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 26

Klausur am 20. Juli 2026

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 **1-23**.
- 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

Nr.	**Seite**	**Aufgabe**	**Punkte**	
1	3	Java-Programmierung	13	
2	6	Tiefensuche	10	
3	8	Mini-Max und Alpha-Beta-Suche	8	
4	9	Flussgraphen und Edmonds–Karp	6	
5	10	Prim, Kruskal und Graphdarstellungen	10	
6	12	Heuristische, approximative und randomisierte Algorithmen	10	
7	15	Bellman-Ford-Algorithmus	6	
8	16	Hashing	6	
9	18	Dynamische Programmierung	10	
10	20	Greedy Algorithmen	11	
$\sum$			/ 90	

Aufgabe 1: Java-Programmierung (13 Punkte)

(a) (3 Punkte) Vervollständige die Klasse `Product`. Die Schnittstelle `Sellable` ist vorgegeben.

- `Product` soll die Schnittstelle `Sellable` implementieren. Das `protected`-Attribut `stock` soll die aktuelle Anzahl an Produkten im Lager speichern.
- Ergänze die Methodendefinition und die fehlenden Zeilen in `sell()`: Wenn noch mindestens ein Produkt im Lager ist, verkauft die Methode genau ein Stück. Die Methode `sell()` gibt `true` zurück, wenn der Verkauf geglückt ist, sonst `false`.

```
1  public interface Sellable {
2      public boolean sell();
3  }
4
5  public class Product ____________________ ____________________ {
6      _____________ _____________ _____________;
7
8      public Product(int stock) {
9          ______._____________ = _____________;
10     }
11
12     @Override
13     _____________ _____________ sell() {
14         if (_____________________________) {
15             _____________________________;
16             return true;
17         }
18         return false;
19     }
20 }
```


(b) (2.5 Punkte) Erstelle nun eine Klasse `PerishableProduct`, welche von `Product` erbt. Die Klasse soll zusätzlich das `private`-Attribut `bestBefore` (`String`) für das Ablaufdatum des Produkts enthalten.

```
1 public class PerishableProduct _____ _____ {
2     _____ _____ _____;
3
4     public PerishableProduct(int stock, String bestBefore) {
5         _____(_____);
6         _____._____ = _____;
7     }
8 }
```

(c) (2.5 Punkte) Erstelle nun eine **abstrakte** Basisklasse `Shop` mit dem `protected`-Attribut `products`. Das Attribut ist eine `LinkedList` von `Product`-Objekten. Die Liste soll anfangs leer sein. Mit der Funktion `addProduct` soll ein Produkt hinzugefügt werden können.

```
1 import java.util.LinkedList;
2
3 public _____ class Shop {
4     _____;
5
6     public void addProduct(Product product) {
7         _____;
8     }
9 }
```


(d) (3 Punkte) Die Klasse `StockCounter` befindet sich im selben Paket wie `Product`. Die Funktion `totalStock` erhält eine bereits initialisierte Liste `products`. Die Funktion soll zurückgeben, wie viele Produkte insgesamt im Lager sind, ohne die Liste zu verändern.

Der Code enthält 4 fehlerhafte Stellen. Finde sie und korrigiere sie direkt im Code.

```
1  import java.util.LinkedList;
2
3  public class StockCounter {
4      public int totalStock(LinkedList<Product> products) {
5          int sum = 0;
6          for (Product p : products) {
7              products.remove(p);
8              int stock = products.size();
9              sum++;
10         }
11         return;
12     }
13 }
```

(e) (2 Punkte) Wähle für jede Frage die passende Antwort aus.

Welche Komponente gibt den Speicher eines nicht mehr erreichbaren Objekts frei?

- ☐ Der Compiler
- ☐ Der Garbage Collector
- ☐ Der Konstruktor
- ☐ Der Quelltext

Was bedeutet `private` bei einem Attribut?

- ☐ Das Attribut ist außerhalb der deklarierenden Klasse nicht direkt zugreifbar.
- ☐ Das Attribut ist in allen Unterklassen direkt zugreifbar.
- ☐ Das Attribut ist im ganzen Programm direkt zugreifbar.
- ☐ Das Attribut ist nicht veränderbar.

Was gilt für `static`-Methoden?

