Aufgabe 5: Hüpfburg

Teilnahme-ID: 66432

Team-ID: 00662

Bearbeiter/-in dieser Aufgabe:
Linus Schumann

20. November 2022

Inhaltsverzeichnis

1 Lösungsidee

1.1 Feststellung des Problems

Das Problem bei dieser Aufgabe liegt darin, dass Sasha und Mika gleichzeitig auf dem selben Feld laden sollen. Sasha startet auf Feld 1 (x_1) und Mika auf Feld 2 (x_2). Sie bewegen sich auf einem Parcour, der als ungewichteter gerichteter Graph dargestellt werden kann. Die Anzahl an Knoten wird im Folgenden immer als n und die Anzahl an Kanten immer als m bezeichnet. Für n gilt $2 \leq n \leq 150$ und für m gilt $1 \leq m \leq 347$. Dies lässt sich dem größten Beispiel (Abschnitt 3.1.3) entnehmen. Die Knoten sind von 1 bis n durchnummeriert und werden immer als x_i (mit $1 \leq i \leq n$) bezeichnet. Auffällig ist dabei, dass n nicht sehr groß ist und es daher wahrscheinlich keine sehr schnelle Lösung gibt, da es sonst größere Beispiele geben würde.
Außerdem muss im Folgenden beachtet werden, dass der Graph mehr als $n-1$ Kanten hat und damit keine Baumstruktur aufweißt. Daher können im Graphen Zyklen enthalten sein. Auch gibt es keine Einschränkung zur mehrfachen Nutzung von Knoten innerhalb eines Pfades.

1.2 Erste Idee

Die erste Idee um dieses Problem nun zu lösen, besteht darin von beiden Startknoten (x_1 und x_2) aus alle möglichen Pfade zu jedem anderen Knoten zu finden. Dannach kann man dann die Länge der Pfade vergleichen und bei gleicher Länge beide Pfade ausgeben. Es kann dabei allerdings Pfade geben in denen ein bestimmter Knoten nur mit der passenden Distanz erreicht werden kann, wenn Knoten mehrfach genutzt werden. Dies funktiert da der Graph zyklisch sein kann. Daher kann die Idee nicht mit einer einfachen Tiefensuche implementiert werden, die den aktuellen Pfad speichert, da kein Abbruch Kriterium definiert wäre. Bei einer "normalen" Tiefensuche würde dazu Vector genutzt werden, in denen die bereits besuchten Knoten gespeichert werden würden.
Durch diese Vorraussetzungen ist die intuitivste Abbruchbedingung eine maximale Tiefe $tMax$ zu definieren, die ein Pfad maximal lang sein darf.

1.3 Optimierung dieser Idee

Da diese Lösung vor allem für längere Pfade sehr sehr langsam ist, macht es Sinn über eine mögliche Optimierung nachzudenken.
Diese Optimierung besteht darin, dass man für jeden Startknoten (x_1 und x_2), eine dreidimensionale Matrix erstellt. Diese beiden Matritzen werden nun a und b genannt. In dieser wird für jeden anderen Knoten für jede Distanz (bis zur maximalen Distanz) ein Pfad als Vector gespeichert. Dabei befindet sich der Pfad Vector zum Knoten i (für $1 \leq i \leq n$) mit der Länge j (für $1 \leq j \leq tMax$) an der Stelle $a_{i,j}$.
Beim Starten des Algorithmus ist wird die Matrix a und b mit leeren Vectoren für $a_{i,j}$ erstellt. Dann kann die Tiefensuche aus der ersten Idee (Abschnitt 1.2) für beide Startknoten genutzt werden. Dabei wird bei jedem Schritt der Tiefensuche überprüft, ob schon ein Pfad zu diesem Knoten x_i und mit der selben Länge t gegeben ist. Nur wenn es noch keinen Vector in a bzw. b an der Stelle $a_{i,t}$ bzw. $b_{i,t}$ gibt, wird der aktuelle Pfad an dieser Stelle gespeichert. Dadurch werden keine doppelten Pfade berechnet und die Laufzeit wird deutlich verbessert.

