Aufgabe 3: Tobis Turnier

Team-ID: 00383

Team: Theodor Teslia

Bearbeiter/-innen dieser Aufgabe:
Theodor Teslia

23. Oktober 2020

Inhaltsverzeichnis

Lösungsidee

Meine Idee war es, die einzelnen Turniervarianten zu programmieren, diese dann basierend auf den Eingabedateien über 100.000 Wiederholungen zu spielen und dann am Ende zu berechnen, wie oft der beste Spieler im Durchschnitt gewonnen hat. Die Berechnung habe ich so durchgeführt, dass die Wahrscheinlichkeit, die am Ende rauskommt, beschreibt, wie wahrscheinlich es ist, dass ein Spieler gewinnt, der die höchste Stärke hat. Das heißt, wenn zwei Spieler die gleiche maximale Spielerstärke haben, wird die Wahrscheinlichkeit für den Sieg mit dieser Anzahl an Stärkepunkten die Wahrscheinlichkeiten von beiden Spielern addiert sein. Ich denke, dass das sinnvoll ist, da Tobi wissen möchte, welche Turniervariante er auswählen sollte, damit die Wahrscheinlichkeit für den Sieg des besten Spielers am höchsten ist. Da es aber egal ist, welcher der besten Spieler gewinnt, sofern es mehrere gibt, sondern darauf geachtet wird, ob der Gewinner „rechtmäßig“ gewonnen hat, also auch in einem nicht zufallsbasierten Spiel gewonnen hätte, finde ich, dass man dies so rechnen sollte.

Umsetzung

Benutzt habe ich die Sprache C++. Die verschiedenen Turniervarianten habe ich in verschiedenen Klassen geschrieben (`Liga`, `KO` und `KOx5`), die alle von der Klasse `Turnier` ein `std::multiset` mit dem Datentyp `Player`, eine Methode zum Hinzufügen von Spielern und

einer virtuellen Methode, die einen Durchlauf simuliert (d.h. die jeweilige Turniervariante spielt, bis ein Gewinner feststeht). Diese heißt `play()`.

`Player`

Die Klasse `Player` besitzt die zwei Attribute `strength`, das die Spielstärke speichert und `playerNumber`, welches die eindeutige Nummer eines Spielers speichert. Diese Struktur stellt somit einen Spieler dar, der an dem Turnier mitspielt.

`Turnier`

Die Klasse `Turnier` besitzt eine öffentliche Methode (`play()`), welche virtuell ist, wodurch auch die Klasse virtuell ist. Auch hat sie ein `std::multiset<Player>` Attribut, das `players` heißt, welches `protected` ist. Dieses Attribut speichert alle Spieler, aufwärts sortiert, nach ihrer Spielerstärke.

Liga

Logischerweise erbt diese Klasse alle Eigenschaften der `Turnier`-Klasse, fügt aber noch mehr hinzu und implementiert die virtuelle Methode `play()`. Hinzu kommt eine `std::map<int, int>`, welche im Key die Spielernummer eines Spielers und die im Wert die Anzahl der von diesem gewonnen Spiele speichert.

Um dann eine Runde zu spielen, wird in einer verschachtelten for-Schleife zwei mal durch das geerbte Attribut `players` iteriert, wobei jede Schleife aufhört, wenn der Iterator gleich `players.end()` ist und die innere Schleife beginnt bei dem Objekt nach dem Objekt, auf das der Iterator der äußeren Schleife zeigt. In dem Körper der inneren Schleife wird dann die Methode Liga::playRound() gespielt. Diese gibt die Spielernummer des dadurch ermittelten Gewinners zurück, dessen dazugehöriger Wert in der `std::map<int, int>` dann um 1 erhöht wird. Schlussendlich wird durch alle Werte dieser map durchgelaufen und der maximale Wert gespeichert, sodass dieser zurückgegeben werden kann.

`Liga::playRound()` und `KO::playRound()`

Beide Methoden unterscheiden sich bloß darin, dass `KO::playRound()` eine passive Referenz von einem Spieler zurückgibt und `Liga::playRound()` die Spielernummer des Spielers zurückliefert. Beide Methoden bekommen als Parameter zwei passive Referenzen eines Spielers (`ply1` und `ply2`), die gegeneinander spielen.

