TECHNISCHE UNIVERSITÄT BERLIN
Fakultät IV – Elektrotechnik und Informatik
Fachgebiet Neurotechnologie (MAR 4-3)
Prof. Dr. Benjamin Blankertz
Röhr / Stahl

Algorithmen und Datenstrukturen, SoSe 20

Bitte füllen Sie alle folgenden Felder aus:

Vorname: _____

Nachname: _____

tubIT-login: _____

Matrikelnummer: _____

Durch meine Unterschrift bestätige ich die Korrektheit obiger Angaben sowie meine Prüfungsfähigkeit und die Anmeldung zur Prüfung!

_____ _____

Ort, Datum Unterschrift

Beachten Sie die folgenden Hinweise!

- Sie brauchen Ihren Namen **nur** auf das Deckblatt zu schreiben. Die restliche Blätter können über die Klausur-ID zugeordnet werden.
- Diese Klausur besteht mit diesem Deckblatt aus den (nummerierten) Seiten **1-16**.
- Am Ende der Klausur befinden sich zwei leere Seiten, die Sie für Notzien verwenden können. Sollten Sie mehr Papier benötigen, können Sie dies von der Aufsicht bekommen. Notieren Sie in diesem Fall die Klausur-ID auf dem Zusatzblatt.
- Notieren Sie Ihre Antworten nur auf dem Blatt (inklusive Rückseite), auf dem die zugehörige Aufgabe steht, da die Aufgaben getrennt korrigiert werden.
- Falls Sie eine Antwort auf ein Zusatzblatt schreiben, markieren Sie dies klar bei der zugehörigen Aufgabe und auf dem Zusatzblatt.
- Geben Sie nur eine Lösung pro Aufgabe ab, streichen Sie alle alternativen Lösungsansätze auf Schmier-/Notitzblättern durch.
- Schreiben Sie **nicht** mit roter Farbe, grüner Farbe (Korrekturfarben) oder Bleistift. Diese Lösungen werden nicht bewertet!
- Insgesamt können in der Klausur **100 Punkte** erreicht werden.

Zusatzblätter:

Punktetabelle

Aufgabe	Punkte		
1			
2			
3			
4			
5			
6			
7			
8			
9			

Aufgabe 1: Vermischtes ($4 + 3 + 4 = 11$ Punkte)

(a) Geben Sie die Wachstumsordnung der Laufzeiten der Methoden `f1(int N)` und `f2(int N)` an (ohne Begründung). (Falls Sie die $\boldsymbol{O}$ Notation an Stelle der Θ Notation verwenden, wählen Sie die kleinst mögliche Wachstumsordnung.) Der Term in der $\boldsymbol{\Theta}$ bzw. $\boldsymbol{O}$ Notation soll möglichst einfach sein.

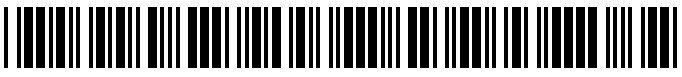
```java
public static int f1(int N) {
    int x = 0;
    for (int i = 0; i < N; i += 2) {
        for (int j = 0; j < Math.min(N, 100); j++) {
            x += i * j;
        }
    }
    return x;
}
```

```java
public static int f2(int N) {
    int x = 0;
    for (int i = 1; i < N; i *= 2) {
        for (int j = 0; j < N*N; j += N) {
            x += j / i;
        }
    }
    return x;
}
```

Wachstumsordnungen:
`f1(N):` , `f2(N):`

(b) Ein Knoten v in einem gerichteten Graphen $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ heißt **römischer Knoten**, wenn es von allen anderen Knoten von $\boldsymbol{G}$ einen gerichteten Pfad nach v gibt.

Beschreiben Sie ein Verfahren mit Laufzeit in $\boldsymbol{O}(V + E)$, um festzustellen, ob ein gegebener Knoten v ein römischer Knoten von $\boldsymbol{G}$ ist. Begründen Sie dabei die Laufzeit.