1.4 Finden der kürzesten Pfade

Am Schluss hat man durch die Verwendungung der beiden Matritzen außerdem den Vorteil, dass man direkt die minimalen Pfade herausfinden kann. Dazu werden die beiden Matritzen so verglichen, dass für eine Distanz t ($1 \leq t \leq tMax$) zuerst jeder Endknoten x_i ($1 \leq i \leq n$) überprüft wird und danach erst die Distanz erhöht wird. Falls also $|a_{i,t} > 0|$ und $|b_{i,t} > 0|$ gilt, wurde eine Lösung gefunden, da falls es einen Pfad an der Stelle i, t gibt $|a_{i,t}| = t$ und $|b_{i,t}| = t$ gilt.

2 Umsetzung

2.1 Allgemeines

Im Folgenden wird die Umsetzung der in Abschnitt 1 beschriebene Lösungsidee, näher erläutert. Grundsätzlich wurde diese Idee dabei in C++, genauer gesagt in der Datei "Aufgabe_5.cpp" implementiert. Diese Datei befindet sich im Verzeichnis "./source/".

Um das implementierte Programm zu starten kann das Batch Skript "Aufgabe_5.bat" im Verzeichnis "./executables/" genutzt werden. Dieses startet das von mir für Windows kompilierte Programm ("Aufgabe_5.exe"). Für andere Betriebssysteme müsste die Source-Datei erneut auf dem entsprechenden Rechner kompiliert werden.

Im Verzeichnis "./beispieleingaben/" befinden sich alle in dieser Dokumentation aufgeführten Beispiele und im Verzeichnis "./beispielausgaben/" befinden sich dementsprechend die gesicherten Ausgaben. Letztere werden mit der Datei-Endung ".out" gespeichert. Außerdem werden bei jeder Ausgabe auch noch zwei ".dot" und zwei ".png" Dateien gespeichert. Näheres zu diesen Dateien wird in Abschnitt 2.3 (Visualisierung) genannt.

2.2 Implementation der Lösungsidee

Nun werden die einzelnen Bestandteile der Implementation näher erläutert.

2.2.1 Einlesen des Graphen

Im ersten Schritt der Implementation wird der Graph aus der Eingabedatei eingelesen. Um den Graph zu speichern, wird eine Adjazenzliste verwendet. Dabei wird für jeden Knoten eine Liste mit den Nachbarknoten erstellt.
Im Vergleich zur Adjazenzmatrix bringt dies den Vorteil, dass die Speicherkomplexität deutlich geringer ist. Genauer gesagt liegt diese bei der Adjazenzliste bei $\mathcal{O}(n + m)$ und bei der Adjazenzmatrix würde diese bei $\mathcal{O}(n^2)$ liegen.

2.2.2 Für jede Distanz Pfade finden

Nach dem Einlesen des Graphen wird wie in der Lösungsidee beschrieben zwei mal eine Tiefensuche ausgeführt, die alle Pfade im Graphen findet, die beim Knoten 0 bzw. 1 starten. Dies wurde in der Funktion "dfs" implementiert. Um diese Tiefensuche zu implementieren wird jeweils eine "Pfade" Matrix erstellt, der für jeden Knoten alle Pfade mit unterschiedlichen Distanzen speichert. Bei jedem Aufruf der Funktion wird dazu geprüft, ob schon ein Pfad mit dieser Länge zu diesem Knoten in dem "Pfad" Vektor gespeichert ist.
Außerdem muss die Funktion natürlich immer den aktuellen Pfad speichern, dazu wird ein positiver Effekt der Rekursion genutzt. Der Pfad wird nämlich in einem Vektor gespeichert, der bei jedem Aufruf der Funktion als Referenz übergeben wird. Die Rekursion bietet dann den Vorteil, dass zuerst der aktuelle Knoten an das Ende des Vektor gehängt wird. Dann wird die Funktion für alle Nachbarknoten aufgerufen und danach der aktuelle Knoten wieder aus dem Vektor entfernt wird. Aus diesem Grund ist der Pfad immer aktuell.

2.2.3 Gleiche und kürzeste Pfade finden

Nach dem Füllen der beiden "Pfade" Matritzen können diese ausgewertet werden. Dazu werden zwei for-Schleifen genutzt. Die Erste iteriert über alle Distanzen und die zweite über alle Knoten. Wenn zwei Pfade an der selben Stelle gefunden werden, wurde die beste Lösung gefunden und die Schleifen werden unterbrochen.

