 0; k < NOCOLLUMNS; k++) {
                    board[j][k] = getTile(k, j - 1);
                }
            }
            rowsCleared++;
        }
    }
    return rowsCleared;
}

/**
 * Generates a new Piece with the id n.
 *
 * @param n is the number of the piece that is to be generated
 * @return true if the piece can was generated. Else false.
 */
public boolean generatePiece(int n) {
    if (canGenerateNewPiece(n)) {
        switch (n) {
            case 1:
                //I Piece
                board[1][3] = -1;
                board[1][4] = -1;
                board[1][5] = -1;
                board[1][6] = -1;
                break;
            case 2:
                //J-Piece
                board[0][5] = -2;
                board[1][5] = -2;
                board[2][4] = -2;
                board[2][5] = -2;
                break;
            case 3:
                //L-Piece
                board[0][4] = -3;
                board[1][4] = -3;
```

```
                board[2][4] = -3;
                board[2][5] = -3;
                break;
            case 4:
                //O-Piece
                board[0][4] = -4;
                board[0][5] = -4;
                board[1][4] = -4;
                board[1][5] = -4;
                break;
            case 5:
                //S-Piece
                board[0][4] = -5;
                board[1][4] = -5;
                board[1][5] = -5;
                board[2][5] = -5;
                break;
            case 6:
                //Z-Piece
                board[0][5] = -6;
                board[1][5] = -6;
                board[1][4] = -6;
                board[2][4] = -6;
                break;
            case 7:
                //T-Piece
                board[0][4] = -7;
                board[1][3] = -7;
                board[1][4] = -7;
                board[1][5] = -7;
                break;
            default:
                break;
        }
        return true;
    } else {
        return false;
    }
}

/**
 * Sets all values in the board to 0.
 */
public void resetBoard() {
    for (int i = 0; i < NOCOLLUMNS; i++) {
        for (int j = 0; j < NOROWS; j++) {
            setTile(i, j, 0);
        }
    }
}

/**
 * Checks to see if it would create a conflict when adding two
     matrices togheter.
```

```
     *
     * @param matrix1 the matrix on which to add matrix2
     * @param matrix2 the matrix which is added.
     * @return false if it creates conflict else not.
     */
    private boolean createdConflict(int[][] matrix1, int[][] matrix2) {
        for (int y = 0; y < matrix1.length; y++) {
            for (int x = 0; x < matrix1[0].length; x++) {
                if (matrix1[y][x] > 0 && matrix2[y][x] < 0) {
                    return true;
                } else {
                    matrix1[y][x] = matrix2[y][x];
                }
            }
        }
        return false;
    }

    /**
     * Removes all collumns and rows where all elements are zero.
     *
     * @param matrix the matrix from which to remove.
     * @return the matrix without empty collumns and rows.
     */
    private int[][] removeEmptyRowsAndCols(int[][] matrix) {
        //Removes all empty rows from rotatedPiece
        ArrayList<int[]> rp = new ArrayList<>(Arrays.asList(matrix));
        for (int i = 0; i < 3; i++) {
            for (int y = 0; y < rp.size(); y++) {
                boolean rowIsEmpty = true;
                for (int x = 0; x < rp.get(0).length; x++) {
                    if (rp.get(y)[x] != 0) {
                        rowIsEmpty = false;
                    }
                }
                if (rowIsEmpty) {
                    rp.remove(y);
                }
            }
        }

        //Removes all empty collumns from the matrix
        for (int n = 0; n < 3; n++) {
            for (int x = 0; x < rp.get(0).length; x++) {
                boolean colIsEmpty = true;
                for (int y = 0; y < rp.size(); y++) {
                    if (rp.get(y)[x] < 0) {
                        colIsEmpty = false;
                        break;
                    }
                }

                if (colIsEmpty) {
                    int[][] k = new int[rp.size()][rp.get(0).length - 1];
```

```
                for (int i = 0; i < rp.size(); i++) {
                    for (int j = 0; j < rp.get(i).length - 1; j++) {
                        if (j >= x) {
                            k[i][j] = rp.get(i)[j + 1];
                        }
                    }
                }
                rp = new ArrayList<>(Arrays.asList(k));
                break;
            }
        }
    }

    int[][] retur = new int[rp.size()][rp.get(0).length];
    for (int i = 0; i < rp.size(); i++) {
        System.arraycopy(rp.get(i), 0, retur[i], 0, rp.get(i).length);
    }

    return retur;
}

/**
 * Rotates a square matrix 90 degrees to the right.
 *
 * @param matrix the matrix which to rotate.
 * @return the rotated matrix.
 */
private int[][] rotateRight(int[][] matrix) {
    matrix = transpose(matrix);
    for (int y = 0; y < matrix.length; y++) {
        for (int x = 0; x < matrix.length / 2; x++) {
            int temp = matrix[y][matrix.length - x - 1];
            matrix[y][matrix.length - x - 1] = matrix[y][x];
            matrix[y][x] = temp;
        }
    }
    return matrix;
}

/**
 * Transposes a matrix
 *
 * @param matrix the matrix to be transposed.
 * @return the transposed matrix.
 */
private int[][] transpose(int[][] matrix) {
    int[][] transpose = new int[matrix.length][matrix.length];

    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix.length; j++) {
            transpose[i][j] = matrix[j][i];
        }
    }
```

```
    return transpose;
}

/**
 * Creates a deep copy of the array.
 *
 * @param original the array to be copied
 * @return the copied array.
 */
private int[][] deepCopy(int[][] original) {
    if (original == null) {
        return null;
    }

    final int[][] result = new int[original.length][];
    for (int i = 0; i < original.length; i++) {
        result[i] = Arrays.copyOf(original[i], original[i].length);
    }
    return result;
}

@Override
public String toString() {
    String s = "  |   1   2   3   4   5   6   7   8   9   10 \n";
    s += "-------------------------------------------------------- \n";
    for (int i = 0; i < NOROWS; i++) {
        if (i < 10) {
            s += "0";
        }
        s += i;
        s += " | ";
        for (int j = 0; j < NOCOLLUMNS; j++) {
            if (board[i][j] >= 0) {
                s += " ";
            }
            s += board[i][j];
            s += ", ";
        }
        s += "\n";
    }
    return s;
}

@Override
public Board clone() {
    Board board = new Board();
    board.setBoard(deepCopy(this.board));
    return board;
}

//***********************
// Getter Methods
//***********************
public int[][] getBoard() {
```

```
    return board;
}

/**
 * Returns the tile at position (x, y).
 *
 * @param x The Tiles X-cooridnate (Collumn id) (0-9 by default)
 * @param y The Tiles Y-coordinate (Row id) (0-19 by default)
 * @return the id of the tile 1 to 7 if it is set, 0 if there is no
     tile and -1 to -7
 * if it it is the current tile.
 */
public int getTile(int x, int y) {
    return board[y][x];
}

/**
 * Returns the id of the currently active piece (-1 to -7).
 *
 * @return
 */
public int getPieceId() {
    for (int i = 0; i < NOCOLLUMNS; i++) {
        for (int j = 0; j < NOROWS; j++) {
            if (getTile(i, j) < 0) {
                return getTile(i, j);
            }
        }
    }
    return 0;
}

/**
 * Returns the height of the highest collumn currently on the board.
 *
 * @return the height of the highest collumn.
 */
public int getHeightOfHighestCollumn() {
    int height = 0;
    for (int i = 0; i < NOCOLLUMNS; i++) {
        if (getHeightOfColumn(i) > height) {
            height = getHeightOfColumn(i);
        }
    }

    return height;
}

/**
 * A Hole is when there's one or more tiles above an empty space in
     a column.
 *
 * @return the number of holes-
 */
```

```
public int getNumberOfHoles() {
    int n = 0;
    for (int i = 0; i < NOCOLLUMNS; i++) {
        for (int j = NOROWS - 1; j >= 0; j--) {
            if (0 == getTile(i, j)) {
                for (int k = j; k >= 0; k--) {
                    int o = getTile(i, k);
                    if (o != 0 && o > 0) {
                        n++;
                        break;
                    }
                }
            }
        }
    }
    return n;
}

/**
 * The sum of the height of all the collumns on the board
 *
 * @return
 */
public int getTotalHeight() {
    int n = 0;
    for (int i = 0; i < NOCOLLUMNS; i++) {
        n += getHeightOfColumn(i);
    }
    return n;
}

