Martin Götze, Bernhard Streit

2. Übung zu Informatik B

Tutorium: Marco Block, Fr 12–14 Uhr

1. **Senke**

   Eine Senke in einem gerichteten Graphen $G = (V, E)$ ist ein Knoten mit Ingrad $|V| - 1$ und Ausgrad 0. Der Graph sei in Adjazenzmatrixdarstellung (mit $|V|^2$ Einträgen) gegeben. Argumentieren Sie, dass man nur linear in $|V|$ viele Anfragen an Matrixeinträge stellen muss, um zu entscheiden, ob V eine Senke besitzt.

   Sei also $G = (V, E)$ ein schlichter ungerichteter Graph $|V| = n$, o.E. $V = \{1, \dots, n\}$, und $A = (a_{ij})$ die Adjazenzmatrix von G.

   Um für ein festes $1 \leq i \leq n$ festzustellen, ob i eine Senke in G ist, braucht man $2n - 1$ (also $\Theta(n)$, d.h. linear viele) Anfragen an Matrizeneinträge:

1 Ist_Senke(i) *//prüft, ob Knoten i eine Senke ist*
2
3 Senke $\leftarrow$ ($a_{ii} = 0$)
4 **Für** $j = 1$ bis n
5   **Wenn** $i \neq j$
6   **Wenn** $a_{ij} \neq 0$ *// ein Weg von i aus*
7   Senke $\leftarrow$ Falsch
8   **Wenn** $a_{ji} = 0$ *// kein Weg* $j \to i$
9   Senke $\leftarrow$ Falsch
10 Senke zurückgeben

   Es wird also (in Zeile 5) einfach überprüft, ob ein Weg von i ausgeght (i ist keine Senke, wenn das der Fall ist) und ob (in Zeile 7) ein Weg von j nach i geht (das ist nötig). Offenbar werden gerade $2(n - 1) + 1 = 2n - 1$ Anfragen an Einträge von A gestellt.

   Nun kann man diesen Algorithmus nicht einfach für jedes i durchführen, da die Anzahl der Anfragen dann $n \cdot (2n - 1) = \Theta(n^2)$ und somit nicht mehr linear wäre. Man beachte aber: Ist $i \neq j$ und $a_{ij} = 0$, so kann j keine Senke sein und ist $a_{ij} = 1$, so kann i keine Senke sein, durch *eine* Anfrage an A kann man also einen Knoten vom Senke sein ausschließen, da ein Graph höchstens eine Senke haben kann muss man Ist_Senke also höchstens für einen Knoten ausführen:

11 Hat_Senke *// prüft, ob G eine Senke hat*
12
13 s $\leftarrow$ 1 *// Initialisierung*
14 **Für** $i = 2$ bis n
15   **Wenn** $a_{si} \neq 0$
16         $s \leftarrow i$ *// s ist sicher keine Senke, aber vielleicht i*
17   *// ansonsten ist i keine Senke, weiter*
18 s ist die einzig mögliche Senke, also gib Ist_Senke(s) zurück

   Nach dem Durchlauf der Schleife in Zeilen 14–17 hat s den Wert des einzigen Knotens der mögliche Senke ist (den jeder andere kann keine sein, da er entweder als ehemaliger Wert von s überschrieben wurde, da von ihm ein Knoten ausgeht, oder es einen Knoten gibt, von dem aus keine Kante zu ihm führt).

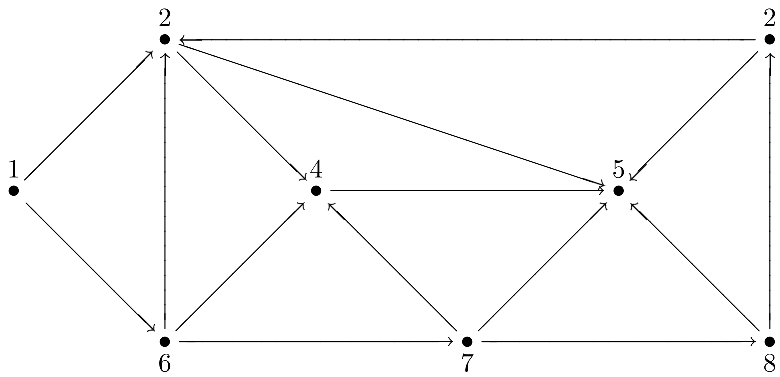
   Mit diesem Algorithmus müssen also insgesamt

