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 ($4+3+2=9$ 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.

```
public static int f1(int N) {
    int x = 0;
    for (int i = 1; i < 5*N; i += 2) {        O(N)
        x += i/2;
    }
    for (int i = N*N/2; i >= 0; i--) {        O(N^2)
        x += i*i;
    }
    return x;
}
```

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

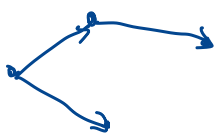
Wachstumsordnung von `f1(int N)`: $O(N^2)$

Wachstumsordnung von `f2(int N)`: $O(N \log N)$

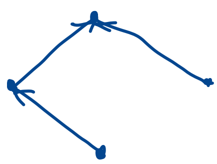
(b) In dem AlgoDat Skript wurde definiert, dass ein gerichteter Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ **stark zusammenhängend** heißt, wenn es für alle Knoten v und w aus $\boldsymbol{V}$ einen gerichteten Weg von v nach w gibt.
Weiterhin definieren wir, dass ein gerichteter Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ **semi zusammenhängend** heißt, wenn es für alle Knoten v und w aus $\boldsymbol{V}$ einen gerichteten Weg von v nach w **oder** einen Weg von v nach w (oder in beiden Richtungen) gibt.
Ein gerichteter Graph heißt **(schwach) zusammenhängend**, wenn der zugehörige ungerichtete Graph (gerichtete Kanten werden durch ungerichtete ersetzt) zusammenhängend ist.

Zeichnen Sie jeweils ein Beispiel für einen Graphen mit folgenden Eigenschaften:

(1) semi-zusammenhängend, aber nicht stark zusammenhängend



(2) (schwach) zusammenhängend, aber nicht semi-zusammenhängend



(c) Gegeben sei ein gerichteter azyklischer Graph (DAG) mit Kantengewichten sowie ein Startknoten s. Bleiben die kürzesten Wege erhalten, wenn alle Gewichte um 10 erhöht werden? Begründen Sie kurz, oder geben Sie ein Gegenbeispiel mit maximal 4 Knoten an.

ja

die kürzesten Wege ändert sich nicht

da die Reihenfolge des Algo von prim und Kruskal sich nicht ändert

$f(x) = x + 10$

Aufgabe 2: Java - Breitensuche ($3 + 7 + 1 + 2 = 13$ Punkte)

Im Folgenden ist die Klasse `BreadthFirstDist` als Lückentext gegeben, die eine Breitensuche in Form der Methode `bfs(Graph G, int s)` implementieren soll. Der Vorgabecode enthält allerdings Fehler, so dass der Ablauf keine korrekte Breitensuche darstellen wird, siehe Aufgabenteil (d). Sie dürfen voraussetzen, dass der Graph und Startknoten fehlerfrei übergeben werden.

```
1  public class BreadthFirstDist {
2      private boolean[] marked;   // marked[v] = is there an s-v path
3      private int[] distTo;       // distTo[v] = number of edges s-v path
4  
5     //constructor
6  
7      public BreadthFirstDist(Graph G, int s) {
8          marked = new boolean[G.V()];
9          distTo = new int[G.V()];
10         bfs(G, s);
11     }
12 
13     private void bfs(Graph G, int s) {
14         Queue<Integer> q = new LinkedList<Integer>();
15         for (int v = 0; v < G.V(); v++)
16             distTo[v] = 0;
17         q.add(s);
18 
19         while (!q.isEmpty()) {
20             int v = q.poll();
21             for (int w : G.adj(v)) {
22                 if (!marked[w]) {
23                     distTo[w] = distTo[v] + 1;
24                     marked[w] = true;
25                     q.add(w);
26                 }
27             }
28         }
29     }
30     public boolean hasPathTo(int v) {
31         return marked[v];
32     }
33 
34     public int distTo(int v) {
35         return distTo[v];
36     }
37 
38 }
```

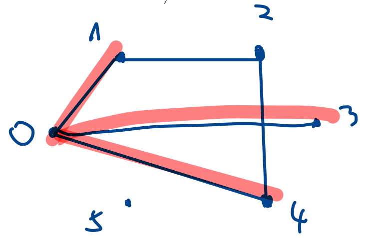
(a) Füllen Sie die Lücken in Zeilen 1, 7, 10, 13, 31 und 35. Wenn alle Lücken gefüllt sind, sollte bei der Instanzierung einer Objektes die fehlerhafte Breitensuche auf dem `Graph G` vom Startknoten `int s` bis zum Ende ohne Fehlermeldung durchlaufen.

