Arukone

Turkins Kirils, Team-ID: 00271

10. Oktober 2023

Inhaltsverzeichnis

0 | Einleitung

Diese Dokumentation beschreibt die Entwicklung eines Programms, das lösbare Arukone-Rätsel erstellt.

0.1 | Vorwort

In dieser Dokumentation kam GPT-4 minimal zum Einsatz, um die Grammatik zu berichtigen und auszubessern. Der Inhalt jedoch wurde komplett von einem Menschen verfasst.

Das Programm, von welchem sich diese Dokumentation handelt, wurde in Java 17 geschrieben mit der Verwendung von Gradle 8.2.1 als Build-Tool, um externe Frameworks wie JUnit für einfacheres und sauberes Testen des Programms zu verwenden. Außerdem erleichtert es dem BWINF-Team und den Lehrern, das

Programm auszuführen, da alle modernen Entwicklungsumgebungen Build-Tools unterstützen und der Quellcode somit mit den meisten Entwicklungsumgebungen kompatibel ist.

Um die Umsetzung anschaulicher zu machen, werden in den Abschnitten *3.1 bis 3.2* kleine Quellcode-Ausschnitte präsentiert. Die vollständigen **relevanten** Quellcode-Komponenten sind in den Abschnitten *5.1 bis 5.2* zu finden.

1 | Problembeschreibung

Eine Schwierigkeit bei dieser Aufgabe besteht darin, alle Rätsel so zu erstellen, dass sie vom Arukone-Checker nicht gelöst werden können. Außerdem soll jedes Rätsel einzigartig sein und nicht dieselbe Anordnung der Paare haben. Das Rätsel soll keine Größenbeschränkung haben und mindestens 4 Felder breit sein. Eine weitere Anforderung ist, dass jedes Rätsel **mindestens** n/2 Zahlenpaare enthalten muss.

2 | Lösungsideen

2.1 | Arukone-Checker Fallen

Das Ziel ist es, eine spezielle Falle zu konstruieren, die verschiedene Variationen hat und vom Arukone-Checker nicht gelöst werden kann.

Durch einiges ausprobieren und testen ist eine Falle ganz besonders ins Auge gekommen.

```
0 0 0 0
1 2 0 0
0 0 0 1
0 0 0 2
```

Bei dieser Aufstellung kann der Arukone-Checker keine Lösung finden. Der Grund dafür ist, dass die kürzeste Linie zwischen den beiden Einsen die kürzeste Linie zwischen den beiden Zweien kreuzt. Wenn man stattdessen mit den Zweien beginnt, kreuzt man ebenfalls die kürzeste Linie zwischen den Einsen.

2.1.1 | Variationen

Die gefundene Falle zeigt auch einige Variationen, zum Beispiel kann man die linke 1 nach unten auf der ganzen Reihe verschieben.

```
0 0 0 0 | 0 0 0 0 | 0 0 0 0
1 2 0 0 | 0 2 0 0 | 0 2 0 0
0 0 0 1 | 1 0 0 1 | 0 0 0 1
0 0 0 2 | 0 0 0 2 | 1 0 0 2
```

Man kann diese aber nicht nach oben verschieben, da es in diesem Beispiel zwei kürzeste Wege gibt, dann probiert der Arukone-Checker beide aus, ob einer die beiden Zweien nicht überkreuzt. Beim oberen kürzestem Weg, überkreuzt der Weg nicht die beiden Zweien, somit kann der Arukone-Checker dieses Rätsel lösen.

```
1 - - |
0 2 0 |
0 0 0 1
0 0 0 2
```

Um noch mehr Variationen dieser Falle zu erzeugen, kann man die Falle von rechts aufbauen. Außerdem kann man das gesamte Gitter rotieren, um noch mehr Variationen zu schaffen, ohne großen Aufwand.

2.2 | Generierung anderer Paare

Der Rest des Rätsels kann dann mit zufälligen Zahlenpaaren gefüllt werden. Um zu überprüfen, ob alle Zahlenpaare miteinander verbunden werden können, wird der A-Star-Algorithmus verwendet, da er besonders effizient ist.

