TECHNISCHE UNIVERSITÄT BERLIN
Fakultät IV – Elektrotechnik und Informatik
Fachgebiet Robotik (MAR 5-1)
Dozent: Prof. Dr. Oliver Brock
WiMi: Adrian Pfisterer, Alexander Koenig

Algorithmen und Datenstrukturen, SoSe 25

Klausur am 07. Oktober 2025

Bitte füllen Sie alle folgenden Felder aus:

Vorname _____

Nachname _____

TUB-Kontoname _____

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!

- Die Klausur dauert **90 Minuten**. Insgesamt können in der Klausur **90 Punkte** erreicht werden.
- Legen Sie nun Ihren Personalausweis und Ihren Studierendenausweis neben sich bereit.
- Diese Klausur besteht mit diesem Deckblatt aus den (nummerierten) Seiten **1-24**.
- Geben Sie nur eine Lösung pro Aufgabe ab, streichen Sie alle alternativen Lösungsansätze durch.
- Notieren Sie Ihre Antworten nur auf dem Blatt (inklusive Rückseite), auf dem die zugehörige Aufgabe steht, da die Aufgaben getrennt korrigiert werden.
- Sie brauchen Ihren Namen **nur** auf das Deckblatt zu schreiben. Die restlichen Blätter können über die Klausur-ID zugeordnet werden.
- Schreiben Sie **nicht** mit roter Farbe, grüner Farbe (Korrekturfarben) oder Bleistift. Diese Lösungen werden nicht bewertet!
- Am Klausurende befindet sich eine Doppelseite, 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 und tragen Sie die Anzahl der erhaltenen Zusatzblätter unten ein.
- Falls Sie eine Antwort auf ein Zusatzblatt schreiben, markieren Sie dies klar bei der zugehörigen Aufgabe und auf dem Zusatzblatt.

Anzahl Zusatzblätter:

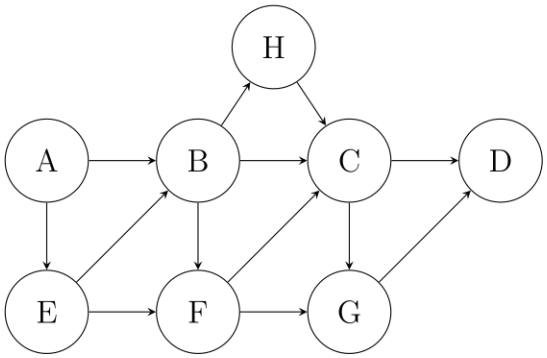
Punktetabelle

Aufgabe	Punkte		
1	11		
2	10		
3	11		
4	11		
5	11		
6	8		
7	8		
8	10		
9	10		
$\sum$	/ 90		

Aufgabe 1: Breitensuche und Tiefensuche (11 Punkte)

(a) (3 Punkte) Der folgende gerichtete Graph ist gegeben. Führen Sie eine Handsimulation der Breitensuche (BFS) und Tiefensuche (DFS) durch und geben Sie die Reihenfolge an, in der die Knoten abgearbeitet werden (**Postorder**-Traversierung für DFS).

Hinweise: Beginnen Sie beide Algorithmen im Knoten A. Gehen Sie in alphabetischer Reihenfolge vor, falls es Wahlmöglichkeiten für benachbarte Knoten gibt.



Breitensuche (BFS):

Tiefensuche (DFS):

(b) (2 Punkte) Handelt es sich bei der folgenden Implementierung um BFS oder DFS? Begründen Sie.

```
1  public void traverse(Graph G, int start) {
2      boolean[] visited = new boolean[G.V()];
3      Queue<Integer> dataStructure = new LinkedList<>();
4
5      dataStructure.add(start);
6      visited[start] = true;
7
8      while (!dataStructure.isEmpty()) {
9          int v = dataStructure.poll();
10         System.out.print(v + " ");
11
12         for (int w : G.adj(v)) {
13             if (!visited[w]) {
14                 visited[w] = true;
15                 dataStructure.add(w);
16             }
17         }
18     }
19 }
```

Antwort:

Begründung:

(c) (1 Punkt) Geben Sie an, welche Zeilen mindestens geändert werden müssten, um den Algorithmus in den jeweils anderen Suchalgorithmus (BFS ↔ DFS) umzuwandeln.

Zu ändernde Zeilen:

(d) (2 Punkte) Entscheiden Sie für jedes Szenario, welcher Algorithmus am besten geeignet ist. Kreuzen Sie das passende Feld an. Pro Aussage ist nur eine Antwort richtig.

Szenario	beide	nur BFS	nur DFS	keiner
Garantiertes Finden eines kürzesten Pfads zwischen zwei Knoten in einem ungewichteten Graphen	☐	☐	☐	☐
Finden aller Knoten in einer Zusammenhangskomponente eines ungerichteten Graphen	☐	☐	☐	☐
Generierung einer topologischen Sortierung in einem azyklischen gerichteten Graphen (DAG)	☐	☐	☐	☐
Navigation in einem Labyrinth mit unendlich tiefen Sackgassen, um den Ausgang zu finden	☐	☐	☐	☐

(e) (3 Punkte) Sie erhalten ein 2D-Grid, das eine Karte darstellt. `1` steht für Land und `0` für Wasser. Eine Insel ist eine Gruppe von horizontal oder vertikal benachbarten Landfeldern, umgeben von Wasser. Ihre Aufgabe ist es, eine Funktion zu implementieren, die die Anzahl der Inseln auf der Karte zählt.

