Aufgabe 5: Hüpfburg

Team-ID: 00811

Team: Lorian

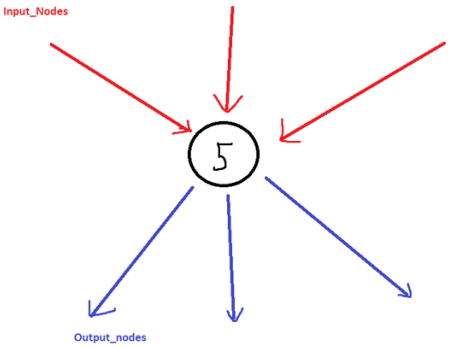
Bearbeiter/-innen dieser Aufgabe:
Lorian Linnertz

15. November 2022

Inhaltsverzeichnis

Lösungsidee

Meine Lösungsidee, besteht aus einer Vereinfachten Bruteforce Methode.



Felder

Die Springfelder, welche ich lieber mit dem Englischen Wort Nodes bezeichne, representiere ich im Code als Klasse. Die Node_Klasse besitzt 2 Hauptlisten, die Liste mit den Input_Nodes, das sind die Nodes welche zu dem Node führen und den Output_Nodes, das sind die Nodes zu welchen die aktuelle führt. Der Begriff Gewicht bedeutet soviel wie Anzahl an sprüngen um zu einem Feld zu kommen.

Ziel

Um die Aufgabe sauber Lösen zu können müssen wir als erstes das Ziel definieren. Das Ziel ist es ein Feld zufinden, aufwelchem Mika und Sascha gleichzeitig landen. Dazu müssen wir für die Beide, raus finden auf welcher Node, die beiden das gleiche Gewicht haben. Wir berechnen somit zweimal alle möglichen Sprünge, jeweils einmal mit dem Ursprung von Sascha und einmal mit dem Ursprung von Mika.

Problem/Lösungsansatz

Da dies jedoch eine exponelle Rechenleistung erfordert, vereinfachen wir das ganze, indem wir sagen, dass jedes Feld nur einmal ein Gewicht benötigt und dass jedes Feld nur eine gewisse Anzahl an Updates zur Verfügung hat, das Update Limit sorgt dafür, dass jedes Feld maximal 100 einzigartige(also keine Doppelten) Gewichte von jeweils jeder Input_Node hat. Dies soll verhindern, dass durch ein Kreis im Graphen, eine unendliche Schleife entsteht.

Dies resultiert in einer theoretischen optimalen Laufzeit:

O(Limit_Updates * Anzahl an Nodes)

Das Problem an diesem Verfahren, ist zu berechnen, welches der optimale Wert für Limit_Updates ist. Da wenn man ihn zu hoch setzt, das Programm eine zulange Laufzeit hat und wenn es zuniedrig ist, in einigen Sonderfällen keine Lösung findet, obwohl theoretisch eine Existiert. Ich habe leider keinen Beweis gefunden, dass es eine Zahl für Limit_Updates gibt, bei alle möglichen Beispielen die Lösung findet. Somit habe ich einfach definiert, dass 3* Anzahl an Nodes eine gute Annäherung ist, und für alle Beispiele mit Ausnahme von Beispiel 3 eine Lösung gefunden hat.

Somit ergibt sich eine Laufzeit von:

$O(3*\text{Anzahl an Nodes} * \text{Anzahl an Nodes}) = O(3*\text{Anzahl an Nodes}^2)$

Umkehrbarkeit

Eine wichtige Frage bleibt noch offen, diese wäre wie man den Weg zurück zum Anfang findet. Dazu gibt es ein einfachen Lösungsansatz, man geht einfach Rückwärts zu der Input_Node welche das Gewicht-1 besitzt, wenn man dies durchführt bis zum Anfang(Gewicht = 0), so erhält man den Weg.

Umsetzung

Erstellung der Nodes

Als erstes müssen wir die Nodes erstellen, dafür habe ich mich für einen Umweg entschieden und zwar über die Adjenz Matrix. Diese ist sehr leicht zu erstellen und später beim erstellen der Nodes erledigt sie die ganze Arbeit. Das Bedeutet, dass wenn man eine Node erstellen will, benötigt man die Input_Nodes und Output_Nodes dieser Node. Wenn man nun die Spalte mit dem Index der Node betrachtet erhält man für alle Input_nodes bei dem entspechenden Index eine 1 und bei den Output_nodes ist es das gleiche mit den Zeilen. Wenn man nun noch die Indexe extrahiert wo sich die 1sen befinden, so erhält man alle Input/Ouput_nodes.

