Aufgabe 4.1 - Spannbaum

Kantennotation: $\{V_a, V_b\} \stackrel{\wedge}{=} \vec{ab}$
Sortierung nach Gewicht: Jede Spalte ist Kante und deren Gewicht

$\vec{45}$	$\vec{47}$	$\vec{29}$	$\vec{38}$	$\vec{37}$	$\vec{48}$	$\vec{78}$	$\vec{12}$	$\vec{19}$	$\vec{26}$	$\vec{69}$	$\vec{56}$	$\vec{03}$	$\vec{08}$	$\vec{59}$	$\vec{67}$	$\vec{01}$	$\vec{04}$	$\vec{15}$	$\vec{23}$
1	2	3	3	4	4	4	5	5	6	6	7	8	8	8	8	9	9	9	9

$[v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9]$
Kürzeste Kanten: $\{v_4, v_5\} = \vec{45}$ mit Gewicht 1; $find(v_4) \neq find(v_5)$
Hinzufügen von $\vec{45}$ und aktualisiere Union, V_4 an V_5 hängen;
Neues Elternarray : $[v_0, v_1, v_2, v_3, v_5, v_5, v_6, v_7, v_8, v_9]$
Nächste kürzeste Kanten: $\vec{47}$, $find(4) \neq find(7)$
Hinzufügen von $\vec{47}$ und aktualisiere Union, V_7 an V_5, da V_5 Wurzel des größeren Baums.
Neues Elternarray : $[v_0, v_1, v_2, v_3, v_5, v_5, v_6, v_5, v_8, v_9]$
Nächste kürzeste Kanten: $\vec{29}$, $find(2) \neq find(9)$
Hinzufügen von $\vec{29}$ und aktualisiere Union, V_2 an V_9, da V_9 größer.
Neues Elternarray : $[v_0, v_1, v_9, v_3, v_5, v_5, v_6, v_5, v_8, v_9]$
Nächste kürzeste Kanten: $\vec{38}$, $find(3) \neq find(8)$
Hinzufügen von $\vec{38}$ und aktualisiere Union, V_3 an V_8, da V_8 größer.
Neues Elternarray : $[v_0, v_1, v_9, v_8, v_5, v_5, v_6, v_5, v_8, v_9]$
Nächste kürzeste Kanten: $\vec{37}$, $find(3) \neq find(7)$
Hinzufügen von $\vec{37}$ und aktualisiere Union, V_5 an V_8, da V_8 größer.
Neues Elternarray : $[v_0, v_1, v_9, v_8, v_5, v_8, v_6, v_5, v_8, v_9]$
Nächste kürzeste Kante: $\vec{48}$, $find(4) = find(8)$!
Nächste kürzeste Kante: $\vec{78}$, $find(7) = find(8)$!
Nächste kürzeste Kante: $\vec{12}$, $find(1) \neq find(2)$
Hinzufügen von $\vec{12}$ und aktualisiere Union, V_1 an V_9, da V_9 Wurzel des größeren Baums. Neues Elternarray : $[v_0, v_9, v_9, v_8, v_5, v_8, v_6, v_5, v_8, v_9]$
Nächste kürzeste Kante: $\vec{19}$, $find(1) = find(9)$!
Nächste kürzeste Kante: $\vec{26}$, $find(2) \neq find(6)$
Hinzufügen von $\vec{26}$ und aktualisiere Union, V_6 an V_9, da V_9 Wurzel des größeren Baums.
Neues Elternarray : $[v_0, v_9, v_9, v_8, v_5, v_8, v_9, v_5, v_8, v_9]$
Nächste kürzeste Kante: $\vec{69}$, $find(6) = find(9)$!
Nächste kürzeste Kante: $\vec{56}$, $find(5) \neq find(6)$
Hinzufügen von $\vec{56}$ und aktualisiere Union, V_9 an V_8, da V_8 Wurzel des größeren Baums.
Neues Elternarray : $[v_0, v_9, v_9, v_8, v_5, v_8, v_9, v_5, v_8, v_8]$
Nächste kürzeste Kante: $\vec{03}$, $find(0) \neq find(3)$
Hinzufügen von $\vec{03}$ und aktualisiere Union, V_0 an V_8, da V_8 Wurzel des größeren Baums.
Alle in einer Union. Damit ist der Spannbaum
G' =V , $\{\vec{45}, \vec{47}, \vec{29}, \vec{38}, \vec{37}, \vec{12}, \vec{26}, \vec{56}, \vec{03}\}$ Letztes Elternarray:$[v_8, v_9, v_9, v_8, v_5, v_8, v_9, v_5, v_8, v_8]$

