252-0027

Einführung in die Programmierung

Übungen

Woche 9: Invarianten, Graphenaufgaben

Henrik Pätzold
Departement Informatik
ETH Zürich

Vorbesprechung & Grading

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.

Sehr Wichtig 5/5

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.

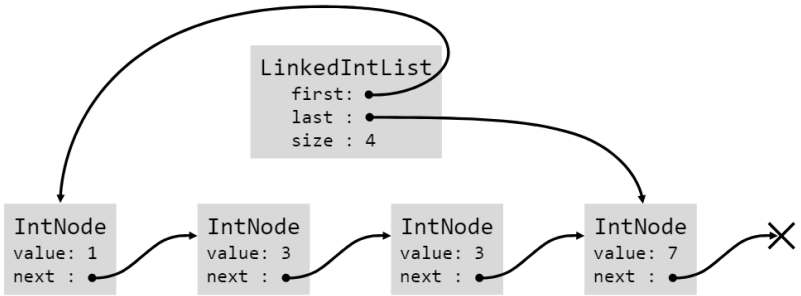


Abbildung 1: Verkettete Liste mit Werten 1, 3, 3, 7.

Eher Wichtig 4/5

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:

Gute Übung - Tricky 3/5

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.

Medium aber sehr einfach 3/5

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.

Räumt euch genug Zeit ein!
Ihr seid verantwortlich dafür, dass das Projekt ordentlich funktioniert.

Invarianten mit Rezept**

**Es gibt keine perfekten Rezepte ☹

Invarianten mit Rezept

Loop Condition und Terminierung kombinieren

Postcondition und Loopbody kombinieren

Zusatz um undefiniertes behaviour zu verhindern

Invarianten mit Rezept

```
public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}
```

counter > 0 ∪ counter = 0 ➡ counter >= 0

Iteration 1: result = 1 * n
Iteration 2: result = 1 * n * (n-1)
Iteration i: result = n! * (n-1)! * … * (n-i)! ➡ result = n! / counter!

Kein Zusatz benötigt (sicher durch testen)

LI: counter >= 0 && result = n! / counter!

Invarianten mit Rezept

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

  x = 1;
  res = 0;

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

$x <= n \cup x == n+1 \rightarrow x <= n+1$

Iteration 0: res = 0*x

Iteration 1: res = 1*x

Iteration 2: res = 2*x

Iteration i: res = i*x → res = (x-1)*x

1 <= x (nicht zwingend notwendig, schadet aber auch nicht)

LI: 1 <= x && x <= n + 1 && res == (x - 1) * x

Invarianten mit Rezept

```
public int compute(int a, int b) {

   // Precondition:  a >= 0

   int x;
   int res;

   x = a;
   res = b;

   // Loop Invariante:

   while (x > 0) {
       x = x - 1;
       res = res + 1;
   }

  // Postcondition:  res = a + b
  return res;
}
```

x > 0 ∪ x = 0 ➡ x >= 0

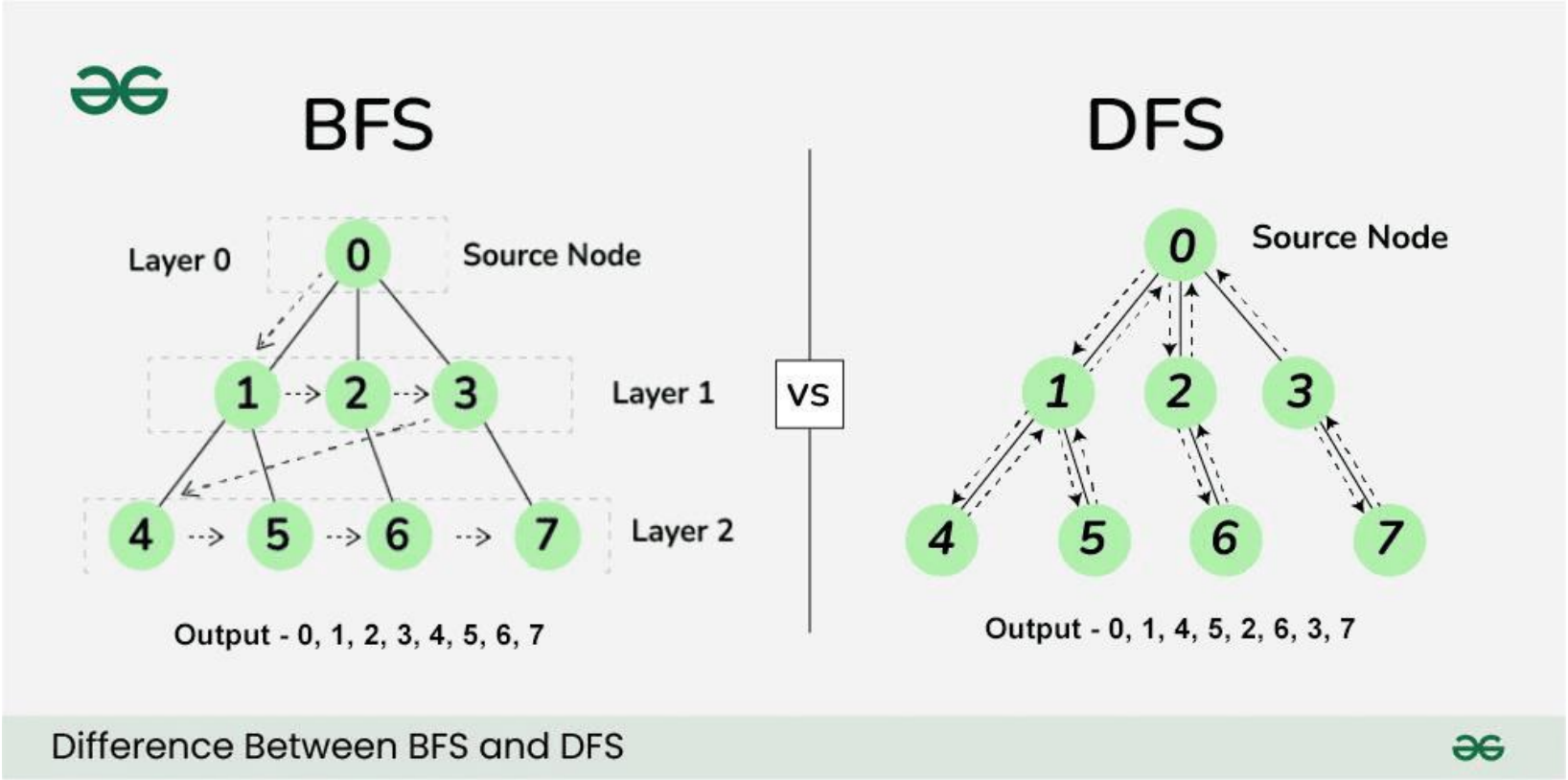
Iteration 1: res = b
Iteration 2: res = b + 1
Iteration i: res = b + i ➡ res = b + (a-x)

z.B. a >= 0 (aus der Precondition, nicht zwingend Notwendig)

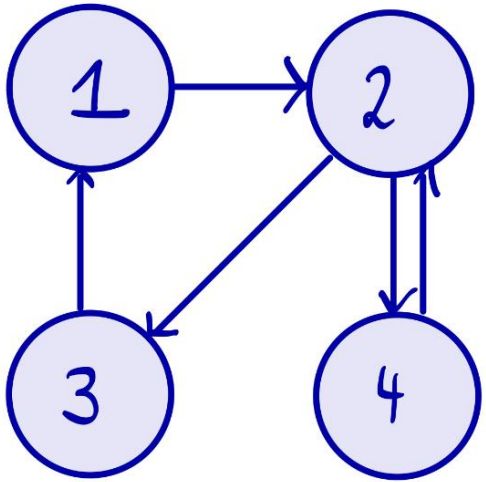
LI: res = b + (a-x) && x >= 0 && a >= 0

Graphenaufgaben

Recap BFS & DFS



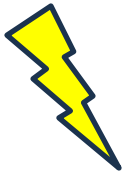
Graphen können auf verschiedenste Weisen dargestellt werden



Wir haben folgende Kanten:
1 → 2
2 → 3
2 → 4
3 → 1
4 → 2

```
{
    1: [2],
    2: [3, 4],
    3: [1],
    4: [2]
}
```

Beispielhafte Adjazenz Liste



Wichtig ist zu verstehen, wie der Graph aufgebaut ist, damit die Aufgaben sinnvoll gelöst werden können!

Simples Beispiel mit Klassen

```java
public class Node {
    private LinkedList<Node> children = new LinkedList<>();
    private int id;
    public Node(int identifier){
        this.id = identifier;
    }
    public LinkedList<Node> getChildren() {
        return children;
    }

    public void addChild(Node node){
        children.add(node);
    }
}
```

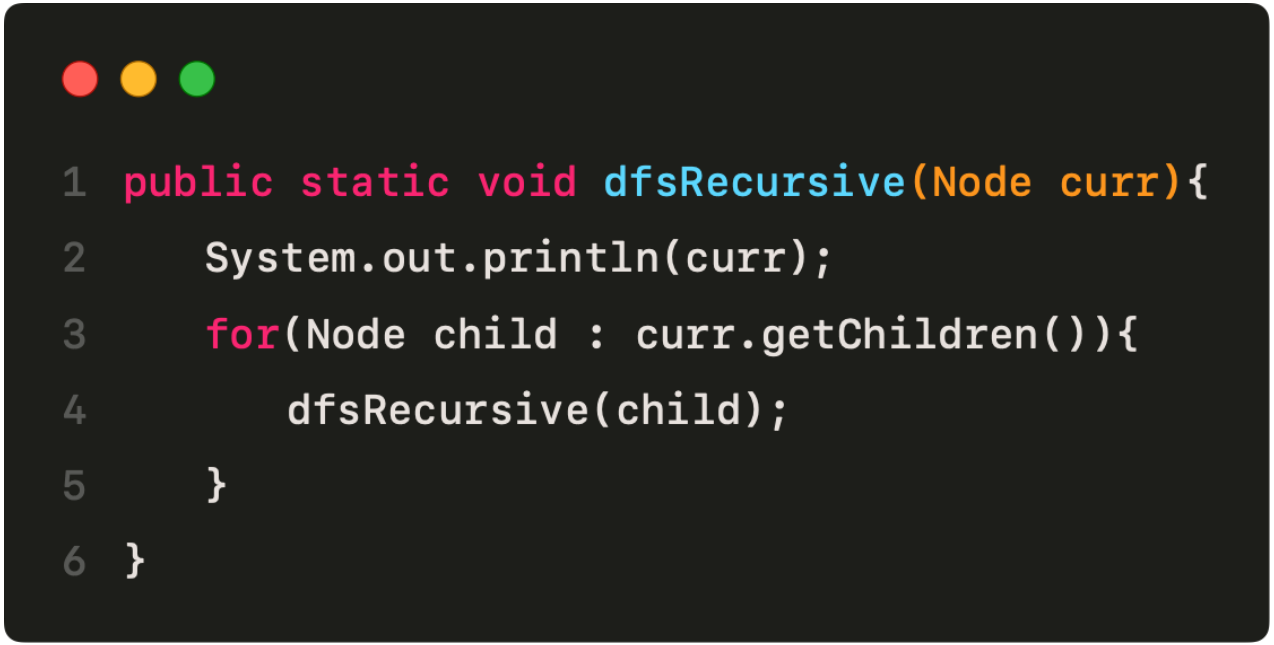
```java
public static void main(String[] args) {
    Node n1 = new Node(1);
    Node n2 = new Node(2);
    Node n3 = new Node(3);
    Node n4 = new Node(4);

    n1.addChild(n2);
    n2.addChild(n3);
    n2.addChild(n4);
    n3.addChild(n1);
    n4.addChild(n2);
}
```

DFS ist euer Freund (BFS *vereinzelt* auch)

```java
public static void dfsIterative(Node start) {
    Set<Node> visited = new HashSet<>();
    LinkedList<Node> stack = new LinkedList<>();
    stack.push(start);
    while(!stack.isEmpty()){
        Node curr = stack.pop();
        visited.add(curr);
        LinkedList<Node> children = curr.getChildren();
        for (Node child : children) {
            if(!visited.contains(child)){
                stack.push(child);
            }
        }
        System.out.println(curr);
    }
}
```

Rekursives DFS ist euer bester Freund,…



```java
public static void dfsRecursive(Node curr){
    System.out.println(curr);
    for(Node child : curr.getChildren()){
        dfsRecursive(child);
    }
}
```

Simples, rekursives DFS

…, weil es leicht anpassbar ist

```java
public static void dfsRecursive(Node curr, Set<Node> visited){
    System.out.println(curr);
    visited.add(curr);
    for(Node child : curr.getChildren()){
        if(!visited.contains(child))dfsRecursive(child, visited);
    }
}
```

Funktional identisches Verhalten zum iterativen DFS vorher aber kompakter in der Schreibweise

Zyklenvermeidung durch weitergeben nötiger Informationen als Methodenparameter

Übung macht hier wieder den Unterschied! :)

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}.

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}.

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}.

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.

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.

