Aufgabe 5: Hüpfburg

Team-ID: 00919

Team-Name: Object Grind

Bearbeiter/-innen dieser Aufgabe:
Lennart Protte

November 20, 2022

Contents

1 Lösungsidee

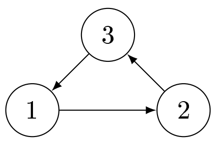
Der Aufgabenstellung zu entnehmen ist, dass sich beide Spieler immer genau gleichzeitig, um genau einen Punkt, entlang der Pfeilrichtung bewegen und der Parcours als absolviert gilt, wenn sich beide gleichzeitig auf den gleichen Punkt bewegen. Dabei ist die Startposition der Spieler jeweils immer identisch.

Als Datenstruktur für den Parcours eignet sich ein gerichteter und ungewichteter Graph. Die Pfeile zwischen den Kästchen werden durch gerichtete Kanten repräsentiert, während die Knoten repräsentativ für die Felder stehen.

Die abstrahierte Problemstellung der Aufgabe ist es, für jeden der zwei Ausgangspunkte, jeweils einen Weg zu einem Knoten, in einem gerichteten und ungewichteten Graphen, zu finden. Dieser muss von beiden Ausgangsknoten mit der gleichen Anzahl an Schritten zu erreichen sein. Dabei darf die Schrittfolge nie entgegengesetzt zur Kantenrichtung verlaufen.

Dies kann erreicht werden, indem für beide Spieler eine Breitensuche parallel schrittweise durchgeführt wird. Davon ausgehend, dass es dabei zu einem gemeinsamen Schnittpunkt kommt, wird die Breitensuche fortgesetzt, bis sich mindestens ein Knoten aus der Schnittmenge der beiden Suchen ergibt. Wenn es sich um einen lösbaren Parcours handelt, wird für beide Spieler im selben Schritt mindestens ein gleicher Knoten gefunden.

Sollte der Parcours nicht lösbar ist, ist die Menge der markierten Knoten, für beide Spieler zur gleichen Zeit, irgendwann gleich der Menge, der zu einem früheren Zeitpunkt markierten Knoten. So kommt es in Figur 1. zu einer zeitgleichen Wiederholung der Knotenmenge bei beiden Spielern.



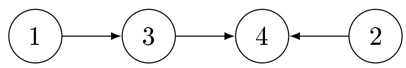
Figur 1: Nicht lösbarer zyklischer Parcours

Schritt	Spieler 1 Knotenmenge	Spieler 2 Knotenmenge
0.	{ 1 }	{ 2 }
1.	{ 2 }	{ 3 }
2.	{ 3 }	{ 1 }
3.	{ 1 }	{ 2 }

Tabelle 1: Schrittfolge für Figur 1.

In Schritt drei findet bei beiden Spielern eine Wiederholung zu Schritt null statt. Damit kann der Parcours sicher für ungültig erklärt werden und die Suche nach einem gemeinsamen Zielpunkt kann abgebrochen werden. Dies gilt, da sich beide Spieler in einem Zyklus befinden und im ersten Durchlauf des Zyklus kein gemeinsamer Punkt gefunden wurde.

Der Parcours ist ebenfalls nicht lösbar, sollte eine der beiden oder beide Knotenmengen leer sein. Dies gilt, da im nächsten Schritt keine weiteren Knoten für die leere Knotenmenge erreichbar wären und somit für einen der beiden Spieler kein Schritt mehr möglich ist. Da die Aufgabenstellung allerdings vorgibt, dass sich in jedem Schritt beide Spieler bewegen müssen, ist der Parcours damit für ungültig zu erklären. So kommt es in Figur 2. zu einer solchen Sackgasse für einen der Spieler.



Figur 2: Nicht lösbarer Parcours mit Sackgasse

Schritt	Spieler 1 Knotenmenge	Spieler 2 Knotenmenge
0.	{ 1 }	{ 2 }
1.	{ 3 }	{ 4 }
2.	{ 4 }	{ }

Tabelle 2: Schrittfolge für Figur 2.

Hier kommt es bereits im zweiten Schritt dazu, dass die Knotenmenge des zweiten Spielers leer ist. Damit ist es unmöglich, dass Spieler eins und Spieler zwei zum gleichen Zeitpunkt auf einen Knoten laufen können, da Spieler zwei keinen weiteren Schritt gehen kann und sie sich noch nicht auf dem gleichen Feld befinden.