b)
Prim in einer Tabelle, mit der Menge S und den jeweiligen kreuzenden Kantengewichten zu den Knoten notiert als deren Indices.

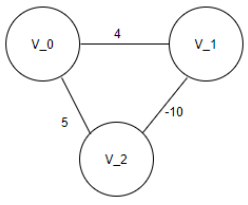
S	0	1	2	3	4	5	6	7	8	9	kürzeste	gewählte:
$\{0\}$	-	9	∞	8	9	∞	∞	∞	8	∞	$\vec{03}, \vec{08}$	$\{0,3\}$
$\{0,3\}$	-	9	9	-	9	∞	∞	4	3	∞	$\vec{38}$	$\{3,8\}$
$\{0,3,8\}$	-	9	9	-	4	∞	∞	4	-	∞	$\vec{37}, \vec{84}, \vec{87}$	$\{3,7\}$
$\{0,3,7,8\}$	-	9	9	-	2	∞	8	-	-	∞	$\vec{47}$	$\{4,7\}$
$\{0,3,4,7,8\}$	-	9	9	-	-	1	8	-	-	∞	$\vec{45}$	$\{4,5\}$
$\{0,3,4,5,7,8\}$	-	9	9	-	-	-	7	-	-	8	$\vec{56}$	$\{5,6\}$
$\{0,3,4,5,6,7,8\}$	-	9	6	-	-	-	-	-	-	6	$\vec{26}, \vec{69}$	$\{2,6\}$
$\{0,2,3,4,5,6,7,8\}$	-	5	-	-	-	-	-	-	-	3	$\vec{29}$	$\{2,9\}$
$\{0,2,3,4,5,6,7,8,9\}$	-	5	-	-	-	-	-	-	-	-	$\vec{12}, \vec{19}$	$\{1,2\}$

Der Spannbaum ist dann identisch zu dem aus der a)

4.2 - Minimales Produkt

(NEIN, ich möchte keine Transformation mit dem Logarithmus nutzen!)
Wir nutzen denselben Ansatz wie der Algorithmus von Kruskal, indem wir in jedem Schritt die kleinste Kante wählen, die keinen Kreis schliesst. So bleibt auch das Produkt minimal. Da bei einem Spannbaum eben nicht die Länge der Wege an sich zählen, sondern nur das Gesamtprodukt aller verwendeten Kanten, so bleibt auch dieses Produkt minimal. Der Algorithmus läuft wie folgt:
Starte mit jedem Knoten als einzelner Knoten in einem Wald. Suche aus allen Kanten die kürzeste heraus und prüfe mithilfe einer union-find Struktur, ob sie einen Kreis schliesst. Falls ja, verwerfe sie. Falls nicht, füge sie dem Spannbaum hinzu und aktualisiere für die adjazenten Knoten die Union. Breche ab, wenn keine Kante über bleibt, sind alle maximalen Komponenten erreicht, ist der Spannbaum gespannt.
Die Korrektheit lässt sich wieder durch einen Widerspruch zeigen. Würde es einen Spannbaum geben, dessen Gesamtprodukt geringer wäre, müsste an einer Stelle eine Kante mit geringerem Gewicht verwendet werden. Dann hätte unser Algorithmus aber diese gewählt. Daher kann es keine Kante mit kleinerem Gewicht und damit auch keinen Spannbaum mit kleinerem Produkt geben.
Die Laufzeit errechnet sich wie folgt.Zunächst müssen die Kanten sortiert werden. Das ergibt $O(|E|log(|E|))$ Es muss für jede Kante bestimmt werden, ob sie einen Kreis schliesst. Mithilfe der Union Operation geschieht das für alle Kanten gesamt in $O(|E|log(|V|))$. Die gesamte Laufzeit sortieren + spannen ist damit $O(|E| \cdot log(|E| \cdot |V|))$