Beispiel mit drei Inseln:

1	1	1	0	0
1	1	0	0	0
0	0	1	0	0
0	0	0	1	1

Implementieren Sie die rekursive Helfermethode `dfs`. Die Methode soll eine gefundene Insel als besucht markieren (indem sie die `1`-Felder auf `2` setzt), um sie nicht doppelt zu zählen. Die Methode `isInBounds` prüft, ob die Koordinaten innerhalb der Grid-Grenzen liegen.

```
public class Solution {
    public int countIslands(int[][] grid) {
        int numIslands = 0;
        for (int r = 0; r < grid.length; r++) {
            for (int c = 0; c < grid[0].length; c++) {
                if (grid[r][c] == 1) {
                    numIslands++;
                    dfs(grid, r, c);
                }
            }
        }
        return numIslands;
    }

    private boolean isInBounds(int[][] grid, int r, int c) {
        return r >= 0 && r < grid.length && c >= 0 && c < grid[0].length;
    }

    private void dfs(int[][] grid, int r, int c) {









    }
}
```


Aufgabe 2: Greedy-Algorithmen (10 Punkte)

(a) (4 Punkte) Beschreiben Sie für jeden der folgenden Algorithmen kurz und präzise die zugrundeliegende **Greedy-Strategie** (**nicht** den gesamten Algorithmus). Erklären Sie, was die **lokal optimale Entscheidung** ist, die der Algorithmus in jedem Schritt trifft, um eine **globale optimale Lösung** zu finden.

1.) **Algorithmus von Prim**

2.) **Algorithmus von Kruskal**

3.) **Algorithmus von Dijkstra**

4.) **A*-Algorithmus (A-Star)**

(b) (6 Punkte) Sie sind für die Programmplanung eines Fernsehsenders zuständig. Für einen Werbeblock mit einer Gesamtlänge **90 Sekunden** sollen Sie möglichst **viele verschiedene Werbespots** unterbringen. Ihnen liegt eine Liste verfügbarer Spots mit **unterschiedlichen Längen** vor, die alle das **gleiche Gewicht** haben.

Ihre Teamleiterin schlägt folgende Greedy-Strategie vor: *„Um sicherzustellen, dass wir unser Zeitfenster von 90 Sekunden auch wirklich **voll ausnutzen**, sollten wir zuerst die Spots wählen, die den **meisten Platz wegnehmen**. Wähle also immer den **längsten** Werbespot aus, der noch in die verbleibende Zeit des Werbeblocks passt“*

Sie sind skeptisch, ob diese Strategie wirklich die Anzahl der Spots maximiert.

1. (3 Punkte) Widerlegen Sie die Strategie Ihrer Teamleiterin durch ein **Gegenbeispiel**. Geben Sie dazu eine Liste von Werbespots (mit Längen in Sekunden) an, für welche die Strategie **„längster Spot zuerst“** versagt.

2. (3 Punkte) Nennen Sie die **optimale Greedy-Strategie** und demonstrieren Sie an Ihrem Gegenbeispiel aus 1., dass diese zu einer besseren Lösung führt. Geben Sie für beide Strategien die **finale Auswahl an Werbespots** und die **Gesamtzahl der gesendeten Spots** an.

Aufgabe 3: Branch and Bound (11 Punkte)

Sie haben heute noch **12 Minuten verbleibende Arbeitszeit**. Sie möchten verschiedene kleine Programmieraufgaben bearbeiten, um Ihren **Verdienst zu maximieren**. Für jede Aufgabe schätzen Sie den Zeitaufwand und die Bezahlung. Die folgende Tabelle zeigt die Details:

Aufgabe	**Bezahlung (€)**	**Dauer (Min)**
A	3	2
B	4	2
C	6	2
D	7	7

Sie verwenden den **Branch and Bound Algorithmus**, um die optimale Aufgabenauswahl zu finden.

(a) (1 Punkt) Auf welches **algorithmische Problem** aus der Vorlesung lässt sich dieses Problem abbilden?

(b) (1 Punkt) Welche Greedy Strategie würden Sie verwenden, um eine möglichst gute **Initiallösung** für dieses Problem zu finden?

(c) (2 Punkte) Berechnen Sie nun die Initiallösung mit Ihrer vorgeschlagenen Strategie und geben sie die **untere Schranke** bei Beginn des Branch and Bound Algorithmus an. Geben sie auch ihren Rechenweg an.

Initiallösung:

Untere Schranke: €

(d) (2 Punkte) Berechnen Sie die **obere Schranke** bei Beginn des Branch and Bound Algorithmus mittels der Lösung des teilbaren Rucksackproblems. Zeigen Sie ihren Rechenweg.

