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

Algorithmen und Datenstrukturen, SoSe 23

Bitte füllen Sie alle folgenden Felder aus:

Vorname: _____

Nachname: _____

tubIT-Benutzername: _____

Matrikelnummer: _____

Studiengang: _____

Hochschule: _____

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

_____ _____

Ort, Datum Unterschrift

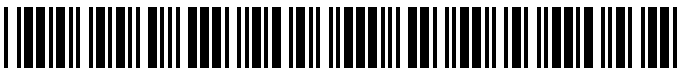
Beachten Sie die folgenden Hinweise!

- Sie brauchen Ihren Namen **nur** auf das Deckblatt zu schreiben. Die restlichen Blätter können über die Klausur-ID zugeordnet werden.
- Diese Klausur besteht mit diesem Deckblatt aus den (nummerierten) Seiten **1-11**.
- Am Ende der Klausur befinden sich zwei leere Seiten, die Sie für Notizen 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-/Notizblättern 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:

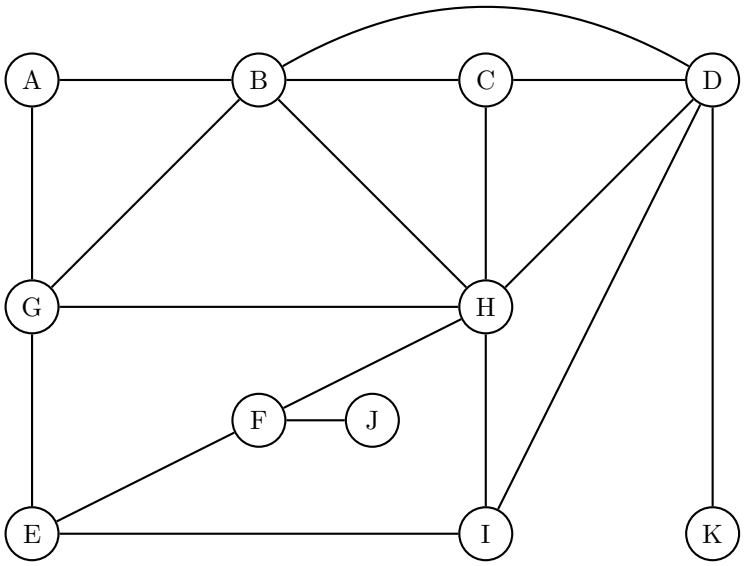
Punktetabelle

Aufgabe	Punkte	Vorkorr.	Kontrolle
1	/ 18		
2	/ 22		
3	/ 18		
4	/ 16		
5	/ 13		
6	/ 13		
$\sum$	/ 100		



Aufgabe 1: Breitensuche ($10 + 8 = 18$ Punkte)

Gegeben ist folgender Graph G:

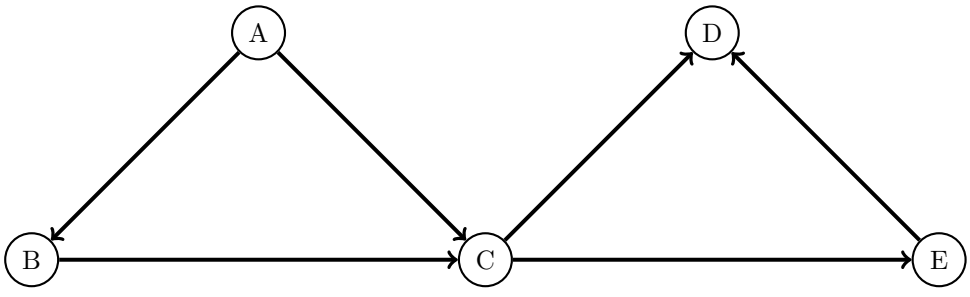


- A
A B G
B G C D
G C D E H
C D E H
D E H I K
E H I K F
H I K F
I K F
K F
F J
J ∅

(a) Führen Sie die Breitensuche auf dem obigen Graphen G aus. Fangen Sie bei **Knoten A** 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 von jedem Knoten die benachbarten Knoten in alphabetischer Reihenfolge besucht werden. Z.B. wird die Kante **B**-**C** vom Algorithmus vor der Kante **B**-**H** bearbeitet.

