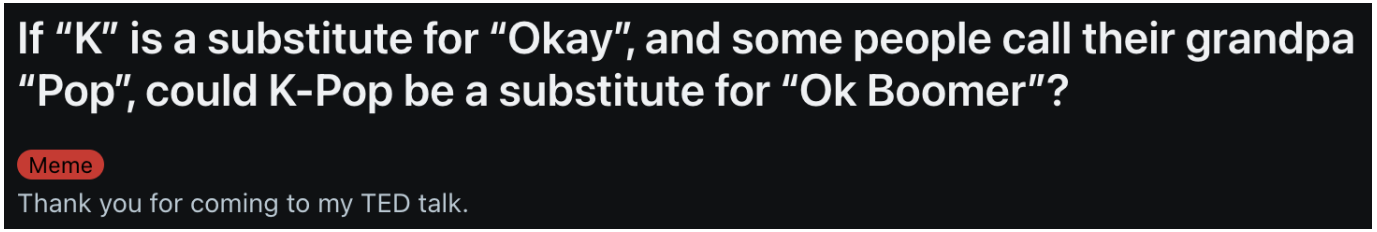


252-0027

Einführung in die Programmierung

Übungen

Woche 9: Klassen, Verlinkte Objekte

Timo Baumberger
Departement Informatik
ETH Zürich

Organisatorisches

- Mein Name: Timo Baumberger
- Website: timobaumberger.com
- Bei Fragen: tbaumberger@student.ethz.ch
  - Mails bitte mit «[EProg24]» im Betreff
- Neue Aufgaben: **Dienstag Abend** (im Normalfall)
- Abgabe der Übungen bis **Dienstag Abend (23:59)** Folgewoche
  - Abgabe immer via Git
  - Lösungen in separatem Projekt auf Git

Programm

- Bonusaufgabe u07
- Dynamische Listen (LinkedList, ArrayList)
- Wrapper Klassen
- Probleme lösen
- Enums
- Vorbesprechung
- Nachbesprechung
- Kahoot

Bonusaufgabe u07

```java
// we may assume that the board is rectangular
public static void revealAllMines(int[][] board) {
    int rows = board.length;
    int columns = board[0].length;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            if (board[i][j] == -2) {
                board[i][j] = 9;
            }
        }
    }
}
```

```java
public static int computeMineCount(int[][] board, int row, int col) {
    if (board[row][col] == -2) {
        return 9;
    }

    int count = 0;
    for (int d = -1; d <= 1; d++) {
        for (int e = -1; e <= 1; e++) {
            if (!isCellAccessible(board, row+d, col+e)) {
                continue;
            }
            int cell = board[row+d][col+e];
            // Since we assume that the game hasn't ended we can safely
            // assume that no mine was uncovered at this time
            if (cell == -2) {
                count++;
            }
        }
    }

    return count;
}

private static boolean isCellAccessible(int[][] board, int row, int col) {
    int rows = board.length;
    int columns = board[0].length;
    boolean validRow = 0 <= row && row < rows;
    boolean validColumn = 0 <= col && col < columns;

    return validRow && validColumn;
}
```

```java
public static void gameTurn(int[][] board, int row, int col) {
    boolean alreadyUncovered = 0 <= board[row][col] && board[row][col] <= 8;
    if (alreadyUncovered) {
        return;
    }
    boolean isMineCell = board[row][col] == -2;
    if (isMineCell) {
        revealAllMines(board);
        return;
    }

    int mineCount = computeMineCount(board, row, col);
    board[row][col] = mineCount;
    if (mineCount != 0) {
        return;
    }
    for (int d = -1; d <= 1; d++) {
        for (int e = -1; e <= 1; e++) {
            if ((d == 0 && e == 0) || !isCellAccessible(board, row+d, col+e)) {
                continue;
            }

            gameTurn(board, row+d, col+e);
        }
    }
}
```

Tipps für (Bonus-)Aufgaben

- Synergien aus vorherigen Teilaufgaben nutzen
- Aufgabe analysieren (so wie Woyzeck von Georg Büchner)

Dynamische Listen

- **Dynamisch weil die Grösse nicht festgelegt ist**
- **Wichtige dynamische Listen in Java**
  - LinkedList (verkettete Liste, List Nodes sind verlinkt)
  - ArrayList (verwendet Array im Hintergrund)
  - Vector (ähnlich wie ArrayList, aber veraltet)
  - CopyOnWriteArrayList

Enhanced For Loop und dynamische Listen

```java
List<Integer> ints = new ArrayList<>();
ints.add(4);
ints.add(2);
for (int i : ints) {
    ints.add(2);
    System.out.println(i);
}
```

```java
List<Integer> ints = new CopyOnWriteArrayList<>();
ints.add(4);
ints.add(2);
for (int i : ints) {
    ints.add(2);
    System.out.println(i);
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException Create breakpoint
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1095)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:1049)
    at Counting.main(Counting.java:15)
```

Iterator

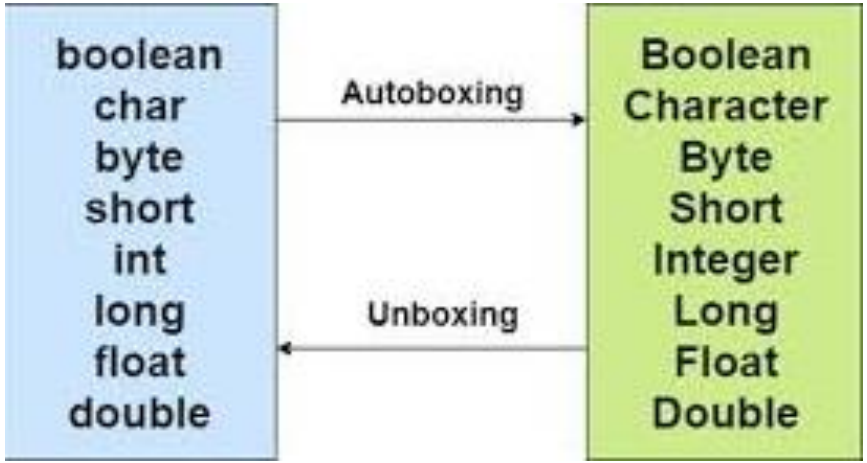
```
List<Integer> ints = new ArrayList<>();
ints.add(4);
ints.add(2);
for (ListIterator<Integer> it = ints.listIterator(); it.hasNext();) {
    int currentIndex = it.nextIndex();
    Integer element = it.next(); // returns current elements and advances position
    it.set(5); // replaces last element returned by next()
    it.remove(); // replaces last element returned by next()
    if (currentIndex == 0) {
        // ListIterator.add() is a lazy add
        it.add(3); // adds new element after last element returned by next()
    }
}

for (int i : ints) {
    System.out.println(i);
}
```

Output: 3

Wrapper Klassen

- List<int> nicht möglich
- Wrapper Klassen wurden eingeführt als Type Parameter eingeführt wurden
- List<T> ist List<Object>
- T ist ein Type Parameter



Probleme Lösen

Probleme Lösen: Labyrinth

Aufgabe 1: Labyrinth (2021 W8)

Ein Labyrinth besteht aus einer Menge von Räumen, welche durch die Klasse `Room` dargestellt werden. Die Klasse hat zwei Attribute: Der Integer `age` (grösser gleich 0) beschreibt das Alter des Raums und der Array `doorsTo` (nie `null`) beschreibt die Türen von diesem Raum zu anderen Räumen. Alle Türen sind Falltüren, d.h. sie funktionieren nur in eine Richtung. Ein Raum ist ein Ausgang aus dem Labyrinth, wenn keine Türen von dem Raum wegführen, das heisst, wenn `doorsTo` eine Länge von 0 hat.

Für alle Aufgaben werden Sie in einen zufälligen Raum geworfen, welcher als Argument gegeben wird (garantiert nicht `null`) und von welchem aus Sie die Aufgabe lösen müssen. Sie dürfen für alle Aufgaben annehmen, dass es im Labyrinth keinen Zyklus gibt. Das heisst, dass man einen Raum, welchen man durch eine Türe verlassen hat, nie wieder erreichen kann indem man weiteren Türen folgt. Eine Sequenz von N Räumen $r_1, \ldots, r_N$ ist ein *Lösungspfad* für einen Raum `room` genau dann wenn: (1) Der erste Raum r_1 ist der Raum `room`, (2) der letzte Raum r_N ist ein Ausgang, und (3) jeder Raum r_i mit $1 \leq i < N$ hat eine Tür zum nächsten Raum in der Sequenz r_{i+1}.

Probleme Lösen: Labyrinth

Aufgabe 1: Labyrinth (2021 W8)

Ein Labyrinth besteht aus einer Menge von Räumen, welche durch die Klasse `Room` dargestellt werden. Die Klasse hat zwei Attribute: Der Integer `age` (grösser gleich 0) beschreibt das Alter des Raums und der Array `doorsTo` (nie `null`) beschreibt die Türen von diesem Raum zu anderen Räumen. Alle Türen sind Falltüren, d.h. sie funktionieren nur in eine Richtung. Ein Raum ist ein Ausgang aus dem Labyrinth, wenn keine Türen von dem Raum wegführen, das heisst, wenn `doorsTo` eine Länge von 0 hat.

Für alle Aufgaben werden Sie in einen zufälligen Raum geworfen, welcher als Argument gegeben wird (garantiert nicht `null`) und von welchem aus Sie die Aufgabe lösen müssen. Sie dürfen für alle Aufgaben annehmen, dass es im Labyrinth keinen Zyklus gibt. Das heisst, dass man einen Raum, welchen man durch eine Türe verlassen hat, nie wieder erreichen kann indem man weiteren Türen folgt. Eine Sequenz von N Räumen $r_1, \ldots, r_N$ ist ein *Lösungspfad* für einen Raum `room` genau dann wenn: (1) Der erste Raum r_1 ist der Raum `room`, (2) der letzte Raum r_N ist ein Ausgang, und (3) jeder Raum r_i mit $1 \leq i < N$ hat eine Tür zum nächsten Raum in der Sequenz r_{i+1}.

Probleme Lösen: Labyrinth

Ausgang wenn:

- `r.doorsTo.length == 0`

Lösungspfad $r_1, \ldots, r_N$ wenn:

- r_1 ist `room` (der Raum der uns gegeben wird)
- r_N ist ein Ausgang
- r_i und r_{i+1} sind jeweils durch eine Tür verbunden.

man weiteren Türen folgt. Eine Sequenz von N Räumen $r_1, \ldots, r_N$ ist ein *Lösungspfad* für einen Raum `room` genau dann wenn: (1) Der erste Raum r_1 ist der Raum `room`, (2) der letzte Raum r_N ist ein Ausgang, und (3) jeder Raum r_i mit $1 \leq i < N$ hat eine Tür zum nächsten Raum in der Sequenz r_{i+1}.

Probleme Lösen: Labyrinth

1. Implementieren Sie die Methode `Labyrinth.task1(Room room)`. Die Methode soll true zurückgeben genau dann, wenn es einen Lösungspfad $r_1, \ldots, r_N$ für `room` gibt, sodass:
   - Für jede Teilsequenz $r_1, ..., r_i$ mit $1 \leq i \leq N$ gilt, dass die Summe der Alter der Räume $r_1, ..., r_i$ nicht durch 3 teilbar ist.
2. Implementieren Sie die Methode `Labyrinth.task2(Room room)`. Die Methode soll true zurückgeben genau dann, wenn es zwei Lösungspfade $r_1, \ldots, r_N$ und $s_1, \ldots, s_N$ für `room` gibt, sodass:
   - Die Räume r_i und s_i haben das gleiche Alter für jedes i mit $1 \leq i \leq N$.
   - Für mindestens ein i mit $1 \leq i \leq N$ gilt, dass r_i und s_i unterschiedlich sind (verschiedene Referenzen).

Wichtig!

Sie dürfen Methoden und Felder der Klasse `Room` hinzufügen. Tests finden Sie in der Datei "LabyrinthTest.java". **Tipp:** Lösen Sie die Aufgaben rekursiv. Für keine der Aufgaben müssen Sie alle Pfade generieren und dann erst prüfen, dass die Eigenschaften gelten. Manche der Tests enthalten Labyrinthe mit einer extrem grossen Anzahl an Pfaden aber leichten Lösungen.

```java
public class Room {
    int age;
    public Room[] doorsTo;

    public Room(int age, Room[] doorsTo) {
        this.age = age;
        this.doorsTo = doorsTo;
    }

    public boolean isExit() {
        return doorsTo.length == 0;
    }

    public int getAge() {
        return age;
    }
}
```

Was befindet sich in der `Room` Klasse?

- `age`**-Attribut (grosser gleich 0)**
- `doorsTo`**-Attribut (nie** `null`**)**

- `isExit()`**-Methode:** Prüft ob ein Raum ein Ausgang ist.

- `getAge()`**-Methode:** Getter-Methode für die das `age` Attribut. Üblicherweise wären Attribute einer Klasse `private` und nur über Getter- / Setter-Methoden erreichbar. Hier der Einfachheit halber weggelassen.

```java
public class Labyrinth {

    public static boolean task1(Room room)
{
        // TODO
        return false;
    }

    public static boolean task2(Room room)
{
        // TODO
        return false;
    }
}
```

Was befindet sich in der `Labyrinth`-Klasse?

- Code-Skeleton für Aufgabe 1 und Aufgabe 2.

Probleme Lösen: Labyrinth

1. Implementieren Sie die Methode `Labyrinth.task1(Room room)`. Die Methode soll true zurückgeben genau dann, wenn es einen Lösungspfad $r_1, \ldots, r_N$ für `room` gibt, sodass:
   - Für jede Teilsequenz $r_1, ..., r_i$ mit $1 \leq i \leq N$ gilt, dass die Summe der Alter der Räume $r_1, ..., r_i$ nicht durch 3 teilbar ist.

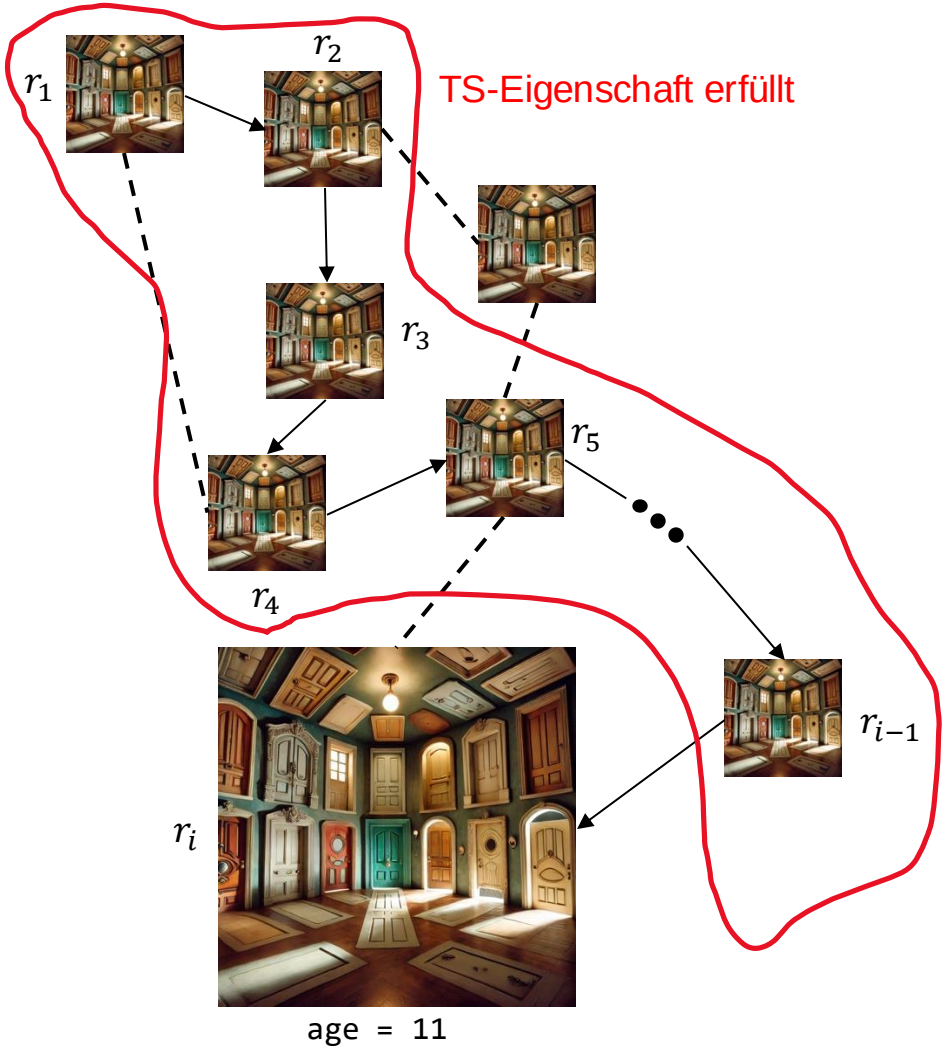
Wie lösen wir das Problem?

- **Rekursive Lösung:** Damit wir das Problem rekursiv lösen können, müssen wir **Teilprobleme** identifizieren.