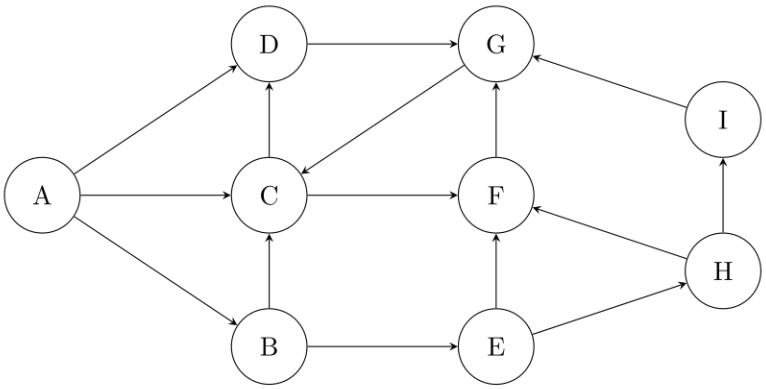
- ☐ Die Methode darf nur einmal aufgerufen werden.
- ☐ Die Methode muss ein Objekt verändern.
- ☐ Sie können aufgerufen werden, ohne vorher ein Objekt der Klasse erzeugen zu müssen.
- ☐ Die Methode startet automatisch beim Programmstart.

Wie werden Parameter in Java immer übergeben?

- ☐ Call by Name
- ☐ Call by Reference
- ☐ Call by Result
- ☐ Call by Value

Aufgabe 2: Tiefensuche (10 Punkte)

(a) (5 Punkte) Gegeben ist ein gerichteter und ungewichteter Graph. Führe die rekursive Tiefensuche (DFS) auf diesem Graphen durch. Beginne im Knoten A. Wenn es mehrere noch nicht besuchte Nachbarn gibt, wähle sie in alphabetischer Reihenfolge.



Pre-Order: Gib die Nebenreihenfolge an.

Post-Order: Gib die Hauptreihenfolge an.

(b) (2 Punkte) Ist auf dem gegebenen Graphen eine topologische Sortierung möglich? Begründe deine Entscheidung in 1–2 Sätzen.

Falls ja, gib eine mögliche topologische Sortierung an. Falls nein, gib eine Kante an, die entfernt werden kann, damit eine topologische Sortierung möglich wird.

(c) (1 Punkt) Welche Kante kannst du aus dem gegebenen Graphen entfernen, damit der Knoten G vor dem Knoten D in der Nebenreihenfolge steht? Falls mehrere Kanten möglich sind, reicht es, eine Kante zu nennen.

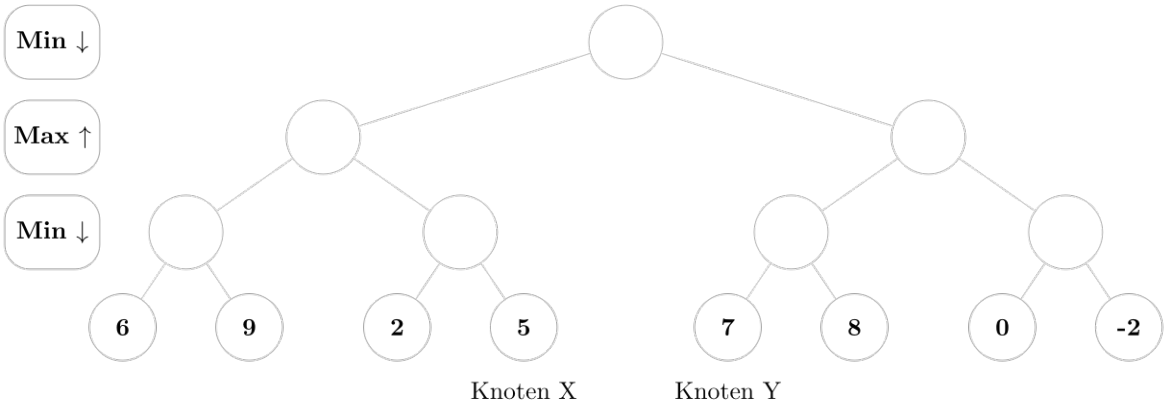
(d) (2 Punkte) Angenommen, du startest in einem beliebigen Knoten eines ungewichteten Graphen eine **Breitensuche** und erhältst den zugehörigen BFS-Baum. Kannst du aus diesem BFS-Baum immer einen kürzesten Pfad zwischen zwei beliebigen Knoten des Graphen ablesen?

Begründe deine Antwort kurz in 1–2 Sätzen oder gib ein Gegenbeispiel mit höchstens 5 Knoten an.

Hinweis: Ein BFS-Baum enthält für jeden erreichten Knoten die Kante, über die er bei der Breitensuche erstmals entdeckt wurde.

Aufgabe 3: Mini-Max und Alpha-Beta-Suche (8 Punkte)

(a) (1 Punkt) Führe auf dem Graphen den MiniMax-Algorithmus aus und trage die fehlenden Werte in die Knoten ein. Spieler Min minimiert den Wert, Spieler Max maximiert den Wert.

