Aufgabe 5: Hüpfburg

Team-ID: 00464

Team: Enrique Lopez

Bearbeiter/-innen dieser Aufgabe:
Enrique Lopez

5. November 2022

Inhaltsverzeichnis

Lösungsidee

Bei dem Spielfeld handelt es sich um einen directed graph. Sasha und Mika müssen nicht nur das selbe Feld erreichen, sondern dies auch gleichzeitig tun.

Zur Lösung der Aufgabe kann eine Art path-finding Algorithmus verwendet werden. Dabei wird jedem vom Feld 1 erreichbaren Feld der Wert "1" zugeordnet, und jedem von einem Feld mit Wert 1 erreichbares Feld der Wert "2". Dabei können einem Feld auch mehrere Werte zugeordnet werden, zum Beispiel wenn Sasha ein Feld durch im Kreis hüpfen mehrmals erreichen kann. Dasselbe wird von Feld 2 aus gemacht, jedoch mit den Werten "-1", "-2" und so weiter. Ein Parkour ist dann erfolgreich absolvierbar, wenn nach Wertezuordnung zwei zugeordnete Werte auf einem Feld zusammenaddiert 0 ergeben (-i und i in einer Liste). Dann können Mika und Sasha das Feld gleichzeitig erreichen. Um die kleinstmögliche Anzahl von Sprüngen zu finden, wird i beginnend von 1 in jeder Liste gesucht.

Die Rekonstruktion des Weges funktioniert so: Jedes Feld, von welchem aus das Treff-Feld erreichbar ist, wird auf Wert i-1 bzw. i+1 geprüft (Werte 4 und -4: Werte 3 und -3 werden gesucht.) Wenn ein Wert gefunden wurde, so wird das Feld ausgegeben und es wird wiederum auf den Feldern, von welchen das aktuelle Feld erreichbar ist, nach dem neuen Wert-1 bzw. Wert+1 gesucht, je nachdem welcher Wert gefunden wurde. Das wird solange wiederholt, bis sowohl Feld 1 als auch Feld 2 gefunden wurden, dann werden diese Felder ausgegeben und das Programm beendet.

Nun wird noch ein Weg benötigt um festzustellen, ob der Parkour nicht zu absolvieren ist. Dies ist ziemlich schwer, da ein sehr grosser Parkour 100000000 Sprünge benötigen könnte, bis er

absolviert ist. Daher führen wir einen zweiten Parameter für unser Programm ein, bei dem der Benutzer immer festlegen muss für wie viele Sprünge das Programm den Algorithmus programmieren soll. Somit gibt das Programm bei fehlender Lösung immer nur aus, dass der Parkour für die vorgegebene Anzahl der Sprünge nicht lösbar ist. Da unser Programm über 100000 Sprünge in unter 2 Sekunden berechnen kann und es unmöglich ist, bei kleinen Graphen wie in den Beispieldateien erst nach 1001 Sprüngen eine Lösung zu finden ist der Parkour dann überhaupt nicht lösbar, wenn das Programm für 1000 Sprünge keine Lösung findet.

Umsetzung

Die Bibliothek networkx

Die Bibliothek networkx übernimmt für uns die Einlesung der Verbindungspfeile (edges) in ein Dictionary. Mithilfe von der Bibliothek matplotlib können wir uns eine visuelle Darstellung des Parkour dann spaßeshalber ausgeben lassen. Dies ist aber nicht die Hauptfunktion der Bibliothek in unserem Programm, denn sie liefert uns zwei wichtige Methoden, welche wir sonst durch for–loops über ein Dictionary selber hätten definieren müssen:

1. graph.succesors(feld) – Liefert uns ein Iterable mit allen Feldern, welches von dem übergebenen Feld erreichbar ist

2. graph.succesors(feld) – Liefert uns ein Iterable mit allen Feldern, von welchen das übergebene Feld erreichbar ist.

Diese Methoden sind wichtig für unseren Algorithmus und in der Bibliothek sehr schnell. Wir hätten die Methoden jedoch auch mit selbst definierten Funktionen, welche über das Dictionary mit den eingelesenen Pfeilen iteriert, ersetzen können. Bevor ihr mir also für die Nutzung dieser Bibliothek Punkte abzieht, schreibt mir lieber eine Mail. Dann schreibe ich das Programm so um, dass es auch ohne die Bibliothek funktioniert.

