Prof. Dr. Ralf Hinze
Sebastian Schweizer, M.Sc.
Markus Heinrich, M.Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Projekt 2: Programmierpraktikum 2020

Ausgabe: 23. Juni 2020
Abgabe: 14. Juli 2020, 15 Uhr

GitLab Team Repositories

Wir verwenden auch für die Projekte wieder die GitLab Repositories. Führen Sie den Befehl `git pull` aus, damit die Vorlagen heruntergeladen werden. Sie sollten nun einen neuen Ordner `P2` sehen, den Sie wie gewohnt in IntelliJ IDEA öffnen können. Es handelt sich wieder um ein Gradle Projekt, sodass Abhängigkeiten automatisch heruntergeladen werden.

Ihre Lösungen müssen Sie in das lokale Repository committen (`git commit`) und dann auf den GitLab Server übertragen (`git push`). Sprechen Sie sich zur Bearbeitung der Aufgaben mit Ihren Abgabepartner*innen ab, damit Sie unnötige Konflikte vermeiden.

Bearbeitungshinweise

Sie haben für die Bearbeitung des Projekts drei Wochen Zeit, d. h. es ist entsprechend umfangreich. Beginnen Sie also rechtzeitig mit der Bearbeitung und nutzen Sie die Sprechstunden!

Wir erwarten, dass folgende Meilensteine erreicht werden (d. h. Code befindet sich auf GitLab):

1. Bis Dienstag, 30.06.2020 23:59 Uhr, müssen – je nachdem, was Sie zuerst bearbeiten möchten – entweder die Implementierung oder die Testfälle[1] für Aufgabe 1 fertig sein.
2. Bis Dienstag, 07.07.2020 23:59 Uhr, muss Aufgabe 1 komplett fertig bearbeitet sein und Sie sollten von Aufgabe 2 zumindest den gesamten Aufgabentext inkl. Anhang gelesen haben.

Einleitung

In diesem Projekt werden wir ein Spiel entwickeln, bei dem mehrere Spieler versuchen, die Knoten eines Graphen mit der eigenen Farbe einzufärben. Die genaue Beschreibung des Spiels finden Sie in Aufgabe 2.

Zuerst sollen Sie jedoch in Aufgabe 1 eine Datenstruktur für gerichtete Graphen implementieren. Wie schon im letzten Projekt müssen Sie auch hier eigene Testfälle für die Datenstruktur erstellen.

In Aufgabe 2 implementieren und testen Sie die Spiellogik.

Aufgabe 3 besteht darin, Ihr Spiel auszuprobieren. Das Spiel bietet eine graphische Benutzeroberfläche. Den Code dazu haben wir Ihnen vorgegeben. Das Programmieren von graphischen Benutzeroberflächen gehört nicht zu den Lernzielen des Programmierpraktikums, diese Kompetenz können Sie zum Beispiel im *Software-Entwicklungsprojekt*[2] erwerben.

In der freiwilligen Aufgabe 4 können Sie einen Computerspieler implementieren.

[1] Wenn Sie sich bei diesem Meilenstein für die Testfälle entscheiden, dann sollten diese einen Umfang haben, mit dem die vorgegebenen Coverage Kriterien voraussichtlich erreicht werden. Sie brauchen für jede Methode aus der `Graph` Schnittstelle mehrere Testfälle: Für das erfolgreiche Ausführen der Methode sowie Tests für die möglichen Ausnahmen.

[2] https://www.cs.uni-kl.de/studium/lehrveranstaltungen/modulhb/#89-0220

Aufgabe 1 Gerichtete Graphen

In der Projektvorlage finden Sie die Schnittstelle `de.tukl.programmierpraktikum2020.p2.a1.Graph`. Die einzelnen Methoden sind mit Javadoc versehen. Lesen Sie sich den restlichen Aufgabentext sowie die Javadoc Kommentare durch, um die Schnittstelle zu verstehen.