/**
 * Number of lines that can be cleared on the board
 *
 * @return
 */
public int getNumberOfLinesCleared() {
    int noRowsCleared = 0;
    for (int i = 0; i < NOROWS; i++) {
        boolean canClearLine = true;
        for (int j = 0; j < NOCOLLUMNS; j++) {
            int t = getTile(j, i);
            if (t == 0) {
                canClearLine = false;
                break;
            }
        }
        if (canClearLine) {
            noRowsCleared++;
        }
    }
    return noRowsCleared;
}
```

```
/**
 * How bumby the board is, i.e. the difference between all the
     adjacent collumns
 * heights summed.
 *
 * @return
 */
public int getBumpiness() {
    int n = 0;
    for (int i = 0; i < NOCOLLUMNS - 1; i++) {
        n += Math.abs(getHeightOfColumn(i) - getHeightOfColumn(i +
            1));
    }
    return n;
}

/**
 * Height of the collumn at the given index
 *
 * @param index of the collumn (0-9 by default)
 * @return
 */
public int getHeightOfColumn(int index) {
    for (int i = 0; i < NOROWS; i++) {
        if (getTile(index, i) > 0) {
            return 20 - i;
        }
    }
    return 0;
}

//***********************
// Setter Methods
//***********************
/**
 * Sets the specified tile to the value specified.
 *
 * @param x the x-coordinate
 * @param y the y-coordinate
 * @param value the new value
 * @return false if the operation was unsucessfull.
 */
private boolean setTile(int x, int y, int value) {
    if (x > 9 || x < 0 || y > 19 || y < 0 || getTile(x, y) > 0) {
        return false;
    }
    board[y][x] = value;
    return true;
}

public void setBoard(int[][] board) {
    this.board = board;
}
```

```
}
```

5.2.2 Coordinator

```
package Coordinator;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.border.*;
import javax.swing.event.ChangeEvent;
import Tetris.GameView;
import Tetris.MyGameModel;
import ga.GeneticAlgorithm;
import ga.GeneticAlgorithmView;
import ga.Genome;

/**
 * @author Pontus Soderlund
 */
public final class Coordinator extends JFrame implements ActionListener {

    //************************
    // Variables
    //************************
    public static double MUTATION_RATE = 0.05;
    public static double MUTATION_STEP = 0.1;

    public static final int GENERATION_LIMIT = 10;
    public static final int POPULATION_LIMIT = 10;
    public static final int MOVE_LIMIT = 1000;
    public static final int FALL_FREQUENZY = 150;
    public static final int DISTRUBUTION_RESOLUTION = 10;
    public static final Dimension STANDARD_DIMENSIONS = new
        Dimension(300, 600);
    public static final Color BACKROUND_COLOR = new Color(230, 230, 230);
    public static final Color GRID_COLOR = new Color(100, 100, 100);

    private Writer generationWriter;
    private Writer eliteWriter;
    private Timer updateTimer;
    private int viewSpeed;

    private Genome fittest;
    private final GeneticAlgorithm ga;
    private final MyGameModel model;

    private JTextField txfGenomeToBePlayed;
    private JSlider sliSpeed;

    //Panels
    private GameView gameView;
    private GeneticAlgorithmView generationView;
    private JPanel controlPanel;
```

```
//***********************
// Constructor(s)
//***********************
public Coordinator(double mutationRate, double mutationStep,
        boolean createWindow, boolean player) {
    Coordinator.MUTATION_RATE = mutationRate;
    Coordinator.MUTATION_STEP = mutationStep;

    model = new MyGameModel(false, 10000);
    ga = new GeneticAlgorithm(model);
    gameView = null;

    if (createWindow) {
        generationView = new GeneticAlgorithmView(ga,
            STANDARD_DIMENSIONS);
        gameView = new GameView(model, true, STANDARD_DIMENSIONS,
            STANDARD_DIMENSIONS);
        controlPanel = new JPanel();
        viewSpeed = 10;

        fixLayoutNStuff();
        addKeyBindings();

        pack();
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        requestFocus();
        gameView.setDrawable(false);
    }

    boolean loop;
    loop = false;
    loop = true;

    if (player) {
        updateTimer = new Timer(FALL_FREQUENZY, this);
        updateTimer.start();
        gameView.setDrawable(true);
    } else if (loop) {
        ga.setDelay(0);
        fittest = ga.initalize();
    } else {
        ga.setDelay(viewSpeed);
        //playGenome();
    }
}

//***********************
// Main Methods
//***********************
public Genome getFittest() {
    return fittest;
}
```

```
    public void playGenome(double a, double b, double c, double d,
        double e) {
//       ga.setCurrentGenome(a, b, c, d, e);
//       System.out.println(ga.getCurrentGenome().toString());
//
//       while (ga.getCurrentGenome().getMovesTaken() < MOVE_LIMIT) {
//           int pieceId = ga.getCurrentState().getNextIdAndRemove();
//           if
    (ga.getCurrentState().getBoard().canGenerateNewPiece(pieceId)) {
//               ga.getCurrentState().getBoard().generatePiece(pieceId);
//           } else {
//               break;
//           }
//           ga.makeNextMove();
//       }
    }

    public void fixLayoutNStuff() {
        GridBagLayout lm = new GridBagLayout();
        this.setLayout(lm);
        GridBagConstraints gbc = new GridBagConstraints();

        gbc.fill = GridBagConstraints.NONE;
        gbc.gridx = 0;
        gbc.gridy = 0;
        this.add(gameView, gbc);

        gbc.fill = GridBagConstraints.BOTH;
        gbc.gridx = 1;
        gbc.gridy = 0;
        this.add(generationView, gbc);

        gbc.fill = GridBagConstraints.BOTH;
        gbc.gridx = 2;
        gbc.gridy = 0;
        this.add(controlPanel, gbc);

        double d = 0.05;
        LineBorder lineBorder = new LineBorder(GRID_COLOR, 1);
        EmptyBorder emptyBorder = new EmptyBorder((int) (d *
            STANDARD_DIMENSIONS.getWidth()),
               (int) (d * STANDARD_DIMENSIONS.getHeight()), (int) (d *
                   STANDARD_DIMENSIONS.getWidth()),
               (int) (d * STANDARD_DIMENSIONS.getHeight()));
        CompoundBorder border = new CompoundBorder(lineBorder,
            emptyBorder);
        gameView.setBorder(lineBorder);
        generationView.setBorder(border);
        controlPanel.setBorder(border);

        generationView.setBackground(BACKROUND_COLOR);
        controlPanel.setBackground(BACKROUND_COLOR);
```

```
    addToControlPanel();
}

//***********************
// Support Methods
//***********************
/**
 * Appends the relevant data to the relevant files...
 */
public void print() {
    try {
        generationWriter = new BufferedWriter(new OutputStreamWriter(
                new FileOutputStream(new File("Generations.txt"),
                    true), "UTF-8"));
        generationWriter.flush();
        eliteWriter = new BufferedWriter(new OutputStreamWriter(
                new FileOutputStream(new File("Elites.txt"), true),
                    "UTF-8"));
        eliteWriter.flush();

        generationWriter.append(ga.getCurrentGeneration().toString());
        eliteWriter.append(ga.getCurrentGeneration().getFittest().toString());
        generationWriter.append("\n");
        eliteWriter.append("\n");

        generationWriter.close();
        eliteWriter.close();
    } catch (IOException ex) {
        System.out.println(ex.toString());
    }
}

/**
 * Calls the movement method in MyGameModel if the timer is running
     and there has been
 * fewer moves than the move limit.
 *
 * @param input the direction in which to move.
 */
private void keyInput(String input) {
    if (model.getNumberOfMoves() < MOVE_LIMIT) {
        switch (input) {
            case "rotate":
            case "left":
            case "down":
            case "right":
                if (updateTimer.isRunning()) {
                    model.movement(input, false);
                }
                break;
        }
    } else {
        model.lost();
    }
```

```
}

/**
 * Toggles the timer on or off
 */
public void toggleTimer() {
    if (updateTimer.isRunning()) {
        updateTimer.stop();
    } else {
        updateTimer.start();
    }
}

@Override
public void actionPerformed(ActionEvent ae) {
    if (model.getNumberOfMoves() > MOVE_LIMIT) {
        model.lost();
    }
    if (model.generatePiece(model.getNextPieceId()) &&
         model.getPieceId() >= 0) {