Teilprobleme identifizieren:

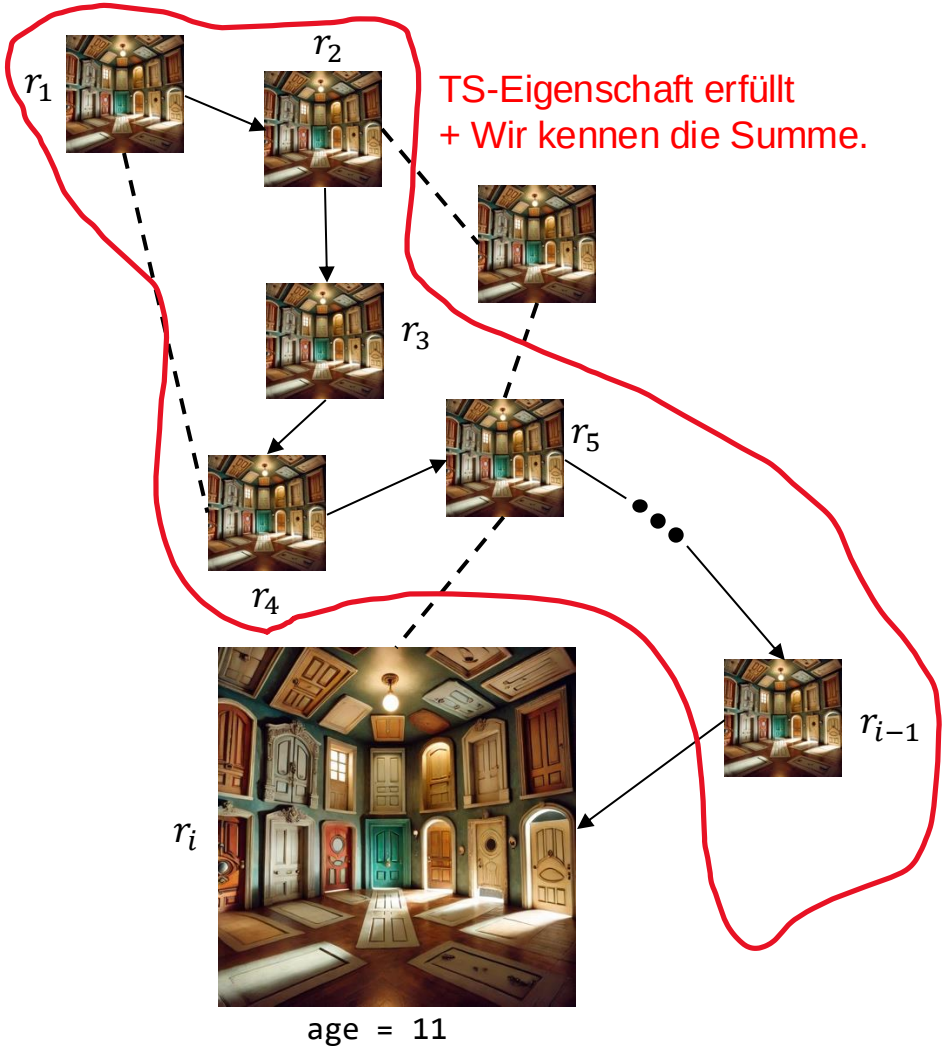
- Angenommen wir befinden uns in einem Raum r_i wie wissen wir ob der Pfad $r_1, \dots, r_i$ eine Lösung ist?

- Wir nennen die Eigenschaft einer Summe **keine** Teilsequenz zu besitzen, deren Alterssumme ein Vielfaches von drei ist ab hier die **TS-Eigenschaft**.



Teilprobleme identifizieren (Versuch 1):

- Angenommen wir befinden uns in einem Raum r_i wie wissen wir ob der Pfad $r_1, \ldots, r_i$ die TS-Eigenschaft erfüllt?
- Was wenn $r_1, \ldots, r_{i-1}$ bereits die TS-Eigenschaft erfüllt?
- Das reicht nicht. Wieso?
- Falls die Summe der Alter vorher 22 ist, dann 22 % 3 != 0 aber (22 + 11) % 3 == 0.



Teilprobleme identifizieren (Versuch 1):

- Angenommen wir befinden uns in einem Raum r_i wie wissen wir ob der Pfad $r_1, \ldots, r_i$ die TS-Eigenschaft erfüllt?
- Was wenn $r_1, \ldots, r_{i-1}$ die Alterssumme `sum` hat und `sum % 3 != 0` ist?
- Das reicht. Wieso?
- Wir prüfen ob `(sum + age) % 3 != 0` ist und dann Wissen wir das $r_1, \ldots, r_i$ ebenfalls die TS-Eigenschaft erfüllt.

```java
public static boolean solve1(Room room, int sum) {
    sum = sum + room.age;

    if(sum % 3 == 0) {
        return false;
    }

    if(room.isExit()) {
        return true;
    }
    for(int i = 0; i < room.doorsTo.length; ++i) {
        if(solve(room.doorsTo[i], sum)) {
            return true;
        }
    }
    return false;
}
```

Wie lösen wir das Problem?

- Wir nehmen an, dass `sum` die Summe der vorherigen Räume enthält.
- Wir nehmen an, dass die vorherigen Räume die TS-Eigenschaft erfüllen.
- Dann erhöhen wir `sum` um das Alter von `room` und prüfen, ob die neue Summe **nicht** durch 3 teilbar ist. (Sonst beenden wir die Suche auf dem jetzigen Pfad)
- Wir prüfen ob der jetzige Raum ein Ausgang ist. (Wenn ja, dann sind wir fertig.)
- Sonst rufen wir die Methode rekursiv für alle Räume auf, mit denen `Room` verbunden ist. (Das dürfen wir, da es keine Zyklen hat)
- Falls einer der Aufrufe erfolgreich war, dann geben wir `true`, sonst `false` zurück.

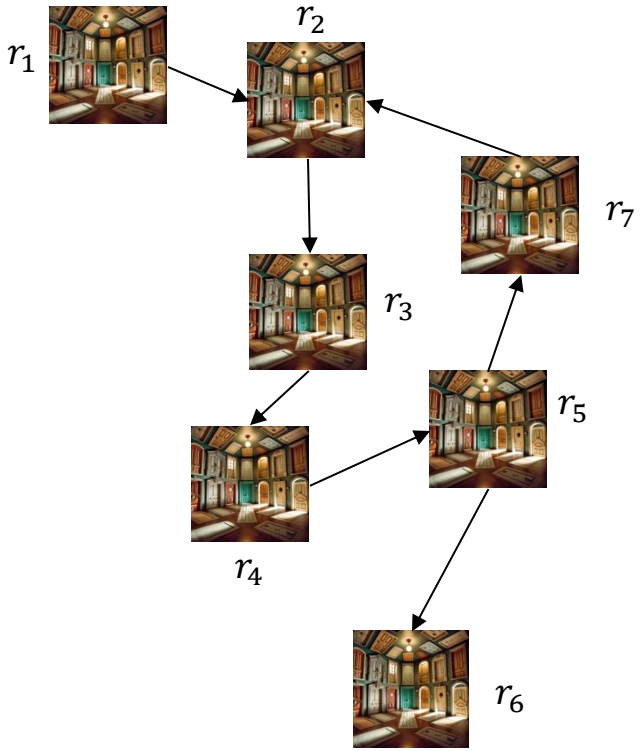
```java
public static boolean solve1(Room room, int sum) {
(…)
}

public static boolean task1(Room room) {
    return solve(room, 0)

}
```

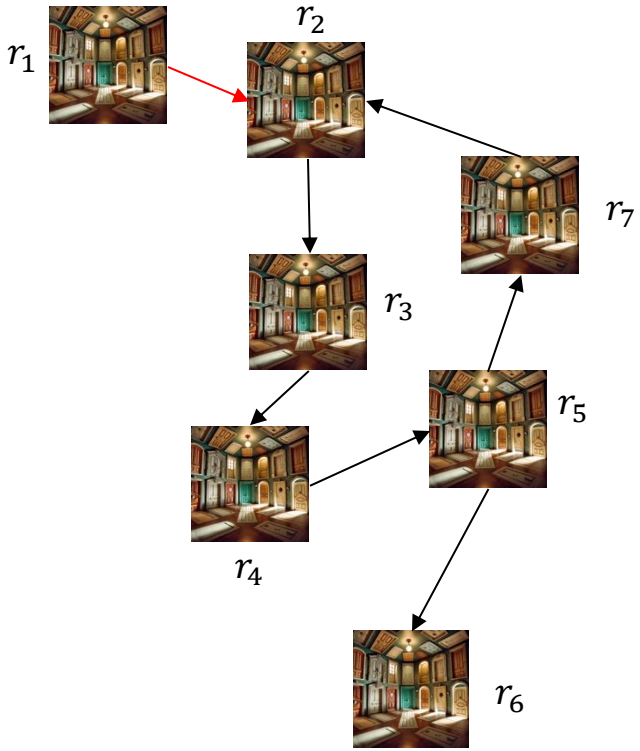
Wie nutzen wir diese Methode nun?

- Wir rufen `solve1` in `task1` auf mit `room` und initialer Summe `0`.



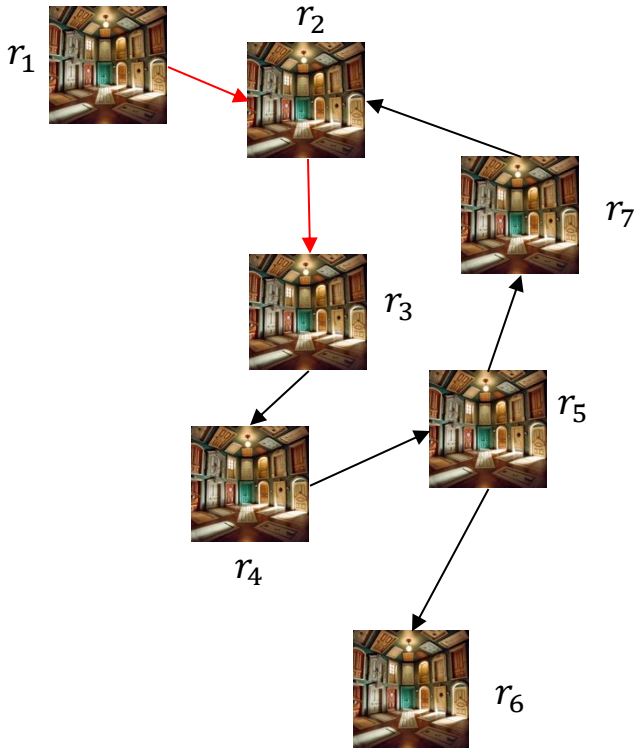
Wieso keine Zyklen?

- Wenn wir Zyklen haben kommt es zu **endloser Rekursion**, ausser wir merken uns explizit, in welchen wir Räumen wir bereits waren.



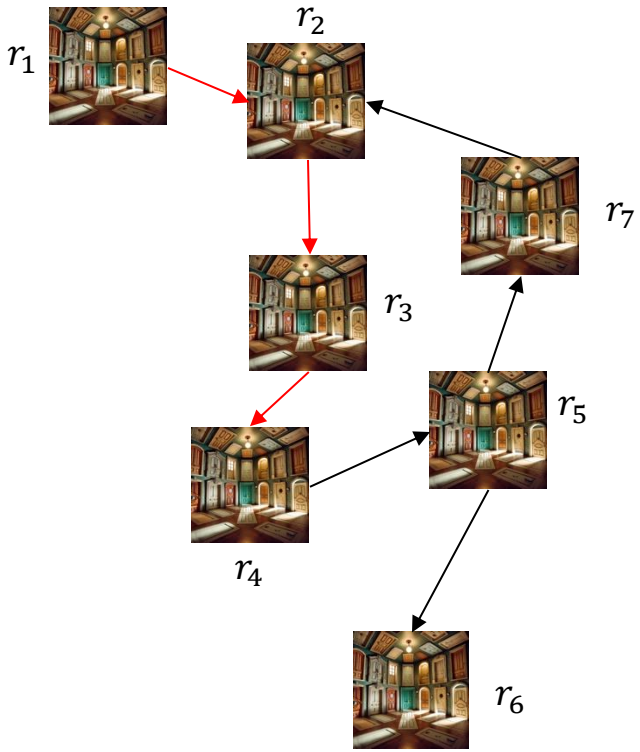
Wieso keine Zyklen?

- Wenn wir Zyklen haben kommt es zu **endloser Rekursion** ohne, dass wir uns merken, in welchen wir Räumen wir bereits waren.



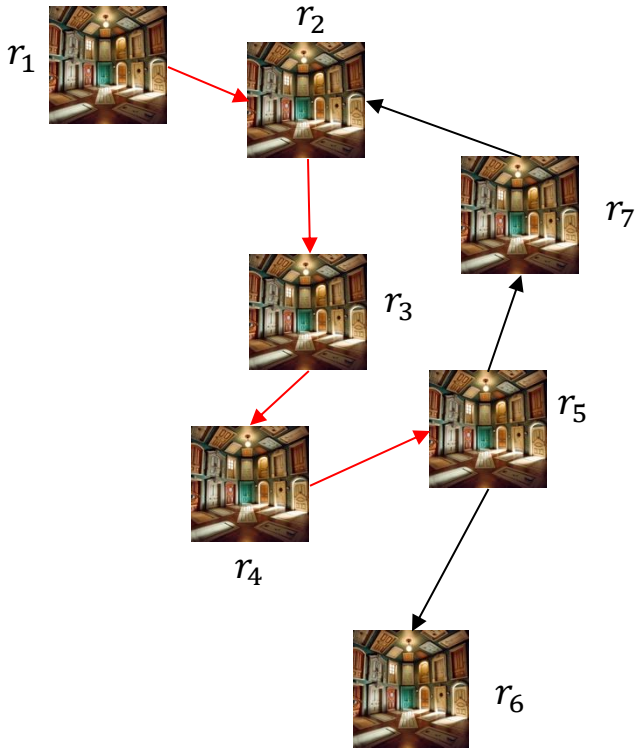
Wieso keine Zyklen?

- Wenn wir Zyklen haben kommt es zu **endloser Rekursion** ohne, dass wir uns merken, in welchen wir Räumen wir bereits waren.



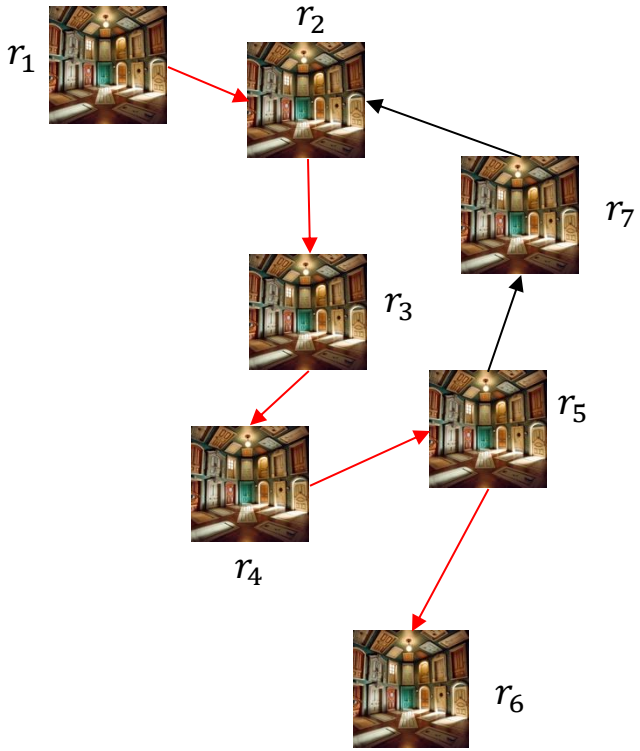
Wieso keine Zyklen?

- Wenn wir Zyklen haben kommt es zu **endloser Rekursion** ohne, dass wir uns merken, in welchen wir Räumen wir bereits waren.



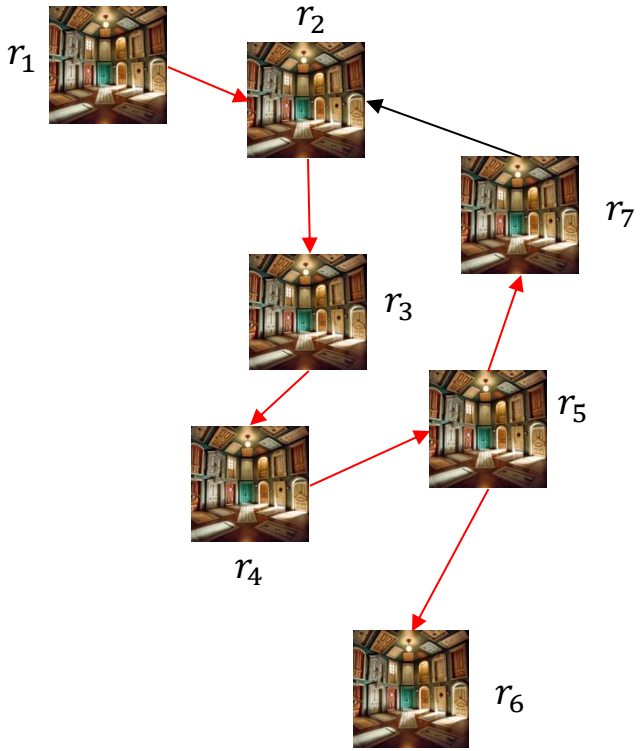
Wieso keine Zyklen?

- Wenn wir Zyklen haben kommt es zu **endloser Rekursion** ohne, dass wir uns merken, in welchen wir Räumen wir bereits waren.



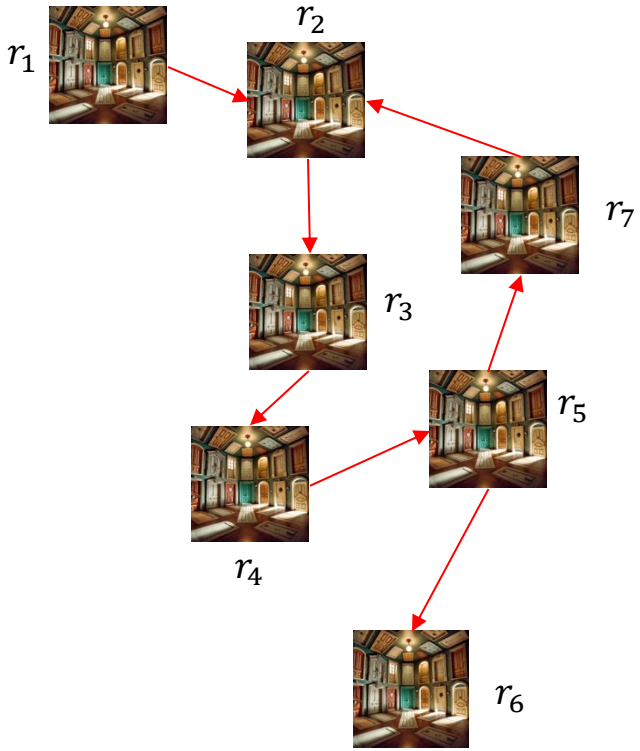
Wieso keine Zyklen?

- Wenn wir Zyklen haben kommt es zu **endloser Rekursion** ohne, dass wir uns merken, in welchen wir Räumen wir bereits waren.



