Aufgabe 5: Hüpfburg

Team-ID: 00382

Team-Name: Sehr gutes BWINF Team

Bearbeiter/-innen dieser Aufgabe:
Paul Franosch, Joshua Benning, Lenny Schürer, Marinus Lentile

20. November 2022

Inhaltsverzeichnis

1 Lösungsidee

Ziel ist es, einen Knoten in einem gerichteten, zyklischen, Graphen zu ermitteln, welcher von den Startkonten mit den Indizes Eins und Zwei gleich weit entfernt ist. Hierzu wird der Graph als Adjazenzmatrix A dargestellt. Es wird sich die Eigenschaft eines solchen Graphens zunutze gemacht, dass die Anzahl der Pfade von Knoten i nach Knoten j der Länge k bestimmt werden kann, indem man die k-te Potenz der Adjazenzmatrix bildet und den Eintrag in der i-ten und j-ten Spalte ausliest. Es wird so lange die nächsthöhere Potenz der Adjazenzmatrix gebildet, bis eine Spalte errechnet wird, welche sowohl in der ersten als auch in der zweiten Zeile, einen Wert ungleich 0 hat. Ist so eine Zeile gefunden, so entspricht die aktuelle Zahl des Exponenten k die Anzahl nötiger Schritte. Der Index der Spalte gibt den Index des gesuchten Knotens z an. Anschließend wird mittels Tiefensuche ein Pfad der Länge k zwischen dem ersten bzw. zweiten Knoten und z ermittelt.
Interessanter sind die Fälle, in denen es nicht möglich ist, einen solchen Knoten zu ermitteln, da dieser nicht existiert. Dieser Fall tritt zwingenderweise dann ein, wenn $A^k = A^{k+j}$ für $k, j \in \mathbb{N}$ gilt, und in den j Schritten kein passender Knoten ermittelt werden konnte. Das ist offensichtlich, wenn man bedenkt, dass in diesem Fall $A^{k+1} = A^{k+j+1}$, $A^{k+2} = A^{k+j+2}$ etc. gelten muss. Es folgt ein sich wiederholendes Muster, in welchem keine neue Information mehr generiert wird.
Tritt allerdings kein solches Muster auf, muss davon ausgegangen werden, dass der Graph schlicht sehr ungünstig aufgebaut ist. Um auch über solche Graphen Aussage treffen zu können, muss die worst-case Laufzeit ermittelt werden, und solange das oben genannte Verfahren angewendet werden, bis entweder der Zielknoten ermittelt werden kann, oder klar ist, dass es nicht möglich ist einen solchen Knoten zu finden (weil dieser nicht existiert). Dies kann entweder passieren, weil der Graph in einem eben besprochenen sich wiederholenden Muster gefangen ist und klar ist, dass keine neuen Fälle auftreten werden, oder aber die für die worst-case Verteilung notwendigen Schritte überschritten wurden.

2 Umsetzung

Die Implementierung erfolgt in Java 17. Zunächst wird der Graph eingelesen und sowohl in Knoten und Kanten, als auch in eine Adjazenzmatrix übersetzt. Die Matrix wird nun iterativ mit der Ausgangsmatrix multipliziert und jeweils abgeglichen, ob ein passender Knoten gefunden wurde. Tatsächlich liefert das Multiplizieren mit der Ursprungsmatrix nicht nur eine Information darüber, ob ein Pfad in k Schritten erreichbar ist, sondern auch über wie viele verschiedene Pfade. Diese Zahl wird allerdings sehr schnell, sehr groß, und sorgt für Probleme. Da diese Information aber auch nicht notwendig ist, da lediglich relevant ist, ob ein Knoten erreicht werden kann, wird zum Zwecke der Performanzoptimierung, jeder Wert, der größer als 1 ist, bei jeder Multiplikation auf 1 gesetzt. Dies verringert den benötigten Speicherplatz und infolge auch die Rechenzeit drastisch.
Um die oben genannten eventuell auftretenden Muster zu erkennen, werden alle Matrizen gemerkt, und beim Einfügen jeweils abgeglichen, ob die einzufügende Matrix bereits gespeichert wurde. Ist dies der Fall, so endet das Programm und gibt aus, welche Matrix welcher vorherigen Matrix gleicht.
Wird ein passender Knoten gefunden, so wird mittels Tiefensuche nach einem Pfad von dem jeweiligen Startknoten zum Zielknoten mit der jeweiligen ermittelten Pfadlänge gesucht. Wird ein Pfad gefunden, wird die Tiefensuche abgebrochen und für den anderen Startknoten das selbe wiederholt. Die gefundenen Pfade sind nicht nur *eine* Lösung, sondern auch die Lösung mit der kürzesten Pfadlänge. Dies ergibt sich aus der iterativ aufsteigenden Matrizenmultiplikation. Wurden beide Pfade gefunden endet das Program und gibt den Zielknoten, die Pfadlänge, sowie die Pfade der Startknoten zum Zielknoten aus. Findet das Programm keine Lösung innerhalb seines errechneten oberen Limits (siehe Unterabschnitt 2.1) endet das Program, und gibt aus, dass es nicht möglich ist, den Parkour zu lösen.