$$n - 1 + 2n - 1 = 3n - 2 = \Theta(n)$$

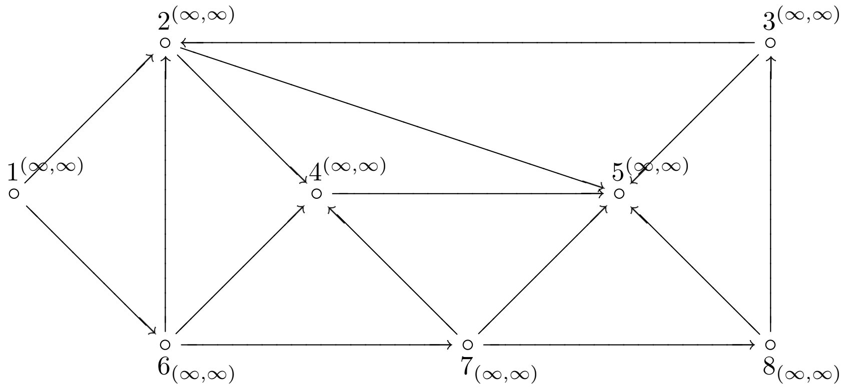
   d.h. linear in n viele Anfragen an A gestellt werden.

2. **topologische Sortierung – DFS** Finden Sie mittels dfs eine topologische Sortierung der Knoten des Graphen in der Abbildung. Beschreiben Sie einen

Algorithmus, der gleichzeitig für den gerichteten Eingabegraphen prüft, ob dieser ein dag ist (d.h. eine Sortierung möglich ist).

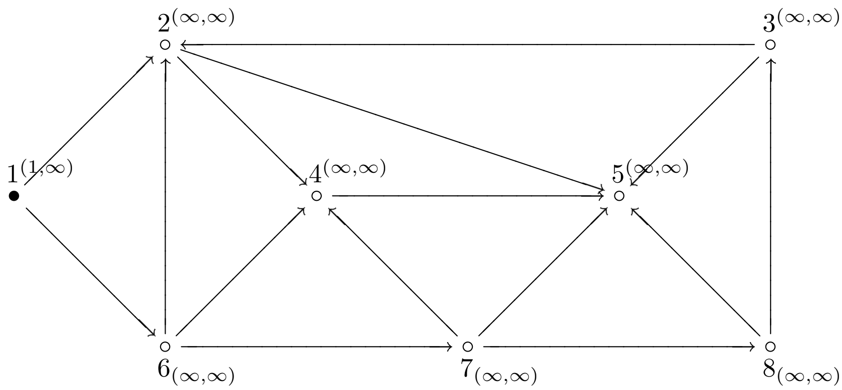


Wir beginnen die DFS. Der Stack S ist am Anfang leer, die Zeiten mit ∞ initialisiert und keine Kante ist als Baumkante durch einen π-Zeiger ausgezeichnet; alle Knoten sind weiß.



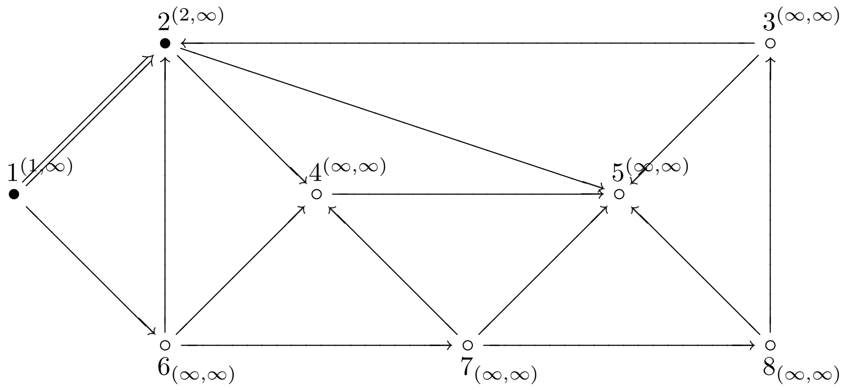
S = []

Wir beginnen bei 1, färben grau und fügen in den Stack ein:



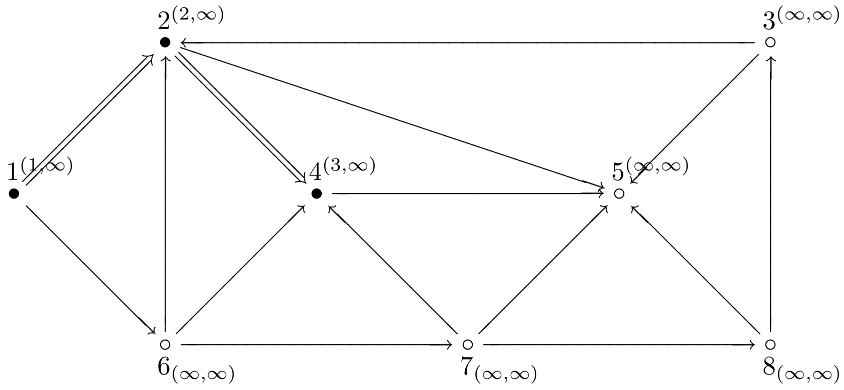
S = [1]

Nun gehen wir über zu 2:



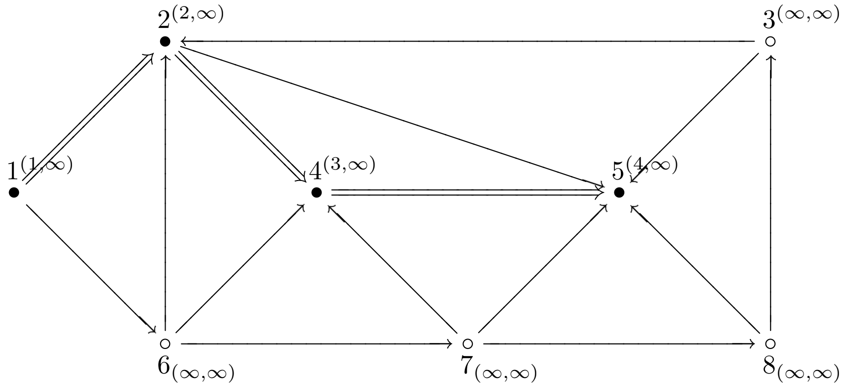
S = [2,1]

Nun 4:



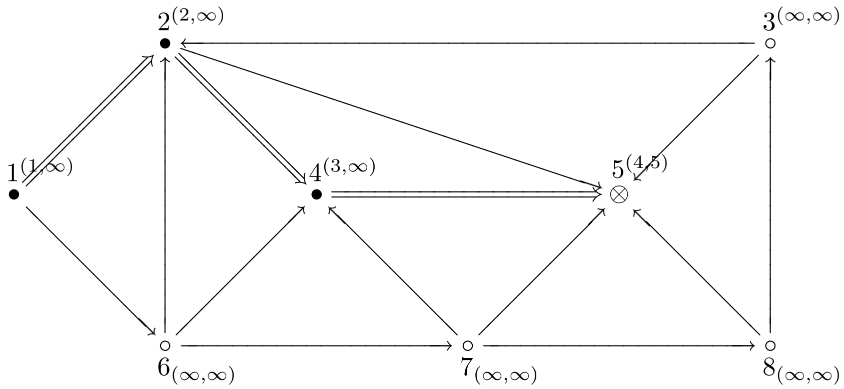
S = [4,2,1]

Nun 5:



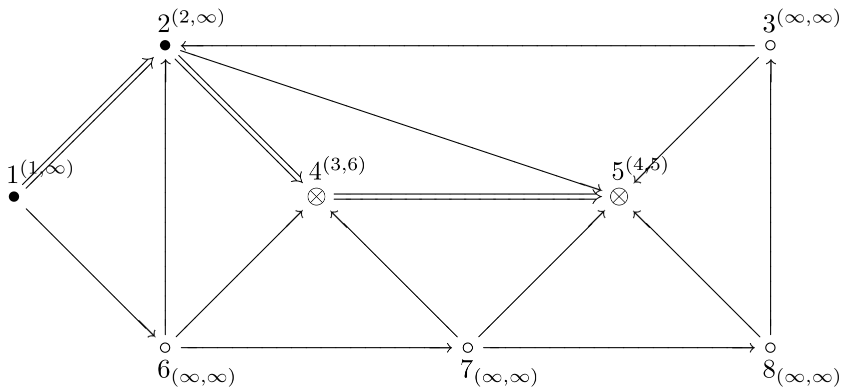
S = [5,4,2,1]