A B G C D E H I K F J

(b) In dem folgenden *gerichteten* Graphen ist nicht bekannt, in welcher Reihenfolge die Knoten in den Adjazenzlisten stehen. Es gilt hier also **nicht** notwendigerweise die alphabetische Ordnung bei dem Besuchen der Nachbarknoten. Geben Sie alle unterschiedlichen Knotenreihenfolgen an, in der die Breitensuche die Knoten **in die Warteschlange** schreiben könnte. Der Startknoten ist **A**.



A B C D E
A B C E D
A C B D E
A C B E D

Aufgabe 2: Hashing (12 + 10 = 22 Punkte)

Es wird eine Hashtabelle der Größe 10 zur Speicherung von Integer-Werten betrachtet, d.h., die Schlüssel sind vom Typ Integer. Die Hashadresse eines Integer k wird durch

$$hash(k) = k \mod 10$$

berechnet.

(a) Fügen Sie die folgenden Schlüssel in der gegebenen Reihenfolge nacheinander in die leere Hashtabelle, die unter der Reihe von Schlüsseln abgedruckt ist. Tragen Sie hierfür jeweils den Schlüssel (Key), also den Integer-Wert, in der leeren Tabelle an dem entsprechenden Index ein. Kollisionen sollen durch Lineares Sondieren (Linear Probing) mit Inkrement 1 aufgelöst werden.

12, 45, 19, 2, 33, 89, 14, 10, 26

Index	0	1	2	3	4	5	6	7	8	9
Key	89	10	12	2	33	45	14	26		19

(b) Für diese Teilaufgabe ist die Hashtabelle mit anderen Schlüsseln gefüllt worden. Es soll zunächst der Schlüssel 22 und danach der Schlüssel 38 mit '*sofortigem Löschen*' (*eager deletion*) gelöscht werden. Tragen Sie den Zustand der Hashtabelle nach diesen beiden Löschoperationen in die untenstehende leere Tabelle ein.

Index	0	1	2	3	4	5	6	7	8	9
Key	28		42	~~22~~ 33	4	33	12		~~38~~ 28	39

Zustand nach dem Löschen der Schlüssel 22 **und** 38**:**

Index	0	1	2	3	4	5	6	7	8	9
Key			42	33	4	12			28	39

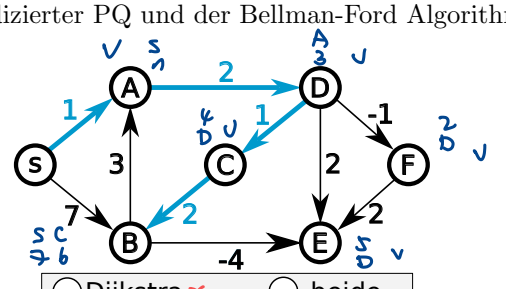
Aufgabe 3: SSSP (12 + 3 + 3 = 18 Punkte)

In der folgenden Abbildung sehen Sie vier mal den gleichen Graphen, wobei jedes mal unterschiedliche Kanten (**blau und breiter**) markiert sind.
Hinweis: Führen Sie beide Algorithmen einmal auf dem Graphen durch, bevor Sie die Aufgabe bearbeiten. Die Buchstabierung der Knoten ist nur für Ihre Handsimulation vorhanden, sie hat keine weitere Relevanz.
Sie finden den Graphen nochmal abgedruckt auf der 1. Notizseite, Seite 10, am Ende.