Obere Schranke: €

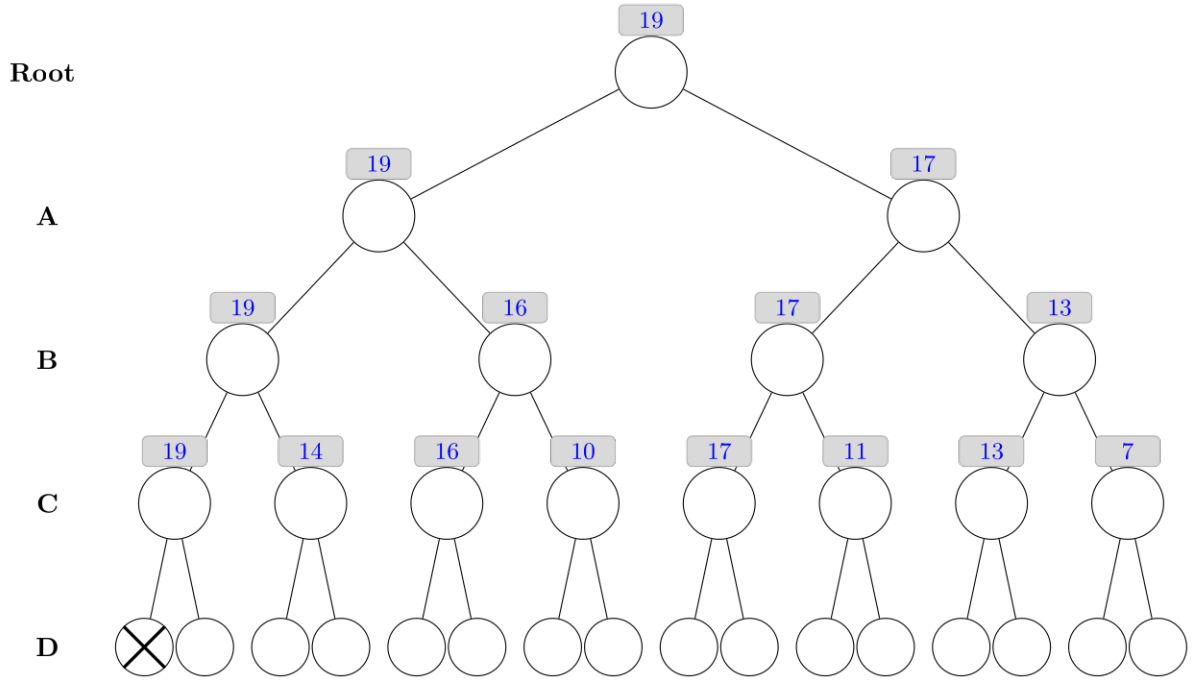
(e) (5 Punkte) Führen Sie den **Branch and Bound Algorithmus** auf dem gegebenen Baum aus.

Hinweise:

- Sie können diese Aufgabe auch ohne die vorherigen Teilaufgaben lösen. Verwenden Sie als initiale untere Schranke 13€ und als initiale obere Schranke 19€ (unabhängig von der Lösung der vorherigen Teilaufgaben). Alle weiteren oberen Schranken (graue hinterlegte Zahlen) sind bereits eingetragen.
- Die Verzweigungsreihenfolge ist A, B, C, D; links bedeutet "Aufgabe bearbeiten", rechts "Aufgabe weglassen".
- Durchgekreuzte Knoten sind wegen der Kapazitätsbeschränkung nicht zulässig.

Anleitung:

- Schreiben Sie den Rückgabewert der rekursiven Branch and Bound Methode in die besuchten Blatt- und Elternknoten.
- Markieren Sie klar, an welchen Stellen Teilbäume abgeschnitten werden.
- Tragen Sie die aktuelle untere Schranke an die besuchten Kanten.
- Geben Sie am Ende die optimale Lösung und ihren Wert in Euro an.



Optimale Lösung:

Erreichter Wert: €

Aufgabe 4: Java Programmierung (11 Punkte)

(a) (6 Punkte) In diesem Code-Snippet soll eine Einkaufswagen-Verwaltung implementiert werden. Der Code enthält jedoch **6 verschiedene Fehler**, die verhindern, dass das Programm korrekt kompiliert oder ausgeführt wird. Geben Sie die Fehler und den korrigierten Java-Code in der Tabelle an.