Die Schnittstelle `Graph` modelliert einen gerichteten Graphen. Der Graph besteht aus einer Menge von Knoten (engl. *node*). Zwischen den Knoten gibt es Kanten (engl. *edge*), die jeweils eine Richtung haben. Für zwei Knoten a, b unterscheiden wir also zwischen den Kanten $a \to b$ und $b \to a$. Der Graph kann keine, eine oder beide dieser Kanten enthalten. Dieselbe Kante (also in derselben Richtung) kann jedoch nicht mehrfach enthalten sein. Auch die Kante $a \to a$ ist möglich.

In den Knoten speichern wir Daten (engl. *data*). Jede Kante ist mit einer Gewichtung (engl. *weight*) versehen. Die Typparameter `D` und `W` der Schnittstelle sind die Typen für die Daten und Gewichtungen.

Der Benutzer der Schnittstelle muss die Knoten durch eindeutige Identifier vom Typ `int` referenzieren. Mit der Methode `int addNode(D data)` wird ein neuer Knoten im Graph hinzugefügt. Der Identifier des neuen Knotens wird von der `Graph` Implementierung vorgegeben und in dieser Methode zurückgegeben. Die `Graph` Implementierung kann zur Festlegung der Identifier die erstellten Knoten fortlaufend nummerieren oder eine andere Vergabemethode benutzen, solange garantiert ist, dass die vergebenen Identifier eindeutig sind.

Sollte der Benutzer der Schnittstelle eine Methode wie z. B. `void setData(int nodeId, D data)` mit einem ungültigen Knoten-Identifier aufrufen, so wird die Ausnahme `InvalidNodeException` geworfen.

Kanten werden durch die Identifier der beiden angrenzenden Knoten identifiziert. Der Aufruf

```
graph.addEdge(42, 4711, weight)
```

fügt eine neue Kante $42 \to 4711$ mit der Kantengewichtung `weight` hinzu. Falls die übergebenen Knoten-Identifier ungültig sind, wird auch hier eine `InvalidNodeException` geworfen. Falls die Kante (in der angegebenen Richtung) bereits im Graph existiert, so wird eine `DuplicateEdgeException` geworfen.

Mit der Methode `W getWeight(int fromId, int toId)` kann die Kantengewichtung abgefragt werden. Die Methode `void setWeight(int fromId, int toId, W weight)` aktualisiert die Gewichtung. Falls die angegebene Kante nicht existiert, werfen die beiden Methoden die Ausnahme `InvalidEdgeException`, unabhängig davon, ob die angegebenen Knoten-Identifier gültig sind. Es wird keine `InvalidNodeException` geworfen.

Die Methode `Set<Integer> getIncomingNeighbors(int nodeId)` gibt die Menge aller Knoten-Identifier x zurück, für die die Kante $x \to$ `nodeId` im Graph enthalten ist. Umgekehrt gibt die Methode `Set<Integer> getOutgoingNeighbors(int nodeId)` die Menge aller Knoten-Identifier y zurück, für die die Kante `nodeId` $\to y$ im Graph enthalten ist.

Legen Sie Ihre Lösungen der folgenden Aufgaben im Paket `de.tukl.programmierpraktikum2020.p2.a1` ab.

a) Implementieren Sie eine Klasse `GraphImpl implements Graph`. Überlegen Sie sich, wie Sie den Graph in Java repräsentieren möchten. Es gibt verschiedene sinnvolle Möglichkeiten. Achten Sie darauf, dass die Implementierung effizient ist. Sie können für Ihre Implementierung gerne Werkzeuge und Datenstrukturen aus der Java Standardbibliothek benutzen (Listen, Arrays, Maps, Sets, ...). Sie dürfen jedoch keine Bibliothek zur Repräsentation von Graphen verwenden, da dies das Lernziel torpedieren würde!

b) Erstellen Sie JUnit Tests für Ihre `Graph` Implementierung. Wenn Sie in der Testmethode Methoden aufrufen, die Ausnahmen werfen können, dann müssen Sie auch die Testmethode mit `throws GraphException` versehen. Zum Testen einer erwarteten Ausnahme sei auf die JUnit Dokumentation[3] verwiesen.

   Stellen Sie dabei sicher, dass Sie (auch ohne die Tests von Aufgabe 2b) eine Instruction Coverage von 100% und eine Branch Coverage von mindestens 95% erreichen.