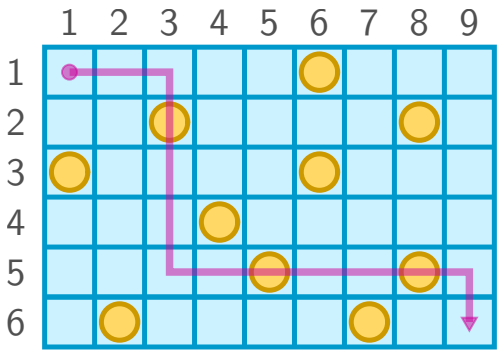
(c) Ein ungerichteter Graph wird k**-fach zusammenhängend** genannt, falls er zusammenhängend ist und es auch dann bleibt, wenn eine beliebige Menge von $k-1$ vielen Knoten aus ihm gelöscht wird. Ein Knoten v eines Graphen $\boldsymbol{G}$ heißt **Artikulationspunkt**, wenn das Löschen von v die Anzahl der Zusammenhangskomponenten von $\boldsymbol{G}$ erhöht.
Zeichnen Sie jeweils ein Beispiel für einen ungerichteten Graphen mit folgenden Eigenschaften:

(1) Der Graph ist zusammenhängend und hat zwei Artikulationspunkte. Markieren Sie alle Artikulationspunkte.

(2) Der Graph hat maximal 6 Knoten und ist 3-fach, aber nicht 4-fach zusammenhängend. Markieren Sie drei Knoten, deren Löschung den Graphen unzusammenhängend macht. Die Menge der Knoten ist dabei nicht unbedingt eindeutig.

Aufgabe 2: Dynamisches Programmieren (1 + 5 + 2 = 8 Punkte)

Auf einem Brett der Größe $N \times M$ liegen auf manchen Feldern Münzen. Ein Roboter startet in der linken oberen Ecke und kann sich in jedem Schritt ein Feld nach unten oder ein Feld nach rechts bewegen. Auf seinem Weg sammelt der Robot alle Münzen ein. Es soll ein Weg zur rechten unteren Ecke gefunden werden, bei dem die maximale Anzahl von Münzen eingesammelt wird. Die Funktion $F(n, m)$ ist für $1 \leq n \leq N$ und $1 \leq m \leq M$ als 1 definiert, wenn auf dem Feld (n, m) eine Münze liegt und als 0 falls das Feld leer ist.



Eine effiziente Lösung dieser Optimierungsaufgabe kann durch Dynamische Programmierung erreicht werden. Als Grundlage dient eine Funktion $\textsc{Opt}(n, m)$, die angibt, wieviele Münzen von Feld $(1, 1)$ bis Feld (n, m) maximal eingesammelt werden können ($n \leq N$ und $m \leq M$).

(a) Die optimale Lösung für ein Zielfeld (n, m), also der Wert $\textsc{Opt}(n, m)$, kann leicht angegeben, wenn die optimale Lösung für zwei andere Zielfelder bekannt ist. Welche zwei Zielfelder sind dies? (Dabei wird angenommen, dass $n > 1$ und $m > 1$ gilt.)

(b) Geben Sie eine rekursive Definition der Opt Funktion an (mathematische Schreibweise, kein Java- oder Pseudocode).

(c) Eine rekursive Implementierung der Opt Funktion wäre nicht effizient. Wie geht der Bottom-Up Ansatz der Dynamischen Programmierung bei gegebener Opt Funktion vor? Wodurch wird der rekursive Aufruf ersetzt? (ca. zwei Sätze zum allgemeinen Vorgehen, nicht konkret auf das Beispiel in dieser Aufgabe bezogen)

Aufgabe 3: Java - Zyklen ($5 + 5 + 2 = 12$ Punkte)