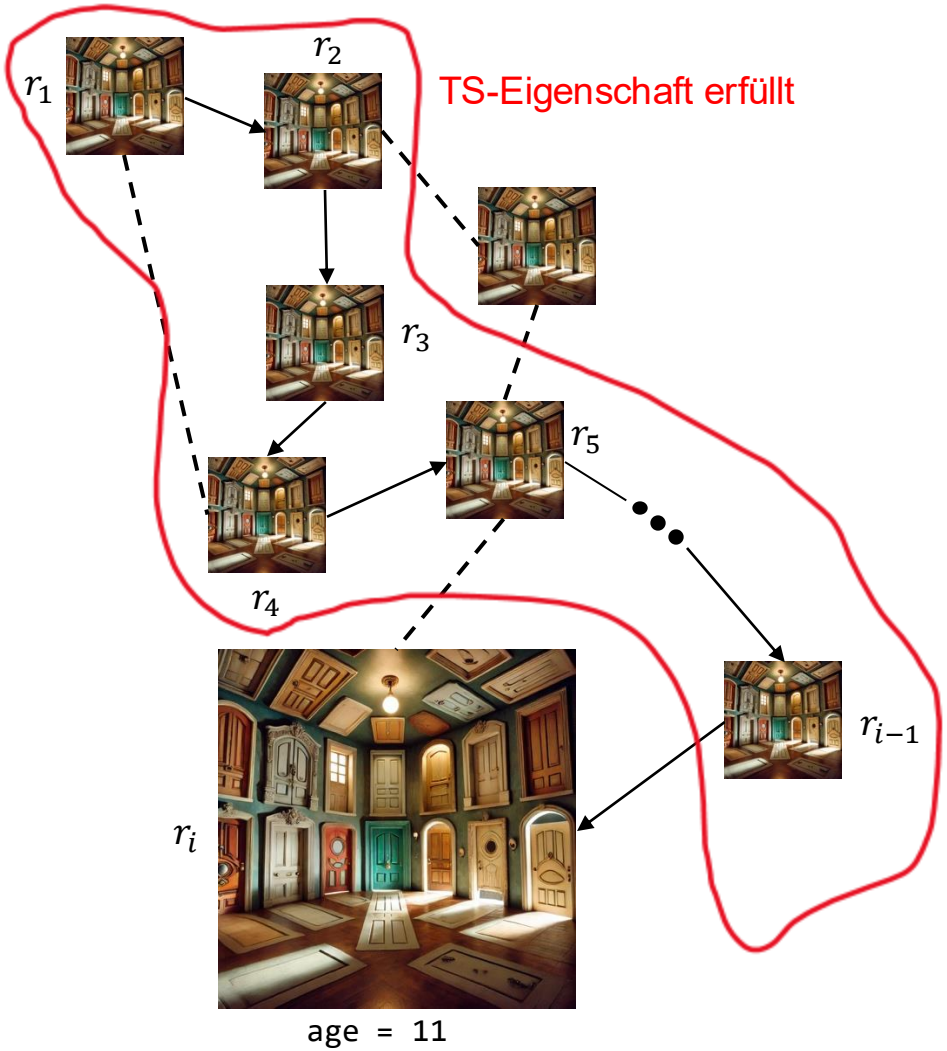
Wie lösen wir das Problem?

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

Teilprobleme identifizieren:

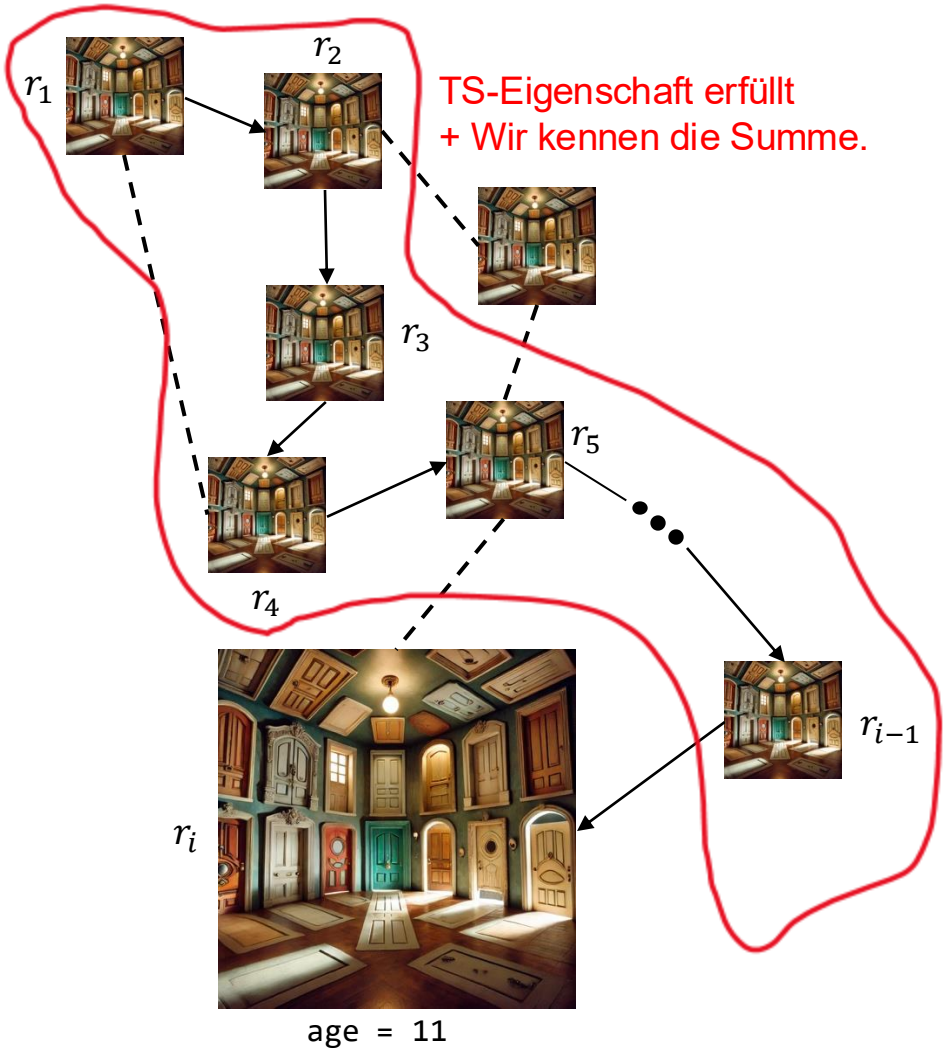
- Angenommen wir befinden uns in einem Raum r_i wie wissen wir ob der Pfad $r_1, \dots, r_i$ eine Lösung ist?

- 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 $r_1, \dots, r_i$ die TS-Eigenschaft erfüllt?
- Was wenn $r_1, \dots, r_{i-1}$ 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 $r_1, \dots, r_i$ 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(solve1(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.

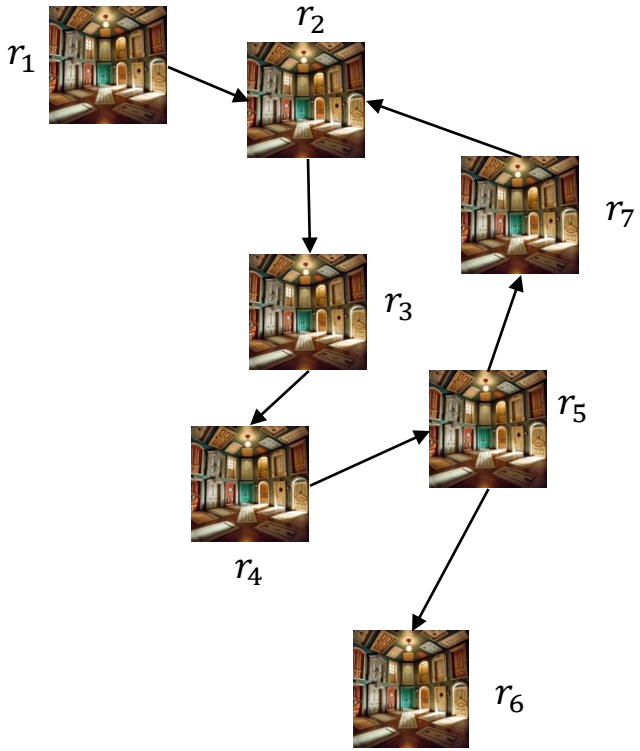
```java
public static boolean solve1(Room room, int sum) {
(…)
}

public static boolean task1(Room room) {
    return solve1(room, 0)

}
```

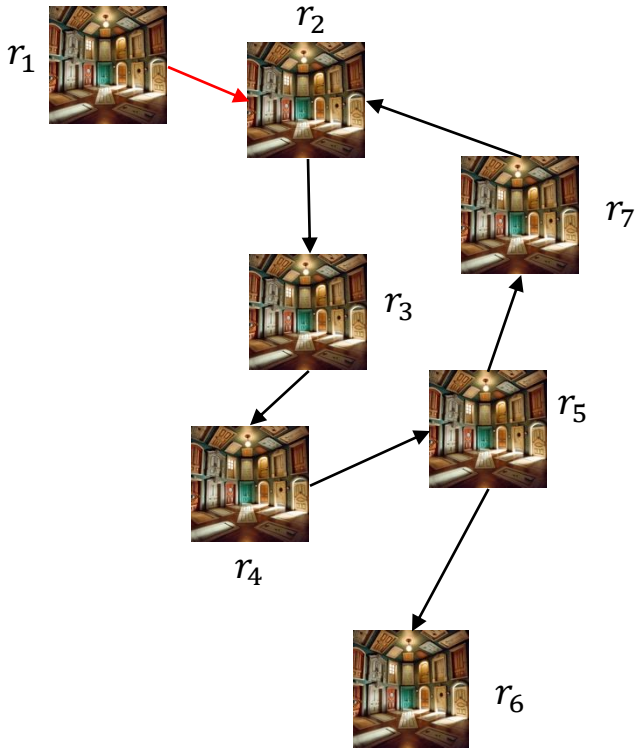
Wie nutzen wir diese Methode nun?

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



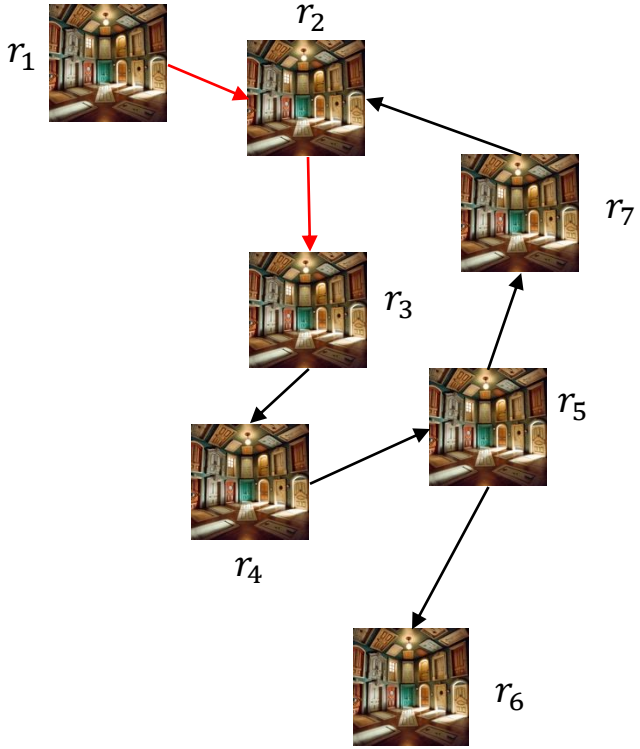
Wieso keine Zyklen?

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



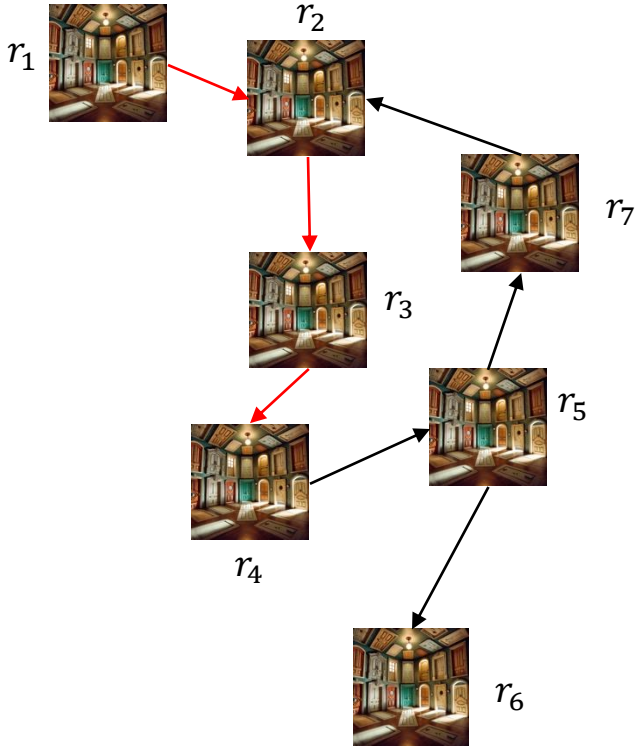
Wieso keine Zyklen?

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



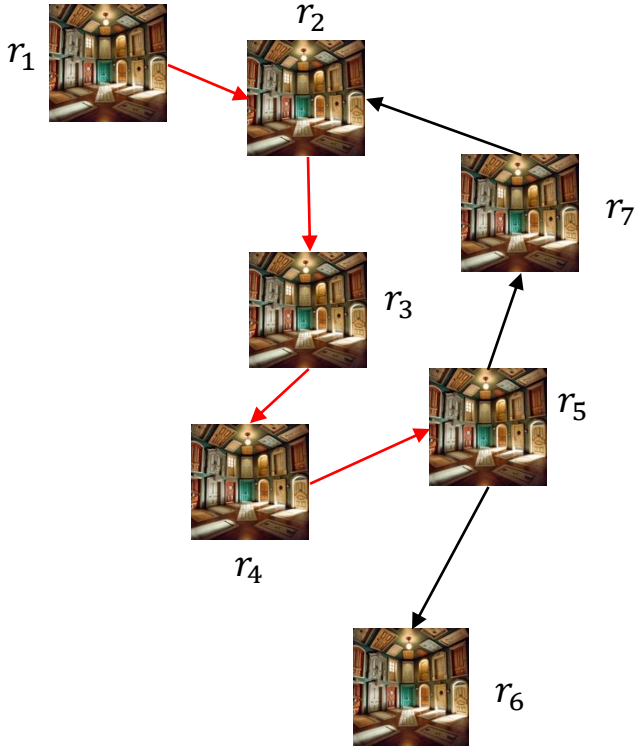
Wieso keine Zyklen?