[3] https://howtodoinjava.com/junit5/expected-exception-example/

Aufgabe 2 Spiellogik

In unserem Spiel spielen zwei bis vier Spieler gegeneinander. Das Spielfeld ist ein Graph mit farbigen Knoten. Jeder Spieler hat eine eigene Farbe. Zu Beginn des Spiels sind alle Knoten weiß – diese Farbe gehört keinem Spieler. Die Spieler sind abwechselnd an der Reihe und können jeweils einen Knoten des Graphen mit der eigenen Farbe einfärben. Ziel des Spiels ist es, möglichst viele Knoten mit der eigenen Farbe zu versehen.

Das taktische Element des Spiels besteht darin, dass Knoten auch unabhängig von der Einfärbung durch einen Spieler ihre Farbe wechseln können: Wenn für einen Knoten x mehr als die Hälfte der eingehenden Kanten von Knoten einer einheitlichen Farbe c stammen, dann wechselt die Farbe von Knoten x auf c.

Das Spielfeld ist ein `Graph<Color, Integer>`. Die Kanten sind also mit `Integer` Werten gewichtet, mehr dazu später. Der für die Knotenmarkierung verwendete Typ `Color` ist folgendes Enum:

```
public enum Color {
    WHITE, RED, GREEN, BLUE, YELLOW
}
```

Zu Beginn des Spiels haben alle Knoten die neutrale Farbe `WHITE`. Die übrigen Farben gehören den Spielern.

Die Spieler sind abwechselnd an der Reihe und müssen einen der folgenden Spielzüge ausführen:

- Die Farbe eines beliebigen Knotens auf die eigene Spieler-Farbe ändern.
- Die Gewichtung einer beliebigen Kante um eins erhöhen.
- Die Gewichtung einer beliebigen Kante um eins verringern, wobei die Kantengewichtung dadurch nicht negativ werden darf. Alle Kanten haben zu jeder Zeit eine Gewichtung ≥ 0.

Nach jedem Spielzug wird überprüft, ob weitere Knoten ihre Farbe ändern müssen. Für einen Knoten x sei

- $w_{\text{total}}(x)$ die Summe der Kantengewichtungen aller eingehenden Kanten $\star \to x$ und
- für eine Spieler-Farbe $c \neq$ `WHITE` sei $w_c(x)$ die Summe der Kantengewichtungen aller eingehenden Kanten $y \to x$ von Knoten y, deren aktuelle Farbe c ist.

Falls $w_c(x) > \frac{w_{\text{total}}(x)}{2}$ ist, so erhält Knoten x automatisch die Farbe c. Beachten Sie, dass die automatische Umfärbung eines Knotens weitere Umfärbungen auslösen kann. Ein Knoten kann niemals zur neutralen Farbe weiß umgefärbt werden. Ein paar konkrete Beispiele finden Sie im Anhang.

Kein gültiger Spielzug ist es, die Farbe eines Knotens zu ändern, wenn dieser gemäß der eben genannten Regel seine aktuelle Farbe aufgezwungen bekommt. Es ist jedoch erlaubt, einen Knoten, der bereits die eigene Farbe hat, erneut einzufärben. Damit verschenkt man zwar seinen Spielzug, aber es ist regelkonform. Beachten Sie bitte die Beispiele im Anhang, insbesondere die Beispiele 4 – 6.

Die Struktur des Graphen wird zu Spielbeginn fest vorgegeben. Während des Spiels können keine Knoten oder Kanten hinzugefügt oder entfernt werden. Im Spielverlauf verändert werden können lediglich die Farben der Knoten sowie die Gewichtungen der Kanten. Zu Spielbeginn sind alle Knoten weiß und die Kantengewichtungen sind ≥ 0.