Im Folgenden ist die Klasse `DirectedCycle` vorgegeben, die eine Tiefensuche implementiert, um damit Zyklen zu finden. Der Vorgabecode enthält allerdings keinen Konstruktor.

```
1  public class DirectedCycle {
2      private boolean[] marked;     // marked[v] = has vertex v been marked?
3      private int[] edgeTo;         // edgeTo[v] = previous vertex on path to v
4      private boolean[] onStack;    // onStack[v] = is vertex on the stack?
5      private Stack<Integer> cycle;// directed cycle (or null if no such cycle
           )
6  |
7  |
8  |
9  |
10 |
11 |
12 |
13 |
14 |
15 |
16 // Algorithmus berechnet entweder die Topologische Sortierung oder findet
       einen gerichteten Zyklus
17     private void dfs(Digraph G, int v) {
18         onStack[v] = true;
19         marked[v] = true;
20         for (int w : G.adj(v)) {
21             if (cycle != null) return;
22             else if (!marked[w]) {
23                 edgeTo[w] = v;
24                 dfs(G, w);
25             }
26             else if (onStack[w]) {
27                 cycle = new Stack<Integer>();
28                 for (int x = v; x != w; x = edgeTo[x]) {
29                     cycle.push(x);
30                 }
31                 cycle.push(w);
32                 cycle.push(v);
33             }
34         }
35         onStack[v] = false;
36     }
37 //Gibt true, wenn ein Zyklus existiert, andernfalls false.
38     public boolean hasCycle() {
39         return cycle != null;
40     }
41     public Iterable<Integer> cycle() {
42         return cycle;
43     }
44 }
```

(a) Schreiben Sie einen Konstruktor der Klasse `DirectedCycle`, dem ein `Digraph G` als Argument übergeben wird. Initialisieren Sie alle Klassenvariablen. Mit dem Erstellen eines Objektes `DirectedCycle` soll außerdem der Graph direkt auf Zyklen überprüft werden. Wenn Sie den Konstruktor ergänzt haben, sollte die Klasse fehlerfrei auch auf **unzusammenhängenden gerichteten** Graphen durchlaufen.
Hinweis: Für die Anzahl der Knoten im Graphen können Sie die Notation aus den Hausaufgaben verwenden: `G.V()`.

(b) Betrachten Sie folgende `main()`-Methode, in der ein gerichteter Graph mit 6 Knoten initialisiert und dann darauf die obige Zyklenerkennung ausgeführt wird:

```
1   public static void main(String[] args) {
2       Digraph G = new Digraph(6);
3       G.addEdge(0, 1);
4       G.addEdge(1, 2);
5       G.addEdge(2, 4);
6       G.addEdge(3, 5);
7       G.addEdge(4, 1);
8       G.addEdge(4, 0);
9       G.addEdge(5, 3);
10
11      DirectedCycle finder = new DirectedCycle(G);
12      if (finder.hasCycle()) {
13          System.out.print("Directed cycle: ");
14          for (int v: finder.cycle()) {
15              System.out.print(v + " ");
16          }
17          System.out.println();
18      }
19      else {
20          System.out.println("No directed cycle");
21      }
22  }
```

Skizzieren Sie den Graphen, der von der `main()` Methode erzeugt wird. Wenn die `main()` Methode ausgeführt wird, was ist dann die Ausgabe dieser `main()` Methode?
Hinweise: Beachten Sie dabei, dass `edgeTo[v]` den letzten Knoten auf dem Pfad zu `v` speichert. Rechts finden Sie den Code Abschnitt zu der Zyklenerkennung nochmal in kleiner, damit Sie nicht blättern müssen.

```
17  private void dfs(Digraph G, int v) {
18        onStack[v] = true;
19        marked[v] = true;
20        for (int w : G.adj(v)) {
21            if (cycle != null) return;
22            else if (!marked[w]) {
23               edgeTo[w] = v;
24               dfs(G, w);
25            }
26            else if (onStack[w]) {
27                cycle = new Stack<Integer>();
28                for (int x = v; x != w; x =
                      edgeTo[x]) {
29                    cycle.push(x);
30                }
31                cycle.push(w);
32                cycle.push(v);
33            }
34        }
35        onStack[v] = false;
36    }
```

(c) Nehmen Sie an, Sie wollten eine Testmethode schreiben, die überprüft, ob `cycle()` auch tatsächlich einen Kreis zurückgibt. Nennen Sie **eine** der Bedingungen, die man dafür überprüfen müsste.

