Aufgabe 4: Würfelglück

Teilnahme-ID: 62344

Manuel Frohn

22.11.2021

Inhalt

Lösungsansatz

In diesem Abschnitt beschäftigen wir uns mit meiner Lösungsidee. Diese ist recht simple: Für die einzigartigen Würfel Kombinationen eine signifikante Menge (ich habe mich für 1000 pro Kombination entschieden, da es zum Bestimmen einer Statistischen Signifikanz ausreicht, und nicht allzu lange zum Rechnen braucht) an Spielen zu simulieren und sie danach anhand der Anzahl der Siege, über andere Würfel zu ordnen. Hierzu bekommt ein Würfel für einen Sieg über einen anderen einen Punkt. Ob ein Würfel über einen anderen gesiegt hat, wird dadurch bestimmt, indem verglichen wird welcher der beiden Würfel die signifikante Menge der Spiele zwischen den beiden gewonnen hat. Signifikant ist eine Menge dann, wenn sie größer als 2/3 der Versuche ist (dieser Wert hat sich aus Experimenten mit gleichen Würfeln ergeben. Bei diesen kommt es unter anderem zu Ergebnissen wie 600 zu 400). Hat keiner der beiden Würfel eine signifikante Menge der Spiele gewonnen so kommt es zu einem unentschieden und keiner der Würfel bekommt einen Punkt.

Umsetzung

In diesem Abschnitt dieser Dokumentation werde ich ihnen erläutern, wie ich meinen Lösungsansatz in ein Programm übersetzt habe. Hierzu müssen wir uns zunächst die Simulation der einzelnen Spiele angucken. Diese geschieht in der Game-Klasse. In dieser wird nach dem aufrufen der *play(int startPlayer)*-Methode ein komplettes Spiel simuliert und der Gewinner zurückgegeben. Danach ist ein Objekt dieser Klasse unbrauchbar. Innerhalb dieser play-Methode wird folgendes so lange wiederholt, bis ein Spieler gewonnen hat: Zuerst wird gewürfelt. Dies geschieht durch den Aufruf der roll()-Methode des, dem momentanen Spieler zugeordneten Dice-Objekts. In dieser wird eine Zufallszahl im Raum von null bis zur Anzahl der Seiten, diese ausgeschlossen ([0; sides[) generiert. Diese dient dann als Index um eine Zahl aus dem sideEyeCount-Array auszuwählen (In dem Array steht jeder Eintrag für die Augenzahl einer Seite des Würfels). Danach wird über die *getFigureToMove(int currentPlayer, int dieResult)*-Methode die zu bewegende Figur bestimmt. In dieser wird zuerst geprüft ob noch eine Figur auf den B-Feldern steht und ob eine Figur auf dem A-Feld steht. Ist diese Bedingung erfüllt wird geprüft, ob die Figur auf dem A-Feld, dem

Würfelergebnis nach bewegt werden kann. Ist auch diese Bedingung erfüllt wird die Figur auf dem A-Feld zurückgegeben. Ist aber eine der beiden Bedingungen nicht erfüllt wird nun geprüft, ob eine Figur auf den B-Feldern steht. Ist dem so wird geprüft, ob diese bewegt werden kann, also ob das Würfelergebnis sechs ist und keine Figur auf dem A-Feld steht. Ist ein dieser beiden Bedingungen nicht erfüllt wird für die Figuren, die auf dem Feld oder schon im Haus stehen, wenn es solche gibt, geprüft, ob diese bewegt werden können. Dabei wird bei der Vordersten gestartet und dann von der Position abhängenden absteigend weitergemacht. Wurde eine Figur gefunden, wird die moveFigure(Figure figure, int currentPlayer, int dieResult)-Methode für die gefundene Figur und das Würfelergebnis aufgerufen. In dieser werden beide Positionsdaten der Figur entsprechenden angepasst und wenn nötig eine Figur geschlagen. Ich verwende zwei Positionswerte, da der Absolutwert, also auf welchem Feld der Laufbahn sich die Figur tatsächlich befindet, zwar notwendig ist, um festzustellen, ob eine Figur des Gegners geschlagen werden muss, aber für alles andere ist der Relativwert, also wo sich die Figur vom A-Feld (0) bis zum Ende des Hauses (43) befindet. Zum Schluss wird noch der momentane Spieler auf den jeweils anderen geändert, insofern das Würfelergebnis nicht sechs ist.