Von 5 gehen keine Verbindungen weg, also zurück:



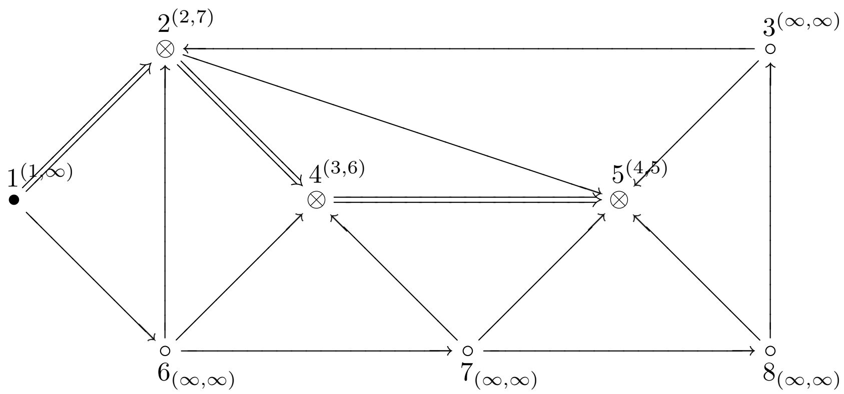
S = [4,2,1]

Von 4 auch nicht:



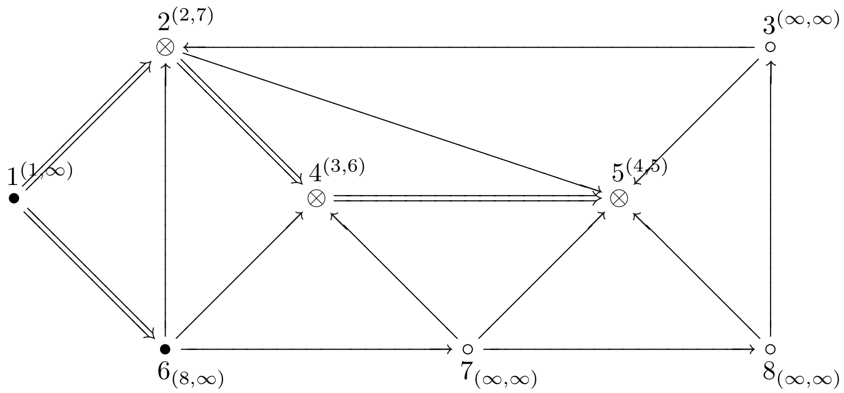
S = [2,1]

Bei 2 kommt man zu 5; da dieser aber bereits schwarz ist, und 2 keine weiteren Kanten hat, wird auch 2 aus dem Stack entfernt und geschwärzt:



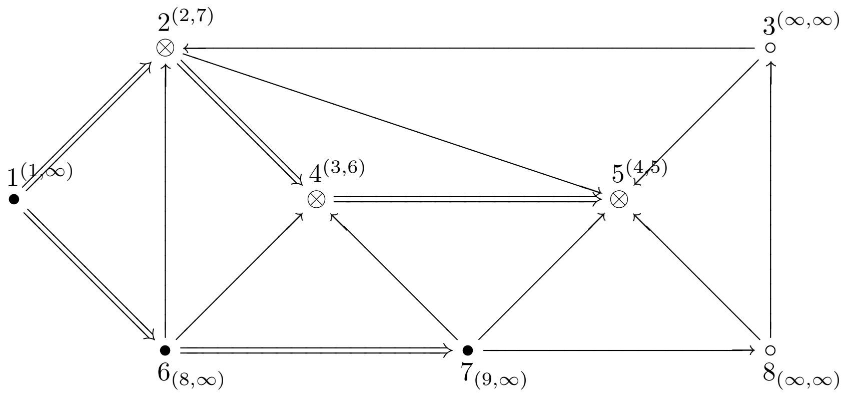
S = [1]

1 hat Nachbar 6:



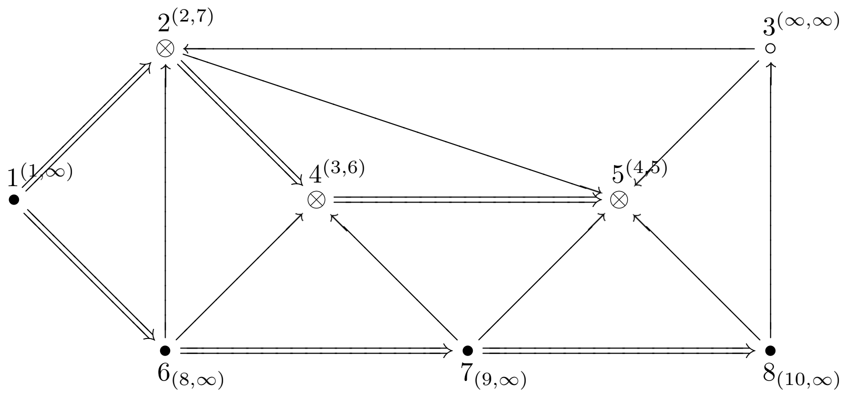
S = [6,1]

Von 6 kommt man zu 2 (bereits schwarz), zu 4 (bereits schwarz) und zu 7:



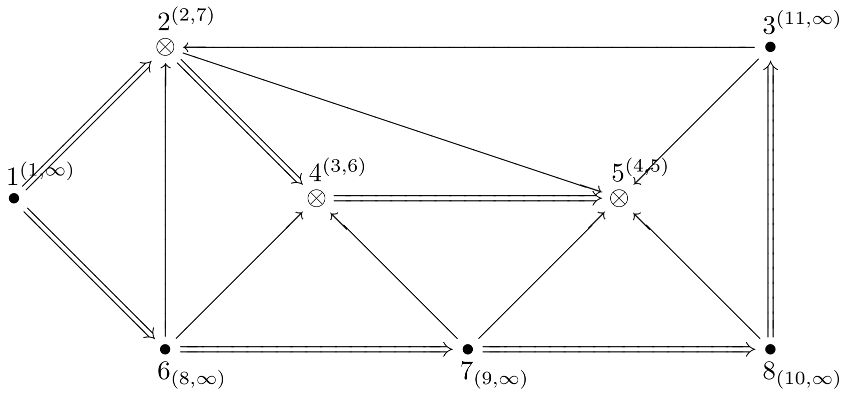
S = [7,6,1]

Von hier geht es weiter zu 4 (schwarz), 5 (schwarz) und 8:



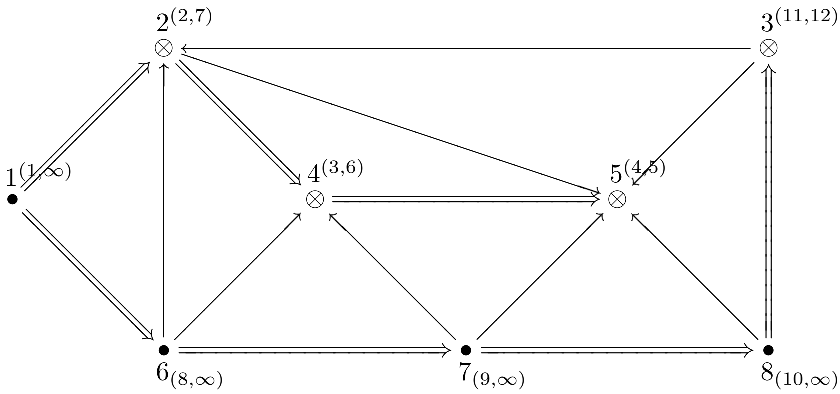
S = [8,7,6,1]

8 hat Nachbarn 5 (bereits schwarz) und 3:



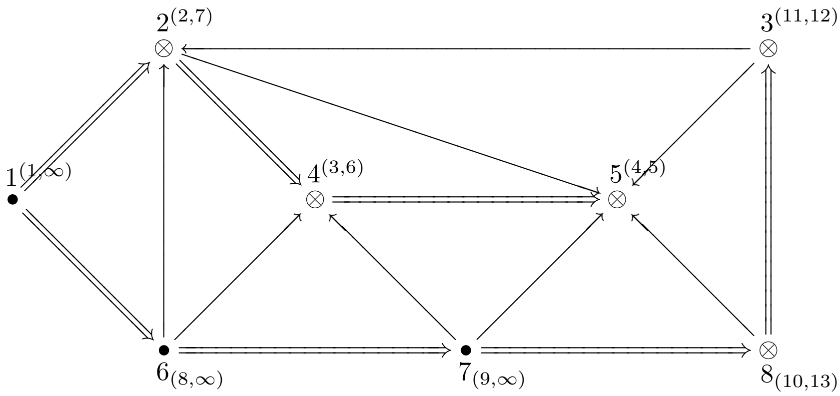
S = [3,8,7,6,1]