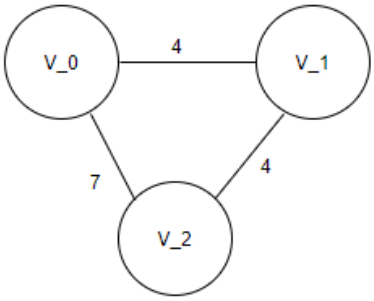
4.3 - Manipulationen und Gewichte

a)


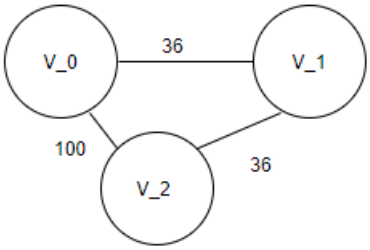
Auf diesem Graphen gibt Dijkstra startend in V_0 den Weg von V_0 direkt zu V_1 als kürzesten aus. Danach findet sich die negative Kante, aber der Weg von 0 zu 1 wird nie aktualisiert. Dabei ist der Weg (v_0, v_2, v_1) kürzer.
$\rightarrow$ Dijkstra: $S = \{v_0\}$ und $\{v_0, v_1\}$ als kürzeste Kante, dann $S = \{v_0, v_1\}$ und update weg zu v_2 mit Länge -6. Hinzufügen von v_2 zu S, $S = \{v_0, v_1, v_2\}$

b)

i) Die Wege bleiben nicht erhalten. Wird der benstehende Graph durch eine Subtraktion mit 2 verändert, somit ergeben sich andere Wege.
Vorher direkt von V_0 zu V_2, danach über V_1.

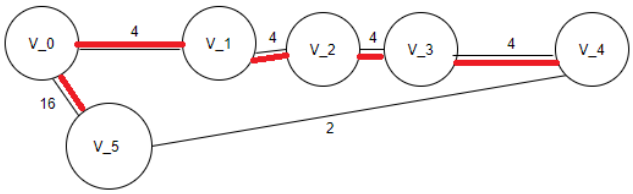


ii)
Der folgende Graph ändert auch seine kürzesten Wege durch das Wurzeln.
Vorher der Weg zu V_2 über V_1, danach direkt.



c)

i)
Multiplizieren wir einzelne Kanten des Baumes mit einer Konstanten c, so ändert sich die gesamtlänge eines beliebigen kürzesten Weges ebenso um den Faktor c. Da dies gleichmäßig und konstant auf den kürzesten Wegen passiert, bleiben alle relativen Verhältnisse erhalten und sie verändern sich untereinander nicht. Auch alle anderen Wege, die nicht komplett im Baum der kürzesten Wege vorher waren bleiben länger als die bisher kürzeren, da wir die kürzeren mit einem Faktor <1 multiplizieren.
ii)
Hier verändern sich die Verhältnisse untereinander, da der Logarithmus nicht gleichmäßig ändert. Es lässt sich ein Beispielgraph konstruieren, der seine kürzesten Wege ändert.



Hier sind die ursprünglichen kürzesten Wege rot markiert, Nach dem Anwenden des Logarithmus ist der Weg von V_0 nach V_4 über V_5 kürzer.