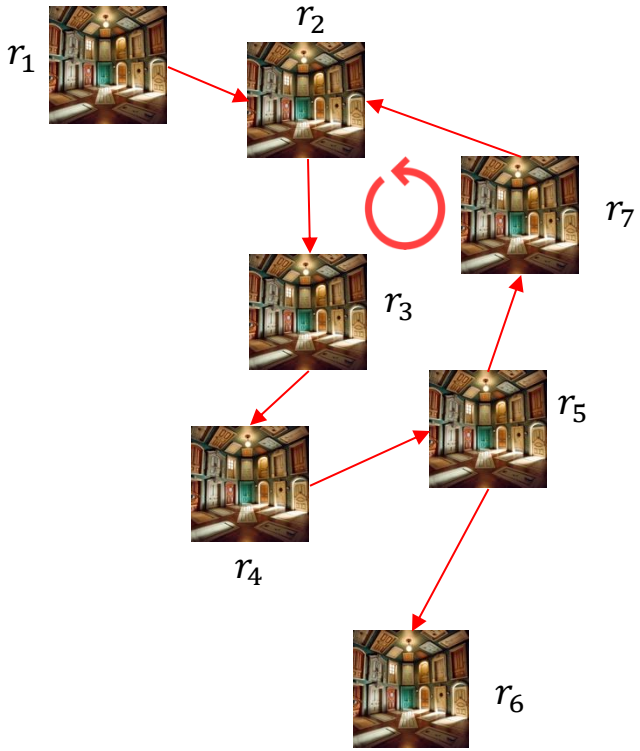
Wieso keine Zyklen?

- Wenn wir Zyklen haben kommt es zu **endloser Rekursion** ohne, dass wir uns merken, in welchen wir Räumen wir bereits waren.



Wieso keine Zyklen?

- Wenn wir Zyklen haben kommt es zu **endloser Rekursion** ohne, dass wir uns merken, in welchen wir Räumen wir bereits waren.



Wieso keine Zyklen?

- Wenn wir Zyklen haben kommt es zu **endloser Rekursion** ohne, dass wir uns merken, in welchen wir Räumen wir bereits waren.

Probleme Lösen: Labyrinth

2. Implementieren Sie die Methode `Labyrinth.task2(Room room)`. Die Methode soll true zurückgeben genau dann, wenn es zwei Lösungspfade $r_1, \ldots, r_N$ und $s_1, \ldots, s_N$ für `room` gibt, sodass:

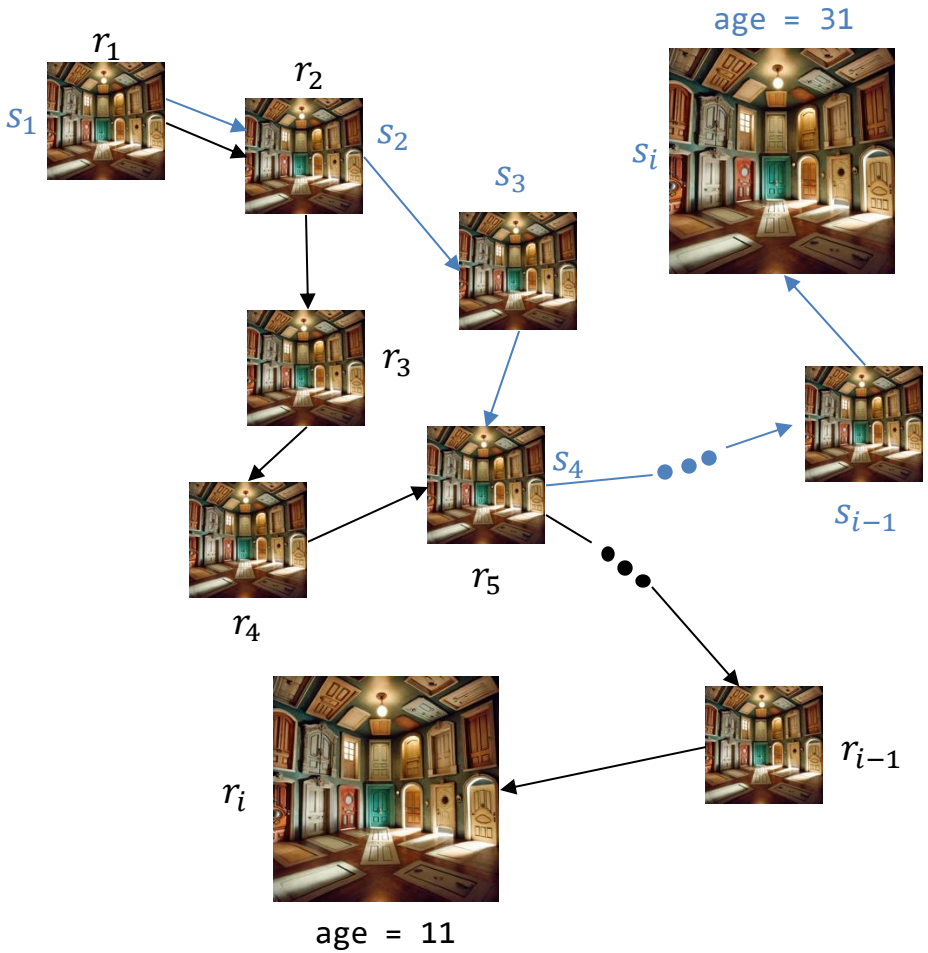
   - Die Räume r_i und s_i haben das gleiche Alter für jedes i mit $1 \leq i \leq N$.
   - Für mindestens ein i mit $1 \leq i \leq N$ gilt, dass r_i und s_i unterschiedlich sind (verschiedene Referenzen).

Wie lösen wir das Problem?

- **Rekursive Lösung:** Damit wir das Problem rekursiv lösen können, müssen wir **Teilprobleme** identifizieren.

Teilprobleme identifizieren:

- Wir betrachten jetzt zwei Räume r_i und s_i und zwei Pfade $r_1, \ldots, r_i$ und $s_1, \ldots, s_i$ wobei r_1 und s_1 beide der Raum `Room` sind.
- Was müssen wir über die Teilsequenzen $r_1, \ldots, r_{i-1}$ und $s_1, \ldots, s_{i-1}$ wissen?
- Wir nehmen an, dass die beiden Teilsequenzen die Alterbedingung erfüllen.



Teilprobleme identifizieren:

- Falls sich die Pfade bereits getrennt haben, so prüfen wir ob r_i und s_i beide Ausgänge sind und das gleiche Alter haben. (Falls ja dann sind wir fertig.)
- Sonst explorieren wir alle möglichen Pfadpaare (r_{i+1}, s_{i+1}).
- Wie merken wir uns ob sich die Pfade bereits getrennt haben?

Mit einem `boolean` Parameter

```java
public static boolean solve2(Room room1, Room room2, boolean samePath) {
    if(room1.isExit() && room2.isExit() && !samePath) {
        return true;
    }
    for(int i = 0; i < room1.doorsTo.length; ++i) {
        for(int k = 0; k < room2.doorsTo.length; ++k) {
            if(room1.doorsTo[i].age == room2.doorsTo[k].age) {
                if(solve2(room1.doorsTo[i], room2.doorsTo[k], (samePath && room1 == room2)))
                    return true;
                }
            }
        }
    }
    return false;
}
```

Pfade haben sich getrennt und beide Räume sind Ausgänge

```java
public static boolean solve2(Room room1, Room room2, boolean samePath) {
    if(room1.isExit() && room2.isExit() && !samePath) {
        return true;
    }
    for(int i = 0; i < room1.doorsTo.length; ++i) {
        for(int k = 0; k < room2.doorsTo.length; ++k) {
            if(room1.doorsTo[i].age == room2.doorsTo[k].age) {
                if(solve2(room1.doorsTo[i], room2.doorsTo[k], (samePath && room1 == room2)))
                    return true;
                }
            }
        }
    }
    return false;
}
```

Falls die Pfade sich getrennt haben und nur einer der Pfade ein Ausgang ist, dann terminiert einer der for loops ohne Ausführung des bod

dann landen wir hier

```
public static boolean solve2(Room room1, Room room2, boolean samePath) {
    if(room1.isExit() && room2.isExit() && !samePath) {
        return true;
    }
    for(int i = 0; i < room1.doorsTo.length; ++i) {
        for(int k = 0; k < room2.doorsTo.length; ++k) {
            if(room1.doorsTo[i].age == room2.doorsTo[k].age) {
                if(solve2(room1.doorsTo[i], room2.doorsTo[k], (samePath && room1 == room2)))
                    return true;
                }
            }
        }
    }
    return false;
}
```

Hier generieren wir die Raumpaare

```java
public static boolean solve2(Room room1, Room room2, boolean samePath) {
    if(room1.isExit() && room2.isExit() && !samePath) {
        return true;
    }
    for(int i = 0; i < room1.doorsTo.length; ++i) {
        for(int k = 0; k < room2.doorsTo.length; ++k) {
            if(room1.doorsTo[i].age == room2.doorsTo[k].age) {
                if(solve2(room1.doorsTo[i], room2.doorsTo[k], (samePath && room1 == room2)))
                    return true;
                }
            }
        }
    }
    return false;
}
```

Ist es ein valides Paar?

```java
public static boolean solve2(Room room1, Room room2, boolean samePath) {
    if(room1.isExit() && room2.isExit() && !samePath) {
        return true;
    }
    for(int i = 0; i < room1.doorsTo.length; ++i) {
        for(int k = 0; k < room2.doorsTo.length; ++k) {
            if(room1.doorsTo[i].age == room2.doorsTo[k].age) {
                if(solve2(room1.doorsTo[i], room2.doorsTo[k], (samePath && room1 == room2)))
                    return true;
                }
            }
        }
    }
    return false;
}
```

Wir prüfen den rekursiven Aufruf und wir geben weiter, ob der Pfad sich getrennt hat.

```java
public static boolean solve1(Room room, int sum) {
(…)
}

public static boolean task1(Room room) {
    return solve(room, 0)

}

public static boolean solve2(Room room1, Room room2, boolean samePath) {
(…)
}

public static boolean task2(Room room) {
    return solve2(room, room, true);
}
```

Dies gibt uns die Lösung

- Wir rufen `solve1` in `task1` auf mit `room` und initialer Summe `0`.
- Wir rufen `solve2` in `task2` auf mit `room` und initialem boolean Parameter `true`.

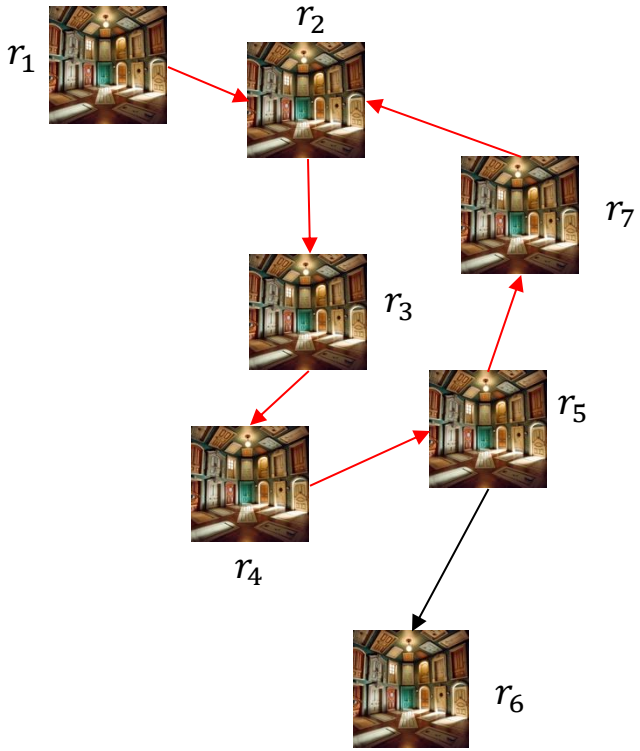
Mit gerichteten Zyklen Arbeiten

Probleme Lösen: Labyrinth (modified)

Aufgabe 1: Labyrinth (2021 W8)

Ein Labyrinth besteht aus einer Menge von Räumen, welche durch die Klasse `Room` dargestellt werden. Die Klasse hat zwei Attribute: Der Integer `age` (grösser gleich 0) beschreibt das Alter des Raums und der Array `doorsTo` (nie `null`) beschreibt die Türen von diesem Raum zu anderen Räumen. Alle Türen sind Falltüren, d.h. sie funktionieren nur in eine Richtung. Ein Raum ist ein Ausgang aus dem Labyrinth, wenn keine Türen von dem Raum wegführen, das heisst, wenn `doorsTo` eine Länge von 0 hat.

Für alle Aufgaben werden Sie in einen zufälligen Raum geworfen, welcher als Argument gegeben wird (garantiert nicht `null`) und von welchem aus Sie die Aufgabe lösen müssen. ~~Sie dürfen für alle Aufgaben annehmen, dass es im Labyrinth keinen Zyklus gibt. Das heisst, dass man einen Raum, welchen man durch eine Türe verlassen hat, nie wieder erreichen kann indem man weiteren Türen folgt.~~ Eine Sequenz von N Räumen $r_1, \ldots, r_N$ ist ein *Lösungspfad* für einen Raum `room` genau dann wenn: (1) Der erste Raum r_1 ist der Raum `room`, (2) der letzte Raum r_N ist ein Ausgang, und (3) jeder Raum r_i mit $1 \leq i < N$ hat eine Tür zum nächsten Raum in der Sequenz r_{i+1}.



Gerichtete Zyklen führen zu Problemen

- Wenn wir Zyklen haben kommt es zu **endloser Rekursion** ohne, dass wir uns merken, in welchen wir Räumen wir bereits waren.
- Wie merken wir uns in welchen Räumen wir bereits waren?
- **Später:** Sets
- **Jetzt:** Benutzen eines `visited` Attributs.

Probleme Lösen: Labyrinth

1. Implementieren Sie die Methode `Labyrinth.task1(Room room)`. Die Methode soll true zurückgeben genau dann, wenn es einen Lösungspfad $r_1, \ldots, r_N$ für `room` gibt, sodass:
   - Für jede Teilsequenz $r_1, \ldots, r_i$ mit $1 \leq i \leq N$ gilt, dass die Summe der Alter der Räume $r_1, \ldots, r_i$ nicht durch 3 teilbar ist.
2. Implementieren Sie die Methode `Labyrinth.task2(Room room)`. Die Methode soll true zurückgeben genau dann, wenn es zwei Lösungspfade $r_1, \ldots, r_N$ und $s_1, \ldots, s_N$ für `room` gibt, sodass:
   - Die Räume r_i und s_i haben das gleiche Alter für jedes i mit $1 \leq i \leq N$.
   - Für mindestens ein i mit $1 \leq i \leq N$ gilt, dass r_i und s_i unterschiedlich sind (verschiedene Referenzen).

Das können wir ausnutzen!

Sie dürfen Methoden und Felder der Klasse `Room` hinzufügen. Tests finden Sie in der Datei "LabyrinthTest.java".

Tipp: Lösen Sie die Aufgaben rekursiv. Für keine der Aufgaben müssen Sie alle Pfade generieren und dann erst prüfen, dass die Eigenschaften gelten. Manche der Tests enthalten Labyrinthe mit einer extrem grossen Anzahl an Pfaden aber leichten Lösungen.

```java
public class Room {
    boolean visited = false;
    int age;
    public Room[] doorsTo;

    public Room(int age, Room[] doorsTo) {
        this.age = age;
        this.doorsTo = doorsTo;
    }
    public boolean isExit() {
        return doorsTo.length == 0;
    }
    public int getAge() {
        return age;
    }
}
```

Wir modifizieren die Klasse `Room`.

- Wir fügen ein `boolean visited` Attribut hinzu.
- Wir setzen visited auf `false` für jedes `Room`-Objekt und auf `true`, wenn wir den Raum besucht haben.