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

```
1      public static void main(String[] args) {
2          Graph G= new Graph(6);
3          G.addEdge(0, 1);
4          G.addEdge(0, 3);
5          G.addEdge(0, 4);
6          G.addEdge(1, 2);
7          G.addEdge(2, 4);
8
9          BreadthFirstDist B= new BreadthFirstDist(G, 0);
10         System.out.println(B.distTo(0));
11         System.out.println(B.distTo(5));
12     }
```

s=0

Skizzieren Sie den Graphen, der von der `main()` Methode erzeugt wird. Simulieren Sie die Ausführung von `bfs(G, 0)`, die durch Zeile 9 ausgelöst wird. Schreiben Sie den Zustand der Klassenvariablen als Array und die Queue **vollständig** am Ende der ersten beiden Durchläufe der **while**-Schleife, nach der **for**-Schleife in der Tabelle auf, also den Zustand nach Zeile 27.
Dabei sind Ihnen die Knoten **int** v, die gerade bearbeitet werden, bereits angegeben. Die Knoten sind in der Adjazenzliste nach ihrem Index sortiert. Rechts finden Sie den Code Abschnitt nochmal in kleiner, damit Sie nicht blättern müssen.




```
13 private ____________ bfs(Graph G, int s) {
14     Queue<Integer> q = new LinkedList<Integer
       >();
15     for (int v = 0; v < G.V(); v++)
16         distTo[v] = 0;
17     q.add(s);
18
19     while (!q.isEmpty()) {
20         int v = q.poll();
21         for (int w : G.adj(v)) {
22             if (!marked[w]) {
23                 distTo[w] = distTo[v] + 1;
24                 marked[w] = true;
25                 q.add(w);
26             }
27         }  // ZWISCHENSTAND TABELLE!!________
28     }
29 }
```

Integer.MAX_VALUE

开始时: Queue={0}

Schritt 0 : 取出Queue第一个元素 → 0   Queue = ∅

0 与 1, 3, 4 连接, 且 1,3,4 不在Queue

所以把 1,3,4 的 distTo 改为 1, ⇒ distTo[] = [0,1,0,1,1,0]

index: 0 1 2 3 4 5

再标记为 besucht → Marked[] = [f,t,f,t,t,f]

index: 0 1 2 3 4 5

更新Queue {} → {1,3,4}

`int v`	`distTo[]`	`marked[]` (t=true, f=false)	Queue q
0	[0, 1, 0, 1, 1, 0]	[f, t, f, t, t, f]	1, 3, 4
1	[0, 1, 2, 1, 1, 0]	[f, t, t, t, t, f]	3, 4, 2

(c) Was ist die Ausgabe der `main`-Methode?

0

0

(d) Notieren und berichtigen Sie die Fehler in der Methode `bfs(Graph G, int s)` hier. Geben Sie dabei die Zeilennummern an. Beachten Sie, dass es keine Laufzeit- oder Kompilierfehler sind.

16

Aufgabe 3: Hashing ($4 + 3 + 2 = 9$ Punkte)

Für diese Aufgabe werden die Hashfunktion $h(x) = x \mod 8$ und die in der Tabelle angegebenen Schlüssel verwendet:

Schlüssel	A	B	C	D	E	F	G
Hashcode	3	4	8	11	13	16	19
Hashadresse	3	4	0	3	5	0	3

Die letzte Tabellenzeile *Hashadresse* wird nicht gewertet, sie ist nur als Hilfestellung gedacht.

(a) Die Schlüssel wurden in diese Hashtabelle eingefügt. Dabei wurde zur Kollisionsauflösung *lineares Sondieren* (mit Inkrement 1, also $s(n) = n$) verwendet.

