Aufgabe 5.1 - Huffmann-Code

DIE KATZE TRITT DIE TREPPE KRUMM

Sortierung nach Wert(Häufigkeit): ([ ] entspricht dem Leerzeichen.)

Zeichen:	D	I	E	[ ]	K	A	T	Z	R	P	U	M
Wert:	2	3	5	5	2	1	5	1	3	2	1	2

Zusammenfassungen:

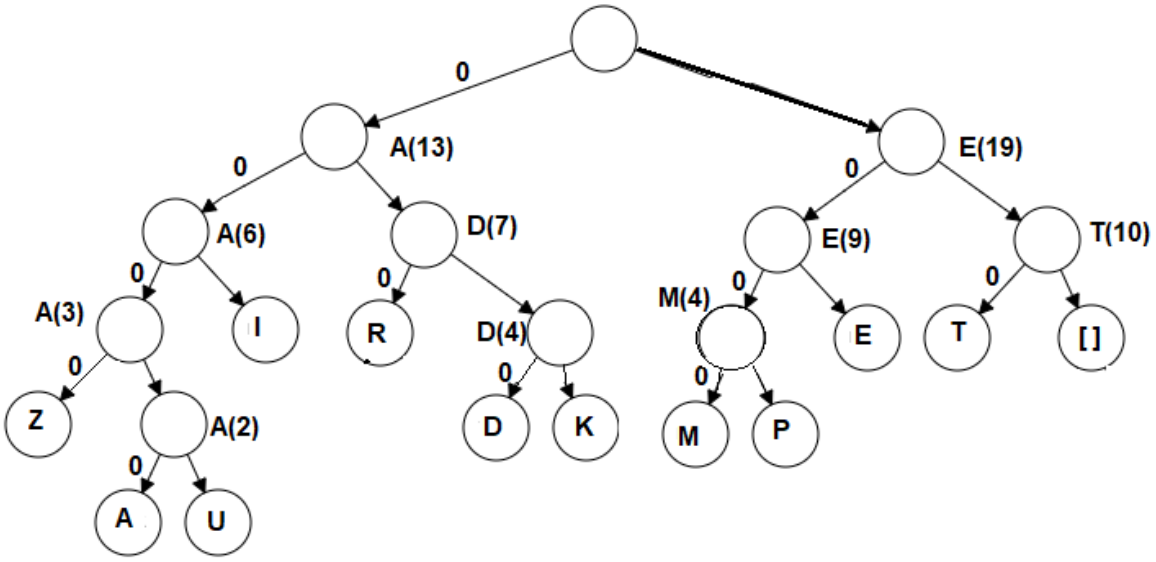
A,U,Z mit je Wert 1, → A&U wird zu A(2) Notiz:Notation als Zeichen(Wert)

Z(1) und A(2) zu A(3) ; D(2) und K(2) zu D(4) ; M(2) und P(2) zu M(4)

A(3) und I(3) zu A(6) ; R(3) und D(4) zu D(7) ; M(4) und E(5) zu E(9)

T(5) und [ ](5) zu T(10) ; A(6) und D(7) zu A(13) ; E(9) und T(10) zu E(19)

Damit bleiben A(13) und E(19) als zwei Zeichen. Der Baum sieht dann wie folgt aus:



Es folgt die Codierung:.

Z -> 0000	A -> 00010	U -> 00011
I -> 001	R -> 010	D -> 0110
K -> 0111	M -> 1000	P -> 1001
E -> 101	T -> 110	[ ] –> 111

Der Satz wird als kodiert zu:

0110,001,101,111,0111,00010,110,0000,101,111,110,010,001,110,110,111,0110,001,101,111,
110,010,101,1001,1001,101,111,0111,010,00011,1000,1000

5.2 - greedy Matching

Die Grundidee ist das durchlaufen des ungerichteten Baumes mit einer Breitensuche. Da der Baum maximal n-1 Kanten besitzt, läuft die Suche in $O(2n) = O(n)$, bleibt also in linearer Laufzeit und es gibt auch explizit keine Rückwärtskanten in Bäumen. Bei der Breitensuche füllen wir der Reihe Nach ein Array mit n Plätzen mit den Knoten und speichern für jeden Knoten den Elternknoten (Es gibt nur einen, da die Breitensuche einen gewurzelten Baum ergibt). So ist die (zufällig gewählte) Wurzel als erstes im Array, die übrigen Knoten nach Tiefe geordnet. Die letzten Elemente sind also die Blätter der untersten Tiefe. Nun durchlaufen wir