3 | Umsetzung

3.1 | Arukone-Checker Fallen

Der y-Wert, auf dem die Falle liegt, wird zufällig zwischen 1 und der maximalen Breite minus 2 gewählt. Der Grund, warum wir bei Index 1 beginnen und nicht bei 0, ist, dass eine Linie von oben gezogen werden muss, um die Falle zu lösen. Ähnlich verhält es sich mit der Begründung, warum wir einen y-Wert von höchstens der Gitterbreite minus 2 haben dürfen. Dies liegt daran, dass die weiteren Zahlen, wie in der Lösungsidee bereits gezeigt, tiefer gelegt werden und Platz benötigen.

```
int x, y = random.nextInt(1, width - 2);
```

Anschließend setzen wir die restliche Falle basierend auf dem y-Wert der Falle mit einer leichten Variation. Nach der Setzung des ersten Zahlenpaars steigern wir die Paar-Ziffer, damit das nächste Paar eine aufsteigende Ziffer enthält.

```
x = 0;

setValue(x, y + random.nextInt(0, 3), pair);
setValue(width - 1, y + 1, pair);
pair++;

setValue(1, y, pair);
setValue(width - 1, y + 2, pair);
```

Um weitere Variation der Falle zu erzeugen, erstellen wir eine Chance von 50 %, dass die Falle von rechts statt von links aufgebaut werden soll.

```
boolean fromLeft = (random.nextInt() & 1) == 0;
```

3.2 | A-Star Algorithmus

3.2.1 | Warum A-Star

Der A-Star-Algorithmus wurde in diesem Programm gewählt, weil er im Vergleich zu anderen Algorithmen wie Dijkstra sehr effizient ist. Außerdem war die ursprüngliche Aufgabenstellung nicht klar verstanden worden und es wurde ein Programm entwickelt, das ein gegebenes Arukone-Rätsel löst. Dabei wurde A-Star zum Experimentieren verwendet und hat sich auch für die nun korrekte Implementierung der Aufgabenstellung gut geeignet.

3.2.2 | Zusammenfassung des Grundprinzips der Pfadsuche am Quellcode-Beispiel

- Zuerst berechnet die Methode ein zweidimensionales Array von Knoten aus dem Gitter, dem Startpunkt und dem Endpunkt mit der Hilfsmethode calculateNodes.

```
Node[][] nodes = calculateNodes(grid, start, end);
```

- Dann fügt die Methode den Knoten am Startpunkt zu einer Liste von aktuellen Knoten hinzu und markiert ihn als besucht.

```
currentNodes.add(nodes[start.x][start.y]);
currentNodes.get(0).setVisited(true);
```

- Dann beginnt die Methode eine Schleife, die so lange läuft, bis der erste Knoten in der aktuellen Knotenliste der Endknoten ist.
- In jeder Iteration der Schleife ruft die Methode die Hilfsmethode getCheapestNeighbors auf, die billigsten Nachbarn des ersten Knotens in der aktuellen Knotenliste zu erhalten. Die billigsten Nachbarn sind die Knoten, die begehbar sind, noch nicht besucht wurden und die niedrigsten Gesamtkosten haben.
- Wenn es keine Nachbarn gibt, bedeutet das, dass es keinen Pfad gibt. In diesem Fall versucht die Methode, einen anderen Knoten in dem Knotenarray zu finden, der die gleiche bisherige Distanz wie der erste Knoten in der aktuellen Knotenliste hat, und setzt die Schleife mit diesem Knoten fort. Wenn es keinen solchen Knoten gibt, gibt die Methode null zurück (kein Pfad).

```
Optional<Node> cheapestOption =
        Arrays.stream(nodes).flatMap(Arrays::stream).filter(node ->
node.getOriginDistance() == finalI).findFirst();

if (cheapestOption.isPresent()) {
    currentNodes.clear();
    currentNodes.add(cheapestOption.get());

    neighbors = getCheapestNeighbors(currentNodes, nodes, endNode);

    if (neighbors != null) break;
}
```

- Wenn der erste Nachbar der Endknoten ist, löscht die Methode die aktuelle Knotenliste und fügt diesen Nachbarn hinzu. Dann bricht die Methode die Schleife ab.

- Andernfalls löscht die Methode die aktuelle Knotenliste und fügt alle Nachbarn hinzu.
- Nachdem die Schleife beendet ist, erstellt die Methode einen Stapel von Knoten, der den Pfad darstellt. Die Methode läuft eine weitere Schleife, die so lange läuft, bis der erste Knoten in der aktuellen Knotenliste der Startknoten ist. In jeder Iteration der Schleife fügt die Methode den ersten Knoten in der aktuellen Knotenliste zu dem Pfadstapel hinzu und setzt den ersten Knoten in der aktuellen Knotenliste auf den Nachfolger dieses Knotens. Wenn der Nachfolger null ist, bricht die Methode die Schleife ab.
- Schließlich gibt die Methode den Pfadstapel zurück.

4 | Beispiele

In diesem Abschnitt finden Sie alle relevanten Beispiele von der Ausgabe des Programms zur jeweiligen Eingabe.

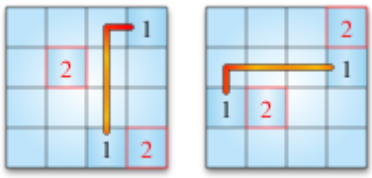
4.1 | Variation der Falle an kleinen Feldern 4x4

In diesem Beispiel werden 8 verschiedene Variationen gezeigt, welche bei einem 4x4 Gitter möglich generierbar sind, die der Arukone-Checker nicht lösen kann. Es sind auch mehr Variationen möglich!

