Aufgabe 5: Hüpfburg

Team-ID: 01102
Team: 42
Bearbeiter dieser Aufgabe:
Max Wenk
19. November 2022

Inhaltsverzeichnis

Dokumentation Aufgabe 5: Hüpfburg

19.11.2022

Aufgabenstellung

Sasha und Mika durchlaufen einen Parcours. Dabei gehen sie immer gleichzeitig entlang eines Pfeiles zu einem nächsten Knoten. Der Parcours gilt als erfolgreich absolviert, wenn sich beide auf dem selben Feld treffen. Es soll ein Lösungsansatz gefunden werden, mit dem man überprüfen kann, ob ein beliebiger Parcours mit beliebigen Startknoten absolviert werden kann. Zudem soll der Weg der beiden ausgegeben werden, sofern das möglich ist.

Lösungsidee

Um einen Lösungsansatz für das gestellte Problem zu finden, kann man den Parcours zu einem gerichteten Graphen abstrahieren. Dadurch kann man allgemeine Prinzipien der Graphentheorie anwenden.

Um herauszufinden, auf welchem Knoten sich die Wege von Sasha und Mika kreuzen, betrachtet man anfangs den Startknoten, auf dem sich Sasha bzw. Mika befindet. Daraufhin betrachtet man die Knoten, auf die direkt vom Startknoten gezeigt wird (auf die der Pfeil in der Darstellung auf dem Aufgabenblatt zeigt). Diese Knoten werden in dieser Lösungsidee „Zielknoten“ genannt. Der Zielknoten der ersten Iteration wird in diesem Beispiel Zielknoten erster Ordnung genannt. Danach werden die Zielknoten erster Ordnung von Sasha und Mika miteinander verglichen, und wenn beide mindestens einen gemeinsamen Zielknoten erster Ordnung haben, wurde ein Knoten gefunden, an dem sich beide nach gleich vielen Schritten (in diesem Fall einem) treffen. Dieser Weg kann dann gespeichert werden. Da das jedoch in den meisten Fällen nicht der Fall ist, muss nach weiteren Knoten gesucht werden, um eine potentielle Lösung zu finden. Daraufhin werden die Zielknoten von jedem Zielknoten erster Ordnung betrachtet (=> Zielknoten zweiter Ordnung). Es kann bei diesem Schritt dazu kommen, dass mehrere Zielknoten erster Ordnung auf einen Zielknoten zweiter Ordnung zeigen. Für den Weg der zu diesem Zielknoten zweiter Ordnung führt, bedeutet, dass es mehrere Wege, in einer gleichen Anzahl an Iterationen, zu diesem Punkt gibt. Daraus folgt, dass mindestens zwei Wege in einer gleichen Zahl an Iterationen zum gleichen Zielpunkt n-ter Ordnung führen. Da jedoch nur ein schnellster Weg gesucht ist, muss sich nur ein Wege zu einem spezifischen Punkt n-ter Ordnung gespeichert werden. Dadurch umgeht man auch das Problem einer exponentiell ansteigenden Datenmenge, wenn man versuchen würde, alle möglichen Wege zu speichern und zu überprüfen. Daraufhin werden die Zielknoten zweiter Ordnung von Sasha und Mika wieder verglichen. Wenn ein gemeinsamer Knoten gefunden wurde, gibt es einen Lösungsweg. Falls nicht wird dieser Schritt wiederholt. Allgemein formuliert würde das bedeuten: Es werden die Zielknoten von jedem Zielknoten n-ter Ordnung betrachtet (=> Zielknoten n+1-ter Ordnung). Dabei kann es wiederum dazu kommen, dass mehrere Zielknoten n-ter Ordnung auf einen Zielknoten n+1-ter Ordnung zeigen, woraus wiederum folgt, dass es mehrere Wege, in der gleichen Anzahl an Iterationen, zum gleichen Zielnoten n+1-ter Ordnung gibt. Jedoch muss wieder nur ein Weg gespeichert werden, da nur nach dem kürzesten gefragt ist, was dem Problem der potentiell stark ansteigenden Datenmenge vorbeugt. Danach werden die Zielknoten n+1-ter Ordnung von Sasha und Mika miteinander verglichen. Wenn die Zielknoten n+1-ter Ordnung von beiden gleich sind, wurde ein gemeinsamer Knoten in einer gleichen Anzahl an Iterationen gefunden (da beide Zielknoten in der Iteration n+1 sind). Wenn kein gemeinsamer Knoten gefunden wurde, wird dieser Schritt wiederholt (=> Zielknoten n+2-ter Ordnung == Zielknoten n+1-ter Ordnung).

