Aufgabe 4: Würfelglück

Team-ID: 0051

Team: MWG go brrrrr

Bearbeiter/-innen dieser Aufgabe:
Theodor Teslia

11. November 2021

Inhaltsverzeichnis

Lösungsidee

Um zu entscheiden, welcher Würfel der beste für *Mensch ärger dich nicht* ist, habe ich jeden Würfel gegen jeden anderen 10000 mal spielen lassen. Dabei zählt der Algorithmus, wie oft jeder davon gewinnt und gibt am Ende in einer Tabelle diese Werte, bezogen auf die Gesamtanzahl der von diesen beiden Würfeln gespielten Spiele. Dazu gibt der Algorithmus davor aus, wie viele Spiele der Würfel jeweils gewonnen hat. Aus diesen Werten kann man darauf schliessen, welcher Würfel sich von den gegebenen Würfeln am besten eignet.

Umsetzung

Für diese Aufgabe wurde die Sprache C++ gewählt, da diese sehr schnell ausgeführt werden kann und das Programm dadurch mehr Iterationen durchführen kann, ohne sehr viel länger zu benötigen.

Die Implementierung des Spielfelds wurde für beide Spieler unabhängig als lineare Strecke umgesetzt. Das heißt, dass jeder Spieler selbst von 0 bis 43 (Ende des Spielbretts) läuft. Daher wird nach jeder Bewegung einer Figur in der dazugehörigen Funktion überprüft, ob der andere Spieler geschlagen werden kann.

Das Programm besteht aus den drei Klassen `Die`, `Figure` und `Player`. Die Klasse `Die` stellt einen Würfel dar und hat demnach auch die Methode um zu würfeln bzw. einen zufälligen Wert des Würfels zurückzugeben.

Die Klasse `Figure` simuliert eine Figur auf dem Spielbrett. Dafür hat es ein Attribut das seine Position darstellt und ein Wert eines `enums` mit dem Namen `Fields`, durch welches markiert wird, ob sich die Figur noch auf einem B-Feld oder dem Spielbrett befindet.

Die Klasse `Player` beherbergt die beiden grundlegenden Algorithmen zum Auswählen und darauf folgenden bewegen einer Figur, sowie dem testen, ob eine Figur des Gegenspielers geschlagen werden kann.

Die Funktion zum bewegen einer Figur ist die Funktion `Player::move()`. Diese würfelt eine Zahl, indem es die Funktion des Würfels, welche der Spieler grad hat, aufruft. Darauf folgt eine Überprüfung, ob eine Figur bereits auf dem Startfeld steht, welche dann wegbewegt werden muss. Weiter werden weitere Fälle überprüft und dementsprechend behandelt.

Die Funktion zum Schlagen eines anderen Spielers (`Player::canHit(Player& ply2)`) funktioniert wie folgt. Zuerst wird herausgefunden, welche Figuren beider Spieler auf dem Spielbrett und nicht auf einem B-Feld sind. Danach wird durch alle „aktiven“ Figuren iteriert und für jede wird eine „korrigierte“ Position berechnet. Diese stellt den Index dieser Figur dar, welcher vom Startpunkt des zweiten Spielers ausgeht. Wenn diese Position also auch von einer Figur des zweiten Spielers belegt wird, befinden sich beide auf dem gleichen Feld und die Figur des zweiten Spielers wird geschlagen.

Um diese Algorithmen umzusetzen, werden für jede Kombination von Würfeln 10000 Spiele gespielt. Für jedes neue Spiel beginnt der jeweils andere Spieler, es wird also abwechselnd angefangen.

Bei manchen Würfeln kann es dazu kommen, dass sich die Figuren nicht mehr bewegen können, da es keine genügend kleinen Zahlen gibt. Um ein dadurch resultierendes steckenbleiben des Programms zu verhindern, wird jedes Spiel abgebrochen, sobald es keine Bewegung von beiden Spielern innerhalb der letzten 200 Züge gab. Ab 500 Zügen wird zudem nach jedem Zug überprüft, ob beide Spieler noch gewinnen können.

Ausführen der Datei

Um die beigelegte Datei zu testen müssen folgenden Schritte befolgt werden.

Die Dateien wurden unter MacOS 11.5.2 erzeugt und mithilfe von folgendem Verfahren getestet.

Im Terminal (`zsh`) muss mithilfe von `cd` Das Arbeitsverzeichnis (`pwd`) auf den Ordner *bin* der Abgabe gesetzt werden. Die Datei lässt sich dann mithilfe von `./Aufgabe4` ausführen. Um verschiedene Beispiele von der BWINF-Webseite zu testen, muss einfach der Name der Datei, bspw. `wuerfel3.txt`, mit einem Leerzeichen abgetrennt dahinter geschrieben werden (sofern sich diese in dem Ordner *examples* befindet).