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



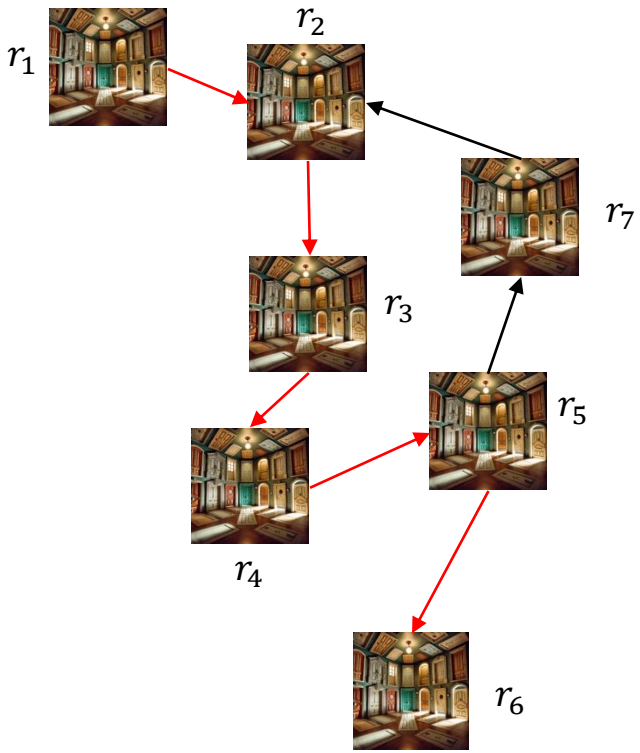
Wieso keine Zyklen?

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



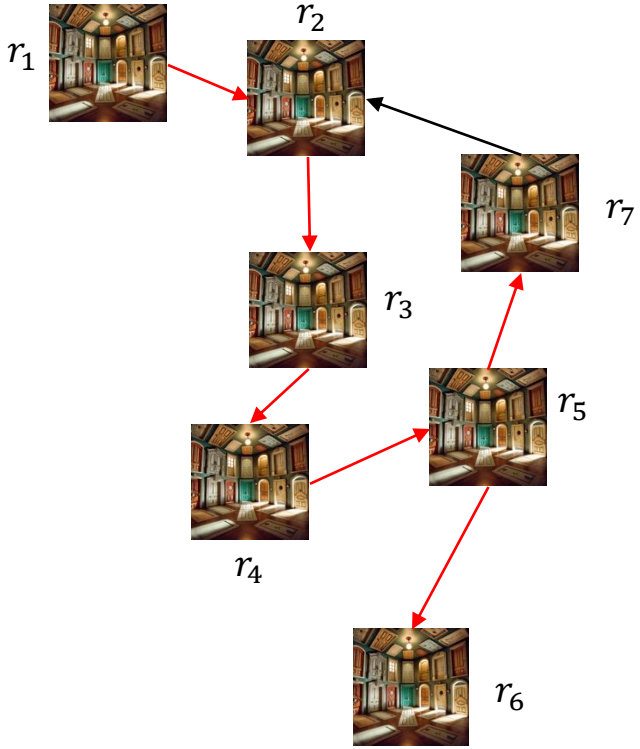
Wieso keine Zyklen?

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



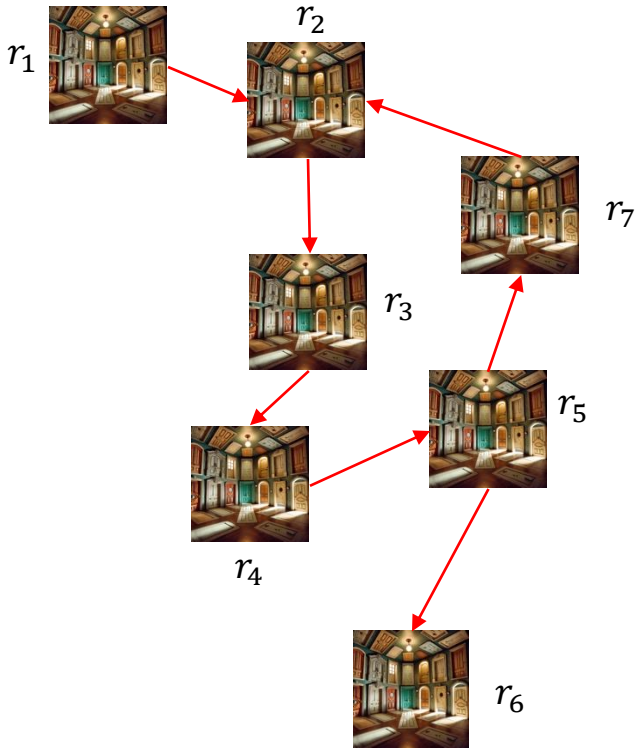
Wieso keine Zyklen?

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



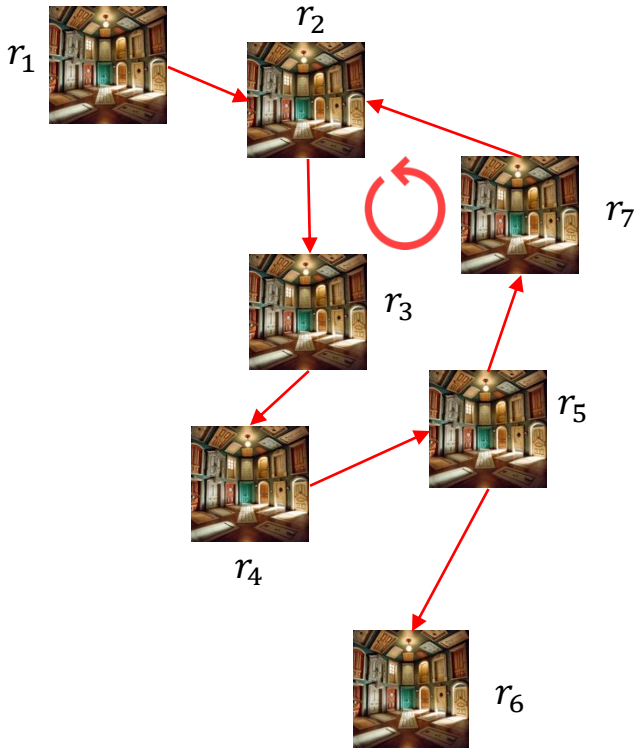
Wieso keine Zyklen?

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



Wieso keine Zyklen?

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



Wieso keine Zyklen?

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

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?

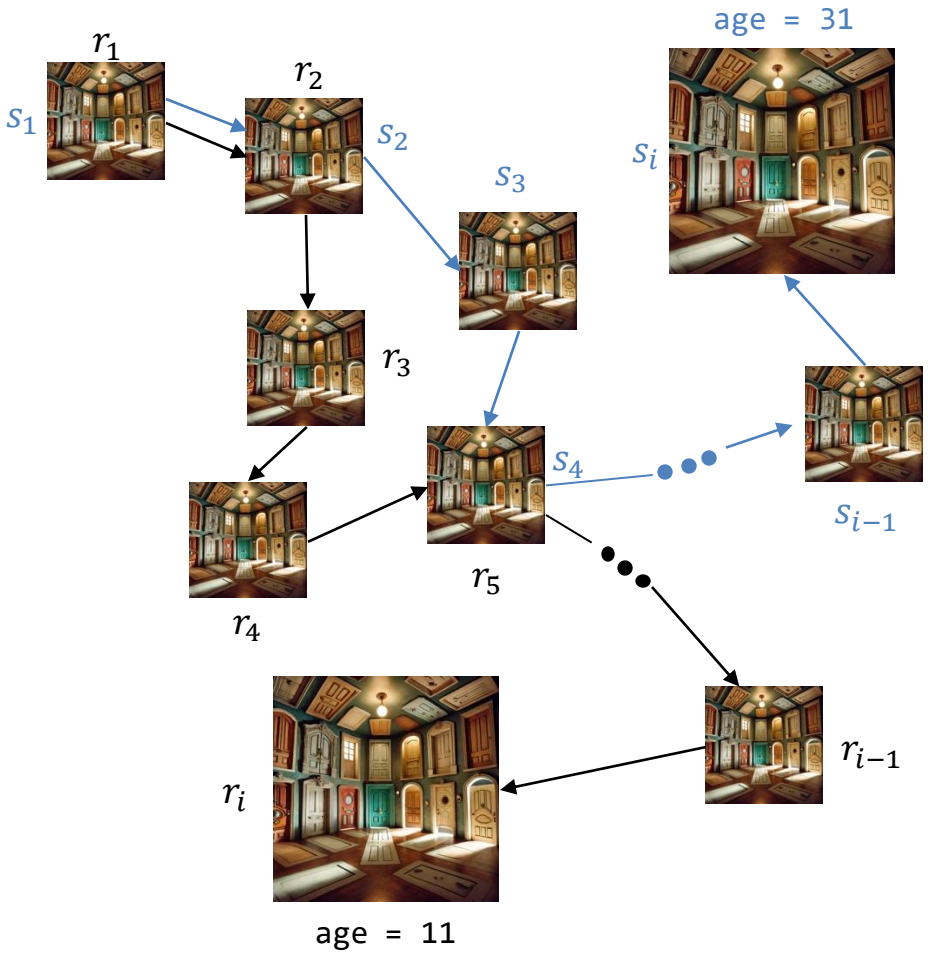
- **Rekursive Lösung:** Damit wir das Problem rekursiv lösen können, müssen wir **Teilprobleme** identifizieren.

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 erkunden 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 solve1(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

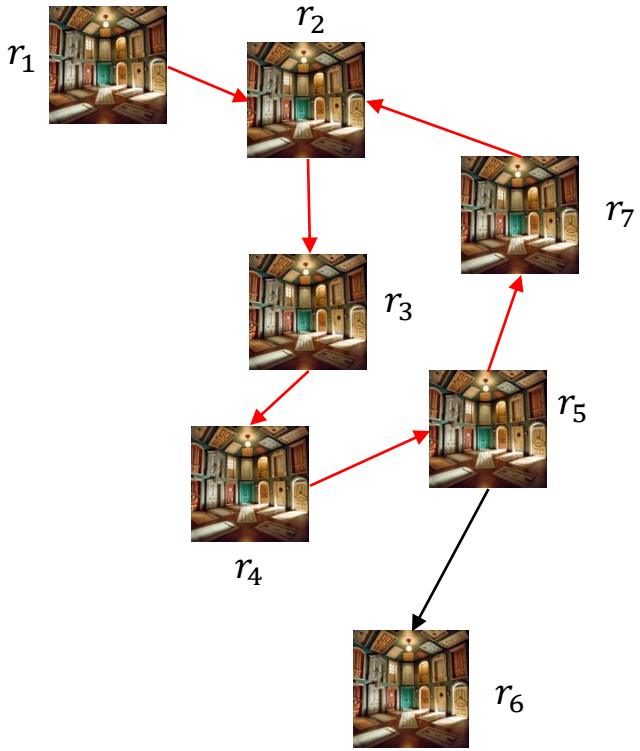
- Wir rufen `solve1` in `task1` auf mit `room` und initialer Summe `0`.
- Wir rufen `solve2` in `task2` auf mit `room` und initialem boolean Parameter `true`.

Mit gerichteten Zyklen Arbeiten

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.

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`.

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

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

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

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

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

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

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

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

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

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

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

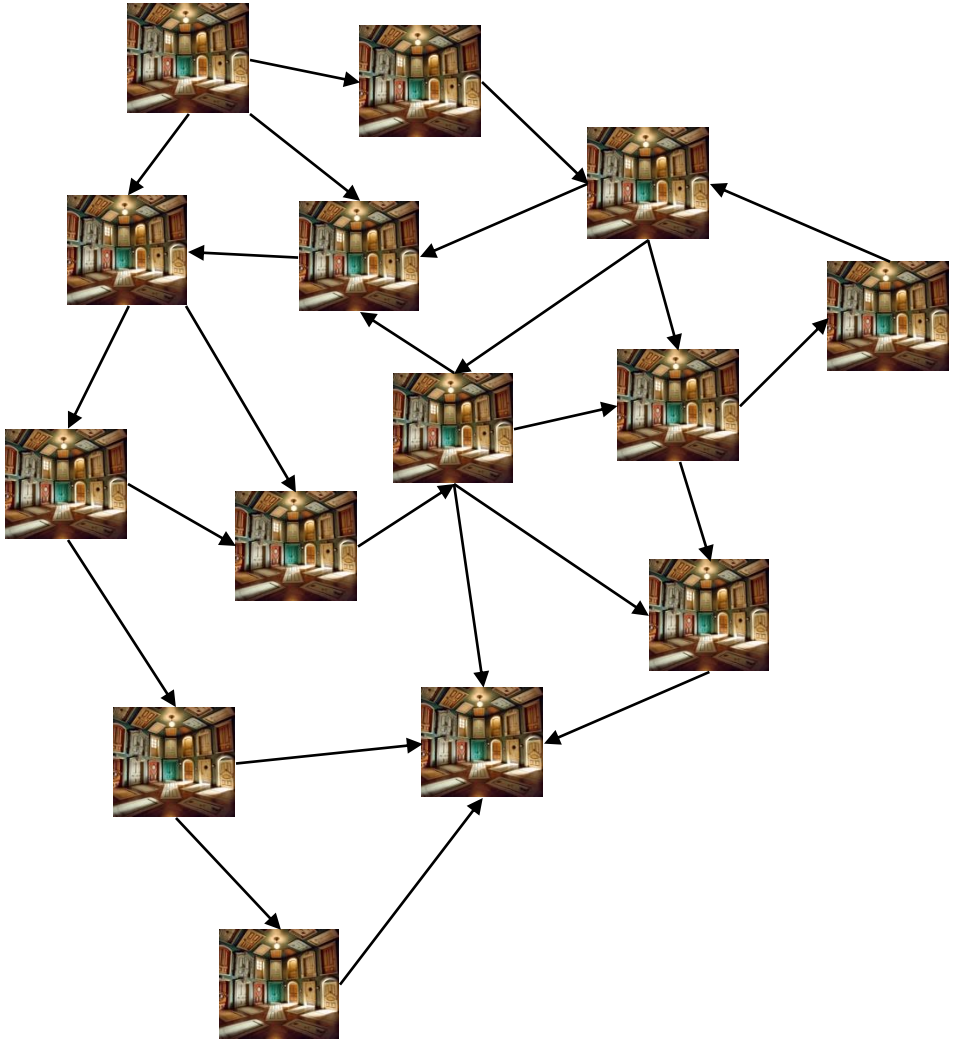
Wir führen den gleichen Code wie vorhin aus.