Ausgabe (leicht modifiziert):

```
0 0 0 1  | 0 0 0 2
0 2 0 0  | 0 0 0 1
0 0 0 0  | 1 2 0 0
0 0 1 2  | 0 0 0 0
```

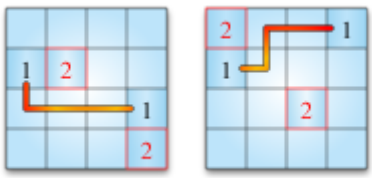
Arukone-Checker lösungs Versuch:



Ausgabe (leicht modifiziert):

```
0 0 0 0  | 2 0 0 1
1 2 0 0  | 1 0 0 0
0 0 0 1  | 0 0 2 0
0 0 0 2  | 0 0 0 0
```

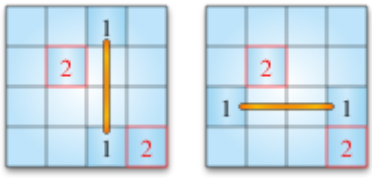
Arukone-Checker lösungs Versuch:



Ausgabe (leicht modifiziert):

```
0 0 1 0  | 0 0 0 0
0 2 0 0  | 0 2 0 0
0 0 0 0  | 1 0 0 1
0 0 1 2  | 0 0 0 2
```

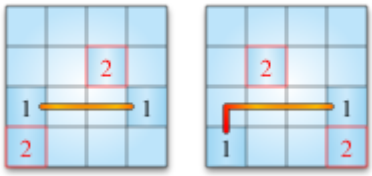
Arukone-Checker lösungs Versuch:



Ausgabe (leicht modifiziert):

```
0 0 0 0  | 0 0 0 0
0 0 2 0  | 0 2 0 0
1 0 0 1  | 0 0 0 1
2 0 0 0  | 1 0 0 2
```

Arukone-Checker lösungs Versuch:



4.2 | Grosse Rätsel

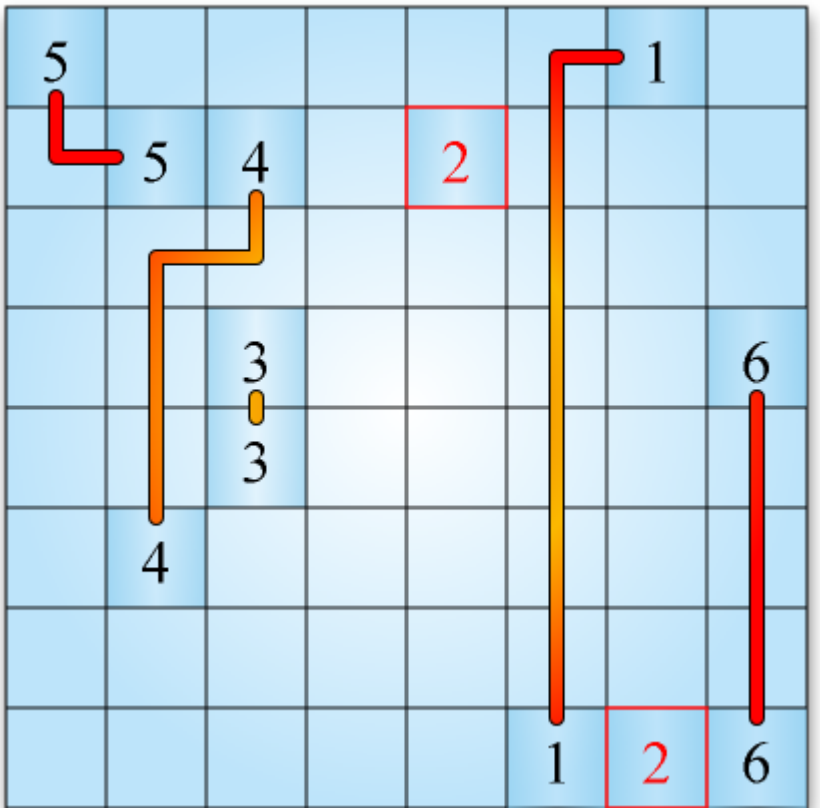
4.2.1 | Zwei 8x8 Rätsel

Dieses Beispiel enthält zwei 8x8 Rätsel, die von diesem Programm erzeugt wurden und sich stark in ihrer Struktur unterscheiden. Diese können vom Arukone-Checker nicht gelöst werden.