    } else if (model.fall()) {
        model.clearRows();
    } else {
        model.deselectPiece();
        if (!model.generatePiece(model.getNextIdAndRemove())) {
            model.lost();
        }
    }
}

//***********************
// Adders
//***********************
/**
 * Adds componennts to the controlPanel
 */
public void addToControlPanel() {
    BoxLayout layout = new BoxLayout(controlPanel, BoxLayout.Y_AXIS);
    controlPanel.setLayout(layout);

    txfGenomeToBePlayed = new JTextField();
    sliSpeed = new JSlider(0, 50);
    addButtons();

    sliSpeed.addChangeListener((ChangeEvent e) -> {
        viewSpeed = sliSpeed.getValue();
        ga.setDelay(viewSpeed);
    });
}

/**
 * Adds buttons to the controlPanel
 */
```

```
private void addButtons() {
    addButton(controlPanel, "Start", (evt) -> {

    });
    addButton(controlPanel, "Start GA", (evt) -> {

    });
    addButton(controlPanel, "Toggle Select moves", (evt) -> {
        if (gameView.isDrawAllMoves()) {
            gameView.setDrawAllMoves(false);
        } else {
            gameView.setDrawAllMoves(true);
        }
    });
    addButton(controlPanel, "Toggle Player", (evt) -> {

    });
    addButton(controlPanel, "Toggle View", (evt) -> {
        if (gameView.isDrawable()) {
            gameView.setDrawable(false);
        } else {
            gameView.setDrawable(true);
        }
    });
    addButton(controlPanel, "Print Elite", (evt) -> {

    });
    addButton(controlPanel, "Set Genome", (evt) -> {

    });
}

/**
 * Adds a button to the specified component.
 *
 * @param comp the component to add to
 * @param text button text
 * @param lambda the action to trigger when the button is pressed.
 */
private void addButton(JComponent comp, String text, ActionListener
     lambda) {
    JButton btn = new JButton(text);
    btn.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            lambda.actionPerformed(e);
        }
    });
    comp.add(btn);
}

/**
 * Adds key bindings to the frame
 */
```

```
private void addKeyBindings() {
    //Movement
    addKeyBinding(gameView, KeyEvent.VK_W, "rotate", evt -> {
        keyInput("rotate");
    });
    addKeyBinding(gameView, KeyEvent.VK_A, "moveLeft", evt -> {
        keyInput("left");
    });
    addKeyBinding(gameView, KeyEvent.VK_S, "moveDown", evt -> {
        keyInput("down");
    });
    addKeyBinding(gameView, KeyEvent.VK_D, "moveRight", evt -> {
        keyInput("right");
    });

    //Other
    addKeyBinding(gameView, KeyEvent.VK_ESCAPE, "quit", evt -> {
        System.exit(0);
    });
    addKeyBinding(gameView, KeyEvent.VK_L, "lost", evt -> {
        model.lost();
    });
    addKeyBinding(gameView, KeyEvent.VK_P, "pause", evt -> {
        toggleTimer();
    });
    addKeyBinding(gameView, KeyEvent.VK_T, "toggleView", evt -> {
        if (gameView.isDrawable()) {
            gameView.setDrawable(false);
        } else {
            gameView.setDrawable(true);
        }
    });
}

/**
 * Adds a key binding to the specifed component.
 *
 * @param comp the component to add to.
 * @param keyCode the key to take action to.
 * @param id the name of the binding.
 * @param lambda the action to trigger.
 */
private void addKeyBinding(JComponent comp, int keyCode, String id,
     ActionListener lambda) {
    InputMap im =
         comp.getInputMap(JComponent.WHEN_IN_FOCUSED_WINDOW);
    ActionMap ap = comp.getActionMap();

    im.put(KeyStroke.getKeyStroke(keyCode, 0, false), id);
    ap.put(id, new AbstractAction() {
        @Override
        public void actionPerformed(ActionEvent e) {
            lambda.actionPerformed(e);
        }
```

```
        });
    }

}
```

5.2.3 GameView

```
package Tetris;

import Coordinator.Coordinator;
import java.awt.*;
import java.beans.*;
import javax.swing.*;
import javax.swing.border.*;

/**
 * @author Pontus Soderlund
 */
public final class GameView extends JPanel implements
    PropertyChangeListener {

   //************************
   // Variables
   //************************
   private MyGameModel gameModel;
   private boolean drawable, drawAllMoves;
   private final DisplayPanel displayPanel;
   private final InfoPanel infoPanel;

   //************************
   // Constructor(s)
   //************************
   public GameView(MyGameModel gameModel, boolean drawable,
           Dimension displayDim, Dimension infoDim) {
       Dimension totDim = new Dimension((int) (displayDim.getWidth() +
           infoDim.getWidth()),
              (int) (displayDim.getHeight()));
       setModel(gameModel);
       setPreferredSize(totDim);
       setMinimumSize(totDim);

       displayPanel = new DisplayPanel(displayDim);
       infoPanel = new InfoPanel(infoDim);
       displayPanel.setBackground(Coordinator.BACKROUND_COLOR);
       infoPanel.setBackground(Coordinator.BACKROUND_COLOR);
       displayPanel.setBorder(new MatteBorder(0, 0, 0, 1,
           Coordinator.GRID_COLOR));
       double d = 0.05;
       EmptyBorder emptyBorder = new EmptyBorder((int) (d *
           infoDim.getWidth()),
              (int) (d * infoDim.getHeight()), (int) (d *
                  infoDim.getWidth()),
              (int) (d * infoDim.getHeight()));
       infoPanel.setBorder(emptyBorder);

       setLayout(new BoxLayout(this, BoxLayout.X_AXIS));
       add(displayPanel);
       add(infoPanel);
```

```
    this.drawable = drawable;
    this.drawAllMoves = false;
}

//***********************
// Main Metod
//***********************
/**
 * Calls the appropiate methods in the inner classes.
 *
 * @param pce The event on which to respond to.
 */
@Override
public void propertyChange(PropertyChangeEvent pce) {
    if (!drawable) {
    } else if (pce.getPropertyName().equals("thinkingMovement") &&
        !drawAllMoves) {
    } else {
        displayPanel.repaint();
        infoPanel.update();
    }
}

//***********************
// Getter Methods
//***********************
public MyGameModel getModel() {
    return gameModel;
}

public boolean isDrawable() {
    return drawable;
}

public boolean isDrawAllMoves() {
    return drawAllMoves;
}

//***********************
// Setter Methods
//***********************
public void setModel(MyGameModel gameModel) {
    this.gameModel = gameModel;
    if (this.gameModel != null) {
        this.gameModel.addPropertyChangeListener(this);
    }
}

public void setDrawable(boolean drawable) {
    this.drawable = drawable;
}

public void setDrawAllMoves(boolean drawAllMoves) {
```

```
        this.drawAllMoves = drawAllMoves;
    }

    //***********************
    // Inner Classes
    //***********************
    private class DisplayPanel extends JPanel {

        //***********************
        // Variables
        //***********************
        private final int panelWidth;
        private final int panelHeight;

        //***********************
        // Constructor(s)
        //***********************
        public DisplayPanel(Dimension dim) {
            this.panelWidth = (int) dim.getWidth();
            this.panelHeight = (int) dim.getHeight();
            setPreferredSize(dim);
            setMinimumSize(dim);
            setMaximumSize(dim);
        }

        //***********************
        // Drawing Methods
        //***********************
        /**
         * Draws the board, tiles and grid.
         */
        @Override
        protected void paintComponent(Graphics g) {
            if (drawable) {
                super.paintComponent(g);
                drawTiles(g);
                drawGrid(g);
            }
        }

        /**
         * Draws all the Tiles on the board
         */
        private void drawTiles(Graphics g) {
            Board board = gameModel.getBoard();
            for (int i = 0; i < Board.NOCOLLUMNS; i++) {
                for (int j = 0; j < Board.NOROWS; j++) {
                    drawTile(g, i, j, board.getTile(i, j));
                }
            }
        }