Dokumentation Aufgabe 5: Hüpfburg

19.11.2022

Umsetzung

Im ersten Schritt müssen die Daten für den Parcours (folgend Graph) in einer sinnvollen Weise gespeichert werden. Dazu müssen die Daten in der gegebenen Textdatei richtig interpretiert werden. Die erste Zahl in der ersten Zeile gibt die Anzahl der unterschiedlichen Knoten in dem Graphen an, und die zweite Anzahl stellt die Anzahl der gesamten Verbindungen von einem Knoten zu einem anderen Knoten dar. Ab der zweiten Zeile gibt die erste Zahl immer einen Knoten an, und die zweite Zahl einen Knoten, mit dem diese verbunden ist. Eine Darstellung die sich dafür anbieten würde, wäre ein zwei-dimensionaler Array, in dem für jeden Knoten in dem Graphen alle Verbindungen zu anderen Knoten angegeben wird (frei nach der Form [Knoten][Verbindung]). In Java kann das relativ einfach mit einer 2D-Array-Liste erreicht werden. Dazu wird ab der zweiten Zeile durch die Textdatei iteriert, und zu jedem Knoten, den es in diesem Graphen gibt, eine Verbindung hinzugefügt. Daraufhin kann dann die eigentliche Umsetzung des Lösungsansatzes umgesetzt werden (in Abb. 1 kann die Implementierung in Java betrachtet werden).

Aus effizienz-technischen Gründen wird im Programm erst überprüft, ob es überhaupt einen gemeinsamen Knoten gibt, und erst danach wird ein Weg dazu gefunden. Dadurch kann das Programm schneller Iterationen bzw. Treffknoten berechnen, wodurch schneller festgestellt werden kann, ob es vielleicht gar keine Lösung gibt.

Dazu werden zuerst die Zielknoten der Startknoten betrachtet, und in einer Array-Liste gespeichert (bietet sich an, da die Länge der Liste dynamisch ist, und pro Iteration eine unterschiedliche Menge an Zielknoten anfallen kann). Daraufhin werden die Zielknoten von Sasha und Mika (Sasha und Mika haben jeweils eigene Array-Listen in denen die Werte ihrer Zielknoten stehen) auf gemeinsame Knoten überprüft. Wenn es zu Gemeinsamkeiten kommt, wird direkt zum nächsten Schritt, der Wegbestimmung übergegangen. Wenn nicht, werden wieder die Zielknoten der Zielknoten betrachtet, und in der Array-Liste gespeichert (bzw. die alte Array-Liste wird mit den neuen Werten überschrieben). Wenn es zu Dopplungen der Zielknoten kommt, werden die Duplikate entfernt, da es sonst bei der Wiederholung (wie in der Lösungsidee) zu Dopplungen in der Lösung kommt, und das zu einem ungewollten Verhalten des Programms führen kann. Daraufhin werden dann wieder die Listen mit Zielknoten von Sasha und Mika auf gemeinsame Knoten überprüft. Dieser Schritt wird (Anzahl der Knoten)*(Anzahl der Verbindungen) oft wiederholt, um auszuschließen, dass es nicht bei einer relativ hohen Anzahl an Iterationen zu einem gültigen Weg kommt (in Abb.2 kann die Implementierung in Java betrachtet werden).

Bei der Wegbestimmung wird der vorherige Schritt wiederholt, aber nur so oft, wie es zuvor Iterationen gab, um einen gemeinsamen Weg zu finden. Die zuvor beschriebenen Schritte werden wiederholt, jedoch wird zusätzlich in einer 2D-Array-Liste der Weg zu jedem aktuellen Zielknoten geschrieben, nach der Form [Knoten][Wege zu diesem Knoten (Startknoten, …, Knoten)].