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  class ShoppingCart {
5      private String customerName;
6      private List<String> items;
7
8      public ShoppingCart(String customerName) {
9          customerName = customerName;
10     }
11
12     public void addItem(String item) {
13         if (item != null & item.length() > 0) {
14             items.add(item);
15         }
16     }
17
18     public String getCartSummary() {
19         String result = "Kunde: " + customerName;
20         for (int i = 0; i <= items.size(); i++) {
21             result += ", Artikel: " + items.get(i);
22         }
23         return result;
24     }
25
26     public boolean containsItem(String item) {
27         for (String cartItem : items) {
28             if (cartItem = item) return true;
29         }
30         return false;
31     }
32 }
33
34 public class ShopSystem {
35     public static void main(String[] args) {
36         ShoppingCart cart = new ShoppingCart("Mueller");
37         cart.addItem("Apfel");
38
39         System.out.println(cart.getCartSummary());
40         System.out.println("Enthaelt Brot: " + cart.containsItem("Brot"));
41         System.out.println("Kunde: " + cart.customerName);
42     }
43 }
```

Zeile	Korrektur

(b) (5 Punkte) Implementieren Sie einen "Shortest Job First"-Scheduler. Verschiedene Prozesse (`Task`) sollen basierend auf ihrer Ausführungszeit (`executionTime`) abgearbeitet werden, wobei Tasks mit der **kürzesten** Zeit zuerst ausgeführt werden sollen.

Hinweis: Nutzen Sie dafür die `java.util.PriorityQueue`. Diese Klasse ist standardmäßig eine **Min-Priority-Queue** und eignet sich daher perfekt für dieses Problem.

Ergänzen Sie den Lückentext auf der nächsten Seite, sodass

- die Klasse `Task` die nötige Schnittstelle implementiert, um von einer `PriorityQueue` anhand der `executionTime` geordnet zu werden.
- eine `PriorityQueue` mit `Task`-Objekten erstellt wird.
- alle Tasks in der korrekten Reihenfolge (kürzeste Ausführungszeit zuerst) aus der Queue entfernt und ihre Namen ausgegeben werden.

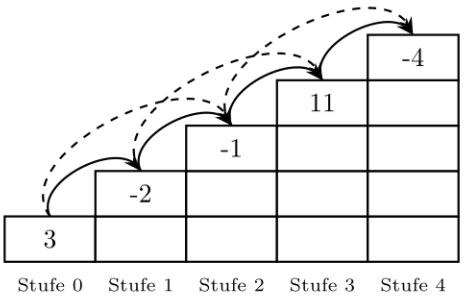
Die folgende Tabelle zeigt die **Dokumentation der Methoden** der `PriorityQueue`.

Methode	**Beschreibung**
`add(Object)`	Fügt das angegebene Element in diese Prioritätswarteschlange ein.
`clear()`	Entfernt alle Elemente aus dieser Prioritätswarteschlange.
`comparator()`	Gibt den Komparator zurück, der zum Sortieren der Elemente in dieser Warteschlange verwendet wird, oder `null`, wenn diese Warteschlange nach der natürlichen Reihenfolge ihrer Elemente sortiert ist.
`contains(Object)`	Gibt `true` zurück, wenn diese Warteschlange das angegebene Element enthält.
`isEmpty()`	Gibt `true` zurück, wenn diese Warteschlange leer ist.
`iterator()`	Gibt einen Iterator über die Elemente in dieser Warteschlange zurück.
`peek()`	Ruft das Kopf-Element dieser Warteschlange ab, ohne es zu entfernen, oder gibt `null` zurück, wenn die Warteschlange leer ist.
`poll()`	Ruft das Kopf-Element dieser Warteschlange ab und entfernt es, oder gibt `null` zurück, wenn die Warteschlange leer ist.
`remove(Object)`	Entfernt eine einzelne Instanz des angegebenen Elements aus dieser Warteschlange, falls vorhanden.
`size()`	Gibt die Anzahl der Elemente in dieser Sammlung zurück.
`toArray()`	Gibt ein Array zurück, das alle Elemente in dieser Warteschlange enthält.
`toArray(T[] a)`	Gibt ein Array zurück, das alle Elemente in dieser Warteschlange enthält; der Laufzeittyp des zurückgegebenen Arrays ist der des angegebenen Arrays.

```java
import java.util.PriorityQueue;
import java.util.Queue;

// Die Task-Klasse vergleichbar machen
class Task ____________________ ____________________<Task> {
    public String name;
    public int executionTime;

    public Task(String name, int executionTime) {
        this.name = name;
        this.executionTime = executionTime;
    }

    @Override
    public int compareTo(Task other) {
        // Vergleich der Ausfuehrungszeiten
        return ____________________________________________________________;
    }
}

public class Scheduler {
    public static void main(String[] args) {
        // PriorityQueue instanziieren und Tasks hinzufuegen
        Queue<Task> taskQueue = ___________________ ____________________;

        taskQueue._______(new Task("Full system scan", 120));
        taskQueue._______(new Task("Compile module", 25));
        taskQueue._______(new Task("Quick backup", 5));