Beispiele

Erklärung der Ausgabe

Die Ausgabe verbindet zum einen die Information, wie viele Spiele ein Würfel von allen von ihm gespielten Spielen gewonnen hat, mit der Information, welcher Würfel gegen welchen anderen wie oft gewonnen hat. Dabei sind alle werte relative Mengen in %.

Schematische Ausgabe mit drei Würfeln (Es wurde nicht auf die Korrektheit der einzelnen Werte geachtet):

```
60.0:  100.00   70.00   31.29
10.0:   30.00  100.00   21.45
30.0:   68.71   78.55  100.00
```

Hier würde die erste Zeile den ersten Würfel der Eingabedatei darstellen, die Zweite den Zweiten usw. Die erste Spalte (links vom Doppelpunkt) stellt die relative Häufigkeit dar, mit der dieser Würfel alle seine Spiele gewonnen hat. Der erste Würfel hat also 60% aller seiner Spiele für sich entschieden, während der zweite Würfel dies nur in 10% der Fälle tun konnte.

Die darauffolgenden Werte in der Zeile zeigen an, wie oft der Würfel der Zeile gegen den Würfel der Spalte gewonnen hat. Der erste Würfel hat also 31.29% der Spiele gegen den dritten Würfel gewonnen.

Ausgaben der Beispieldateien

wuerfel0.txt:

```
 18.54:  100.00    1.97  100.00   54.57   54.61   83.11
 24.38:   98.03  100.00  100.00   96.17   91.42   96.81
  0.00:    0.00    0.00    0.00    0.00    0.00    0.00
 20.73:   45.43    3.83  100.00  100.00   49.58   77.29
 20.68:   45.39    8.58  100.00   50.42  100.00   75.71
 15.67:   16.89    3.19  100.00   22.71   24.29  100.00
```

Man kann sehen, dass der zweite Würfel mit den Seiten {1, 1, 1, 6, 6, 6} sich am besten eignet. Insgesamt hat er um etwa 4 Prozentpunkte die meisten Spiele gewonnen und im Spiel gegen jeden anderen Würfel hat er immer mehr als 91.41% der Spiele gewonnen. Dies erreicht kein anderer Würfel dieser Datei. Dies liegt vermutlich daran, dass die Wahrscheinlichkeit eine 6 zu würfeln sehr hoch ist und das Problem das viele große Würfel haben (dass sie nicht klein genug würfeln können um genau in das Ziel zu kommen) durch die drei Einsen auch hier nicht existiert.

wuerfel1.txt:

```
 27.03:  100.00   39.28   51.01   59.75   79.70   90.21
 23.94:   60.72   92.60   45.15   47.52   60.47   65.52
 19.64:   48.99   42.66   77.96   39.76   45.32   50.48
 16.55:   40.25   37.20   34.15   68.12   37.00   40.49
  8.66:   20.30   19.25   17.88   17.45   39.17   20.57
  4.17:    9.79    9.00    9.13    8.62    9.14   19.15
```

Hier ist der erste Würfel am sinnvollsten. Allein schon, da dieser der einzige Würfel ist, der es immer schafft das Spiel durchzuspielen. Die fehlende Eins, Zwei etc. bei den anderen verhindert teilweise, dass alle Figuren ins Ziel kommen können. Der zweite Würfel wäre auch noch möglich, da dieser das Beenden nur in 7.3% der Falle nicht schafft. Jedoch würde sich das nur lohnen, wenn alle anderen Gegner den ersten Würfel benutzen würden, da es sonst wegen der höheren Wahrscheinlichkeiten sinnvoller ist den ersten Würfel zu nehmen.

wuerfel2.txt:

```
  6.71:  100.00    0.64    0.00    0.00    0.00
 13.55:   99.36  100.00    3.75    0.07    0.00
 20.27:  100.00   96.25  100.00    7.47    0.28
 26.81:  100.00   99.93   92.53  100.00    9.76
 32.66:  100.00  100.00   99.72   90.24  100.00
```

Am besten schneidet hier der fünfte Würfel ab. Insgesamt gewinnt dieser fast ein Drittel aller Spiele und nur im Spiel gegen den vierten Würfel ist die Wahrscheinlichkeit zu gewinnen mit 90.24% *relativ* niedrig. Der Grund dafür ist wahrscheinlich der gleiche wie bei der Wahl des zweiten Würfels in *wuerfel0.txt*. Bloß hat dieser noch mehr Sechsen und kann dadurch sehr oft hintereinander würfeln und durch die Eins kann er sich dazu noch genau im Ziel bewegen.