Nun da sie wissen wie die Simulation eines Spiels funktioniert, gucken wir uns noch an wie diese verwendet werden, um zu bestimmen welcher der Würfel der beste ist. Hierzu wird zunächst jede Würfelkombination gegeneinander antreten gelassen. Hierzu werden in einer geschachtelten for-Schleife zwei Würfel ausgewählt und gegeneinander antreten gelassen, es werden also 1000 Spiele mit den beiden Würfeln durchgeführt. Danach wird geprüft, ob einer der beiden Würfel mindestens 2/3 der Spiele gewonnen hat. Ist dies für einen der beiden Würfel so, dann wird diesem ein Punkt angerechnet, sonst bekommen beide einen halben Punkt. Nachdem diese Prozedur für alle Würfelkombinationen durchlaufen wurde, werden die Würfel ihrer Punktzahl nach geordnet (höchste als erstes) und zurückgegeben. Um sie dann dazustellen wird die erzeugte Liste noch an die *getSolutionAsText(ArrayList<Dice> dicesInOrder)*-Methode gegeben, wo den Würfeln der Reinfolge nach ein Platz zugeordnet wird. Sollten Würfel gleiche Punkte haben, wird ihnen derselbe Platz zugeordnet.

Lösungen

1. mit 5.0 Siegen: 20 Seiten [1;2;3;4;5;6;7;8;9;10;11;12;13;14;15;16;17;18;19;20;]
2. mit 4.0 Siegen: 12 Seiten [1;2;3;4;5;6;7;8;9;10;11;12;]
3. mit 3.0 Siegen: 10 Seiten [0;1;2;3;4;5;6;7;8;9;]
4. mit 2.0 Siegen: 4 Seiten [1;2;3;4;]
5. mit 1.0 Siegen: 6 Seiten [1;1;1;6;6;6;]
6. mit 0.0 Siegen: 6 Seiten [1;2;3;4;5;6;]