Dieser Schritt hat auf Grund der zwei for-Schleifen eine maximale Laufzeit von $\mathcal{O}(1000 * n)$ (Die 1000 würden theoretisch wegfallen).

2.3 Visualisierung

Um die unterschiedlichen Graphen zu visualiseren wurde das Programm Graphviz genutzt. Dieses muss auf dem ausführenden System installiert sein, da sonst kein PNG-Bild generiert werden kann. Dazu wird nämlich am Ende der main-Funktion zweimal der Befehl "dot" aufgerufen, um aus den beiden vorher generierten ".dot" Dateien ein Bild zu machen.

Diese beiden ".dot" Dateien werden vorher im Programm generiert und beschreiben einfach den Aufbau des Graphen. Dabei wird zuerst der Graphtyp auf "digraph" (gerichteter Graph) gesetzt, dann werden

alle Knoten aufgeführt und dann alle Kanten.

Die erste ".dot" Datei beschreibt einfach den Graphen, wie er in der Eingabedatei vorliegt.
In der zweiten Datei werden dann alle Kanten der zwei Pfade farbig dargestellt. Außerdem wird neben diesen Kanten ein Text generiert, der angibt wie oft eine Kante in dem entsprechenden Pfad genutzt wurde. Bei doppelten Nutzung einer Kante (vom Startpunkt x_1 und x_2), wird diese einfach dupliziert und zwei Mal gezeichnet.

3 Beispiele

In diesem Abschnitt befinden sich alle Ausgaben zu allen eigenen und vorgegebenen Beispielen. Dabei wird bei Beispiel 0 und 1 auch in jeweils einer Grafik der Ausgangsgraph sowie der Graph mit eingezeichneten Linien gezeigt. Bei Beispiel 2-5 wird dann nur noch eine der beiden Grafiken geweils dargestellt. Die Genaue Umsetzung dieser Visualisierung wurde im Abschnitt Umsetzung unter Visualisierung (Abschnitt 2.3) näher erläutert.
Um einen mögliche Zyklen besser in den Grafiken zu erkennen, wird die Anzahl an Verwendungungen im jeweiligen Pfad durch einen kleinen Text neben der jeweiligen Kante definiert. Außerdem wird bei der Nutzung einer Kante durch Pfad 1 und 2, die Kante zweimal angezeigt, damit beide Pfade nachvollziehbar sind.
Damit dieser Abschnitt nicht zu lang wird, wird nicht für jedes Beispiel jede Grafik eingefügt.

3.1 Vorgegebene Beispiele

3.1.1 Beispiel 0

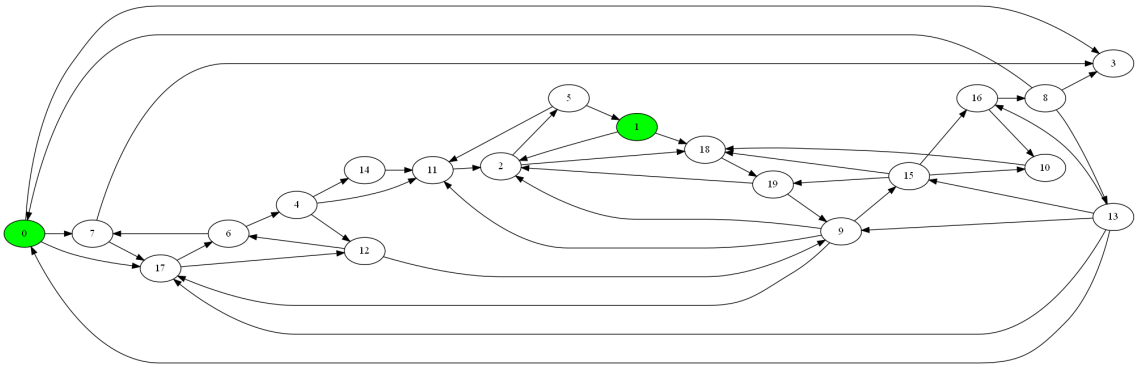

Abbildung 1: Ausgangsgraph 0

```
path 1:  ->   0  ->  17  ->  12  ->   9
path 2:  ->   1  ->  18  ->  19  ->   9

Lenght of paths: 3
Calculated Solution in 181 ms
```

