Fabian Dasler

-

41. Bundeswettbewerb für Informatik 2022

Name: Fabian Dasler

Klasse: ITM7

Schule Robert-Bosch-Berufskolleg

Inhaltsverzeichnis

1. Lösungsidee

Jede Person startet auf einer Seite des Feldes, x Felder voneinander entfernt. Da die Felder mehrfach untereinander verknüpft sind, bietet sich hier eine Breiten- oder Tiefensuche an. D.h. beide Personen gehen Schritt für Schritt, entweder von links nach rechts, oder immer eine Ebene Tiefer die Felder durch um zu gucken wo sich die beiden Personen treffen.

Da C# die Programmiersprache meiner Wahl ist und diese sehr gut geeignet für Multi-Threading ist, habe ich mich für eine asynchrone/ parallele Breitensuche entschieden.

Es starten beide Personen von einem Feld. In der ersten Iteration stehen die Personen auf deren Startfelder. Von dort wird geguckt welche Felder von dem Startfeld aus erreichbar sind. Dann wird von dem aktuellen Feld aus jeweils auf das nächste gegangen. Dies passiert bei allen Feldern gleichzeitig, da dies asynchron/ parallel passiert. Von diesen Feldern aus wird wieder für jedes Feld geguckt welche von dort aus betreten werden können und es wird wieder ein Schritt gemacht. Dies soll so lange passieren, bis sich beide Personen auf einem Feld treffen. Danach sollen Informationen über das erreichte Ziel angegeben werden.

2. Aufbau des Programmes

Das Projekt enthält drei C#-Klassen. Diese unterteilen sich in die „Program“-, die „Classes“- und die „Utility“-Klasse. Dabei ist in der Program-Klasse die Logik des Programms hinterlegt. Die Classes Datei enthält alle Definitionen von Klassen, die ich für das Programm brauche. Die Utility-Klasse enthält weitere Funktionen, die relevant für das Programm sind, aber nicht in eine der anderen Beiden Dateien, bzw. Klassen rein passt.

2.1. Aufbau der Classes-Datei

In der Datei werden die Klassen Definiert, die Ich für das Programm benötige. Dort enthalten sind die „Person“ Klasse, die „Field“ Klasse und die „Path“ Klasse.

2.1.1 Die Person Klasse

Diese Klasse definiert eine Person. Sie soll die Zuordnung von gegangenen Pfaden und der Person vereinfachen, genauso wie indizieren ob die Person auf ein Feld getroffen ist, auf dem bereits eine andere Person steht. Ebenso Kann das Startfeld mit angegeben werden.

```
/// <summary>
/// Definiert eine Person
/// </summary>
public class Person
{
    public string Name { get; init; }

    public Field StartField { get; init; }

    public List<Path> Paths { get; set; }

    public bool FinishRun { get; set; }
}
```

2.1.2 Die Field Klasse

Diese Klasse definiert ein Feld. Dieses Feld hat einen numerischen Wert und ein Array mit allen Feldern, die von diesem aus erreichbar sind. Ebenso hat ein Feld auch eine Property „PersonOnField“. Diese Property gibt an, ob eine Person bereits auf dem Feld steht. Die Property

```
/// <summary>
/// Definiert ein Feld
/// </summary>
public class Field
{
    /// <summary>
    /// Der nummerische Wert des <see cref="Field"/>
    /// </summary>
    public int Value { get; init; }

    /// <summary>
    /// Ein Array mit den nächsten, von diesem <see cref="Field"/> aus, betretbaren Feldern
    /// </summary>
    public Field[] NextPossibleFields { get; private set; } = Array.Empty<Field>();

    /// <summary>
    /// Die Person, die das Feld betreten hat
    /// </summary>
    public Person? PersonOnField { get; set; }

    /// <summary>
    /// Die zweite Person, die das Feld betreten hat
    /// </summary>
    public Person? SecondPerson { get; set; }

    /// <summary>
    /// Initialisiert ein Objekt der <see cref="Field"/> Klasse
    /// </summary>
    /// <param name="value">Der nummerische Wert des Feldes</param>
    public Field(int value)
    {
        Value = value;
    }
```

„SecondPerson“ ist im Normalfall null, aber soweit „PersonOnField“ nicht null ist, wird SecondPerson gleich der aktuellen Person gesetzt. Dies vereinfacht später die Suche des Pfades, in der „Paths“ Property der Person, bei dem das Feld ist, auf dem sich die beiden Personen treffen.

