TECHNISCHE UNIVERSITÄT BERLIN

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

Algorithmen und Datenstrukturen, SoSe 18
Klausur am 08.08.2018

Bitte füllen Sie alle folgenden Felder aus:

Klausur-ID:	**666A**
tubIT-login:	
Vorname:	
Nachname:	
Matrikelnummer:	
Studiengang:	
Hochschule:	

Durch meine Unterschrift bestätige ich die Korrektheit obiger Angaben.

Ort, Datum — Unterschrift

Beachten Sie die folgenden Hinweise!

- Die Klausuren sind nummeriert und werden anonymisiert korrigiert. Bitte schreiben Sie Ihren Namen **nur** auf das Deckblatt.
- Insgesamt können in der Klausur **100 Punkte** erreicht werden.
- Diese Klausur besteht mit diesem Deckblatt aus den (nummerierten) Seiten **1-14**.
- Schreiben Sie **nicht** mit roter Farbe, grüner Farbe oder Bleistift. Diese Lösungen werden nicht bewertet!
- Notieren Sie Ihre Antworten nur auf dem Blatt (oder dessen Rückseite), auf dem auch die zugehörige Aufgabe steht, da die Aufgaben getrennt korrigiert werden.
- Geben Sie nur eine Lösung pro Aufgabe ab, streichen Sie alle alternativen Lösungsansätze auf Schmier-/Notitzblättern durch.
- Bitte den Barcode am Ende der Seiten nicht beschädigen oder überschreiben.

Zusatzblätter No.:	

Aufgabe 1: Multiple Choice ($8 \times 2 = 16$ Punkte)

Bitte kreuzen Sie alle wahren Aussagen an.

- Es ist bei allen Teilaufgaben mindestens eine Aussage wahr, aber auch mindestens eine Aussage falsch.
- Ein gesetztes Kreuz kann durch ein leeres ◯-Symbol links neben der Aussage aufgehoben werden.
- Es werden nur für vollständig richtige Teilaufgaben Punkte vergeben.

(a) Welche Wachstumsordnung beschreibt die Laufzeit der folgenden Methode `f()` am genausten?

```
public static int f(int N) {
  if (N <= 1)
    return 0;
  return 2 * f(N/2);
}
```

◯	$\boldsymbol{O}(\log(N))$
◯	$\boldsymbol{O}(N)$
◯	$\boldsymbol{O}(2N)$
◯	$\boldsymbol{O}(N \log(N))$
◯	$\boldsymbol{O}(N^2)$
◯	$\boldsymbol{O}(2^N)$

(b) Welche Wachstumsordnung hat die Laufzeit des Dijkstra Algorithmus für einen Graphen $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ mit nicht-negativen Kantengewichten, wenn er mit einer indizierten Vorrangwarteschlange (*index PQ*) implementiert wird?

◯	$\boldsymbol{O}(V + E)$
◯	$\boldsymbol{O}(VE)$
◯	$\boldsymbol{O}(V \log E)$
◯	$\boldsymbol{O}(V^2 \log E)$
◯	$\boldsymbol{O}(E \log V)$

(c) Für einen Spannbaum T eines zusammenhängenden Graphen $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ gilt:

◯	T hat V Kanten.
◯	T ist zusammenhängend und das Entfernen einer beliebigen Kante macht ihn unzusammenhängend.
◯	T hat genau einen Zyklus.
◯	T benutzt jede Kante des Graphen $\boldsymbol{G}$ genau einmal.
◯	Zwei beliebige Knoten von $\boldsymbol{G}$ werden durch einen einfachen Pfad der in T liegt verbunden. Ein einfacher Pfad ist ein Pfad, in dem keine Knoten doppelt vorkommen.
◯	T ist eindeutig, wenn $\boldsymbol{G}$ gerichtet ist.

(d) Welche Aussagen treffen auf den Algorithmus von Kruskal zur Bestimmung eines minimalen Spannbaums zu?