Aufgabe 4: Hashing ($1,5 + 4,5 + 1 + 1 = 8$ Punkte)

Für diese Aufgabe wird

$$h(x) = (2 \cdot x) \; mod \; 7$$

als Hashfunktion verwendet.

(a) Berechnen und notieren Sie die restlichen Hashadressen in der folgenden Tabelle.

Name des Schlüssels	Hashcode x	Hashadresse
A	11	1
B	7	0
C	13	5
D	35	0
E	24	6
F	22	2
G	39	1
H	18	
I	45	

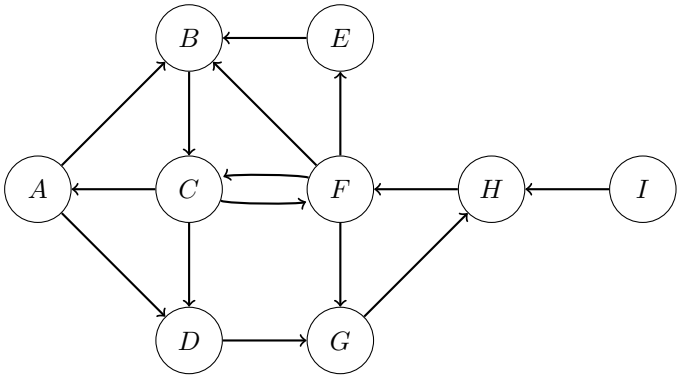
(b) Fügen Sie die ersten neun Schlüssel (A bis I) aus a) in alphabetischer Reihenfolge in die Hashtabelle ein. Verwenden Sie dabei Kollisionsauflösung durch Verkettung (seperate chaining). Die Listen verlaufen dabei horizontal von links nach rechts.

0						
1						
2						
3						
4						
5						
6						

(c) Gehen Sie davon aus, dass die Hashtabelle mit anderen Schlüsseln gefüllt worden ist. Unter welchen Bedingungen hat das Finden (Suchen) eines Schlüssels eine besonders lange Laufzeit?

(d) Nennen Sie einen Vorteil von Hashing mit verketteten Listen (seperate chaining) gegenüber Hashing mit linearer Sondierung.

Aufgabe 5: Breitensuche ($4 + 6 + 4 = 14$ Punkte)



(a) Geben Sie für die Knoten $v =$ **A**, **C**, **E** und **I** vom Graphen G einen Zyklus minimaler Länge an, der den Knoten v enthält, wenn er auf einem Zyklus liegt. Andernfalls geben Sie an, dass v nicht auf einem Zyklus liegt.

A

C

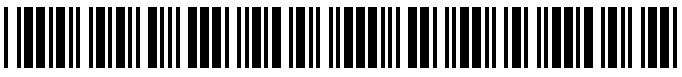
E

I

(b) Führen Sie die Breitensuche auf dem Graphen G aus. Fangen Sie bei Knoten I an und notieren Sie alle Knoten in der Reihenfolge, in der sie von der Breitensuche in die Warteschlange geschrieben werden. Gehen Sie dabei davon aus, dass bei Wahlmöglichkeit die Nachbarn in **alphabetischer Reihenfolge** besucht werden.