Alle Nachbarn von 3 sind schwarz, also zurück:



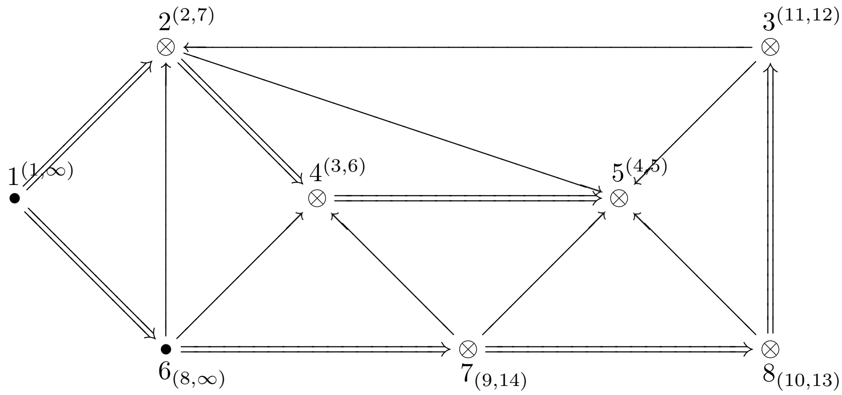
S = [8,7,6,1]

Genauso 8:



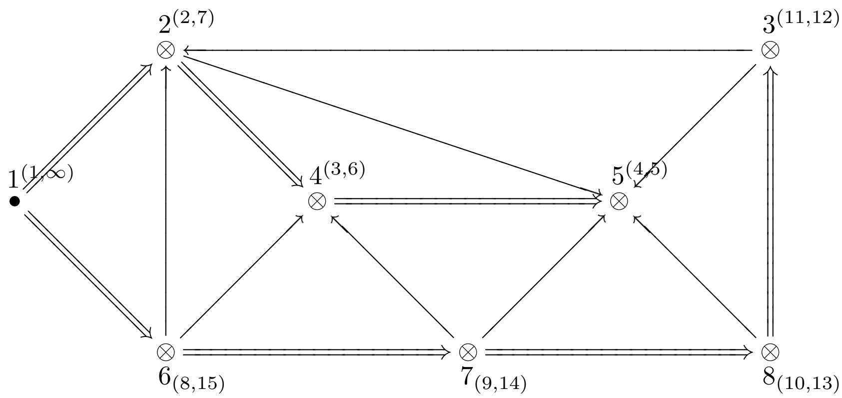
S = [7,6,1]

Und 7:



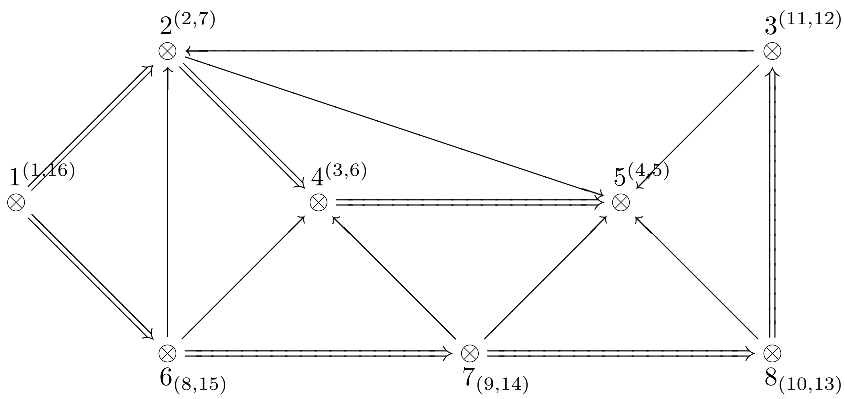
S = [6,1]

Und 6:



S = [1]

1 ist der letzte Knoten; alle Nachbarn sind schwarz. Also sind wir fertig.