Im Algorithmus muss daher in jedem Schritt für beide Spieler in der Breitensuche überprüft werden, ob die aktuelle Menge der besuchten Knoten in der Breitensuche bereits in der Schrittfolge des jeweiligen Spielers enthalten ist. Dazu müssen die Schrittfolgen beider Spieler gespeichert werden. Des weiteren muss überprüft werden ob beide Spieler erreichbare Knoten haben. Sollte dies nicht der Fall sein, kann dieser Spieler sich nicht weiter bewegen und sollte noch kein Zielknoten gefunden sein, ist der Parcours damit ungültig. Ist einer dieser Bedingungen erfüllt, ist der Parcours nicht lösbar und die Suche nach einem gemeinsamen Feld kann abgebrochen werden. Sollten sich die beiden Knotenmengen zu einem gemeinsamen Zeitpunkt an mindestens einem Knoten überschneiden, kann der Parcours in diesem Schritt an diesem Schnittknoten von beiden Spielern gleichzeitig erreicht werden und ist somit lösbar. Ist keine der Bedingungen erfüllt, folgt ein weiterer Schritt der Breitensuche.

Sobald der Zielpunkt gefunden ist, an dem sich die beide Schrittfolgen überschneiden, gilt es einen Weg von den Startknoten zu diesem Zielknoten zu finden. Gibt es mehrere Zielpunkte, kann ein beliebiger gewählt werden, da alle mit der gleichen Schrittanzahl erreicht werden können.

Aus dem ermittelten Zielknoten, den gegebenen Startknoten und der generierten Schrittfolge von jedem Startknoten zum Zielknoten, lässt sich nun der Weg für jeden der Startknoten bestimmen. Der letzte Eintrag in den Schrittfolgen beinhaltet den Zielknoten, während der erste Eintrag den jeweiligen Startknoten enthält.

Wie im Wegsuche Algorithmus in Zeile 4 dargestellt, kann zunächst im Weg der Zielknoten an letzter Stelle eingetragen werden. Es lässt sich nun in jeder Schrittfolge der vorletzte Schritt betrachten. In diesem Schritt kann nun ein Knoten ermittelt werden, welcher als Nachbar entlang der Kantenrichtung den Zielknoten besitzt (Zeile 8). Ist dieser Knoten gefunden, kann er im Weg an der entsprechenden Stelle gespeichert werden (Zeile 9). Dieses Verfahren wird nun solange für den nächsten Schritt und den zuletzt in den Weg eingefügten Knoten wiederholt, bis man am Startknoten angelangt ist.

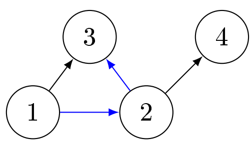
Algorithmus 1: Wegsuche Algorithmus

$A = Anzahl_{Schritte}$
$T = Zielknoten$
$Weg_{Knotenfolge}$
Füge T zu Weg an Stelle A hinzu
for $A > 0$ **do**
    $A = A - 1$
    for $K_{Knoten} = 1, 2, \ldots, N$ **do**
        if K *zeigt auf* T **then**
            füge K zu Weg an Stelle A hinzu
            $T = K$
        end if
    end for
end for

2 Umsetzung

Eine Menge von Knoten wird als `BitSet` dargestellt. Ein `BitSet` in Java ist eine lineare statische Datenstruktur, welche Bits, die alle anhand eines Indexes gesetzt werden können, speichert. Die Länge jedes `BitSet` ist gleich der Anzahl der Knoten des Graphen und jeder Index steht dabei für einen Knoten. Ist an diesen Index im `BitSet` eine eins, ist der Knoten in der Knotenmenge enthalten, andernfalls ist er es nicht. Beispielsweise gilt $0101 \widehat{=} \{2, 4\}$. Da der erste Index eines `BitSet` null ist, korrespondiert der Index null zu Knoten eins des Parcours.

Ein Graph wird als `BitSet[]` dargestellt. Jeder Index des Arrays korrespondiert zu einem Knoten. Das am Index eines Knoten enthaltene `BitSet`, ist eine Knotenmenge. Diese stellt da, zu welchen Knoten eine Kante ausgeht.