2.1 Laufzeitbetrachtung

Im Folgenden wird die Laufzeit hinsichtlich des Entscheidungsproblems, ob ein Zielknoten existiert, betrachtet. In einem Graphen bestehend aus n Knoten und e Kanten ist die schlecht möglichste Verteilung hinsichtlich der notwendigen Schritte ein Graph, welcher zwei Kreise enthält, die sich in der Länge um eins unterscheiden. Beide Spieler müssen diese Kreise so oft durchlaufen, bis die unterschiedliche Pfadlänge zum Zielknoten ausgeglichen ist. Die Pfadlänge entspricht dann dem kleinsten gemeinsamen Vielfachen, welches dann maximal bezüglich n ist. Dies wäre also ein Kreis der Größe n und ein Kreis der Größe $n-1$.
Für solche Kreise wäre das kleinste gemeinsame Vielfache $n * (n-1) = n^2 - n$.
Das obere Limit für simulierte Schritte ist dementsprechend in der Implementierung n^2.
Für jeden einzelnen Schritt wird je eine Matrizenmultiplikation der Komplexität $\mathcal{O}(n^3)$ durchgeführt.
Zudem wird, um zu überprüfen, ob eine Wiederholung der Matrizen vorliegt, jeweils die aktuelle Matrix gegen alle vorherigen abgeglichen. Dies hat eine Komplexität von $\mathcal{O}\left(n^2\right)$ und wird maximal n^2 mal durchlaufen, es folgt also $\mathcal{O}\left(n^4\right)$.
Insgesamt ergibt sich also eine Komplexität von $\mathcal{O}\left(n^3 \cdot \left(n^2 - n\right) + n^4)\right) = \mathcal{O}\left(n^5\right)$

3 Beispiele

Es folgen die Programmausgaben für die Beispiele der BWINF-Website.

Beispiel	Matrix
0	3
1	4
2	19
3	4
4	10

Tabelle 1: Laufzeit in Millisekunden

3.1 huepfburg0

SOLUTION IS NODE 10 IN 3 STEPS
Pfad von 1 nach 10: [Node[id=1], Node[id=18], Node[id=13], Node[id=10]]
Pfad von 2 nach 10: [Node[id=2], Node[id=19], Node[id=20], Node[id=10]]

3.2 huepfburg1

SOLUTION IS NODE 4 IN 121 STEPS
Pfad von 1 nach 4: [Node[id=1], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=4]]
Pfad von 2 nach 4: [Node[id=2], Node[id=3], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=2], Node[id=3], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=2], Node[id=3], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=2], Node[id=3], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=2], Node[id=3], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=2], Node[id=3], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=2], Node[id=3], Node[id=4], Node[id=5], Node[id=6], Node[id=7], Node[id=8], Node[id=9], Node[id=10], Node[id=11], Node[id=12], Node[id=13], Node[id=14], Node[id=15], Node[id=16], Node[id=17], Node[id=1], Node[id=2], Node[id=3], Node[id=4]]

3.3 huepfburg2

SOLUTION IS NODE 27 IN 8 STEPS
Pfad von 1 nach 27: [Node[id=1], Node[id=51], Node[id=76], Node[id=59], Node[id=42], Node[id=65], Node[id=54], Node[id=92], Node[id=27]]
Pfad von 2 nach 27: [Node[id=2], Node[id=24], Node[id=53], Node[id=2], Node[id=106], Node[id=136], Node[id=108], Node[id=100], Node[id=27]]

3.4 huepfburg3

Es ist nicht möglich den Parkour erfolgreich zu absolvieren, Matrix in Schritt 13 ist identisch zu der vorherigen Matrix in Schritt 5.

3.5 huepfburg4

SOLUTION IS NODE 12 IN 16 STEPS
Pfad von 1 nach 12: [Node[id=1], Node[id=99], Node[id=89], Node[id=88], Node[id=78], Node[id=77],

Node[id=76], Node[id=66], Node[id=56], Node[id=55], Node[id=54], Node[id=44], Node[id=43], Node[id=33], Node[id=32], Node[id=22], Node[id=12]]
Pfad von 2 nach 12: [Node[id=2], Node[id=12], Node[id=11], Node[id=100], Node[id=2], Node[id=12], Node[id=11], Node[id=100], Node[id=2], Node[id=12], Node[id=11], Node[id=100], Node[id=2], Node[id=12], Node[id=11], Node[id=100], Node[id=12]]