Abbildung 1: Lösung Beispiel 0

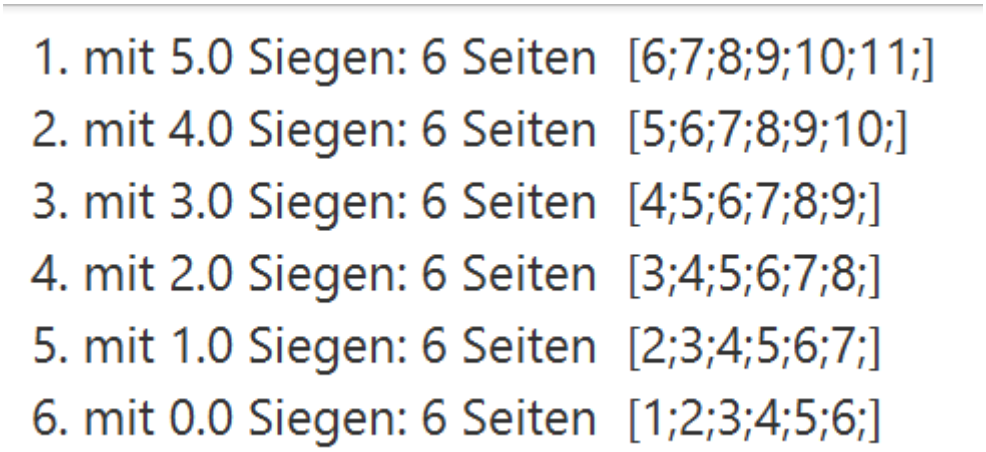

Abbildung 2: Lösung Beispiel 1

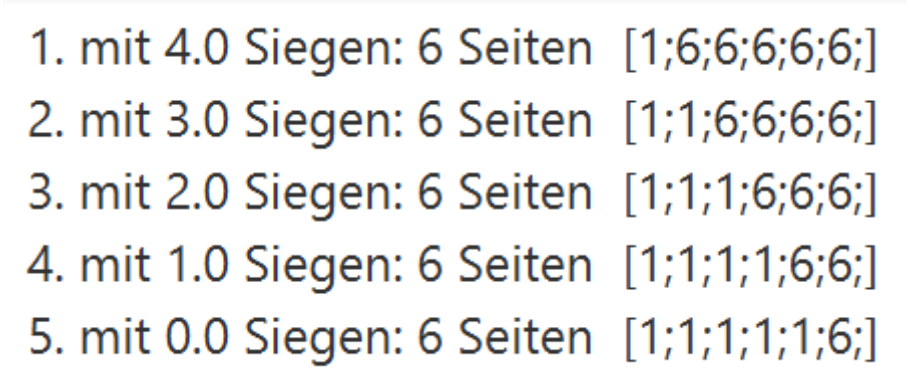

Abbildung 3: Lösung Beispiel 2

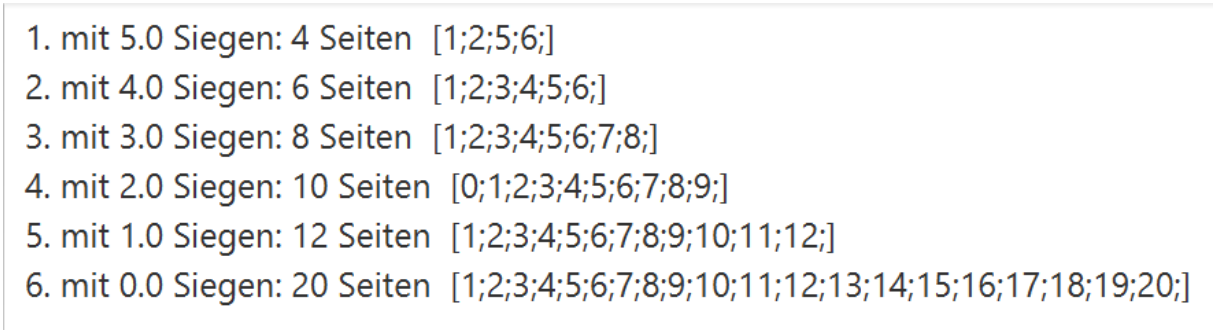

Abbildung 4: Lösung Beispiel 3

Code-Ausschnitte

```
for(int diceA = 0; diceA < diceCount; diceA++)
{
    for(int diceB = diceA + 1; diceB < diceCount; diceB++)
    {
        int diceAWins = 0;
        int diceBWins = 0;

        for(int i = 0; i < SIMULATIONS_PER_PAIR; i++)
        {
            int starter;

            if(i < (SIMULATIONS_PER_PAIR/2))
            {
                starter = 0;
            }
            else
            {
                starter = 1;
            }

            Dice[] gameDices = new Dice[2];
            gameDices[0] = dices[diceA];
            gameDices[1] = dices[diceB];
            Game game = new Game(dices);

            int winner = game.play(starter);

            if(winner == 0)
            {
                diceAWins++;
            }
            else
            {
                diceBWins++;
            }
        }

        if(diceAWins >= SIMULATIONS_PER_PAIR * 0.66 || diceBWins >= SIMULATIONS_PER_PAIR * 0.66 )
        {

            if(diceAWins > diceBWins)
            {
                dices[diceA].doWinsPlusOne();
                System.out.println("Dice " + diceA + " won against " + diceB + " with " + diceAWins + " wins.");
            }
            else
            {
                dices[diceB].doWinsPlusOne();
                System.out.println("Dice " + diceB + " won against " + diceA + " with " + diceBWins + " wins.");
            }
        }
        else
        {
            System.out.println("Draw between " + diceA + " and " + diceB);
            dices[diceA].doWinsPlusOneHalf();
            dices[diceB].doWinsPlusOneHalf();
        }
    }
}
```

Abbildung 5: Simulierung der Spiele für die unterschiedlichen Würfelkombinationen

```java
ArrayList<Dice> dicesInOrder = new ArrayList<>();

for(int i = 0; i < dices.length; i++)
{
    Dice highestDice = null;
    for(Dice dice : dices)
    {
        if(!dicesInOrder.contains(dice) && highestDice == null)
        {
            highestDice = dice;
        }
        else if(!dicesInOrder.contains(dice) && highestDice.getWins() <= dice.getWins())
        {
            highestDice = dice;
        }
    }
    dicesInOrder.add(highestDice);
}

return dicesInOrder;
```