Hashadresse	0	1	2	3	4	5	6	7
Schlüssel	C ^{0}	F ^{0}		D ^{3}	A ^{3}	E ^{5}	B ^{4}	G ^{3}

Welche der folgenden Reihenfolgen können **nicht** zu der obigen Hashtabelle geführt haben? Begründen Sie jeweils.

(1) C, E, B, F, D, A, G

falsch Schlüssel von B soll 4 sein

(2) D, A, E, F, B, C, G
3 3 5 0

falsch Schlüssel von F soll 0 sein

(3) D, E, A, C, B, F, G
3 5 3 0 4 0 3

richtig

(b) Wenn Schlüssel D mit der Methode des *sofortigen Löschens* (*eager deletion*) aus der Hashtabelle entfernt wird, wie sieht die Tabelle danach aus?

Hashadresse	0	1	2	3	4	5	6	7
Schlüssel	C	F		A	B	E	G	

(c) Unten sehen Sie zwei Varianten der Funktion *noDoubles()*, die zu einem gegebenen Array $a[]$ eine Liste zurückgibt, in der jedes Element von a nur einmal auftritt. Erklären Sie, welchen Vorteil die Variante (B) hat, die als temporäre Datenstruktur eine Hashtabelle benutzt!

(A)

```
procedure noDoubles(a) // a: ein Array
NoD : verkettete Liste

for i in a
    if i ist nicht in NoD
        füge i in NoD ein
    end
end
return NoD
```

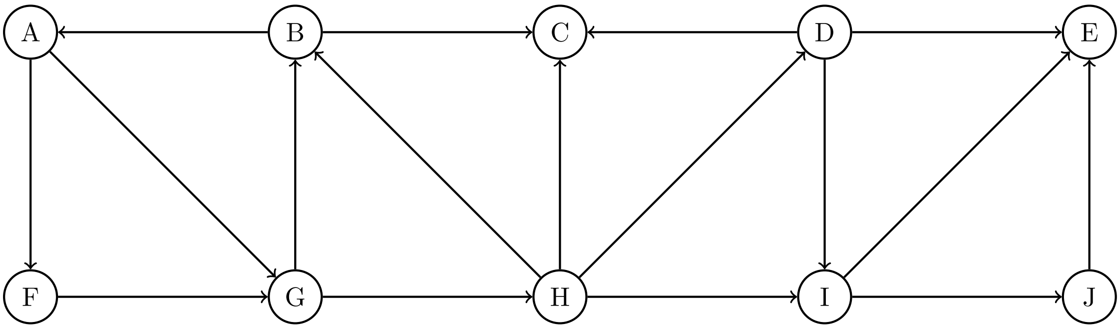
(B)

```
procedure noDoubles(a) // a: ein Array
NoD : verkettete Liste
H : Hashtabelle
for i in a
    if i ist nicht in H
        füge i in H ein
        füge i in NoD ein
    end
end
return NoD
```

schneller

Hashtabellen realisieren meisten eine konstante Laufzeit

Aufgabe 4: Tiefensuche ($6 + 6 + 2 = 14$ Punkte)



AFGBC
AFGB
AFGHDE
AFGHD
AFGHDIJ

Führen Sie eine Tiefensuche auf dem Graphen aus. Fangen Sie bei 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-E vor der Kante I-J bearbeitet werden.

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

A F G B C H D E I J

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

C B E J I D H G F A

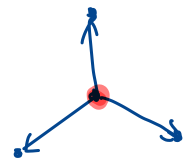
(c) Der obige gerichtete Graph hat keine Topologische Sortierung. Es kann aber eine topologische Sortierung hergestellt werden, wenn man aus dem Graphen eine Kante löscht. Welche Kante ist das?

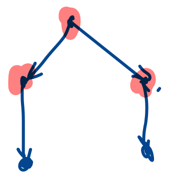
B → A

Aufgabe 5: Algorithmus für Urknoten ($3 + 3 + 1 + 3 = 10$ Punkte)

Ein Knoten u in einem gerichteten Graphen $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ heißt **Urknoten** von $\boldsymbol{G}$, wenn allen anderen Knoten über einen gerichteten Pfad von u aus erreicht werden können.