I

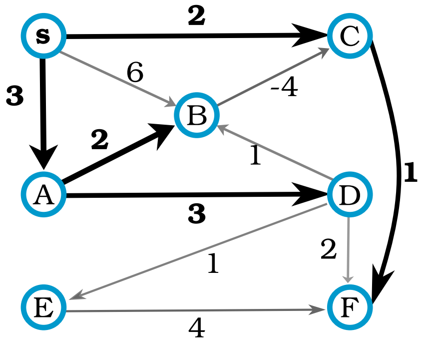
(c) Erklären Sie, wie man die Breitensuche nutzen kann, um festzustellen, ob **ein Knoten v** im gerichteten Graphen G auf einem Zyklus liegt. Wenn mithilfe der Breitensuche ein Zyklus der Länge ≥ 2 gefunden wird, hat dieser Zyklus dann minimale Länge? Begründen Sie.



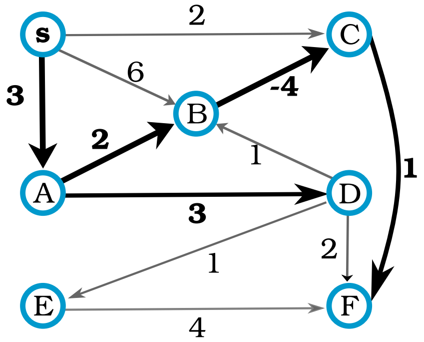
Aufgabe 6: SSSP $(12 + 3 + 1 = 16$ Punkte)

In der folgenden Abbildung sehen Sie vier mal den gleichen Graphen, wobei jedes Mal unterschiedliche Kanten markiert sind. Diese Kanten markieren Zwischenlösungen für den Baum der kürzesten Wege, die entweder nur vom Dijkstra Algorithmus mit indizierter Warteschange, nur vom Bellman-Ford Algorithmus mit Warteschlange, durch beide oder durch keinen von beiden gefunden werden. Die Algorithmen starten immer im markierten Startknoten s. Bei gleichem Abstand werden Knoten alphabetisch abgearbeitet.

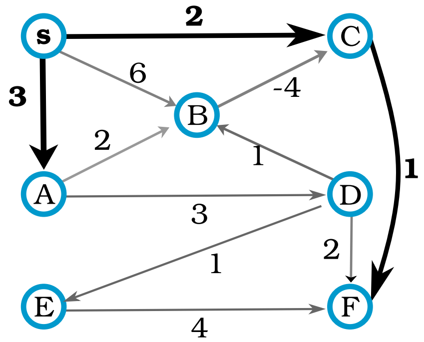
(a) Kreuzen Sie jeweils an, ob die Zwischenlösung nur durch den Dijkstra Algorithmus oder nur den durch Bellman-Ford Algorithmus, wie sie in der VL behandelt wurden, durch beide oder durch keinen von beiden entstanden sein kann.
Hinweis: Führen Sie kurz einen Teil der Handsimulationen für beide Algorithmen auf dem Graphen durch.



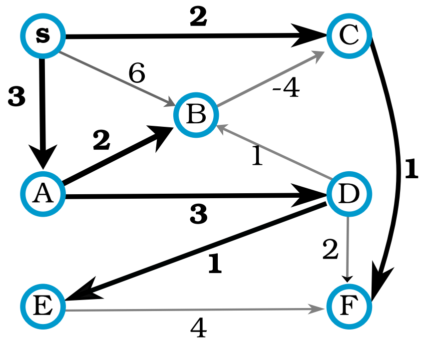
○ nur Dijkstra ○ beide
○ nur Bellman-Ford ○ keiner