Abbildung 6: Ordnung der Würfel nach deren Punktzahl

```java
public int play(int startPlayer)
{
    int currentPlayer = startPlayer;

    while (!hasGameEnded(currentPlayer))
    {
        int dieResult = dices[currentPlayer].roll();

        Figure figure = getFigureToMove(currentPlayer, dieResult);

        if(figure != null)
        {
            moveFigure(figure,currentPlayer ,dieResult);
        }

        if(dieResult != 6)
        {
            currentPlayer = Math.abs(currentPlayer - 1);
        }
    }

    return currentPlayer;
}
```

Abbildung 6: play-Methode

```java
public class Dice
{
    private double wins = 0;
    private int sides;
    private int[] sideEyeCount;

    public Dice (String constructor)
    {
        String[] constructorParts = constructor.split(" ");

        sides = Integer.parseInt(constructorParts[0]);
        sideEyeCount = new int[sides];

        for(int n = 1; n < constructorParts.length; n++)
        {
            sideEyeCount[n - 1] = Integer.parseInt(constructorParts[n]);
        }
    }

    public int roll()
    {
        int side =(int) (Math.random() * sides);
        return sideEyeCount[side];
    }

    public double getWins()
    {
        return wins;
    }

    public void doWinsPlusOne()
    {
        wins++;
    }

    public void doWinsPlusOneHalf()
    {
        wins += 0.5;
    }

    public int getSides()
    {
        return sides;
    }

    public int[] getSideEyeCount()
    {
        return sideEyeCount;
    }
}
```

Abbildung 7: Dice-Klasse

```java
public Figure getFigureToMove(int currentPlayer, int dieResult)
{

    if(getFigureOnBField(currentPlayer) != null && getFigureOnAField(currentPlayer) != null)
    {
        if(canFigureBeMoved(getFigureOnAField(currentPlayer), currentPlayer, dieResult))
        {
            return getFigureOnAField(currentPlayer);
        }
    }

    if(getFigureOnBField(currentPlayer) != null)
    {
        if(canFigureBeMoved(getFigureOnBField(currentPlayer), currentPlayer, dieResult))
        {
            return getFigureOnBField(currentPlayer);
        }
    }

    if(getRemainingFiguresInOrder(currentPlayer).size() > 0)
    {
        for(Figure figure : getRemainingFiguresInOrder(currentPlayer))
        {
            if(canFigureBeMoved(figure, currentPlayer, dieResult))
            {
                return figure;
            }
        }
    }

    return null;

}
```

Abbildung 8: getFigureToMove-Methode

```java
public boolean canFigureBeMoved(Figure figure,int currentPlayer ,int dieResult)
{
    if(figure.getRelativePosition() == -1)
    {
        if(dieResult == 6 && getFigureOnAField(currentPlayer) == null)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    else
    {
        int nextPosition = figure.getRelativePosition() + dieResult;
        boolean isBlocked = false;

        for(Figure otherFigure : figures[currentPlayer])
        {
            if(otherFigure != figure && nextPosition == otherFigure.getRelativePosition())
            {
                isBlocked = true;
            }
        }


        if(nextPosition < 44 && !isBlocked)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

Abbildung 9: canFigureBeMoved-Methode

```java
public void moveFigure(Figure figure, int currentPlayer, int dieResult)
{
    if(figure.getRelativePosition() == -1 && dieResult == 6)
    {
        figure.setRelativePosition(0);
        figure.updateAbsolutePosition();
    }
    else
    {
        figure.setRelativePosition(figure.getRelativePosition() + dieResult);
        figure.updateAbsolutePosition();
    }

    int otherPlayer = Math.abs(currentPlayer - 1);

    for(Figure otherPlayerFigure : figures[otherPlayer])
    {
        if(figure.getAbsolutePosition() == otherPlayerFigure.getAbsolutePosition())
        {
            otherPlayerFigure.setRelativePosition(-1);
            otherPlayerFigure.updateAbsolutePosition();
        }
    }
}
```

Abbildung 10: moveFigure-Methode