        /**
         * Draws a single Tile on the board when called
```

```
 *
 * @param x The X position of the Tile
 * @param y The Y position of the Tile
 * @param c If the Tile is the current Tile then true, else false
 */
private void drawTile(Graphics g, int x, int y, int tileId) {
    Color c = null;
    Color iColor = new Color(54, 224, 255);
    Color jColor = new Color(28, 118, 188);
    Color lColor = new Color(248, 147, 29);
    Color oColor = new Color(254, 227, 86);
    Color sColor = new Color(83, 213, 4);
    Color zColor = new Color(249, 35, 56);
    Color tColor = new Color(201, 115, 255);

    switch (Math.abs(tileId)) {
        case 0:
            return;
        case 1:
            c = iColor;
            break;
        case 2:
            c = jColor;
            break;
        case 3:
            c = lColor;
            break;
        case 4:
            c = oColor;
            break;
        case 5:
            c = sColor;
            break;
        case 6:
            c = zColor;
            break;
        case 7:
            c = tColor;
            break;
    }
    g.setColor(c);

    int a = (int) (((double) panelWidth / (double)
        Board.NOCOLLUMNS));
    int b = (int) (((double) panelHeight / (double)
        Board.NOROWS));
    g.fillRect(x * a, y * b, a, b);
}

/**
 * Draws the grid.
 */
private void drawGrid(Graphics g) {
    g.setColor(Coordinator.GRID_COLOR);
```

```
            int wMult = (int) ((double) panelWidth / (double)
                Board.NOCOLLUMNS);
            int hMult = (int) ((double) panelHeight / (double)
                Board.NOROWS);
            for (int i = 0; i < Board.NOCOLLUMNS; i++) {
                for (int j = 0; j < Board.NOROWS; j++) {
                    g.drawLine(i * wMult, 0, i * wMult, panelHeight);
                    g.drawLine(0, j * hMult, panelWidth, j * hMult);
                }
            }
        }
    }

    private class InfoPanel extends JPanel {

        //************************
        // Variables
        //************************
        private JLabel lblScore, lblMoves, lblButtons, lblBoard,
            lblController;

        //************************
        // Constructor(s)
        //************************
        public InfoPanel(Dimension dim) {
            setPreferredSize(dim);
            setMinimumSize(dim);
            setMaximumSize(dim);
            addComponents();
        }

        //************************
        // Main ethod
        //************************
        /**
         * Updates the values of the labels lblScore and lblMoves
         */
        public void update() {
            lblScore.setText("Score: " + gameModel.getScore());
            lblMoves.setText("Moves: " + gameModel.getNumberOfMoves() + "
                / " + Coordinator.MOVE_LIMIT);
        }

        //************************
        // Support Methods
        //************************
        /**
         * Adds all the components to the InfoPanel
         */
        private void addComponents() {
            BoxLayout layout = new BoxLayout(this, BoxLayout.Y_AXIS);
            setLayout(layout);
            lblScore = new JLabel();
            lblMoves = new JLabel();
```

```
            lblButtons = new JLabel();
            lblBoard = new JLabel();
            lblController = new JLabel();

            Font font = new Font(Font.MONOSPACED, Font.PLAIN, 18);
            addLabel(this, lblController, "<html> <h1> Info and Data
                </h1> <html>", font);
            addLabel(this, lblButtons, "<html> "
                    + "A - Left   D - Right <br> "
                    + "S - Down   W - Rotate <br> "
                    + "P - Pause  L - Lost <br>"
                    + "Escape - Exit  "
                    + "<br>   </html>", font);
            //addLabel(this, lblBoard, font, "lblBoard");
            addLabel(this, lblMoves, "Moves: 0", font);
            addLabel(this, lblScore, "Score: 0", font);
        }

        /**
         * Adds a label to the specified component where the font and
             text to be displayed
         * are specified.
         *
         * @param comp The component to add to
         * @param label The label to add
         * @param text The text to display on the panel
         * @param font The font in which to displat the text
         */
        private void addLabel(JComponent comp, JLabel label, String
             text, Font font) {
            label.setText(text);
            label.setFont(font);
            comp.add(label);
        }

    }
}
```

5.2.4 Generation

```
package ga;

import java.util.*;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * @author Pontus Soderlund
 */
public class Generation implements Cloneable, Comparable<Generation> {

    private int generationNumber;
    private int generationFitness;
    private int numberOfIndivuals;
    private int averageFitness;
    private double averageScorePerMove;
    ArrayList<Genome> genomes;

    //***********************
    // Constructors
    //***********************
    public Generation() {
        generationFitness = 0;
        generationNumber = 0;
        numberOfIndivuals = 0;
        genomes = new ArrayList<>();
    }

    public Generation(int generationNumber) {
        this();
        this.generationNumber = generationNumber;
    }

    public Generation(int generationNumber, ArrayList<Genome> genomes) {
        this();
        this.generationNumber = generationNumber;
        this.genomes = genomes;
    }

    private Generation(int gn, int gf, int noi, ArrayList<Genome> g) {
        this.generationNumber = gn;
        this.generationFitness = gf;
        this.numberOfIndivuals = noi;
        this.genomes = g;
    }

    //***********************
    // Getter Methods
    //***********************
    public int getGenerationNumber() {
        return generationNumber;
```

```
}

public int getGenerationFitness() {
    generationFitness = 0;
    for (int i = 0; i < genomes.size(); i++) {
        generationFitness += genomes.get(i).getFitness();
    }
    return generationFitness;
}

public int getAverageFitness() {
    averageFitness = getGenerationFitness() / genomes.size();
    return averageFitness;
}

public double getAverageScorePerMove() {
    double spm = 0;
    for (int i = 0; i < genomes.size(); i++) {
        spm += genomes.get(i).getFitnessPerMove();
    }
    averageScorePerMove = spm / ((double) genomes.size());
    return Math.round(averageScorePerMove * 1000.0) / 1000.0;
}

public Genome getFittest() {
    Collections.sort(genomes);
    return genomes.get(genomes.size() - 1);
}

public Genome getGenome(int index) {
    return genomes.get(index);
}

public Genome getRandomGenome() {
    return genomes.get((int) (Math.random() * size()));
}

public ArrayList<Genome> getGenomes() {
    return genomes;
}

//***********************
// Setter Methods
//***********************
public void setNumberOfIndivuals(int numberOfIndivuals) {
    this.numberOfIndivuals = numberOfIndivuals;
}

public void setGenerationFitness(int generationFitness) {
    this.generationFitness = generationFitness;
}

public void setGenerationNumber(int generationNumber) {
    this.generationNumber = generationNumber;
```

```
}

public void setGenomes(ArrayList<Genome> genomes) {
    this.genomes = genomes;
}

//***********************
// Support Methods
//***********************
public void removeGenome(int index) {
    genomes.remove(index);
}

public void sortAZ() {
    Collections.sort(genomes);
}

public int size() {
    return genomes.size();
}

public void addGenome(Genome g) {
    genomes.add(g);
    numberOfIndivuals++;
}

@Override
public Generation clone() {

    Generation g = new Generation();
    try {
        super.clone();
    } catch (CloneNotSupportedException ex) {
        Logger.getLogger(Generation.class.getName()).log(Level.SEVERE,
            null, ex);
    }
    ArrayList<Genome> genomesCopy = new ArrayList<>();
    for (int i = 0; i < genomes.size(); i++) {
        genomesCopy.add(genomes.get(i).clone());
    }

    g.setGenerationFitness(generationFitness);
    g.setGenerationNumber(generationNumber);
    g.setNumberOfIndivuals(numberOfIndivuals);
    g.setGenomes(genomesCopy);

    return g;
}

@Override
public String toString() {
    String s = "";
    s += "Gen: " + getGenerationNumber();
    s += ", Fitness: " + getGenerationFitness();
```

```
        s += ", NO Individuals: " + size();
        s += ", Avg. Fitness: " + getAverageFitness();
        return s;
    }

    @Override
    public int compareTo(Generation t) {
        return this.getGenerationNumber() - t.getGenerationNumber();
    }
}
```

5.2.5 GeneticAlgorithm

```
package ga;

import static Coordinator.Coordinator.GENERATION_LIMIT;
import static Coordinator.Coordinator.MOVE_LIMIT;
import static Coordinator.Coordinator.MUTATION_RATE;
import static Coordinator.Coordinator.MUTATION_STEP;
import static Coordinator.Coordinator.POPULATION_LIMIT;
import Tetris.Board;
import java.beans.*;
import java.util.*;
import Tetris.Model;
import Tetris.MyGameModel;