Das Programm

Das Programm besteht aus einer Hauptfunktion hüpfburg() welche als Argumente die Textdatei mit den Pfeilen und die Anzahl der zu berechnenden Sprünge nimmt. Es ist überraschend schnell und kann 100000 Sprünge in unter 2 Sekunden berechnen.

Zunächst wird ein Directed Graph Objekt initialisiert (networkx), in welches daraufhin die nodes und edges des Graphen hinzugefügt werden. Das Objekt erstellt die nodes aus den edges automatisch. Gleichzeitig wird mit einer List Comprehension eine Nested List mit n Listen erstellt, wobei n die Zahl der nodes ist. Daraufhin wird der Graph visuell dargestellt, was zwar nicht in der Aufgabestellung gefragt war, jedoch trägt es zum Verständnis der Lösung beim Benutzer bei.

add_values_positive/negative()

Es gibt jede Funktion, welche Teil des Algorithmus ist, zweimal: Eine für positive (vom Feld 1 aus) und eine für Negative Werte (Vom Feld 2 aus). Die Funktion add_values nimmt ein node Namen und eine Zahl n entgegen und fügt allen von diesem Feld aus erreichbaren Feldern (successors) in ihre Werteliste 1 bzw. –1 (positive / negative Version) hinzu, woraufhin diejenigen Felder, welche vom einem Feld mit dem Wert 1 in der Werteliste aus erreichbar sind, Die Zahl 2 / –2 in die Werteliste hinzugefügt bekommen und so weiter. Das wird n mal wiederholt, wobei das Argument n in der Hauptfunktion immer mit der Anzahl der zu berechnenden Sprünge gefüllt wird.

find_goal_node()

Die Funktionen add_values_positive/negative() werden jeweils für Feld 1 und 2 Mit der zu berechnenden Sprunganzahl als number ausgeführt. Nun da in den Wertelisten aller Felder Werte hinzugefügt wurden sucht eine Schleife nach einer Liste (Feld), in welcher eine Zahl sowohl mit positivem als auch negativem Vorzeichen vorkommt. Dabei startet die Schleife von 1 aus um garantiert die kürzeste Sprunganzahl zu finden. Wenn eine Liste gefunden wurde so wird das Feld der Liste und die Anzahl der benötigten Sprünge in einem Tupel ausgegeben. Diesem Tupel werden später zwei Variablen zugewiesen, wobei sich hier ein Check befindet: Kommt es beim unpacking des Tupels zu einem Error, so muss er leer sein, was bedeutet, dass es keine Lösung gibt. Dann wird eine Nachricht ausgegeben. Wenn die zu berechnenden Sprünge größer als 1000 waren, so gibt es in den Beispielen garantiert keine Lösung, was gegebenenfalls auch ausgegeben wird.

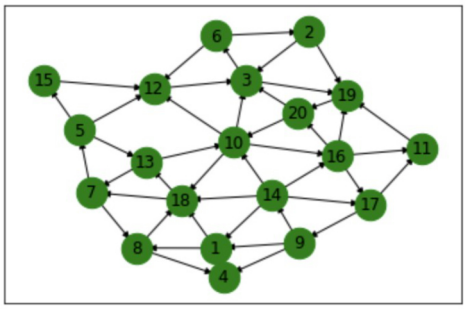
find_path_positive/negative()

Die beiden Funktionen nehmen in der Hauptfunktion als Argumente immer das Zielfeld und die benötigten Sprünge aus der Funktion find_goal_node(). Die Funktionen suchen in einem Loop jeweils in der Liste der predecessor des Zielfeldes nach dem Wert jumps –1/+1, woraufhin es in den predecessor dieses Feldes nach dem Wert jumps –2/+2 sucht. Dabei wird jedes Feld in welchem der Wert gefunden wurde in eine Liste path_sasha/mika an den Index 0 hinzugefügt. Nachdem das letzte Feld gefunden wurde wird in die Liste das Feld 1 / 2 hinzugefügt und die Werte in der Liste werden verbunden mit Pfeilen ausgegeben.