Figur 3: Graph der Matrix

So kann beispielsweise Figur 3. als das folgende `BitSet[]` dargestellt werden.

$$\texttt{BitSet[]}_{Graph} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

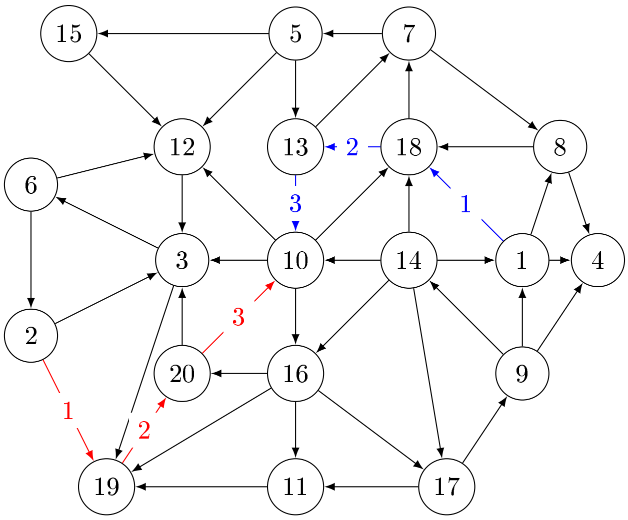
Die Schrittfolge ist eine `List<BitSet>`, d.h. eine `List` aus `BitSet`. Index null der `List` korrespondiert zum Ausgangspunkt, daher Schritt null beziehungsweise zur Startposition.

$$\texttt{List<BitSet>}_{Schrittfolge} = \left\{ \begin{array}{ll} \underline{\text{List Index:}} & \underline{\text{BitSet:}} \\ 0 & \{\ 1000\ \} \\ 1 & \{\ 0100\ \} \\ 2 & \{\ 0010\ \} \end{array} \right\}$$

Ein Weg ist ein `int[]`. Dabei repräsentiert der Index null den Ausgangspunkt des Weges. Aufgrund der Tatsache, dass im Graphen der Index null, dem Knoten eins entspricht, sind die Werte im Weg um eins niedriger eingetragen, als die tatsächlichen Knoten. Ist beispielsweise am Index zwei der Integer drei eingetragen, befindet sich an der dritten Position des Weges der Knoten vier. Dies wird in der Ausgabe berücksichtigt, wo zu jedem Wert eins addiert wird.

In der Implementation wurde der Algorithmus in der Methode `int[][] sameTargetRoute(BitSet[] graph)` umgesetzt. Dort werden die Schrittfolgen in einer `do-while Schleife` schrittweise erweitert und auf die in der Lösungsidee beschriebenen Bedingungen geprüft. Wenn in der Schleife ein Zielknoten gefunden wird, wird unter der verwendung der Hilfsmethode `int[] findSingleRouteInTimeline(List<BitSet> timeline, int target, BitSet[] graph)`, welche im Wegsuche Algorithmus beschrieben ist, der Weg zwischen den Startknoten und dem Zielknoten ermittelt und das Erbegnis anschließend zurückgegeben. Des weiteren wird in der Schleife die Hilfsmethode `BitSet neighbourNodes(BitSet nodes, BitSet[] graph)`, zur Ermittlung der Nachbarsknoten verwendet. Um auf die Wiederholungen der Schrittfolgen zu prüfen, wird außerdem die Methode `boolean timelineRepeats(List<BitSet> sashaTimeline, List<BitSet> mikaTimeline)` aufgerufen. Zur Ermittlung der Schnittmenge der zwei Schrittfolgen wird `int firstSameTargetOfTimeli sashaTimeline, List<BitSet> mikaTimeline)` benutzt.

3 Beispiele



Figur 4: huepfburg0.txt Parcours

Schritt	**Spieler 1 Knotenmenge**	**Spieler 2 Knotenmenge**
0.	{ 1 }	{ 2 }
1.	{ 4, 8, 18 }	{ 3, 19 }
2.	{ 4, 7, 13, 18 }	{ 6, 19, 20 }
3.	{ 5, 7, 10, 13 }	{ 2, 3, 10, 12, 20 }

Tabelle 3: Schrittfolge für Figur 4.