/**
 * @author Pontus Soderlund
 */
public class GeneticAlgorithm implements Model {

    //************************
    // Variables
    //************************
    private PropertyChangeSupport pcs;
    private int currentGenomeId, currentGenerationId, delay;
    private Generation currentGeneration;
    private ArrayList<Generation> previousGenerations;
    private MyGameModel modelState;

    //************************
    // Contstructor
    //************************
    public GeneticAlgorithm(MyGameModel state) {
        this.pcs = new PropertyChangeSupport(this);
        this.currentGeneration = new Generation(0);
        this.previousGenerations = new ArrayList<>();
        this.modelState = state;
    }

    //************************
    // Evolution Methods
    //************************
    public Genome initalize() {
        for (int i = 0; i < POPULATION_LIMIT; i++) {
            currentGeneration.addGenome(new Genome(i,
                currentGenerationId));
        }
        while (!stopCriteriaIsFulfilled()) {
            loopThroughGeneration();
            evolve();
        }
        return previousGenerations.get(currentGenerationId -
            1).getFittest();
```

```
}

private void loopThroughGeneration() {
    for (Genome g : currentGeneration.getGenomes()) {
        modelState.reset();

        while (getCurrentGenome().getMovesTaken() < MOVE_LIMIT) {
            int pieceId = getCurrentState().getNextIdAndRemove();
            if (getCurrentState().generatePiece(pieceId)) {
                makeNextMove();
                try {
                    Thread.sleep(delay);
                } catch (InterruptedException ex) {
                    System.out.println(ex);
                }
            } else {
                break;
            }
        }

        g.setFitness(modelState.getScore());
        currentGenomeId++;
        pcs.firePropertyChange("nextGenome", currentGenomeId - 1,
             currentGenomeId);
    }
}

/**
 * Creates a new generation by adding the best from the previous
     generation as well as
 * creating new genomes. When created there is a 75% percent chance
     that the genome is
 * a combination of random genomes from upper half of the previous
     generation, a 12.5%
 * chance that the genome is a new one and a 12.5% chance that the
     genome is a
 * mutation of the fittest genome from thre previous generateion.
 */
private void evolve() {
    Generation previousGen = currentGeneration.clone();
    Generation modGen = currentGeneration.clone();
    previousGenerations.add(previousGen);

    Genome fittest = currentGeneration.getFittest().clone();
    fittest.setId(fittest.getNewId(0, currentGenerationId + 1));
    fittest.setFitness(-1);
    fittest.setMovesTaken(0);

    currentGenerationId++;
    currentGeneration = new Generation(currentGenerationId);
    currentGeneration.addGenome(fittest);
    currentGenomeId = 0;

    //Remove the worst genomes
```

```
        for (int i = 0; i < POPULATION_LIMIT / 2; i++) {
            modGen.removeGenome(i);
        }

        //Creates new Genomes.
        int n = currentGeneration.size() - 1;
        while (currentGeneration.size() < POPULATION_LIMIT) {
            n++;
            if (Math.random() < 0.75) {
                Genome parent1 = modGen.getRandomGenome();
                Genome parent2 = modGen.getRandomGenome();
                currentGeneration.addGenome(makeChild(n, parent1,
                    parent2));
            } else {
                if (Math.random() < 0.5) {
                    currentGeneration.addGenome(new
                        Genome(currentGenerationId, n));
                } else {
                    Genome g = fittest.clone();
                    g.setId(g.getNewId(currentGenerationId, n));
                    currentGeneration.addGenome(mutateGenome(g));
                }
            }
        }

        pcs.firePropertyChange("nextGeneration", previousGen,
            currentGeneration);
    }

    private Genome mutateGenome(Genome genome) {
        Genome g = genome.clone();
        int n = 2;
        if (Math.random() < MUTATION_RATE) {
            g.setBumpiness(g.getBumpiness() + MUTATION_STEP * (n *
                Math.random() - 1));
        }
        if (Math.random() < MUTATION_RATE) {
            g.setCumulativeHeight(g.getCumulativeHeight() + MUTATION_STEP
                * (n * Math.random() - 1));
        }
        if (Math.random() < MUTATION_RATE) {
            g.setHoles(g.getHoles() + MUTATION_STEP * (n * Math.random()
                - 1));
        }
        if (Math.random() < MUTATION_RATE) {
            g.setRowsCleared(g.getRowsCleared() + MUTATION_STEP * (n *
                Math.random() - 1));
        }
        if (Math.random() < MUTATION_RATE) {
            g.setWeightedHeight(g.getWeightedHeight() + MUTATION_STEP *
                (n * Math.random() - 1));
        }
        return g;
    }
```

```
/**
 * Creates a child genome based on two genomes.
 *
 * @param numberInGeneration the number in the generation.
 * @param g1 One of the parents
 * @param g2 The other parent
 * @return The child that was created and mutated
 */
private Genome makeChild(int numberInGeneration, Genome g1, Genome
    g2) {
    return new Genome(numberInGeneration, currentGenerationId, g1,
        g2);
}

private boolean stopCriteriaIsFulfilled() {
    if (currentGenerationId < GENERATION_LIMIT) {
        return false;
    } else {
        return true;
    }
}

/**
 * If the number of moves already taken are greater than the move
     limit the current
 * generation evolves. If not the best move of all moves are
     selected based on the the
 * move score and then the best one is played.
 */
private void makeNextMove() {
    ArrayList<Move> possibleMoves = getPossibleMoves();
    Collections.sort(possibleMoves);
    Move nextMove = new Move();
    if (!possibleMoves.isEmpty()) {
        nextMove = possibleMoves.get(0);
    } else {
        return;
    }

    for (int i = 0; i < nextMove.getNumberOfMoves(); i++) {
        switch (nextMove.getMoves().charAt(i)) {
            case 'w':
                modelState.movement("rotate", false);
                break;
            case 'a':
                modelState.movement("left", false);
                break;
            case 's':
                modelState.movement("down", false);
                break;
            case 'd':
                modelState.movement("right", false);
                break;
```

```
        }
        getCurrentGenome().incrementMovesTaken();
    }
    modelState.deselectPiece();
    modelState.clearRows();
}

//***********************
// Support Methods
//***********************
public void loadState(MyGameModel saveState) {
    modelState.setBoard(saveState.getBoard().clone());
    modelState.setScore(saveState.getScore());
    modelState.setNumberOfMoves(saveState.getNumberOfMoves());
}

//***********************
// Setter Methods
//***********************
public void setCurrentState(MyGameModel modelState) {
    this.modelState = modelState;
}

public void setCurrentGenome(double rc, double wh,
        double th, double h, double b) {
    Genome g = new Genome("none", 0, rc, wh,
            th, h, b, -1, -1);
    currentGeneration.addGenome(g);
    System.out.println(currentGeneration.size());
    currentGenomeId = POPULATION_LIMIT;
}

public void setDelay(int delay) {
    this.delay = delay;
}

//***********************
// Getter Methods
//***********************
/**
 * Puts all the possible moves and their value (calculated by using
     the current
 * genomes values) in an ArrayList containing Moves.
 *
 * @return An ArrayList with all possible moves.
 */
public ArrayList<Move> getPossibleMoves() {
    MyGameModel saveState = getCurrentState().clone();

    ArrayList<Move> moves = new ArrayList<>();

    for (int rotations = 0; rotations < 4; rotations++) {
        for (int i = -5; i <= 5; i++) {
            Move m = new Move("", 0);
```