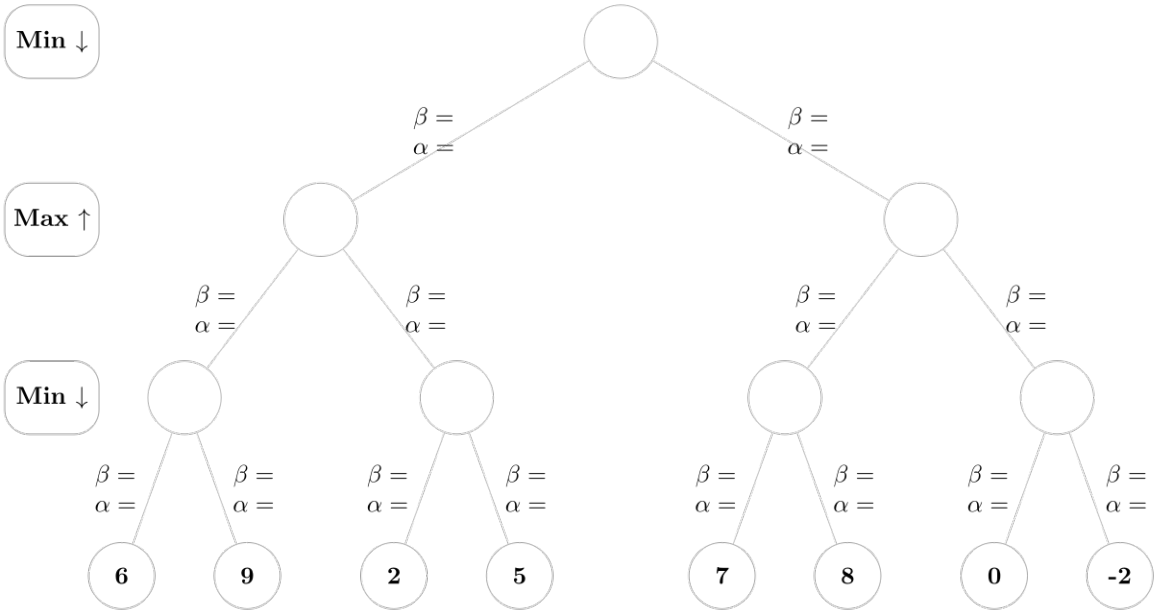


(b) (2 Punkte) Können Knoten X oder Knoten Y den Wurzelwert verändern, wenn an ihnen andere Werte stehen würden? Falls ja, welche Werte müssten an diesen Knoten stehen, damit sich der Wert an der Wurzel ändert? Falls nein, begründe deine Antwort kurz.

Knoten X

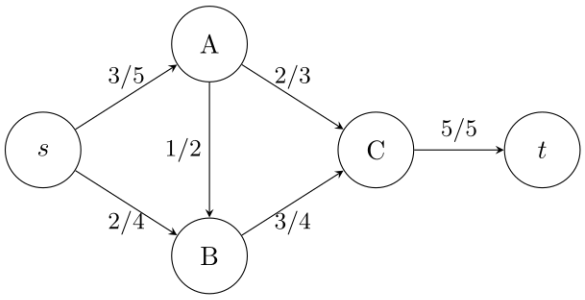
Knoten Y

(c) (5 Punkte) Führe auf der folgenden Kopie des Graphen die Alpha-Beta-Suche aus. Werte die Kindknoten immer von links nach rechts aus. Trage an jeder besuchten Kante die aktuellen Schranken beim Aufruf des Kindknotens ein. Trage außerdem die zurückgegebenen Werte in die besuchten inneren Knoten ein. Markiere alle Schnitte und benenne jeden Schnitt als α-Cut oder β-Cut.



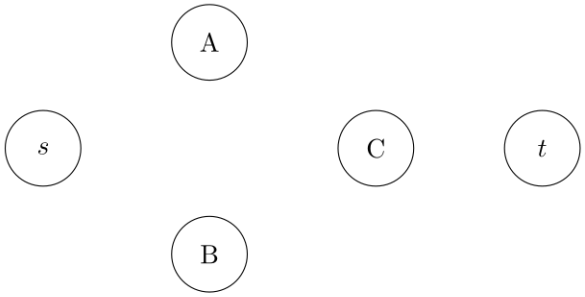
Aufgabe 4: Flussgraphen und Edmonds–Karp (6 Punkte)

(a) (1 Punkt) Der folgende Flussgraph mit Quelle s und Senke t ist gegeben. Jede Kante ist mit f/c beschriftet. Dabei ist f der Fluss auf der Kante und c die Kapazität der Kante.



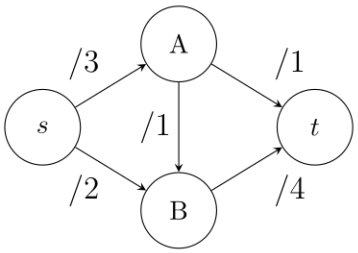
Bestimme den Wert des Flusses.