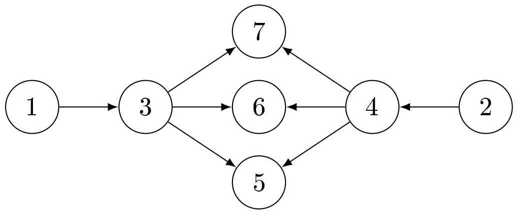
Bei Figur 4. handelt es sich um den Beispielparcours aus der Aufgabenstellung beziehungsweise um den Parcours aus huepfburg0.txt aus den Eingabedateien. Es handelt sich daher um einen lösbaren Parcours, in dem sich beide Spieler im dritten Schritt auf Feld zehn treffen. Wie Tabelle 3. zu entnehmen ist, ergibt sich im dritten Schritt der Knoten zehn aus der Schnittmenge der zwei Breitensuchen. In der Methode `int[][] sameTargetRoute(...)` liefert die Bedingung in der `do-while Schleife` in Schritt drei `false`. Daher wird im Anschluss der Zielknoten durch die Methode `int firstSameTargetOfTimelines(...)` gesetzt und die Wege mit `int[] findSingleRouteInTimeline(...)` gesetzt und anschließend wie folgt ausgegeben ausgegeben.

```
Ergebnis für huepfburg0.txt
Der Parcours hat folgende Lösung:
Zielfeld: 10
Anzahl an Schritten: 4
Sasha's Weg:
1 -> 18 -> 13 -> 10
Mika's Weg:
2 -> 19 -> 20 -> 10

```

Programmausgabe Figur 4.

In Figur 5. erreichen beide Spieler im gleichen Schritt die Knoten 5, 6, 7 . Wie in Figur 4. beschrieben, wird die `do-while Schleife` beendet. Die Methode `int firstSameTargetOfTimelines(...)` gibt hier



Figur 5: Parcours mit mehreren Zielknoten

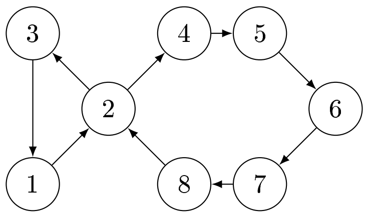
Schritt	Spieler 1 Knotenmenge	Spieler 2 Knotenmenge
0.	{ 1 }	{ 2 }
1.	{ 3 }	{ 4 }
2.	{ 5, 6, 7 }	{ 5, 6, 7 }

Tabelle 4: Schrittfolge für Figur 5.

Knoten fünf zurück, da `nextSetBit(...)` den Knoten mit dem kleinsten Bezeichner gibt. Für die Lösung ist es unerheblich, ob ein oder mehrere Zielknoten gefunden werden, da alle Zielknoten im gleichen Schritt erreicht werden. Daher wird eine Lösung für den Parcours ausgegeben.

```
Ergebnis für Figur5.txt
Der Parcours hat folgende Lösung:
Zielfeld: 5
Anzahl an Schritten: 3
Sasha's Weg:
1 -> 3 -> 5
Mika's Weg:
2 -> 4 -> 5
```

Programmausgabe Figur 5.



Figur 6: Unendlicher Parcours

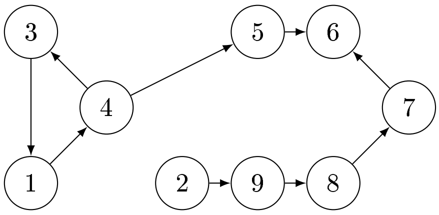
Schritt	Spieler 1 Knotenmenge	Spieler 2 Knotenmenge
0.	{ 1 }	{ 2 }
1.	{ 2 }	{ 3, 4 }
2.	{ 3, 4 }	{ 1, 5 }
3.	{ 1, 5 }	{ 2, 6 }
4.	{ 2, 6 }	{ 3, 4, 7 }
5.	{ 3, 4, 7 }	{ 1, 5, 8 }
6.	{ 1, 5, 8 }	{ 2, 6 }
7.	{ 2, 6 }	{ 3, 4, 7 }

Tabelle 5: Schrittfolge für Figur 6.

Bei Figur 6. handelt es sich um einen Parcours ohne Lösung. Dies ist erkennbar, da beide Spieler sich zeitgleich in ihrer Schrittfolge mit einem vorherigem Schritt wiederholen. Relevant ist hierbei die Abbruchbedingung der Breitensuche in `int[][] sameTargetRoute(...)`. Ohne diese würde es zu einer Endlosschleife kommen. In der `if Abfrage` in der Schleife, liefert der Aufruf der Methode `boolean timelineRepeats(...)` ein `true` in Schritt sieben. Die Knotenmenge $\{2, 6\}$ von Spieler eins, wiederholt sich hier mit Schritt vier, während sich zeitgleich die Knotenmenge $\{3, 4, 7\}$ bei Spieler zwei, ebenfalls mit Schritt vier wiederholt. Daher wird hier `null` zurückgegeben und der Algorithmus beendet.

```
Ergebnis für Figur6.txt
Der Parcours hat keine Lösung!
```

Programmausgabe Figur 6.