Beispiele

huepfburg0.txt

```
In [204]: hüpfburg("huepfburg0.txt",3)
```



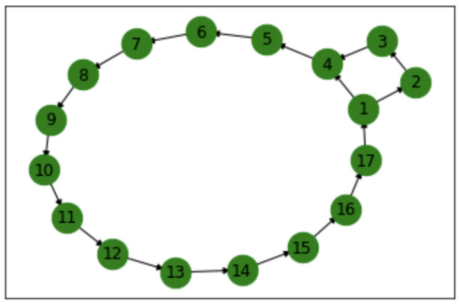

```
Erfolgreich absolviert in 3 Sprüngen!

Sashas Weg: 1->18->13->10

Mikas Weg: 2->19->20->10
```

huepfburg1.txt

```
In [206]: hüpfburg("huepfburg1.txt",200)
```



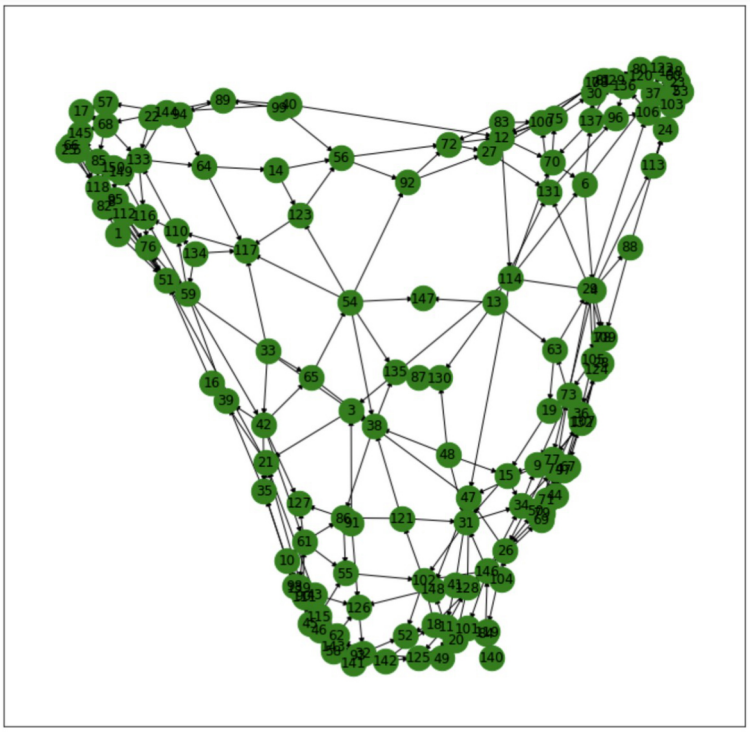

```
Erfolgreich absolviert in 121 Sprüngen!

Sashas Weg: 1->4->5->6->7->8->9->10->11->12->13->14->15->16->17->1->4->5->6->7->8->9->10->11-
>12->13->14->15->16->17->1->4->5->6->7->8->9->10->11->12->13->14->15->16->17->1->4->5->6->7->
8->9->10->11->12->13->14->15->16->17->1->4->5->6->7->8->9->10->11->12->13->14->15->16->17->1-
>4->5->6->7->8->9->10->11->12->13->14->15->16->17->1->4->5->6->7->8->9->10->11->12->13->14->1
5->16->17->1->4->5->6->7->8->9->10->11->12->13->14->15->16->17->1->4

Mikas Weg: 2->3->4->5->6->7->8->9->10->11->12->13->14->15->16->17->1->2->3->4->5->6->7->8->9-
>10->11->12->13->14->15->16->17->1->2->3->4->5->6->7->8->9->10->11->12->13->14->15->16->17->1
->2->3->4->5->6->7->8->9->10->11->12->13->14->15->16->17->1->2->3->4->5->6->7->8->9->10->11->
12->13->14->15->16->17->1->2->3->4->5->6->7->8->9->10->11->12->13->14->15->16->17->1->2->3->4
->5->6->7->8->9->10->11->12->13->14->15->16->17->1->2->3->4
```