        // Tasks in Prioritaets-Reihenfolge entfernen
        while (_____________________________________) {
            _____________________________________;
            // Die naechste Zeile simuliert die Ausfuehrung des Tasks
            System.out.println("Bearbeite: " + current.name);
        }
    }
}
```


Aufgabe 5: Dynamische Programmierung (11 Punkte)

Ein Roboter steigt eine Treppe mit n Stufen. Er betritt zuerst Stufe 0 und kann dann **pro Zug eine oder zwei Stufen** nach oben steigen. Jede Stufe i hat einen Punktwert $P[i]$ (dargestellt durch ein Java-Array P). Der Roboter erhält die Punkte einer Stufe beim Betreten der Stufe. Ziel ist es, die oberste Stufe $n-1$ zu erreichen und dabei die **maximale Punktzahl zu sammeln**.



(a) (3 Punkte) Die Abbildung zeigt eine Treppe mit $n = 5$ Stufen und $P = [3, -2, -1, 11, -4]$. Tragen Sie in die Tabelle **alle möglichen Pfade** mit den entsprechenden Punktzahlen ein und bestimmen Sie den optimalen Pfad zur **Maximierung der Punktzahl**.

Pfad (Stufenfolge)	**Punktzahl**

Optimaler Pfad:

(b) (3 Punkte) Geben Sie eine **rekursive Funktion** Opt(i) an, die die **maximale Punktzahl** eines Pfades berechnet, der bei Stufe 0 beginnt und bei Stufe i endet. Verwenden Sie eine mathematische Schreibweise und keinen Java- oder Pseudo-Code.

Hinweis: Geben Sie die allgemeine rekursive Funktion an, unabhängig von den im vorherigen Aufgabenteil gegebenen Werten.

(c) (2 Punkte) Geben Sie die Laufzeitkomplexität für die Berechnung der **maximal erreichbaren Punktzahl** mit dynamischer Programmierung an. **Vergleichen** Sie diese mit der Laufzeit einer Brute-Force-Lösung und begründen Sie beide Ergebnisse kurz.

(d) (3 Punkte) Implementieren Sie eine **rekursive** Methode `calculateMaxPoints(int i, int[] P)`, die den Top-Down-Ansatz der dynamischen Programmierung verwendet, um die **maximale Punktzahl** für Stufe i zu berechnen. Die Methode soll bereits berechnete Ergebnisse speichern (Memoization). Sie dürfen die Java-Klasse `Math` verwenden.

```
public class StepClimber {
    private Integer[] memo;

    public StepClimber(int n) {
        // Alle Elemente werden mit `null` initialisiert
        memo = new Integer[n];
    }

    public int calculateMaxPoints(int i, int[] P) {
        // Implementieren Sie hier Ihre Loesung