○	Es wird eine `UnionFind` Stuktur benutzt, um effizient zu überprüfen, ob das Hinzufügen einer neuen Kante einen Zyklus erzeugen würde.
○	Es wird eine `UnionFind` Stuktur benutzt, um effizient zu überprüfen, ob das Weglassen einer neuen Kante den Graphen unzusammenhängend machen würde.
○	Der Algorithmus hat große Ähnlichkeit mit dem Dijkstra-Algorithmus.
○	Der Baum wird schrittweise ausgehend von einem Startknoten s aufgebaut.
○	Der Algorithmus ist nicht effizient, da er Rekursion verwendet.
○	Die Kanten werden nach Gewicht sortiert durchlaufen, angefangen bei einer Kante mit minimalem Gewicht.

(e) Der A* Algorithmus ...

○	benutzt eine Heuristik, um die kürzesten Pfade in Richtung eines gegebenen Zielknotens schneller zu finden.
○	hat eine Laufzeit in $\boldsymbol{O}(V + E)$, wenn die Heuristik konsistent ist.
○	relaxiert jede Kante nur einmal, wenn die Heuristik konsistent ist.
○	löst Kollisionen durch sternförmige Sondierung auf.
○	ist ein approximativer Algorithmus.
○	findet kürzere Pfade als der Dijkstra Algorithmus, wenn die Heuristik zulässig ist.

(f) Welche Aussagen zum Thema Hashing sind zutreffend?

○	Bei Hashing mit separater Verkettung sollte der Belegungsfaktor (load factor) unter $1/2$ liegen.
○	Kollisionen können durch universelles Hashing vermieden werden.
○	Gute Hashfunktionen bilden die Schlüssel möglichst gleichmäßig auf die Hashadressen ab.
○	Beim Hashing kann man beliebige Objekte als Schlüssel verwenden, nicht nur Integerzahlen.
○	Hashtabellen benötigen in den meisten Fällen weniger Speicher, als normale verkettete Listen.
○	Der Zugriff über Hashtabellen ist bei Schlüsseln vom Typ Integer schneller als bei einem normalen Array.

(g) Die Busverbindungen einer Region seien als Graph modelliert. Dabei stellen die Knoten Haltestellen dar und die Kanten stehen für eine Fahrt von Haltestelle zu Haltestelle. Für jede Kante ist die Streckenlänge bekannt. Es soll die Verbindung mit den wenigsten Zwischenstopps (also Haltestellen auf dem Weg) gefunden werden. Start- und Zielhaltestelle sind dabei gegeben. Welcher der folgenden Algorithmen eignet sich dafür? Bei unterschiedlichen Möglichkeiten soll derjenige mit der besten Laufzeit-Garantie angekreuzt werden.

◯	Breitensuche
◯	Tiefensuche
◯	Topologisches Sortieren
◯	Dijksta-Algorithmus
◯	Kruskal-Algorithmus
◯	Edmonds-Karp

(h) Welche der folgenden Aussagen treffen den Kern des Beweises, der zeigt, dass sich das *Traveling Salesman Problem* (unter der Annahme P≠NP) nicht in polynomieller Zeit approximieren lässt?

◯	Zu einem vorgegebenen $\varepsilon > 0$ konstruiert man einen Algorithmus, dessen Kosten größer als $(1 + \varepsilon)$ mal den Kosten der optimalen TSP Tour sind.
◯	Der approximative Algorithmus wird genutzt, um das NP-vollständige Problem der Existenz eines Hamilton-Zyklus in polynomieller Zeit zu lösen.
◯	Man fängt mit einer hohen Kostenschranke an und halbiert diese solange, bis man die Approximationsgrenze unterschreitet.
◯	Man startet mit dem minimalen Spannbaum des Graphen und baut diesen zu einem Zyklus um, dessen Kosten maximal das doppelte der Kosten der optimalen TSP Tour sind.

Aufgabe 2: Hashing ($1 + 4 + 6 = 11$ Punkte)

Benutzen Sie die Hashfunktion: $h((x, y)) = (7x - 2y) \mod 7$

(a) Berechnen und notieren Sie die restlichen Hashwerte in der folgenden Tabelle:

Schlüssel	Hashcode	Hashvalue
A	$(1, 1)$	5
B	$(3, 6)$	2
C	$(2, 1)$	5
D	$(3, 3)$	1
E	$(4, 5)$	4
F	$(4, 3)$	1
G	$(4, 2)$	3
H	$(3, 2)$	
I	$(2, 3)$	

(b) Fügen Sie die Schlüssel A-G in alphabetischer Reihenfolge in die Hashtabelle ein. Verwenden Sie dabei zur Kollisionsauflösung lineares Sondieren mit Inkrement 1 (d.h. $s(n) = n$).

