NimMC

Vereinfachtes Monte-Carlo-Verfahren mit mcMove für das Nim-Spiel

Die Klasse *MoveResult*

Die Klasse *MoveResult* speichert einen Spielzug (*move*) sowie die Anzahl der Gewinne (*g*) und Verluste (*v*), die bei den Simulationen für diesen Zug erzielt wurden. Der Konstruktor initialisiert *g* und *v* jeweils mit 0, um mit einer leeren Zählung zu beginnen. Diese Struktur hilft uns später bei der Bewertung des besten Zugs, da *g* und *v* die Erfolgsbilanz jedes Zugs enthalten.

```
class MoveResult {
    Move move;
    int g; // Gewinne
    int v; // Verluste

    // Konstruktor
    MoveResult(Move move) {
        this.move = move;
        this.g = 0;
        this.v = 0;
    }
}
```

Die Methode *nimMc*:

Die Liste *possibleMoves*

In der *mcMove*-Methode wird *possibleMoves* als ArrayList genutzt, um alle möglichen Züge im aktuellen Spielzustand zu speichern. Diese Liste enthält für jeden Zug ein *MoveResult*-Objekt. Ich habe eine ArrayList gewählt, da sie eine schnelle (O(1)) Einfügeoperation bietet und die Anzahl der möglichen Züge in Nim relativ klein ist. Die Laufzeit bleibt daher effizient, selbst bei mehreren Durchläufen. Jeder Eintrag in der Liste repräsentiert einen möglichen Zug mit einem Ergebniszähler (*g* und *v*), der in den Simulationen gefüllt wird.

```
public Move mcMove(int N) {
    List<MoveResult> possibleMoves = new ArrayList<>(); // Liste mit allen möglichen
    // alle Züge nacheinander durchgehen und in der Liste abspeichern
    for (int row = 0; row < rows.length; row++) {
        for (int number = 1; number <= rows[row]; number++) {
            Move move = Move.of(row, number); // Spielzug erstellen und abspeichern
            MoveResult result = new MoveResult(move); // g und v werden für den neue
            possibleMoves.add(result); // Zug zur Liste hinzufügen
        }
    }
```

Die Simulation

Für jeden Zug in *possibleMoves* werden *N* zufällige Spiele simuliert. Der Zug wird auf den aktuellen Zustand angewendet, und das Spiel wird fortgesetzt, bis keine Züge mehr möglich sind. Eine innere Schleife erstellt *possibleMovesSim* und fügt alle möglichen Züge für den aktuellen Spielzustand hinzu. Falls *possibleMovesSim* leer ist, endet das Spiel. Bei jedem Zugwechsel wird *currentPlayerWins* umgeschaltet, um den Spielerwechsel zu simulieren. Nach jeder Simulation wird *g* erhöht, wenn der Spieler, der begann, gewann, oder *v*, falls nicht.

```
for (MoveResult result : possibleMoves) { // durch die Liste über jeden Spielzug
    for (int i = 0; i < N; i++) { // N-Simulationen für jeden Zug ausführen (N =
        Nim simulatedGame = this.play(result.move); // führt auf den aktuellen Z
        boolean currentPlayerWins = false; // verfolgt den aktuellen Spieler wäh

        // Simulation
        while (true) { // solange keine weiteren Züge mehr möglich sind
            List<Move> possibleMovesSim = new ArrayList<>(); // Liste mit möglic
            for (int ro = 0; ro < simulatedGame.rows.length; ro++) {
                for (int no = 1; no <= simulatedGame.rows[ro]; no++) {
                    possibleMovesSim.add(Move.of(ro, no));
                }
            }
            if (possibleMovesSim.isEmpty()) { // wenn keine weiteren Züge möglic
                break;  // Spielende
            }
            // wenn es noch Spielzüge gibt
            Move randomMove = possibleMovesSim.get(r.nextInt(possibleMovesSim.si
            simulatedGame = simulatedGame.play(randomMove); // zufälliger Spielz
            currentPlayerWins = !currentPlayerWins; // Spieler wechseln
        }
        // gewonnen oder verloren
        if (currentPlayerWins) {
            result.g++;
        } else {
            result.v++; // (der Spieler, der begonnen hat)
        }
    }
}
```

Auswerten des besten Zugs (*bestMove*)

Am Ende durchläuft die Methode *possibleMoves*, um den Zug mit der besten Erfolgsquote (*score = g / (g + v)*) zu finden. *bestMove* speichert den Zug mit dem höchsten Score; bei Gleichstand wählt ein zufälliger Boolean *nextBoolean()* einen der gleich guten Züge. In den Tests habe ich *N = 40000* gewählt, da dies auf einem MacBook M1 circa 2 Sekunden für die Berechnung des besten Zugs benötigt.

```
// Besten Zug anhand der Erfolgsrate auswählen
Move bestMove = null; // noch nicht bekannt
double bestScore = -1; // Startwert sorgt dafür, dass erster Score als bester Sc

for (MoveResult result : possibleMoves) { // über alle möglichen Züge iterieren
    double score = (double) result.g / (result.g + result.v); // Ergebnis liegt
    if (score > bestScore) {
        bestScore = score;
        bestMove = result.move;
    } else if (score == bestScore && r.nextBoolean()) { // zufällig einen von me
        bestMove = result.move; // zufällig bestimmt, falls true
    }
}

return bestMove;
```

Beispiel in der jshell:

```
jshell> Nim game = new Nim(3, 5, 4, 2, 6)
game ==>
I  I  I
I  I  I  I  I
```

```
I  I  I  I
I  I
I  I  I  I  I  I

jshell> Move bestMove = game.mcMove(40000)
bestMove ==> (4, 3)
```