(b) (2 Punkte) Zeichne den Restflussgraph zum Flussgraphen aus Teilaufgabe (a). Die Knoten sind schon vorgegeben. Zeichne nur Restkanten mit positiver Restkapazität und beschrifte jede gezeichnete Restkante mit ihrer Restkapazität.

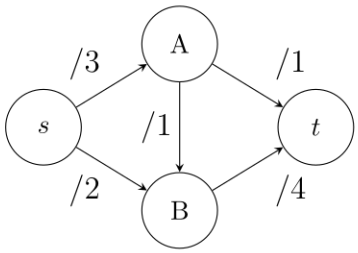


(c) (3 Punkte) Wende Edmonds–Karp auf den folgenden Flussgraphen an. Die Kanten wurden bereits mit Fluss 0 initialisiert. Fülle die drei Augmentierungen aus. Zeichne in jeder Kopie den jeweils gewählten augmentierenden Pfad deutlich ein. Falls es mehrere kürzeste augmentierende Pfade gibt, darf ein beliebiger davon gewählt werden. Trage links vom Schrägstrich die Flüsse ein, die nach dieser Augmentierung gelten.

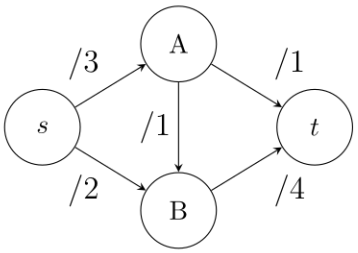
Iteration 1



Iteration 2



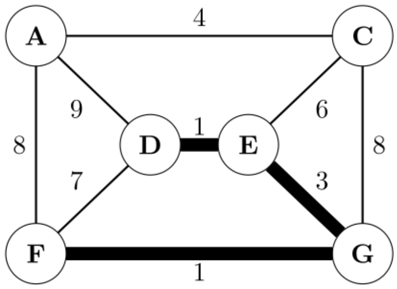
Iteration 3



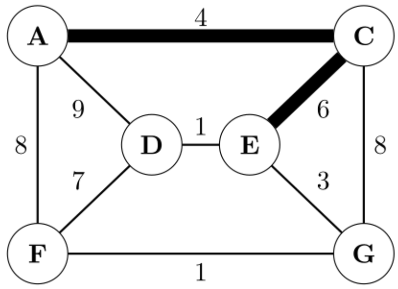
Aufgabe 5: Prim, Kruskal und Graphdarstellungen (10 Punkte)

(a) (4 Punkte) Entscheide für jeden Graphen, ob der markierte Zwischenstand von Prim, Kruskal, beiden oder keinem der Algorithmen stammen kann. Kreuze die richtige Antwort an.

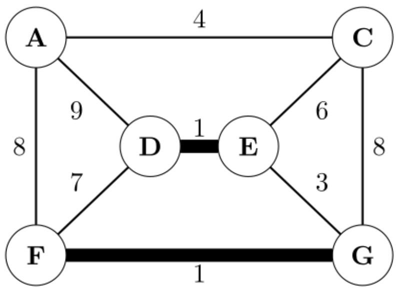
Hinweis: Prim kann hier von einem beliebigen Startknoten aus beginnen.



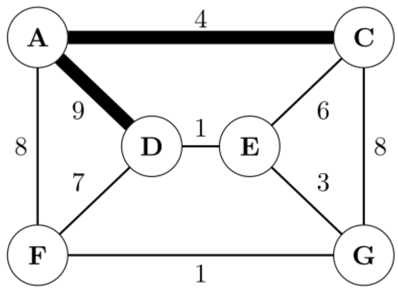
☐ beide ☐ nur Kruskal
☐ nur Prim ☐ keiner



☐ beide ☐ nur Kruskal
☐ nur Prim ☐ keiner



☐ beide ☐ nur Kruskal
☐ nur Prim ☐ keiner



☐ beide ☐ nur Kruskal
☐ nur Prim ☐ keiner

(b) (2 Punkte) Der folgende Pseudocode soll mit Kruskals Algorithmus einen **maximalen** Spannbaum bestimmen und seine Kanten zurückgeben. Eine Kante wird als (u, v, w) beschrieben: u und v sind die beiden Knoten, w ist das Gewicht. Die Union-Find-Struktur speichert, welche Knoten bereits in derselben Zusammenhangskomponente liegen.

Ergänze Zeile 2 und Zeile 6.

```
1  function maxSpanningTree(n, edges)
2
3      uf ← UnionFind(n)
4      tree ← empty list
5      for each edge (u, v, w) in edges do
6
7              uf.union(u, v)   // verbindet die Komponenten von u und v
8              add (u, v, w) to tree
9      return tree
```

(c) (2 Punkte) Für MST-Algorithmen werden Graphen oft als Adjazenzlisten gespeichert. Folgende Adjazenzlisten beschreiben einen ungerichteten, gewichteten Graphen.

```
A → (B, 2) → (C, 5)
B → (A, 2) → (C, 1)
C → (A, 5) → (B, 1) → (D, 3)
D → (C, 3)
```

Zeichne die in den Adjazenzlisten kodierten Kanten mit ihren Gewichten ein.