4 Quellcode

```
@RequiredArgsConstructor
public class Solver {

    private final int number;
    private final boolean useTestResources;

    private static boolean solutionFound = false;

    public void solve() {
        FileReader fileReader = new FileReader(useTestResources);
        final Graph graph = fileReader.read("huepfburg" + number);

        final int cap = graph.getNodes().size() * graph.getNodes().size();
        long startTime = System.currentTimeMillis();
        Result result = findTargetNode(graph, cap);
        long afterResult = System.currentTimeMillis();
        if (result == null) {
            System.out.println("Der Parkour ist unlösbar.");
            System.out.println("Dauer Zielknoten finden via Matrizen: " + (afterResult - startTime));
            return;
        }

        Node a = new Node(1);
        Node b = new Node(2);

        Set<List<Node>> pathFromA = findPathViaDFS(graph, a, result.node, result.steps);
        solutionFound = false;
        Set<List<Node>> pathFromB = findPathViaDFS(graph, b, result.node, result.steps);
        for (final List<Node> nodes : pathFromA) {
            System.out.println("Pfad von 1 nach " + result.node.id() + ": " + nodes);
        }
        for (final List<Node> nodes : pathFromB) {
            System.out.println("Pfad von 2 nach " + result.node.id() + ": " + nodes);
        }

        System.out.println("Dauer Zielknoten finden via Matrizen: " + (afterResult - startTime));

    }

    private Result findTargetNode(Graph graph, int cap) {
        Matrix adjacencyMatrix = graph.toAdjacencyMatrix();

        List<Matrix> previous = new ArrayList<>();
        Matrix current = adjacencyMatrix;
        for (int i = 1; i < cap; i++) {
            int[] reachableFirst = current.values()[0];
            int[] reachableSecond = current.values()[1];

            for (int j = 0; j < reachableFirst.length; j++) {
                if (reachableFirst[j] != 0 && reachableFirst[j] == reachableSecond[j]) {
                    System.out.println("SOLUTION IS NODE " + (j + 1) + " IN " + i + " STEPS");
                    return new Result(i, new Node(j + 1));
```

```
                }
            }
            int alreadyInSet = isNew(previous, current);
            if (alreadyInSet != -1) {
                System.out.println(
                "Es ist nicht möglich den Parkour erfolgreich zu absolvieren, Matrix in Schritt "
                + (previous.size() + 1) + " ist identisch zu der vorherigen Matrix in Schritt "
                + (alreadyInSet + 1) + ".");
                return null;
            }
            previous.add(current);

            current = MatrixManipulator.multiply(current, adjacencyMatrix);
        }
        System.out.println("Es konnte kein passender Knoten in " + cap + " Schritten ermittelt werden!");
        return null;
    }

    private int isNew(List<Matrix> matrices, Matrix matrix) {
        for (int i = 0; i < matrices.size(); i++) {
            final Matrix current = matrices.get(i);
            if (current.equals(matrix)) {
                System.out.println(matrix + " is the same as matrix with index " + i + ": " + current);
                return i;
            }
        }
        return -1;
    }

    private Set<List<Node>> findPathViaDFS(Graph graph, Node start, Node target, int steps) {
        Set<List<Node>> paths = new HashSet<>();
        findPathViaDFS(graph, start, target, 0, new ArrayList<>(), paths, steps);
        return paths;
    }

    private void findPathViaDFS(Graph graph, Node current, Node target, int depth,
                                List<Node> path, Set<List<Node>> paths, int steps) {
        if (solutionFound) return;
        path.add(current);
        if (depth == steps) {
            if (current.equals(target)) {
                paths.add(path);
                solutionFound = true;
                return;
            }
            return;
        }
        final Set<Node> nodes = graph.getNeighbours().get(current);
        for (Node node : nodes) {
            int depthNew = depth + 1;
            findPathViaDFS(graph, node, target, depthNew, new ArrayList<>(path), paths, steps);
        }
    }

    @RequiredArgsConstructor
    @Getter
    class Result {
        private final int steps;
        private final Node node;
    }

```

```
115 }

117 public class MatrixManipulator {

119     public static Matrix multiply(Matrix a, Matrix b) {

121         final int[][] valuesA = a.values();
            final int[][] valuesB = b.values();
123
            int aRows = valuesA.length;
125         int aColumns = valuesA[0].length;
            int bRows = valuesA.length;
127         int bColumns = valuesA[0].length;

129         if (aColumns != bRows) {
                throw new IllegalArgumentException("A: Rows: " + aColumns
131                     + " did not match B: Columns: " + bRows + ".");
            }
133
            int[][] result = new int[aRows][bColumns];
135
            for (int i = 0; i < aRows; i++) { // aRow
137             for (int j = 0; j < bColumns; j++) { // bColumn
                    for (int k = 0; k < aColumns; k++) { // aColumn
139                     int sum = result[i][j];
                        if (sum == 0) {
141                         // abbrechen nach 1 möglich → laufzeitverbesserung
                            result[i][j] = valuesA[i][k] * valuesB[k][j];
143                     }
                    }
145             }
            }
147
            return new Matrix(result);
149     }

151 }
```