○ nur Dijkstra ○ beide
○ nur Bellman-Ford ○ keiner

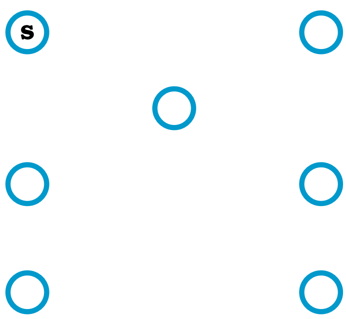


○ nur Dijkstra ○ beide
○ nur Bellman-Ford ○ keiner

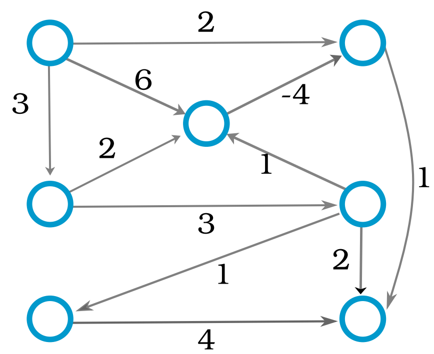


○ nur Dijkstra ○ beide
○ nur Bellman-Ford ○ keiner

(b) Zeichnen Sie den vollständigen Baum der kürzesten Wege des obigen Graphen.

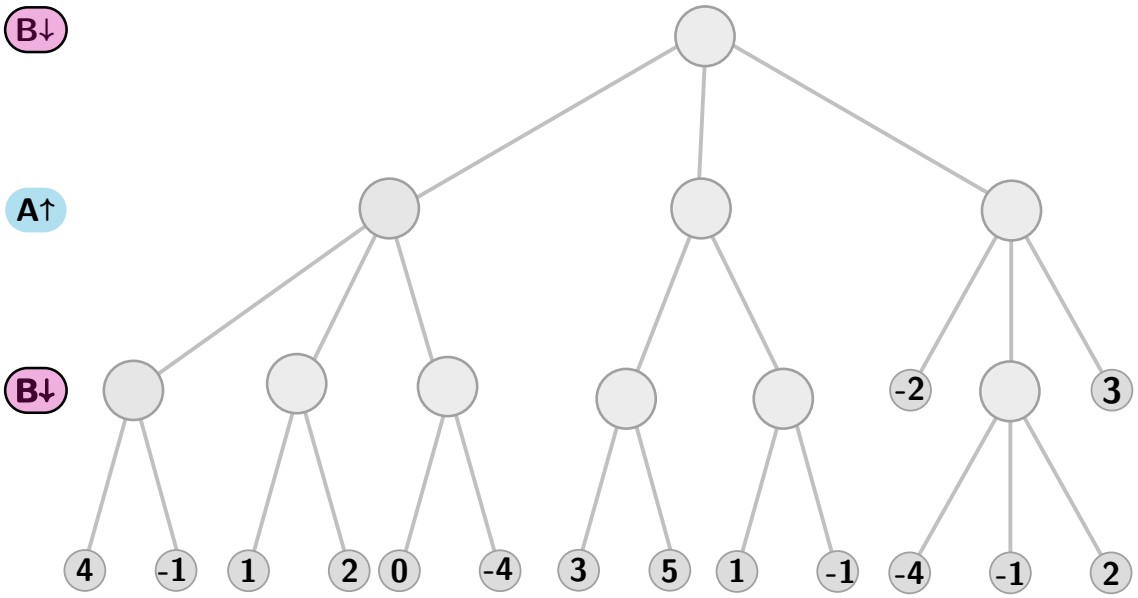


(c) Fügen Sie eine Kante in den Graphen ein, sodass keiner der beiden Algorithmen weiterhin den kürzesten Weg finden würde.



Aufgabe 7: Minimax- und Alpha-Beta-Algorithmus

(4 + 4 = 8 Punkte)