Ausgabe:

```
5 0 0 0 0 0 1 0
0 5 4 0 2 0 0 0
0 0 0 0 0 0 0 0
0 0 3 0 0 0 0 6
0 0 3 0 0 0 0 0
0 4 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 1 2 6
```

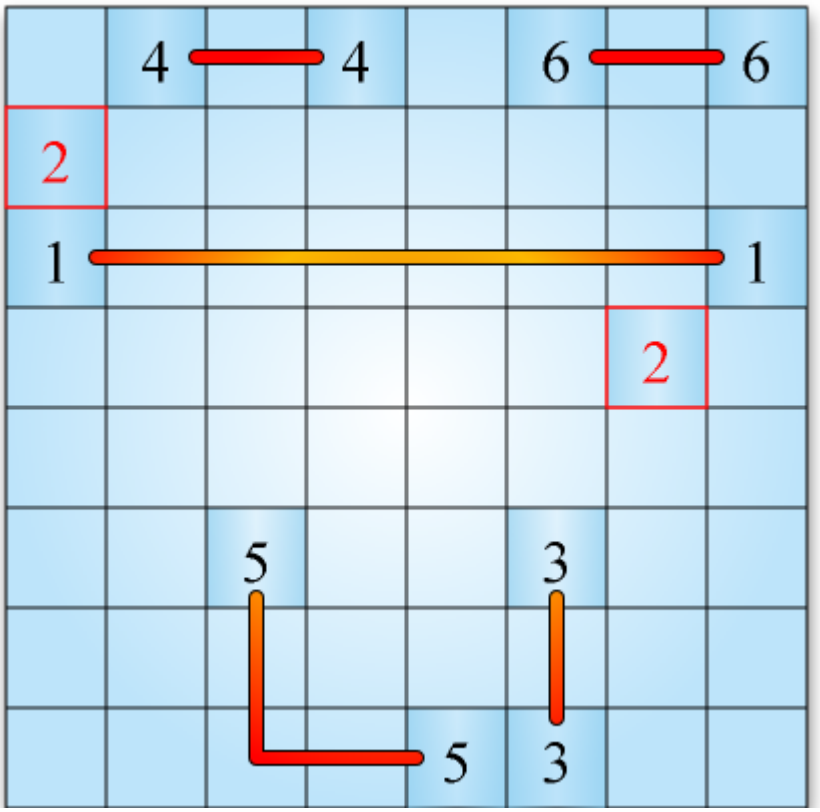
Arukone-Checker lösungs Versuch:



Ausgabe:

```
0 4 0 4 0 6 0 6
2 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1
0 0 0 0 0 0 2 0
0 0 0 0 0 0 0 0
0 0 5 0 0 3 0 0
0 0 0 0 0 0 0 0
0 0 0 0 5 3 0 0
```

Arukone-Checker lösungs Versuch:



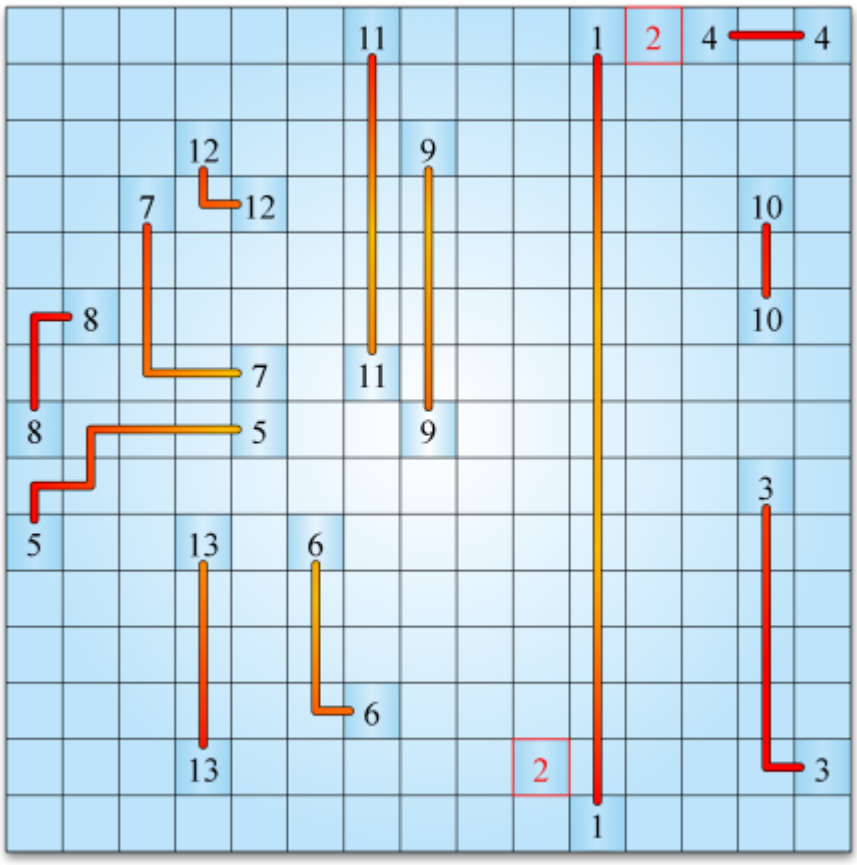
4.2.2 | 15x15 Rätsel

An diesem 15x15 Rätsel wird nochmal verdeutlicht wie die Falle eingebaut ist.