(d) (2 Punkte) Beantworte die folgenden Fragen. Entscheide jeweils, ob die Antwort beide Datenstrukturen, nur Adjazenzlisten, nur Adjazenzmatrizen oder keine der beiden Datenstrukturen ist.

Frage	**beide**	**nur Adjazenz-listen**	**nur Adjazenz-matrizen**	**keine**
Welche Datenstruktur kann gerichtete Graphen darstellen?	☐	☐	☐	☐
Welche Datenstruktur benötigt typischerweise Speicher proportional zu $\|V\| + \|E\|$?	☐	☐	☐	☐
Welche Datenstruktur ist bei sehr großen Graphen mit vergleichsweise wenigen Kanten typischerweise speichereffizienter?	☐	☐	☐	☐
Welche Datenstruktur erlaubt es, in konstanter Zeit zu prüfen, ob zwischen zwei gegebenen Knoten eine Kante existiert?	☐	☐	☐	☐

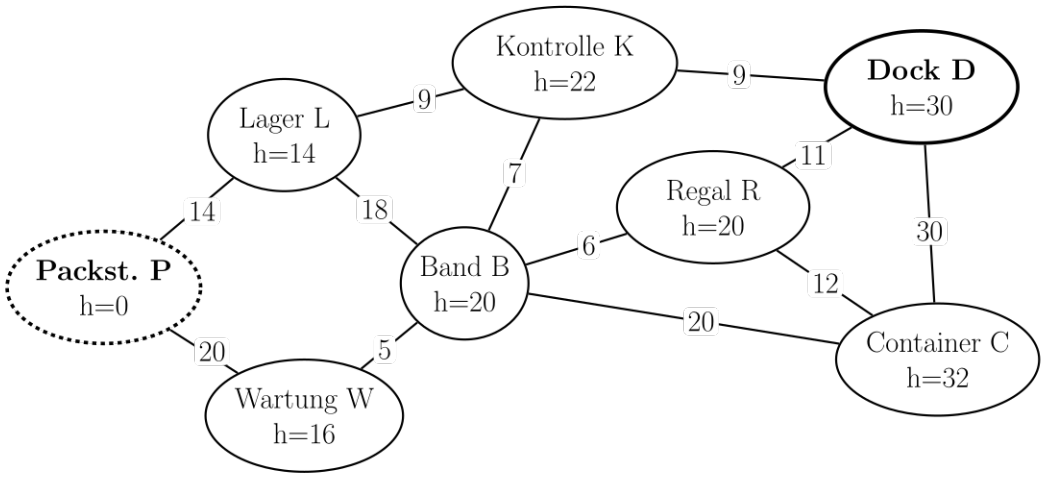
Aufgabe 6: Heuristische, approximative und randomisierte Algorithmen (10 Punkte)

(a) (8 Punkte) Gegeben ist der folgende Graph, der Wegpunkte in einem Lager darstellt. Die Kantengewichte geben die Fahrzeit eines Roboters in Sekunden an. Der Roboter darf entlang der Kanten in beide Richtungen fahren. Die angegebenen h-Werte schätzen die verbleibende Fahrzeit bis zur Packstation P.

Führe den A*-Algorithmus aus, um den schnellsten Weg vom Dock D zur Packstation P zu bestimmen.

Verwende dabei folgende Konventionen:

- Jeder Knoten kommt höchstens einmal in der Priority-Queue PQ vor. Wird sein g-Wert verbessert, wird sein Eintrag in der PQ aktualisiert.
- Bereits aus der PQ entnommene Knoten werden nicht erneut eingefügt.
- Die PQ wird nach aufsteigenden f-Werten sortiert. Bei gleichen f-Werten wird alphabetisch sortiert.
- Sobald P aus der PQ entnommen wird, endet der Algorithmus.



Notiere vor jeder Iteration den Inhalt der PQ in der Tabelle. Gib für jeden Knoten den aktuellen f-Wert in Klammern an.

Iteration	PQ vor der Iteration
1	D(30)
2	
3	
4	
5	

Trage anschließend für jede Iteration den entnommenen Knoten sowie alle in dieser Iteration erstmals gesetzten oder verbesserten g-Werte in die Tabelle ein. Lass alle anderen Felder leer.

Iteration	Knoten	D	C	R	K	B	L	W	P
0	/	0	∞	∞	∞	∞	∞	∞	∞
1	D								
2									
3									
4									
5									

Gib abschließend die kürzeste Route an:

(b) (2 Punkte) Wähle für jede Frage genau eine passende Antwort aus.

Welche Aussage beschreibt eine typische Eigenschaft approximativer Algorithmen?