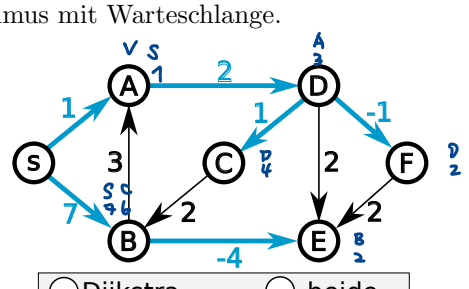
(a) Kreuzen Sie für jeden der vier Graphen an, ob die markierten Kanten nur durch den Dijkstra Algorithmus, nur durch den Bellman-Ford Algorithmus, durch beide oder durch keinen von beiden dem Baum der kürzesten Wege hinzugefügt worden sein können. Der Startknoten ist **s**.

Dabei muss der jeweilige Algorithmus nicht bis zum Ende durchgelaufen sein. Es können also auch die Markierungen **während** der Berechnung der kürzesten Wege abgebildet sein. Das heißt bei diesen beiden Algorithmen, dass auch Kanten markiert sein können, die **nicht final** im Baum der kürzesten Wege enthalten, sondern nur die bisher gefundenen kürzesten Wege sind.

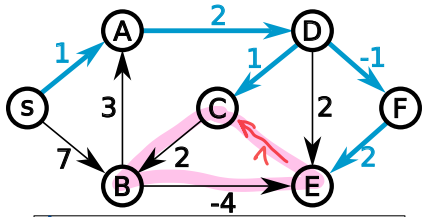
Es werden folgende Versionen der Algorithmen aus der Vorlesung verwendet: Der Dijkstra Algorithmus mit indizierter PQ und der Bellman-Ford Algorithmus mit Warteschlange.



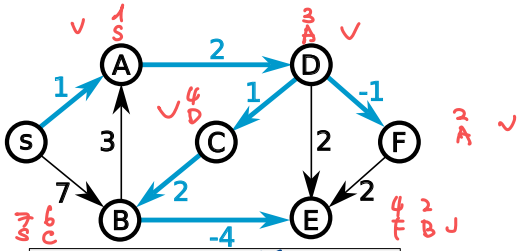
◯ Dijkstra ✗ ◯ beide
◯ Bellman-Ford ⊗ keiner



◯ Dijkstra ◯ beide
⊗ Bellman-Ford ◯ keiner



⊗ Dijkstra ◯ beide
◯ Bellman-Ford ◯ keiner



◯ Dijkstra ⊗ beide
◯ Bellman-Ford ◯ keiner

(b) Erläutern Sie anhand des Graphen aus a) warum Dijkstra für Graphen mit negativen Kantengewichten kein Greedy Verfahren mehr ist.

Der kürzeste Weg zu Knoten E wird zunächst als 4 festgestellt und später zu 2 korrigiert. Greedy Algorithmen nehmen keine Korrekturen vor.
Genauer als Erklärung: Knoten E wird der PQ entnommen (Gewicht 4 als kürzester Weg) und später noch einmal eingefügt und wieder entnommen (Gewicht 2: der kürzeste Weg dorthin wurde korrigiert).

(c) Fügen Sie eine neue Kante mit positivem Gewicht in den obigen Graphen ein, sodass keiner der beiden Algorithmen einen kürzesten Weg von **s** nach **D** finden kann. Geben Sie die Kante und ihr Gewicht an, und begründen Sie kurz.

Es muss ein negativer Zyklus entstehen, der auf einem Weg von **s** nach **D** durchlaufen werden kann. Dann kann es keinen kürzesten Weg geben, da die Länge mit jedem zusätzlichen Durchlauf des Zyklus kürzer wird. Z.B. **E** →**C**, Gewicht 1.