0	1	2	3	4	5	6

(c) Entscheiden Sie welche der folgenden Szenarien durch das Einfügen der Schlüssel A-G aus a) in beliebiger Reihenfolge entstehen können und kreuzen Sie diese an. Hierbei wird wie oben zur Kollisionsauflösung lineares Sondieren mit Inkrement 1 genutzt.
Hinweis: Es ist mindestens eine Antwort richtig und eine Antwort falsch. Für jedes richtig gesetzte und jedes richtig nicht gesetzte Kreuz bekommen Sie 2 Punkte.

○

0	1	2	3	4	5	6
F	A	D	B	G	E	C

○

0	1	2	3	4	5	6
C	A	B	G	F	E	D

○

0	1	2	3	4	5	6
A	F	D	B	G	E	C

Aufgabe 3: Laufzeit ($4 + 6 = 10$ Punkte)

Eine Variante zum Finden minimaler Spannbäume (die nicht in der Vorlesung besprochen wurde) beruht auf der folgenden Eigenschaft:

> Sei ein beliebiger Zyklus in einem Graphen gegeben, und sei e eine der darin enthaltenen Kanten mit dem höchsten Gewicht. Dann gibt es einen minimalen Spannbaum, der die Kante e nicht enthält.

Der Pseudocode für diesen MST-Algorithmus sieht wie folgt aus:

```
1 // gegeben ein gewichteter Graph G = (V, E)
2  sortiere  die Kanten absteigend nach ihrem Gewicht
3 for jede Kante e ∈ E in dieser Reihenfolge:
4   if e ist Teil eines Zyklus von G:
5     entferne e aus G
6   end
7 end
8 return G   // G ist der MST
```

(a) Als Teil des Algorithmus muss für eine gegebene Kante e geprüft werden, ob e Teil eines Zyklus in $\boldsymbol{G}$ ist. Beschreiben Sie einen Algorithmus, der dies in linearer Zeit erledigt. Dabei können Sie Verfahren benutzen, die in der Vorlesung besprochen wurden.

(b) Wie ist die Laufzeit des gegebenen Algorithmus? Die Laufzeit muss für jeden Teil des Pseudocodes angegeben und daraus die Gesamtlaufzeit folgerichtig abgeleitet werden.

Aufgabe 4: Objektorientierte Programmmierung ($4+4+2+1=11$ Punkte)

Im Folgenden ist unvollständiger Code vorgegeben.

(a) Vervollständigen Sie den Code (an den unterstrichenen Stellen) so, dass er kompiliert und dass die Klassen `Car` und `Bicycle` von der Klasse `Vehicle` erben.

```
1  _____________________________ Vehicle {
2     protected int numberOfWheels;
3
4     abstract void refillAir(int num);
5
6     public Vehicle(int numberOfWheels) {
7        this.numberOfWheels = numberOfWheels;
8     }
9
10    public void repair() {
11       for (int i = 0; i < numberOfWheels; i++) {
12          refillAir(i);
13       }
14    };
15 }
```

```
1  _____________________________ Car _____________________________ {
2     public Car() {
3        super(4);
4     }
5
6     public void refillAir(int num)
7     {
8        System.out.print("c"+num);
9     }
10 }
```

```
1  _____________________________ Bicycle _____________________________ {
2     public Bicycle() {
3        super(2);
4     }
5
6     public void refillAir(int num)
7     {
8        System.out.print("b"+num);
9     }
10 }
```


ID **666A**-7

(b) Betrachten Sie nun die folgende Methode:

```
1  public class Garage {
2     public static void main(String[] args) {
3        Vehicle [] allVehicles = new Vehicle[3];
4        Vehicle [] vehicles = new Vehicle[3];
5        Car c1 = new Car();
6        allVehicles[0]=c1;
7        Bicycle b1 = new Bicycle();
8        allVehicles[1]=b1;
9        Bicycle b2 = new Bicycle();
10       allVehicles[2]=b2;
11
12       try{
13          for (Vehicle v : allVehicles)
14             v.repair();
15          }
16       catch(Exception e){
17          System.out.println(e);
18          }
19    }
20 }
```

Was wird bei der Ausführung auf der Konsole ausgegeben?