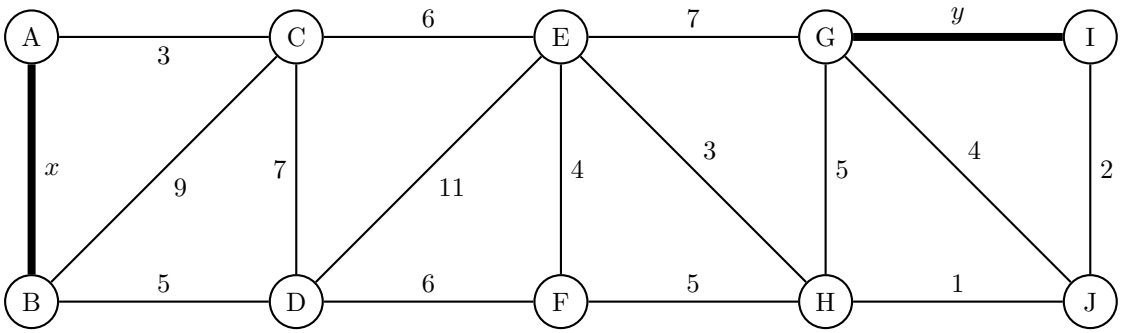


(a) Vervollständigen Sie den obigen **Minimax** Suchbaum.

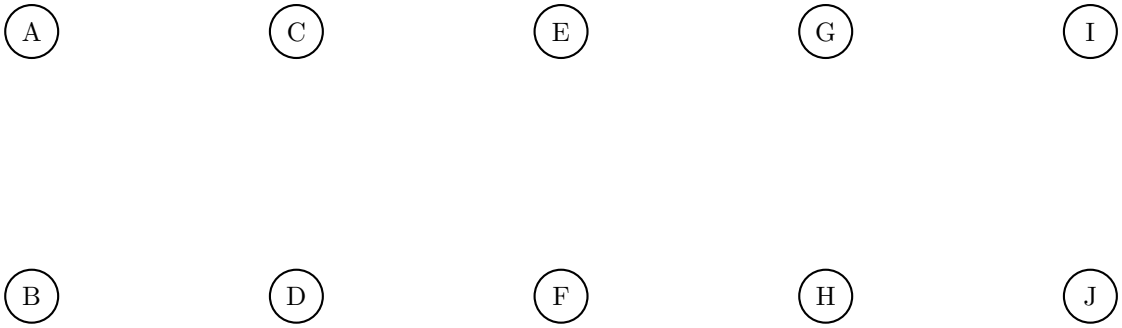
(b) Nehmen Sie an, Sie würden auf dem obigen Suchbaum eine Alpha-Beta-Suche ausführen, die von links nach rechts läuft. Welche Zweige würden nicht besucht? Tragen Sie α- und β- Cutoffs in den Baum ein. Sie brauchen nicht zu kennzeichnen, welcher Cut ein α oder ein β-Cutoff ist.

Aufgabe 8: Minimaler Spannbaum ($7 + 4 = 11$ Punkte)

Nehmen Sie an, der *Minimum Spanning Tree* (MST) dieses Graphen **enthalte** die Kanten (A,B) mit dem unbekannten Gewicht x und (G,I) mit dem unbekannten Gewicht y.



(a) Zeichnen Sie die Kanten, die unter der obigen Annahme zum MST dieses Graphen gehören müssen, ein.



(b) Geben Sie jeweils die größte obere Schranke für die Gewichte x und y der Kanten (E,G) und (G,I) an, mit der garantiert ist, dass beide Kanten tatsächlich Teil des MST sind.
Hinweis: Geben Sie individuell für jedes Gewicht x, y eine Schranke in Form einer Ungleichung an.

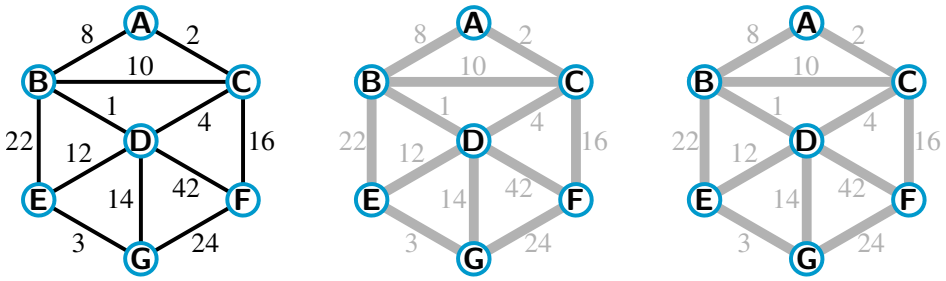
Aufgabe 9: Der Borůvka Algorithmus für minimale Spannbäume ($3+4+2+3=12$ **Punkte**)