Die Methode beginnt durch das überprüfen, ob beide Parameter eine Spielstärke von 0 haben, wenn dies der Fall ist wird per 50/50 Chance entschieden, welcher Spieler gewinnt und wird dementsprechend zurückgegeben. Ansonsten wird eine Zufallszahl erzeugt, die von 0 bis zur Menge der „eingelegten Kugeln-1“ geht. Wenn diese Zahl größer oder gleich der Spielerstärke des zweiten

Spielers hat dieser gewonnen und er wird zurückgegeben, ansonsten passiert dies mit dem ersten Spieler.

KO

Die Klasse KO besitz ein enum, mit drei Einträgen, die die Möglichkeiten zum paaren der Teilnehmer darstellen und ein `std::vector<Player>` mit dem Namen `plyVec` welches als Zweitspeicher für die Spieler dient. Die drei Möglichkeiten zum paaren sind: Sortiert nach aufsteigender Spielerstärke (Dafür werden die Spieler aus `players` einfach in `plyVec` kopiert, da die Spieler dort schon sortiert sind), zufällig (Dafür wird einfach `std::shuffle()` benutzt) und nach dem Besten gegen den schlechtesten, dem zweitschlechtesten gegen den zweibesten usw. Um dies zu erreichen iterieren zwei Iteratoren einmal von vorne und einmal von hinten und kopieren ihre Objekte in `plyVec`.

Die Methode `play()` ist so implementiert, dass nach dem paaren der Spieler eine while-Schleife solange läuft, wie `plyVec` eine größere Größe als 1 hat. In der while-Schleife wird durch `plyVec` iteriert und es spielen immer die beiden ersten Spieler gegeneinander. Danach werden beide gelöscht und der Gewinner wird in einen Zwischenspeicher kopiert. Nach dem Ablauf der Schleife durch `plyVec` ist dieses leer und die Inhalte des Zwischenspeichers werden hinein kopiert, wodurch auch `plyVec` geleert wird.

KOx5

KOx5 erbt alle Eigenschaften von KO, weshalb auch der Programmablauf fast genauso ist. Unterschiedlich ist bloß die Methode `playRound()`. Diese ist so implementiert, dass fünf mal die Methode KO::playRound() aufgerufen und aufgezeichnet wird, welcher Spieler wie oft diese Runden gewinnt. Nach dem fünfmaligen Ausführen wird überprüft, welcher Spieler öfter gewonnen hat und gibt dementsprechend einen Spieler zurück.

Ausführung

Folgendes gilt nur für das Betriebssystem Mac OS X (10.15.7)

- Das aktuelle Arbeitsverzeichnis (*pwd*) lautet: */Users/BwInf*
- Die Ausführbare Datei befindet sich im Verzeichnis: */Users/BwInf/2020/Runde1/TheodorTeslia/Aufgabe3/bin*
- In Terminal eingeben: *cd 2020/Runde1/TheodorTeslia/Aufgabe3/*
- Dann: *./Aufgabe3*
    - Durchläuft das Programm mit der Standard-Eingabedatei (wichteln1.txt)
    - Zum testen mit anderen Dateien: *./Aufgabe3 <TextDatei.txt>*

- *<TextDatei.txt>* muss sich in Verzeichnis */Users/BwInf/2020/Runde1/TheodorTeslia/Aufgabe3/examples* befinden

Beispiele

In den Zellen der Tabellen ist die relative Häufigkeit des Sieges des besten Spielers (weitere Erklärung dazu siehe Lösungsidee).

Spielstaerken1.txt:

	Aufsteigend sortiert	**Zufällig angeordnet**	**Bester gegen schlechtester**
Liga	34,616%		
KO	35,249%	40,682%	47,499%
KOx5	55,52%	60,018%	61,01%

Spielstaerken2.txt:

	Aufsteigend sortiert	**Zufällig angeordnet**	**Bester gegen schlechtester**
Liga	20,656%		
KO	28,044%	30,116%	30,662%
KOx5	36,385%	35,548%	36,344%

spielstaerken3.txt:

	Aufsteigend sortiert	**Zufällig angeordnet**	**Bester gegen schlechtester**
Liga	31,531%		
KO	13,632%	16,567%	21,108%
KOx5	22,517%	28,273%	31,552%

spielstaerken4.txt:

	Aufsteigend sortiert	**Zufällig angeordnet**	**Bester gegen schlechtester**
Liga	11,515%		
KO	6,901%	7,071%	6,83%
KOx5	7,506%	7,612%	7,563%

Eigenes Beispiel: *beispiel1.txt*:

Werte: 0, 0, 0, 0, 0, 0, 0, 20; Sinn: Testen, was bei mehreren Spielern mit der Stärke 0 passiert

	Aufsteigend sortiert	**Zufällig angeordnet**	**Bester gegen schlechtester**
Liga	100%		
KO	100%	100%	100%
KOx5	100%	100%	100%

Fazit

Mein Fazit nach dem Ausführen aller Möglichkeiten und aller Dateien ist, dass, wenn Tobi die Stärken der Spieler nicht kennt, definitiv die Turnierform KO x5 nehmen und die Paare nach dem Schlechtestem-mit-dem-Besten-Prinizip paaren sollte. Dies liegt daran, dass in einer Mehrheit der Fälle, dieser Weg zu dem besten Ergebnis, nämlich, dass ein Spieler gewinnt, der die höchste Stärke hat, führt.