```
                String s = "";
                double moveScore = 0.0;

                loadState(saveState);

                for (int j = 0; j < rotations; j++) {
                    modelState.getBoard().move("rotate");
                    m.move('w');
                }
                if (0 > i) {
                    for (int j = 0; j < Math.abs(i); j++) {
                        if (modelState.getBoard().move("left")) {
                            m.move('a');
                        }
                    }
                } else if (0 < i) {
                    for (int j = 0; j < Math.abs(i); j++) {
                        if (modelState.getBoard().move("right")) {
                            m.move('d');
                        }
                    }
                }

                while (modelState.getBoard().move("down")) {
                    m.move('s');
                }
                try {
                    Thread.sleep(delay);
                } catch (InterruptedException ex) {
                }

                modelState.getBoard().deselectPiece();
                moveScore += getCurrentGenome().getBumpiness() *
                     modelState.getBoard().getBumpiness();
                moveScore += getCurrentGenome().getCumulativeHeight() *
                     modelState.getBoard().getTotalHeight();
                moveScore += getCurrentGenome().getHoles() *
                     modelState.getBoard().getNumberOfHoles();
                moveScore += getCurrentGenome().getRowsCleared() *
                     modelState.getBoard().getNumberOfLinesCleared();
                moveScore += getCurrentGenome().getWeightedHeight() *
                     modelState.getBoard().getHeightOfHighestCollumn();

                m.setMoveScore(moveScore);
                if (m.getNumberOfMoves() +
                     getCurrentGenome().getMovesTaken() < MOVE_LIMIT) {
                    moves.add(m);
                }
            }
        }

        loadState(saveState);
        return moves;
    }
```

```
public MyGameModel getCurrentState() {
    return this.modelState;
}

public Genome getCurrentGenome() {
    return currentGeneration.getGenome(currentGenomeId);
}

public int getCurrentGenerationId() {
    return currentGenerationId;
}

public int getCurrentGenomeId() {
    return currentGenomeId;
}

public Generation getPreviousGeneration() {
    Collections.sort(previousGenerations);
    return previousGenerations.get(previousGenerations.size() - 1);
}

public Generation getCurrentGeneration() {
    return currentGeneration;
}

public ArrayList<Generation> getPreviousGenerations() {
    return previousGenerations;
}

public int getDelay() {
    return delay;
}

//***********************
// Property Support Methods
//***********************
@Override
public void addPropertyChangeListener(PropertyChangeListener l) {
    pcs.addPropertyChangeListener(l);
}

@Override
public void addPropertyChangeListener(String propertyName,
     PropertyChangeListener l) {
    pcs.addPropertyChangeListener(propertyName, l);
}

@Override
public void removePropertyChangeListener(PropertyChangeListener l) {
    pcs.addPropertyChangeListener(l);
}

@Override
```

```
    public void removePropertyChangeListener(String propertyName,
        PropertyChangeListener l) {
        pcs.addPropertyChangeListener(propertyName, l);
    }

}
```

5.2.6 GeneticAlgorithmView

```
package ga;

import java.awt.*;
import java.beans.*;
import java.util.*;
import javax.swing.*;
import javax.swing.border.LineBorder;
import Coordinator.Coordinator;
import static Coordinator.Coordinator.GENERATION_LIMIT;
import static Coordinator.Coordinator.MUTATION_STEP;

/**
 * @author Pontus Soderlund
 */
public class GeneticAlgorithmView extends JPanel implements
    PropertyChangeListener {

   protected GeneticAlgorithm gaModel;

   private JPanel topPanel;
   private BotPanel botPanel;

   private JLabel lblGeneration;
   private JLabel lblGenome;
   private JLabel lblFittest;
   private JLabel lblAverage;
   private JLabel lblMutationRate;
   private JLabel lblMutationStep;
   private JProgressBar prgGeneration;
   private JProgressBar prgGenome;

   public GeneticAlgorithmView(GeneticAlgorithm gaModel, Dimension dim)
       {
       setModel(gaModel);
       BoxLayout layout = new BoxLayout(this, BoxLayout.Y_AXIS);
       setLayout(layout);

       setPreferredSize(dim);
       Dimension rightDim = new Dimension((int) dim.getWidth(), (int)
           (dim.getHeight() * (1.0 / 3.0)));
       topPanel = new JPanel();
       botPanel = new BotPanel(rightDim);

       setupLeftPanel();
       setupRightPanel();
       this.setBackground(Coordinator.BACKROUND_COLOR);
       topPanel.setBackground(Coordinator.BACKROUND_COLOR);
       botPanel.setBackground(Coordinator.BACKROUND_COLOR);

       LineBorder border = new LineBorder(Color.GRAY);
       topPanel.setBorder(border);
```

```
    botPanel.setBorder(border);
    add(topPanel);
    add(botPanel);
}

private void setupRightPanel() {

}

private void setupLeftPanel() {
    BoxLayout layout = new BoxLayout(topPanel, BoxLayout.Y_AXIS);
    topPanel.setLayout(layout);
    lblMutationRate = new JLabel("Mutation rate: " +
        Coordinator.MUTATION_RATE);
    lblMutationStep = new JLabel("Mutation step: " + MUTATION_STEP);
    lblGeneration = new JLabel("Generation: 0 / " +
        GENERATION_LIMIT);
    lblGenome = new JLabel("Genome: 1 / " +
        Coordinator.POPULATION_LIMIT);
    lblAverage = new JLabel("<html> Avg Fit: <br>"
           + "Avg Spm:");
    lblFittest = new JLabel("<html> <b>Fittest</b> <br>"
           + "Fittness: " + "<br>"
           + "Moves: " + "<br>"
           + "Spm: ");
    prgGeneration = new JProgressBar(0, GENERATION_LIMIT);
    prgGenome = new JProgressBar(0, Coordinator.POPULATION_LIMIT);

    Font font = new Font(Font.MONOSPACED, Font.PLAIN, 18);
    lblMutationRate.setFont(font);
    lblMutationStep.setFont(font);
    lblGeneration.setFont(font);
    lblGenome.setFont(font);
    lblAverage.setFont(font);
    lblFittest.setFont(font);
    prgGeneration.setFont(font);
    prgGenome.setFont(font);

    topPanel.add(lblMutationRate);
    topPanel.add(lblMutationStep);
    topPanel.add(lblGeneration);
    topPanel.add(prgGeneration);
    topPanel.add(lblGenome);
    topPanel.add(prgGenome);
    topPanel.add(lblAverage);
    topPanel.add(lblFittest);
}

private void updateLeftPanel(PropertyChangeEvent pce) {
    if (pce.getPropertyName().equals("nextGeneration")) {
        Generation preGen = gaModel.getPreviousGeneration();
        Genome fittest = gaModel.getPreviousGeneration().getFittest();
        //System.out.println(fittest.toString());
```

```
            prgGeneration.setValue(gaModel.getCurrentGenerationId());
            lblGeneration.setText("Generation: " +
                gaModel.getCurrentGenerationId() + " / " +
                GENERATION_LIMIT);
            lblAverage.setText("<html> Avg Fit: " +
                preGen.getAverageFitness()
                   + "<br> Avg Spm: " + preGen.getAverageScorePerMove());
            lblFittest.setText("<html> <b>Fittest </b> <br>"
                   + "Fitness: " + fittest.getFitness() + "<br>"
                   + "Moves: " + fittest.getMovesTaken() + "<br>"
                   + "Spm: " + fittest.getFitnessPerMove());

        } else if (pce.getPropertyName().equals("nextGenome")) {
            prgGenome.setValue(gaModel.getCurrentGenomeId());
            lblGenome.setText("Genome: " + gaModel.getCurrentGenomeId() +
                " / " + Coordinator.POPULATION_LIMIT);
        }
    }

    private void setModel(GeneticAlgorithm gaModel) {
        this.gaModel = gaModel;
        if (this.gaModel != null) {
            this.gaModel.addPropertyChangeListener(this);
        }
    }

    @Override
    public void propertyChange(PropertyChangeEvent pce) {
        updateLeftPanel(pce);
        if (pce.getPropertyName().equals("nextGeneration")) {
            botPanel.updateRightPanel(pce);
        }
    }

    private class BotPanel extends JPanel {

        private Dimension dim;
        private int height;
        private int width;

        //Size equals to the resolution, number of fitness values within
            the "step" span
        private int[] distrubution;
        //All the genomes fitness
        private int[] fitness;
        //Total number of bars
        private int resolution = Coordinator.DISTRUBUTION_RESOLUTION;
        private int maxFitness;
        private int minFitness;
        //How large each bar should be
        private int fitnessStep;
        private int xStep;
        private int yStep;
```