Listing 1: Ausgabe Beispiel 0

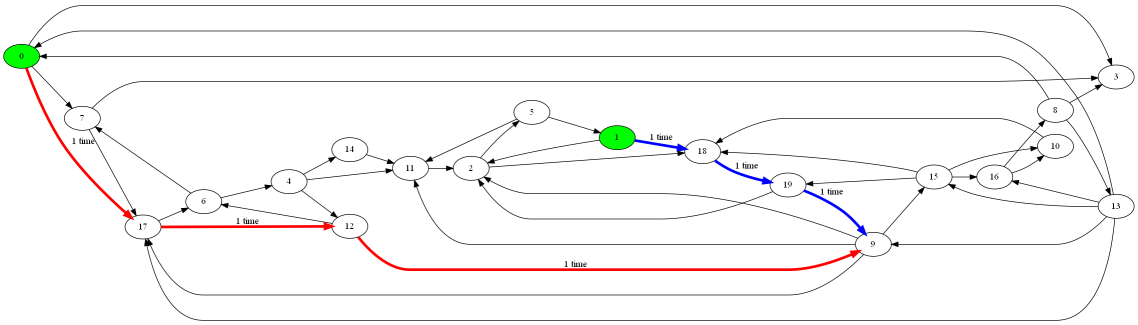

Abbildung 2: Endgraph 0

3.1.2 Beispiel 1

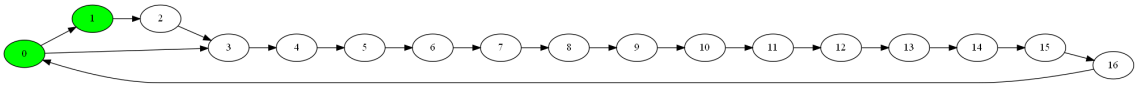

Abbildung 3: Ausgangsgraph 1

```
path 1:  ->  0  ->  3  ->  4  ->  5  ->  6  ->  7  ->  8  ->  9  ->  10  ->  11
    ↪ ->  12  ->  13  ->  14  ->  15  ->  16  ->  0  ->  3  ->  4  ->  5  ->  6  ->
    ↪  7  ->  8  ->  9  ->  10  ->  11  ->  12  ->  13  ->  14  ->  15  ->  16  ->
    ↪  0  ->  3  ->  4  ->  5  ->  6  ->  7  ->  8  ->  9  ->  10  ->  11  ->  12
    ↪  ->  13  ->  14  ->  15  ->  16  ->  0  ->  3  ->  4  ->  5  ->  6  ->  7
    ↪ ->  8  ->  9  ->  10  ->  11  ->  12  ->  13  ->  14  ->  15  ->  16  ->  0  ->
    ↪  3  ->  4  ->  5  ->  6  ->  7  ->  8  ->  9  ->  10  ->  11  ->  12  ->
    ↪ 13  ->  14  ->  15  ->  16  ->  0  ->  3  ->  4  ->  5  ->  6  ->  7  ->  8
    ↪  ->  9  ->  10  ->  11  ->  12  ->  13  ->  14  ->  15  ->  16  ->  0  ->  3
    ↪ ->  4  ->  5  ->  6  ->  7  ->  8  ->  9  ->  10  ->  11  ->  12  ->  13  ->
    ↪  14  ->  15  ->  16  ->  0  ->  3  ->  4  ->  5  ->  6  ->  7  ->  8  ->
    ↪  9  ->  10  ->  11  ->  12  ->  13  ->  14  ->  15  ->  16  ->  0  ->  3
path 2:  ->  1  ->  2  ->  3  ->  4  ->  5  ->  6  ->  7  ->  8  ->  9  ->  10
    ↪ ->  11  ->  12  ->  13  ->  14  ->  15  ->  16  ->  0  ->  1  ->  2  ->  3  ->
    ↪  4  ->  5  ->  6  ->  7  ->  8  ->  9  ->  10  ->  11  ->  12  ->  13  ->
    ↪ 14  ->  15  ->  16  ->  0  ->  1  ->  2  ->  3  ->  4  ->  5  ->  6  ->  7
    ↪  ->  8  ->  9  ->  10  ->  11  ->  12  ->  13  ->  14  ->  15  ->  16  ->  0
    ↪ ->  1  ->  2  ->  3  ->  4  ->  5  ->  6  ->  7  ->  8  ->  9  ->  10  ->
    ↪  11  ->  12  ->  13  ->  14  ->  15  ->  16  ->  0  ->  1  ->  2  ->  3  ->
    ↪  4  ->  5  ->  6  ->  7  ->  8  ->  9  ->  10  ->  11  ->  12  ->  13  ->  14
    ↪  ->  15  ->  16  ->  0  ->  1  ->  2  ->  3  ->  4  ->  5  ->  6  ->  7
    ↪ ->  8  ->  9  ->  10  ->  11  ->  12  ->  13  ->  14  ->  15  ->  16  ->  0  ->
    ↪  1  ->  2  ->  3  ->  4  ->  5  ->  6  ->  7  ->  8  ->  9  ->  10  ->
    ↪ 11  ->  12  ->  13  ->  14  ->  15  ->  16  ->  0  ->  1  ->  2  ->  3

Lenght of paths: 121
Calculated Solution in 212 ms
```