In der Projektvorlage finden Sie die Schnittstelle `de.tukl.programmierpraktikum2020.p2.a2.GameMove`, die für jeden der möglichen Spielzüge eine Methode beinhaltet:

- **`void`** `setColor(`**`int`** `nodeId, Color color)` ändert die Farbe des Knotens `nodeId` auf `color`. Falls der Spielzug nicht legal sein sollte, weil der Knoten direkt wieder seine bisherige Farbe annehmen würde, dann wird eine `ForcedColorException` geworfen.
- **`void`** `increaseWeight(`**`int`** `fromId, `**`int`** `toId)` erhöht die Gewichtung der Kante `fromId` → `toId` um eins.
- **`void`** `decreaseWeight(`**`int`** `fromId, `**`int`** `toId)` verringert die Gewichtung der Kante `fromId` → `toId` um eins. Sofern die Gewichtung dadurch negativ werden würde, wird eine `NegativeWeightException` geworfen und die Gewichtung bleibt unverändert.

Legen Sie Ihre Lösungen der folgenden Aufgaben im Paket `de.tukl.programmierpraktikum2020.p2.a2` ab.

a) Implementieren Sie eine Klasse `GameMoveImpl` **`implements`** `GameMove`. Der Konstruktor der Klasse erhält als Parameter einen Graphen vom Typ `Graph<Color, Integer>`. Dieser Graph stellt den Initialzustand dar, d. h. die für das Spiel zu verwendende Struktur wurde angelegt und alle Knoten sind weiß. Sie sollen im Konstruktor keine Änderungen an diesem Graphen vornehmen.

   Denken Sie daran, in den drei Methoden die oben erklärte automatische Umfärbung zu implementieren.

b) Erstellen Sie JUnit Tests für Ihre `GameMove` Implementierung. Sie können sich an den Beispielen im Anhang orientieren. Es bietet sich an, pro Beispiel einen Testfall (d. h. eine mit `@Test` markierte Methode) zu erstellen und darin die Spielzüge auszuführen. Nach jedem Spielzug sollte mit `assertEquals` und der `getData` Methode aus der `Graph` Schnittstelle die Farbe aller Knoten überprüft werden.

   Sie müssen dem Konstruktor der Klasse `GameMoveImpl` ja einen `Graph<Color, Integer>` übergeben. Verwenden Sie hierzu die Klasse `GraphImpl` aus Aufgabe 1. Erstellen Sie einige sinnvolle Graphen, die die Anforderungen für den Anfangszustand des Spiels erfüllen, und führen Sie dann mit den `GameMove` Methoden darauf unterschiedliche Spielzüge aus. Am Ende müssen Sie überprüfen, dass die von den Spielregeln vorgegebenen Änderungen am Graph vorgenommen wurden.

   Erstellen Sie mindestens einen Test so wie in dieser Teilaufgabe gefordert (also mit `GraphImpl`).

c) Mit den Testfällen der vorherigen Teilaufgabe testen Sie nicht nur Ihre `GameMove` Implementierung, sondern auch die `Graph` Implementierung. Dadurch handelt es sich dabei streng genommen nicht mehr um Komponententests sondern um Integrations- oder Systemtests. Das Problem dabei ist, dass es deutlich schwieriger ist, einen fehlgeschlagenen Testfall zu analysieren: Wir wissen nicht, ob das Problem in der Implementierung für `Graph` oder in der für `GameMove` liegt, oder ob der Test selbst fehlerhaft ist.

   Daher sollen Sie nun weitere Tests erstellen, die unabhängig von der Implementierung aus Aufgabe 1 sind. Erstellen Sie dazu innerhalb des `src/test` Ordners eine oder mehrere weitere Implementierungen der Klasse `Graph` und verwenden Sie diese in Ihren Testfällen. Der gewaltige Unterschied ist nun, dass Sie die Struktur des Graphen (d. h. die Knoten und Kanten) in der Klasse fest vorgeben können. Für einen Graphen mit drei Knoten enthält diese spezielle `Graph` Implementierung also drei Objektattribute vom Typ `Color`. Die Kantengewichtungen können als einzelne Objektattribute vom Typ **`int`** gespeichert werden. Sie benötigen also kein Array, keine Liste, keine Map oder ähnliche Datenstrukturen. Da die Methoden `addNode` und `addEdge` durch `GameMoveImpl` nicht aufgerufen, können Sie diese in der Implementierung auslassen: **`throw new`** `UnsupportedOperationException();`

   Erstellen Sie mindestens einen Test so wie in dieser Teilaufgabe gefordert (also ohne `GraphImpl`).

d) Stellen Sie sicher, dass Sie mit den Testfällen der beiden vorherigen Teilaufgaben für den in Teilaufgabe a) geschriebenen Code eine Instruction Coverage von mindestens 95% und eine Branch Coverage von mindestens 85% erreichen. Außerdem müssen alle im Anhang gezeigten Beispiele getestet werden.