- ☐ Sie liefern immer eine optimale Lösung.
- ☐ Sie können eine Gütegarantie wie einen Faktor ρ besitzen.
- ☐ Sie dürfen keine Heuristik verwenden.
- ☐ Sie müssen immer randomisiert sein.

Was passiert bei A*, wenn für alle Knoten $h(n) = 0$ gilt?

- ☐ A* findet dann keine kürzesten Wege mehr.
- ☐ A* betrachtet nur den direkten Nachbarn des Startknotens.
- ☐ A* bleibt unverändert.
- ☐ A* verhält sich im Wesentlichen wie Dijkstra.

Welche Aussage über eine zulässige Heuristik bei A* ist richtig?

- ☐ Sie überschätzt die echten Restkosten zum Ziel nicht.
- ☐ Sie muss für jeden Knoten den echten Abstand zum Ziel angeben.
- ☐ Sie erlaubt negative Kantengewichte.
- ☐ Sie sortiert die Priority-Queue nur nach $h(n)$.

Welche Aussage unterscheidet Las-Vegas- und Monte-Carlo-Algorithmen korrekt?

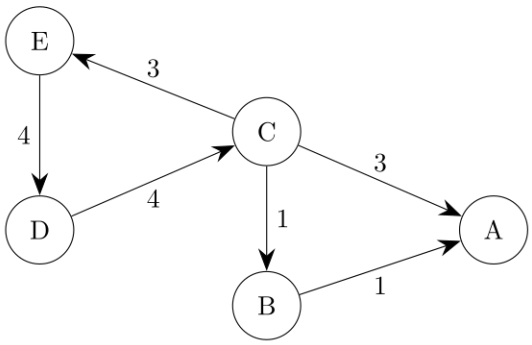
- ☐ Las-Vegas-Algorithmen liefern stets ein korrektes Ergebnis, ihre Laufzeit kann jedoch variieren. Monte-Carlo-Algorithmen haben eine beschränkte Laufzeit, können aber mit einer gewissen Wahrscheinlichkeit ein falsches Ergebnis liefern.
- ☐ Las-Vegas-Algorithmen haben stets eine beschränkte Laufzeit, können aber falsche Ergebnisse liefern. Monte-Carlo-Algorithmen liefern stets korrekte Ergebnisse.
- ☐ Beide Algorithmustypen liefern stets korrekte Ergebnisse; sie unterscheiden sich nur in ihrer Laufzeit.
- ☐ Beide Algorithmustypen können falsche Ergebnisse liefern; nur ihre Fehlerwahrscheinlichkeiten unterscheiden sich.

Aufgabe 7: Bellman-Ford-Algorithmus (6 Punkte)

(a) (4 Punkte) Gegeben ist der folgende gewichtete Digraph. Führe den Bellman-Ford-Algorithmus **ohne** Warteschlange aus. Der Startknoten ist E.

Betrachte in jeder Iteration die Knoten in der Reihenfolge A, B, C, D, E. Betrachte die Nachbarn eines Knotens in lexikographischer Reihenfolge.

Übernimm jede Distanzverbesserung sofort. Sie darf bei später betrachteten Knoten derselben Iteration verwendet werden. Trage nach der Initialisierung und nach jeder vollständigen Iteration die aktuellen Distanzen in die Tabelle ein.



Iteration	A	B	C	D	E
Initial	∞	∞	∞	∞	0
1					
2					
3					
4					

(b) (2 Punkte) Wir fügen dem gegebenen Graphen eine gerichtete Kante von B nach D mit ganzzahligem Gewicht w hinzu. Dadurch entsteht der Zyklus $D \to C \to B \to D$.

Wir betrachten Kantenfolgen, in denen Knoten und Kanten mehrfach vorkommen dürfen.

Hinweis: Als kürzeste Kantenfolge bezeichnen wir im Folgenden eine Kantenfolge mit minimalem Gesamtgewicht.

1. Bestimme das Gewicht des Zyklus in Abhängigkeit von w.

2. Begründe kurz, warum es für $w = -5$ von E aus **keine** eindeutige kürzeste Kantenfolge gibt.

3. Bestimme die kleinste ganze Zahl w, für die von E aus zu allen Knoten eine eindeutige kürzeste Kantenfolge existiert.

Aufgabe 8: Hashing (6 Punkte)

Gegeben sind zwei Hashtabellen. Beide Tabellen enthalten bereits die Schlüssel $3, 11, 19, 6$. Es wird die folgende Hashfunktion verwendet:

$$h(k) = k \bmod 8.$$

In der Tabelle **Verkettung** werden Kollisionen durch Verkettung aufgelöst. Die Schlüssel einer Kette stehen in Einfügereihenfolge von links nach rechts. In der Tabelle **Lineares Sondieren** wird bei einer Kollision zyklisch das jeweils nächste Tabellenfeld untersucht.