Ausgabe:

```
0 0 0 0 0 0 11 0 0 0 1 2 4 0 4
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 12 0 0 0 9 0 0 0 0 0 0 0
0 0 7 0 12 0 0 0 0 0 0 0 0 10 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 8 0 0 0 0 0 0 0 0 0 0 0 10 0
0 0 0 0 7 0 11 0 0 0 0 0 0 0 0
8 0 0 0 5 0 0 9 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 3 0
5 0 0 13 0 6 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 6 0 0 0 0 0 0 0 0
0 0 0 13 0 0 0 0 0 2 0 0 0 0 3
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
```

Arukone-Checker lösungs Versuch:



4.2.3 | 30x30 Rätsel

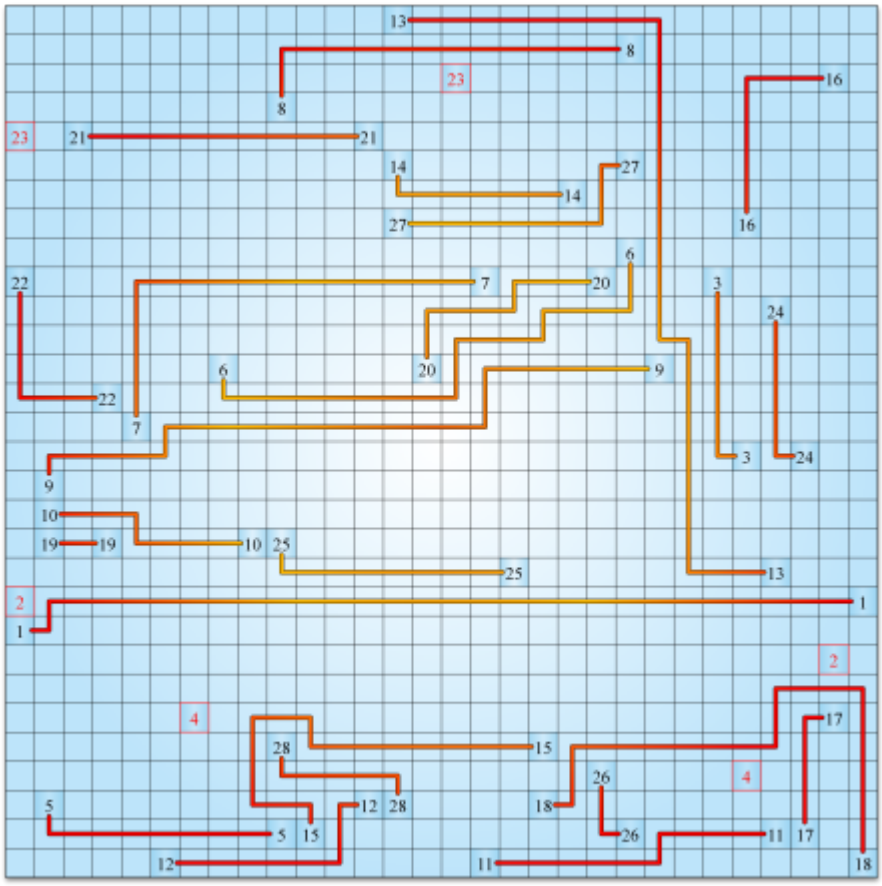
Dies ist ein 30x30 Rätsel, das die größte Größe darstellt, die der Arukone-Checker lösen kann. In diesem Beispiel sieht man auch, wie manchmal zusätzliche Schwierigkeiten eingebaut sind, die der Arukone-Checker nicht bewältigen kann, die aber lösbar sind.

Ausgabe (Die Wahrscheinlichkeit ausgerechnet durch Versuche liegt bei 2,04 % für eine Ausgabe mit 2 zusätzlichen Fallen):

```
0 0 0 0 0 0 0 0 0 0 0 0 0 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 23 0 0 0 0 0 0 0 0 0 0 0 16 0
0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
23 0 21 0 0 0 0 0 0 0 0 21 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 14 0 0 0 0 0 0 27 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 14 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 27 0 0 0 0 0 0 0 0 0 0 16 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0
22 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7 0 0 0 20 0 0 0 3 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 24 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 6 0 0 0 0 0 0 20 0 0 0 0 0 0 0 9 0 0 0 0 0 0 0
0 0 0 22 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 24 0 0
0 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 19 0 19 0 0 0 0 10 25 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 25 0 0 0 0 0 0 0 0 13 0 0 0
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

