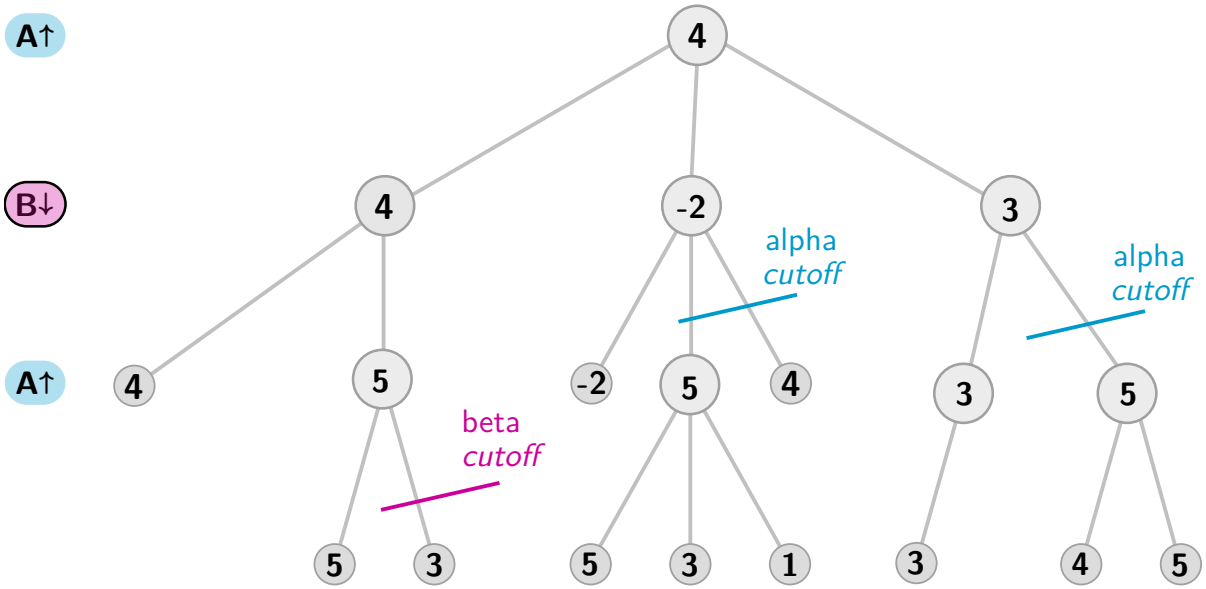
Klausurvorbereitende Aufgaben

Dies ist eine Sammlung von Aufgaben, die Sie zur Klausurvorbereitung zusätzlich zu den Aufgaben auf den Übungsblättern nutzen können. Uu diesen Aufgaben wird es Lösungen geben, die am Ende des Semesters für Sie ins git geladen werden. Sie sollten alle Aufgaben bearbeiten, ohne sich die Lösungen anzuschauen und Ihre Lösungen dann damit vergleichen. Es besteht kein Anspruch auf Vollständigkeit. Schauen Sie sich zur weiteren Ergänzung auch die Altklausuren an.
Denken Sie daran, dass es dieser Jahr erlaubt sein wird ein von Hand beidseitig beschriebenes DIN A4 Blatt (`Spickzettel`) zuverwenden. Es wurden einige Themen als Klausurthemen ausgeschlossen als Erleichterung für Sie aufgrund des Covid-Semesters: Greedy-Algorithmen, Branch-and-Bound, Flussgraphen (Woche 10), Heuristische Algorithmen und Approximative Algorithmen (Woche 11). Wir empfehlen, dass Sie sich gut überlegen, welche Inhalte Sie auswählen und wie Sie diese auf dem Zettel kurzgefasst darstellen. Die sorgfältige Erstellung ist nach unserer Erfahrung der wichtigste Aspekt eines Spickzettels.

Aufgabe 1: Minimax und Alpha-Beta

1.1 Vervollständigen Sie den obigen Minimax Suchbaum.

1.2 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. Kennzeichnen Sie, welcher Cut ein α oder ein β-Cutoff ist.