Die Klasse hat Hilfsmethoden um aus einem Dictionary<int, int[]> die passenden Field Objekte zu erzeugen.

```
/// <summary>
/// Erstellt aus einem Dictionary eine Liste mit <see cref="Field"/> Objekte
/// </summary>
/// <param name="fieldDict">Das Dictionary aus dem die Liste erstellt wird</param>
public static void CreateFields(Dictionary<int, int[]> fieldDict)
{
    CreateList(fieldDict);
    CreateNextValues(fieldDict);
}

/// <summary>
/// Erstellt die Liste mit <see cref="Field"/>s und fügt jedem sein nummerisches Value hinzu
/// </summary>
/// <param name="fieldDict">Das Dictionary aus dem die Liste erstellt wird</param>
private static void CreateList(Dictionary<int, int[]> fieldDict)
{
    List<Field> f = new();
    foreach (var field in fieldDict)
        f.Add(new Field(field.Key));
    Utility.FieldsList = f;
}

/// <summary>
/// Füllt die Liste mit <see cref="Field"/>s mit den nächsten möglichen <see cref="Field"/>s
/// </summary>
/// <param name="fieldDict">Das Dictionary aus dem die Liste erstellt wird</param>
private static void CreateNextValues(Dictionary<int, int[]> fieldDict)
{
    foreach (var field in Utility.FieldsList)
    {
        var values = fieldDict[field.Value];
        var nextFields = new List<Field>();

        foreach (var val in values)
        {
            var nf = Utility.FieldsList.Find(f => f.Value == val);

            if (nf is not null)
                nextFields.Add(nf);
        }

        field.NextPossibleFields = nextFields.ToArray();
    }
}
```

Dabei wird in der „CreateList“ Methode für jedes Key-Value-Pair im Dictionary ein neues Field erzeugt und die „Value“ Property gleich dem Key des Key-Value-Pair gesetzt.

In der der „CreateNextValues“ Methode wird für jedes Feld das Array mit den nächsten möglichen Feldern (Value des Key-Value-Pair) in „values geladen“. Dies sind dabei aber nur die Integer. Im inneren For-Each-Loop wird für jedes Value in der Liste mit Feldern, nach dem richtigen Feld gesucht. Soweit dieses nicht null ist wird dies dann der temporären Liste mit Feldern hinzugefügt. Diese wird zum Schluss dann gleich der nächsten möglichen Felder des aktuellen Feldes gesetzt.

2.1.3 Die Path Klasse

Diese Klasse hat nur ein Array mit Feldern, die den Pfad repräsentieren.

Ebenso enthält die Klasse eine Property zum einfachen bestimmen der Länge des Pfades.

Die „Path“ Klasse hat ebenso zwei Methoden. Eine um das hinzufügen eines Feldes zu dem Field-Array zu vereinfachen. Die zweite Methode ist eine Methode um eine Deep-Copy des Pfades zu erstellen, damit nicht mehrere Referenzen des Types Pfad auf die gleiche Instanz zeigen, sondern jede Referenz, die genutzt wird, eine eigene Objekt-Instanz hat.

```csharp
/// <summary>
/// Definiert einen Pfad
/// </summary>
public class Path
{
    /// <summary>
    /// Eine Liste mit <see cref="Field"/> die den Pfad darstellen
    /// </summary>
    public Field[] Value { get; private set; } = Array.Empty<Field>();

    /// <summary>
    /// Gibt die Anzahl an Feldern des Pfades an
    /// </summary>
    public int Count => Value.Length;
```

```csharp
    /// <summary>
    /// Fügt ein <see cref="Field"/> dem <see cref="Path"/> hinzu
    /// </summary>
    /// <param name="field"></param>
    public void AddField(Field field)
    {
        var newVal = new Field[Value.Length + 1];

        for (int i = 0; i < Value.Length; i++)
            newVal[i] = Value[i];

        newVal[Value.Length] = field;

        Value = newVal;
    }

    /// <summary>
    /// Erstellt eine Schattenkopie des <see cref="Path"/>
    /// </summary>
    /// <returns>Die Kopie des <see cref="Path"/></returns>
    public Path Clone()
    {
        return new Path()
        {
            Value = (Field[])this.Value.Clone()
        };
    }
}
```