Initialisiert das `visited` Attribut für jedes Objekt mit `false`.

```java
public static boolean solve1(Room room, int sum)  {
    if(!room.visited) {
        room.visited = true;
        sum = sum + room.age;

        if(sum % 3 == 0) { return false; }
        if(room.isExit()) { return true; }

        for(int i = 0; i < room.doorsTo.length; ++i) {
            if(solve1(room.doorsTo[i], sum)) {
                return true;
            }
        }
    }
    room.visited = false;
    return false;
}
```

Wir prüfen ob `room` bereits auf unserem Pfad liegt.

```java
public static boolean solve1(Room room, int sum)  {
    if(!room.visited) {
        room.visited = true;
        sum = sum + room.age;

        if(sum % 3 == 0) { return false; }
        if(room.isExit()) { return true; }

        for(int i = 0; i < room.doorsTo.length; ++i) {
            if(solve1(room.doorsTo[i], sum)) {
                return true;
            }
        }
    }
    room.visited = false;
    return false;
}
```

Wir merken uns, dass `room` neu auf unserem Pfad liegt.

```java
public static boolean solve1(Room room, int sum)  {
    if(!room.visited) {
        room.visited = true;
        sum = sum + room.age;

        if(sum % 3 == 0) { return false; }
        if(room.isExit()) { return true; }

        for(int i = 0; i < room.doorsTo.length; ++i) {
            if(solve1(room.doorsTo[i], sum)) {
                return true;
            }
        }
    }
    room.visited = false;
    return false;
}
```

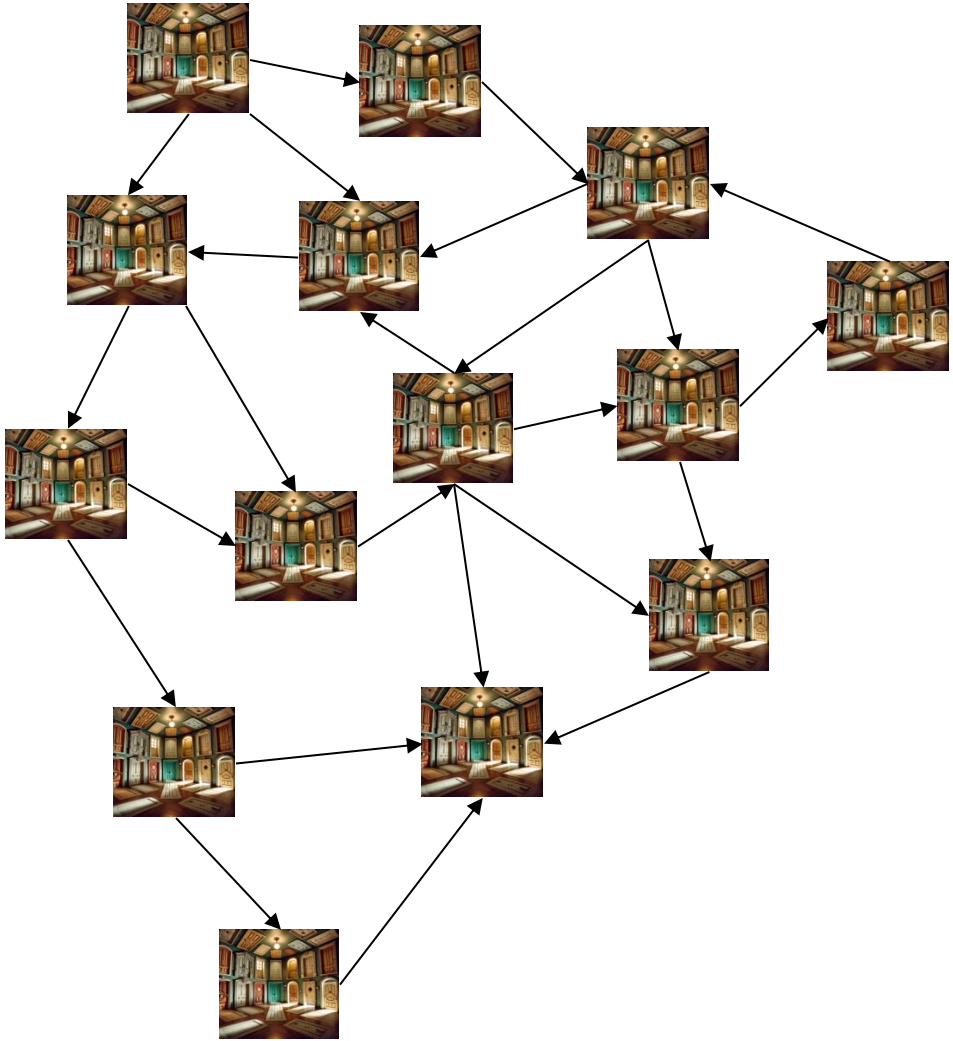
Wir führen den gleichen Code wie vorhin aus.

```java
public static boolean solve1(Room room, int sum)  {
    if(!room.visited) {
        room.visited = true;
        sum = sum + room.age;

        if(sum % 3 == 0) { return false; }
        if(room.isExit()) { return true; }

        for(int i = 0; i < room.doorsTo.length; ++i) {
            if(solve1(room.doorsTo[i], sum)) {
                return true;
            }
        }
    }
    room.visited = false;
    return false;
}
```

Wir setzen das Attribut wieder zurück, da wir `solve1` mehrmals auf dem gleichen Labyrinth ausführen wollen.