(c) Betrachten Sie weiterhin die Klasse `Garage`. Wenn Sie den try-catch-Block (Zeile 12-18) mit dem folgenden ersetzen, was gibt das Programm aus?

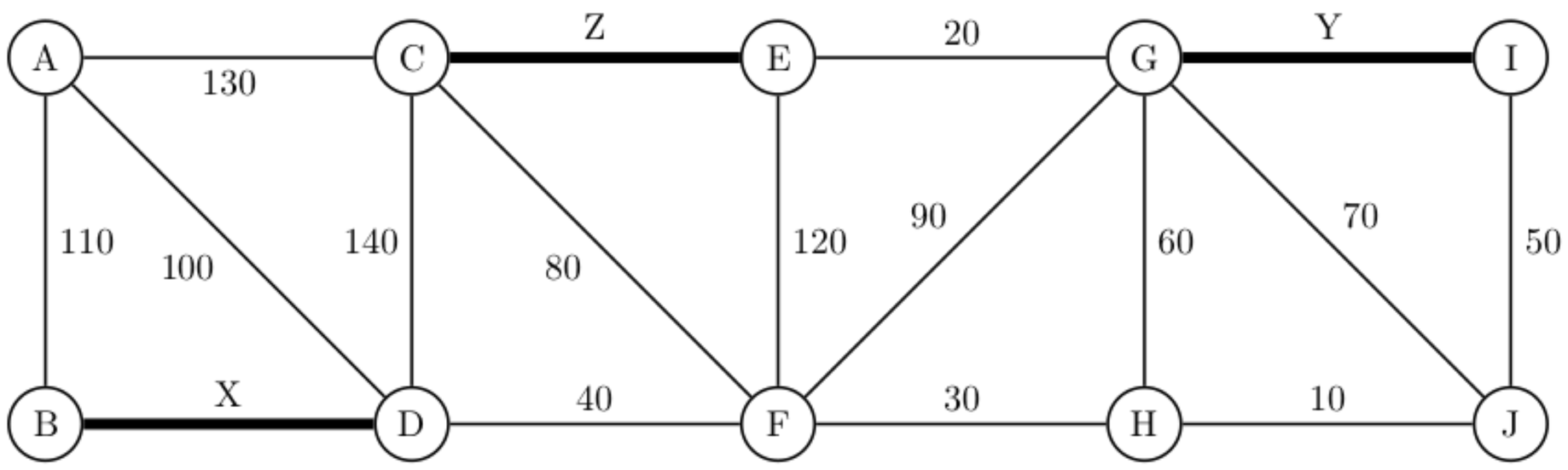
```
1        try{
2           for (int i=0; i<=3; i++)
3              allVehicles[i].repair();
4           }
5        catch(Exception e){
6            System.out.println(e);
7           }
```

(d) Betrachten Sie weiterhin die Klasse `Garage`. Wenn Sie den try-catch-Block (Zeile 12-18) mit dem folgenden ersetzen, was gibt das Programm nun aus?

```
1        try{
2           for (int i=0; i<3; i++)
3              vehicles[i].repair();
4           }
5        catch(Exception e){
6             System.out.println(e);
7           }
```


Aufgabe 5: Minimum Spanning Tree ($6 + 3 = 9$ Punkte)