Listing 2: Ausgabe Beispiel 1

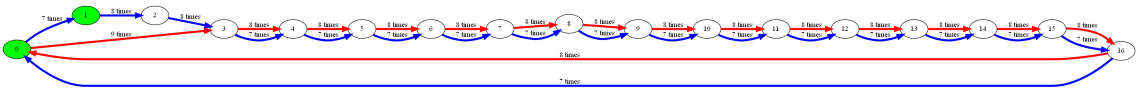

Abbildung 4: Endgraph 1

3.1.3 Beispiel 2

```
path 1:  ->  0  ->  50  ->  75  ->  58  ->  41  ->  64  ->  53  ->  91  ->  26
```

```
path 2:  ->  1  ->  23  ->  52  ->  1  -> 105  -> 135  -> 107  ->  99  ->  26

Lenght of paths: 8
Calculated Solution in 836 ms
```

Listing 3: Ausgabe Beispiel 2

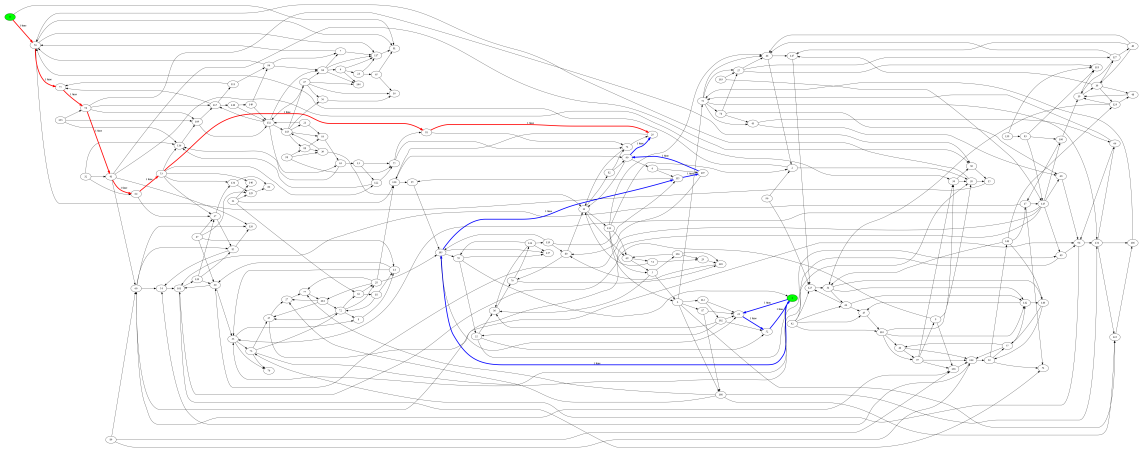

Abbildung 5: Endgraph 2

3.1.4 Beispiel 3

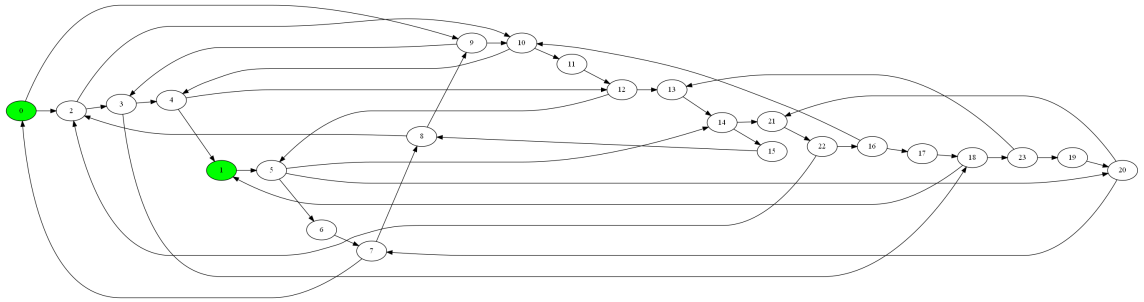

Abbildung 6: Ausgangsgraph 3

```

there is no solution!
Calculated Solution in 34 ms
```

Listing 4: Ausgabe Beispiel 3

3.1.5 Beispiel 4