Index	0	1	2	3	4	5	6	7
Verkettung				3, 11, 19			6	

Index	0	1	2	3	4	5	6	7
Lineares Sondieren				3	11	19	6	

(a) (2 Punkte) Berechne $h(27)$ und füge den Schlüssel 27 in beide Tabellen ein. Trage die neuen Inhalte direkt in die Tabellen oben ein.

$h(27) =$ _____

(b) (2 Punkte) Ausgehend von den oben gezeigten Ausgangstabellen wird der Schlüssel 19 gesucht. Gib für beide Tabellen die Schlüssel an, die dabei der Reihe nach verglichen werden.

Verkettung:

Lineares Sondieren:

(c) (2 Punkte) Beantworte jede der folgenden Fragen, indem du genau eine passende Antwort ankreuzt.

Wann ist eine Hashtabelle besonders gut geeignet?

- ☐ Wenn aus einem großen Schlüsseluniversum nur wenige Schlüssel gespeichert und häufig exakt gesucht werden.
- ☐ Wenn häufig Schlüssel aus einem Wertebereich gesucht werden.
- ☐ Wenn die Schlüssel stets sortiert ausgegeben werden müssen.
- ☐ Wenn jede Suche garantiert $\mathcal{O}(1)$ dauern muss.

Welche Laufzeit kann die Suche in einer Hashtabelle mit n gespeicherten Schlüsseln im schlechtesten Fall haben?

- ☐ $\mathcal{O}(1)$
- ☐ $\mathcal{O}(\log n)$
- ☐ $\mathcal{O}(n)$
- ☐ $\mathcal{O}(n \log n)$

Unter welcher Bedingung ist die erwartete Suchzeit in einer Hashtabelle $\mathcal{O}(1)$?

- ☐ Die Schlüssel werden gleichmäßig verteilt, der Füllgrad darf aber wachsen.
- ☐ Der Füllgrad bleibt konstant; alle Schlüssel liegen im selben Feld.
- ☐ Die Hashfunktion wird für jeden Schlüssel nur einmal ausgewertet.
- ☐ Die Schlüssel werden gleichmäßig verteilt und der Füllgrad bleibt konstant.

Welche Art von Hashfunktion wird für eine im Voraus bekannte statische Schlüsselmenge so konstruiert, dass keine Kollisionen auftreten?

- ☐ universelle Hashfunktion
- ☐ perfekte Hashfunktion
- ☐ kryptographische Hashfunktion
- ☐ probabilistische Hashfunktion

Aufgabe 9: Dynamische Programmierung (10 Punkte)

Ein Würfel mit 6 Seiten und den Augenzahlen $1, 2, 3, 4, 5$ und 6 wird mehrfach geworfen. Die Augenzahlen werden aufsummiert. Mithilfe dynamischer Programmierung soll für eine gegebene Summe N ermittelt werden, auf wie viele Arten diese Summe geworfen werden kann.

Beispiel: Für $N = 3$ gibt es 4 mögliche Wurffolgen: (3), $(1, 2)$, $(2, 1)$ und $(1, 1, 1)$. In folgender Grafik siehst du die Gruppierung der Wurffolgen nach dem ersten Wurf.

Erster Wurf		**Restsumme**	**Wurffolge inkl. erstem Wurf**
$\boxed{\mathbf{1}}$	$\Rightarrow$	$\boxed{N = 2}$	$(\mathbf{1}, 2)$, $(\mathbf{1}, 1, 1)$
$\boxed{\mathbf{2}}$	$\Rightarrow$	$\boxed{N = 1}$	$(\mathbf{2}, 1)$
$\boxed{\mathbf{3}}$	$\Rightarrow$	$\boxed{N = 0}$	$(\mathbf{3})$

Ziel dieser Aufgabe ist es, eine rekursive Funktion Anz zu bestimmen, mit der das Problem mithilfe dynamischer Programmierung gelöst werden kann.

(a) (1.5 Punkte) Gib alle möglichen Wurffolgen für $N = 4$ an.

(b) (4 Punkte) Definiere eine rekursive Funktion Anz(n), die die Anzahl möglicher Wurffolgen mit Summe n berechnet. Verwende eine mathematische Schreibweise und keinen Java- oder Pseudocode.

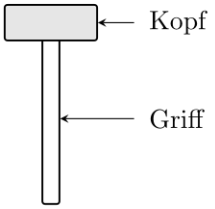
Hinweis: Gruppiere die Wurffolgen wie in der Abbildung für $N = 3$ danach, welche Augenzahl zuerst geworfen wird. Überlege, welche Restsumme danach noch geworfen werden muss.