S = []

Wir sortieren nach den Endzeiten und erhalten:

1 ¡ 6 ¡ 7 ¡ 8 ¡ 3 ¡ 2 ¡ 4 ¡ 5

4. **zweitschwerster Baum**

Beschreiben Sie in Pseudocode einen Algorithmus, der einen zweitschwersten aufspannenden Baum in einem ungerichteten kantengewichteten zusammenhängenden Graphen findet. Welche Datenstrukturen verwenden Sie? Illustieren Sie Ihren Algorithmus an einem sinnvoll gewählten Beispiel. Dieses Beispiel sollte etwa zeigen, dass es nicht immer richtig ist, die schwerste nicht im maximalen aufspannenden Baum enthaltene Graphkante gegen eine Baumkante einzutauschen.

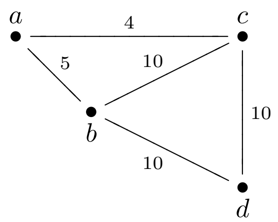
Die Idee ist, im maximalen aufspannenden Baum eine neue Kante so einzutauschen, dass die Differenz der neuen zu der alten Kante möglichst gering ist. Dazu geht mann nacheinander alle Baumkanten des (z.B. mit Prim bestimmten) schwersten aufspannenden Baumes durch und untersucht gegen welche nicht vorkommenden Kanten man sie eintauschen könnte und welche Gewichtedifferenz dabei zu erwarten ist. Die Kante, die dabei die kleinste Differenz zugeordnet bekommt, wird dann gegen die entsprechende andere Kante eingetauscht und man erhält so einen zweitschwersten aufspannenden Baum:

Nun kann man leicht einen Algorithmus angeben, der einen zweitschwersten aufspannenden Baum findet, gegeben sei $G = (V, E)$ durch seine Adjazenzlistendarstellung, verwendet wird der Baum-Prüf-Algorithmus aus der 1.Übung, Aufg. 2(a):

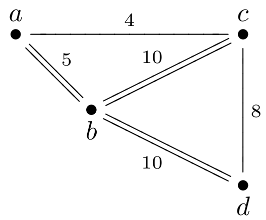
```
Zweiter_ST

Finde (z.B. mit Prim) einen MST T = (V, E) und seine Adjazenzlistendarstellung
Min_alt_Kante ← (nil , nil)
Min_neu_Kante ← (nil , nil)
min_d ← ∞
Für jeden Knoten a ∈ V
    Für jeden Knoten b ∈ Adj_T(a) // Jeden Baumnachbarn
        Wenn b > a // Jede Kante wird nur einmal geprüft
            Für jeden Knoten c ∈ V
                Für jeden Knoten d ∉ Adj_T(c)
                    Wenn d > c
                        Wenn Ist_Baum(T − {a,b} + {c,d})
                                Und (w({a, b}) − w({c, d})) > min_d
                            min_d ← w({a, b}) − w({c, d})
                            Min_alt_Kante ← {a,b}
                            Min_neu_Kante ← {c,d}
Wenn min_d ≠ ∞
    der zweitschwerste aufspannende Baum ist gleich dem MST T und
    G = T ist ein Baum → gib T zurück
Sonst
    ein zweitschwerster aufspannender Baum ist T − {a,b} + {c,d}
```

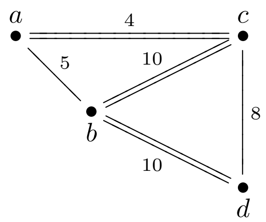
Nun zum Beispiel, betrachte etwa den Graphen



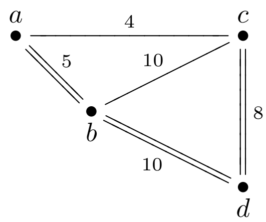
Führt man den Algorithmus von Prim vom Startknoten a aus aus, so erhält man etwa den MST



Durchsucht man nun den Baum & den Graphen entsprechend obigen Algorithmus, so erhält man: Lässt man die Kante a–b weg, so muss man die Kante a–c aufnehmen, die zugehörige Differenz ist 1, ließe man b–c weg, so müsste man c–d aufnehmen oder a–c aufnehmen, die auftretenden Differenzen sind 6 resp. 2, also größer, ließe man b–d weg, so müsste man c–d aufnehmen, auch diese Differenz ist mit 2 größer als 1. Der zweitschwerste aufspannende Baum ist also:



mit dem Kantengewicht 24, nähme man die schwerste nicht im MST auftretende Kante, i.e. c–d auf, so müsste man eine Kante mit Gewicht 10, also etwa b–c entfernen, der entstehende Baum



hat nur Kantengewicht 23, ist also kein zweitschwerster Baum.