2.2 Aufbau der Utility-Klasse

Dies ist eine statische Klasse, die Hilfsmethoden bereitstellt. Dort ist unter anderem das Dictionary mit dem Werten für die erste Aufgabe von Aufgabe 5 hinterlegt.

Außerdem ist dort ist die Liste mit Feldern hinterlegt, die in der „Field“ Klasse erzeugt wird.

```csharp
/// <summary>
/// Key: Value des Feldes, Value: Array mit den values von den zu erreichenden Feldern
/// </summary>
public static Dictionary<int, int[]> FieldsDict { get; set; } = new()
{
    {1, new[] { 8, 4, 18 } },
    {2, new[] { 3, 19 } },
    {3, new[] { 6, 19 } },
    {4, Array.Empty<int>() },
    {5, new[] { 12, 13, 15 } },
    {6, new[] { 2, 12 } },
    {7, new[] { 5, 8 } },
    {8, new[] { 4, 18 } },
    {9, new[] { 1, 4, 14 } },
    {10, new[] { 3, 12, 16, 18 } },
    {11, new[] { 19 } },
    {12, new[] { 3 } },
    {13, new[] { 7, 10 } },
    {14, new[] { 1, 10, 16, 17, 18 } },
    {15, new[] { 12 } },
    {16, new[] { 11, 17, 19, 20 } },
    {17, new[] { 19 } },
    {18, new[] { 7, 13 } },
    {19, new[] { 20 } },
    {20, new[] { 3, 10 } },
};
```

Neben diesen beiden statischen Properties umfässt die Klasse noch zwei Methoden.

```csharp
/// <summary>
/// Eine Liste mit den fertigen <see cref="Field"/> Objekten
/// </summary>
public static List<Field> FieldsList { get; set; } = new();
```

Da das Programm asynchron, bzw. im Multi-Thread läuft, bietet die „AddPath“ Methode das Thread-Safe hinzufügen eines Pfades zu der List mit Pfaden einer Person.

```csharp
/// <summary>
/// Eine threadsichere Methode um einer <see cref="Person"/> einen <see cref="Path"/> hinzuzufügen
/// </summary>
/// <param name="person">Die Person der der <see cref="Path"/> hinzugefügt werden soll</param>
/// <param name="path">Der <see cref="Path"/> der hinzugefügt werden soll</param>
public static void AddPath(this Person person, Path path)
{
    bool cantEnter;
    do
    {
        if (cantEnter = Monitor.TryEnter(person.Paths))
        {
            try
            {
                person.Paths.Add(path);
            }
            finally
            {
                Monitor.Exit(person.Paths);
                cantEnter = false;
            }
        }
    } while (cantEnter);
}
```

