Zauberschule

Turkins Kirils, Team-ID: 00271

18. Novemember 2023

Inhaltsverzeichnis

0 | Einleitung

Diese Dokumentation beschreibt die Entwicklung eines Programms, welches einen geeigneten Weg von Punkt A zum Punkt B in der Zauberschule berechnet.

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 im Abschnitt *5* zu finden.

1 | Problembeschreibung

Ein Problem bei dieser Aufgabe ist, dass es eine zweite Etage gibt, in der weitere Wege mit zusätzlichen Zeitkosten verbunden sind. Der Algorithmus soll sich entscheiden können, ob er den "längeren" Weg mit geringeren Zeitkosten oder den kürzeren Weg mit höheren Zeitkosten wählt.

2 | Lösungsideen

Die Lösung dieses Problems ist recht einfach, da die Problembeschreibung genau den Fall darstellt, für den der A*-Algorithmus geeignet ist. Eine leicht modifizierte Implementierung des A*-Algorithmus sollte diese Aufgabe vollständig und effizient lösen können.

Die zweite Etage kann genauso wie jeder andere Knoten repräsentiert werden, mit einem zusätzlichen Kostenfaktor von 3.

3 | Umsetzung

In diesem Abschnitt werden spezielle Anpassungen des A*-Algorithmus vorgestellt. Die reine A*-Implementierung wird hier nicht erklärt. Wenn Sie sich dafür interessieren, können Sie den Abschnitt *5-Quellcode* lesen oder diesem Link folgen: *https://www.geeksforgeeks.org/a-search-algorithm/*

3.1 | Berechnung der extra Kosten

Um alle Knoten der zweiten Etage zu berücksichtigen, verwenden wir eine dritte Schleife, die über die verschiedenen Etagen iteriert. Wenn der Knoten zur zweiten Etage gehört, werden die Kosten dieses Knotens um 4 erhöht. Obwohl ein Wechsel zur zweiten Etage nur 3 Sekunden dauert und nicht 4, funktioniert 4 besser, um diese Bedingung zu erfüllen.

```
for (int floor = 0; floor < 2; floor++)
            for (int y = 0; y < nodes[0].length; y++) {
                for (int x = 0; x < nodes[0][0].length; x++) {
                    // Calculate the distance of the node from the start point
                    int originDistance = Math.abs(start.getX() - x) +
Math.abs(start.getY() - y);
                    // Calculate the distance of the node from the end point
                    int goalDistance = Math.abs(end.getX() - x) +
Math.abs(end.getY() - y);

                    // Create a new node with the calculated parameters and assign
it to the array
                    nodes[floor][y][x].setOriginDistance(originDistance);
                    nodes[floor][y][x].setTotalCost(goalDistance + (nodes[floor]
[y][x].isSecondFloor() ? 4 : 0));
                }
            }
```

3.2 | Suche der Nachbarn

Anstatt nur die seitlichen Nachbarn mit den passenden Offsets zu betrachten, suchen wir auch nach dem Knoten, der die gleichen Koordinaten hat, aber auf einer anderen Etage liegt. Diesen Knoten behandeln wir als Nachbarknoten für alle aktuellen Knoten auf dem Pfad.

```
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[0].length, nodes[0][0].length)) {
                    // Get the node from the matrix
                    Node node = nodes[currentNode.isSecondFloor() ? 1 : 0][y][x];

                    // 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.setChild(currentNode);
                        neighbors.add(node);
                    }
                }
            }
            Node node = nodes[currentNode.isSecondFloor() ? 0 : 1]
[currentNode.getY()][currentNode.getX()];

            // 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.setChild(currentNode);
                neighbors.add(node);
            }
        }
```

3.2.1 | Suche der billigsten Nachbarn

Wir sortieren hier alle Nachbarn zuerst nach ihren Gesamtkosten und dann nach ihrer Entfernung zum Ursprung.

```
Comparator<Node> nodeComparator =
Comparator.comparingInt(Node::getTotalCost).thenComparingInt(Node::getOriginDistan
ce);
```

Nachdem wir den Knoten mit den niedrigsten Kosten gefunden haben, speichern wir alle Knoten, die die gleichen Kosten haben könnten. Dann werden alle diese günstigen Knoten als mögliche Pfadknoten akzeptiert.

```
int lowestCost = minNode.get().getTotalCost();
int lowestOriginDistance = minNode.get().getOriginDistance();
// Return a list of nodes that match the minimum value
return neighbors.stream()
        .filter(node -> node.getTotalCost() == lowestCost &&
node.getOriginDistance() == lowestOriginDistance)
        .collect(Collectors.toList());
```

4 | Beispiele

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

4.1 | Gegebene starter Beispiele

Dieser Abschnitt zeigt, dass dieses Programm die gegebenen Beispiele lösen kann. Es werden nur die ersten drei Beispiele in dieser Dokumentation aufgeführt, da die weiteren außer ihrer Größe uninteressant sind.