Am Ende wird dann die Anzahl der Iterationen die gebraucht wurden, um einen Weg zu finden, der Knoten an dem sie sich treffen und die Wege der beiden ausgegeben, bzw. es wird eine Exception ausgegeben, wenn es keine gemeinsamen Knoten gibt.

Dokumentation Aufgabe 5: Hüpfburg

19.11.2022

Beispiele

1. Beispiel auf dem Aufgabenblatt

Der Startknoten von Sasha ist 1, und der von Mika ist 2.
Nach der ersten Iteration kann sich Sasha auf den Knoten 4, 8 oder 18 befinden. Mika kann sich auf Knoten 3 oder 19 befinden.
Nach der zweiten Iteration kann sich Sasha auf den Knoten 4, 7, 13 oder 18 befinden, da dieser von den letzten Knoten aus weiter geht. Währenddessen kann sich Mika auf den Knoten 6, 19 oder 20 befinden.
Nach der dritten Iteration kann sich Sasha auf Knoten 5, 7, 8, 10 oder 13 befinden. Es zeigen zwar mehrere Wege von den vorherigen Knoten auf den Knoten 7, jedoch muss dieser auch nur einmal gespeichert werden, da nur ein schnellster Weg gesucht ist. Mika kann sich währenddessen auf den Knoten 2, 3, 10, 12 oder 20 befinden. Da sich beide in der gleichen Iteration auf dem Knoten 10 befinden können, wurde ein Lösung für diesen Parcours gefunden.
Sie treffen sich nach drei Iterationen auf Knoten 10. Der jeweilige Weg der zu diesem Knoten führt muss dann nur noch gespeichert werden. Ein Weg für Sasha wäre 1 → 18 → 13 → 10 und ein Weg für Mika wäre 2 → 19 →20 → 10. Die Ausgabe im Programm sieht etwas anders aus (es wird nur die Anzahl der Iterationen, der Knoten auf dem sie sich treffen und der jeweilige Weg ausgegeben).

2. Beispiel mit anderen Startpunkten

Der Startknoten von Sasha ist 5, und der von Mika ist 6.
Nach der ersten Iteration kann sich Sasha auf den Knoten 12, 13 oder 15 befinden. Mika kann sich auf den Knoten 2 oder 12 befinden. Da sie sich in der gleichen Iteration auf einem gleichen Knoten befinden können, wurde eine Lösung für den Parcours nach einer Iteration gefunden.
Die beiden treffen sich nach einer Iteration auf Knoten 12. Ein Weg für Sasha wäre 5 → 12 und für Mika 6 → 12.

3. Beispiel mit anderen Startpunkten

Der Startknoten von Sasha ist 7, und der von Mika ist 15.
Nach der ersten Iteration kann sich Sasha auf den Knoten 5 und 8 befinden. Mika kann sich auf dem Knoten 12 befinden.
Nach der zweiten Iteration kann sich Sasha auf den Knoten 4, 12, 13, 15 und 18 befinden. Mika kann sich währenddessen nur auf Knoten 3 befinden.
Nach der dritten Iteration kann sich Sasha auf den Knoten 3, 7, 10, 12 oder 13 befinden. Mika kann sich nur auf Knoten 6 befinden.
Nach der vierten Iteration kann sich Sasha auf den Knoten 3, 5, 6, 7, 8, 10, 12, 16 oder 18 befinden. Währenddessen kann sich Mika auf den Knoten 2 oder 12 befinden. Da sie sich wieder in der gleichen Iteration auf einem gleichen Knoten befinden können, wurde auch wieder für diesen Parcours eine Lösung gefunden.
Sie treffen sich nach vier Iterationen auf Knoten 12. Ein für Weg für Sasha lautet 7 → 5 → 13 → 10 → 12 und für Mika 15 → 12 → 3 → 6 → 12.

4. Beispiel mit anderem Parcours

Für dieses Beispiel wird der Parcours aus „huepfburg4.txt“ genommen.
Der Startknoten von Sasha ist 1, und der von Mika ist 2.