4.4 - Kantenrichtung

Die Grundidee ist das durchlaufen des ungerichteten Graphens in eine Richtung, das belegen in $\vec{E}$ dieser Kante in Laufrichtung und das abspeichern bisher erreichter Knoten in einer Menge Z. Gibt es einen Weg zurück in diese Menge Z, so sind alle dazwischenliegenden Knoten stark zusammenhängend, da dieser Weg einen Kreis bildet. Kann ich auf dem Weg durch den Graphen alle Knoten erreichen, so ist der entsehende Graph $\vec{G}$ auf jeden Fall schwach zusammenhängend. Da der gegebene Graph zusammenhängend ist, ist dieses Kriterium erfüllt. Bleibt die Frage nach den Richtungen der nicht durchlaufenen Kanten. Falls ich auf dem Weg durch den

Graph eine Kante finde, die zurück führt, dann richte ich sie in die Richtung, die vom aktuellen Standpunkt weg zeigt. Da die ungerichteten Kanten auch im Folgeknoten existieren, muss ich die zum direkten Vorgänger ignorieren, denn diese sind gerade in eine Richtung gerichtet worden.
Dieses gesamte Konstrukt lässt sich durch eine leicht modifizierte Tiefensuche darstellen, bei der wir als zusatz noch den direkten Vorgänger übergeben, damit wir die Kanten nicht doppelt belegen.
Bleibt noch zu überprüfen, ob der entstehende Zielgraph auch stark zusammenhängend ist. Von dem ersten Knoten erreiche ich stets alle Knoten, das ergibt tsuche und der gegebene Zusammenhang. Also muss noch für den inversen Graphen geprüft werden, ob alle besucht werden. Sind alle von 0 aus in beiden richtungen erreichbar, ist der Graph stark zusammenhängend, alle können zur 0 und von dort zu allen.
Durch den gegebenen Code aus dem Skript lässt sich das leicht in Pseudocode realisieren:

```
\\brauche keinen globalen Start, da zusammenhaengend
void moddedsuche(aktiv,last):
        {
        Knoten *p ; besucht[aktiv] = 1;
        for (p = A_Liste [aktiv]; p !=0; p = p->next)
        if (p != last){
                if (!besucht [p->name]){
                targetV.add(p->name);
                targetE.add( (aktiv->name,p->name) );
                moddedsuche(p->name, aktiv);
                        }
                else: //rueckwaertskante
                        targetE.add( (aktiv->name,p->name);                    }
        }
besucht[v] = [0..0] // array aus v Nullen v= anzahl der Knoten
targetV = {};
targetE = {}
moddedtsuche(knoten0,knoten0)\\ starten im ersten zufaelligen Knoten
dirg = (targetV,targetE)
//der gerichtete Graph ist nun in dirg gespeichert.
Sei Die Funktion reverse(Graph) gegeben, die alle Kanten dreht.
besucht2[v] = [0..0]
//tsuche aus VL in inversem dirg:
tsuche(Graph=reverse(dirg),start=knoten0)
if besucht == besucht2 : // alle besucht
        return dirg
else:  ERROR(Keine Loesung)
```

Korrektheit:
Wir wissen aus der Vorlesung, Tiefensuche liefert Baumkanten und Rückwärtskanten, Alle Baumkanten sind in Richtung der Tiefensuche laufende neuen Kanten. Alle nicht Baumkanten sind somit Rückwärtskanten und schliessen Kreise.Sie zeigen in Richtung Start. N.V. erreicht die Tsuche alle Knoten, bleibt nur zu prüfen, ob der resultierende Graph stark zusammenhängend ist. Das wird mit dem inversen Suche abgefangen. Die ausgegebene Lösung ist also immer Korrekt, und wird keine Lösung ausgegeben, fehlt mindestens eine Rückwärtskante um einen Kreis zu schliessen. Auf diesem Weg gibt es dann eine Brücke.
Laufzeit:
Die Ergänzungen in der modifizierten tsuchehaben je nur Laufzeit O(1), also 2 Tiefensuchen und einmal invertieren ergibt $O(2|V| + 2|E| + |E|) = O(|V| + |E|)$