Aufgabe 4: Java-Programmierung ($6 + 6 + 4 = 16$ Punkte)

Im Folgenden ist die Klasse `LongestPathDAG` gegeben. Sie soll für gerichtete, gewichtete und azyklische Graphen einen **längsten** Pfad berechnen. Man kann dafür die topologische Sortierung anpassen. Diese Anpassungen fehlen in der vorliegenden Implementierung. Die korrekte Initialisierung der Variablen fehlt vollständig und die relax-Funktion ist fehlerhaft.

```
1  public class LongestPathDAG {
2      private double[] distTo;
3      // distTo[v] = Distanz des laengsten Pfad s->v
4      private DirectedEdge[] edgeTo;
5      // edgeTo[v] = letzte Kante auf dem laengsten Pfad s->v
6
7     public LongestPathDAG(WeightedDigraph G, int s) {
8  // TODO: b) Initialisierung
9          // erstellt eine topologische Sortierung:
10         Topological topological = new Topological(G);
11         for (int v : topological.order()) {
12             for (DirectedEdge e : G.adj(v))
13                 relax(e);
14         }
15     }
16 // TODO a) Korrektur
17     private void relax(DirectedEdge e) {
18         int v = e.from(), w = e.to();
19         if (distTo[w] > distTo[v] + e.weight()) {
20             distTo[w] = distTo[v];
21         }
22     }
23 }
```

Anmerkung: Die Klassen `DirectedEdge`, `WeightedDigraph` und `Topological` kennen Sie aus den Hausaufgaben. Die genaue Implementierung ist für die Bearbeitung dieser Aufgabe nicht relevant. `G.adj(v)` gibt die adjanzenten Kanten und `topological.order()` gibt ein Iterable über alle Knoten in der Reihenfolge der topologischen Sortierung zurück. Für eine gerichtete Kante `e`, welche von `v` nach `w` geht, können Sie mit `e.from()` auf `v`, mit `e.to()` auf `w` und mit `e.weight()` auf das Gewicht der Kante `e` zugreifen.

(a) Schreiben Sie eine (korrigierte) Methode `relax`, die die Relaxierung der Kanten vornimmt. Dabei muss das Problem des **längsten** Weges mit der vorliegenden Klasse korrekt gelöst werden. Mit der korrigierten Fasssung muss ein längster Pfad über `edgeTo` rekonstruiert werden können.

```
1  private void relax(DirectedEdge e){
2          int v=e.from(), w=e.to();
3          if(distTo[w]<distTo[v]+e.weight()){
4              distTo[w]=distTo[v];
5
6              edgeTo[w]=e;
7
8          }
9
10
11
12
13
14
15
16
17 }
```

(b) Ergänzen Sie die fehlenden Zeilen des Konstruktors `LongestPathDAG` für die Initialisierung der Variablen. Diese sollen die Variablen der Klasse korrekt initialisieren, sodass ein **längster** Weg mithife der topologischen Sortierung gefunden werden kann.
Notation: `G.V()` gibt die Anzahl der Knoten des Graphen `G` zurück.

```
1  public LongestPathDAG(WeightedDigraph G, int s) {
2        distTo = new double[G.V()];
3
4        edgeTo = new DirectedEdge[G.V()];
5
6        for(int v = 0; v<G.V(); v++){
7            distTo[v] = Double.NEGATIVE_INFINITY;
8
9        }
10       distTo[s] = 0.0;
11
12
13
14       ` L \
15
16
17 }
```