Die andere Methode „ReadTextfile“ bietet die Funktionalität, um die anderen Hüpfburgen aus den Textdateien auszulesen und gibt diese dann als Dictionary zurück, um daraus dann wieder Felder erzeugen lassen zu können.

```
public static Dictionary<int, int[]> ReadTextfile(string path)
{
    var tempFiledDict = new Dictionary<int, int[]>();

    using StreamReader sr = new StreamReader(File.Open(path, FileMode.Open));

    while (!sr.EndOfStream)
    {
        int key = 0;
        int value = 0;

        try
        {
            var line = (sr.ReadLine() ?? "").Split(" ");
            if (line.Length == 0)
                continue;

            key = int.Parse(line[0]);
            value = int.Parse(line[1]);

        }catch(IOException ex)
        {
            Console.WriteLine("Fehler beim einlesen der Textdatei:");
            Console.WriteLine(ex.Message);
            Console.WriteLine(ex.StackTrace);
        }

        if (!tempFiledDict.ContainsKey(key))
            tempFiledDict.Add(key, Array.Empty<int>());

        tempFiledDict[key] = tempFiledDict[key].Append(value).ToArray();
    }

    foreach(var kvp in tempFiledDict)
    {
        if (kvp.Value is null)
            tempFiledDict[kvp.Key] = Array.Empty<int>();
    }

    return tempFiledDict;
}
```

2.3 Aufbau der Programm-Klasse

Diese Klasse enthält die Logik des Programmes, um die gegebene Aufgabe zu lösen. Dabei wird diese in sechs Funktionen unterteilt.

2.3.1 Die Main Methode

```
var p1 = new Person()
{
    Name = "Sasha",
    StartField = Utility.FieldsList
        .Where(f => Equals(f.Value, 2))
        .SingleOrDefault()!,
    Paths = new()
};
```

In dieser Methode startet das Programm und macht eine Abfrage ob die Standardfelder genutzt werden sollen, oder ob die Hüpfburgfelder aus einer Textdatei geladen werden soll. Danach wird ggf. die Textdatei geladen und anschließend werden sowohl die Felder als auch die beiden Personen erzeugt. Den Personen wird ein Startfeld aus der Liste mit Feldern zugeordnet.

2.3.2 Die Start Methode

INFO: Alle Angaben werden im Singular für eine Person formuliert, sind aber parallel auf die andere Person zu projizieren.

```
// Variablen für Person 1
bool p1Hit = false;
var path1 = new Path();

path1.AddField(person1.StartField);
person1.AddPath(path1);
```

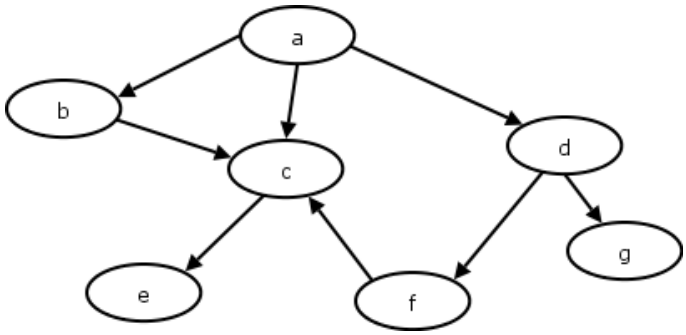
Als erstes wird ein bool erstellt, der die Überprüfung auf Person.FinishRun vereinfacht.

Danach wird ein Pfad erzeugt. Diesem wird das

Startfeld der Person hinzugefügt und der Pfad wird der Person hinzugefügt.

Als nächstes wird das Startfeld einer Liste hinzugefügt, die alle Felder enthält von denen sich im nächsten Schritt weiter bewegt werden soll. Dies wird für spätere Iterationen wichtig.

Im asynchronen Teil der Funktion wird die „ForEachField“ Methode aufgerufen, diese führt Schritte für jedes Feld in der Liste „fieldsToMoveFrom“ aus. Diese Methode gibt die Felder zurück, auf die die Person gerade steht und von denen sich aus in der nächsten Iteration weiter bewegt werden soll. Diese Felder werden dann, nach dem leeren der Liste, der Liste hinzugefügt, von der aus die Felder genommen werden um weitere Schritte zu gehen.



(Am Anfang ist nur a in der List. Die Person bewegt sich parallel auf b, c und d weiter vor. Die „ForEachField“ Methode gibt diese Felder zurück.)