Algorithmus

Benötigte Variabeln:

Nodes_list => Enthält alle Nodes

Waiting_set => Enthält alle Nodes, welche noch bearbeitet werden müssen, maximal 1 mal

Klasse:

self.weight_set => Enthält alle Gewichte der Node maximal 1mal

self.waiting_dic => Speichert alle Gewichte welche die Output_nodes noch brauchen unter ihrem Index

self.visit_counter => Zählt die Besuche/Durchläufe der Node, dient als Begrenzung um nicht in eine Endlos-Schleife zu geraten

self.visit_limit => Definiert die maximale Grenze für die visits dar

Algorithmus

1) Man fügt die Start_Node(Ist das Ursprungs Feld. Weight = 0) in das Waiting_set ein.

2) Man entnimmt(.pop(0)) von Waiting_set das Element [0]. Falls der visit_counter von der Node größer gleich visit_limit ist,

A) So fängt man wieder beim Anfang von Schritt 2) bis die Länge von Waiting_set == 0 ist.

B) Falls das Limit noch nicht erreicht ist, so führt man folgende Schritte in der Klasse aus.

a) Man iteriert durch die Input_nodes und nimmt die Gewicht aus Input_nodes.Waiting_dic welche zur Node passen

b) Man nimmt die neuen Gewichte und fügt sie zum self.weight_set hinzu

c) Man iteriert durch die Output_Nodes und Überprüft ob sie das neue Gewicht+1 noch nicht besitzen

I) Falls die zutrifft, so wird diese Node zur Waiting_list hinzugefügt und das Gewicht+1 wird zum Waiting_dic unter dem key vom Index der Node gespeichert.

II) Falls nicht, so wird nichts unternommen

d) Beginne wieder beim Anfang von Schritt 2) bis die Länge von Waiting_set == 0 ist.

Analyse

Für die Analyse der beiden Graphen, betrachtet man jeden Node_index einzeln. Bedeutet, dass man zum Beispiel wenn man bei Sascha die Node[i] betrachtet, so betrachtet man die gleiche auch bei Mika. Um nun festzustellen ob es einen Treffer gibt, nutzen wir die Build-In Funktion von Sets (intersection/Überlappung = Set_A&Set_B). Nun nehmen wir die kleinste Überlappung und folgen wie oben Beschrieben den Weg zurück. Damit haben wir nun den Sprungweg von Sascha und Mika.

Beispiele

-----------------HUEPFBURG0.TXT------------------

Sasha muss auf folgende Felder springen um das Parkour zu absolvieren: 1 -> 18 -> 13 -> 10

Mika muss auf folgende Felder springen um das Parkour zu absolvieren: 2 -> 19 -> 20 -> 10

-----------------HUEPFBURG1.TXT------------------

Sasha muss auf folgende Felder springen um das Parkour zu absolvieren: 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 -> 15 -> 16 -> 17 -> 1 -> 4

Mika muss auf folgende Felder springen um das Parkour zu absolvieren: 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

-----------------HUEPFBURG2.TXT------------------

Sasha muss auf folgende Felder springen um das Parkour zu absolvieren: 1 -> 51 -> 76 -> 59 -> 42 -> 65 -> 54 -> 92 -> 27

Mika muss auf folgende Felder springen um das Parkour zu absolvieren: 2 -> 106 -> 136 -> 108 -> 81 -> 12 -> 83 -> 72 -> 27

-----------------HUEPFBURG3.TXT------------------

Dieser Parkour kann nicht erfolgreich absolviert werden.

-----------------HUEPFBURG4.TXT------------------

Sasha muss auf folgende Felder springen um das Parkour zu absolvieren: 1 -> 99 -> 89 -> 79 -> 78 -> 77 -> 76 -> 66 -> 56 -> 55 -> 54 -> 44 -> 43 -> 33 -> 23 -> 13 -> 12

Mika muss auf folgende Felder springen um das Parkour zu absolvieren: 2 -> 12 -> 11 -> 100 -> 12 -> 11 -> 100 -> 2 -> 12 -> 11 -> 100 -> 2 -> 12 -> 11 -> 100 -> 2 -> 12