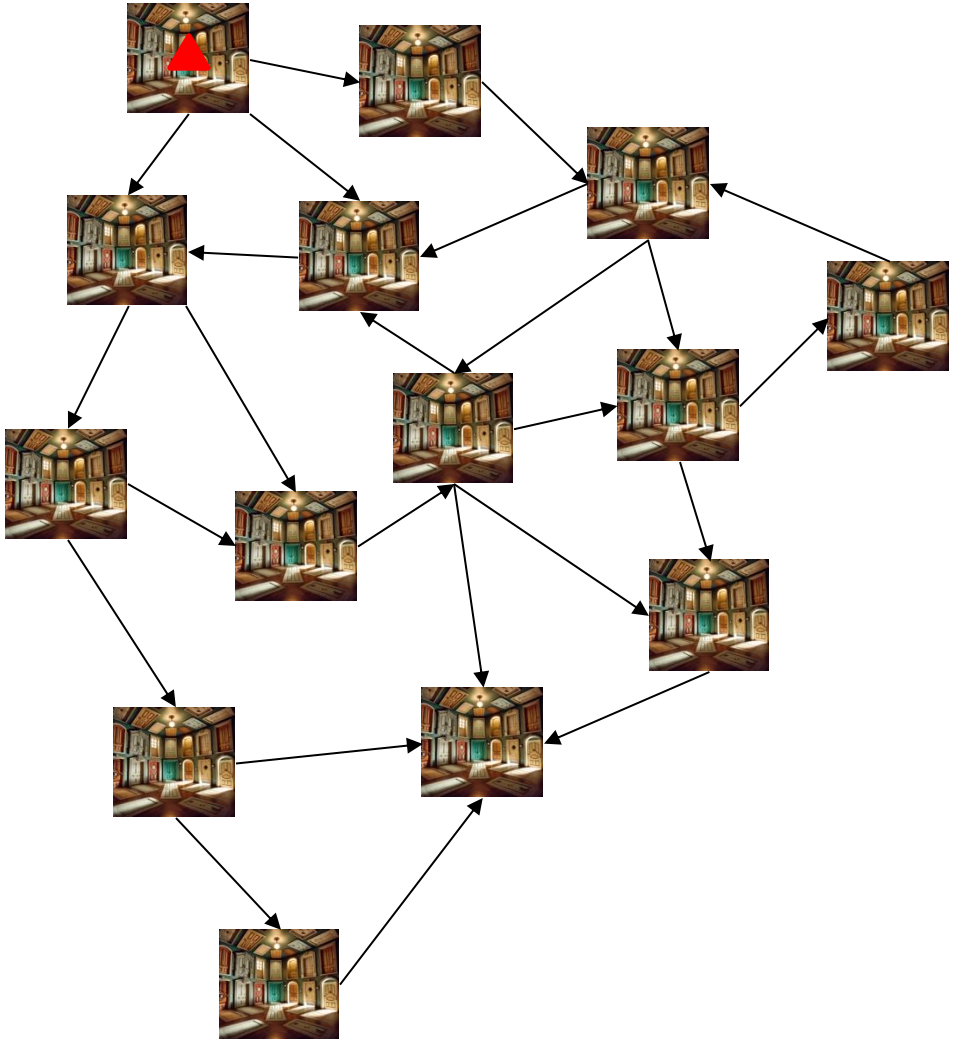


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

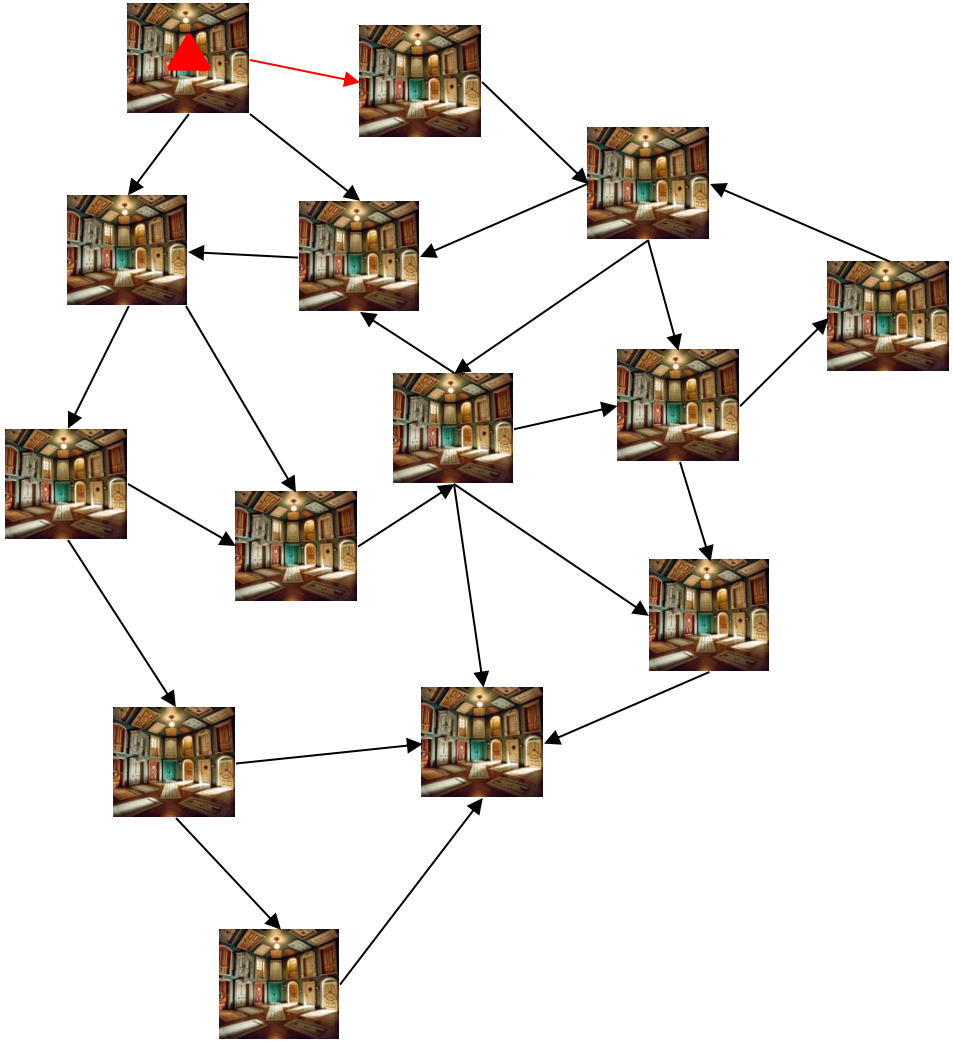


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

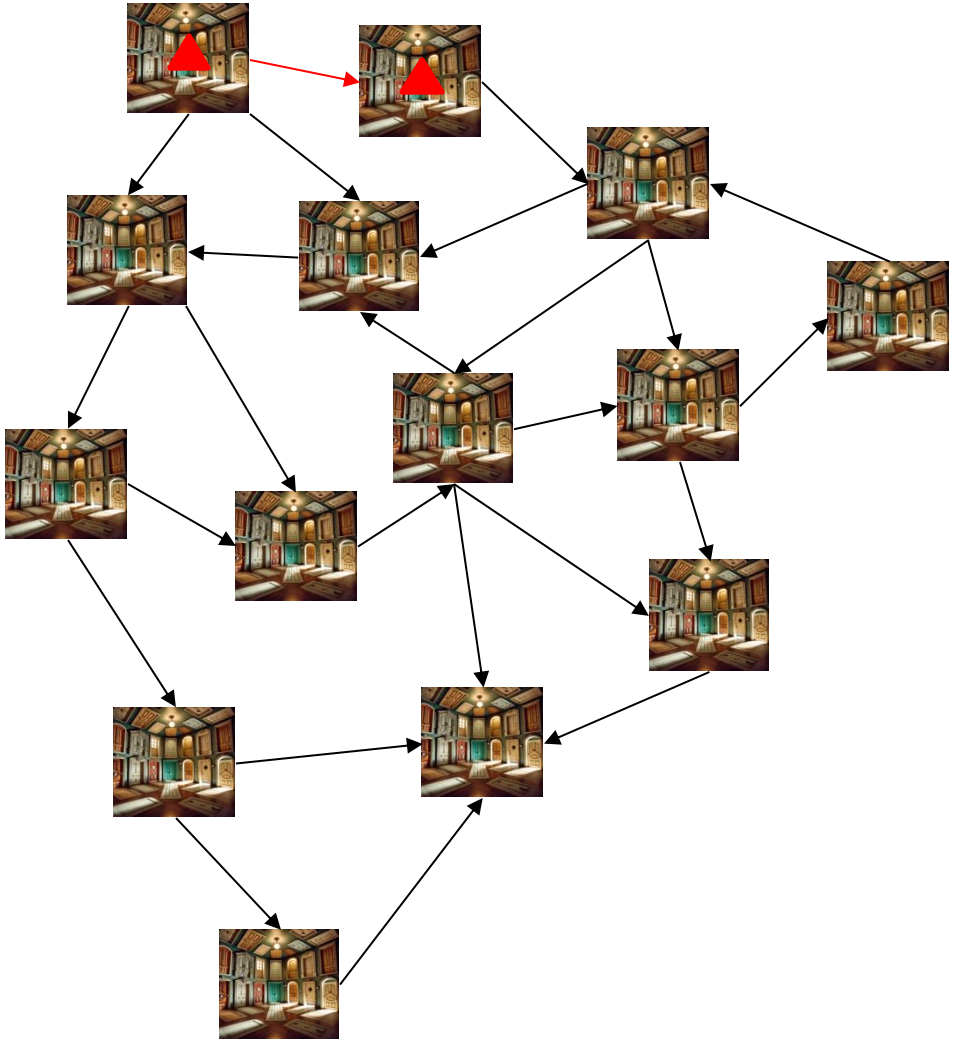


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

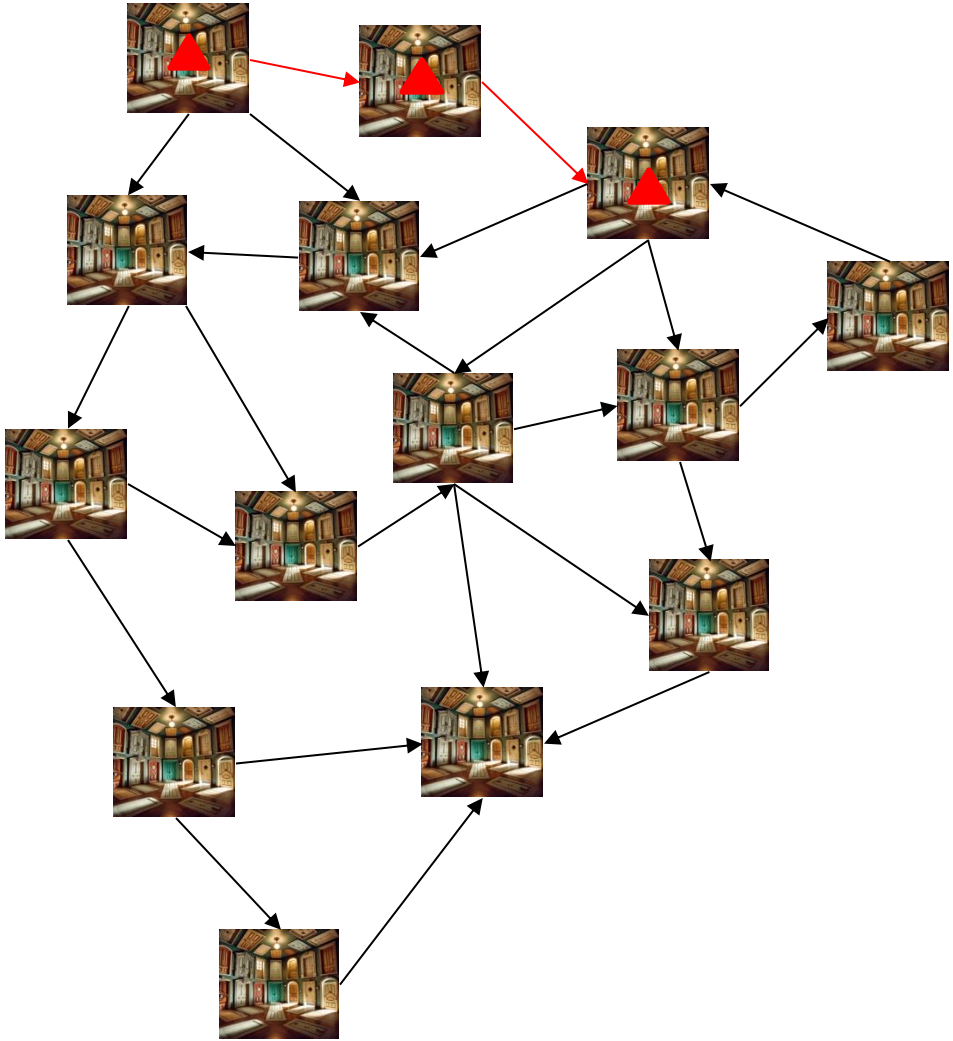


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

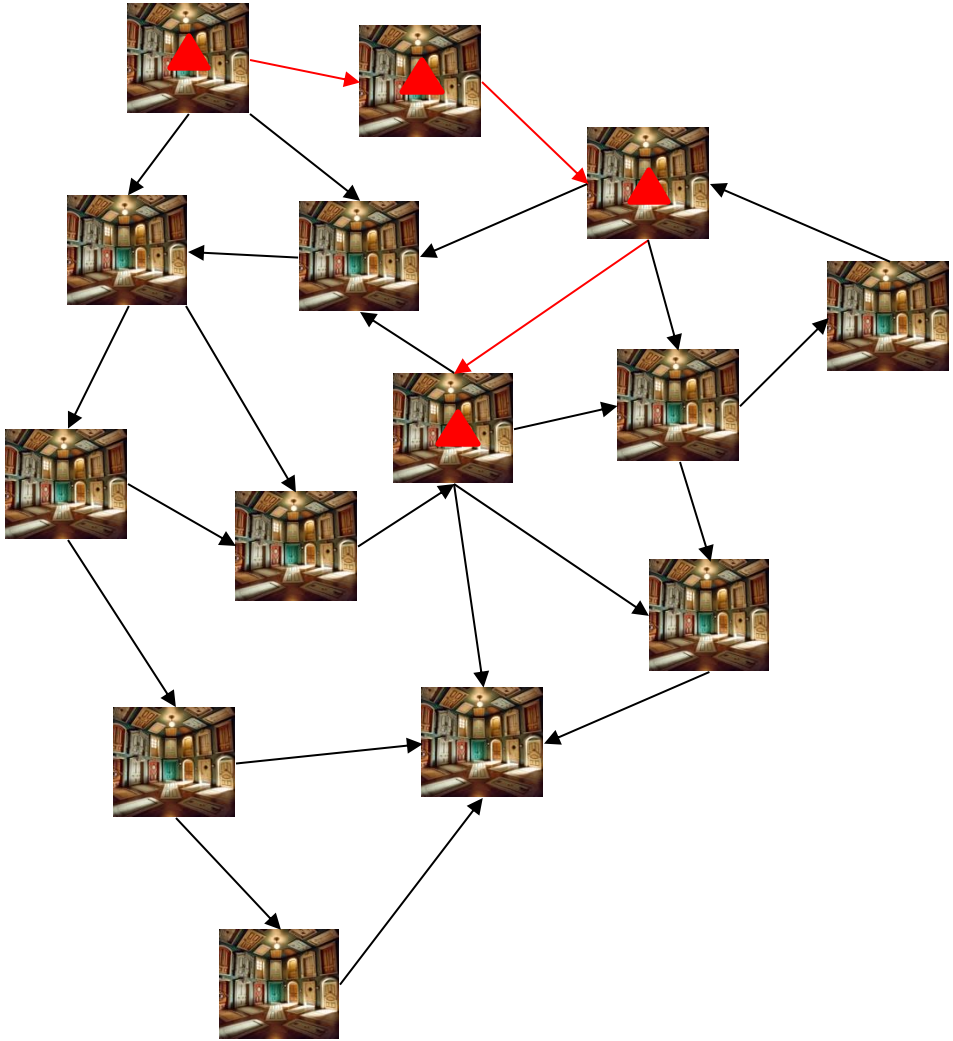


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

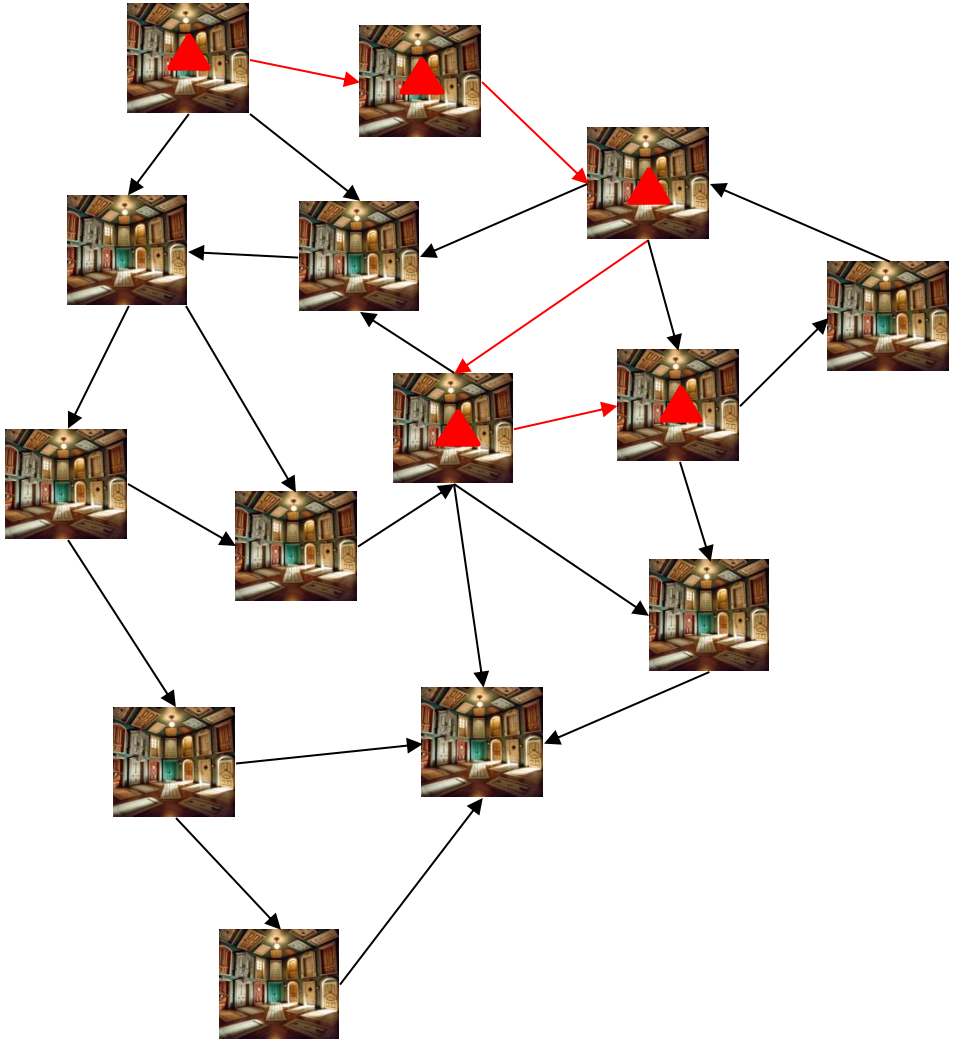


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

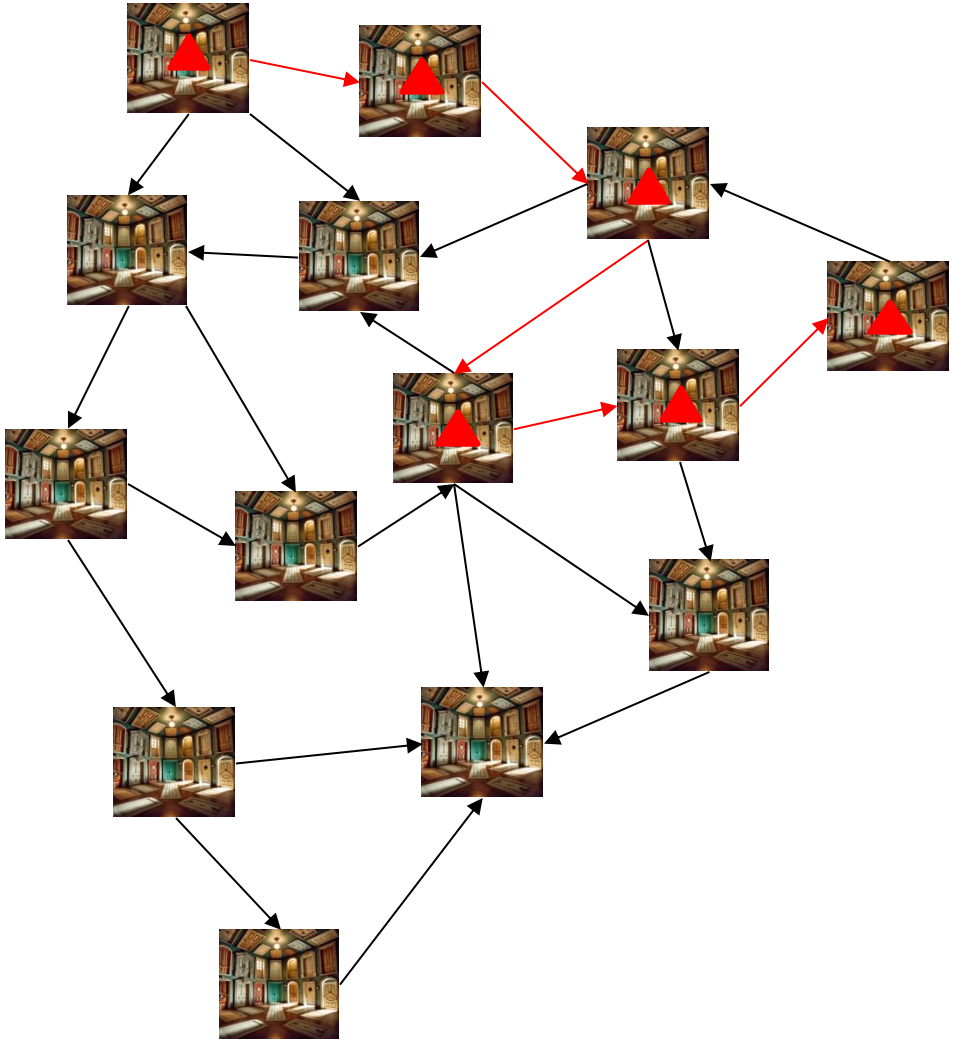


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