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

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

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

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

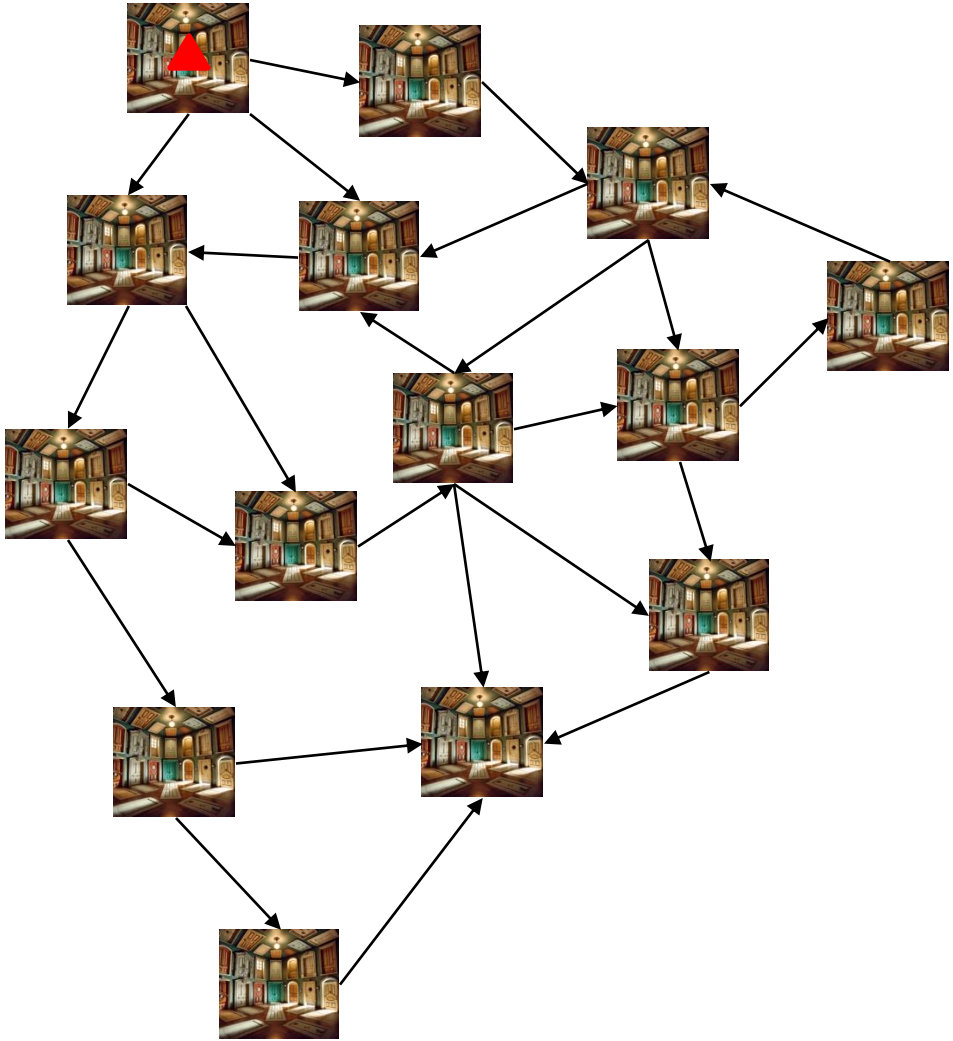


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

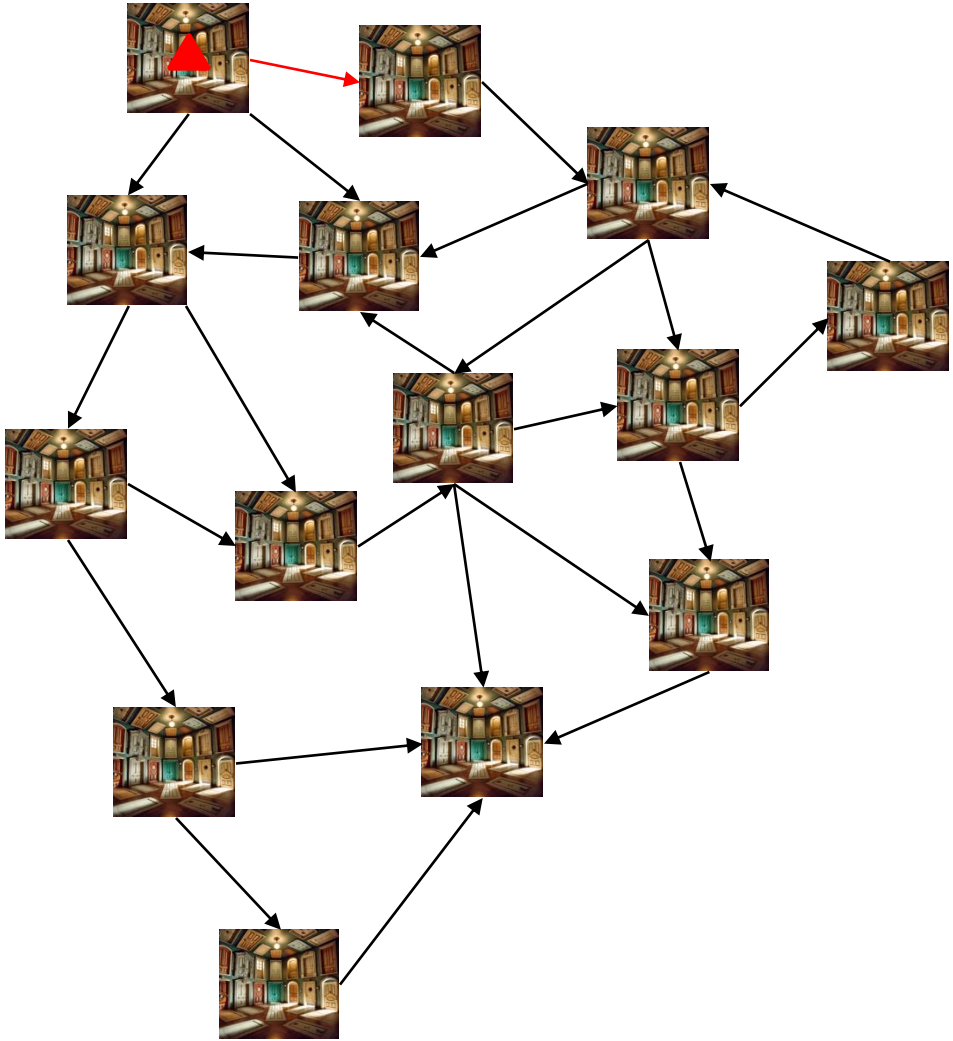


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

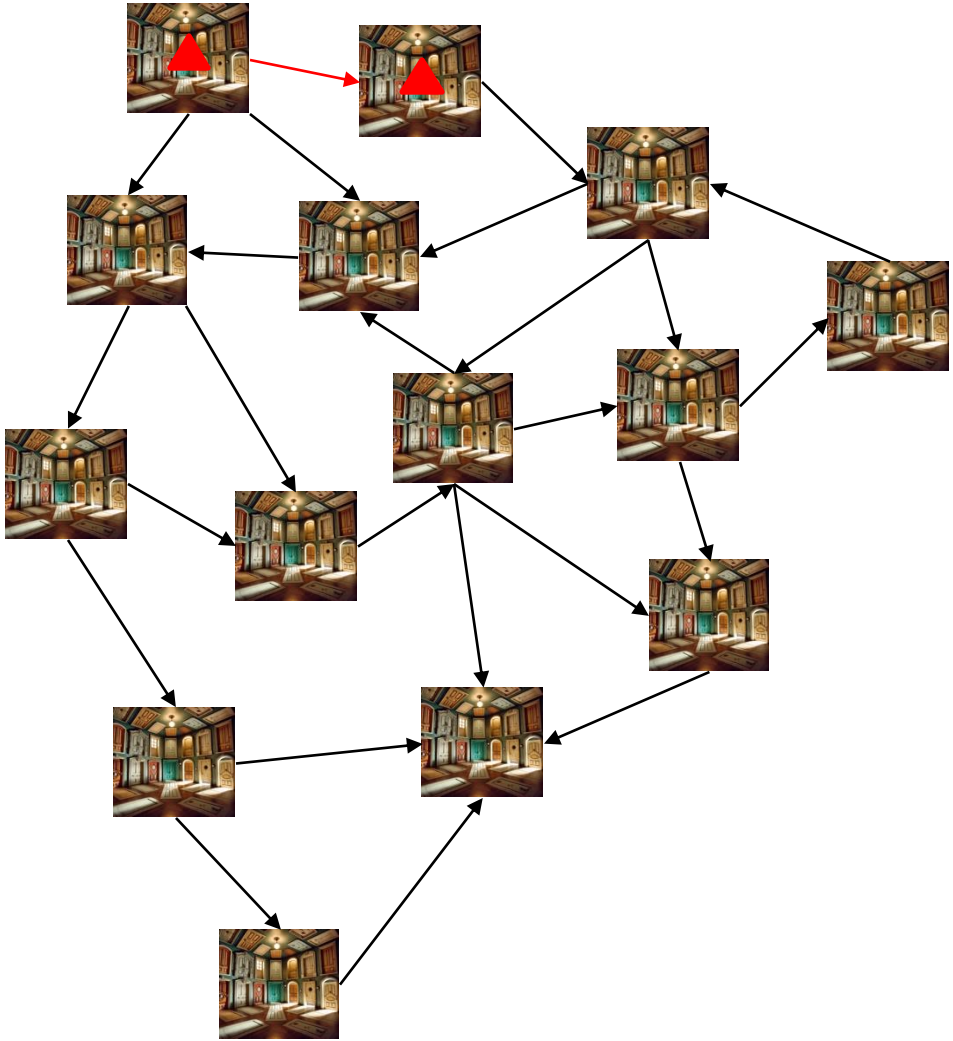


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