Dokumentation Aufgabe 5: Hüpfburg

19.11.2022

Nach der ersten Iteration kann sich Sasha auf Knoten 99 befinden. Mika kann sich auf Knoten 12 befinden.
Nach der zweiten Iteration kann sich Sasha auf den Knoten 89 oder 98 befinden. Mika kann sich auf Knoten 11 befinden.
Nach der dritten Iteration kann sich Sasha auf den Knoten 79, 88, 90 oder 97 befinden. Mika kann sich auf Knoten 100 befinden.
Nach der vierten Iteration kann sich Sasha auf Knoten 1, 78, 80, 87 oder 98 befinden. Mika kann sich auf den Knoten 2 oder 12 befinden.
Das ganze würde so weiter gehen ...
Die beiden treffen sich nach der 16. Iteration auf Knoten 12. Ein möglicher Weg für Sasha wäre 1 → 99 → 89 → 79 → 78 → 77 → 76 → 66 → 56 → 55 → 54 → 44 → 43 → 33 → 23 → 13 → 12. Ein möglicher Weg für Mika wäre 2 → 12 → 11 → 100 → 12 → 11 → 100 → 12 → 11 → 100 → 12 → 11 → 100 → 12 → 11 → 100 → 12.

Anmerkung zum Programm

Das Programm, mit dem eine Lösung (oder auch keine) für einen beliebigen Parcours gefunden werden kann, ist in Java geschrieben. Sie ist in dem „Aufgabe_5“ Ordner gespeichert und heißt „Main“. Es gibt zwei Versionen des Programms. Das Originale Programm ist auf Englisch verfasst und hat auch eine englische Ein- bzw. Ausgabe. Das zweite Programm ist auf Deutsch übersetzt und hat das Suffix „DE“, und ist auf Englisch verfasst, jedoch ist die Ein- bzw. Ausgabe auf Deutsch. Die Dateien sind jeweils als „.java“ vorhanden.

Der Parcours im vorgegebenen Format in der „parcours.txt“ Datei stehen. Diese Datei muss sich im gleichen Ordner wie auch das Programm befinden.

Nachdem das Programm gestartet wurde, und die erste Eingabeaufforderung erscheint, muss der erste Startknoten eingegeben werden, danach wird der zweite Startknoten eingegeben. Die Eingabewerte müssen ganzzahlige Werte sein, die auch Knoten im Parcours darstellen. Danach berechnet das Programm, ob es einen absolvierbaren Parcours gibt. Wenn nicht, wird ausgegeben, dass es zu keiner Lösung kommt. Wenn es einen gibt, wird eine mögliche Lösung für den Parcours ausgegeben. Als erstes wird ausgegeben, nach wie vielen Iterationen und auf welchem Knoten sie sich treffen. Danach wird der jeweilige Weg ausgegeben, startend mit dem Startknoten, dann die folgenden Knoten und am Schluss Zielknoten auf dem sie sich treffen. Es wird ein Knoten im Weg mehr ausgegeben als es Iterationen gibt, da die 0-te Iteration (der Startknoten) auch dargestellt wird.

Dokumentation Aufgabe 5: Hüpfburg

19.11.2022

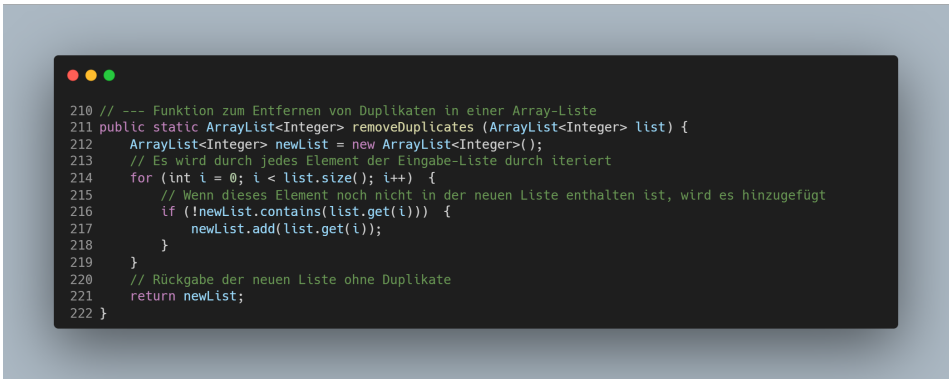
Abbildungen

Die Abbildungen sind noch als Bilder separat angefügt.