```
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 17 0
0 0 0 0 0 0 0 0 0 28 0 0 0 0 0 0 0 0 15 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 26 0 0 0 0 4 0 0 0 0
0 5 0 0 0 0 0 0 0 0 0 0 12 28 0 0 0 0 18 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 5 15 0 0 0 0 0 0 0 0 0 0 26 0 0 0 0 11 17 0 0
0 0 0 0 0 12 0 0 0 0 0 0 0 0 0 0 11 0 0 0 0 0 0 0 0 0 0 0 18
```

Arukone-Checker lösungs Versuch:



5 | Quellcode

Dieser Abschnitt enthält alle **wichtigen** Quellcode Komponenten, geschrieben in Java. Bei jedem Quellcode-Komponenten finden Sie den Pfad zur Klasse im Programm. Eine weitere Anmerkung ist, dass Kommentare und Quellcode, einschließlich Variablen- und Klassennamen, auf Englisch verfasst sind. Dies entspricht der allgemeinen Konvention in der Programmierung. Der vollständige Quellcode ist im Programmordner angegeben.

5.1 | Gitter

Pfad im Programmordner: *src/main/java/de/arukone/Grid.java*

```java
/**
 * generates a new solvable Arukone grid
 */
    public Grid(int size) {
```

```
        this.grid = new int[size * size];
        this.width = size;

        int trapYPos = setTrap();
        generateRandomPairs(trapYPos);

        randomRotate();
    }
```

5.1.1 | Fallen setzung

```
// Sets a two-pair trap for the Arukone solver with small variation
    private int setTrap() {
        boolean fromLeft = (random.nextInt() & 1) == 0;
        int x, y = random.nextInt(1, width - 2);

        if (fromLeft) {
            x = 0;

            setValue(x, y + random.nextInt(0, 3), pair);
            setValue(width - 1, y + 1, pair);
            pair++;

            setValue(1, y, pair);
            setValue(width - 1, y + 2, pair);
        } else {
            x = width - 1;

            setValue(x, y + random.nextInt(0, 3), pair);
            setValue(0, y + 1, pair);
            pair++;

            setValue(x - 1, y, pair);
            setValue(0, y + 2, pair);
        }
        pair++;
        return y - 1;
    }
```

5.1.2 | Generierung weiterer Paare

```
// A method to generate random pairs of points and connect them with a path
    private void generateRandomPairs(int trapYPos) {
        List<Node> takenNodes = new ArrayList<>();

        // Loop until the pair variable reaches the width
        while (pair < width - 1) {
            Point[] points = generateRandomPoints(trapYPos, takenNodes);

            // Get the path between the points using AStar algorithm
            Stack<Node> nodes = AStar.getPath(this, points[0], points[1]);
```

```
            // Check if the path is valid
            if (nodes == null || nodes.isEmpty()) {
                resetValues(points);
            } else if (isPathInvalid(nodes, takenNodes, trapYPos)) {
                resetValues(points);
            } else {
                setValues(points, pair);

                takenNodes.addAll(nodes);
                pair++;
            }
        }
    }
```

5.1.3 | Gitter Rotation

```
// A method that randomly rotates the grid
    private void randomRotate() {
        int n = (int) (Math.random() * 3);

        // Rotate the grid n times by 90 degrees clockwise
        for (int i = 0; i < n; i++) rotate90();
    }

    // A helper method that rotates the grid by 90 degrees clockwise
    private void rotate90() {
        int[][] newGrid = new int[width][width];

        // Loop through the original grid and copy the values to the new grid
        for (int i = 0; i < width; i++) {
            for (int j = 0; j < width; j++) {
                // The value at (i, j) in the original grid will be at (j, width -
i - 1) in the new grid
                newGrid[j][width - i - 1] = getValue(i, j);
            }
        }

        // Loop through the new grid and set the values to the original grid
        for (int i = 0; i < width; i++) {
            for (int j = 0; j < width; j++) {
                setValue(i, j, newGrid[i][j]);
            }
        }
    }
```

5.2 | A-Star Algorithmus

Pfad im Programmordner: *src/main/java/de/arukone/AStar.java*

```
public class AStar {

    // A method that returns a stack of nodes that represents the path from the
```

```
start point to the end point in a grid
    public static Stack<Node> getPath(Grid grid, Point start, Point end) {
        // A list of nodes that are currently being considered for the path
        List<Node> currentNodes = new ArrayList<>(grid.width);
        // A two-dimensional array of nodes that are calculated from the grid, the
start point, and the end point
        Node[][] nodes = calculateNodes(grid, start, end);

        // Add the node at the start point to the current nodes list
        currentNodes.add(nodes[start.x][start.y]);
        currentNodes.get(0).setVisited(true);

        Node endNode = nodes[end.x][end.y];