```
public BotPanel(Dimension dim) {
    this.setPreferredSize(dim);
    this.setMinimumSize(dim);
    height = (int) this.getPreferredSize().getHeight();
    width = (int) this.getPreferredSize().getWidth();
}

public void updateRightPanel(PropertyChangeEvent pce) {
    ArrayList<Genome> gen =
        gaModel.getPreviousGenerations().get(gaModel.getPreviousGenerations().size()
        - 1).getGenomes();
    Collections.sort(gen);
    fitness = new int[Coordinator.POPULATION_LIMIT];
    distrubution = new int[resolution];
    maxFitness = gen.get(gen.size() - 1).getFitness();
    minFitness = gen.get(0).getFitness();
    fitnessStep = (maxFitness - minFitness) / resolution;
    fitnessStep += 10;

    for (int i = 0; i < fitness.length; i++) {
        fitness[i] = gen.get(i).getFitness();
    }

    for (int i = 0; i < distrubution.length; i++) {
        for (int j = 0; j < fitness.length; j++) {
            if (fitness[j] < ((fitnessStep) * (i + 1))
                    && fitness[j] >= (fitnessStep * i)
                    && fitness[j] != -1) {

                distrubution[i]++;
                fitness[j] = -1;
            }
        }
    }

    getXStep();
    getYStep();

    botPanel.repaint();
}

public void getXStep() {
    width = this.getWidth();
    xStep = width / (resolution + 2);
}

public void getYStep() {
    height = this.getHeight();
    int m = 0;
    for (int i = 0; i < distrubution.length; i++) {
        if (distrubution[i] > m) {
            m = distrubution[i];
        }
    }
```

```
            yStep = height / (m + 2);
        }

        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            if (xStep != 0 && yStep != 0) {
                int n = (width / xStep);
                g.setColor(Color.black);
                for (int i = 0; i < resolution + 2; i++) {
                    if (i != 0 && i != resolution + 1) {
                        g.fillRect((i * xStep),
                                (height - distrubution[i - 1] * yStep),
                                    xStep, height - yStep);
                    }
                }
            }
        }
    }
}
```

5.2.7 Genome

```
package ga;

import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * @author Pontus Soderlund
 */
public final class Genome implements Comparable<Genome>, Cloneable {

    //***********************
    // Variables
    //***********************
    private String id;
    private int movesTaken, generation, fitness;
    private double rowsCleared;      //The number of rows cleared
    private double weightedHeight;   //The height of the highest collumn
    private double cumulativeHeight; //The sum of all heights
    private double holes;            //The number of holes
    private double bumpiness;        //The bumpiness of the board
    private double fitnessPerMove;

    //***********************
    // Constructor(s)
    //***********************
    public Genome(int numberInGeneration, int generation) {
        this.id = getNewId(generation, numberInGeneration);
        this.generation = generation;
        this.rowsCleared = Math.random();
        this.weightedHeight = Math.random();
        this.cumulativeHeight = Math.random();
        this.holes = Math.random();
        this.bumpiness = Math.random();
        this.fitness = 0;
    }

    public Genome(int numberInGeneration, int generation, Genome g1,
         Genome g2) {
        this.id = getNewId(generation, numberInGeneration);
        this.generation = generation;
        if (Math.random() < 0.5) {
            this.rowsCleared = g1.getRowsCleared();
        } else {
            this.rowsCleared = g2.getRowsCleared();
        }
        if (Math.random() < 0.5) {
            this.weightedHeight = g1.getWeightedHeight();
        } else {
            this.weightedHeight = g2.getWeightedHeight();
        }
        if (Math.random() < 0.5) {
```

```
            this.cumulativeHeight = g1.getCumulativeHeight();
        } else {
            this.cumulativeHeight = g2.getCumulativeHeight();
        }
        if (Math.random() < 0.5) {
            this.holes = g1.getHoles();
        } else {
            this.holes = g2.getHoles();
        }
        if (Math.random() < 0.5) {
            this.bumpiness = g1.getBumpiness();
        } else {
            this.bumpiness = g2.getBumpiness();
        }
        this.fitness = 0;
    }

    /**
     *
     * @param id the id of the genome
     * @param gen the generation
     * @param rc weight for number of rowsCleared
     * @param wh weight for the weightedHeight
     * @param ch weight for cumulativeHeight
     * @param h weight for holes
     * @param b weight for holes
     * @param f the fitness
     * @param mt number of moves taken
     */
    public Genome(String id, int gen, double rc, double wh,
            double ch, double h, double b, int f, int mt) {
        this.id = id;
        this.generation = gen;
        this.rowsCleared = rc;
        this.weightedHeight = wh;
        this.cumulativeHeight = ch;
        this.holes = h;
        this.bumpiness = b;
        this.fitness = f;
        this.movesTaken = mt;
    }

    //***********************
    // Support Methods
    //***********************
    public void incrementMovesTaken() {
        this.movesTaken++;
    }

    @Override
    public int compareTo(Genome t) {
        return (int) (this.getFitness() - t.getFitness() + 0.5);
    }
```

```
@Override
public Genome clone() {
    try {
        super.clone();
    } catch (CloneNotSupportedException ex) {
        Logger.getLogger(Genome.class.getName()).log(Level.SEVERE,
            null, ex);
    }
    Genome g = new Genome(id, generation, rowsCleared,
        weightedHeight,
            cumulativeHeight, holes, bumpiness, fitness, movesTaken);
    return g;
}

@Override
public String toString() {
    String s = "";
    s += "ID: " + id;
    s += ", Gen: " + generation;
    s += ", Moves Taken: " + movesTaken;
    s += ", Fitness: " + fitness + " ";
    s += "\n" + rowsCleared;
    s += ", " + weightedHeight;
    s += ", " + cumulativeHeight;
    s += ", " + holes;
    s += ", " + bumpiness;
    return s;
}

//***********************
// Getter Methods
//***********************
public String getNewId(int generation, int numberInGeneration) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < 7 - String.valueOf(generation).length();
        i++) {
        sb.append(0);
    }
    sb.append(generation);
    for (int i = 0; i < 3 -
        String.valueOf(numberInGeneration).length(); i++) {
        sb.append(0);
    }
    sb.append(numberInGeneration);
    return sb.toString();
}

public int getFitness() {
    return fitness;
}

public int getMovesTaken() {
    return movesTaken;
}
```

```
public String getId() {
    return id;
}

public double getFitnessPerMove() {
    return Math.round((double) fitness / (double) movesTaken *
        1000.0) / 1000.0;
}

public double getRowsCleared() {
    return rowsCleared;
}

public double getWeightedHeight() {
    return weightedHeight;
}

public double getCumulativeHeight() {
    return cumulativeHeight;
}

public double getHoles() {
    return holes;
}

public double getBumpiness() {
    return bumpiness;
}

//***********************
// Setter Methods
//***********************
public void setMovesTaken(int movesTaken) {
    this.movesTaken = movesTaken;
}

public void setRowsCleared(double rowsCleared) {
    this.rowsCleared = rowsCleared;
}

public void setWeightedHeight(double weightedHeight) {
    this.weightedHeight = weightedHeight;
}

public void setCumulativeHeight(double cumulativeHeight) {
    this.cumulativeHeight = cumulativeHeight;
}

public void setHoles(double holes) {
    this.holes = holes;
}

public void setBumpiness(double bumpiness) {
```

```
        this.bumpiness = bumpiness;
    }

    public void setFitness(int fitness) {
        this.fitness = fitness;
    }

    public void setId(String id) {
        this.id = id;
    }
}
```

5.2.8 Model

```
package Tetris;

import java.beans.PropertyChangeListener;
import java.io.Serializable;

/**
 * @author Pontus Soderlund
 */
public interface Model extends Serializable {

    void addPropertyChangeListener(PropertyChangeListener l);

    void addPropertyChangeListener(String propertyName,
        PropertyChangeListener l);

    void removePropertyChangeListener(PropertyChangeListener l);

    void removePropertyChangeListener(String propertyName,
        PropertyChangeListener l);
}
```

5.2.9 Move

```
package ga;

/**
 * @author Pontus Soderlund
 */
public class Move implements Comparable<Move> {

    //************************
    // Variables
    //************************
    private Double moveScore;
    private String moves;

    //************************
    // Constructor(s)
    //************************
    public Move(){

    }

    public Move(String moves, double moveScore) {
        this.moves = moves;
        this.moveScore = moveScore;
    }

    //************************
    // Support Methods
    //************************
    public void move(char move) {
        moves += String.valueOf(move);
    }

    public void calculateScore() {

    }

    @Override
    public String toString() {
        String moveSequence = "";
        for (int i = 0; i < moves.length(); i++) {
            moveSequence += moves.charAt(i);
            if (i < moves.length() - 1) {
                moveSequence += ", ";
            }
        }
        return "Score: " + moveScore + " \t Moves: " + moveSequence;
    }