Eingabe:

```
#############
#.....#.....#
#.###.#.###.#
#...#.#...#.#
###.#.###.#.#
#...#.....#.#
#.#########.#
#.....#.....#
#####.#.###.#
#....A#B..#.#
#.#########.#
#...........#
#############

#############
#.......#...#
#...#.#.#...#
#...#.#.....#
#.###.#.#.###
#.....#.#...#
#####.###...#
#.....#.....#
#.#########.#
#...#.......#
#.#.#.#.###.#
#.#...#...#.#
#############
```

Ausgabe:

```
From point A, switch floor, move 2 times to right, switch floor, now you reached
point B! It did take you 8 seconds
```

Eingabe:

```
######################
#...#.....#...#.....#
#.#.#.###.#.#.#.###.#
#.#.#...#.#.#...#...#
###.###.#.#.#####.###
#.#.#...#.#B....#...#
```

```
#.#.#.###.#.###.#####
#.#...#.#.#..A#.....#
#.#####.#.#########.#
#...................#
#####################

#####################
#.......#.....#.....#
#.###.#.#.###.#.###.#
#.....#.#.#.....#.#.#
#######.#.#######.#.#
#.....#.#.....#...#.#
#.###.#.#.###.###.#.#
#.#.#...#.#...#...#.#
#.#.#######.###.###.#
#...........#.......#
#####################
```

Ausgabe:

```
From point A, move 2 times to left, up, up, now you reached point B! It did take
you 4 seconds
```

Eingabe:

```
#############################################
#...#.....#...........#.#...........#.......#
#.#.#.###.#########.#.#.#.#######.#.#.#####.#
#.#.#...#.#.........#A..#.#.....#.#.#...#...#
###.###.#.#.##############.#.#.###.#.###.#.###
#.#.#...#.#.................B#.#...#.#...#.#.#
#.#.#.###.############.#########.###.#.###.#.#
#.#...#.#.#.........#.#.#.....#.#.....#.#.#.#
#.#####.#.#.#######.#.#.#.###.#.#######.#.#.#
#.....#...#...#.#...#...#.#.#.#.......#.#.#.#
#.#####.#####.#.#.#######.#.#.#######.#.#.#.#
#.....#.......#.#.#.....#.#.#.#...#...#...#.#
#.###.#########.#.###.#.#.#.#.#.#.#.###.###.#
#...#.................#...#.....#...#.......#
#############################################

#############################################
#...#.......#.....#.....#...#...#.....#.....#
#.#.#.#####.###.#.###.#.#.#.#.#.#.###.###.###
#.#.#.....#.#...#.....#...#.#.#...#.#...#...#
###.#.###.#.#.##############.#.#####.###.###.#
#.#.#...#.#.#.#.....#...#.#.#...#.#...#.#...#
#.#.#####.#.#.#.###.#.#.#.#.#.#.#.#.#.#.#.###
#.#...#...#.#.....#.#.#...#.#.#.#...#.#.#...#
#.###.#.###.#.#####.#.###.#.###.#####.#.###.#
#...#.#.#...#...#...#...#.#...#.#.....#.....#
#.###.#.#.#######.#####.#####.#.#.#.#######.#
```

```
#...#...#...#...#.....#.......#.#.#.#.....#.#
#.#.#######.#.#.#####.#.#######.#.###.###.#.#
#.#...........#.......#.........#.......#...#
###############################################
```

Ausgabe:

```
From point A, move 2 times to right, switch floor, move 2 times to right, switch
floor, move 2 times down, to right,
to right, now you reached point B! It did take you 14 seconds
```

4.2 | Eigene Beispiele

Die eigenen Beispiele in diesem Abschnitt werden noch einmal aufzeigen, wie sich der Algorithmus bei der Auswahl von längeren Wegen mit geringerem Zeitaufwand statt kürzerer Wege mit größerem Zeitaufwand entscheidet.

In diesem Beispiel wählt der Algorithmus den kürzesten Weg, indem er die zweite Etage benutzt. Dieser Weg dauert nur 9 Sekunden. Wenn der Algorithmus stattdessen den Weg ohne die Nutzung der zweiten Etage wählen würde, würde er 11 Sekunden brauchen.

Eingabe:

```
######
#A##B#
#.##.#
#.##.#
#.##.#
#....#
######

######
#....#
#....#
#....#
#....#
#....#
######
```

Ausgabe:

```
From point A, switch floor, move 3 times to right, switch floor, now you reached
point B! It did take you 9 seconds
```

Wir verkleinern nun den Plan in diesem Beispiel. Der Algorithmus entscheidet sich hier für den Weg, der die zweite Etage nicht benutzt. Dieser Weg dauert 9 Sekunden. Der Weg über die zweite Etage würde ebenfalls 9 Sekunden dauern. In diesem Fall ist es egal, welchen Weg der Algorithmus wählt.