(a) Skizzieren Sie einen gerichteten Graphen mit vier Knoten, von denen genau einer ein Urknoten ist und einen zweiten Graphen mit fünf Knoten, von denen genau drei Urknoten sind. Markieren Sie alle Urknoten.





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

wir benutzen dfs oder bfs mit Anfangsknoten u

u ist ein Urknoten, falls alle knoten in G besucht werden

sonst nicht.

Laufzeit von dfs/bfs: $O(V+E)$ Cheken ob alle knoten in G besucht: $O(V)$

insgesamt: $O(V+E) + O(V) = O(V+E)$

(c) Mit der Methode aus (b) kann ein einfacher Algorithmus erworfen werden, der feststellt, ob ein Graph $\boldsymbol{G}$ einen Urknoten besitzt: Das Verfahren aus (b) wird der Reihe nach mit allen Knoten gestartet. Gegeben Sie die Wachstumsordnung der Laufzeit möglichst genau an.
Hinweis: Dieser Aufgabenteil kann auch gelöst werden, wenn man (b) nicht bearbeitet hat.

$O(V(V+E))$

(d) Den Ansatz aus (c) soll verbessert werden. Beschreiben Sie einen Algorithmus mit Laufzeit in $\boldsymbol{O}(V + E)$, der entscheidet, ob ein Graph $\boldsymbol{G}$ einen Urknoten besitzt. Begründen Sie die Laufzeit. Sie können die folgende Eigenschaft verwenden:

> Falls $\boldsymbol{G}$ (mindestens) einen Urknoten hat, so ist der Knoten am Ende der Hauptordnung (*post-order*), der also als letzter fertig bearbeitet wurde, ein Urknoten.

Wir nutzen DFS von einem random Startknoten und bekommt Post-order.

Danach nehmen wir der Knoten am Ende der Post-order und nutzen DFS

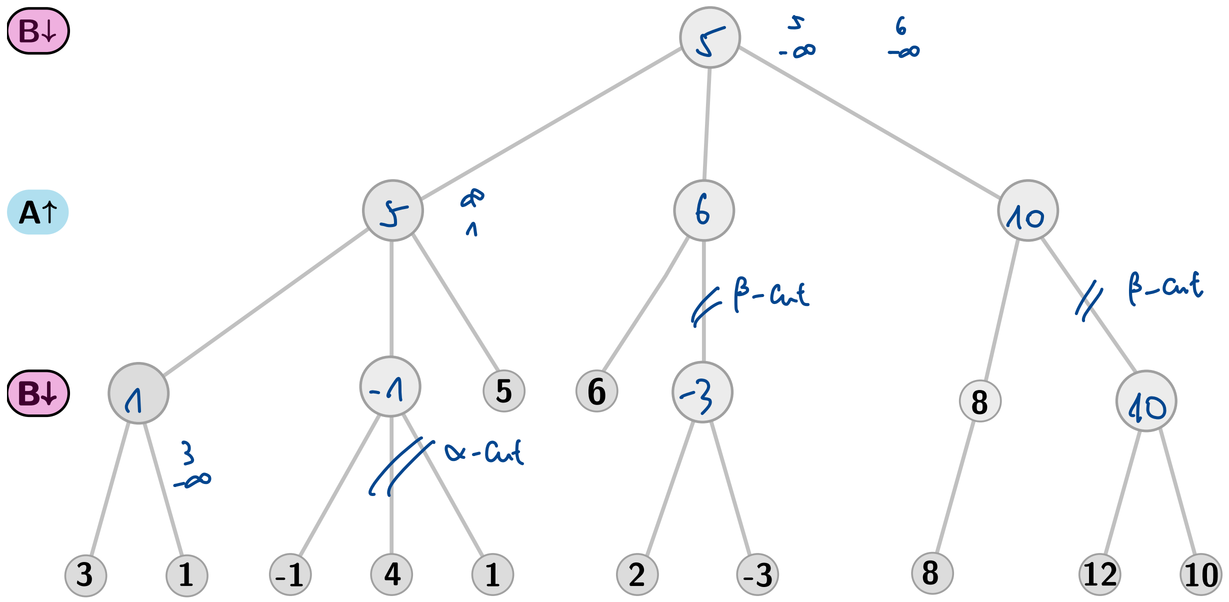
mit diese Knoten als Startknoten.