```
path 1:  ->  0  ->  98  ->  88  ->  78  ->  77  ->  76  ->  75  ->  65  ->  55  ->  54
    ↪ ->  53  ->  43  ->  42  ->  32  ->  22  ->  12  ->  11
path 2:  ->  1  ->  11  ->  10  ->  99  ->  1  ->  11  ->  10  ->  99  ->  1  ->  11
    ↪ ->  10  ->  99  ->  1  ->  11  ->  10  ->  99  ->  11

Lenght of paths: 16
Calculated Solution in 153 ms
```

Listing 5: Ausgabe Beispiel 4

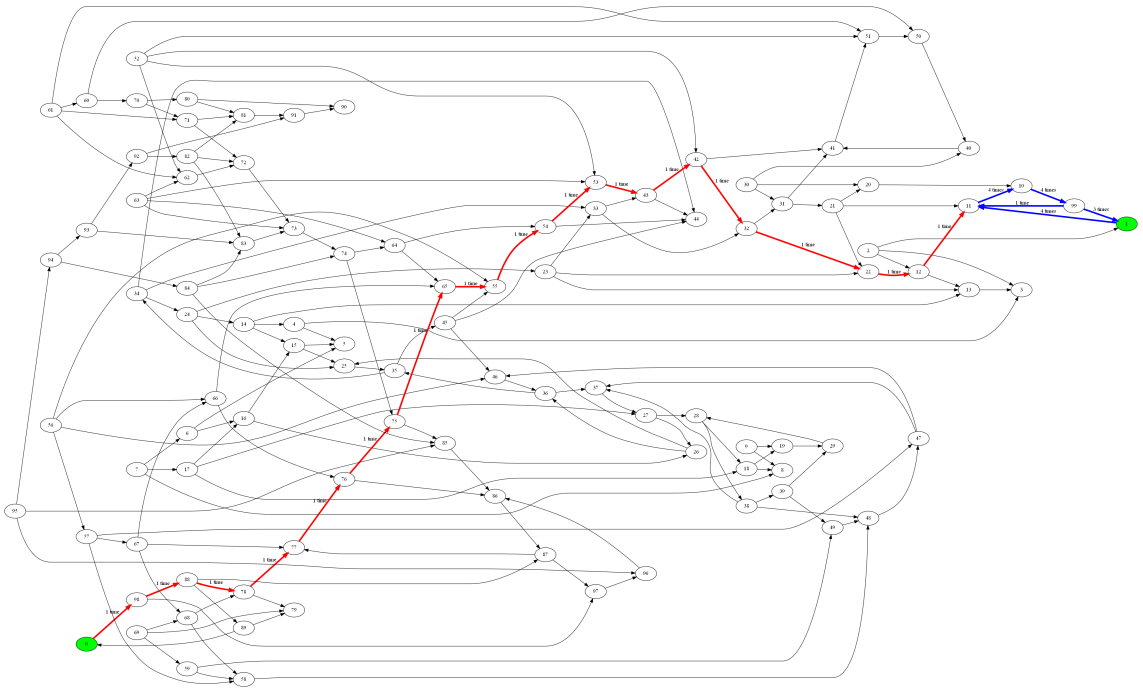

Abbildung 7: Endgraph 4

3.2 Eigene Beispiele

3.2.1 Beispiel 5

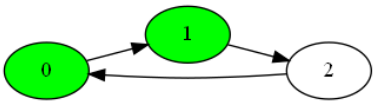

Abbildung 8: Ausgangsgraph 5

```
there is no solution!
Calculated Solution in 11 ms
```