Abb. 1:

```java
// Initialisieren der Variablen um die Knoten nach einer beliebigen Anzahl an Iterationen zu speichern
ArrayList<Integer> path1 = new ArrayList<Integer>();
ArrayList<Integer> path2 = new ArrayList<Integer>();

// --- Überprüfung auf einen gemeinsamen Knoten (Maximum sind Knoten*Verbingen Iterationen)
// Die erste Iteration muss manuel ausgeführt werden, da die Knoten-Arrays noch leer sind
// In der "connections" Variable sind die Knoten in folgender Form gespeichert [Knoten][Verbindung]
for (int j = 0; j < readFile[0][0]; j++) {  // An dieser Position sind in der Textdatei die Anzahl der
                                            // unterschiedlichen Knoten gespeichert
    if (connections[j][0].get(0) == startingPoint1) {
        path1 = connections[j][1];
    }
    if (connections[j][0].get(0) == startingPoint2) {
        path2 = connections[j][1];
    }
}

vaildPath:
for (int i = 0; i < readFile[0][0] * readFile[0][1]; i++) {
    // Initialisieren von temporären Listen-Arrays um die Zielknoten n+1-ter Ordnung zu speichern
    ArrayList<Integer> path1Save = new ArrayList<Integer>();
    ArrayList<Integer> path2Save = new ArrayList<Integer>();

    // Zurücksetzen der temporären Variablen
    path1Save.clear();
    path2Save.clear();

    // Die Zielknoten der Zielknoten n-ter Ordnung werden zu den temporären Listen-Arrays hinzugefügt
    // Das wird bei beiden Wegen (dem von Sasha und Mika) separat durchgeführt
    for (int j = 0; j < path1.size(); j++) {
        for (int k = 0; k < readFile[0][0]; k++) {
            if (path1.get(j) == connections[k][0].get(0)) {
                path1Save.addAll(connections[k][1]);
            }
        }
    }
    for (int j = 0; j < path2.size(); j++) {
        for (int k = 0; k < readFile[0][0]; k++) {
            if (path2.get(j) == connections[k][0].get(0)) {
                path2Save.addAll(connections[k][1]);
            }
        }
    }

    // Entfernen von Duplikaten aus den temporären Zielknoten n+1-ter Ordnung
    // Die eigentlichen Zielknoten werden auf die der temporären gesetzt => für die nächste Iteration
    path1 = removeDuplicates(path1Save);
    path2 = removeDuplicates(path2Save);

    // Überprüfung ob Pfad 1 und Pfad 2 gemeinsame Knoten hat, wenn ja wurde ein gemeinsamer Weg
    // gefunden, wenn nicht, wiederholt sich dieser Schritt
    for (int j = 0; j < path1.size(); j++)  {
        if (path2.contains(path1.get(j)))  {
            pathIteration = i +2;   // Es muss +2 addiert werden, da die ersten zwei Iterationen schon
                                    // vor der Schleife geschen sind
            pathPoint = path1.get(j);   // Der gemeinsame Knoten wird gespeichert
            pathFound = true;
            break validPath;        // Es wird aus der äußersten for-Schleife rausgesprungen
        }
    }
}
```

Abb. 3:



```java
// --- Funktion zum Entfernen von Duplikaten in einer Array-Liste
public static ArrayList<Integer> removeDuplicates (ArrayList<Integer> list) {
    ArrayList<Integer> newList = new ArrayList<Integer>();
    // Es wird durch jedes Element der Eingabe-Liste durch iteriert
    for (int i = 0; i < list.size(); i++)  {
        // Wenn dieses Element noch nicht in der neuen Liste enthalten ist, wird es hinzugefügt
        if (!newList.contains(list.get(i)))  {
            newList.add(list.get(i));
        }
    }
    // Rückgabe der neuen Liste ohne Duplikate
    return newList;
}
```