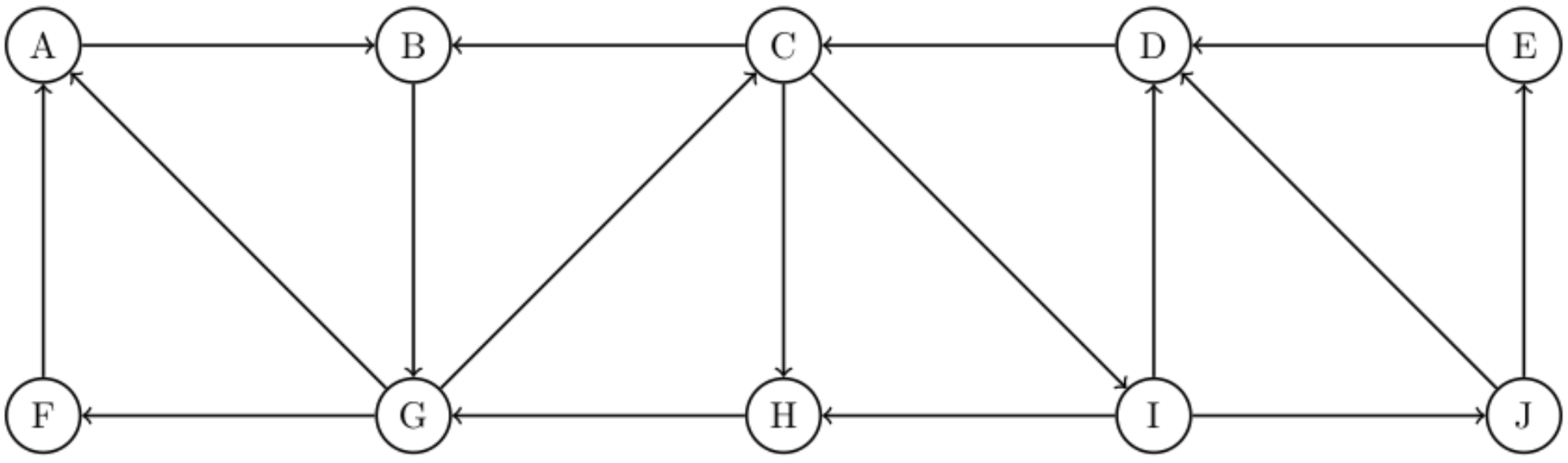
Nehmen Sie an, der Minimum Spanning Tree (MST) dieses Graphen enthalte die Kanten X, Y und Z.



(a) Notieren Sie die Gewichte der restlichen Kanten, d.h. alle außer X, Y und Z, die zum MST dieses Graphen gehören müssen unter der obigen Annahme.

(b) Geben Sie jeweils die größte obere Schranke für die Gewichte der Kanten X, Y und Z an, mit der garantiert ist, das alle drei Kanten tatsächlich Teil des MST sind.
Hinweis: Geben Sie individuell für jedes Gewicht x, y, z eine Schranke an, wobei x, y und z die entsprechenden Gewichte zu den Kanten X,Y und Z sind.

Aufgabe 6: Tiefensuche ($6 + 6 = 12$ Punkte)



Führen Sie eine Tiefensuche auf dem Graphen aus. Fangen Sie im Knoten A an.
Gehen Sie dabei davon aus, dass in jedem Knoten die benachbarten Knoten in alphabetischer Reihenfolge abgearbeitet werden. Z.B. würde die Kante I-D vor der Kante I-J bearbeitet werden.

(a) Notieren Sie die Knoten in der Hauptreihenfolge (*pre-order*), d.h. in der Reihenfolge, in der sie entdeckt werden.