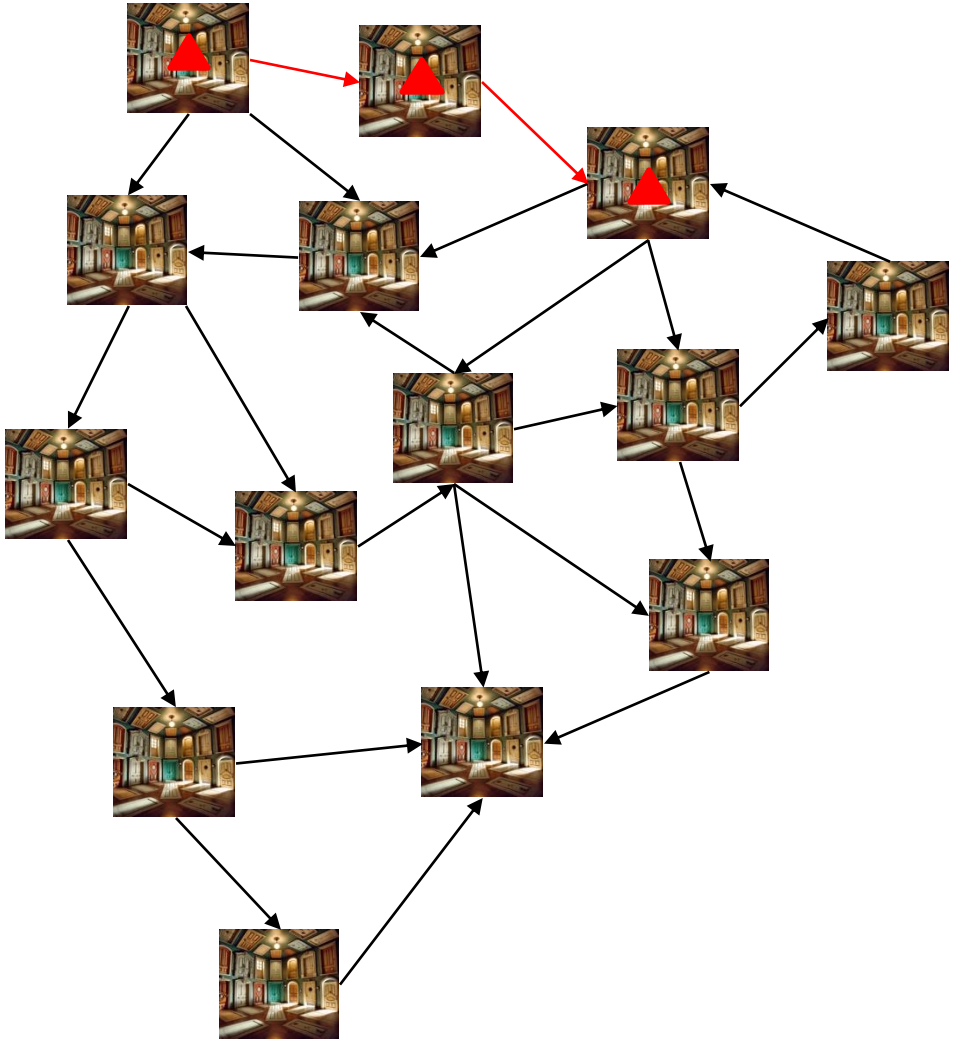


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

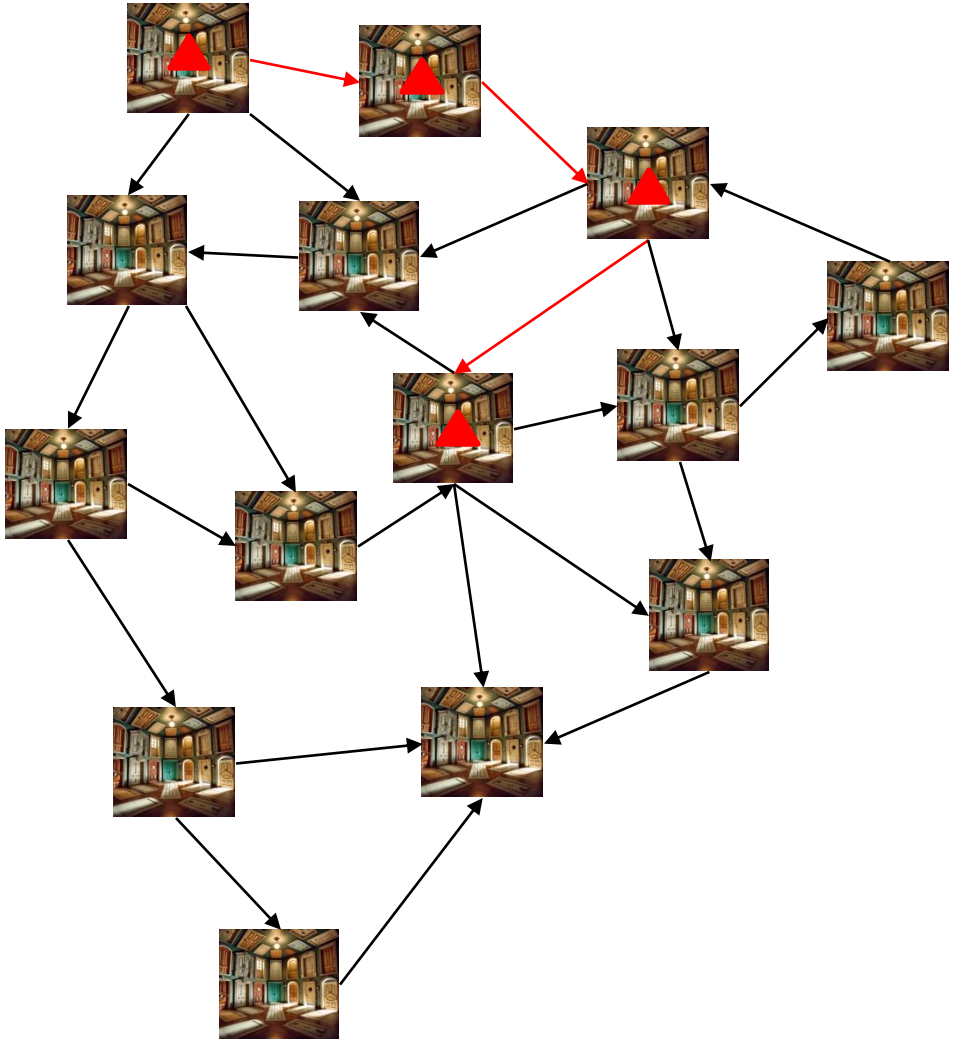


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

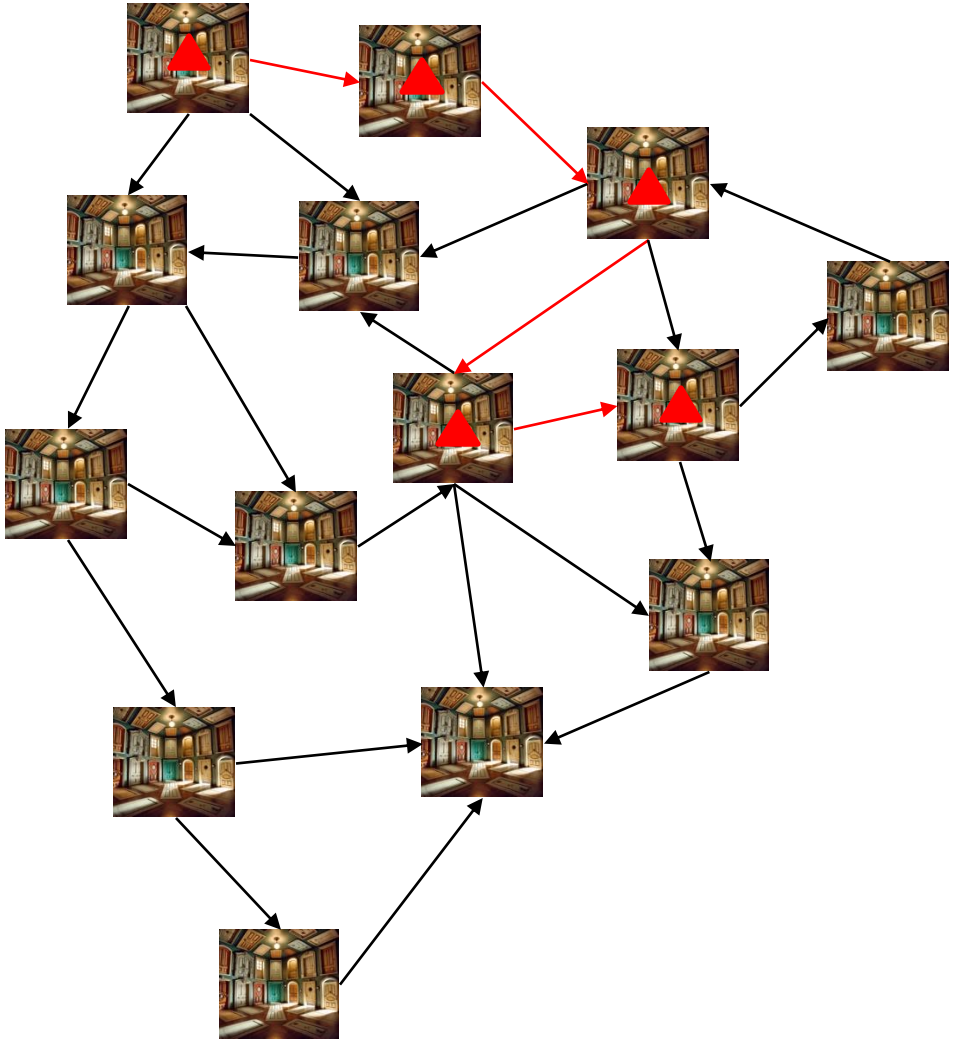


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

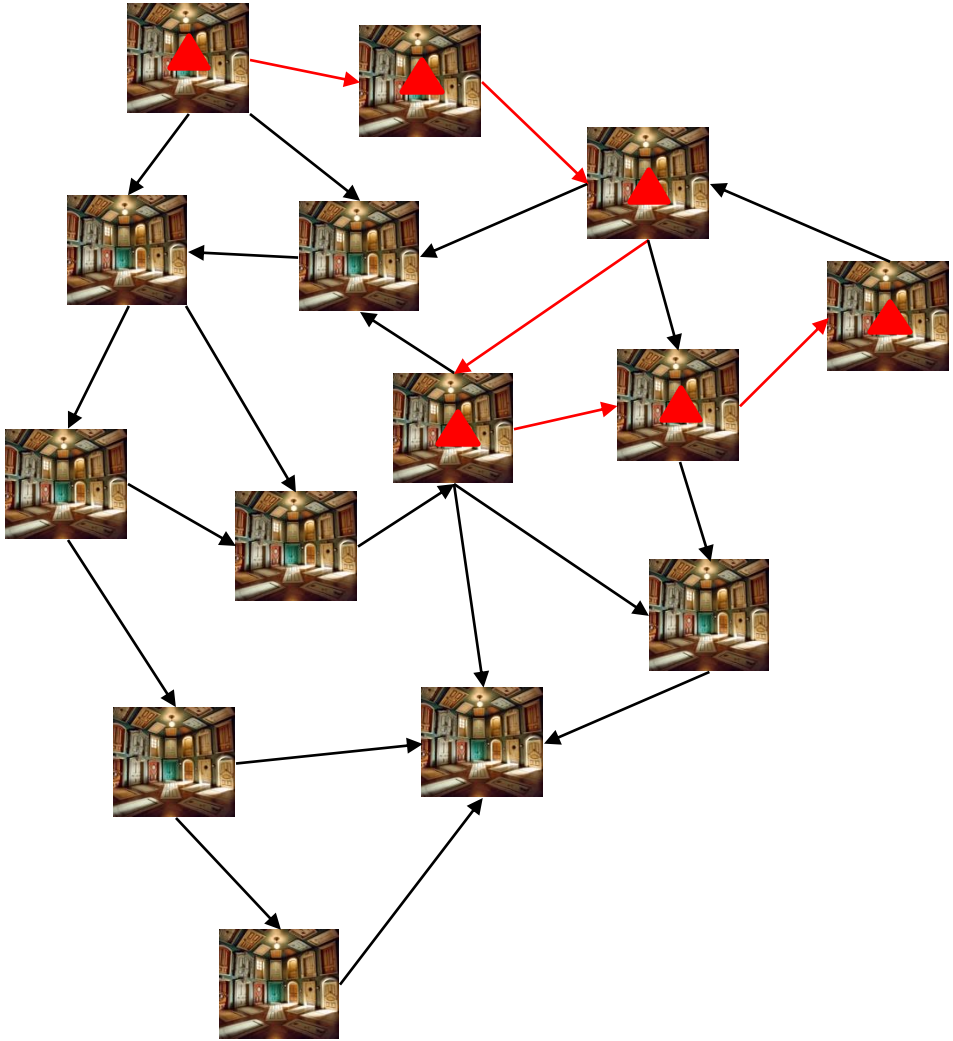


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

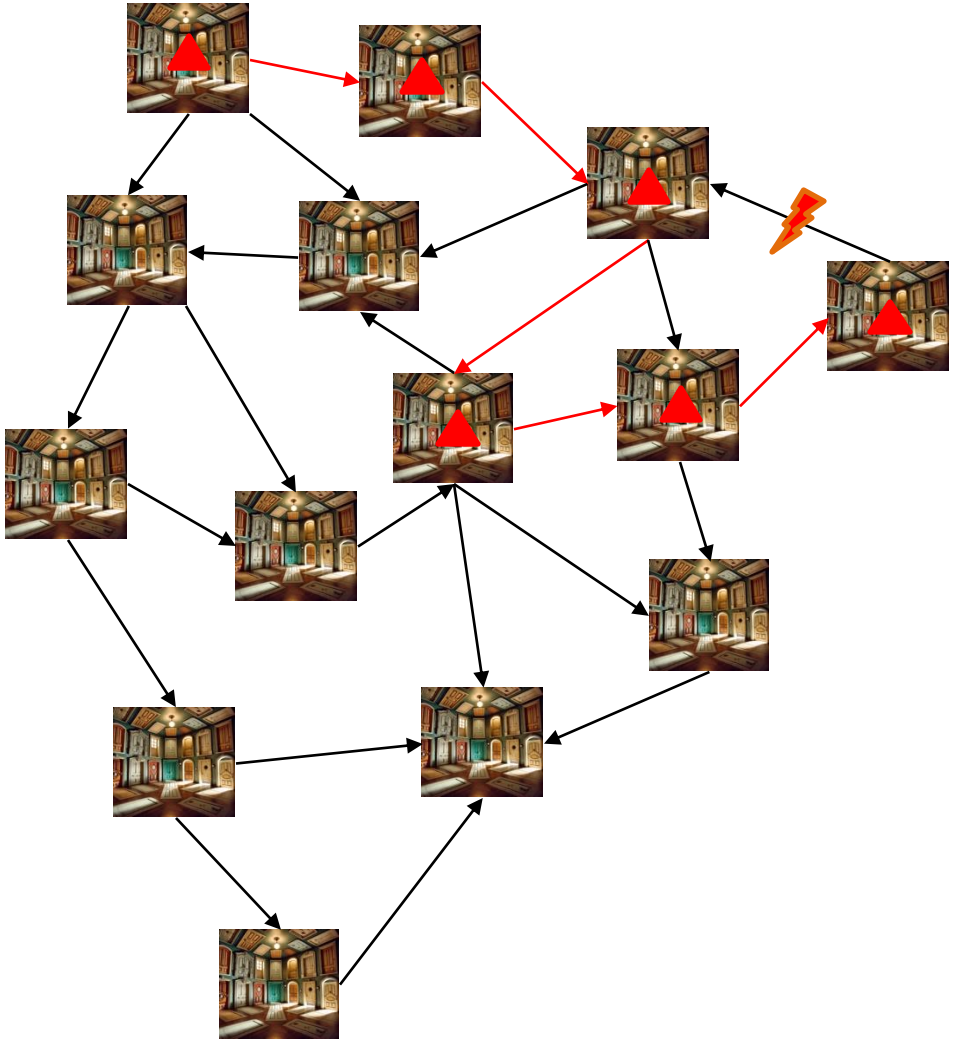


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

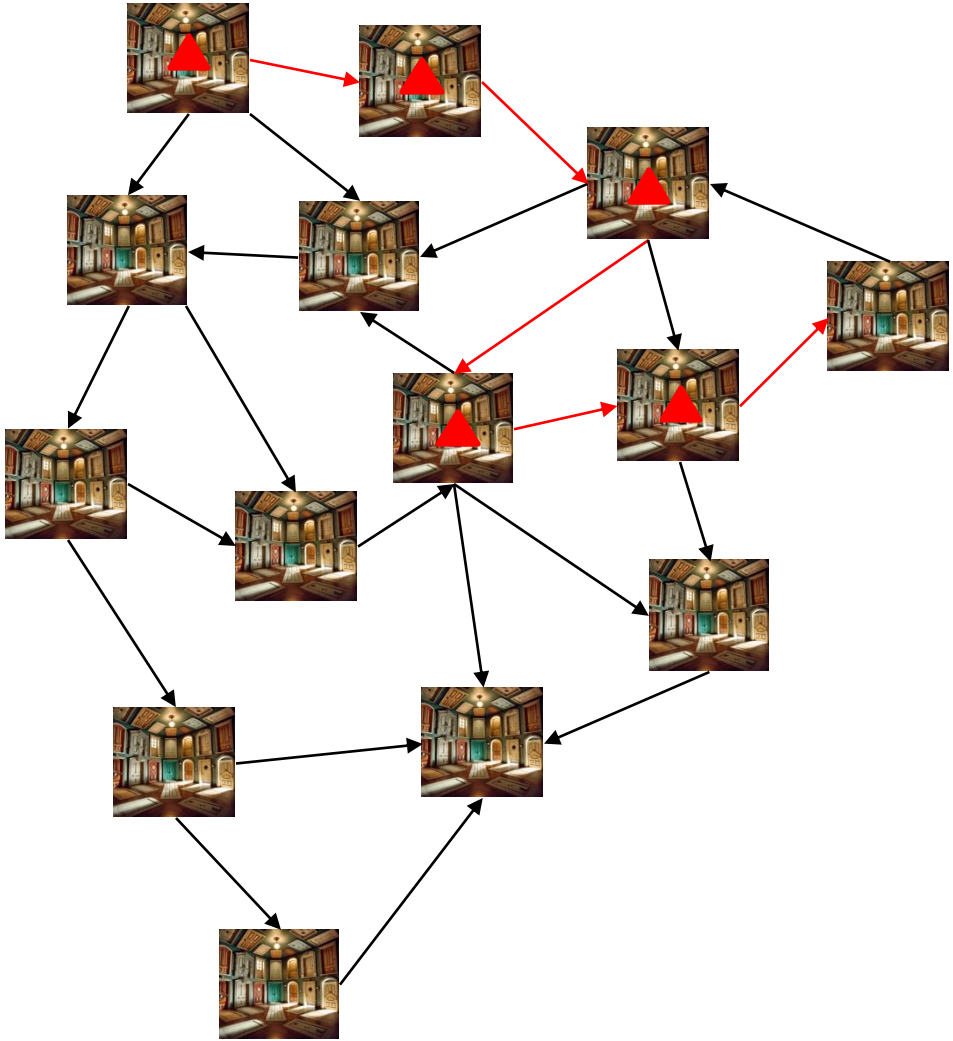