Wenn Tobi aber die Stärken der Spieler kennt, sollte er sich für das Liga-Prinizp entscheiden, wenn es sehr viele Spieler gibt, die sehr nah an der höchsten vorkommenden Stärke liegen. Dies sieht man sehr gut an *spielstaerken4.txt*, da dort alle Spieler, bis auf einen, eine Stärke von 95 haben und der letzte eine Stärke von 100 besitzt. Das Ergebnis bei dieser Auslegung war, dass man das sinnvollste Ergebnis bei der Liga erziehlt.

Quellcode

```
int Liga::play() {
	this->initializeScores();

	for (std::set<Player>::iterator i = players.begin(); i != players.end(); i++) {  // Everyone "fights" against everyone
		for (std::set<Player>::iterator j = std::next(i, 1); j != players.end(); j++) {
			int winningPlayer = Liga::playRound((Player&)*i, (Player&)*j);
			this->scores[winningPlayer] = scores.at(winningPlayer) + 1;
		}
	}

	std::pair<int, int> bestPlayer(scores.begin()->first, scores.begin()->second);
	for (std::map<int, int>::iterator i = scores.begin(); i != scores.end(); i++) {
		if (i->second > bestPlayer.second) {
			bestPlayer.first = i->first;
			bestPlayer.second = i->second;
		} else if (i->second == bestPlayer.second && i->first < bestPlayer.first) {
			bestPlayer.first = i->first;
```

```
                        bestPlayer.second = i->second;
                }
        }  // Determine the best Player by saving the current best Player and comparing to the next Player

        this->scores.clear();
        return bestPlayer.first;  // return playernumber of winning player
}

int KO::play() {  // Play a tournament
        fillPlayerVec();
        std::vector<Player> winnerBank;

        organizeThePairs(WORSTVSBEST); // Other options: ORDER, RANDOM

        while (plyVec.size() > 1) {  // Loop until the winner is determined
                std::size_t formerSize;

                formerSize = plyVec.size();
                for (int i = 0; i < formerSize/2; i++) {  // Two players play against each other, winner is
stored in second arr
                        Player winningPlayer = KO::playRound(plyVec.at(0), plyVec.at(1));
                        plyVec.erase(plyVec.begin(), plyVec.begin() + 2);
                        winnerBank.push_back(winningPlayer);
                }  // Both elements are deleted afterwards

                plyVec = winnerBank;
                winnerBank.clear();
        }

        return plyVec.at(0).getNumber();  // Return the last remaining player's number
}

Player& KO::playRound(Player &ply1, Player &ply2) {  // Simulates a match between two players
        int rndNum = 0;
        if ((ply1.getStrength() + ply2.getStrength() != 0)) {
                rndNum = rand() % (ply1.getStrength() + ply2.getStrength());
                if (rndNum >= ply1.getStrength()) {
                        return ply2;  // Return playernumber
                } else {
                        return ply1;
                }
        } else {  // Prevent error if both players have strength of 0
```

```
                rndNum = rand() % 2;  // Decide by a chance of 50/50 who wins
                if (rndNum == 0)
                        return ply1;
                else
                        return ply2;
        }
}


void KO::organizeThePairs(concretePlan plan) {
        if (plan == ORDER) {  // Players are sorted by increasning strength
                return;
        } else if (plan == RANDOM) {  // The players are randomly distributed
                auto rd = std::random_device {};
                auto rng = std::default_random_engine { rd() };
                std::shuffle(plyVec.begin(), plyVec.end(), rng);
        } else if (plan == WORSTVSBEST) {  // The worst plays against the best, the 2. best against the 2. worst
and so on
                plyVec.clear();
                auto reverse = std::prev(players.end(), 1);  // Iterators from front and back
                auto normal = players.begin();
                for (auto i = 0; i < players.size()/2; i++, normal++, reverse--) {
                        plyVec.push_back(*normal);
                        plyVec.push_back(*reverse);
                }
        }
}
```