Falls in 2.DFS alle Knoten besucht werden, ist die Knoten ein Urknoten

Laufzeit: $O(2(V+E)) = O(V+E)$

Aufgabe 6: Minimax- und Alpha-Beta-Algorithmus

($4 + 4 = 8$ **Punkte)**



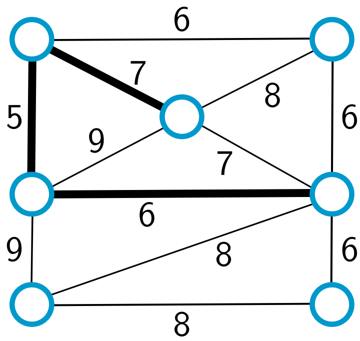
(a) Vervollständigen Sie den obigen Minimax Suchbaum.

(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.

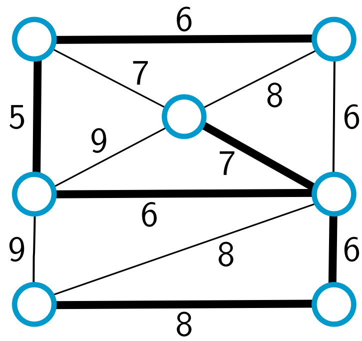
Aufgabe 7: Minimaler Spannbaum (13 Punkte)

In der folgenden Abbildung sehen Sie fünf mal den gleichen Graphen, wobei jedes mal unterschiedliche Kanten markiert sind. Kreuzen Sie an, ob diese Kanten nur durch den Prim Algorithmus, nur durch den Kruskal Algorithmus, durch beide oder durch keinen von beiden ausgewählt worden sein können. Dabei muss der jeweilige Algorithmus nicht bis zum Ende durchgelaufen sein; es können also auch partielle Lösungen markiert sein.

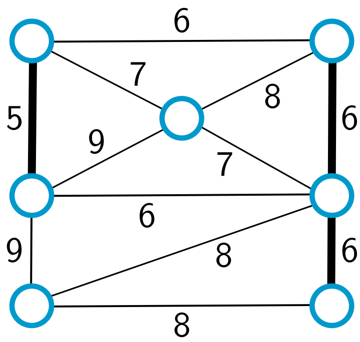
Markieren Sie in jedem Graphen, für den Sie *Prim* (oder *beide*) ausgewählt haben, einen **Startknoten** für den Prim Algorithmus, der zu der dargestellten Kantenauswahl führt.
Markieren Sie in jedem Graphen, für den Sie nicht *Kruskal* (bzw. *keiner*) ausgewählt haben, eine der hervorgehobenen **Kanten**, die in einer partiellen Lösung von Kruskal **nicht** ausgewählt worden wäre.



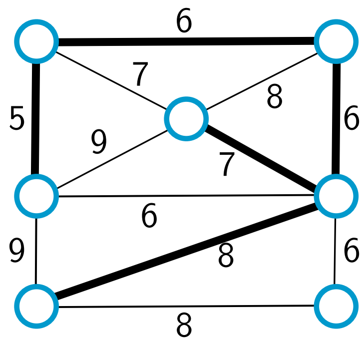
☒ nur Prim ○ beide
○ nur Kruskal ○ keiner



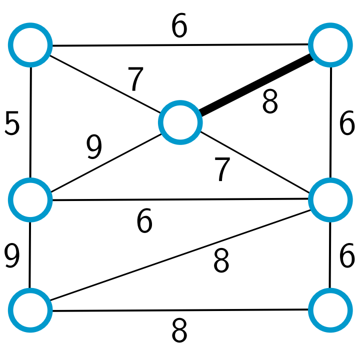
○ nur Prim ☒ beide
○ nur Kruskal ○ keiner