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

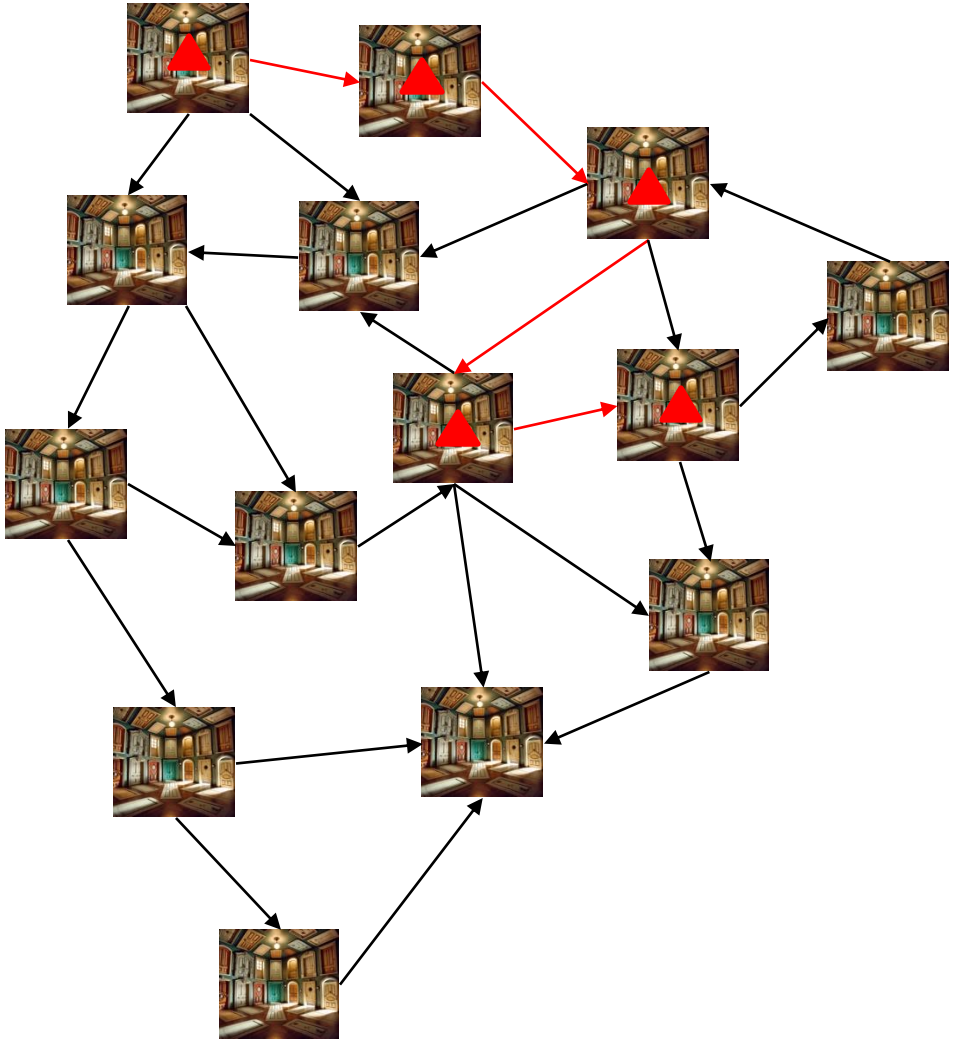


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

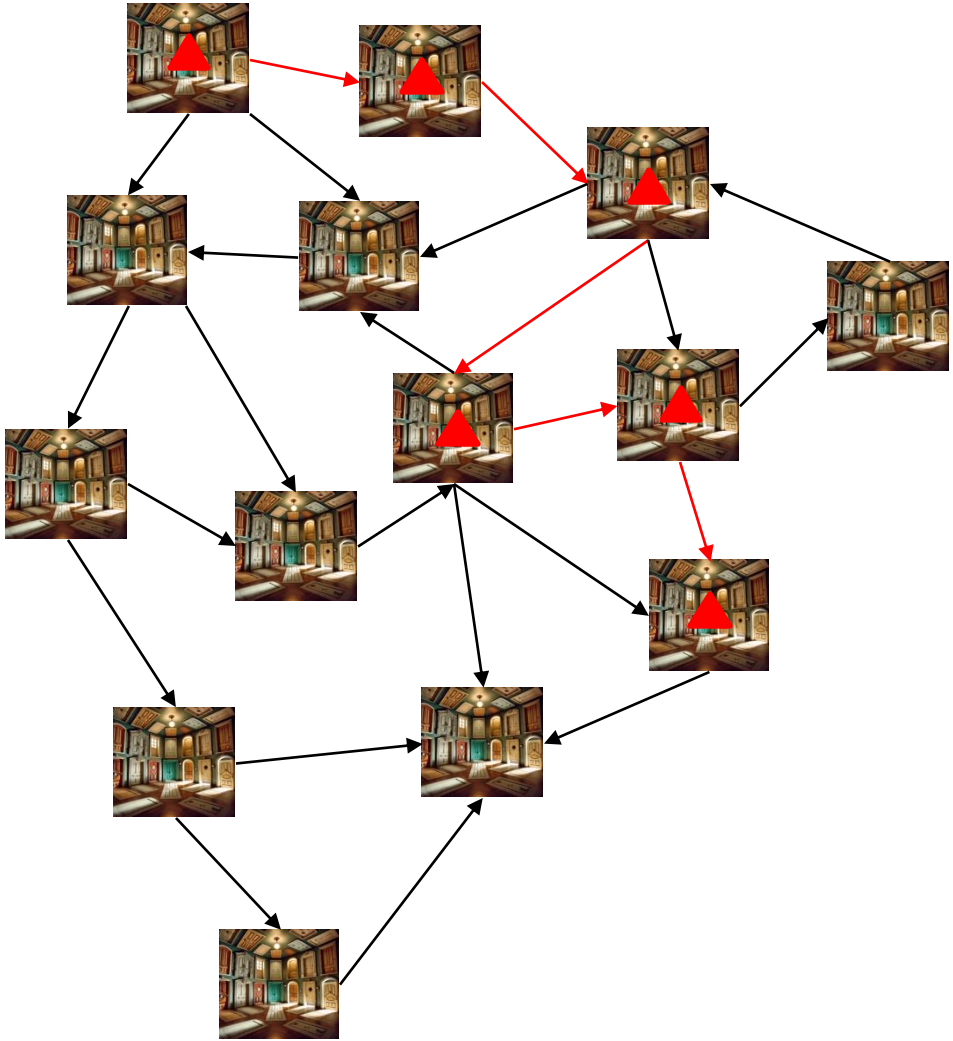


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

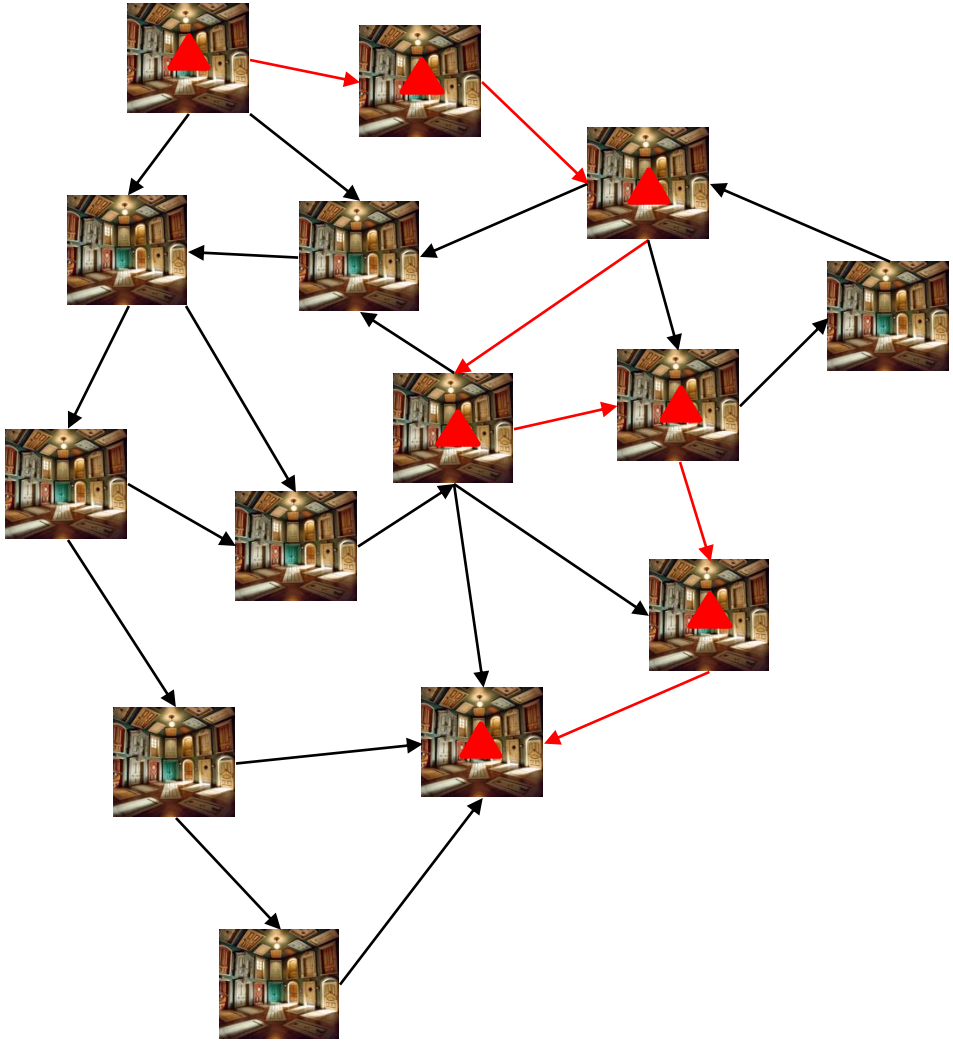


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

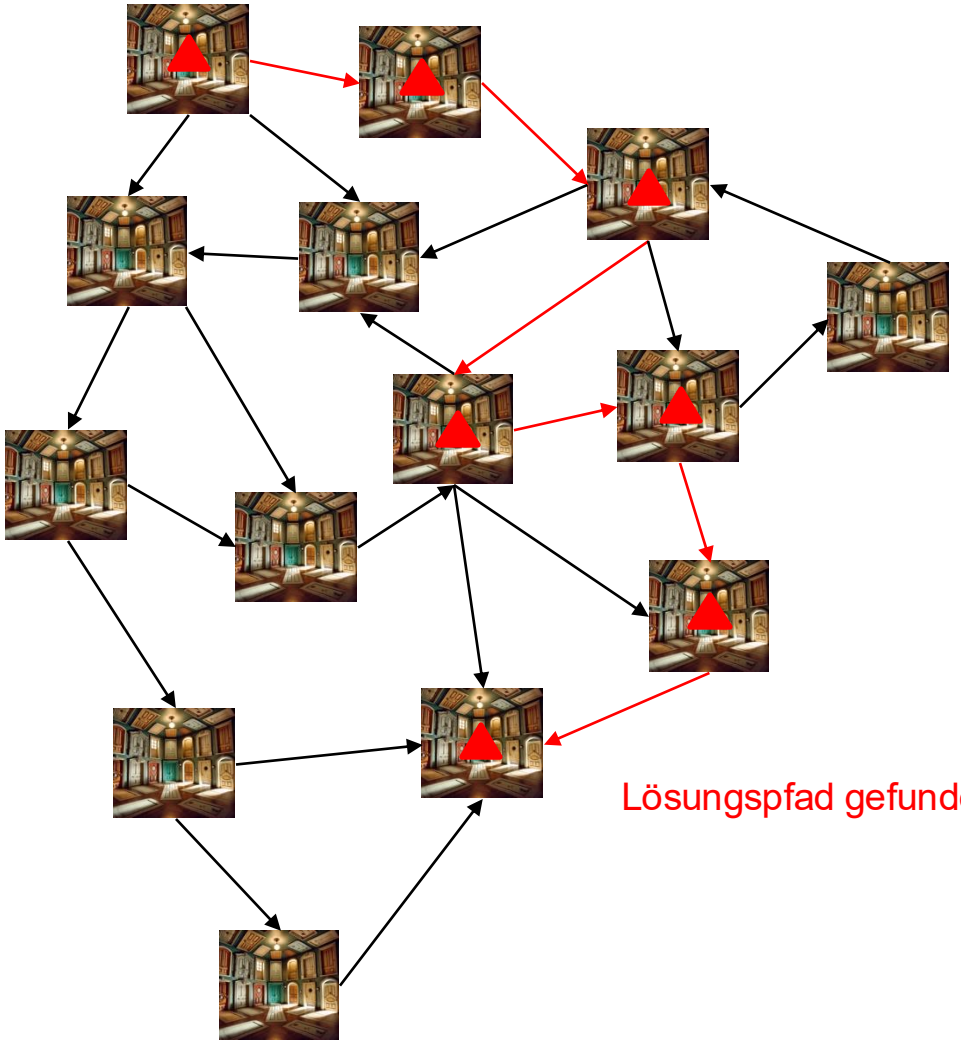


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited



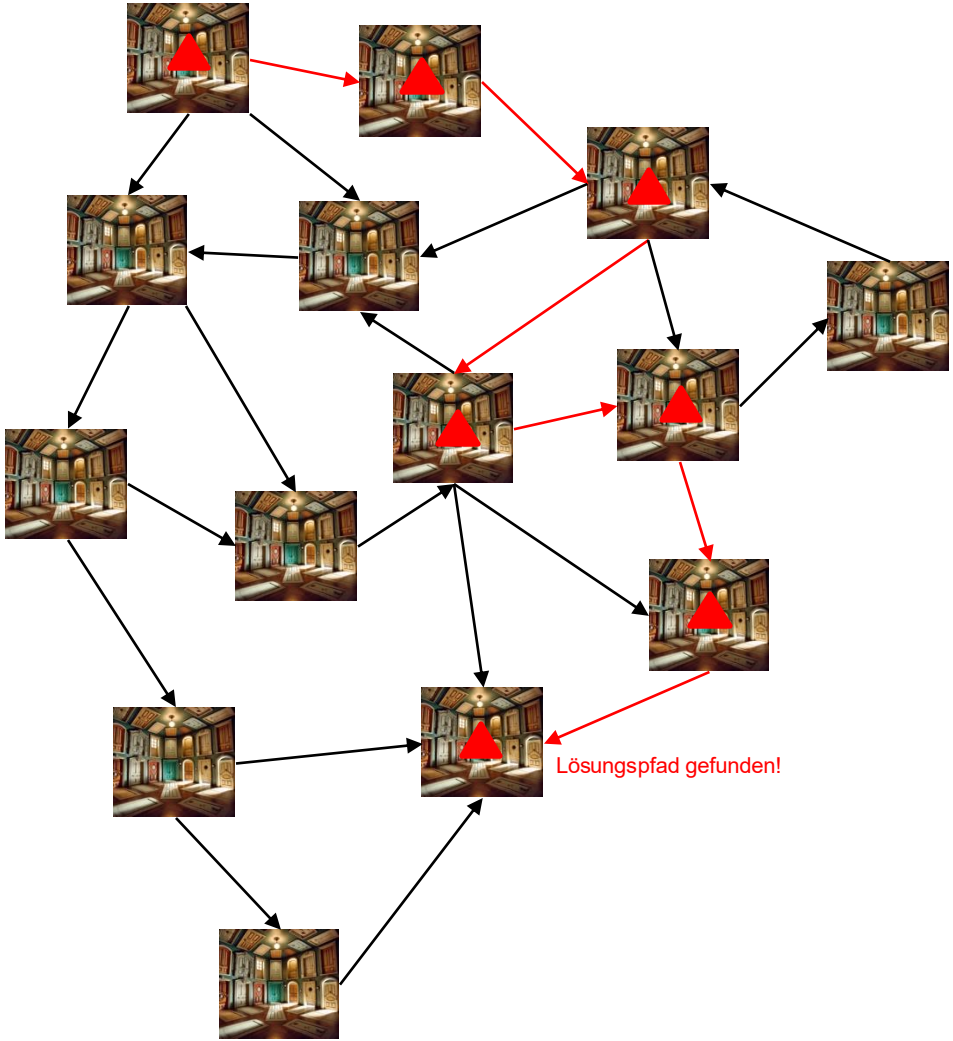
Lösungspfad gefunden!

Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

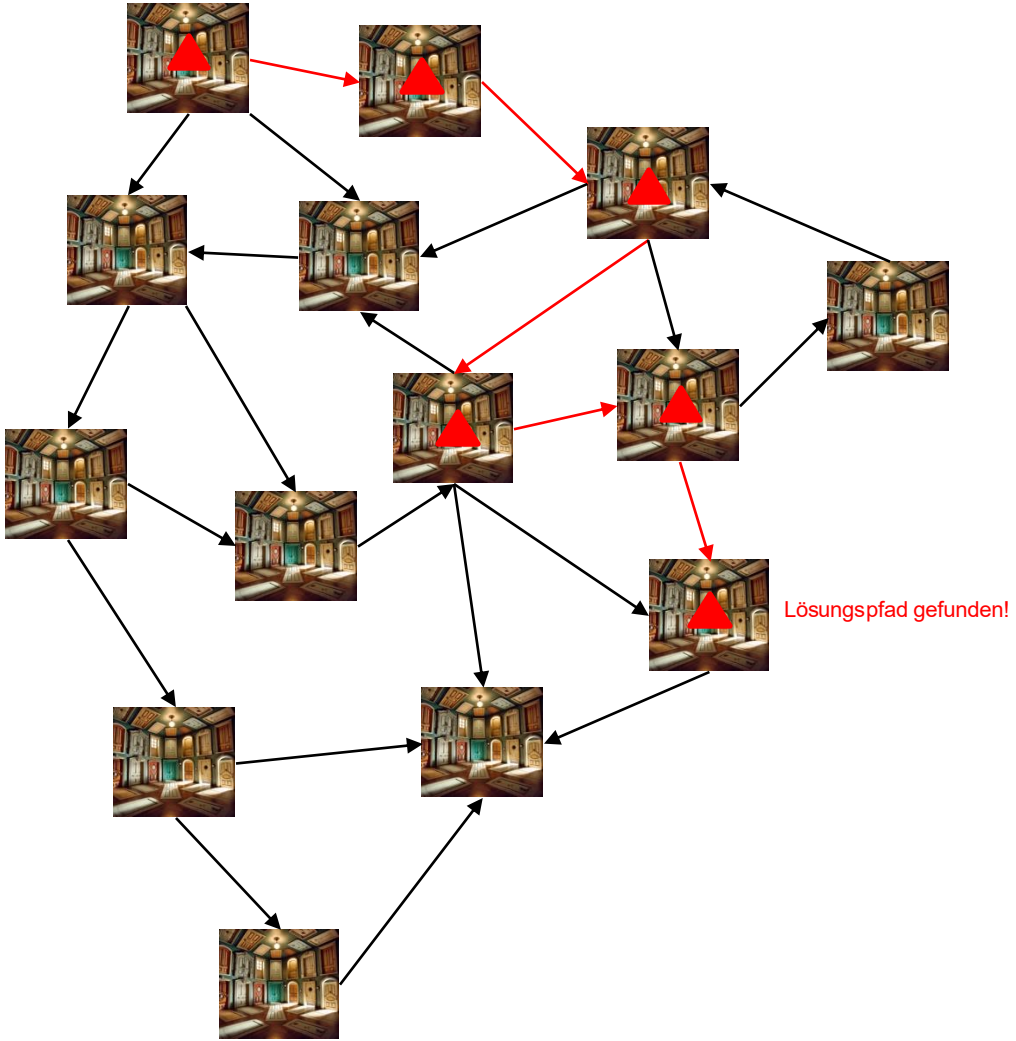


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

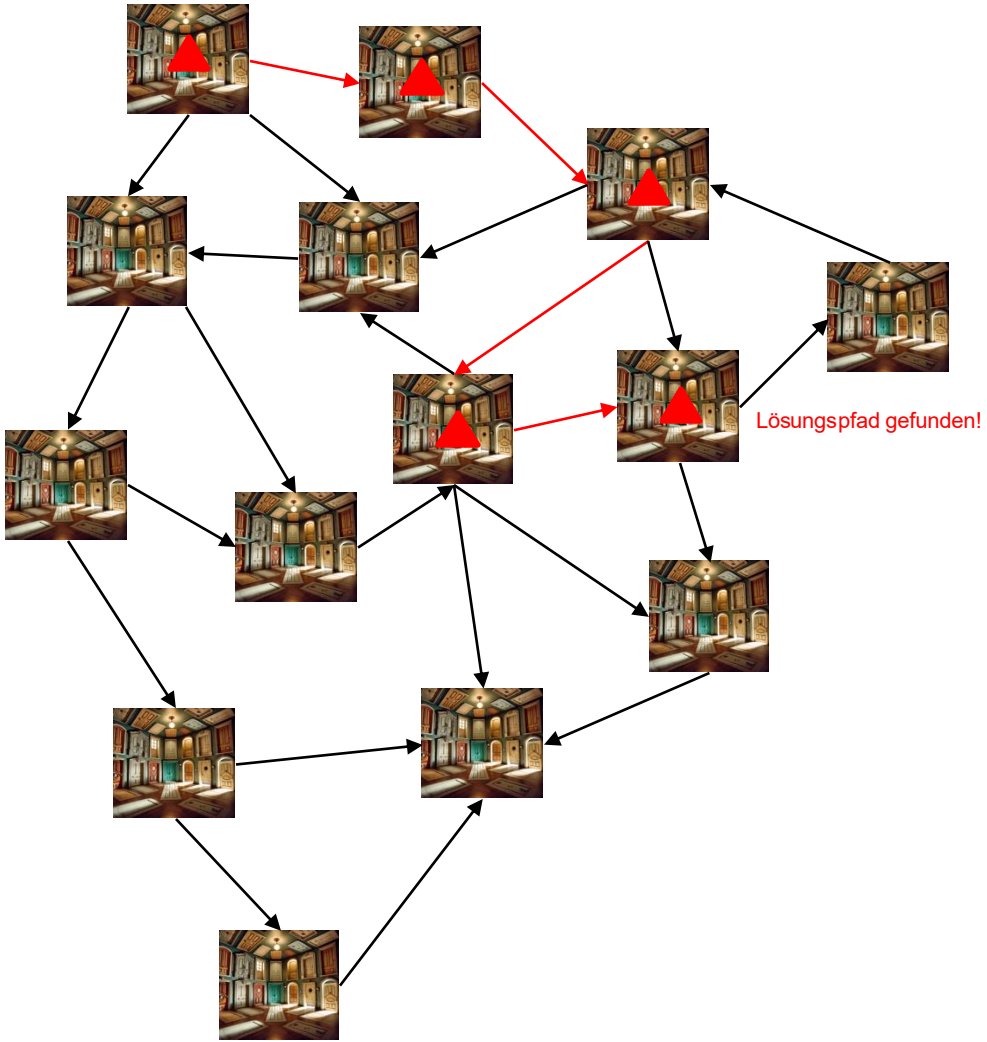


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

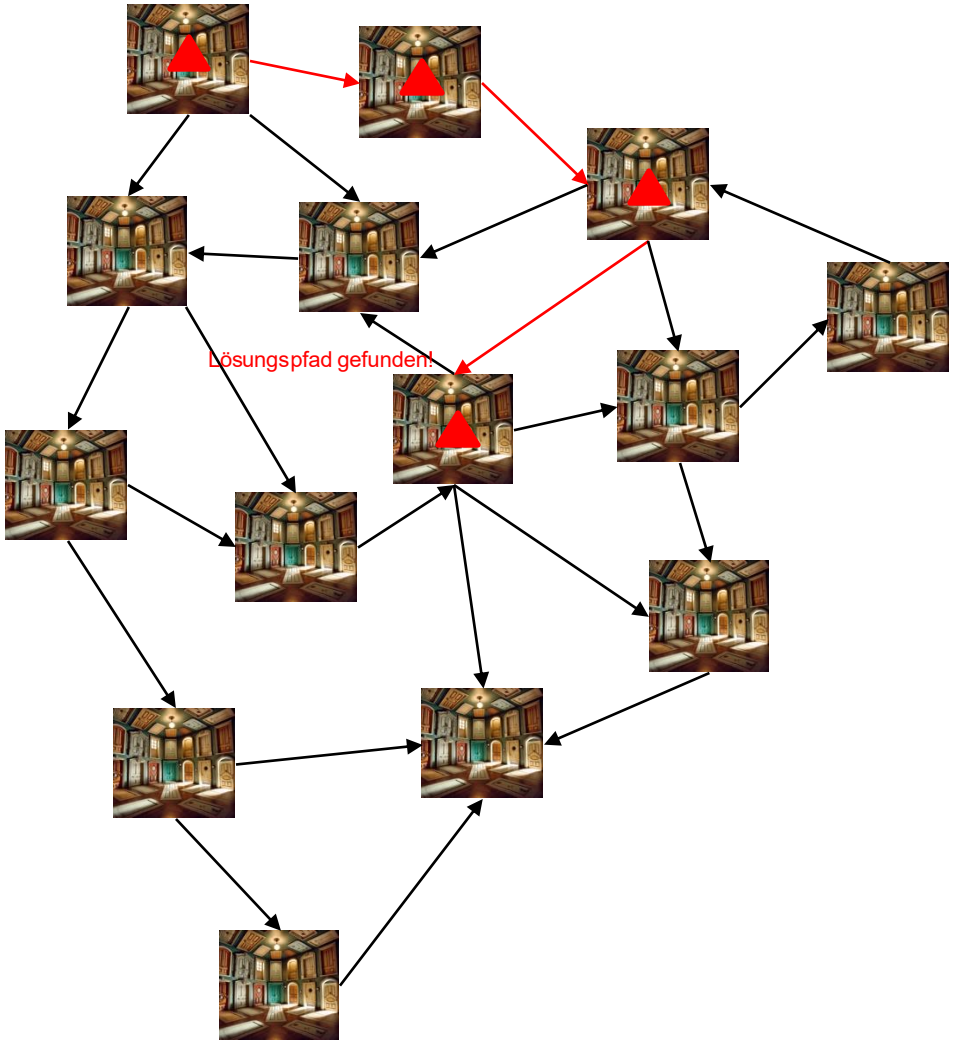


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

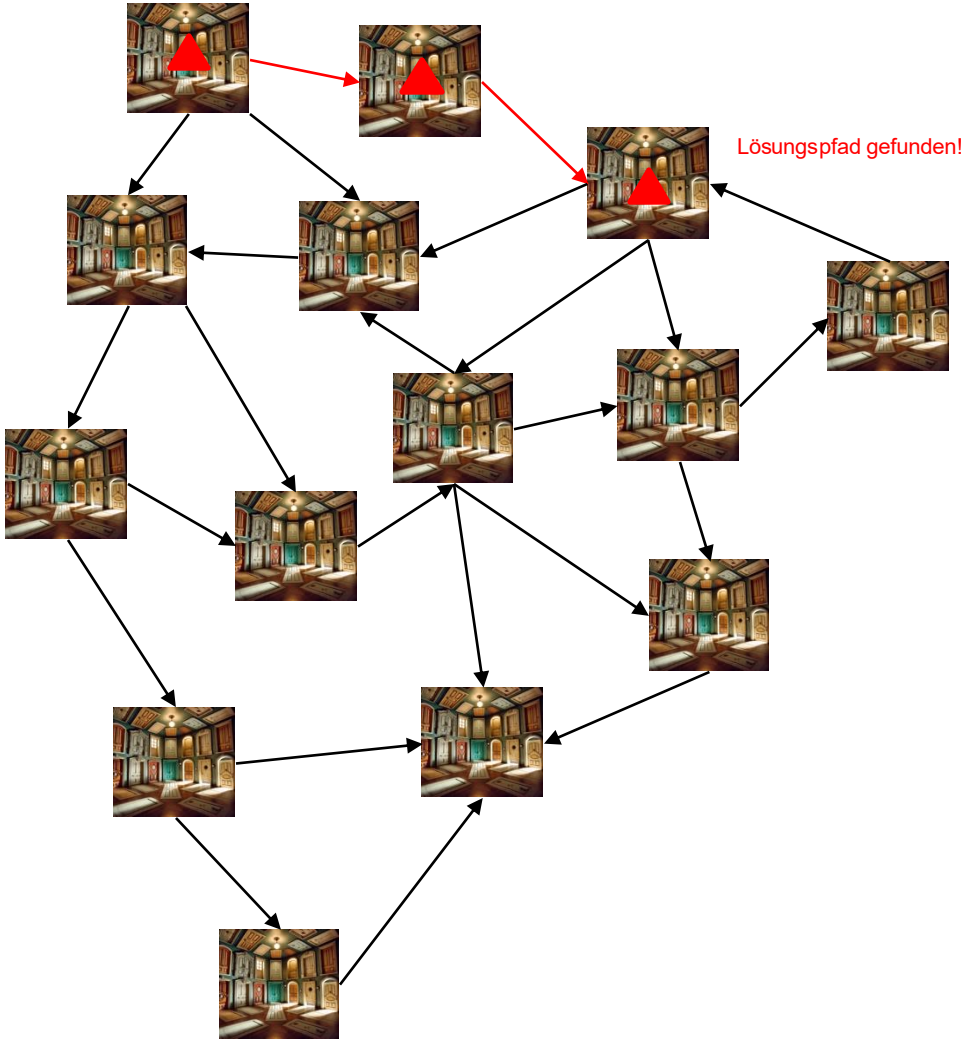


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

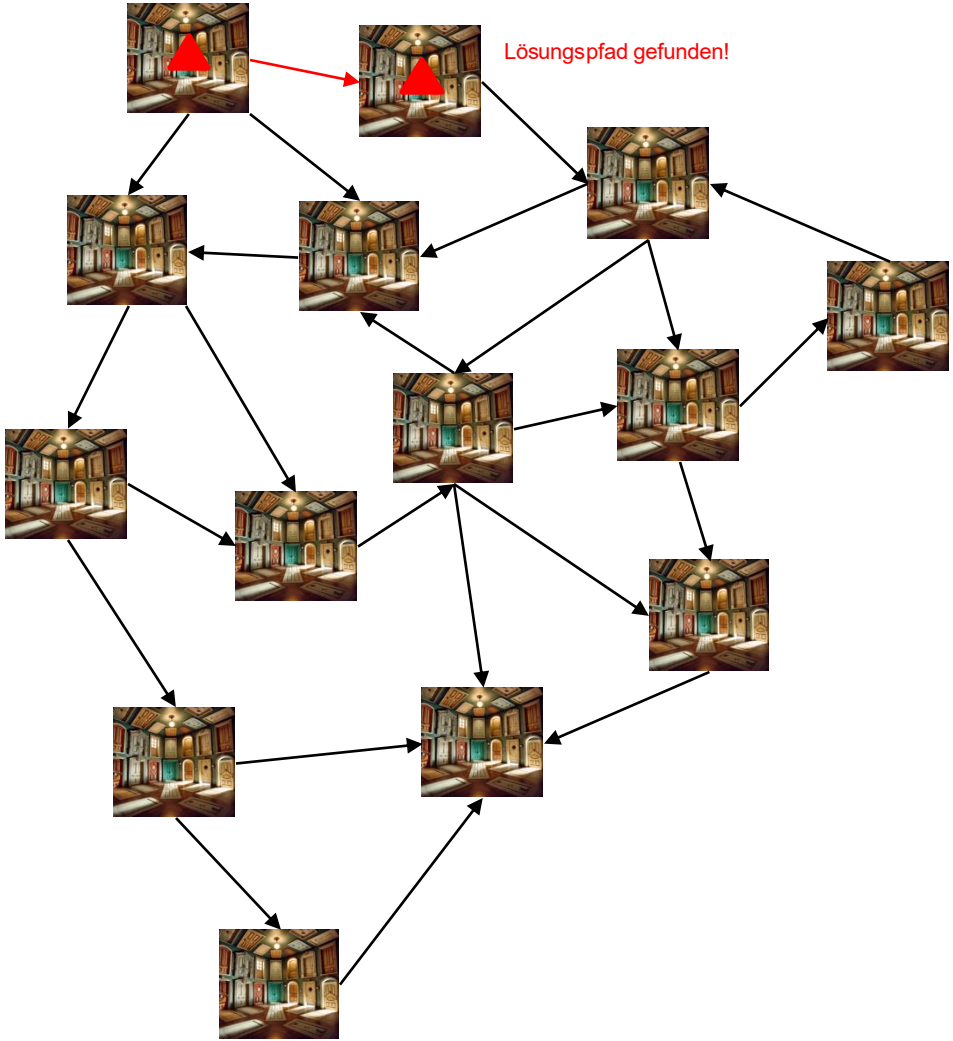


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

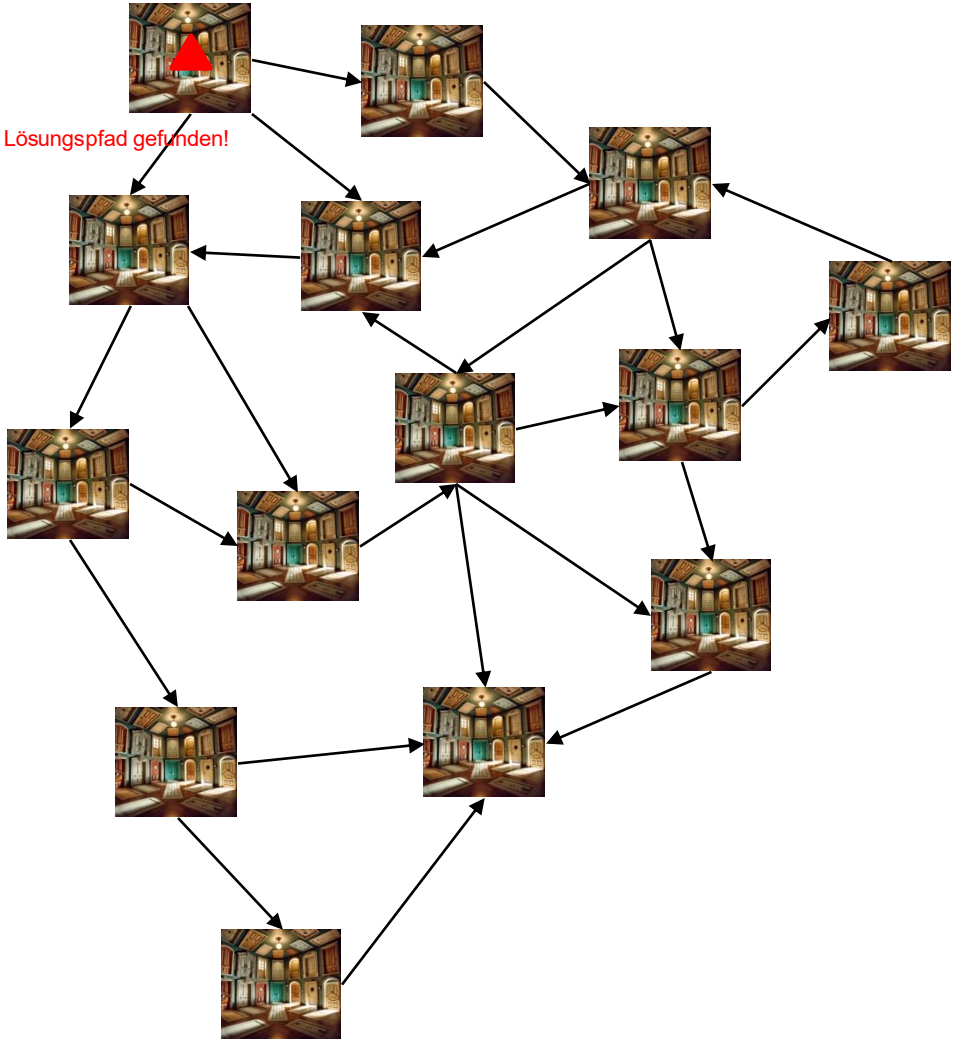


Wieso funktioniert das?

- Betrachten wir ein Beispiel mit

nicht visited

visited

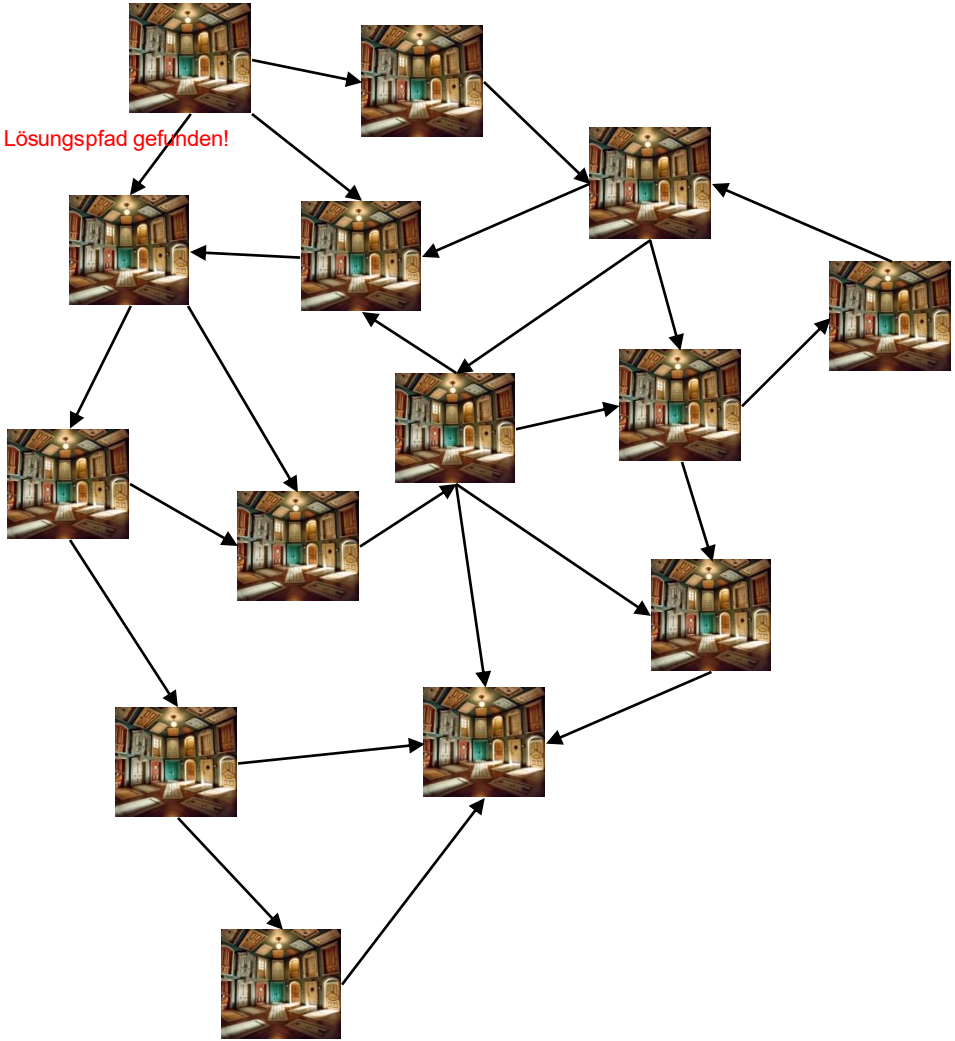


Wieso funktioniert das?

- 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?

```
1 public class Player {
2   private ??? name;
3   private ??? role;
4   private int energy;
5   …
  }
```

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?

```
1  public class Game {
2      private Player[] players;
3      private Random random;
4      …
5      public void energizeAll() {
6          for (int i = 0; i < players.length; i++) {
7              Player player = players[i];
8              if (player.role == Role.HONEST) {
9                  player.energize(1);
10             } else if (player.role == Role.TRICKSTER) {
11                 player.energize(random.nextInt(-4, 5));
12             }
           }
       }
   }
```

Vorteile von Enums:

- **Typsicherheit** – Der Compiler zwingt euch einen validen Typen zu verwenden. Schützt vor Tippfehlern, die sonst bis zu execution nicht erkannt werden
- **Konsistenz** – über verschiedene Klassen werden dieselben Konventionen verwendet.
- **Einfachere Comparison** – Vergleiche von Enums funktionieren mit **(==)**
- **Automatisierte Vervollständigung im IDE**

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