    }
}
```

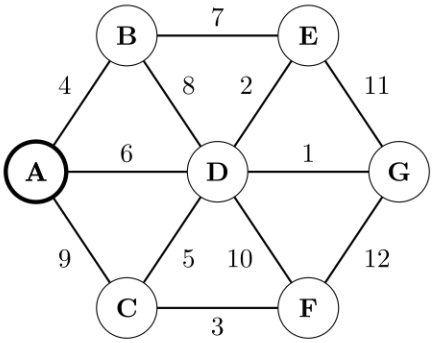

Aufgabe 6: Minimale Spannbäume (8 Punkte)

(a) (3 Punkte) Lesen Sie die folgenden Aussagen. Entscheiden Sie, ob sie nur für den Prim-Algorithmus, nur für den Kruskal-Algorithmus, für beide oder für keinen der beiden Algorithmen gelten. Kreuzen Sie das passende Feld an. Pro Aussage ist nur eine Antwort richtig.

Aussage	beide	nur Prim	nur Kruskal	keiner
Der Algorithmus kann korrekt auf einem gerichteten Graphen ausgeführt werden, sofern alle Kantengewichte positiv sind.	☐	☐	☐	☐
Der Algorithmus kann so implementiert werden, dass er ohne expliziten Vergleich der Kantengewichte korrekt arbeitet, wenn diese bereits in aufsteigender Reihenfolge gegeben sind.	☐	☐	☐	☐
Der Algorithmus kann bei geschickter Implementierung auch in einem Graphen mit negativen Kantengewichten korrekt einen MST berechnen.	☐	☐	☐	☐
Der Algorithmus kann so implementiert werden, dass sein Speicherbedarf asymptotisch nur von der Knotenzahl V abhängt, nicht von der Kantenzahl E.	☐	☐	☐	☐
Die Reverse-Delete Variante dieses Algorithmus entfernt explizit Kanten aus dem Graphen.	☐	☐	☐	☐
Der Algorithmus eignet sich besser für eine parallele Implementierung.	☐	☐	☐	☐

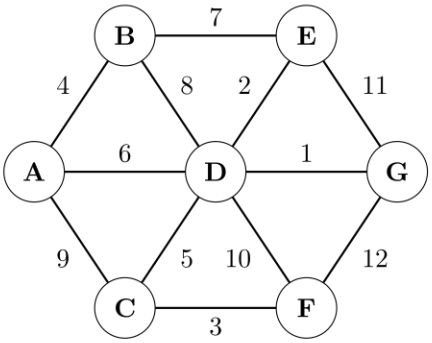
(b) (4 Punkte) Führen Sie den Prim-Algorithmus und den Kruskal-Algorithmus auf den folgenden Graphen aus. Markieren Sie die ausgewählten Kanten direkt in den Graphen und geben Sie in den Tabellen die Reihenfolge an, in der die Kanten zum minimalen Spannbaum hinzugefügt werden.

Prim: Startknoten ist A.



Schritt	Hinzugefügte Kante
1	
2	
3	
4	
5	
6	

Kruskal:



Schritt	Hinzugefügte Kante
1	
2	
3	
4	
5	
6	

(c) (1 Punkt) Sind die gefundenen minimalen Spannbäume aus Teil (b) identisch oder nicht? Begründen Sie kurz warum sie identisch oder nicht identisch sind.

Aufgabe 7: Dijkstra und Bellman-Ford (8 Punkte)

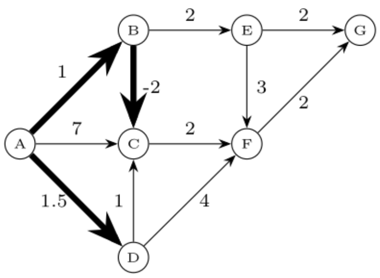
(a) (4 Punkte) Gegeben ist der folgende Graph. Die vier Abbildungen zeigen jeweils einen möglichen Zustand *während der Ausführung* eines Algorithmus zur Bestimmung kürzester Wege mit Startknoten A. Die *dick gezeichneten Kanten* stellen den *bisher konstruierten Kürzeste-Wege-Baum dar.*

Kreuzen Sie an, welcher Algorithmus (nur Dijkstra, nur Bellman-Ford, beide oder keiner) den gezeigten Zustand erzeugt haben könnte.

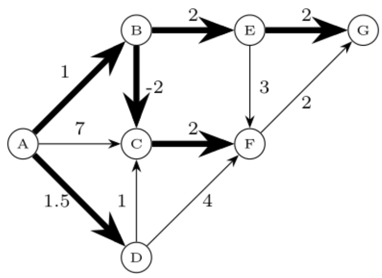
(b) (2 Punkte) Markieren Sie direkt im Graphen durch Umkreisen des Kantengewichts. Die konkrete Markierung hängt von Ihrer Antwort in Teil a) ab:

- Falls Sie **Dijkstra** angekreuzt haben: Umkreisen Sie das Gewicht der Kante, die Dijkstra als **Nächstes** dem Baum hinzufügen würde.
- Falls Sie **Bellman-Ford** angekreuzt haben: Umkreisen Sie das Gewicht der Kante, deren Relaxation als **Nächstes** zu einer Aktualisierung des Kürzeste-Wege-Baums führen würde. (Gehen Sie von einer alphabetischen Abarbeitung der Knoten und Kanten aus).
- Falls Sie **beide** angekreuzt haben: Führen Sie beide oben genannten Markierungen durch. Kennzeichnen Sie eindeutig, welche Markierung zu welchem Algorithmus gehört (z. B. mit „D“ und „BF“).
- Falls Sie **keiner** angekreuzt haben: Umkreisen Sie das Gewicht einer der dick gezeichneten Kanten, die einen Widerspruch zur Funktionsweise beider Algorithmen darstellt und begründen Sie Ihre Wahl stichpunktartig.

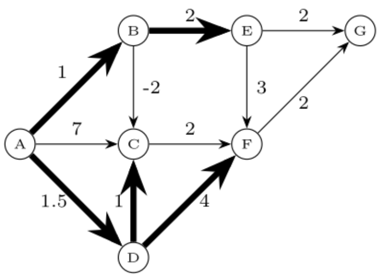
Hinweis: Falls Sie eine Markierung korrigieren, streichen Sie die alte eindeutig durch. Eine lesbare Legende ist bei Abweichungen zulässig.



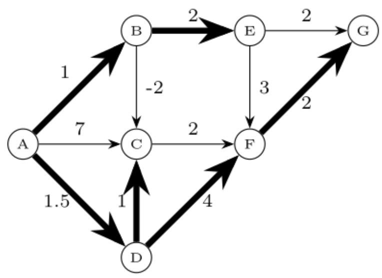
○ nur Dijkstra ○ beide
○ nur Bellman-Ford ○ keiner



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



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



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