Abb. 2:

```
// --- Finden und speichern eines gültigen Weges
// Initialisierung von temporären und resultierenden Variablen zur Speicherung der möglichen Wege
ArrayList<Integer>[][] path1PossiblePaths = new ArrayList[readFile[0][0]][2];
ArrayList<Integer>[][] path2PossiblePaths = new ArrayList[readFile[0][0]][2];
ArrayList<Integer>[][] path1PossiblePathsSave = new ArrayList[readFile[0][0]][2];
ArrayList<Integer>[][] path2PossiblePathsSave = new ArrayList[readFile[0][0]][2];

// Nur wenn auch ein gemeinsamer Knoten (wenn es eine Lösung gibt), wird auch nach einem Wege gesucht
if (pathFound) {
    // Deklarierung der Wege-Variablen als leere Listen, zum Teil auch mit einer festen Länge
    for (int i = 0; i < readFile[0][0]; i++) {
        path1PossiblePaths[i][0] = new ArrayList<>(1);  // An dieser Stelle wird der Knoten gespeichert
                                                        // => hat nur ein Element
        path1PossiblePaths[i][1] = new ArrayList<>();
        path2PossiblePaths[i][0] = new ArrayList<>(1);
        path2PossiblePaths[i][1] = new ArrayList<>();
        path1PossiblePathsSave[i][0] = new ArrayList<>(1);
        path1PossiblePathsSave[i][1] = new ArrayList<>();
        path2PossiblePathsSave[i][0] = new ArrayList<>(1);
        path2PossiblePathsSave[i][1] = new ArrayList<>();
    }

    // Die erste Iteration muss, wie in der anderen Erklärung auch, manuell gesetzt werden
    for (int j = 0; j < readFile[0][0]; j++) {
        if (connections[j][0].get(0) == startingPoint1) {
            path1 = connections[j][1];
        }
        if (connections[j][0].get(0) == startingPoint2) {
        path2 = connections[j][1];
        }
    }

    // --- Berechnen der Wege (da die Anzahl der Iterationen für eine Lösung schon berechnet wurde,
    // wird diese auch als Maximum in der for-Schleife benutzt)
    for (int i = 0; i < pathIteration -1; i++) {
        /* ...
        An dieser Stelle ist der Programmcode gleich wie in der anderen Programm-Erklärung.
        An dieser Stelle steht das Gleiche wie von Zeile 20-43 in dem anderen Programm-Erklärung.
        */

        // Duplikate werden wieder entfernt, aber in der temporären Variable gespeichert
        path1Save = removeDuplicates(path1Save);
        path2Save = removeDuplicates(path2Save);

        // Der erste Iteration der für die Wegedaten muss manuel durchgeführt werden, da die Variablen
        // für weitere Berechnungen benötigt werden
        for (int j = 0; j < readFile[0][0]; j++) {
            if (connections[j][0].get(0) == startingPoint1 && i == 0) { // Wird nur in der 0 Iteration
                                                                        // ausgeführt
                path1 = connections[j][1];
                // Die möglichen Wege zu einem Knoten werden zu den Wegedaten hinzugefügt
                for (int k = 0; k < path1.size(); k++) {
                    path1PossiblePaths[k][0].add(path1.get(k));
                    path1PossiblePaths[k][1].add(startingPoint1);
                    path1PossiblePaths[k][1].add(path1.get(k));
                }
            }
            // Das gleiche wie zuvor, nur für den zweiten Weg
            if (connections[j][0].get(0) == startingPoint2 && i == 0) {
                path2 = connections[j][1];
                for (int k = 0; k < path2.size(); k++) {
                    path2PossiblePaths[k][0].add(path2.get(k));
                    path2PossiblePaths[k][1].add(startingPoint2);
                    path2PossiblePaths[k][1].add(path2.get(k));
                }
            }
        }

        // Berechnen des Weges zu einem Knoten n+1-ter Ordnung von einem Knoten n-ter Ordnung
        // Dabei muss immer nur ein Weg gespeichert werden, der zu einem Knoten n-ter Ordnung führt
        for (int j = 0; j < path1Save.size(); j++) {