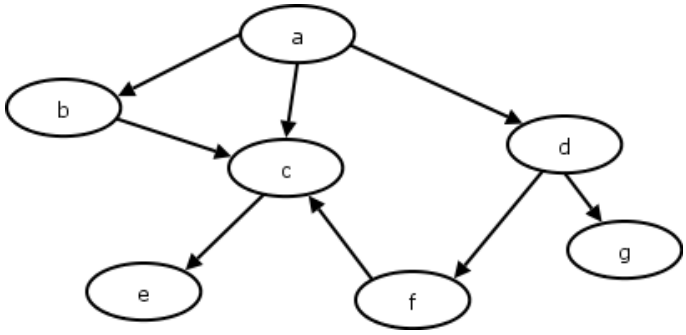
```
// Geht für jedes Feld auf die nächsten möglichen Felder
var nextFields = await ForEachField(person1, fieldsToMoveFrom1);

// setzt ob die Person sich auf einem Feld getroffen haben
p1Hit = person1.FinishRun;
if (p1Hit)
{
    // gibt die benötigten Infos aus
    var destField = Finish(person1);
    // gibt Infos über die andere Person aus
    ReviewOtherPerson(person2, destField);
    // beendet das Programm
    Environment.Exit(0);
}
```

Als nächstes wird geprüft, ob die Person ein Feld mit einer anderen Person getroffen hat. Wenn nein, geht das Programm normal weiter, ansonsten wird die „Finish“ und die „ReviewOtherPerson“ Funktionen aufgerufen um informationen über den Verlauf ausgegeben zu bekommen.

Solange nicht eine Person seinen lauf beendet hat läuft das Programm in der Wile-Schleife weiter.

2.3.3 Die ForEachField() Methode

Diese Methode nimmt eine Person und eine Liste mit Feldern als Parameter an. (Im ersten Durchlauf ist nur a in der Liste, im zweiten Durchlauf ist b, c und d enthalten, usw.)



Danach wird für jedes Feld in dieser Liste der längste Pfad der Person, welcher mit dem aktuellen Feld endet rausgesucht.

```
var path = person.Paths
    .OrderByDescending(pp => pp.Count)
    .Where(pp => pp.Value[pp.Count - 1] == field)
    .FirstOrDefault();
```

Für diese Feld wird eine weitere asynchrone Funktion gestartet, in der MoveNextField() aufgerufen wird. In MoveNextField() wird die Person, das aktuelle Feld und der zuvor ermittelten Pfades übergeben. Die von MoveNextField() zurückgegebene Liste mit Feldern wird zwischengespeichert und von der Methode ebenfalls zurück gegeben.

```
// geht für das ausgewählte Feld einen schritt weiter
var newFields = await MoveNextField(person, field, path!.Clone());

// setzt die nächsten felder auf die Liste
foreach (var nf in newFields)
    fields.Add(nf);
```

2.3.4 Die MoveNextField()

Diese Methode nimmt eine Person, ein Feld und den Pfad des Feld, der in der vorherigen Methode ermittelt wurde entgegen.

Für jedes, von dem aktuellen Feld, erreichbare Feld (Field.NextPossibleFields Property) wird eine weitere asynchrone Funktion gestartet, die folgendes tut:

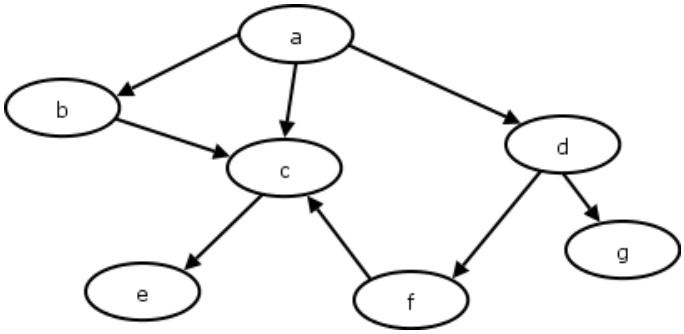
Es wird geprüft ob auf dem neuen Feld bereits die Person selbst drauf steht, um direkte zyklische Schritte zu vermeiden.

```
if (nextfield.PersonOnField is not null)
{
    if(Equals(nextfield.PersonOnField, person))
        return;
}
```

Im nächsten Schritt wird von dem Pfad eine Deep-Copy erstellt, um für jedes neue Feld eine eigene Objektinstanz des Pfades mit den schon beinhaltenden Feldern als auch mit dem neuen Feld zu haben.