   Wie Sie diese Anforderung auf die beiden Vorgehensweisen (mit oder ohne Verwendung von `GraphImpl`) aufteilen, bleibt Ihnen überlassen. Wir verlangen lediglich, dass Sie für beide Vorgehensweisen je mindestens einen Testfall erstellen.

Aufgabe 3 Spiel starten

Wie eingangs schon erwähnt haben wir Ihnen den kompletten Code für die graphische Benutzeroberfläche vorgegeben. Des Weiteren haben wir Methoden bereitgestellt, die basierend auf der Delaunay-Triangulierung[4] Knoten und Kanten so generieren, dass sich der Graph schön darstellen lässt. Außerdem liefern wir eine recht simple Implementierung für einen Computerspieler mit, sodass Sie gegen den Computer spielen können. Diese Implementierung dürfen Sie gerne in der freiwilligen Aufgabe 4 optimieren.

Für diese Aufgabe müssen Sie die Klasse `de.tukl.programmierpraktikum2020.p2.main.Factory` bearbeiten. Die beiden enthaltenen statischen Methoden sollen Objekte vom Typ `Graph` bzw. `GameMove` zurückgeben. In der Vorlage werfen die beiden Methoden eine Ausnahme. Ersetzen Sie diese Zeile einfach durch den Aufruf des Konstruktors Ihrer `GraphImpl` bzw. `GameMoveImpl` Klasse aus den vorherigen beiden Aufgaben. Dieser kleine Umweg ist notwendig, weil wir in unserer Vorlage nicht den Konstruktor einer noch nicht existierenden Klasse aufrufen können. Stattdessen nutzen wir die statischen Methoden aus der `Factory` Klasse. So ist sichergestellt, dass die unbearbeitete Vorlage keine Compilerfehler enthält.

Nachdem Sie die `Factory` Klasse vervollständigt haben, können Sie das Spiel über den Gradle Task `run` starten. Es öffnet sich erst ein Willkommensbildschirm auf dem Sie einige Einstellungen wie die Anzahl und Art der Spieler sowie die Größe des Graphen vornehmen können. Nach einem Klick auf “Start Game” startet das Spiel. Oben steht, welcher Spieler gerade am Zug ist. Klicken Sie auf einen Knoten, um dessen Farbe zu ändern. Klicken Sie auf eine Kante, um deren Gewichtung zu verändern – hierzu öffnet sich dann ein Dialogfenster. Wenn Sie eine `ForcedColorException` oder eine `NegativeWeightException` auslösen, dann erscheint eine entsprechende Fehlermeldung und Sie müssen einen anderen Spielzug ausführen.

Falls Sie von Ihrem Spiel so begeistert sind, dass Sie es auch unabhängig von IntelliJ oder Gradle starten wollen, dann führen Sie den Gradle Task `assemble` (in der Rubrik `build`) aus. Dadurch wird im Ordner `build/libs` eine Datei `GraphColoringGame.jar` erstellt. Diese Datei können Sie auf jedem System mit installierter Java Runtime Environment (JRE) ausführen – den Projektordner, Gradle oder IntelliJ brauchen Sie dazu dann nicht mehr.

[4] https://de.wikipedia.org/wiki/Delaunay-Triangulierung

Aufgabe 4 Computerspieler (freiwillig)

Im Paket `de.tukl.programmierpraktikum2020.p2.a4` finden Sie eine Schnittstelle `ComputerPlayer` sowie eine Implementierungsklasse `RandomComputerPlayer`. In der Methode `Factory.constructComputerPlayer` wird festgelegt, welche Implementierung verwendet werden soll. Wenn Sie also eine neue Implementierung erstellen, dann sollten Sie deren Konstruktor in der `Factory` Methode eintragen.