Listing 6: Ausgabe Beispiel 5

4 Quellcode

```
#include <bits/stdc++.h>

#define ve vector
#define vi ve<int>
#define vvi ve<vi>
#define vvvi ve<vvi>
#define everyN(var, start) for(int var = start; var < n; var++)
#define everyM(var, start) for(int var = start; var < m; var++)

using namespace std;
using namespace chrono;
const int maxNodes = 151;
const int maxDist = 1000;
```

```
ifstream inputFile;
ofstream outputFile, outputFileGraph, outputFileGraphEnd;
vvi G(maxNodes);
int n, m;

/*
This function prints a path to cout and the ouput file
*/
void printPath(vi path, string name){
    cout << name << ": "; // print paht name
    outputFile << name << ": ";
    for(int x : path){ // print each node in path
        cout << " -> " << setw(3) << x << " ";
        outputFile << " -> " << setw(3) << x << " ";
    }
    cout << endl;
    outputFile << endl;
}

/*
This functions prints the result paths
*/
void printResults(vi path_1, vi path_2, int i){
    printPath(path_1, "path 1"); // print path 1
    printPath(path_2, "path 2"); // print path 2
    cout << endl << "Lenght of paths: " << i << endl; // print length of path to cout
    outputFile << endl << "Lenght of paths: " << i << endl; // print length of path to
    ↪ output file
}

/*
This function creates the Graph End file, which represents the graph with colored paths.
*/
void writeContentToGraphEndFile(vi path_1, vi path_2, int i){
    everyN(x,0){ // go through every edge
        for(int y : G[x]){
            int a = 0, b = 0;
            for(int k = 0; k < i; k++){ // check if edge has to be colored
                if(path_1[k] == x && path_1[k+1] == y){
                    a++;
                }
                if(path_2[k] == x && path_2[k+1] == y){
                    b++;
                }
            }
            if(a == 0 && b == 0){
                outputFileGraphEnd << x << "->" << y << endl; // write edge to output
            } else {
                if(a != 0){
                    outputFileGraphEnd << x << "->" << y << " [color = red; penwidth=4; 
    ↪ label=\" " << a << ((a == 1) ? " time\"]" : " times\"]") << endl;  // color edges
    ↪ on path 1 red
                }
                if(b != 0){
                    outputFileGraphEnd << x << "->" << y << " [color = blue; penwidth=4 
    ↪ label=\" " << b << ((b == 1) ? " time\"]" : " times\"]") << endl; // color edges
    ↪ on path 2 blue
                }
            }
        }
    }
    outputFileGraphEnd << "}" << endl; // print end of graph file with colored edges
}

/*
This function is used recursive and performs a Depht-First-Search to find all paths
*/
void dfs(int node, int dist, vi &path, vvvi &paths){
    if(paths[node][dist].size() == 0 && dist < maxDist){ // check if edge hasn't been
    ↪ visited with same distance and path isn't too long
        if(dist > 0){ // only write path if distance is greater than 0
            paths[node][dist] = path; // save path
            paths[node][dist].push_back(node); // push back end node
```

```
        }
        path.push_back(node); // push back next node to path
        for(int newNode : G[node]){
            dfs(newNode, dist+1, path, paths); // perform next dfs for all neighbors
        }
        path.pop_back(); // remove last element from path
    }
}

/*
This function tries to find two paths with the same length to the same destination,
    ↪ starting from node 0 and 1
*/

void getPath(vvvi &paths_1, vvvi &paths_2){
    for(int i = 1; i < maxDist; i++){  // check distances from 0 to max distance
        everyN(j,2){ // check for each node
            if(paths_1[j][i].size() != 0 && paths_2[j][i].size() != 0){ // check if there
    ↪  are 2 paths with same lengths
                printResults(paths_1[j][i], paths_2[j][i], i); // print out results
                writeContentToGraphEndFile(paths_1[j][i], paths_2[j][i], i); // write to
    ↪ Graph file with colored edges
                return;
            }