wuerfel3.txt:

```
 21.52:  100.00   71.50   55.07   70.21   66.76   88.46
 17.07:   28.50  100.00   38.19   54.41   54.62   82.77
 19.85:   44.93   61.81  100.00   64.26   60.79   85.07
 16.15:   29.79   45.59   35.74  100.00   49.65   78.44
 16.33:   33.24   45.38   39.21   50.35  100.00   74.79
  9.07:   11.54   17.23   14.93   21.56   25.21   99.99
```

Von allen Beispieldateien ist die Entscheidung hier am schwierigsten. Am besten wäre aber wahrscheinlich der erste. Dieser hat gegen jeden anderen Würfel mehr als 50% Gewinnwahrscheinlichkeit und gewinnt mit 21.52% die meisten der Spiele. Das ist wahrscheinlich der Fall, da der Würfel mit Eins und Zwei zwei kleine Zahlen zum genaueren Bewegen und mit Fünf und Sechs zwei hohe Zahlen zum schnellen Bewegen besitzt. Außerdem hat der Würfel nur vier Seiten, wodurch die Wahrscheinlichkeit für eine 6 im Vergleich zu einem normalen Würfel hoch ist.

Fazit

Der perfekte Würfel hätte am besten möglich wenig Seiten mit möglichst vielen Sechsen, aber auch eine Eins ist sehr wichtig um ein Beenden des Spielen zu garantieren. Alternativ zu wenigen Seiten wären viele Seiten mit fast nur Sechsen am besten, um das Spiel als erstes zu gewinnen.

Quellcode

```
bool Player::move() {
	if (this->currentDie == nullptr) {
		return false;
	}
	this->alreadyMovedInTurn = false;
	bool changed = false;

	int dieResult = this->currentDie->roll();

	if (onField(0)) {  // Still figure on start field -> Needs to be moved
away
		for (int i = figs.size() - 1; i >= 0; i--) {  // Start from last
figure
			int nextPosition = figs.at(i).getPosition() + dieResult;
			if (nextPosition <= boardLength-1 && !onField(nextPosition) &&
figs.at(i).getField() == Fields::RUNWAY) {
				changePosition(i, dieResult);
				changed = true;
				break;
			}
		}

		if (dieResult == 6) {
			this->move();
		}
		return changed;
	}

	if (dieResult == 6 && !everythingOn(Fields::RUNWAY)) {  // Set figure from
B field on runway
		int firstOnBIndex = -1;
		for (int i = 0; i < figs.size(); i++) {
			if (figs.at(i).getField() == Fields::B) {
				firstOnBIndex = i;
				break;
			}
```

```
            }
            if (firstOnBIndex != -1) {
                  figs.at(firstOnBIndex).setField(Fields::RUNWAY);
                  figs.at(firstOnBIndex).setPosition(0);
                  occupiedSpaces.insert(0);
                  changed = true;

                  sortFigures();
                  this->alreadyMovedInTurn = true;

                  this->move();
                  return changed;
            }
      }

      if (dieResult != 6 && everythingOn(Fields::B)) {  // Everything on B and
no 6 -> Nothing can move out
            return changed;
      }
      for (int i = 0; i < figs.size(); i++) {  // Search for first element that
can move
            int nextPos = figs.at(i).getPosition() + dieResult;

            if (nextPos <= boardLength-1 && !onField(nextPos) &&
figs.at(i).getField() == Fields::RUNWAY) {
                  changePosition(i, dieResult);
                  changed = true;
                  break;
            }
      }

      if (dieResult == 6) {
            this->move();
      }
      return changed;
}


void Player::canHit(Player& ply2) {
      std::vector<Figure*> f1, f2;
      fillActivePlayers(f1, this->figs);  // Ignore player on B fields
      fillActivePlayers(f2, ply2.figs);

      int tempBoardLength = boardLength-4;

      for (auto& i : f1) {
            int adjustedIndex = (tempBoardLength + i->getPosition() -
(tempBoardLength-4)/2) % tempBoardLength;  // Readjust board for offset second
board
            for (auto& j : f2) {

                  if (adjustedIndex == j->getPosition()) {
                        ply2.occupiedSpaces.erase( ply2.occupiedSpaces.find(j-
>getPosition()) );
                        j->setField(Fields::B);
                        j->setPosition(-1);
                        ply2.sortFigures();
```

```
                    }
                }
            }
        }
```