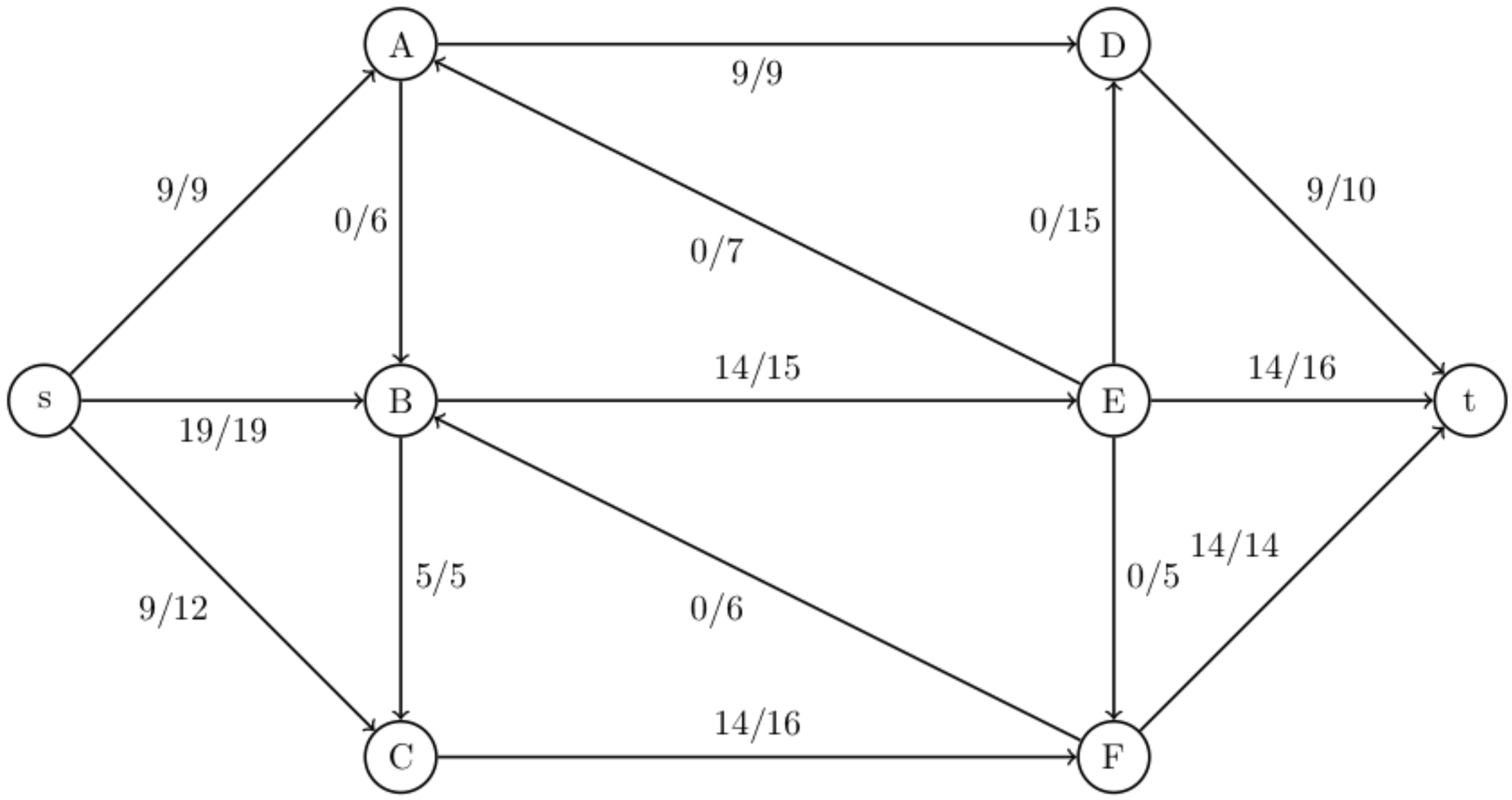
A B

(b) Notieren Sie die Knoten in der Nebenreihenfolge (*post-order*), d.h. in der Reihenfolge, in der sie fertig abgearbeitet sind. Ein Knoten ist abgearbeitet, wenn alle ausgehenden Nachbarn bereits bearbeitet wurden.

H D

Aufgabe 7: Edmonds-Karp Algorithmus ($2+5+1+1=9$ Punkte)

Betrachten Sie den folgenden Flussgraphen mit Quelle s und Senke t und dem eingetragenen Fluss:



(a) Bestimmen Sie den Wert, den der Fluss in der Abbildung hat.