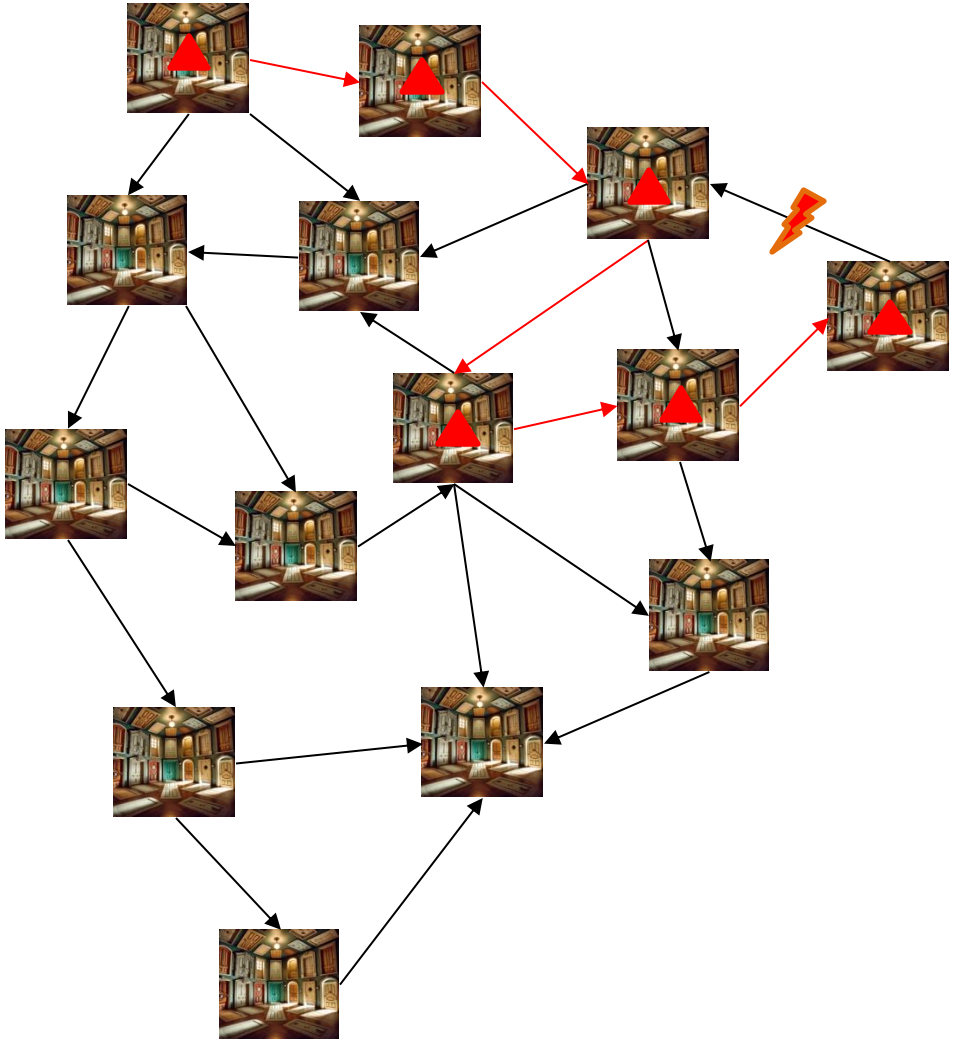


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

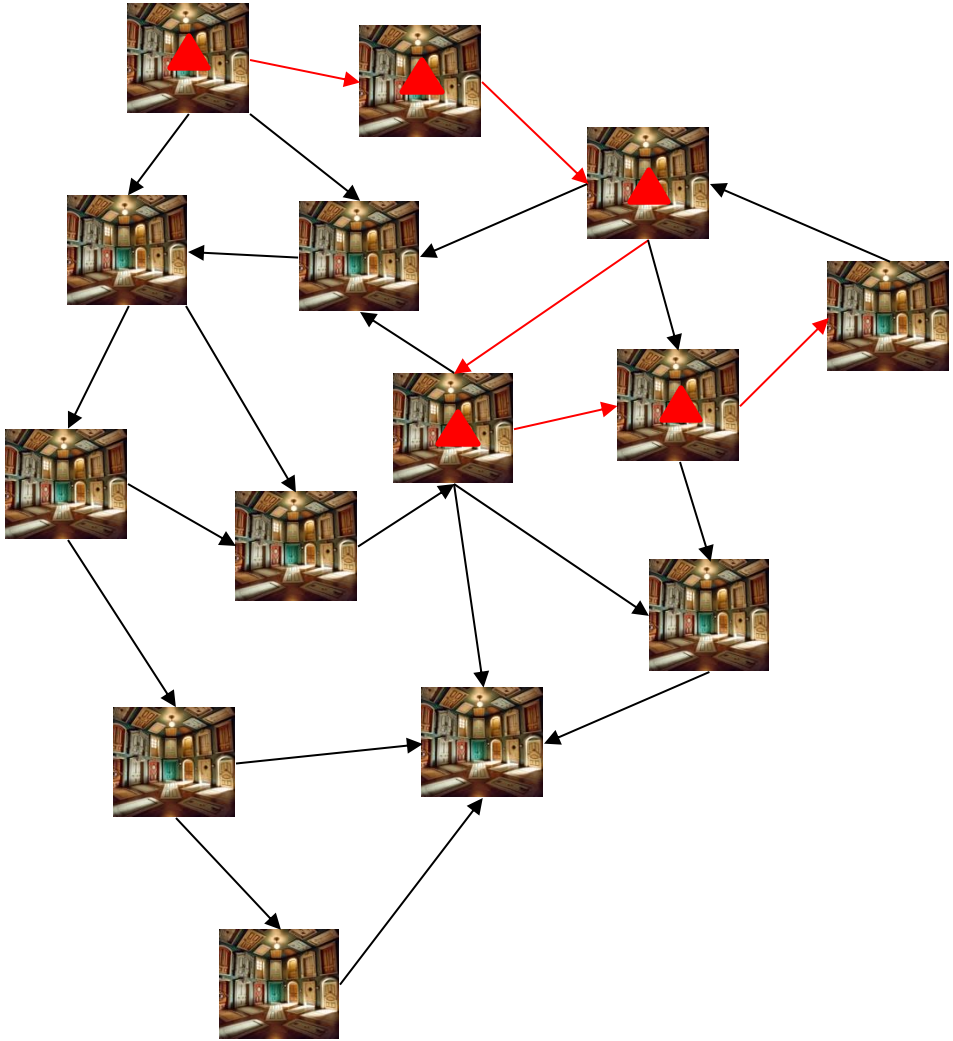


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

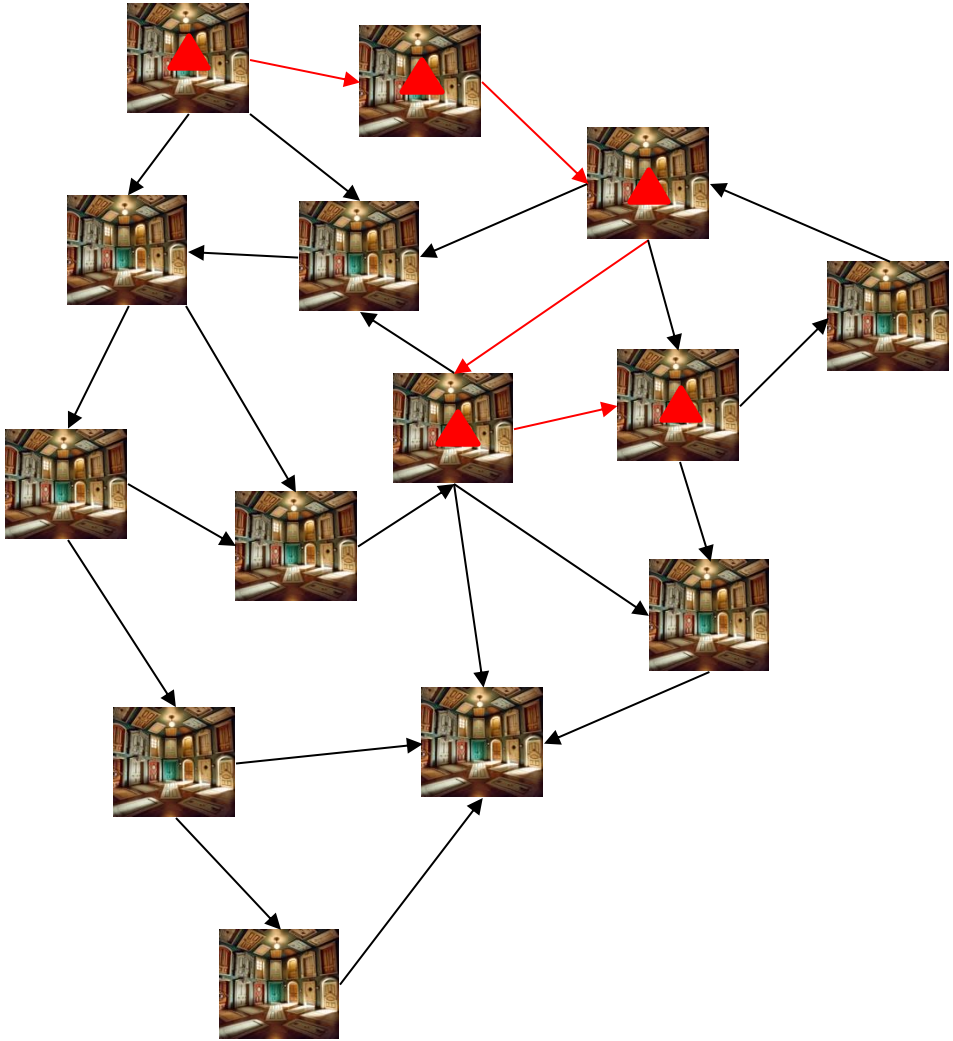


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

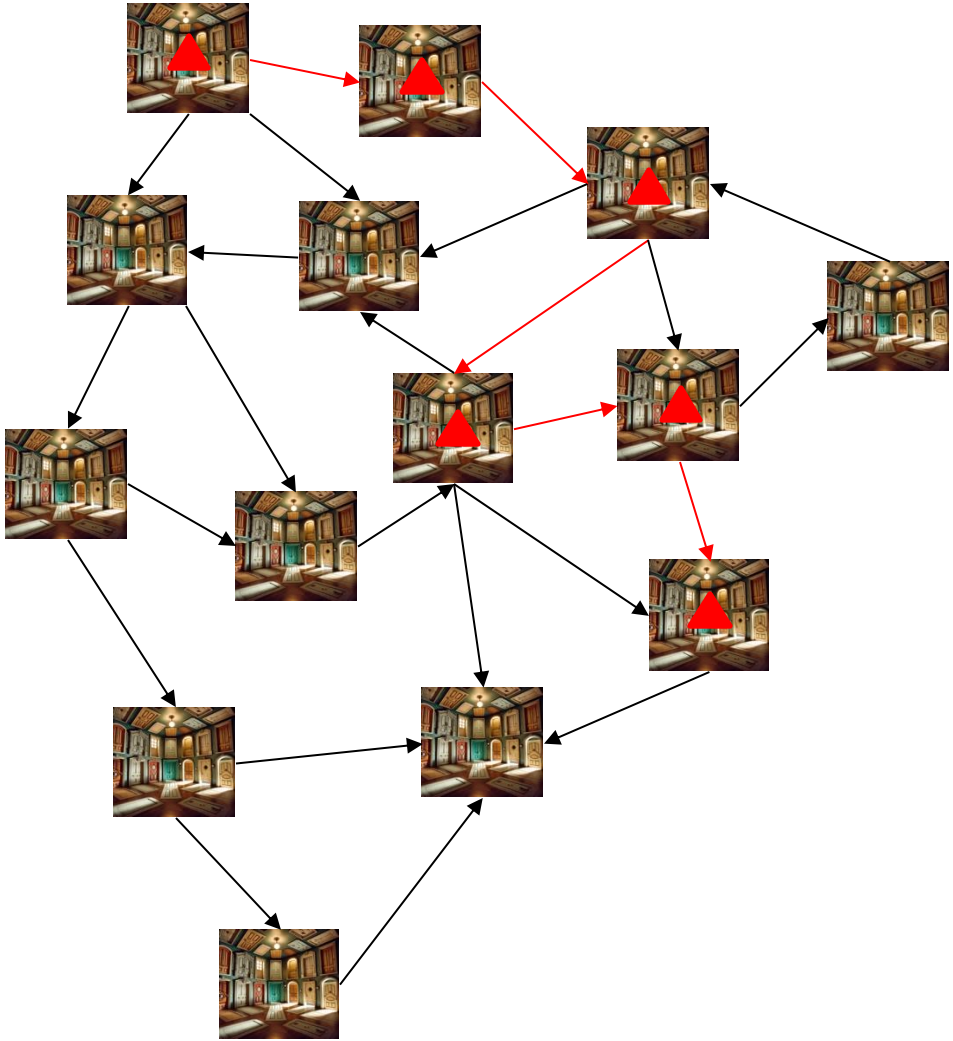


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

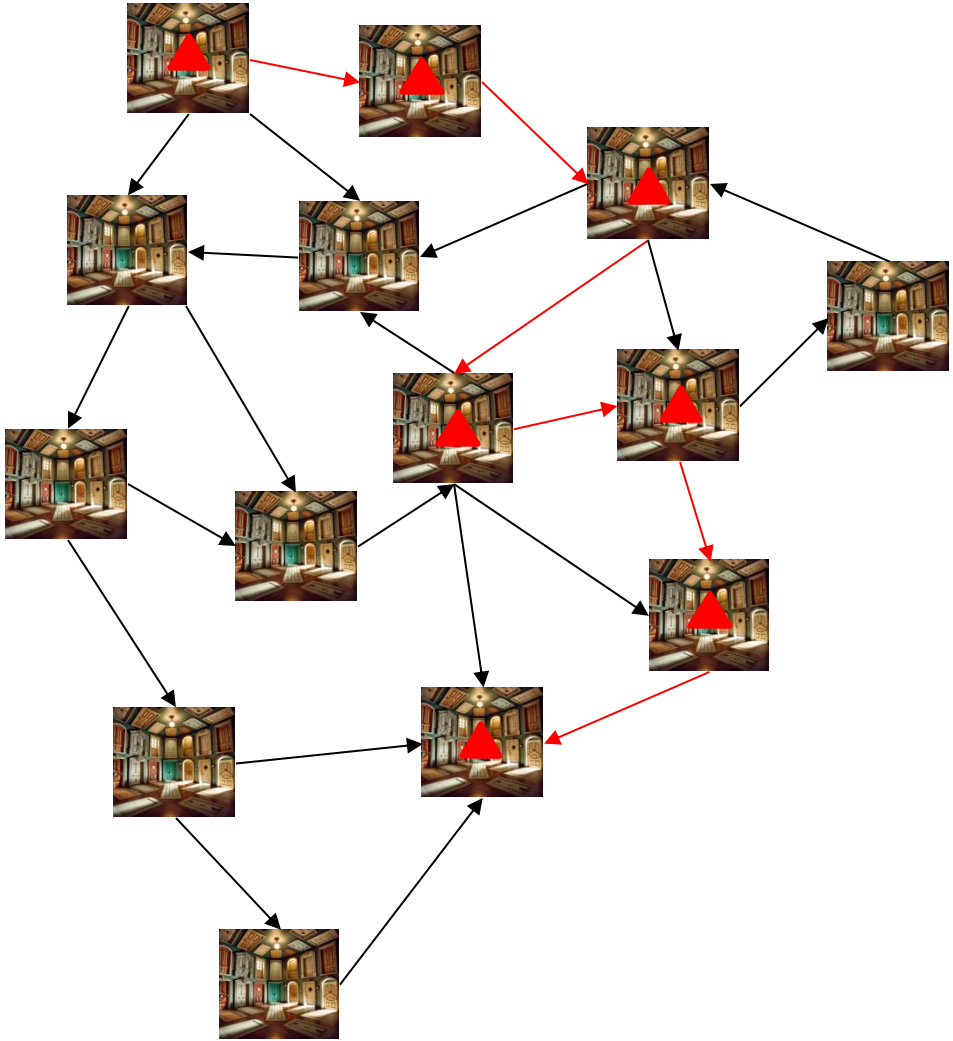


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

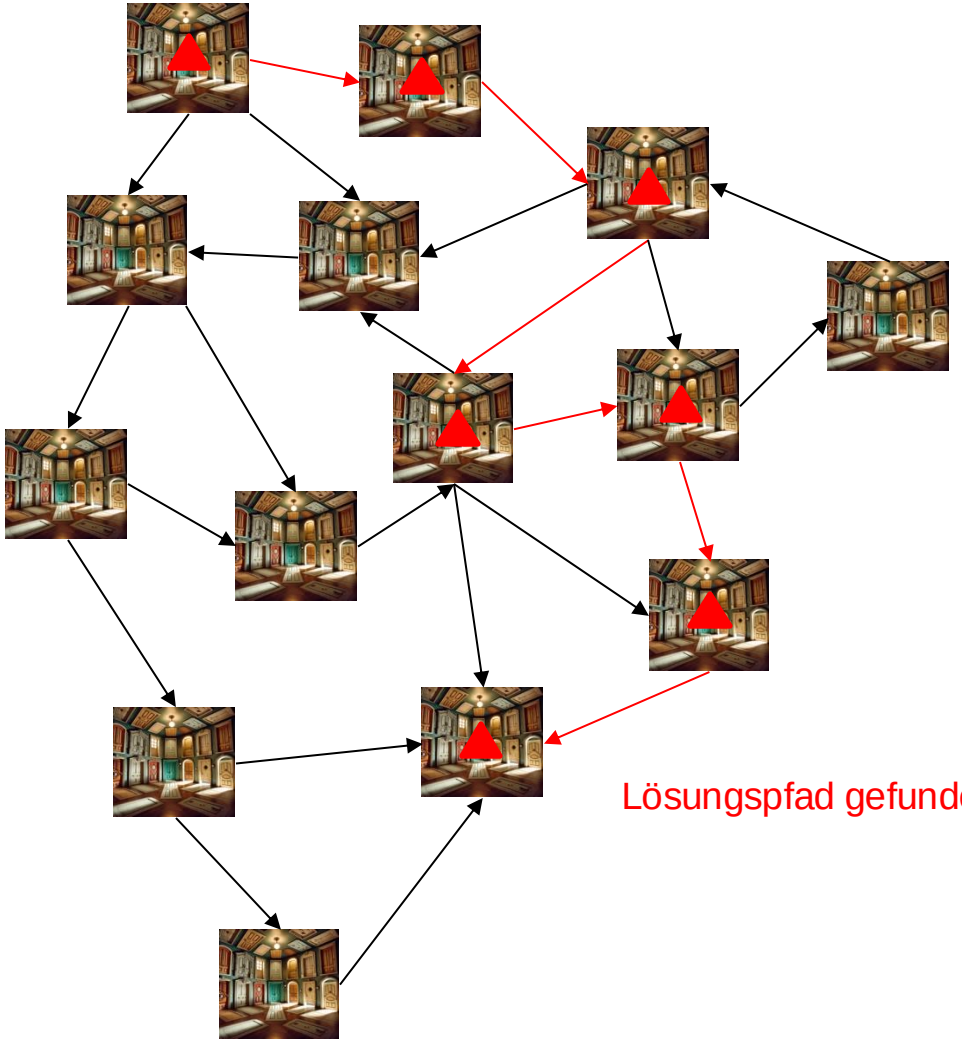


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited



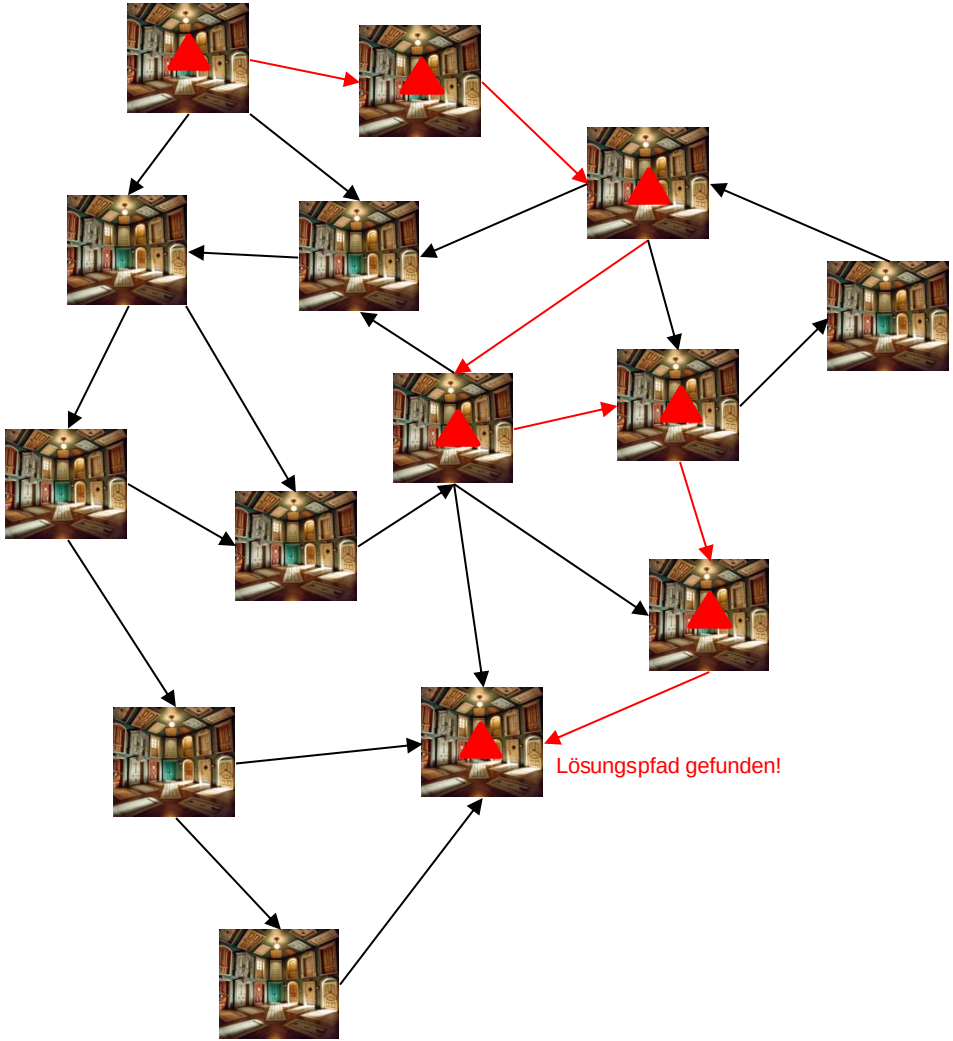
Lösungspfad gefunden!

Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

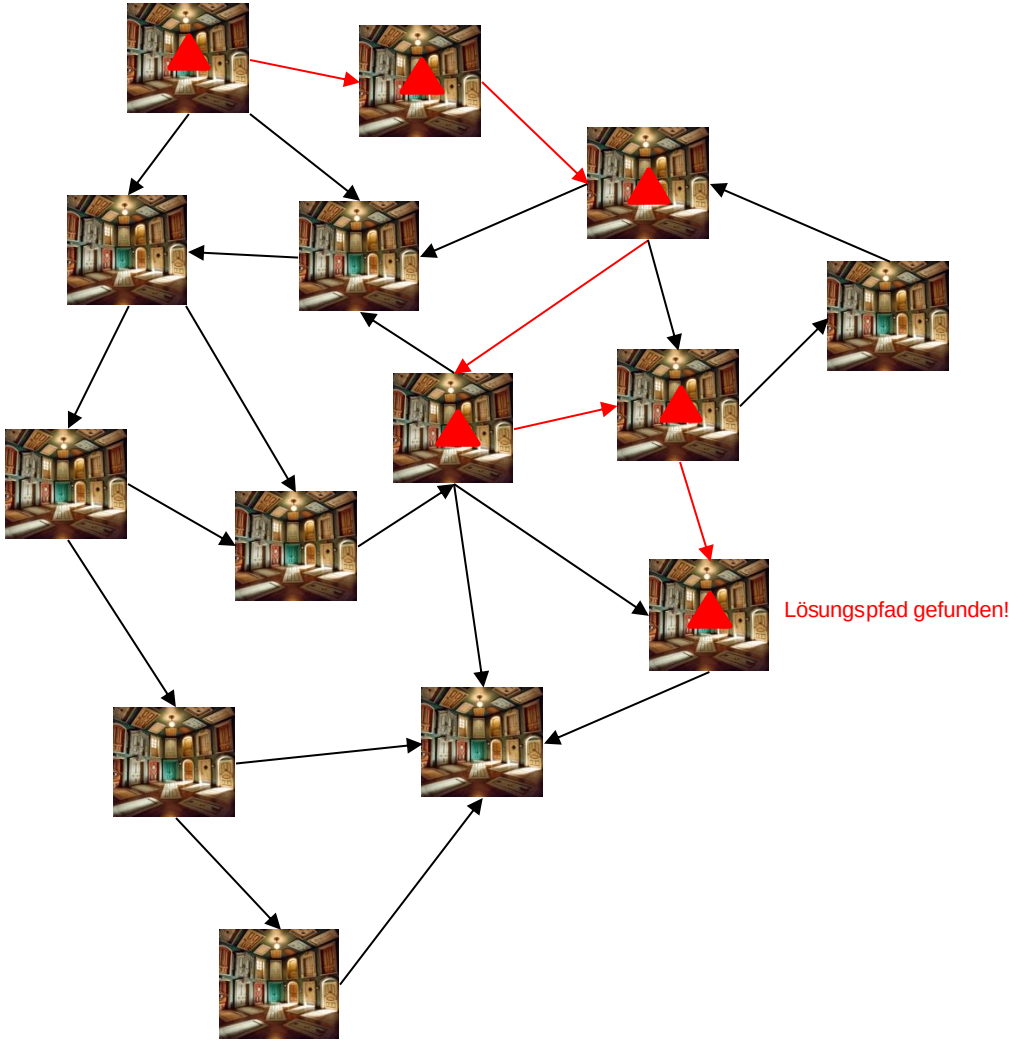


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

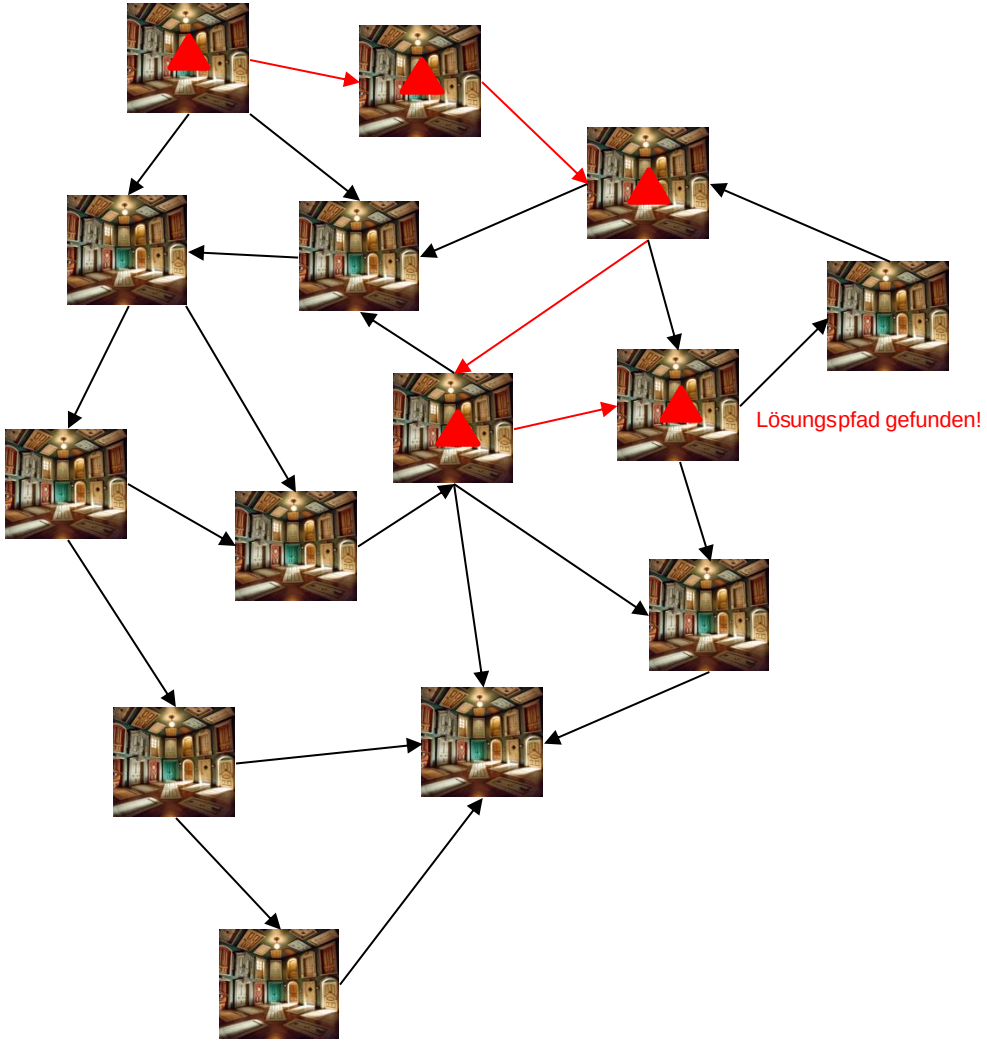


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

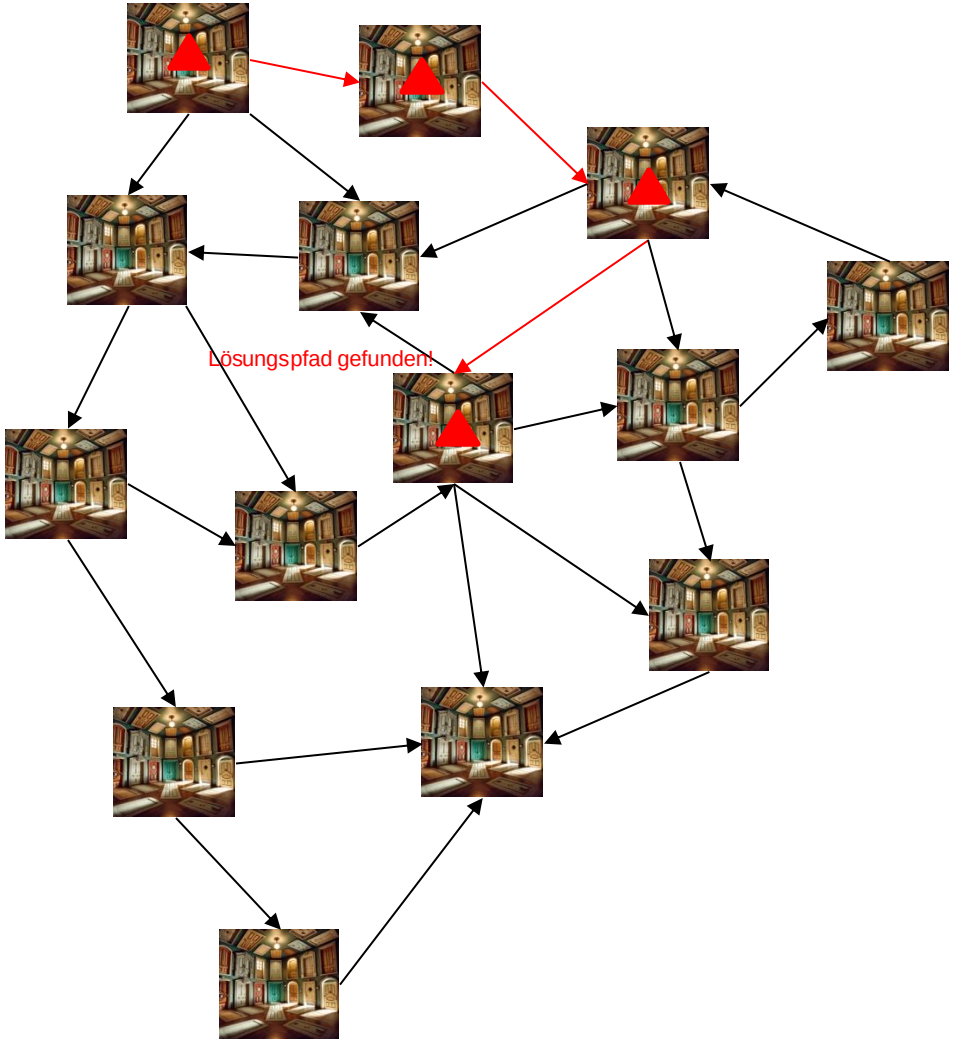


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

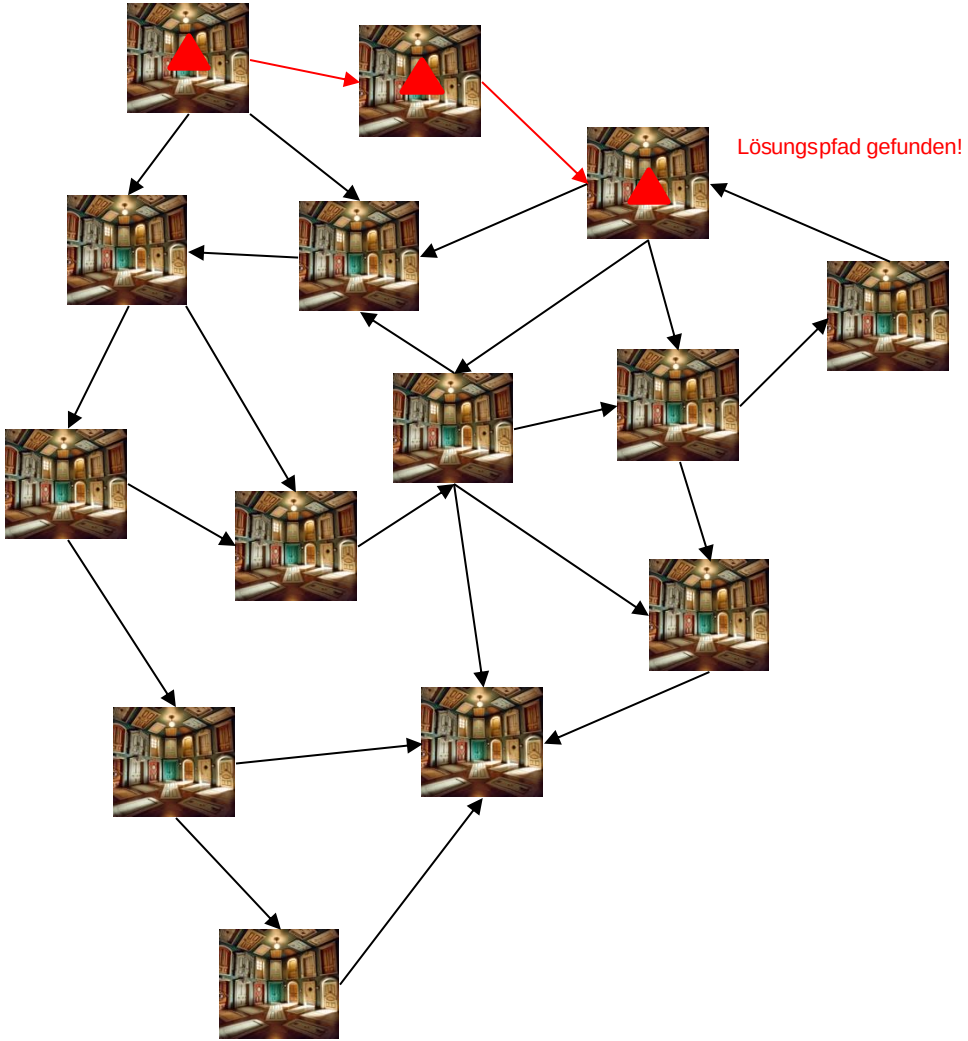


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

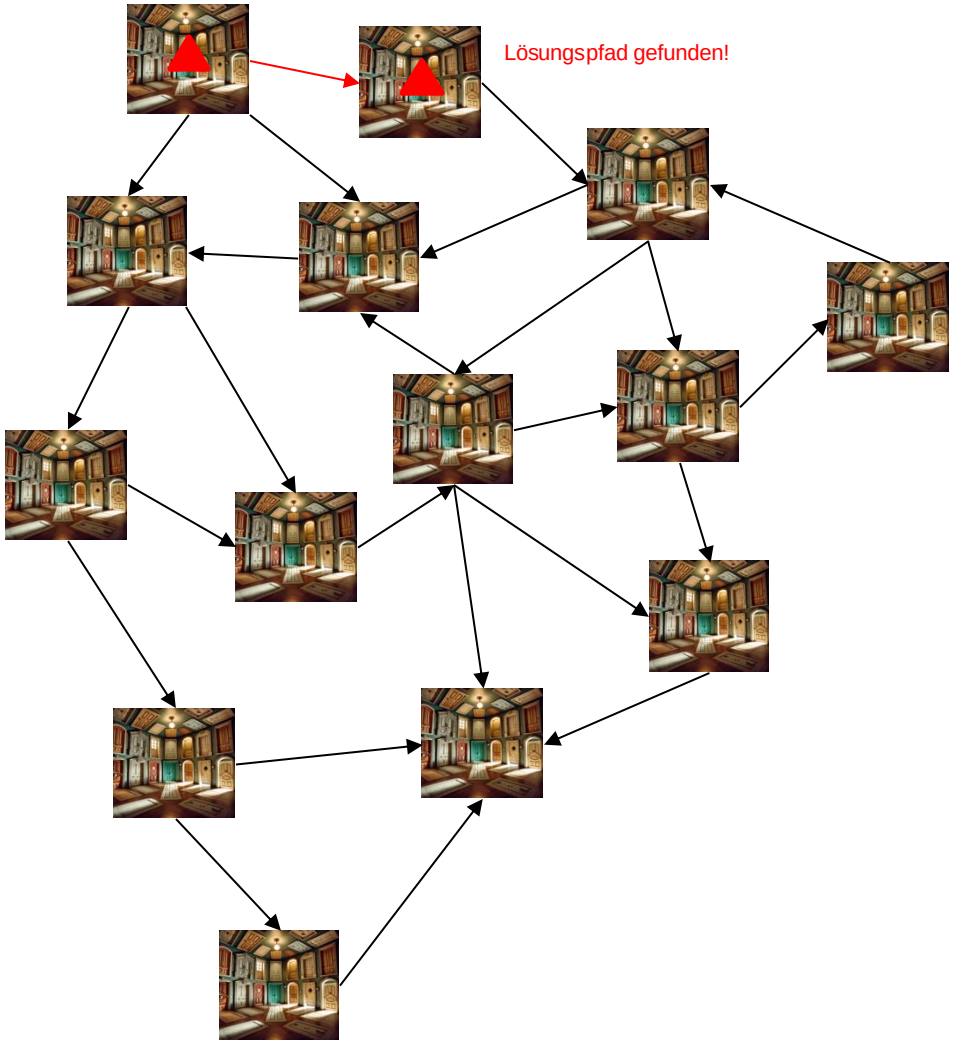


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

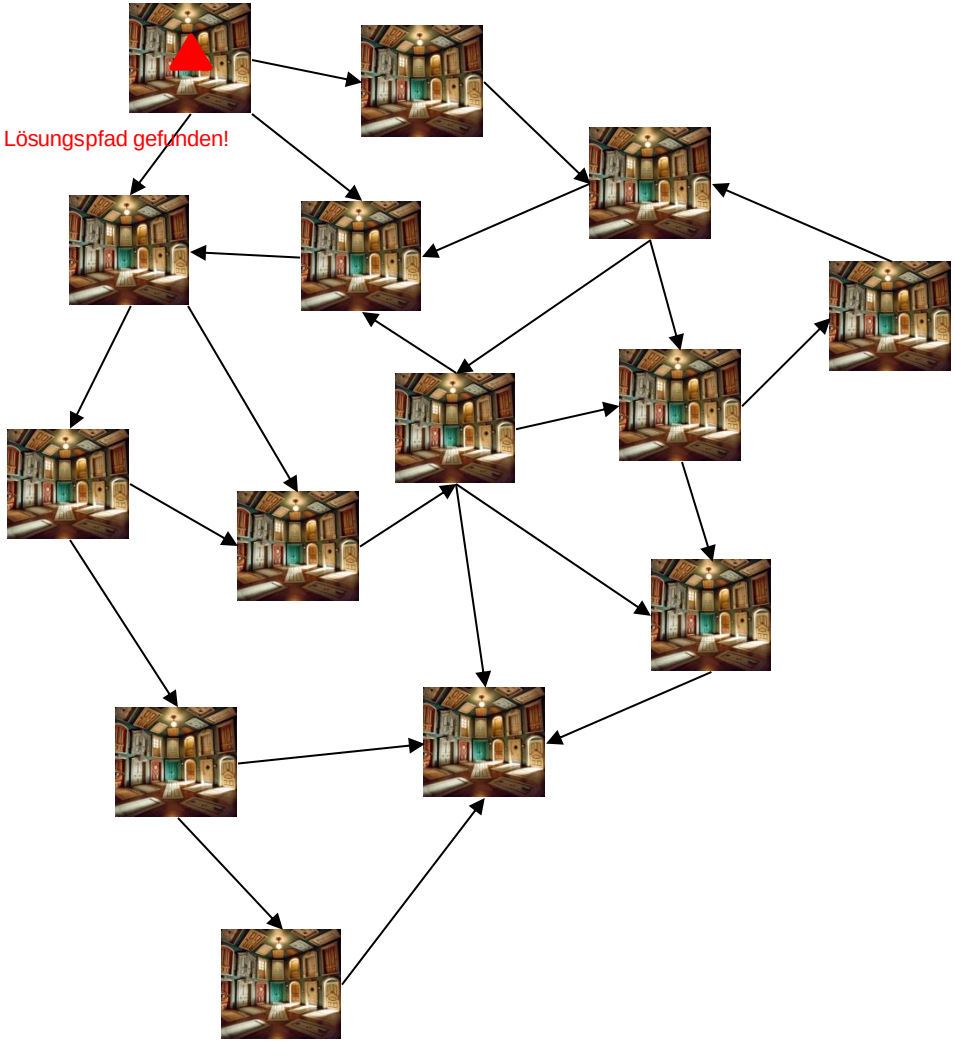


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

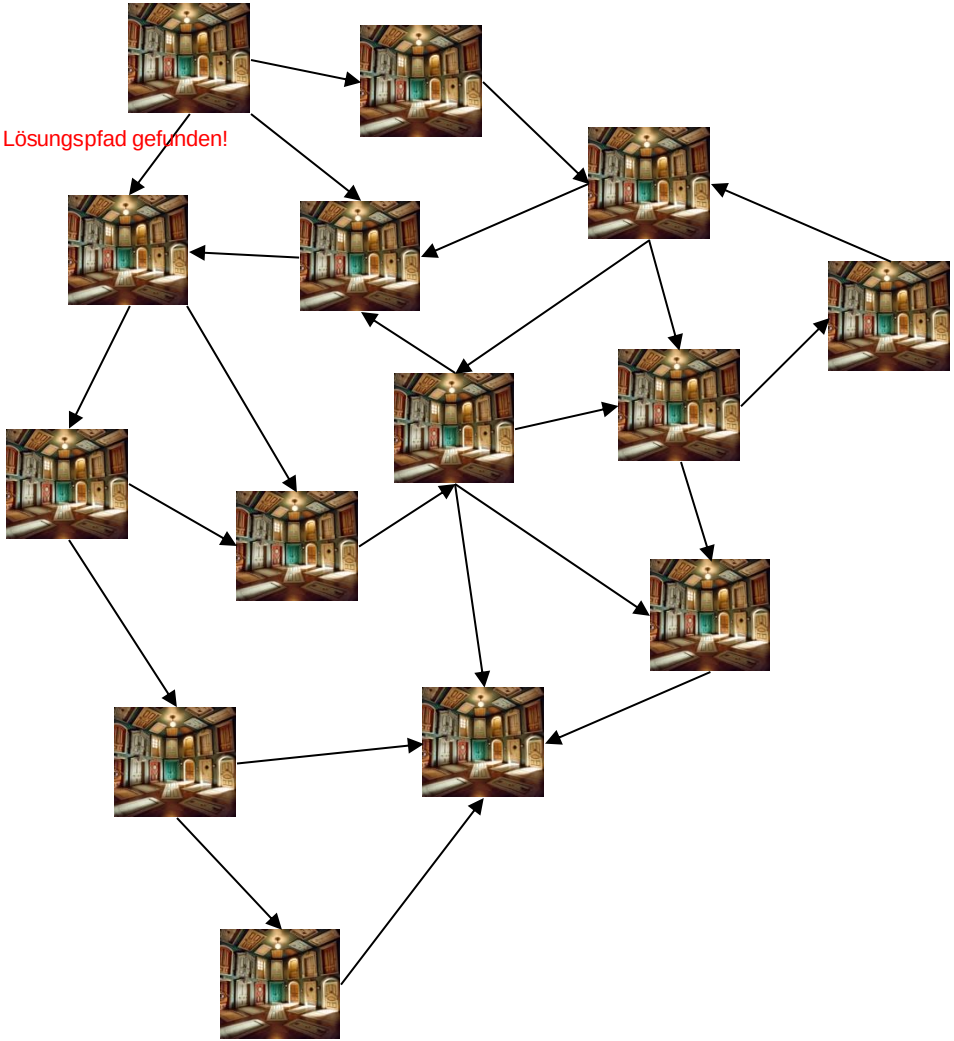


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited



Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

Enums

Kontext: Mögliches Spiel

- `Player` haben
  - Namen – beliebig
  - Rolle – «honest» oder «trickster»
  - Energielevel – steigt/fällt während des Spiels
    - Änderung hängt (auch) von Rolle ab
- Game verwaltet alle Spieler in `Player[] players`
- Frage: Wie Spieler-Rollen umsetzen?

```
public class Player {
  private ??? name;
  private ??? role;
  private int energy;
  …
}
```

Erste Version `Player` und `Game`

```java
public class Player {
  private String name;
  private String role;
  private int energy;
  …
  public void energize(int value) {
    energy += value;
  }
}
```

Rolle als Strings modelliert – gute Idee?

```java
public class Game {
  private Player[] players;
  private Random random;
  …

  public void energizeAll() {
    for (int i = 0; i < players.length; i++) {
      Player player = players[i];
      if (player.role.equals("HONEST")) {
        player.energize(1);
      } else if (player.role.equals("TRICKSTER")) {
        player.energize(random.nextInt(-4, 5));
      }
    }
  }
}
```

Besser: Mit Enums

```
public enum Role {
  HONEST,
  TRICKSTER,
  SOCERER,
}
```

```
public class Player {
  private String name;
  private Role role;
  private int energy;
  …
}
```

Vorteile?

```
public class Game {
  private Player[] players;
  private Random random;
  …

  public void energizeAll() {
    for (int i = 0; i < players.length; i++) {
      Player player = players[i];
      if (player.role == Role.HONEST) {
        player.energize(1);
      } else if (player.role == Role.TRICKSTER) {
        player.energize(random.nextInt(-4, 5));
      }
    }
  }
}
```

Loop - Invarianten

Wir wollen das folgende Hoare Triple beweisen:

```
{ Precondition }
  while ( Condition ) { Body };
{ Postcondition }
```

Dies können wir tun, falls eine Invariante existiert, für welche Folgendes gilt:

1. Precondition ⇒ Invariante

2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel.