Sasha bleibt immer im regulären Kreis, während Mika zusätzlich über die Felder 2 und 3 hüpft. Dadurch hat Sasha Mika irgendwann eingeholt.

huepfburg2.txt



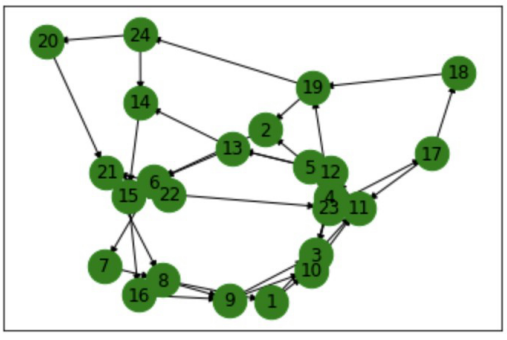
```
Erfolgreich absolviert in 8 Sprüngen!

Sashas Weg: 1->51->76->59->42->65->54->92->27

Mikas Weg: 2->106->136->108->100->27->131->100->27
```

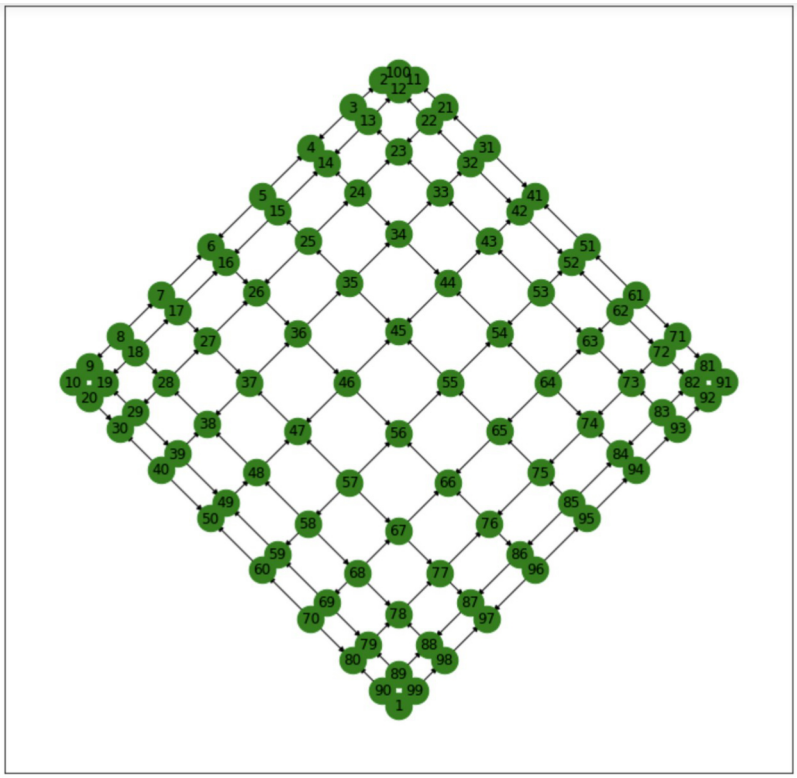
huepfburg3.txt

```
In [211]: hüpfburg("huepfburg3.txt",100000)
```



```
Nicht lösbar mit dieser Anzahl von Sprüngen
100000 Sprünge bei lediglich 24 Feldern und keine Lösung -> Der Parkour ist nicht lösbar
```

huepfburg4.txt



```
Erfolgreich absolviert in 20 Sprüngen!

Sashas Weg: 1->99->89->90->1->99->89->79->78->77->76->66->56->55->54->44->43->33->32->22->12

Mikas Weg: 2->12->11->100->2->12->11->100->2->12->11->100->12->11->100->2->12->11->100->2->12
```

Mika ist in dem oberen Kreisweg auf 4 Feldern gefangen. Sasha muss von unten zu ihm kommen

endless.txt

```
In [215]: hüpfburg("endless.txt",10000000)
```


```
Nicht lösbar mit dieser Anzahl von Sprüngen
10000000 Sprünge bei lediglich 1 Feldern und keine Lösung -> Der Parkour ist nicht lösbar
```

Eigenes Beispiel: 2 Felder die jeweils durch Pfeil verbunden sind. Sie springen aneinander vorbei.