        // Loop until the first node in the current nodes list is the end node
        while (currentNodes.get(0).getX() != end.x || currentNodes.get(0).getY()
!= end.y) {
            // Get the cheapest neighbors
            List<Node> neighbors = getCheapestNeighbors(currentNodes, nodes,
endNode);

            // If there are no neighbors, then there is no path
            if (neighbors == null) {
                // Get the longest distance of the first node in the current nodes
list
                int longestDistance = currentNodes.get(0).getOriginDistance();

                // Loop through all possible distances
                for (int i = 0; i < longestDistance; i++) {
                    int finalI = i;
                    // Find the first node in the nodes array that has the same
distance as i
                    Optional<Node> cheapestOption =
Arrays.stream(nodes).flatMap(Arrays::stream).filter(node ->
node.getOriginDistance() == finalI).findFirst();
                    // If such a node exists, clear the current nodes list and add
that node to it
                    if (cheapestOption.isPresent()) {
                        currentNodes.clear();
                        currentNodes.add(cheapestOption.get());

                        neighbors = getCheapestNeighbors(currentNodes, nodes,
endNode);

                        if (neighbors != null) break;
                    }
                }
                // If there are still no neighbors, return null (no path)
                if (neighbors == null) return null;
            }
            // If the first neighbor is the end node, clear the current nodes list
and add that neighbor to it
            else if (neighbors.get(0).equals(endNode)) {
                currentNodes.clear();
                currentNodes.add(neighbors.get(0));
                break;
            }
```

```
            // Otherwise, clear the current nodes list and add all the neighbors
to it
            else {
                currentNodes.clear();
                currentNodes.addAll(neighbors);
            }
        }

        Stack<Node> path = new Stack<>();
        // Loop until the first node in the current nodes list is the start node
        while (currentNodes.get(0).getX() != start.x || currentNodes.get(0).getY()
!= start.y) {
            // Push the first node in the current nodes list to the path stack
            path.push(currentNodes.get(0));
            // Get the child node of the first node in the current nodes list
            Node child = currentNodes.get(0).getChild();

            // If the child node is not null, set it as the first node in the
current nodes list
            if (child != null) {
                currentNodes.set(0, child);
            }
            // Otherwise, break out of the loop
            else break;
        }
        // Return the path stack
        return path;
    }


    // A method that returns a list of the cheapest neighbors of a list of current
nodes in a grid
    private static List<Node> getCheapestNeighbors(List<Node> currentNodes, Node[]
[] nodes, Node endNode) {
        // Use a list of offsets to avoid nested loops
        int[][] offsets = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
        // Create a list of neighbors to store the valid and unvisited nodes
around the current nodes
        List<Node> neighbors = new ArrayList<>(nodes.length);

        for (Node currentNode : currentNodes) {
            for (int[] offset : offsets) {
                // Add the offset to the current node coordinates
                int x = currentNode.getX() + offset[0];
                int y = currentNode.getY() + offset[1];
                // Check if the node is valid and add it to the list
                if (inBounds(x, y, nodes.length)) {
                    // Get the node from the matrix
                    Node node = nodes[x][y];

                    // If the node is not an obstacle and not visited, add it to
the neighbors list and set its properties
                    if (!node.isObstacle() && !node.isVisited()) {
                        node.setHorizontal(offset[0] != 0);
                        node.setVisited(true);
                        node.setChild(currentNode);
                        neighbors.add(node);
```

```
                    }
                    // If the node is equal to the end node, set its child and
return a singleton list containing it
                    else if (node.equals(endNode)) {
                        endNode.setChild(currentNode);
                        return List.of(endNode);
                    }
                }
            }
        }

        // If the neighbor list is not empty, return the lowest cost neighbors
from it
        if (!neighbors.isEmpty()) {
            return getLowestCostNeighbors(neighbors);
        }
        // Otherwise, return null
        else return null;
    }



    private static List<Node> getLowestCostNeighbors(List<Node> neighbors) {
        Comparator<Node> nodeComparator = Comparator.comparingInt(Node::getCost)
                .thenComparingInt(Node::getOriginDistance);
        Optional<Node> minNode = neighbors.stream().min(nodeComparator);
        if (minNode.isPresent()) {
            // Get the minimum cost and origin distance from the minNode
            int lowestCost = minNode.get().getCost();
            int lowestOriginDistance = minNode.get().getOriginDistance();
            // Return a list of nodes that match the minimum values
            return neighbors.stream()
                    .filter(node -> node.getCost() == lowestCost &&
node.getOriginDistance() == lowestOriginDistance)
                    .collect(Collectors.toList());
        } else {
            // Return an empty list if the input list is empty
            return Collections.emptyList();
        }
    }