Zu der Deep-Copy des Pfades wird nun das aktuelle Feld hinzugefügt und der neue Pfad wird an die Person angehangen.

Somit besteht nach dem zweiten Durchlauf des Programmes für eine Person folgende 4 Pfade (Anhand der Beispielabbildung rechts):



Feld a (1. Iteration)

Feld a - Feld b (2. Iteration)

Feld a - Feld c (2. Iteration)

Feld a - Feld d (2. Iteration)

```
// kopiert den Pfad für das neue Feld
var nextPath = path.Clone();
//fügt das nächste Feld dem Pfad hinzu
nextPath.AddField(nextfield);
// fügt der Person dem Pfad hinzu
person.AddPath(nextPath);
```

Darauf hin wird das aktuelle, neue Feld der temporären Liste mit Pfaden hinzugefügt.

Als nächstes wird geprüft, ob auf dem aktuellen Feld die andere Person steht. Wenn nicht, wird die Person auf das Feld gesetzt und das Programm geht normal weiter. Wenn die andere Person bereits auf dem Feld steht, dann wird das Field.SecondPerson von der Person eingenommen und Person.FinishRun auf true gesetzt. Dies indiziert, das die Personen sich getroffen haben.

```
// prüft ob auf dem nächsten Feld eine Person schon drauf ist und es nicht sie selbst ist
if (nextfield.PersonOnField is not null && !Equals(nextfield.PersonOnField, person))
{
    // setzt sich als zweites auf das Feld
    nextfield.SecondPerson = person;
    // definiert, das sich die Personen getroffen habe - Ziel erreicht
    person.FinishRun = true;
    return;
}
else // ansonsten setzt sich die Person auf das Feld
    nextfield.PersonOnField = person;
```

Zum Schluss wird die List mit den neuen Feldern (b, c und d aus dem obigen Beispiel) von der Methode zurück gegeben.

2.3.5 Die Finish und die ReviewOther Methode

In diesen Methoden werden Informationen bezüglich den Personen und den erfolgreich absolvierten Pfaden ausgegeben.

In der „Finish“ Methode wird die Person übergeben, die den Lauf beendet hat. Von dieser Person wird aus den Pfaden der Pfad raus gesucht, bei dem beim letzte Feld die „SecondPerson“ Property nicht null ist. Danach wird die Ausgabe erzeugt und in der Konsole ausgegeben. Danach gibt die Methode das Feld

zurück, bei dem sich beide Personen getroffen haben.

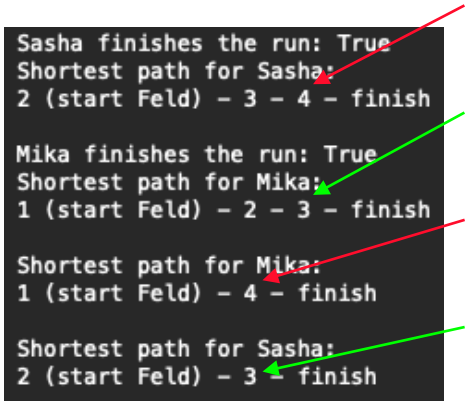
```
var shortestPath = person.Paths
    .Where(p => p.Value[^1].SecondPerson is not null)
    .OrderBy(fp => fp.Count)
    .FirstOrDefault();

string pathAsString = "";

foreach (var field in shortestPath!.Value)
{
    if (Equals(field, person.StartField))
        pathAsString += field.Value.ToString() + " (start Feld) - ";
    else
        pathAsString += field.Value.ToString() + " - ";
}
Console.WriteLine($"{person.Name} finishes the run: {person.FinishRun}");
Console.WriteLine($"Shortest path for {person.Name}:");
Console.WriteLine(pathAsString + "finish");
Console.WriteLine("");

return shortestPath.Value[shortestPath.Count - 1];
```