Die Schnittstelle sieht vor, dass zu Beginn des Spiels einmalig die `initialize` Methode aufgerufen wird. Als Parameter übergeben werden die Referenzen vom Typ `Graph` und `GameMove`, sowie die Farbe des eigenen Spielers und die Anzahl der am Spiel teilnehmenden Spieler. Die Farben der teilnehmenden Spieler sind `Color.values()[1]` (das ist `Color.RED`) bis `Color.values()[numPlayers]`. Die Spieler sind immer in dieser Reihenfolge am Zug. In der `initialize` Methode müssen Sie sich das was Sie von den übergebenen Daten benötigen als Objektattribute abspeichern. Außerdem können Sie vorbereitende Berechnungen durchführen, etwa Informationen zur Struktur des Graphen in geeigneter Form abspeichern.

Die `makeMove` Methode wird aufgerufen, wenn Ihr Spieler am Zug ist. Die Implementierung muss dann eine Methode des in `initialize` übergebenen `GameMove` Objektes aufrufen.

Es liegt in der Verantwortung der `ComputerPlayer` Implementierung, dass dieser sich an die Regeln hält. Er sollte zum Beispiel keine Änderungen direkt am `Graph` Objekt durchführen oder mehrere Spielzüge ausführen. Die Spiellogik selbst nimmt keine zusätzliche Überprüfung vor und vertraut dem Computerspieler.

Wir haben eine Verzögerung von einer Sekunde vor jedem Computerspieler-Spielzug eingebaut. Dies ist Absicht und dient dazu, dass man in der graphischen Oberfläche auch schnell hintereinander ausgeführte Spielzüge nachvollziehen kann, zum Beispiel wenn man mehrere Computerspieler gegeneinander antreten lässt. Sollten Sie diese Wartezeit ändern wollen, so werden Sie in der Methode `nextPlayer` in der Klasse `Game` aus dem Paket `de.tukl.programmierpraktikum2020.p2.main` fündig. Den Code der Klasse `Game` müssen Sie nicht ganz nachvollziehen können. Hauptproblem ist, dass wir es hier mit nebenläufiger Programmierung zu tun haben, weil die graphische Benutzeroberfläche ja nicht einfrieren soll, während der Computerspieler rechnet oder die Pause vor seinem Zug abwartet. Mehr dazu werden Sie in der Veranstaltung *Verteilte und nebenläufige Programmierung*[5] lernen.

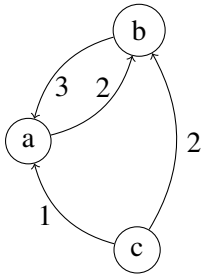
[5] https://www.cs.uni-kl.de/studium/lehrveranstaltungen/modulhb/#89-0203

Anhang: Beispiele für Spielabläufe

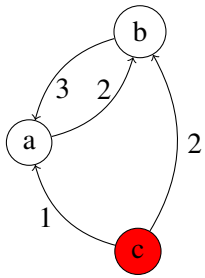
Die folgenden Beispiele sind zeilenweise von link nach rechts zu lesen. Oben links ist immer der Startzustand, rechts daneben (und ggf. in der nächsten Zeile fortgesetzt) sind die Spielzüge und der Zustand des Graphen nach dem jeweiligen Spielzug. Es werden immer einige Spielzüge vom Startzustand ausgehend gezeigt, das Spiel könnte selbstverständlich danach noch (theoretisch unendlich) fortgesetzt werden.

Beispiel 1 (2 Spieler, `setColor` und `increaseWeight`)

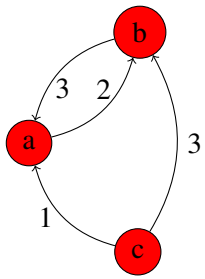
Startzustand



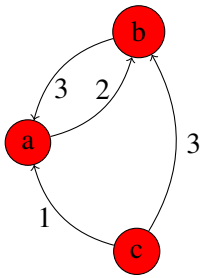
Spielzug von `RED`:
`setColor(c, Color.RED)`



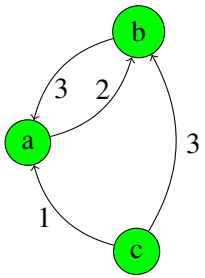
Spielzug von `GREEN`:
`increaseWeight(c, b)`