Flusswert =

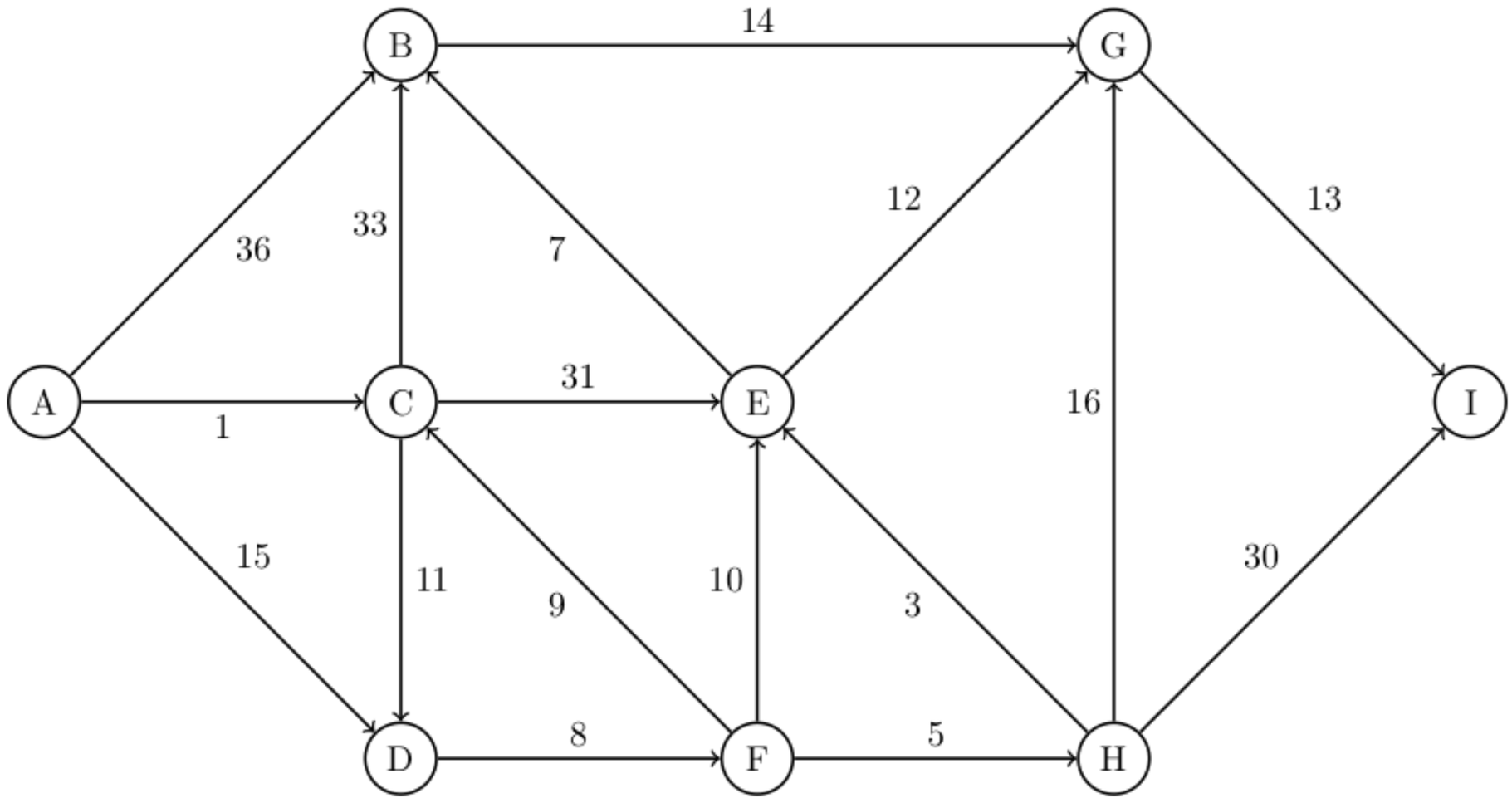
(b) Führen Sie eine Iteration des Edmonds-Karp Algorithmus aus und notieren Sie die Knoten eines vergrößernden Pfades beginnend in s und endend in t.

(c) Notieren Sie den Wert des maximalen Flusses des oben gezeigten Flussgraphen.
Hinweis: Nach b) ist der Fluss maximal ist.

(d) Notieren Sie die Kapazität des minimalen Schnittes in dem obigen Flussgraphen.

Aufgabe 8: Dijkstra Algorithmus ($8 + 2 = 10$ Punkte)

Betrachten Sie den folgenden gerichteten Graphen. Die Zahlen an den Pfeilen geben die Kantengewichte an.