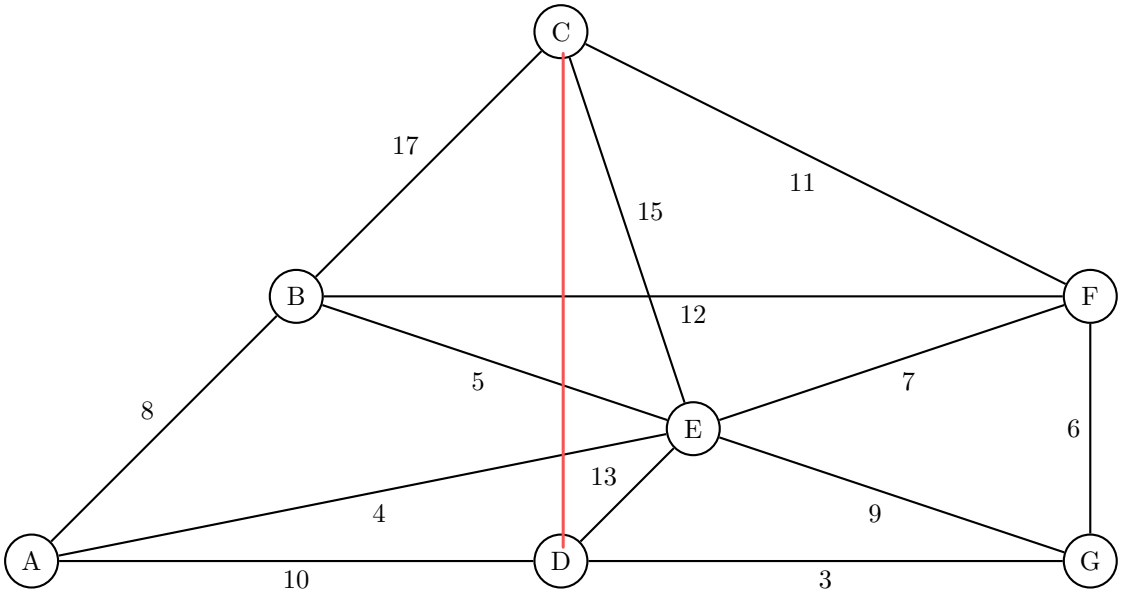
(c) Könnte man alternativ zur topologischen Sortierung den Dijkstra Algorithmus oder den Bellman-Ford Algorithmus verwenden, um den längsten Pfad in einem gewichteten, gerichteten Graphen **mit möglichen positiven Zyklen** zu finden? Begründen Sie. (1 Satz)

Nein

positiven Zyklen haben hier gleichen Effekt wie negativen Zyklen in dem SSSP

Aufgabe 5: Minimum Spanning Tree ($10 + 3 =$ **13 Punkte**)

Hinweis: Der folgende Graph hat unterschiedliche ganzzahlige Gewichte zwischen 3 und 17. Die Kantengewichte stehen in der Mitte der Kante.



(a) Notieren Sie die Gewichte der Kanten, die zum MST des oben gegebenen Graphen gehören in der Reihenfolge, in der Prims Algorithmus sie hinzufügen würde.
Starten Sie im Knoten **A**.

4 - 5 - 7 - 6 - 3 - 11

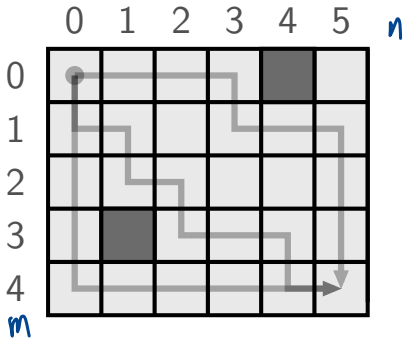
(b) Nehmen Sie an, die Kante **C**–**D** mit dem Gewicht α wird dem Graphen hinzugefügt. Unter welcher Bedingung ist diese Kante immer Teil eines MST? Geben Sie dabei das größtmögliche ganzzahlige Kantengewicht an.

< 11

10

Aufgabe 6: Dynamische Programmierung ($7 + 3 + 3 = 13$ Punkte)

Es ist ein Raster aus $N \times M$ Feldern gegeben. Die Funktion $F(n, m)$ ist für $0 \leq n < N$ und $0 \leq m < M$ als *0* definiert, wenn Feld (n, m) *frei* ist. Falls das Feld **blockiert** ist, ist $F(n, m)$ als **1** definiert. Ein Roboter startet in der linken oberen Ecke $(0, 0)$ und kann sich in jedem Schritt ein Feld nach unten oder ein Feld nach rechts bewegen. Es soll die Anzahl unterschiedlicher Wege von der oberen linken zur unteren rechten Ecke bestimmt werden. Die Wege dürfen nur über freie Felder verlaufen. Die Felder des Start- und Endknoten sind immer frei.