Spielzug von `RED`:
`setColor(c, Color.RED)`

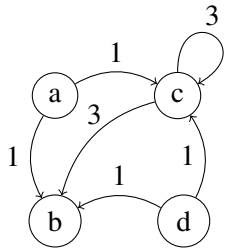


Spielzug von `GREEN`:
`setColor(c, Color.GREEN)`

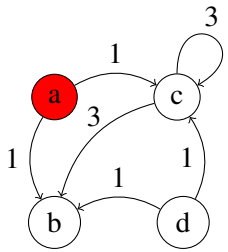


Beispiel 2 (2 Spieler, `setColor` und `decreaseWeight`)

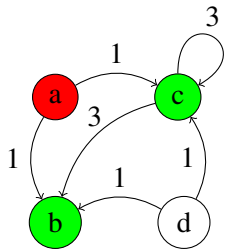
Startzustand



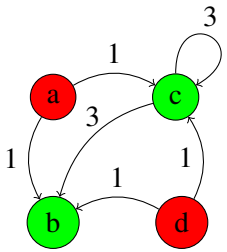
Spielzug von `RED`:
`setColor(a, Color.RED)`



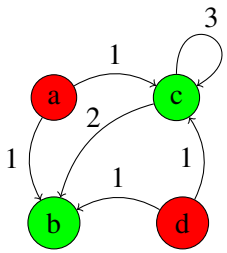
Spielzug von `GREEN`:
`setColor(c, Color.GREEN)`



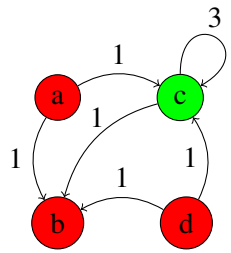
Spielzug von `RED`:
`setColor(d, Color.RED)`



Spielzug von `GREEN`:
`decreaseWeight(c, b)`

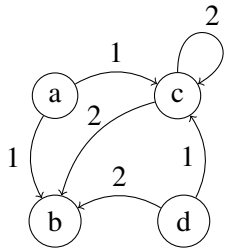


Spielzug von `RED`:
`decreaseWeight(c, b)`

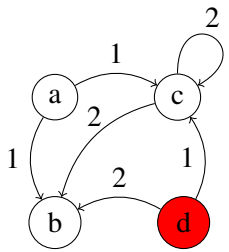


Beispiel 3 (4 Spieler, `setColor` und `increaseWeight`)

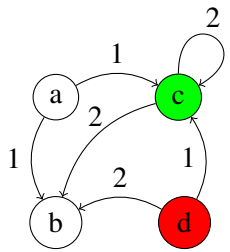
Startzustand



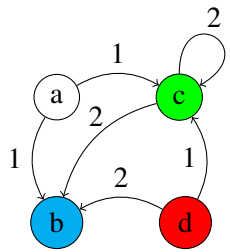
Spielzug von `RED`:
`setColor(d, Color.RED)`



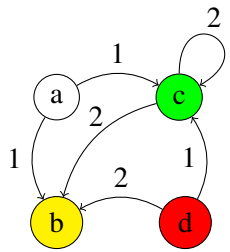
Spielzug von `GREEN`:
`setColor(c, Color.GREEN)`



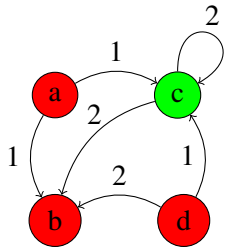
Spielzug von `BLUE`:
`setColor(b, Color.BLUE)`



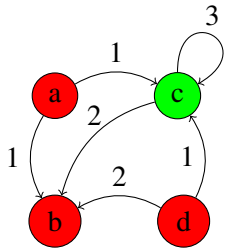
Spielzug von `YELLOW`:
`setColor(b, Color.YELLOW)`



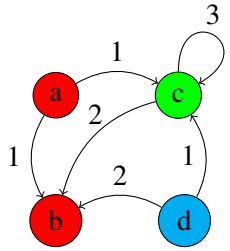
Spielzug von `RED`:
`setColor(a, Color.RED)`