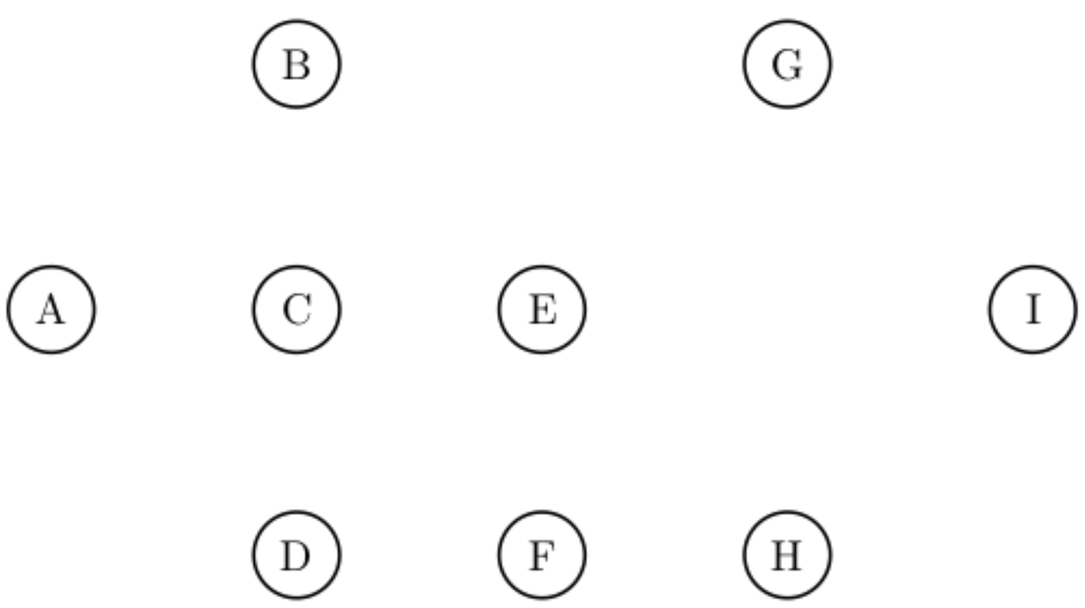


Führen Sie Dijkstras Algorithmus auf dem obigen Graphen ausgehend von dem Startknoten A aus.

(a) Notieren Sie die Knoten in der Reihenfolge, in der sie aus der Priorityqueue entfernt werden. Geben Sie zudem zu jedem Knoten die Distanz zum Knoten A an, definiert über die Kantengewichte.

Knoten	A								
Distanz	0								

(b) Zeichnen Sie alle kürzesten Pfade in den Graphen ein.



Aufgabe 9: Dynamisches Programmieren ($3 + 7 + 2 = 12$ Punkte)

Ein *Palindrom* ist eine Zeichenfolge die rückwärts gelesen dieselbe Folge ergibt wie vorwärts gelesen. Das Wort RENTNER ist also ein Palindrom. Eine *Palindrom-Teilfolge* ist eine Teilfolge einer Zeichenkette, die ein Palindrom ist. Dabei muss die Teilfolge nicht an einem Stück in der gegeben Zeichenkette vorkommen.
Beispiel: DATENSTRUKTUREN enthält u.a. die Palindrom-Teilfolgen NRUKURN und ERUTURE. Auch UKU, NN und jeder einzelne Buchstabe sind (kurze) Palindrom-Teilfolgen der Zeichenkette DATENSTRUKTUREN.
Es soll ein Algorithmus entwickelt werden, der zu einer gegebenen Zeichenkette `str` die Länge der längsten Palindrom Teilfolge bestimmt.

(a) Welche Laufzeit hat der *brute-force* Ansatz (alle Möglichkeiten durchprobieren)?

(b) Geben Sie eine rekursive Definition der $\textsc{Opt}(i, j)$ an, die als Grundlage für dynamisches Programmieren verwendet werden kann.

$\textsc{Opt}(i, j)$ ist hierbei die Länge der längsten Palindrom-Teilfolge von `str[i:j]`.

(c) Was ist die Laufzeit einer Implementation mit Hilfe der Opt Funktion und dynamischer Programmierung.
Hinweis: Sie können c) unabhängig von b) lösen.

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