Die Abbildung zeigt ein Raster der Größe 5×4 in dem die Felder $(4, 0)$ und $(1, 3)$ blockiert sind. Drei mögliche Wege von der oberen linken zur unteren rechten Ecke sind eingezeichnet.

(a) Eine effiziente Lösung dieser Aufgabe kann durch Dynamische Programmierung erreicht werden. Als Grundlage dient eine Funktion $\textsc{Opt}(n, m)$, die die Anzahl der Wege von Feld $(0, 0)$ bis Feld (n, m) angibt ($n < N$ und $m < M$). Stellen Sie eine rekursive Definition der $\textsc{Opt}$ Funktion auf (mathematische Schreibweise, kein Java- oder Pseudocode).

$$OPT(n,m) = \begin{cases} 0 & \text{falls } F(n,m) = 1 \\ 1 & \text{falls } m = 0 \text{ und } n = 0 \\ OPT(n-1,m) & \text{falls } n > 0 \text{ und } m = 0 \\ OPT(n,m-1) & \text{falls } n = 0 \text{ und } m > 0 \\ OPT(n-1,m) + OPT(n,m) & \text{sonst} \end{cases}$$

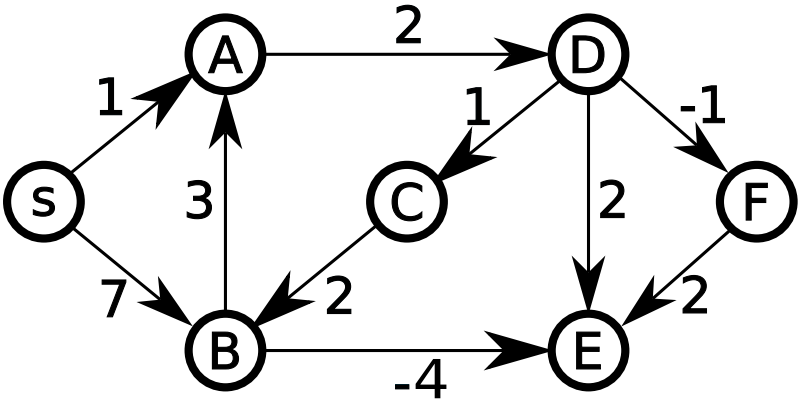
(b) Eine solche rekursive Funktion kann auf direkte Weise durch rekursive Aufrufe implementiert werden. Bei Dynamischer Programmierung werden Ergebnisse in einem Array gespeichert. Warum kann dadurch eine wesentliche bessere Laufzeit erzielt werden? (1-2 Sätze)

(Eine Voraussetzung ist, dass die Aufgabe überlappenden Teilprobleme besitzt (overlapping subproblems) und sich die Lösung auf die Lösung von Teilproblemen zurückführen lässt (optimal substructure).)
Es wird die vielfache Berechnung derselben Teilergebnisse dadurch vermieden, dass die Lösungen für Subprobleme nur einmal berechnet, im Array gespeichert und dann vielfach wieder abgerufen werden.

(c) Für die Speicherung von Teilergebnissen bei der Dynamischen Programmierung gibt es den *top-down* Ansatz (*memoization*) und den *bottom-up* Ansatz (*tabulation*). Beschreiben Sie grob in welchen Fällen der *bottom-up* Ansatz vorzuziehen ist. (1-2 Sätze)

Bei dem *bottom-up* Ansatz werden die Werte emphaller Teillösungen berechnet, bei dem *top-down* Ansatz nur die *tatsächlich benötigten*. Wenn die Werte aller, oder zumindest der allermeisten Teillösungen für die Lösung benötigt werden, ist der *bottom-up* Ansatz schneller und daher vorzuziehen.

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!