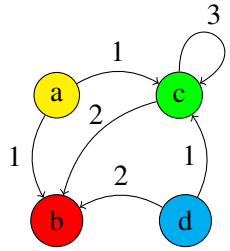
Spielzug von `GREEN`:
`increaseWeight(c, c)`



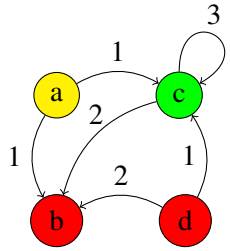
Spielzug von `BLUE`:
`setColor(d, Color.BLUE)`



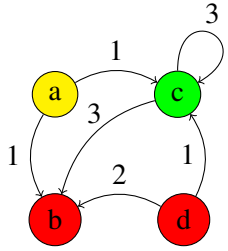
Spielzug von `YELLOW`:
`setColor(a, Color.YELLOW)`



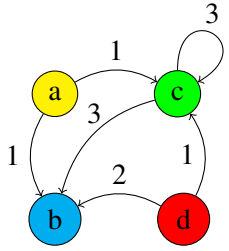
Spielzug von `RED`:
`setColor(d, Color.RED)`



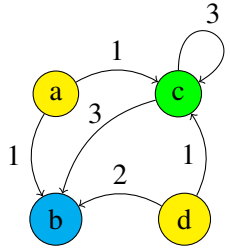
Spielzug von `GREEN`:
`increaseWeight(c, b)`



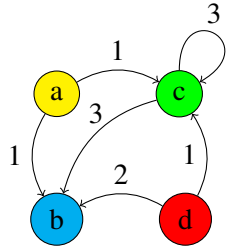
Spielzug von `BLUE`:
`setColor(b, Color.BLUE)`



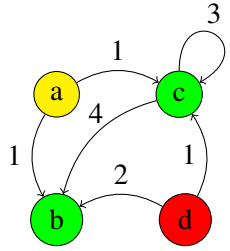
Spielzug von `YELLOW`:
`setColor(d, Color.YELLOW)`



Spielzug von `RED`:
`setColor(d, Color.RED)`

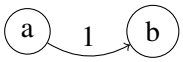


Spielzug von `GREEN`:
`increaseWeight(c, b)`

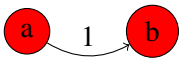


Beispiel 4 (2 Spieler, `setColor` und `ForcedColorException`)

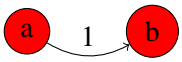
Startzustand



Spielzug von `RED`:
`setColor(a, Color.RED)`



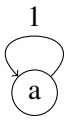
Spielzug von `GREEN`:
`setColor(b, Color.GREEN)`



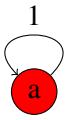
Es wird eine `ForcedColorException` geworfen.

Beispiel 5 (2 Spieler, `setColor` und `ForcedColorException`)

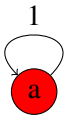
Startzustand



Spielzug von `RED`:
`setColor(a, Color.RED)`



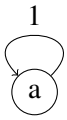
Spielzug von `GREEN`:
`setColor(a, Color.GREEN)`



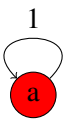
Es wird eine `ForcedColorException` geworfen.

Beispiel 6 (2 Spieler, `setColor` und `decreaseWeight`)

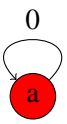
Startzustand



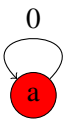
Spielzug von `RED`:
`setColor(a, RED)`



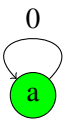
Spielzug von `GREEN`:
`decreaseWeight(a, a)`



Spielzug von `RED`:
`setColor(a, RED)`

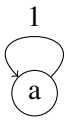


Spielzug von `GREEN`:
`setColor(a, GREEN)`

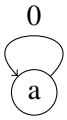


Beispiel 7 (2 Spieler, `decreaseWeight` und `NegativeWeightException`)

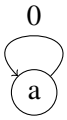
Startzustand



Spielzug von `RED`:
`decreaseWeight(a, a)`



Spielzug von `GREEN`:
`decreaseWeight(a, a)`



Es wird eine `NegativeWeightException` geworfen.