(c) (1.5 Punkte) Gib die Größenordnung der Laufzeit in Abhängigkeit von N an, wenn die ANZ-Werte mithilfe einer dynamischen Programmierungstabelle (DP-Tabelle) berechnet werden. Begründe deine Antwort kurz in 1–2 Sätzen.

(d) (1.5 Punkte) Gib eine sinnvolle obere Schranke in Abhängigkeit von N für die Laufzeit einer rekursiven Lösung ohne Memoisierung an. Begründe deine Antwort kurz in 1 bis 2 Sätzen.

(e) (1.5 Punkte) Wie verändert sich die Laufzeit mit DP-Tabelle, wenn nun ein M-seitiger Würfel verwendet wird? Dabei ist M ein neuer Eingabeparameter des Problems und es gilt stets $M \leq N$. Begründe deine Antwort kurz in 1–2 Sätzen.

Aufgabe 10: Greedy Algorithmen (11 Punkte)



Für einen Hammer brauchst du genau einen Griff und genau einen Kopf. Es gibt Maschinen, die Griffe produzieren, und Maschinen, die Köpfe produzieren. Jede Maschine M wird durch ein Tupel aus zwei Werten beschrieben:

$$M = (t_{\text{start}}, d)$$

- t_{start}: der Zeitpunkt, ab dem die Maschine verfügbar ist,
- d: die Dauer deines Auftrags auf dieser Maschine.

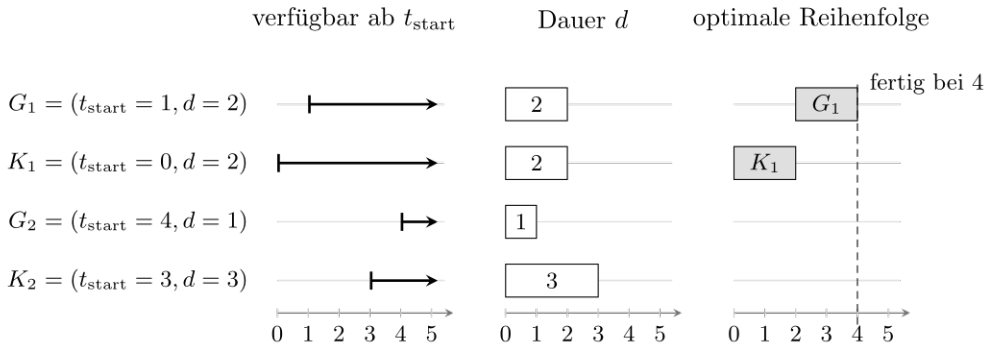
Wenn du eine Maschine zum Zeitpunkt $t \geq t_{\text{start}}$ startest, ist dein Auftrag zum Zeitpunkt $t + d$ fertig. Du kannst immer nur eine Maschine gleichzeitig benutzen. Danach kannst du den nächsten Auftrag starten.

Seien

$G_1, \ldots, G_n$ die Maschinen für Griffe,
$K_1, \ldots, K_m$ die Maschinen für Köpfe.

Gesucht ist der frühestmögliche Zeitpunkt, zu dem der Hammer fertig ist.

Beispiel:



Hier ist es optimal, zuerst K_1 und danach G_1 zu benutzen. Die Lösung ist also 4.

Alle Teilaufgaben sind separat voneinander lösbar.

(a) (4 Punkte) Jemand schlägt folgenden Algorithmus vor:

> Wähle zuerst die Maschine mit der frühesten Endzeit $t_{\text{start}} + d$. Wähle danach für das andere Teil die Maschine, die nach diesem Zeitpunkt am frühesten fertig wird.

Gib ein Gegenbeispiel an, das zeigt, dass dieser Algorithmus nicht immer optimal ist.

Hinweis: Es gibt ein Gegenbeispiel mit je einer Maschine für Griffe und Köpfe.

(b) (4 Punkte) Du bist nun auf der Suche nach dem optimalen Algorithmus für dieses Problem. Betrachte zunächst den Spezialfall, dass immer zuerst ein Kopf und danach ein Griff produziert werden muss. Beschreibe einen optimalen Greedy-Algorithmus für diesen Spezialfall und gib seine Laufzeit in Abhängigkeit von n und m an. Begründe kurz die Laufzeit.

(c) (3 Punkte) Entwirf nun einen optimalen Algorithmus mit Laufzeit $O(n + m)$, der das ursprüngliche Problem löst, bei dem die Maschinenreihenfolge frei gewählt werden darf. Begründe, warum dein Algorithmus die Laufzeit $O(n + m)$ hat.

Hinweis: Du kannst den Algorithmus aus Teilaufgabe b) als gegeben voraussetzen und hier wiederverwenden.

Diese Seite kannst Du für Notizen verwenden. Bitte nur im Ausnahmefall für Lösungen verwenden!

Diese Seite kannst Du für Notizen verwenden. Bitte nur im Ausnahmefall für Lösungen verwenden!