Figur 7: Sackgasse Parcours

Schritt	**Spieler 1 Knotenmenge**	**Spieler 2 Knotenmenge**
0.	{ 1 }	{ 2 }
1.	{ 4 }	{ 9 }
2.	{ 3, 5 }	{ 8 }
3.	{ 1, 6 }	{ 7 }
4.	{ 4 }	{ 6 }
5.	{ 3, 5 }	{ }

Tabelle 6: Schrittfolge für Figur 7.

Der Graph von Figur 7. führt in eine Sackgasse. Nach Schritt fünf ist es Spieler zwei nicht möglich weiter zu laufen, da es keine erreichbaren Knoten gibt. Die Aufgabenstellung erfordert, dass sich in jedem Schritt beide Spieler bewegen müssen. Da sich beide Spieler nicht auf dem gleichen knoten befinden (daher der Parcours noch nicht gelößt ist), kann er für ungültig erklärt werden. In `int[][] sameTargetRoute(...)` wird in der `if` Abfrage, durch `sashaTimeline.get(sashaTimeline.size() - 1).isEmpty() || mikaTimeline .get(mikaTimeline.size() - 1) .isEmpty()` gefrüft, ob einer der beiden Spieler keine erreichbaren Knoten mehr hat. Hier ist diese Bedingung `true` womit `null` zurückgegeben wird.

```
Ergebnis für Figur7.txt
Der Parcours hat keine Lösung!
```

Programmausgabe Figur 7.

4 Quellcode

```
/**
 * Generiert den Graphen aus den Zeilen einer Eingabedatei.
 *
 * @param lines Eine String Liste, wo jeder String eine Zeile im Eingabeformat ist.
 * @return Den generierten Graphen.
 */
private static BitSet[] graphFromLines(List<String> lines) {
    int countOfNodes = Integer.parseInt(lines.get(0).split("␣")[0]);
    BitSet[] graph = new BitSet[countOfNodes];
    for (int i = 0; i < countOfNodes; i++) {
        graph[i] = new BitSet(countOfNodes);
    }
    List<String> rawArrows = lines.subList(1, lines.size());
    for (String rawArrow : rawArrows) {
        int arrowBegin = Integer.parseInt(rawArrow.split("␣")[0]);
        int arrowEnd = Integer.parseInt(rawArrow.split("␣")[1]);
        graph[arrowBegin - 1].set(arrowEnd - 1);
    }
    return graph;
```

```
    }
```

Methode graphFromLines

```
\label{lst:sameTargetRoute}
/**
 * Findet eine Route, vom nullten und ersten Knoten,
 * zu einem gemeinsamen Knoten in einer gleichen Anzahl von Schritten.
 *
 * @param graph Der Graph, zu dem eine Route vom nullten und ersten Knoten zu einem
gemeinsamen Knoten gebildet werden soll.
 * @return Eine Route zu einem Knoten der vom 0. und 1. Knoten in der gleichen Anzahl
 von Schritten erreichbar ist.
 */
private static int[][] sameTargetRoute(BitSet[] graph) {
    BitSet sashaFirst = new BitSet(graph.length);
    BitSet mikaFirst = new BitSet(graph.length);
    sashaFirst.set(0);
    mikaFirst.set(1);
    List<BitSet> sashaTimeline = new ArrayList<>(List.of(sashaFirst));
    List<BitSet> mikaTimeline = new ArrayList<>(List.of(mikaFirst));
    do {
        sashaTimeline.add(neighbourNodes(sashaTimeline.get(sashaTimeline.size() - 1),
 graph));
        mikaTimeline.add(neighbourNodes(mikaTimeline.get(mikaTimeline.size() - 1),
graph));
        if (sashaTimeline.get(sashaTimeline.size() - 1).isEmpty() || mikaTimeline
                .get(mikaTimeline.size() - 1)
                .isEmpty() || timelineRepeats(sashaTimeline, mikaTimeline)) {
            return null;
        }
    } while (!sashaTimeline.get(sashaTimeline.size() - 1).intersects(mikaTimeline.get
(mikaTimeline.size() - 1)));

    int target = firstSameTargetOfTimelines(sashaTimeline, mikaTimeline);
    int[][] routes = new int[2][];
    routes[0] = findSingleRouteInTimeline(sashaTimeline, target, graph);
    routes[1] = findSingleRouteInTimeline(mikaTimeline, target, graph);
    return routes;
}
```