Sei $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E}, \mathit{weight})$ ein ungerichteter, gewichteter Graph, der zusammenhängend ist. Zur Vereinfachung nehmen wir an, dass alle Gewichte unterschiedlich sind. Der Algorithmus von Borůvka zur Erzeugung des minimalen Spannbaums geht wie folgt vor:

> Beginne mit dem Wald $\boldsymbol{F} = (\boldsymbol{V}, \varnothing)$, der alle Knoten von $\boldsymbol{G}$ umfasst, jedoch keine Kanten besitzt. In einer Iteration wird für **jede** Zusammenhangskomponente von $\boldsymbol{F}$ die minimal kreuzende Kante zu $\boldsymbol{F}$ hinzugefügt. Es wird so lange iteriert, bis $\boldsymbol{F}$ nur noch aus einer Zusammenhangskomponente besteht.

(a) Begründen Sie in ca. drei Sätzen, warum $\boldsymbol{F}$ am Ende des oben beschriebenen Algorithmus ein MST von $\boldsymbol{G}$ ist.

(b) Führen Sie zwei Iterationen des Algorithmus auf dem in (a) abgebildeten Graphen aus. Markieren Sie die Kanten, die in der ersten Iteration ausgewählt werden in (b) und die aus der zweiten Iteration in (c).



(a) (b) (c)

(c) Jede Iteration des Borůvka Algorithmus benötigt eine Laufzeit in $\boldsymbol{O}(E)$. Geben Sie eine möglichst niedrige (also genaue) Wachstumsordnung für die Gesamtlaufzeit des Algorithmus an und begründen Sie diese.

(d) Vervollständigen Sie den Pseudocode in den Zeilen 8, 13, 15 und 18 so, dass er dem oben beschriebenen Borůvka Algorithmus entspricht.
Die wichtigsten Variablen im folgenden Codeabschnitt sind folgende: **`count`** ist die **Anzahl der Zusammenhangskomponenten**, auch abgekürzt mit Zhk. **`id`** speichert den **Index** der jeweiligen Zusammenhangskomponenten für jeden Knoten und **`mkk`** enthält die **minimal kreuzende Kante** für jede Zusammenhangskomponente.
Um das Gewicht einer Kante `e` zu beschreiben, können Sie die Notation **`weight(e)`** verwenden.

Hinweis: Die Leerzeilen 7, 12, 14, 17 dienen nur der Vergößerung des Zeilenabstandes und brauchen nicht gefüllt zu werden.

```
1  procedure boruvka(G)
2  F ←(V, ∅)              // Wald bestehend aus Knoten von G und ohne Kanten
3  count ← 0              // count: Anzahl der Zusammenhangskomponenten (Zhk)
4  for all v ∈ V
5      count ← count + 1
6      id[v] ← count      // id: Index der Zhk
7
8  while  ______________________________
9      for c ← 1 to count
10         mkk[c] ← null   // mkk[c]: minimal kreuzende Kante der Zhk mit Index c
11     for all (u,v) ∈ E
12
13         if  ____________________  then
14
15             if  mkk[id[u]]=null or  ________________________________________  then
16                 mkk[id[u]] ← (u,v)
17
18             if  mkk[id[v]]=null or  ________________________________________  then
19                 mkk[id[v]] ← (u,v)
20     for c ← 1 to count
21         füge Kante mkk[c] zu F
22      identifiziere  Zhk von F mit DFS –> id wird aktualisiert; count= count−1
23 return F            // F ist der MST von G
```


Diese Seite können Sie für Notizen verwenden. Bitte nur im Ausnahmefall für Lösungen verwenden!

Diese Seite können Sie für Notizen verwenden. Bitte nur im Ausnahmefall für Lösungen verwenden!