(c) (1 Punkt) Nehmen Sie nun an, dem Graphen wird eine **neue** Kante hinzugefügt: eine Kante von Knoten **C** nach Knoten **A** mit dem Gewicht **0**.

Erklären Sie kurz und präzise, welches **fundamentale Problem** für die Suche nach einem kürzesten Pfad durch diese neue Kante entsteht. Nennen Sie den **problematischen Pfad** und sein **Gesamtgewicht**.

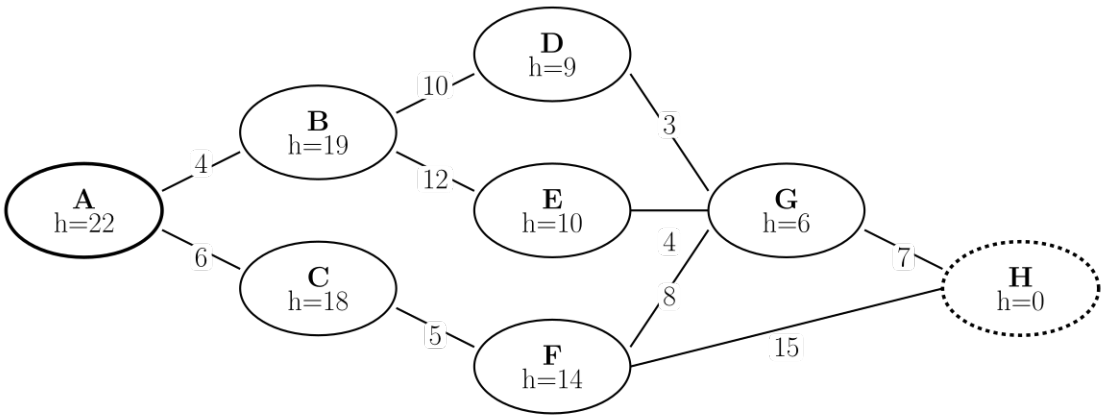
(d) (1 Punkt) Der gegebene Graph hat sieben Knoten. Beschreiben Sie, wie der Bellman-Ford-Algorithmus das in der vorigen Teilaufgabe genannte Problem in der **siebten Iteration** erkennt.

Aufgabe 8: Heuristische und randomisierte Algorithmen (10 Punkte)

(a) (5 Punkte) Sie sind für die Logistikplanung eines Online-Versandhändlers verantwortlich. Ihre Aufgabe ist es, die schnellste Lieferroute zwischen zwei Verteilzentren zu finden.

Gegeben ist ein Graph, der die Verteilzentren (Knoten) und die Transportwege (Kanten) darstellt. Die Kantengewichte entsprechen der Lieferzeit in Stunden. Die heuristischen h-Werte schätzen die verbleibende Lieferzeit bis zum Ziel-Verteilzentrum **H**.

Ihre Aufgabe ist es, den A*-Algorithmus auszuführen, um den schnellsten Weg von **Verteilzentrum A** nach **Verteilzentrum H** zu finden.



Hinweis zur Erinnerung:

- $g(n)$-Wert: Die Kosten des bisher kürzesten bekannten Pfades vom Startknoten zum Knoten n.
- $h(n)$-Wert: Eine Schätzung/Heuristik der Kosten vom Knoten n zum Ziel.
- $f(n)$-Wert: Eine Schätzung der Gesamtkosten vom Start zum Ziel über n. Formel: $f(n) = g(n) + h(n)$.

Protokollieren Sie die Ausführung des Algorithmus, indem Sie die folgenden zwei Tabellen vervollständigen.

1. Entwicklung der Priority-Queue (PQ)

Notieren Sie den Zustand der Priority-Queue (`PQ`) **vor** Beginn jeder Iteration. Tragen Sie die Knoten, mit Komma getrennt, ein. Geben Sie den f-Wert eines Knotens in der `PQ` in Klammern an. Zum Beispiel: A(22), C(24).

Regel: Sortieren Sie die `PQ` nach aufsteigenden f-Werten. Bei gleichen f-Werten sortieren Sie alphabetisch.

Iteration	`PQ` vor der Iteration
1	A(22)
2	
3	
4	
5	
6	

2. Untersuchte Knoten und g-Werte

Geben Sie in jeder Iteration den aktuell aus der `PQ` entnommenen und untersuchten Knoten an. Tragen Sie für dessen Nachbarn die **neu berechneten oder verbesserten** g-Werte in die jeweilige Spalte ein. Bereits ermittelte und nicht verbesserte g-Werte müssen nicht erneut eingetragen werden.

Iter.	Knot.	$g(A)$	$g(B)$	$g(C)$	$g(D)$	$g(E)$	$g(F)$	$g(G)$	$g(H)$
0	/	0	∞	∞	∞	∞	∞	∞	∞
1	A								
2									
3									
4									
5									
6									

3. Ergebnis

Geben Sie die von Ihnen gefundene schnellste Route als Sequenz von Knoten an:

(b) (3 Punkte) Betrachten Sie ein unendlich großes 2D Gitter, in dem man sich von einer Zelle zu jeder benachbarten Zelle (horizontal oder vertikal, **nicht** diagonal) bewegen kann. Jede Bewegung **kostet 1**. Das Ziel ist es, einen **kürzesten Weg** vom Startknoten S zum Zielknoten G zu finden. Für einen beliebigen Knoten n mit Koordinaten (x_n, y_n) und den Zielknoten G mit Koordinaten (x_G, y_G) werden die folgenden zwei Heuristiken zur Schätzung der verbleibenden Kosten vorgeschlagen:

- **Heuristik 1 (Manhattan-Distanz):** $h_1(n) = |x_n - x_G| + |y_n - y_G|$
- **Heuristik 2 (Euklidische Distanz):** $h_2(n) = \sqrt{(x_n - x_G)^2 + (y_n - y_G)^2}$

Prüfen Sie für **beide** Heuristiken (h_1 und h_2), ob sie für das beschriebene Problem **zulässig** (*admissible*) sind. **Begründen** Sie Ihre jeweilige Antwort kurz. Ein formeller Beweis ist nicht erforderlich.

(c) (2 Punkte) Ordnen Sie die folgenden Algorithmen einer der folgenden Klassen zu:

- **Approximativer Algorithmus (Approx. Alg.)**
- **Deterministischer Algorithmus (Determ. Alg.)**
- **Las-Vegas (LV)**
- **Monte-Carlo (MC)**

Kreuzen Sie für jeden Algorithmus die zutreffende Klasse in der Tabelle an.