das Array von hinten.
Für den letzten Knoten wird geprüft ob die Kante zum Eltern Knoten das Matching bricht, wenn ja, wird er einfach übergangen. Falls nicht, wird die Kante zum Elternknoten ins Matching gesetzt und für den Elternknoten notiert, dass er gematcht ist. Das passiert für jeden Knoten in einer einzelnen Prüfung, da nur geschaut werden muss, ob er selbst oder der Elternknoten bereits gematcht ist. Stellen wir uns vor alle gematchtetn würden aus dem Graph entfernt, würden wir stets nur auf Blättern arbeiten und uns somit auch keine Optionen für mehr Kanten verbieten.

```
\\v0 ist zufallswurzel A_list ist adjazenzliste
i = 0
depth_list = [(v0,v0)] \\array der aktuellen tiefe.
next_list = [] \\array der naechsten tiefe
visited = {v0} \\alle bereits bearbeiteten, da ungerichtet
Array_np = [].size(n) \\Array mit nodes und deren parents mit size n
while i < n:{
    for (node,parent) in depthlist:{ \\jedes aktueller Tiefe
        Array_np[i] = (node,parent) \\im Zielarray einsetzen
        i++                         \\ und anzahl plus 1
        for child in A_list[node]:{  \\ und die Kinder
                if child not in visited:{ \\ parent ist visited!
                        next_list.append((child,node)) \\child,parent
                        visited.add(child)
                        } \\endif
        } \\end for child

    } \\depthlist ende
    depth_list = next_list \\ naechste tiefe
    next_list = []
} \\endwhile
\\ alle im Array, tiefe nach sortiert.
M = {}
matched = {}
for i = n to 1:{ \\ von hinten nach vorne (v0,v0) nicht zu pruefen
        (blatt,parent) = Array_np[i]
        if (blatt not in matched) and (parent not in matched):{
        M.add({blatt,parent}) \\Kante dazugeben
        matched.add(blatt)        \\ und adjazente gematcht setzen.
        matched.add(parent)
        }

}
return M
```

Laufzeit:
Die untere for schleife bleibt offensichtlich in linearer Laufzeit, da nur max. 5 Operationen pro durchlauf stattfinden, und sie n-mal durchläuft. Die Obere Schleife ist eine modifizierte Breitensuche, die immer tiefen weise durchläuft. Auf einem gewöhnlichen Graphen wäre deren Laufzeit in O(#Knoten + #Kanten). In Bäumen gibt es n-1 Kanten, bzw. in allen Adjazenzlisten zusammen maximal O(n) Einträge. Also läuft diese Suche in O(n+O(n)) = =O(n).

Korrektheit:
Die Breitensuche liefert mir die geforderte Arraystruktur durch das strikte durchlaufen einzelner Tiefen. Beim Durchlaufen von hinten nach vorne und dem gedachten Streichen der gematchten Knoten werden stets nur Kanten von Quasi-Blättern zu deren Eltern ins Matching eingefügt. Damit entsteht ein maximales Matching.
Angenommen es gäbe ein anderes Matching M', das mehr Kanten enthielt, kann ich für jede Kante in diesem M', die einen Knoten v trifft, eine aus meinem Matching nehmen, die auch v trifft. Da diese neue Kante stets zu einem Blatt führt, verletzt sie kein weiteres Matching, kann also nie die Größe reduzieren. Im Gegenteil, dadurch könnten sogar weitere Kanten zum hinzufügen entstehen, damit ist das gefundene Matching stets mindestens so groß wie alle anderen, hat also die maximale Kardinalität.

5.3 - Algolaner Brücken.