Methode sameTargetRoute

```
\label{lst:firstSameTargetOfTimelines}
/**
 * Vergleicht den jeweils letzten Schritt der Schrittfolgen miteinander
 * und gibt den gemeinsamen besuchten Knoten zurück.
 *
 * @param sashaTimeline Sasha's Zeitleiste
 * @param mikaTimeline  Mika's Zeitleiste
 * @return Der Index im Graphen des gemeinsamen Knotens.
 */
private static int firstSameTargetOfTimelines(List<BitSet> sashaTimeline, List<BitSet
> mikaTimeline) {
    BitSet targets = (BitSet) sashaTimeline.get(sashaTimeline.size() - 1).clone();
    targets.and(mikaTimeline.get(mikaTimeline.size() - 1));
    return targets.nextSetBit(0);
}
```

Methode firstSameTargetOfTimelines

```
\label{lst:findSingleRouteInTimeline}
/**
 * Ermittelt den Weg vom Zielpunkt zurück zum Startpunkt entgegengesetzt der
Kantenrichtung des Graphen.
 *
 * @param timeline Die Zeitleiste der erreichbaren Knoten,
 *                 von der eine Route zu einem bestimmten Knoten im letzten Zeitpunkt
 der Zeitleiste gebaut werden soll.
```

```
     * @param target   Das Ziel am Ende der Zeitleiste der erreichbaren Knoten, zudem
    eine Route gebaut werden soll.
     * @param graph    Der Graph aus dem die Zeitleiste der erreichbaren Knoten (und
    dementsprechend auch das Ziel) stammt.
     * @return Eine Route zum Ziel in der Zeitleiste.
     */
    private static int[] findSingleRouteInTimeline(List<BitSet> timeline, int target,
    BitSet[] graph) {
        int[] route = new int[timeline.size()];
        steps:
        for (int currentStep = timeline.size() - 1; currentStep > 0; currentStep--) {
            route[currentStep] = target;
            BitSet currentNodes = timeline.get(currentStep - 1);
            for (int i = 0; i < graph.length; i++) {
                if (currentNodes.get(i)) {
                    BitSet arrows = graph[i];
                    if (arrows.get(target)) {
                        target = i;
                        continue steps;
                    }
                }
            }
        }
        route[0] = target;
        return route;
    }
```

Methode findSingleRouteInTimeline

```
    /**
     * Gibt die Knoten zurück, welche in einem Schritt entlang der Kantenrichtung,
     * von den aktuellen Knoten erreichbar sind.
     *
     * @param nodes Die aktuellen Knoten
     * @param graph Der Graph, welcher betrachtet wird
     * @return Die Menge der Knoten die in einem Schritt von den gegebenen Knoten
    erreichbar ist
     */
    private static BitSet neighbourNodes(BitSet nodes, BitSet[] graph) {
        BitSet neighbours = new BitSet();
        for (int i = 0; i < graph.length; i++) {
            if (nodes.get(i)) {
                neighbours.or(graph[i]);
            }
        }
        return neighbours;
    }
```

Methode neighbourNodes

```
    \label{lst:timelineRepeats}
    /**
     * Ermittelt, ob die zwei Zeitleisten sich zu einem Zeitpunkt mit ihrem letzten
    Eintrag wiederholen.
     * Es wird daher festgestellt, ob beide Zeitleisten zu einem vorherigen Zeitpunkt an
    den genau gleichen Knoten waren,
     * wie zum aktuellen Zeitpunkt.
     *
     * @param sashaTimeline Sasha's Zeitleiste der erreichbaren Knoten
     * @param mikaTimeline  Mika's Zeitleiste der erreichbaren Knoten
     * @return True, wenn die Zeitleisten sich beide zu einem gleichen Zeitpunkt
     * mit dem neuen/letzten Zeitpunkt wiederholen.
     */
    private static boolean timelineRepeats(List<BitSet> sashaTimeline, List<BitSet>
    mikaTimeline) {
        BitSet current = sashaTimeline.get(sashaTimeline.size() - 1);
        BitSet repetitions = new BitSet(sashaTimeline.size());
        List<BitSet> timeline = new ArrayList<>(sashaTimeline.subList(0, sashaTimeline.
    size() - 1));
        for (int i = 0; i < timeline.size(); i++) {
            if (timeline.get(i).equals(current)) {
```

```java
                repetitions.set(i);
            }
        }
        for (int i = repetitions.nextSetBit(0); i >= 0; i = repetitions.nextSetBit(i + 1)
    ) {
            if (mikaTimeline.get(mikaTimeline.size() - 1).equals(mikaTimeline.get(i))) {
                return true;
            }
        }
        return false;
    }
```

Methode timelineRepeats