    // A method that returns a two-dimensional array of nodes that are calculated
from a grid, a start point, and an end point
    private static Node[][] calculateNodes(Grid grid, Point start, Point end) {
        // Create a new array of nodes with the same size as the grid
        Node[][] nodes = new Node[grid.width][grid.width];

        for (int x = 0; x < grid.width; x++) {
            for (int y = 0; y < grid.width; y++) {
                // Calculate the distance of the node from the start point
                int originDistance = Math.abs(start.x - x) + Math.abs(start.y -
y);
                // Calculate the distance of the node from the end point
                int goalDistance = Math.abs(end.x - x) + Math.abs(end.y - y);
                // Check if the node is an obstacle based on the grid value
                boolean isObstacle = grid.getValue(x, y) != 0;
```

```
                // Create a new node with the calculated parameters and assign it
to the array
                nodes[x][y] = new Node(x, y, originDistance, originDistance +
goalDistance, isObstacle);
            }
        }

        return nodes;
    }


    private static boolean inBounds(int x, int y, int width) {
        return x >= 0 && x < width && y >= 0 && y < width;
    }
}
```

5.2.1 | Node

Pfad im Programmordner: *src/main/java/de/arukone/Node.java*

```
/**
 * A class that represents a node in a grid.
 * A node has coordinates, distance from the origin, cost, obstacle status, child
node, visited status, and horizontal flag.
 * A node can be compared to another node based on the sum of its distance and
cost.
 */
public class Node implements Comparable<Node> {

    // The x-coordinate of the node
    private final int x;
    // The y-coordinate of the node
    private final int y;
    // The distance of the node from the origin
    private final int originDistance;
    // The cost of the node
    private final int cost;
    // The obstacle status of the node
    private final boolean obstacle;

    // The child node of the current node
    private Node child;
    // The visited status of the node
    private boolean visited;
    // The horizontal flag of the node
    private boolean horizontal;

    /**
     * Constructs a new node with the given parameters.
     * @param x The x-coordinate of the node
     * @param y The y-coordinate of the node
     * @param originDistance The distance of the node from the origin
     * @param cost The cost of the node
```

```
     * @param obstacle The obstacle status of the node
     */
    public Node(int x, int y, int originDistance, int cost, boolean obstacle) {
        this.x = x;
        this.y = y;
        this.originDistance = originDistance;
        this.cost = cost;
        this.obstacle = obstacle;
    }

    /**
     * Returns the x-coordinate of the node.
     * @return The x-coordinate of the node
     */
    public int getX() {
        return x;
    }

    /**
     * Returns the y-coordinate of the node.
     * @return The y-coordinate of the node
     */
    public int getY() {
        return y;
    }

    /**
     * Returns the distance of the node from the origin.
     * @return The distance of the node from the origin
     */
    public int getOriginDistance() {
        return originDistance;
    }

    /**
     * Returns the cost of the node.
     * @return The cost of the node
     */
    public int getCost() {
        return cost;
    }

    /**
     * Returns the obstacle status of the node.
     * @return True if the node is an obstacle, false otherwise
     */
    public boolean isObstacle() {
        return obstacle;
    }

    /**
     * Returns the child node of the current node.
     * @return The child node of the current node
     */
    public Node getChild() {
        return child;
    }
```

```java
    /**
     * Sets the child node of the current node.
     * @param child The child node to be set
     */
    public void setChild(Node child) {
        this.child = child;
    }

    /**
     * Returns the visited status of the node.
     * @return True if the node has been visited, false otherwise
     */
    public boolean isVisited() {
        return visited;
    }

    /**
     * Sets the visited status of the node.
     * @param visited The visited status to be set
     */
    public void setVisited(boolean visited) {
        this.visited = visited;
    }

    /**
     * Returns the horizontal flag of the node.
     * @return True if the node is horizontal, false otherwise
     */
    public boolean isHorizontal() {
        return horizontal;
    }

    /**
     * Sets the horizontal flag of the node.
     * @param horizontal The horizontal flag to be set
     */
    public void setHorizontal(boolean horizontal) {
        this.horizontal = horizontal;
    }

    /**
     * Compares this node to another node based on their sum of distance and cost.
     * @param anotherNode The other node to be compared with
     * @return A negative integer, zero, or a positive integer as this sum is less
than, equal to, or greater than anotherNode's sum
     */
    @Override
    public int compareTo(Node anotherNode) {
        return Integer.compare(originDistance + cost, anotherNode.originDistance +
anotherNode.cost);
    }

    /**
     * Checks if this object is equal to another object based on their
coordinates.
     * @param o The other object to be checked for equality with
```

```
     * @return True if this object is equal to o, false otherwise
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Node)) return false;
        Node that = (Node) o;
        return x == that.x && y == that.y;
    }

    /**
     * Returns the hash code of this object based on its coordinates.
     * @return The hash code of this object
     */
    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }
}
```