Quellcode

```
def create_nodes(matrix): #Diese Funktion soll aus den Matrixen die Nodes erstellen
    amount = matrix.shape[0] #Die amount gibt an wie viele Nodes es insgesamt gibt
    nodes_list = [] #Die Liste mit den Nodes
    for i in range(amount): #Iteriert durch die Indexe der Nodes
        input_nodes = get_list_to_nodes(matrix[:,i]) #Entnimmt die Input_nodes aus der Adjenz_matrix
        output_nodes = get_list_to_nodes(matrix[i,:]) #Entnimmt die Output_nodes aus der Adjenz_matrix
        n = Node(i,input_nodes,output_nodes,3*amount) #Als visit_limit nehmen wir die Anzahl an Nodes*3
        nodes_list.append(n)
    return nodes_list



class Node:
    def __init__(self,node_index,input_nodes,output_nodes,visit_limit):
        self.node_index = node_index
        self.input_nodes = input_nodes
        self.output_nodes = output_nodes
        self.weight_set = set() #Hier werden alle Gewicht drin gespeichert. Dieses Set soll es erleichtern zu überprüfen ob es ein Gewicht bereits gibt und später beim abgleich mit den anderen Nodes
        self.waiting_dic = Node.init_dic(self,self.output_nodes) #In diesem Dic werden die Gewichte gespeichert, welche die Output_nodes noch benötigen PS: die Gewichte wurden bereits angepasst dh. Gewicht +1
        self.visit_counter = 0 #diese Variable soll zählen wie oft die Node besucht worden ist
        self.visit_limit = visit_limit#visit_limit


    def init_dic(self,list): #Erstellt aus einer Liste ein Dic, die Elemente der Liste dienen dabei als Keys und jeder Key wird jeweils als leere Liste definiert
        dic = {}
        for i in list:
            dic[i] = []
        return dic


    def start_node(self): #initialisiert diese Node als Ursprung
        self.weight_set.add(0) #Da der Ursprung das Gewicht 0 hat
        to_do = copy.deepcopy(self.output_nodes)
        for n in self.output_nodes: #Da keine Node bislang einen Wert zugewiesen bekommen hat, bedeutet das dass keine Node bislang das Gewicht 1 hat
            self.waiting_dic[n].append(1)
        return to_do #Gibt die Nodes wieder welche zum Waiting_set hinzugefügt werden müssen


    def update_node(self,nodes_list): #Diese Funktion soll die Node und ihre Variabeln mit den neuen Informationen Updaten
```

```
        to_do = set()
        self.visit_counter +=1 #Erhöht den Zähler um 1
        for n in self.input_nodes:
            node = nodes_list[n] #Da self.input_nodes nur den Index der Input_nodes speichert, wird in diesem Schritt die entsprechende Klassse entnommen
            new_weights = node.waiting_dic[self.node_index] #Wir rufen das Waiting_dic von der entsprechenden Node auf und entnehmen die gewichte die zu unserer Node passen
            for w in new_weights: #Nun iterieren wir durch die neuen Gewichte.
                if w not in self.weight_set and self.visit_counter < self.visit_limit: #überprüft ob dieses Gewicht nicht bereits vorhanden ist und ob man nicht bereits das visit_limit überschritten hat
                    self.weight_set.add(w) #Fügt w in weight_set
                    to_do = to_do.union(Node.check_outputs(self,w,nodes_list)) #Fügt die gefundenen Outputs ins to_do set
        return to_do


    def check_outputs(self,weight,nodes_list): #Diese Funktion soll überprüfen welche der Output_nodes nochmal geupdatet werden müssen und gibt ein Set mit diesen zurück(Indexe)
        to_do_set = set()
        for i in self.output_nodes: #iteriert durch alle Output_Nodes
            n = nodes_list[i] #Da die Output_nodes als indexe gespeichert sind, muss man sie noch in die Eigentliche Node um wandeln
            if weight+1 not in n.weight_set and n.visit_counter < n.visit_limit: #überprüft ob das Gewicht+1 noch nicht in n vorhanden ist und ob der visit_counter von n noch nicht das Limit erreicht hat
                to_do_set.add(n.node_index) #Falls dies zutrifft so wird der index von n zur ToDo_list hinzu gefügt
                self.waiting_dic[n.node_index].append(weight+1) #Und das Gewicht+1 wird zum waiting_dic unter dem Index von n eingespecieherrt
        return to_do_set


def main(nodes_list,start_index): #Dies soll die Haupt Funktion darstellen, welche alle anderen aufruft
    waiting_list = nodes_list[start_index].start_node() #Man initiert sowohl die start_node als auch das Waiting_list
    while True:
        next_index = waiting_list.pop(0) #Man entnimmt das [0] Element von waiting_list
        next_node = nodes_list[next_index] #Da es ein Index ist muss dieser noch dem entsprechendem Node zugewiesen werden
        to_do = next_node.update_node(nodes_list) #Diese Funktion gibt die nächsten zuberarbeitenden Nodes zurück
        for i in to_do:
            if i not in waiting_list: #iteriert durch die To_dos und überprüft ob sie nicht bereits in der Warte_liste sind
                waiting_list.append(i) #Falls sie noch nicht in der Waiting_list sind, so werden sie hinzugefügt
        if len(waiting_list) == 0: #Falls keine Elemente mehr im Waiting_list vorhanden sind, so ist der Algorithmus fertig
            return nodes_list

#-------------------------------------------------------------------------------
```

```
def check_intersection(nodes_Sasha,nodes_Mika): #Diese Funktion soll überprüfen ob sich Mika und Sasha überhaupt einmal auf einem Feld treffen
    for i in range(len(nodes_Sasha)): #Iteriert durch die Indexe von den Knoten von Mika und Sasha. Beide listen sind gleich lang dh. es macht kein unterschied ob wir da Sasha oder Mika einfügen
        intersection = nodes_Sasha[i].weight_set & nodes_Mika[i].weight_set #Der Operator & gibt ein Set zurück welcher die Gemeinsamen Gewicht von nodes_Sasha[i].weight_set und nodes_Mika[i].weight_set enthält
        if len(intersection) > 0: #Falls es eine übereinstimmung gibt bedeutet das dass es eine möglichkeit gibt das Parkour zu lösen
            return True #Falls es einen Treffer gibt, so wird True zurück gegeben
    return False #Ansonsten False

def get_smallest_intersection(nodes_Sasha,nodes_Mika): #Dies soll den Knoten angeben an welchem sich Mika und Sasha mit den wenigesten Sprüngen treffen
    smallest_intersection = None #Hier soll die Node gespeichert werden mit welcher man den Parkour am schnellsten lösen kann. In der Form (Gewicht,Index)
    for i in range(len(nodes_Sasha)): #Iteriert durch die Indexe von den Knoten von Mika und Sasha. Beide listen sind gleich lang dh. es macht kein unterschied ob wir da Sasha oder Mika einfügen
        intersection = nodes_Sasha[i].weight_set & nodes_Mika[i].weight_set #Der Operator & gibt ein Set zurück welcher die Gemeinsamen Gewicht von nodes_Sasha[i].weight_set und nodes_Mika[i].weight_set enthält
        for weight in intersection: #iteriert durch die übereinstimmungen, falls es keine gibt, so wird dieser Teil übersprungen und man beginnt wieder vom Anfang der Schleife
            if smallest_intersection is None: #Falls für smallest_intersection nach kein Tupel existiert,
                smallest_intersection = (weight,i) #So wird unabhängig davon was im Tupel ist, dieser als smallest_intersection definiert.
            elif smallest_intersection[0] > weight: #Falls man nun ein neue Übereinstimmung findet, welches ein kleineres Gewicht als das bereits bekannt,
                smallest_intersection = (weight,i) #so updatet man smallest_intersection mit diesem neuen Tupel
    return smallest_intersection #Gibt smallest_intersection zurück

def track_back(nodes_somebody,smallest_intersection): #Diese Funktion soll den Weg zurück finden, von der kleinsten Übereinstimmung zurück zum Ursprung
    # smallest_intersection => (weight,index)
    path = [smallest_intersection] #Speichert den Weg in Form von Tupeln. Diese wiederum sind aufgebaut wie smallest_intersection
    while True:
        if path[0][0] == 0:
            return path
        past_step = path[0] #Da der letzte Sprung immer vorne eingesetzt wird, erhält man mit path[0] den letzten Sprung
        path.insert(0,search(past_step,nodes_somebody)) #Fügt den neu gefundenen Sprung wieder an den Anfang, die Funktion search() findet den nächsten Sprung

def search(past_step,nodes_somebody): #Sucht nach dem Feld, welches das gesuchte Gewicht besitzt
    past_node = nodes_somebody[past_step[1]] #wandelt den Index in das tatsächliche Objekt um
    for node_index in past_node.input_nodes: #iteriert durch die Input_nodes
        weight_set = nodes_somebody[node_index].weight_set #Entnimmt den Input_nodes die Gewichtsliste
```

if past_step[0]-1 in weight_set: #Falls das Gewicht-1 in dem weight_set der betrachteten Node ist,

return (past_step[0]-1,node_index) #dann gibt man den Tupel zurück und subtrahiert von past_step[0] -1, da dies nun das nächst kleinere Feld ist.