Eingabe:

```
######
#A##B#
#.##.#
#.##.#
#....#
######

######
#....#
#....#
#....#
#....#
######
```

Ausgabe:

```
From point A, move 3 times down, move 3 times to right, move 2 times up, up,
now you reached point B! It did take you 9 seconds
```

In diesem größeren Beispiel wählt der Algorithmus den schnellsten Weg, indem er zweimal die Etagen wechselt. Dieser Weg dauert nur 24 Sekunden. Der längere Weg, der keine Etagenwechsel beinhaltet, würde 26 Sekunden dauern.

Eingabe:

```
################
#.#...#.########
#.#####.########
#A#..B........#
#.############.#
#.............#
################

################
#...#...########
#.##############
#.#.....########
#.##############
#.......########
################
```

Ausgabe:

```
From point A, switch floor, move 2 times up, move 2 times to right, switch floor,
move 2 times to right,
switch floor, move 2 times to right, switch floor, move 2 times down, to left, to
```

```
left,
now you reached point B! It did take you 24 seconds
```

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 | Modifizierter A-Star

Pfad im Programm Ordner: *src/main/java/de/zauberschule/A_Star.java*

```
public class A_Star {

    // 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(Node[][][] nodes, Node start, Node end) {
        // A list of nodes that are currently being considered for the path
        List<Node> currentNodes = new ArrayList<>();

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

        calculateNodes(nodes, start, end);

        // Loop until the first node in the current nodes list is the end node
        while (!currentNodes.contains(end)) {
            // Get the cheapest neighbors
            List<Node> neighbors =
getCheapestNeighbors(Arrays.stream(nodes).flatMap(Arrays::stream).flatMap(Arrays::
stream)
                    .filter(Node::isVisited).toList(), nodes, end);
            neighbors.forEach(node -> node.setVisited(true));
            //currentNodes.forEach(node -> System.out.println(node.getX() + " " +
node.getY() + " " + node.isSecondFloor() + " cost: " + node.getTotalCost()));
            //System.out.println("current^");
            //neighbors.forEach(node -> System.out.println(node.getX() + " " +
node.getY() + " " + node.isSecondFloor() + " cost: " + node.getTotalCost()));
            //System.out.println("neigh^");

            // If the first neighbor is the end node, clear the current nodes list
and add that neighbor to it
            if (neighbors.contains(end)) {
                currentNodes.clear();
                currentNodes.add(end);
                break;
            } else {
                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) != start) {
            // 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 end) {
        // 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[0].length, nodes[0][0].length)) {
                    // Get the node from the matrix
                    Node node = nodes[currentNode.isSecondFloor() ? 1 : 0][y][x];

                    // 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.setChild(currentNode);
                        neighbors.add(node);
                    }
                }
            }
            Node node = nodes[currentNode.isSecondFloor() ? 0 : 1]
[currentNode.getY()][currentNode.getX()];

            // 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.setChild(currentNode);
                neighbors.add(node);
            }
        }

        // 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::getTotalCost).thenComparingInt(Node::getOriginDistan
ce);
        //neighbors.forEach(node -> System.out.println(node.getX() + " " +
node.getY() + " " + node.isSecondFloor() + " cost: " + node.getTotalCost()));
        Optional<Node> minNode = neighbors.stream().min(nodeComparator);
        //System.out.println(minNode.get().getX() + " " + minNode.get().getY() + "
" + minNode.get().isSecondFloor());
        if (minNode.isPresent()) {
            // Get the minimum cost and origin distance from the minNode
            int lowestCost = minNode.get().getTotalCost();
            int lowestOriginDistance = minNode.get().getOriginDistance();
            // Return a list of nodes that match the minimum values
            return neighbors.stream()
                    .filter(node -> node.getTotalCost() == 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 void calculateNodes(Node[][][] nodes, Node start, Node end) {
        for (int floor = 0; floor < 2; floor++)
            for (int y = 0; y < nodes[0].length; y++) {
                for (int x = 0; x < nodes[0][0].length; x++) {
                    // Calculate the distance of the node from the start point
                    int originDistance = Math.abs(start.getX() - x) +
Math.abs(start.getY() - y);
                    // Calculate the distance of the node from the end point
                    int goalDistance = Math.abs(end.getX() - x) +
Math.abs(end.getY() - y);

                    // Create a new node with the calculated parameters and assign
it to the array
                    nodes[floor][y][x].setOriginDistance(originDistance);
```

```
                    nodes[floor][y][x].setTotalCost(goalDistance + (nodes[floor]
[y][x].isSecondFloor() ? 4 : 0));
                }
            }
        }


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