○ nur Prim ○ beide
☒ nur Kruskal ○ keiner



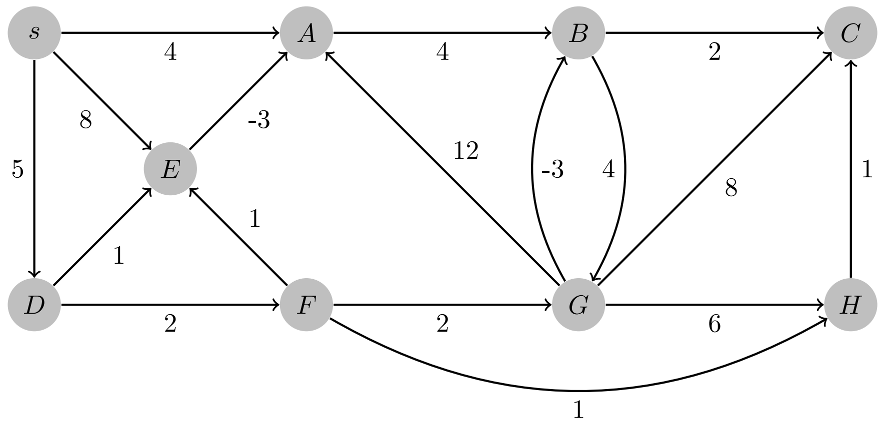
○ nur Prim ○ beide
○ nur Kruskal ☒ keiner



○ nur Prim ○ beide
○ nur Kruskal ☒ keiner

Aufgabe 8: Kürzeste Wege ($9 + 2 + 2 + 1 = 14$ Punkte)

In dieser Aufgabe betrachten wir das Problem der kürzesten Pfade auf dem folgenden Graphen:



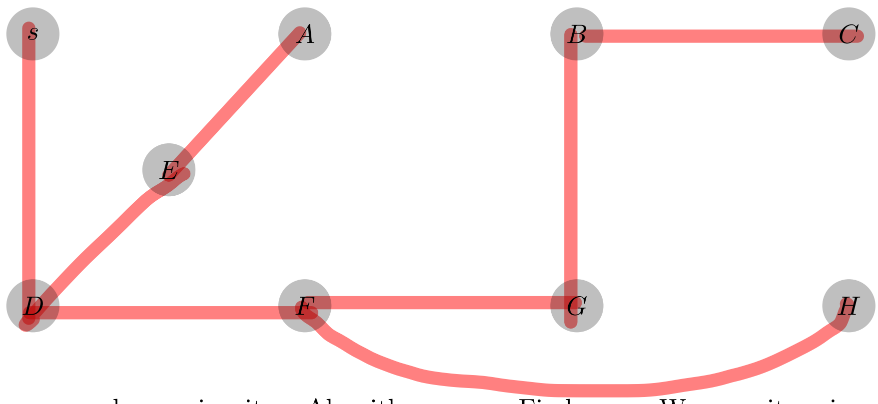
(a) Führen sie den Bellman-Ford-Algorithmus mit Warteschlange aus der Vorlesung zum Finden kürzester Wege auf obigem Graphen aus. Relaxieren Sie dabei die ausgehenden Kanten eines Knotens in alphabetischer Reihenfolge nach den Zielknoten. Der Startknoten für den Algorithmus ist s.

Nutzen Sie für jeden Relaxierungsschritt, bei dem ein Distanzwert *dist* geändert wird, eine Zeile in der Tabelle. Tragen Sie in die zweiten Spalte den *Knoten* ein, der einen neuen Distanzwert erhält, in die dritte Spalte eben diesen neuen *dist*-Wert und in die vierte Spalte den Vorgängerknoten *parent*, also den Knoten, von dem die Relaxierung ausgeht.

In der erste Spalte können Sie die Nummer des Durchlaufs notieren. (Die Eintragungen in der ersten Spalte sind optional. Sie werden nicht bewertet.)