Die „ReviewOther“ Methode wird in der „Start“ Methode nach der „Finish“ Methode aufgerufen und nimmt das Feld, welches „Finish“ zurückgegeben hat und die andere Person als Parameter entgegen. Damit wird dann bei der anderen Person der Pfad gesucht, bei dem das Feld das letzte Element ist. Danach wird parallel zur „Finish“ Methode die Ausgabe erzeugt und in der Konsole ausgegeben.

3. Die Ergebnisse

Die Bilder der Ausgaben der Ergebnisse sind auf der nächsten Seite zu finden. Dabei sieht mann immer oben über der Ausgabe des Pfades, wer den Lauf beendet hat (Also wer auf ein Feld getroffen ist, wo bereits eine Person drauf Stand). Danach wird Der Pfad mit den Feldern ausgegeben. Wobei das erste Feld immer das Start Feld ist, wo die Person jeweils drauf stand und das letzte Feld vor dem „finish“ immer das Feld ist, auf dem sich die beiden Personen getroffen haben.

Da das Programm für beide Personen parallel läuft, sieht man bei Hüpfburg 1 - 3 das beide Personen den lauf beendet haben. Dabei sind die beiden ersten ausgaben immer die von der Person, welche den Lauf beendet hat und die unteren beiden die Pfade der anderen Personen. Wenn man auf die Endfelder und die Anzahl der Felder achtet, sieht man, das beides gültige Läufe und eine richtige Lösung sind, da beide Läufe jeweils gleich lang sind und beide auf einem gemeinsamen Feld landen.



```
Sasha finishes the run: True
Shortest path for Sasha:
2 (start Feld) - 19 - 20 - 10 - finish

Shortest path for Mika:
1 (start Feld) - 18 - 13 - 10 - finish
```

Hüpfburg 0

```
Sasha finishes the run: True
Shortest path for Sasha:
2 (start Feld) - 3 - 4 - finish

Mika finishes the run: True
Shortest path for Mika:
1 (start Feld) - 2 - 3 - finish

Shortest path for Mika:
1 (start Feld) - 4 - finish

Shortest path for Sasha:
2 (start Feld) - 3 - finish
```

Hüpfburg 1

```
Mika finishes the run: True
Shortest path for Mika:
1 (start Feld) - 51 - 76 - 59 - 42 - 65 - 54 - 92 - 72 - finish

Sasha finishes the run: True
Shortest path for Sasha:
2 (start Feld) - 106 - 136 - 108 - 81 - 12 - 114 - 3 - 21 - finish

Shortest path for Sasha:
2 (start Feld) - 106 - 136 - 108 - 81 - 12 - 83 - 72 - finish

Shortest path for Mika:
1 (start Feld) - 51 - 76 - 59 - 42 - 35 - 21 - finish
```

Hüpfburg 2

```
Sasha finishes the run: True
Shortest path for Sasha:
2 (start Feld) - 6 - 15 - 16 - 9 - 3 - finish

Mika finishes the run: True
Shortest path for Mika:
1 (start Feld) - 3 - 4 - 19 - 2 - 6 - finish

Shortest path for Mika:
1 (start Feld) - 3 - finish

Shortest path for Sasha:
2 (start Feld) - 6 - finish
```

Hüpfburg 3

```
Mika finishes the run: True
Shortest path for Mika:
1 (start Feld) - 99 - 89 - 79 - 78 - 77 - 76 - 66 - 56 - 55 - 54 - 44 - 43 - 33 - 23 - 13 - 12 - finish

Shortest path for Sasha:
2 (start Feld) - 12 - finish
```

Hüpfburg 4