Quellcode

November 8, 2022

[8]:
```
import networkx as nx
import matplotlib.pyplot as plt

def hüpfburg(datei,sprünge):

    #Directed Graph Objekt wird initialisiert, Pfeile und Felder werden␣
 ↪eingelesen

        G = nx.DiGraph()
        with open(datei) as f:
            node_values = [[] for i in range(int(f.readline().split()[0]))]
            line = f.readline()
            while line:
                edge = tuple(line.split())
                G.add_edges_from([edge])
                line = f.readline()

        #Dieser Code gibt den Parkour visuell aus
        pos = nx.kamada_kawai_layout(G)
        if len(node_values) > 30:
            plt.figure(3,figsize=(12,12))
        nx.draw_networkx_nodes(G,pos, node_size = 500,node_color = "green")
        nx.draw_networkx_edges(G,pos, edgelist = G.edges(), edge_color =␣
 ↪"black")
        nx.draw_networkx_labels(G,pos)
        plt.show()

        #Algorithmusfunktion: Wie viele Sprünge von welchem Feld
        #Positive Werte (1,2,3) werden Nested List hinzugefügt

        def add_values_positive(node,number):

                for i in list(G.successors(str(node))):
                    node_values[int(i)-1].append(1)

                for i in range(number):
                    for b,node in enumerate(node_values):
```

```
                    if node.count(1+i) == 1:
                        for n in G.successors(str(b+1)):
                            node_values[int(n)-1].append(2+i)


        #Die selbe Algorithmusfunktion, aber mit negativen Werten
        def add_values_negative(node,number):

                for i in list(G.successors(str(node))):
                    node_values[int(i)-1].append(-1)

                for i in range(number):
                    for b,node in enumerate(node_values):
                        if node.count(-1-i) == 1:
                            for n in G.successors(str(b+1)):
                                node_values[int(n)-1].append(-2-i)


        #Auf welchem Feld treffen sich Sasha und Mika und mit wie vielen␣
↪Sprüngen
        def find_goal_node(sprünge):
            add_values_positive(1,sprünge)
            add_values_negative(2,sprünge)
            for i in range(1,sprünge+1):
                for node in node_values:
                    if node.count(i) > 0 and node.count(-i) > 0:
                        goal_node = node_values.index(node)+1
                        jumps = i
                        return (goal_node,jumps)


        #Rekonstrunktion des Weges mit Anzahl Sprünge von Feld 1
        def find_path_positive(node,jumps):
            path_sasha = []
            path_sasha.append(str(node))
            current = node
            for n in range(jumps):
                for i in G.predecessors(str(current)):
                    if node_values[int(i)-1].count(jumps-1-n):
                        current = i
                        path_sasha.insert(0,str(current))
                        break
            path_sasha.insert(0,"1")
            return print("Sashas Weg:","->".join(path_sasha),"\n")

        #Rekonstrunktion des Weges mit Anzahl Sprünge von Feld 2
        def find_path_negative(node,jumps):
```

```
            path_mika = []
            path_mika.append(str(node))
            current = node
            for n in range(jumps):
                for i in G.predecessors(str(current)):
                    if node_values[int(i)-1].count(-jumps+1+n):
                        current = i
                        path_mika.insert(0,str(current))
                        break
            path_mika.insert(0,"2")
            return print("Mikas Weg:","->".join(path_mika))




        #Wenn mehr als 100000 Sprünge ohne Lösung berechnet wurden gibt es␣
↪keine (Siehe Dokumentation)
        try:
            goal_node,jumps = find_goal_node(sprünge)
        except:
            if sprünge >= 1000:
                return print('Nicht lösbar mit dieser Anzahl von Sprüngen \n{}␣
↪Sprünge bei lediglich {} Feldern und keine Lösung -> Der Parkour ist nicht␣
↪lösbar'.format(sprünge,len(node_values)))
            else:
                return print('Nicht lösbar mit dieser Anzahl von Sprüngen.␣
↪Versuch es mit mehr!')
        print('Erfolgreich absolviert in {} Sprüngen!\n'.format(jumps))
        find_path_positive(goal_node,jumps)
        find_path_negative(goal_node,jumps)
```