    @Override
    public int compareTo(Move t) {
        double msThis = this.getMoveScore();
```

```
        double msOther = t.getMoveScore();
        if (msThis > msOther) {
            return 1;
        } else if (msThis < msOther) {
            return -1;
        } else {
            return 0;
        }
    }

    //***********************
    // Getter Methods
    //***********************
    public int getNumberOfMoves() {
        return moves.length();
    }

    public Double getMoveScore() {
        return moveScore;
    }

    public String getMoves() {
        return moves;
    }

    //***********************
    // Setter Methods
    //***********************
    public void setMoveScore(Double moveScore) {
        this.moveScore = moveScore;
    }

    public void setMoves(String moves) {
        this.moves = moves;
    }
}
```

5.2.10 MyGameMode

```
package Tetris;

import Coordinator.Coordinator;
import java.beans.*;
import java.io.*;
import java.time.LocalDateTime;
import java.util.Scanner;
import java.util.LinkedList;
import javax.swing.*;

/**
 * @author Pontus Soderlund
 */
public class MyGameModel implements Model, Cloneable {

    //***********************
    // Variables
    //***********************
    private LinkedList<Integer> numberList;
    private final PropertyChangeSupport pcs;
    private Board board;

    private int score;
    private int numberOfMoves;
    private int nextPieceId;

    private int numberOfPieces;
    private boolean randomNumbers;
    private static boolean generateRandomNumber;

    //***********************
    // Constructor(s)
    //***********************
    /**
     * Creates a new MyGameModel with Board if true else without Board.
     *
     * @param randomNumbers
     * @param numberOfPieces
     * @param b with or without board
     */
    public MyGameModel(boolean randomNumbers, int numberOfPieces) {
        this.pcs = new PropertyChangeSupport(this);
        this.numberList = new LinkedList<>();
        this.numberOfPieces = numberOfPieces;
        this.randomNumbers = randomNumbers;
        this.board = new Board();
        generateNumbers();
    }

    /**
     * Creates a board where all the parameters have to be defined.
```

```
 *
 * @param pcs
 * @param board
 * @param score
 * @param numberOfMoves
 */
public MyGameModel(PropertyChangeSupport pcs, Board board, int
    score, int numberOfMoves) {
    this.pcs = pcs;
    this.board = board;
    this.score = score;
    this.numberOfMoves = numberOfMoves;
    generateNumbers();
}

//***********************
// Board Methods
//***********************
/**
 * Moves or rotates the current piece if possible and fires a a
    PropertyChangeEvent if
 * the "thinking" variable is false.
 *
 * @param direction Which direction to move/rotate
 * @param thinking Whether or not the Genetic Algorithm is making
    all possible moves
 */
public void movement(String direction, boolean thinking) {
    int[][] old = board.getBoard().clone();

    if (board.move(direction)) {
        numberOfMoves++;
        if (direction.equals("down")) {
            score++;
        }
    }
    if (!thinking) {
        pcs.firePropertyChange("movement", old, board.getBoard());
    } else {
        pcs.firePropertyChange("thinkingMovement", old,
            board.getBoard());
    }
}

/**
 * Moves all currently selected tiles down one step and increases
    the score by one.
 * Fires a PropertyChangeEvent.
 *
 * @return false if it cant fall, else true.
 */
public boolean fall() {
    if (board.canMove("down")) {
        int[][] old = board.getBoard().clone();
```

```
            board.move("down");
            score++;
            numberOfMoves++;
            pcs.firePropertyChange("fall", old, board.getBoard());
            return true;
        } else {
            return false;
        }
    }

    /**
     * Clears all the rows that are clearable and increases the score by
         the appropiate
     * amount (depending of number of rows cleared). Fires a
         ProprtyChangeEvent if
     * drawable is true.
     */
    public void clearRows() {
        MyGameModel old = this.clone();
        switch (board.clearRows()) {
            case 1:
                score += 100;
                break;
            case 2:
                score += 300;
            case 3:
                score += 600;
                break;
            case 4:
                score += 1200;
                break;
        }
        pcs.firePropertyChange("clearRows", old, this);
    }

    public int getPieceId() {
        return board.getPieceId();
    }

    public boolean generatePiece(int pieceId) {
        return board.generatePiece(pieceId);
    }

    public void deselectPiece(){
        board.deselectPiece();
    }

    //***********************
    // Support Methods
    //***********************
    /**
     * Fills the numberList with numbers from the
         pseudoRandomNumbers.txt file or random
     * numbers.
```

```
 */
private void generateNumbers() {
    numberList = new LinkedList<>();
    if (!randomNumbers) {
        try {
            Scanner numberReader = new Scanner(new
                File("pseudoRandomNumbers.txt"));
            for (int i = 0; i < numberOfPieces; i++) {
                numberList.add(numberReader.nextInt());
            }
        } catch (FileNotFoundException e) {
            System.out.println("404: File Not Found");
        }
    } else {
        for (int i = 0; i < numberOfPieces; i++) {
            numberList.add((int) (1 + Math.random() * 7));
        }
    }
}

/**
 * The player loses and the relevant data (name, score, speed, date
     and number of
 * moves) is appended to the Scores.txt file.
 */
public void lost() {
    JPanel frame = new JPanel();
    JLabel lblLost = new JLabel("You Lost! Your score is: " + score);
    frame.add(lblLost);

    String name = JOptionPane.showInputDialog(frame);
    String date = LocalDateTime.now().toString();

    try {
        try (Writer w = new BufferedWriter(new OutputStreamWriter(
                new FileOutputStream(new File("Scores.txt"), true),
                    "UTF-8"))) {
            w.append(System.lineSeparator() + date + " \t Score: "
                    + score + "\t Speed: " + Coordinator.FALL_FREQUENZY
                    + "\t Name: " + name + "\t Moves: " +
                        numberOfMoves);
        }
    } catch (IOException e) {
        System.out.println(e);
    }
    System.exit(score);
}

/**
 * Resets the current MyGameModel. Fires a PropertyChangeEvent if
     drawable is true.
 */
public void reset() {
    MyGameModel old = this.clone();
```

```
    board = new Board();
    score = 0;
    numberOfMoves = 0;
    generateNumbers();

    pcs.firePropertyChange("reset", old, this);
}

/**
 * Clones the current MyGameModel and returns an immutable version.
 *
 * @return and immutable copy of this MyGameModel.
 */
@Override
public MyGameModel clone() {
    Board boa = board.clone();
    Integer sco = ((Integer) score);
    Integer nom = ((Integer) numberOfMoves);

    MyGameModel m;
    m = new MyGameModel(pcs, boa, sco, nom);
    return m;
}

//***********************
// Getter Methods
//***********************
public Board getBoard() {
    return board;
}

public int getScore() {
    return score;
}

public int getNumberOfMoves() {
    return numberOfMoves;
}

/**
 * Returns the element at index 0 of the numberList.
 *
 * @return the index of the piece id.
 */
public int getNextPieceId() {
    return numberList.peek();
}

/**
 * Returns the element at index 0 of the numberList and removes it.
 *
 * @return the index of the piece id.
 */
public int getNextIdAndRemove() {
```

```
        return numberList.pop();
    }

    //***********************
    // Setter Methods
    //***********************
    public void setBoard(Board board) {
        this.board = board;
    }

    public void setScore(int score) {
        this.score = score;
    }

    public void setNumberOfMoves(int numberOfMoves) {
        this.numberOfMoves = numberOfMoves;
    }

    public void setGenerateRandomNumber(boolean generateRandomNumber) {
        this.generateRandomNumber = generateRandomNumber;
    }

    //***********************
    // Property Support Methods
    //***********************
    @Override
    public void addPropertyChangeListener(PropertyChangeListener l) {
        pcs.addPropertyChangeListener(l);
    }

    @Override
    public void addPropertyChangeListener(String propertyName,
         PropertyChangeListener l) {
        pcs.addPropertyChangeListener(propertyName, l);
    }

    @Override
    public void removePropertyChangeListener(PropertyChangeListener l) {
        pcs.addPropertyChangeListener(l);
    }

    @Override
    public void removePropertyChangeListener(String propertyName,
         PropertyChangeListener l) {
        pcs.addPropertyChangeListener(propertyName, l);
    }

}
```