        }
    }
    // there is no path
    everyN(x,0){ // write original graph to Graph file with colored edges
        for(int y : G[x]){
            outputFileGraphEnd << x << "->" << y << endl;
        }
    }
    outputFileGraphEnd << "}" << endl;
    cout << endl << "there is no solution!" << endl; // write to output
    outputFile << endl << "there is no solution!" << endl; // write to output file
}

/*
This function is called to solve the problem
*/
void solve(){
    vvvi paths1(n, vvi(maxDist)); // create vector for path saving
    vvvi paths2(n, vvi(maxDist));
    vi path1; // create vector for path
    vi path2;
    dfs(0, 0, path1, paths1); // start dfs from start point 1
    dfs(1, 0, path2, paths2); // start dfs from start point 2
    getPath(paths1, paths2); // calculate path from results of dfs
}

/*
This function gets the filename of the input File
*/
string getFilename(){
    string filename;
    cout << "Please enter filename without file extension" << endl; // print message to
    ↪ user
    cout << "Files must be located in '/beispieleingaben/'" << endl;
    cout << "-> ";
    cin >> filename; // get input from user
    return filename;
}

/*
    This function is the main Function and is called at the execution start
    It reads the input file and starts the solve() methode
*/
int main(){
    string filename = getFilename(); // get filename for in-/output
    inputFile.open("../beispieleingaben/huepfburg"+filename+".txt"); // open input file
    if(inputFile.is_open()){
        outputFile.open("../beispielausgaben/huepfburg"+filename+".out"); // open output
    ↪ file
```

```
        outputFileGraph.open("../beispielausgaben/huepfburg"+filename+".dot"); // open
    ↪ output file for graph
151     outputFileGraphEnd.open("../beispielausgaben/huepfburg"+filename+"_end.dot"); //
    ↪ open output file for graph with colored paths
        outputFileGraph << "digraph{" << endl << "rankdir=LR" << endl; // write head into
    ↪  graph file
153     outputFileGraphEnd << "digraph{" << endl << "rankdir=LR" << endl; // write head
    ↪ into graph file with colored paths
        inputFile >> n >> m; // get number of edges and nodes
155     for(int i = 0; i < n; i++){ // write nodes to graph files
            outputFileGraph << i;
157         outputFileGraphEnd << i;
            if(i == 0 || i == 1){ // set color of start nodes to green
159             outputFileGraph << " [fillcolor = green; style=filled]";
                outputFileGraphEnd << " [fillcolor = green; style=filled]";
161         }
            outputFileGraph << endl;
163         outputFileGraphEnd << endl;
        }
165     everyM(i,0){ // read edges from input file
            int n1, n2;
167         inputFile >> n1 >> n2;
            n1--; n2--;
169         G[n1].push_back(n2); // write to adjacency list
            outputFileGraph << n1 << "->" << n2 << endl; // write edges to graph file
171     }
        outputFileGraph << "}" << endl; // print end of graph file
173     inputFile.close(); // close input file
    } else {
175     cout << "Unable to open file";
    }
177
    auto t1 = high_resolution_clock::now();
179 solve(); // call function to solve problem
    auto t2 = high_resolution_clock::now();
181
    auto ms_int = duration_cast<milliseconds>(t2 - t1);
183 cout << "Calculated Solution in " << ms_int.count() << " ms" << endl;
    outputFile << "Calculated Solution in " << ms_int.count() << " ms" << endl;
185 system(("dot -Tpng -o ../beispielausgaben/huepfburg"+filename+".png ../
    ↪ beispielausgaben/huepfburg"+filename+".dot").c_str()); // perform dot command for
    ↪ graph file
    system(("dot -Tpng -o ../beispielausgaben/huepfburg"+filename+"_end.png ../
    ↪ beispielausgaben/huepfburg"+filename+"_end.dot").c_str()); // perform dot command
    ↪ for graph file with colored edges
187 outputFile.close();
    outputFileGraph.close();
189 outputFileGraphEnd.close();
    return 0;
191 }
```