	#	Knoten	dist	parent	
		s	0	–	
	1	A	4	s	
	1	D	5	s	
	1	E	8	s	
A	2	B	8	A	
D	3	E	6	D	
	3	F	7	D	
E	4	A	3	E	
B	5	C	10	B	
	5	G	12	B	
F	6	G	8	F	
	6	H	8	F	
G	7	B	6	G	C → H → F → D → S
	7	C	17	G	
H	8	C	9	H	
	9	C	8	B	

(b) Zeichnen Sie den Baum aller kürzesten Pfade zu dem gegebenen Graphen, der durch die Anwendung des Bellman-Ford Algorithmus entsteht.



(c) In der Vorlesung wurden zwei weitere Algorithmen zum Finden von Wegen mit geringstem Gewicht in Digraphen vorgestellt. Welche von ihnen würden auf diesen Graphen angewendet ein korrektes Ergebnis liefern und welche nicht? Begründen Sie Ihre Antwort.

Dijkstra - Algo: für Digraphen ohne negativen Gewichte

Bellman - Ford - Algo: Digraphen ohne negativen Zyklen

(d) Welches Argument spricht bei Graphen wie dem in dieser Aufgabe für die Verwendung des Bellman-Ford Algorithmus gegenüber anderen anwendbaren Algorithmen?

In diese Aufgaben gibt es negativen Gewichte

Aufgabe 9: Backtracking und Dynamisches Programmieren

(5 + 5 = 10 Punkte)

Eine **Quadratsummendarstellung** ist eine Darstellung einer natürlichen Zahl N als Summe von Quadratzahlen. Ein paar Beispiele:

- Für $N = 6$ sind $1 + 1 + 4$ und $1 + 1 + 1 + 1 + 1 + 1$ Quadratsummendarstellungen.
- Für $N = 17$ sind u. a. $1 + 16$ und $4 + 4 + 9$ Quadratsummendarstellungen.
- Für $N = 25$ sind u. a. 25 und $9 + 16$ und $1 + 1 + 1 + 4 + 9 + 9$ Quadratsummendarstellungen.

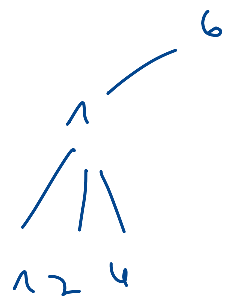
16+4+1+1+1+1

17	18	19	20	21	22	23	24
2	3	4	2	3	4	5	3

(a) Schreiben Sie ein Verfahren mit *Backtracking* als Pseudocode, das zu einer gegebenen Zahl N alle möglichen Quadratsummendarstellungen ausgibt. Für die volle Punktzahl darf immer nur *eine* Permutation der Summanden ausgegeben werden (also z. B. nur $1 + 1 + 4$ und nicht zusätzlich $1 + 4 + 1$ und $4 + 1 + 1$). Wenn mehrere Permutation ausgegeben werden, gibt es einen Punkt Abzug.

Tipp: Bedenken Sie, dass Sie die Summanden speichern müssen. Dazu können Sie ein globales Array *a[]* nutzen. Implementieren Sie dann z. B. eine Prozedur *backtracking(k, sum)*, die die Kandidaten für das Element *a[k]* durchprobiert und dem Backtracking Schema folgt.

```
N
a[]
Prozedure backtracking (k, sum):
    if ( sum = N)   return
    a[k] > a[k-1]
    for i from 1 to √N :
        a[k] += i^2
        if k==0 || a[k] <= N
            backtracking (k+1, sum+a[k])
```



(b) Es soll die kleinste Anzahl von Summanden in einer Quadratsummendarstellung einer Zahl N bestimmt werden. Geben Sie eine rekursive Definition $\text{OPT}(N)$ für diese Aufgabe an, die als Basis für einen Ansatz mit Dynamischer Programmierung geeignet ist.

$$\text{OPT}(N) = \begin{cases} 1 & N = 1 \\ \min_{i \in \{1, 2, \dots, \lfloor \sqrt{N} \rfloor\}} (1 + \text{OPT}(N - i^2)) & \text{sonst} \end{cases}$$

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!