Algorithmus	**Approx. Alg.**	**Determ. Alg.**	**LV**	**MC**
Quicksort mit zufälligem Pivot-Element	☐	☐	☐	☐
Zur Approximation eines Funktionsminimums werden exakt 10 Iterationen einer zufälligen lokalen Suche durchgeführt. Das beste bisher gefundene Ergebnis wird zurückgegeben.	☐	☐	☐	☐
Aus einem Sack mit 100 unsortierten Socken wird eine zufällige Teilmenge von 20 Socken entnommen. Aus dieser Teilmenge wird der Socken ausgewählt, der einem gesuchten Muster am besten entspricht.	☐	☐	☐	☐
Aus einem Sack unsortierter Socken werden nacheinander einzelne Socken gezogen und geprüft, bis der exakt passende Socken für einen gegebenen Einzelsocken gefunden wurde.	☐	☐	☐	☐

Aufgabe 9: Hashing (10 Punkte)

(a) (4 Punkte) Sie evaluieren verschiedene Hashing-Strategien. Sie untersuchen das Verhalten für eine Hashtabelle der Größe $M = 7$ unter Verwendung der **Divisions-Rest-Methode** als Hashfunktion, also:

$$h(k) = k \mod 7.$$

Die folgende Sequenz von Schlüsseln soll in der angegebenen Reihenfolge in eine anfangs leere Tabelle eingefügt werden:

`[2, 5, 9, 12, 16, 19, 23].`

Simulieren Sie das Einfügen der obigen Schlüssel für zwei verschiedene Kollisionsstrategien. Stellen Sie den **finalen** Zustand der Hashtabelle nach dem Einfügen aller sieben Schlüssel für beide Verfahren dar.

1. **Hashing mit Verkettung (Separate Chaining):** Neue Schlüssel werden stets am **Anfang** der jeweiligen Liste eingefügt.
2. **Lineare Sondierung (Linear Probing):** Nutzen Sie die folgende Sondierungsfolge: $(h(k) + n) \pmod M$ für $n = 0, 1, 2, \ldots$.

Füllen Sie dazu die beiden untenstehenden Tabellen aus.

1. Hashing mit Verkettung

Hinweis: Stellen Sie die Liste dar, indem Sie die Schlüssel mit Pfeilen verbinden (z.B. k1 → k2)

Index	Verkettete Liste (Schlüssel)
0	
1	
2	
3	
4	
5	
6	

2. Lineare Sondierung

Index	Schlüssel
0	
1	
2	
3	
4	
5	
6	

(b) (2 Punkte) Betrachten Sie Ihr Ergebnis der **Linearen Sondierung** aus Teil a). Das dort sichtbare Phänomen der primären Häufung entsteht, weil auch Schlüssel mit *unterschiedlichen* Hash-Werten um dieselben freien Plätze konkurrieren.

Erklären Sie diesen Effekt kurz am Beispiel der Einfügung von Schlüssel `16` und `19`.

(c) (2 Punkte) Der Belegungsfaktor beider Hashtabellen in a) beträgt nach dem Einfügen aller Schlüssel $\alpha = 1$.

Vergleichen Sie kurz die Konsequenz dieses Zustands für eine **erfolglose Suche** bei beiden Verfahren (**lineare Sondierung** und **Verkettung**). Welches Verfahren ist **robuster** gegenüber diesem hohen Belegungsgrad und warum?

(d) (2 Punkte) Kreuzen Sie an, ob die folgenden Aussagen wahr oder falsch sind. Eine Begründung ist nicht erforderlich.

Aussage	**Wahr**	**Falsch**
Quadratisches Sondieren vermeidet primäre Häufungen, die bei linearem Sondieren auftreten. Das Problem der sekundären Häufung bleibt jedoch bestehen, da Schlüssel mit demselben initialen Hashwert stets derselben Sondierungssequenz folgen.	☐	☐
Bei der Divisions-Rest-Methode ($h(k) = k \mod M$) ist die Wahl von M als eine Zweierpotenz (z.B. $M = 64$) besonders empfehlenswert, da hierbei alle Bits des Schlüssels für die Berechnung der Hashadresse berücksichtigt werden, was zu einer guten Streuung führt.	☐	☐
Beim quadratischen Sondieren ist, im Gegensatz zur linearen Sondierung, nicht immer garantiert, dass die Sondierungsfolge alle (M) Adressen der Tabelle durchläuft. Daher kann die Suche nach einem freien Platz fehlschlagen, obwohl die Tabelle noch nicht voll ist ($\alpha < 1$).	☐	☐
Gemäß dem Vertrag in Java gilt: Wenn für zwei Objekte `a` und `b` die Bedingung `a.equals(b)` zutrifft, dann *muss* auch `a.hashCode() == b.hashCode()` gelten. Die Umkehrung, also dass aus gleichen Hashcodes die Gleichheit der Objekte folgt, ist hingegen nicht zwingend erforderlich.	☐	☐

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!