Kreisfreiheit lässt eine Art Spannbaum vermuten, der zwar durch das Budget minimal gehalten werden sollte, aber nicht unbedingt alle Inseln verbindet. Wichtig ist hierbei die Maximale Anzahl an Brücken festzuhalten. Könnte jede mit jeder und sich selbst verbunden werden, wären n^2 Brücken möglich. Die maximale Anzahl ist also durch n^2 nach oben beschränkt.
a)
Wenn wir die möglichen Brücken nach ihren Kosten sortieren können, so erhalten wir eine Analogie zu einem gewichtete Graphen, deren Kantengewichte (=Kosten) wir sortieren. Analog sollte dann das Gewicht minimal bleiben, damit wir die maximale Anzahl an kreisfreien Kanten mit Gesamtgewicht unter B bestimmen können. Der Kruskal-MST Algorithmus tut genau dies, wenn er in jedem Schritt prüft, ob das Gesamtgewicht noch unter B bleiben würde. Nach prüfen der Union-Find u=v noch eine Prüfung ob bisheriges Gesamtgewicht+ k(u,v) > B. Falls ja bricht der Algorthmus hier ab und gibt die bisherigen Kanten aus. Also ist der gewünschte Algorithmus Kruskal mit Abbruchbedingung.
b)
Es stellt sich die Frage, ob Kruskal mit k = n^2 Kanten noch in der Laufzeit $O(n^2 \cdot log\ n)$ bleibt. Die Laufzeit setzt sich zusammen aus dem sortieren der Kanten und dem erstellen des Baumes. Das sortieren von k Kanten geht in $O(k\ log\ k) = O(n^2 \cdot log(n^2))$. doch mit den Logarithmusgesetzen ist das eben auch $O(n^2 \cdot 2log\ n) = O(n^2 log\ n)$. Das erzeugen des Baumes benötigt $O(n+klog\ n) = O(n+n^2 \cdot log\ n) = O(n^2 \cdot log\ n)$ Somit bestimmt der Algorithmus den (eventuell nicht kompletten) Spannbaum in $O(n^2 \cdot log\ n)$. Die Laufzeitschranke bleibt eingehalten.
c)
Der Kruskal Algortihmus wählt stets die günstigste mögliche Brücke, die keinen Kreis schliesst. Das Ergebnis ist auf jeden Fall Kreisfrei. Er bricht ab, wenn das Budget überstiegen werden würde, also wird auch das Budget eingehalten. Und Kruskal wählt in jedem Schritt die günstigste Brücke. Gäbe es ein Brückennetzwer, das mehr Brücken ohne Kreis hätte, dann hätte Kruskal an einem Punkt nicht die günstigste genommen. Widerspruch zur Tatsache, dass Kruskal immer die günstigste Brücke nimmt. Auch der Fall, dass eine günstigste später noch möglich wäre, wenn eine andere genommen worden wäre, trit nicht ein, da sonst die Wahl zu dem Zeitpunkt nicht optimal ist, dass muss sie aber sein.

5.4 - Dynamisch mit Teilsequenzen

a)
Die Basisfälle sind die Startwerte die wir kennen. Dabei ist lgt(0,0) mit 0 Zeichen aus S und 0 Zeichen aus T offensichtlich 0. Auch die beiden Fälle mit beliebigen k oder l lgt(k,0) oder lgt(0,l) haben offensichtlich die Länge 0. Das sind die drei Basisfälle.
b)
lgt(k,l) ist maximal um eins größer als einer der drei Vorgänger und da wir in jedem Schritt nur einen weiteren Buchstaben hinzufügen, müssen wir auch nur eine Gleichheit prüfen. Wenn der jeweils nächste Buchstabe nicht gleich ist, dann ändert sich die Folge nicht.
Ist also S1[k] ungleich S2[l], dann gilt lgt(k,l) = maxlgt(k-1,l), lgt(k,l-1), das bisherige Maximum bleibt Maximum.
Ist aber beim schrittweisen durchgehen der nächste Buchstabe gleich, so kommt bei beiden Längen der nächste Buchstabe hinzu und die neue Länge ist lgt(k,l) = lgt(k-1, l-1) +1.
c)

```
\\Rekursiv laesst es sich wie folgt loesen:
lgt(n,m) = if (m*n = 0) then 0
        else:{
                if S1[n]!=S2[m] then: max{lgt(k-1,l) , lgt(k,l-1)}
                else:{lgt(k-1, l-1) +1}
        }
\\Die Iterationsschleife waere auch eine gute Moeglichkeit:
\\sie zeigt auch die Laufzeit perfekt:
int lgt(string s1, string s2,int len1, int len2):{
values[len1][len2] = 0 \\zeros in 2D-Array of len1*len2
    for(int k=1; k<=len1;k++){
        for(int l=1; l<= len2;l++){
            if s1[k-1] == s2[l-1] :{ // Strings bei 0 indiziert.
            values[k][l] = values[k-1][l-1] +1;}
            else:{
            values[k][l] = maximum(values[k-1][l], values[k][l-1]);}
        }
    }
    return vlaues[len1-1][len2-1] \\last value computed
}
```

Anhand der Iterationsschleifen lässt sich leicht erkennen, die Laufzeit beträgt $O(n \cdot m)$.