            // Dazu wird durch die Knoten n+1-ter Ordunung iteriert und auf das erste Element eines
            // möglichen Weges gesetzt
            path1PossiblePathsSave[j][0].add(path1Save.get(j));
            outer:
            for (int k = 0; k < readFile[0][0]; k++) {
                if (connections[k][1].contains(path1Save.get(j))) {
                    for (int l = 0; l < path1.size(); l++) {
                        if (connections[k][0].get(0) == path1.get(l)) {
                            // Der Weg, der zu dem vorherigen Knoten (der zum aktuellen Knoten zeigt)
                            // gehört, wird zum aktuellen hinzugefügt
                            path1PossiblePathsSave[j][1].addAll(path1PossiblePaths[l][1]);
                            break outer;    // Danach wird aus der Schleife gesprungen, da nur ein Weg
                                            // gesucht ist
                        }
                    }
                }
            }
            // Danach wird noch der aktuelle Knoten hinzugefügt, um den gesamten Weg darzustellen
            path1PossiblePathsSave[j][1].add(path1Save.get(j));
        }
        // Das gleiche wie zuvor, nur für den zweiten Weg
        for (int j = 0; j < path2Save.size(); j++) {
            path2PossiblePathsSave[j][0].add(path2Save.get(j));
            outer:
            for (int k = 0; k < readFile[0][0]; k++) {
                if (connections[k][1].contains(path2Save.get(j))) {
                    for (int l = 0; l < path2.size(); l++) {
                        if (connections[k][0].get(0) == path2.get(l)) {
                            path2PossiblePathsSave[j][1].addAll(path2PossiblePaths[l][1]);
                            break outer;
                        }
                    }
                }
            }
            path2PossiblePathsSave[j][1].add(path2Save.get(j));
        }

        // Löschen der Array-Liste der möglichen Wege -> um unvorhergesehenen Fehlern vorzubeugen, da
        // im nächsten Schritt möglicherweise nicht alle alten Daten überschrieben werden
        for (int j = 0; j < readFile[0][0]; j++) {
            path1PossiblePaths[j][0].clear();
            path1PossiblePaths[j][1].clear();
        }

        // Setzen der Wegedaten auf das Resultat der Ergebnisse aus den vorherigen Berechnungen (für
        // zukünftige Berechnungen oder für die Lösung) und löschen der Daten aus den temporären
        // Variablen (das muss für jedes Element geschehen, da es sonst zu Fehlern führen kann)
        for (int j = 0; j < path1Save.size(); j++) {
            path1PossiblePaths[j][0].add(path1PossiblePathsSave[j][0].get(0));
            path1PossiblePaths[j][1].addAll(path1PossiblePathsSave[j][1]);
            // Löschen der Daten aus der temporären Variable
            path1PossiblePathsSave[j][0].remove(0);
            for (int k = path1PossiblePathsSave[j][1].size() -1; k >= 0; k--) {
                path1PossiblePathsSave[j][1].remove(k);
            }
        }
        // Das gleiche wie zuvor, nur für den zweiten Weg
        for (int j = 0; j < readFile[0][0]; j++) {
            path2PossiblePaths[j][0].clear();
            path2PossiblePaths[j][1].clear();
        }
        for (int j = 0; j < path2Save.size(); j++) {
            path2PossiblePaths[j][0].add(path2PossiblePathsSave[j][0].get(0));
            path2PossiblePaths[j][1].addAll(path2PossiblePathsSave[j][1]);
            path2PossiblePathsSave[j][0].remove(0);
            for (int k = path2PossiblePathsSave[j][1].size() -1; k >= 0; k--) {
                path2PossiblePathsSave[j][1].remove(k);
            }
        }
        // Die eigentlichen Zielknoten werden wieder auf die der Temporären gesetzt -> für die nächste
        // Iteration, bzw für das Ergebnis
        path1 = path1Save;
        path2 = path2Save;
    }
}
```