3. ¬ Condition ∧ Invariante ⇒ Postcondition

Das funktioniert nicht immer…

Wir wollen das folgende Hoare Triple beweisen:

```
{ Precondition }
  Codeblock1
  while ( Condition ) { Body };
  Codeblock2
{ Postcondition }
```

Dies können wir tun, falls eine Invariante existiert, für welche folgendes gilt:

1. { Precondition } Codeblock1; { Invariante } ist ein valides Tripel.

2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel.

3. { ¬ Condition ∧ Invariante } Codeblock2; { Postcondition } ist ein valides Tripel.

Vorbesprechung

Aufgabe 1: Loop-Invariante

Gegeben ist eine Postcondition für das folgende Programm

```
public int compute(String s, char c) {
  // Precondition s != null
  int x;
  int n;

  x = 0;
  n = 0;

  // Loop Invariante:
  while (x < s.length()) {
    if (s.charAt(x) == c) {
      n = n + 1;
    }
    x = x + 1;
  }

  // Postcondition: count(s, c) == n
  return n;
}
```

Die Methode `count(String s, char c)` gibt zurück wie oft der Character `c` im String `s` vorkommt. Schreiben Sie die Loop Invariante in die Datei "LoopInvariante.txt". **Tipp:** Benutzen Sie die `substring` Methode.

Aufgabe 2: Linked List

Bisher haben Sie Arrays verwendet, wenn Sie mit einer grösseren Anzahl von Werten gearbeitet haben. Ein Nachteil von Arrays ist, dass die Grösse beim Erstellen des Arrays festgelegt werden muss und danach nicht mehr verändert werden kann. In dieser Aufgabe implementieren Sie selbst eine Datenstruktur, bei welcher die Grösse im Vornherein nicht bestimmt ist und welche jederzeit wachsen und schrumpfen kann: eine *linked list* oder *verkettete Liste*.

Eine verkettete Liste besteht aus mehreren Objekten, welche Referenzen zueinander haben. Für diese Aufgabe besteht jede Liste aus einem "Listen-Objekt" der Klasse `LinkedIntList`, welches die gesamte Liste repräsentiert, und aus mehreren "Knoten-Objekten" der Klasse `IntNode`, eines für jeden Wert in der Liste. Die Liste heisst "verkettet", weil jedes Knoten-Objekt ein Feld mit einer Referenz zum nächsten Knoten in der Liste enthält. Das `LinkedIntList`-Objekt schliesslich enthält eine Referenz zum ersten und zum letzten Knoten und hat ausserdem ein Feld für die Länge der Liste.

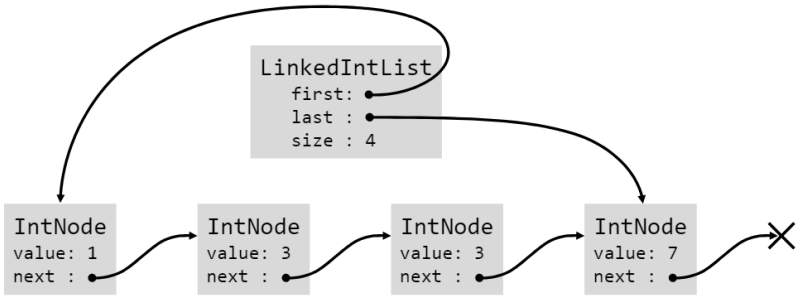


Abbildung 1: Verkettete Liste mit Werten 1, 3, 3, 7.

Aufgabe 2: Linked List

Name	Parameter	Rückg.-Typ	Beschreibung
`addLast`	`int value`	`void`	fügt einen Wert am Ende der Liste ein
`addFirst`	`int value`	`void`	fügt einen Wert am Anfang der Liste ein
`removeFirst`		`int`	entfernt den ersten Wert und gibt ihn zurück
`removeLast`		`int`	entfernt den letzten Wert und gibt ihn zurück
`clear`		`void`	entfernt alle Wert in der Liste
`isEmpty`		`boolean`	gibt zurück, ob die Liste leer ist
`get`	`int index`	`int`	gibt den Wert an der Stelle `index` zurück
`set`	`int index, int value`	`void`	ersetzt den Wert an der Stelle `index` mit `value`
`getSize`		`int`	gibt zurück, wie viele Werte die Liste enthält

Einige dieser Methoden dürfen unter gewissen Bedingungen nicht aufgerufen werden. Zum Beispiel darf `removeFirst()` nicht aufgerufen werden, wenn die Liste leer ist, oder `get()` darf nicht aufgerufen werden, wenn der gegebene Index grösser oder gleich der aktuellen Länge der Liste ist. In solchen Situationen soll sich Ihr Programm mit einer Fehlermeldung beenden. Verwenden Sie folgendes Code-Stück dafür:

```
if(condition) {
    Errors.error(message);
}
```

Ersetzen Sie *`condition`* mit der Bedingung, unter welcher das Programm beendet werden soll, und *`message`* mit einer hilfreichen Fehlermeldung. Die `Errors`-Klasse befindet sich bereits in Ihrem Projekt, aber Sie brauchen sie im Moment nicht zu verstehen.

Aufgabe 3: Executable Graph

In dieser Aufgabe verwenden wir gerichtete azyklische Graphen, um Programme zu repräsentieren. Der Programmzustand ist dabei immer durch ein Tupel $(sum, counter)$ gegeben, wobei *sum* und *counter* ganze Zahlen sind. Programmzustände werden durch `ProgramState`-Objekte modelliert, wobei `ProgramState.getSum()` (bzw. `ProgramState.getCounter()`) dem ersten Element (bzw. dem zweiten Element) des Tupels entspricht.

Eine Ausführung des Programms manipuliert den Programmzustand und das Resultat eines Programms ist gegeben durch den erreichten Programmzustand, nachdem alle Operationen im Programm ausgeführt wurden. Programme können nichtdeterministisch sein: Das heisst, für ein einzelnes Programm kann es für den gleichen Startzustand mehrere Programmausführungen geben, welche zu unterschiedlichen Resultaten führen.

Knoten in Graphen werden durch `Node`-Objekte modelliert. `Node.getSubnodes()` gibt die Kinderknoten als ein Array zurück (m ist genau dann ein Kinderknoten von n, wenn es eine ausgehende gerichtete Kante von n zu m gibt). Wir unterscheiden drei Arten von Knoten, wobei die Methode `Node.getType()` die Knotenart als `String` zurückgibt. Um ein Programm, welches durch den Knoten `n` repräsentiert wird, auszuführen, muss man den "Knoten `n` ausführen". Wir beschreiben nun die drei Knotenarten und jeweils die Ausführung der Knoten:

Aufgabe 3: Executable Graph

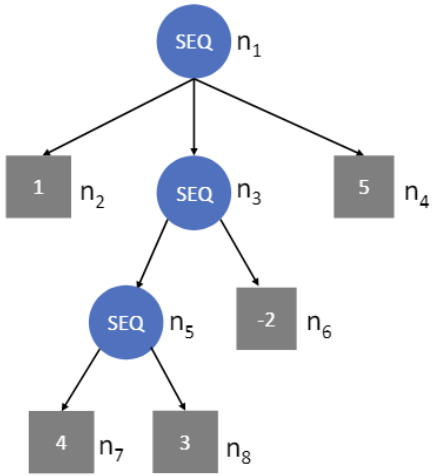
1. Additionsknoten (`Node.getType()` ist "ADD"): Solche Knoten besitzen einen Additionswert a gegeben durch `Node.getValue()` (eine ganze Zahl) und bei der Ausführung dieses Knotens wird der Programmzustand von $(sum, counter)$ zu $(sum + a, counter + 1)$ aktualisiert. Die Kinderknoten von solchen Knoten werden bei der Ausführung ignoriert.

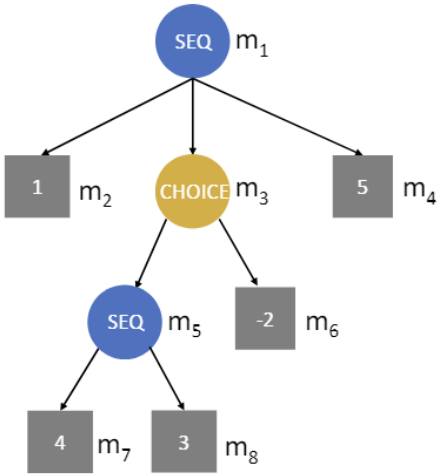
2. Sequenzknoten (`Node.getType()` ist "SEQ"): Bei der Ausführung eines Sequenzknoten `n` werden die Kinderknoten von `n` nacheinander ausgeführt. Die Reihenfolge in welcher die Kinderknoten ausgeführt werden spielt keine Rolle, da der erreichte Programmzustand für jede Reihenfolge gleich ist. `Node.getValue()` ist irrelevant.

3. Auswahlknoten (`Node.getType()` ist "CHOICE"): Bei der Ausführung eines Auswahlknoten `n` wird ein beliebiger Kinderknoten von `n` ausgewählt und ausgeführt. `Node.getValue()` ist irrelevant. Diese Knoten führen zu Nichtdeterminismus.

Sie dürfen davon ausgehen, dass Sequenz- und Auswahlknoten immer mindestens einen Kinderknoten haben, und dass es zwischen zwei Knoten immer höchstens einen Pfad gibt. Die folgende Abbildung zeigt zwei Beispielgraphen, wobei Knoten mit der Beschriftung "SEQ" (bzw. "CHOICE") Sequenzknoten (bzw. Auswahlknoten) entsprechen und die Zahlen in Additionsknoten den Additionswerten entsprechen.

Aufgabe 3: Executable Graph





Aufgabe 4: Energiespiel

In dieser Aufgabe üben Sie den Umgang mit Enums. Dafür haben Sie einen Ordner `EnergieSpiel` mit drei Klassen `GameApp`, `Game` und `Player`, sowie ein Enum `Character`. Diese sind bereits so implementiert, dass alles funktioniert. Die Klasse `Player` hat jedoch ein Feld `character` von Typ `String`. Java lässt also zu, dass in diesem Feld ein beliebiger String abgespeichert werden kann. Das Spiel hat aber eigentlich nur genau drei Möglichkeiten: `HONEST`, `TRICKSTER` oder `SORCEROR`. Das Enum `Character` mit diesen drei Optionen existiert bereits. Ändern Sie den Typ des Feldes zu `Character` und passen Sie den Code in allen drei Klassen so an, dass die Charakter-Logik überall den Typ `Character` statt `String` verwendet.

Aufgabe 5: Timed Bonus

Die Bonusaufgabe für diese Übung wird erst am Dienstag Abend der Folgewoche (also am 19. 11.) um 17:00 Uhr publiziert und Sie haben dann 2 Stunden Zeit, diese Aufgabe zu lösen. Der Abgabetermin für die anderen Aufgaben ist wie gewohnt am Dienstag Abend um 23:59. Bitte planen Sie Ihre Zeit entsprechend.
Checken Sie mit Eclipse wie bisher die neue Übungs-Vorlage aus. Importieren Sie das Eclipse-Projekt wie bisher.

Nachbesprechung

Aufgabe 1: Close Neighbors

In den Testcases hat es einen Fehler! (siehe Webseite für die korrigierten Tests)

Schreiben Sie ein Programm, welches für eine sortierte Folge X von `int`-Werten $(x_1, x_2, ..., x_n)$ und einen `int`-Wert `key` die drei unterschiedlichen Elemente x_a, x_b und x_c aus X zurückgibt, die dem Wert `key` am nächsten sind. Für x_a, x_b und x_c muss gelten, dass $|\texttt{key} - x_a| \leq |\texttt{key} - x_b| \leq |\texttt{key} - x_c| \leq |\texttt{key} - x_i|$ für alle $i \neq a, b, c$ und dass $x_a \neq x_b \neq x_c \neq x_a$. Wenn die drei Werte nicht eindeutig bestimmt sind, dann ist jede Lösung zugelassen, die die obige Bedingung erfüllt.

Beispiele:
Die nächsten Nachbarn für `key == 5` in $(1, 4, 5, 7, 9, 10)$ sind $5, 4, 7$.
Die nächsten Nachbarn für `key == 5` in $(1, 4, 5, 6, 9, 10)$ sind $5, 4, 6$ oder $5, 6, 4$.
Die nächsten Nachbarn für `key == 10` in $(1, 4, 5, 6, 9, 10)$ sind $10, 9, 6$.

Implementieren Sie die Berechnung in der Methode `int[] neighbor(int[] sequence, int key)`, welche sich in der Klasse `Neighbor` befindet. Die Deklaration der Methode ist bereits vorgegeben. Sie können davon ausgehen, dass das Argument `sequence` nicht `null` ist, sortiert ist, nur unterschiedliche Elemente enthält, und mindestens drei Elemente enthält. Denken Sie daran, dass der Wert `key` nicht unbedingt in der Folge X auftritt. Sie dürfen das Eingabearray `input` nicht ändern.

In der `main` Methode der Klasse `Neighbor` finden Sie die oberen Beispiele als kleine Tests, welche Beispiel-Aufrufe zur `neighbor`-Methode machen und welche Sie als Grundlage für weitere Tests verwenden können. In der Datei `NeighborTest.java` geben wir die gleichen Tests zusätzlich auch als JUnit Test zur Verfügung. Sie können diese ebenfalls nach belieben ändern. Es wird *nicht* erwartet, dass Sie für diese Aufgabe den JUnit Test verwenden.

Aufgabe 2: Loop-Invariante

1. Gegeben sind die Precondition und Postcondition für das folgende Programm

```
public int compute(int n) {
    // Precondition:  n >= 0
    int x;
    int res;

     x = 0;
    res = x;

    // Loop Invariante:
    while (x <= n) {
        res = res + x;
        x = x + 1;
    }
    // Postcondition:  res == ((n + 1) * n) / 2
    return res;
}
```

Schreiben Sie die Loop Invariante in die Datei “LoopInvariante.txt”.

Aufgabe 2: Loop-Invariante

2. Gegeben sind die Precondition und Postcondition für das folgende Programm.

```
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = 0;
    res = b;

    // Loop Invariante:
    while (x < a) {
        res = res - 1;
        x = x + 1;
    }
    // Postcondition:  res == b - a
    return res;
}
```

Schreiben Sie die Loop Invariante in die Datei “LoopInvariante.txt”.

Aufgabe 3: Bills

In dieser Aufgabe sollen Sie einen Teil des Systems implementieren, das für den lokalen Stromversorger die Rechnungen erstellt.

Vervollständigen Sie die `process`-Methode in der Klasse `Bills`. Die Methode hat zwei Argumente: einen `Scanner`, von dem Sie den Inhalt der Eingabedatei lesen sollen, und einen `PrintStream`, in welchen Sie die unten beschriebenen Informationen schreiben.

Ihr Programm muss nur korrekt formatierte Eingabedateien unterstützen. Ein Beispiel einer solchen Datei finden Sie im Projekt unter dem Namen "Data.txt". Exceptions im Zusammenhang mit Ein- und Ausgabe können Sie ignorieren.

Eine valide Eingabedatei enthält Zeilen, die entweder einen Tarif oder den Stromverbrauch eines Kunden beschreiben. Der Verbrauch eines Kunden ist niemals grösser als 100000 Kilowattstunden.

Eine Tarifbeschreibung hat folgendes Format:

$$\texttt{Tarif}\,␣\,n\,␣\,l_1\,␣\,p_1\,\ldots\,l_n\,␣\,p_n$$

Aufgabe 4: Minesweeper (Bonus)

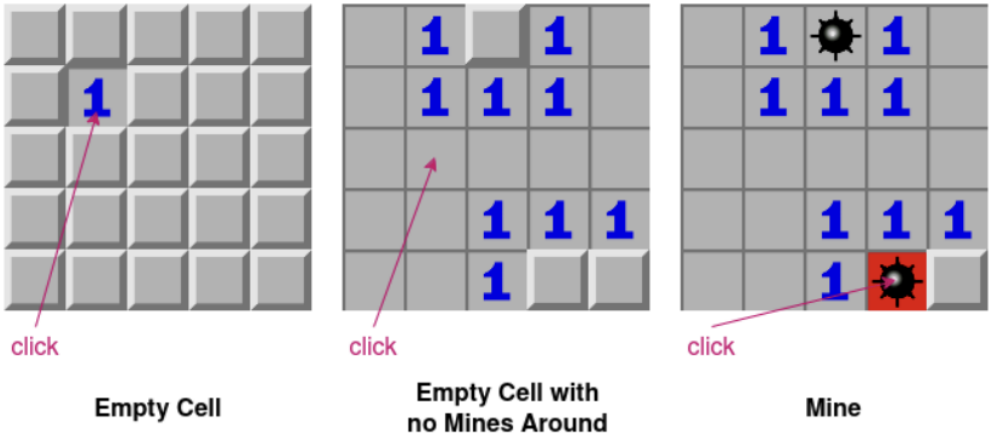


Abbildung 1: Spielbretter nach dem ersten, zweiten und dritten Klick von links nach rechts.

Achtung: Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im `www.vvz.ethz.ch`). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihr Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin. Auch wenn Sie vor der Deadline committen, aber nach der Deadline pushen, gilt dies als eine zu späte Abgabe. Bitte lesen Sie zusätzlich die allgemeinen Regeln.

Kahoot

https://create.kahoot.it/share/21-u07-eprog-mschoeb/19bdfeae-3305-4cb8-8af0-2fc07ae3e127