Aufgabe 2: Dynamische Programmierung

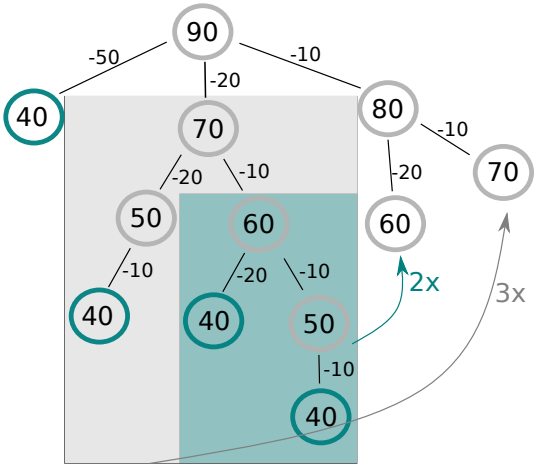
Nehmen Sie an, es ist ein Array an Münzwerten `coins` gegeben, z. B. `[10, 20, 50]`, sowie ein Gesamtwert, z. B. 90 Cent. Nun sollen mithilfe von dynamischer Programmierung die Frage beantwortet werden, welches die minimale Anzahl von Münzen ist, mit der der Gesamtwert zusammengestellt werden kann. Dabei dürfen nur Münzen mit den angegebenen Münzwerten verwendet werden. Gehen Sie davon aus, dass der Betrag immer passend mit dem Münzen darstellbar ist.

2.1 **Vorübung** Welche beiden Voraussetzungen muss ein Problem erfüllen, damit dynamische Programmierung erfolgreich angewendet werden kann? Welche beiden Varianten werden zur Speicherung der Lösungswert verwendet?

- optimale Substruktur, d.h. die optimale Lösung des Problems lässt sich auf die optimale Lösung von Teilproblemen zurückführen
- überlappende Teilprobleme, d.h. Teilprobleme tauchen an vielen Stellen im Lösungsbaum auf
- Speicherung der Teillösungen:
  1. top-down Ansatz: Teillösungen werden nach Bedarf berechnet und z.B. in einem Feld gespeichert (memoization)
  2. bottom-up Ansatz: die Teillösungen werden von Grund auf berechnet, je nach Problem müssen dabei nicht alle Werte gespeichert werden (tabulation)

2.2 Stellen Sie sich vor, Sie sollen die Lösung des Münzproblems für einen Betrag v angeben. Für welche Subprobleme (also Probleme mit kleineren Beträgen als v) benötigen Sie die Lösung, um daraus die Lösung für v einfach bestimmen zu können? Welche Randfälle gibt es zu beachten? Für die Lösung zum Wert v benötigt man die Lösungen für $v - coins[i]$, $i = 1, \ldots$. Also die Lösungen für die Teilprobleme, bei denen von v der Münzenwert abgezogen wurde. Ein Randfall, den man betrachen sollte, ist $v = 0$ und für die OPT-Funktion ist es pranktisch den Fall $v < 0$ zu betrachten-

2.3 Wie oft wird die Teillösung für den Betrag 40 Cent benötigt, um die Aufgabe für den Betrag 90 Cent zu lösen? – 9 Mal:



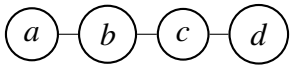
2.4 Stellen Sie eine Opt-Funktion für das Problem auf, die die minimale Anzahl der benötigten Münzen angibt. Die Funktion sollte für ein beliebiges Array `coins` definiert werden.
Tipp: Bei der Fallunterscheidung ist es sinnvoll, für den Fall, dass der Geldwert < 0 ist, die Anzahl als *unendlich* zu definieren. Dies bedeutet, dass der Geldwert mit den vorhandenen Münzwerten nicht darstellbar ist.

$$\textsc{Opt}(v) = \begin{cases} 0 & \text{falls } v = 0 \\ \infty & \text{falls } v < 0 \\ \min\{1 + \textsc{Opt}(v - c) \mid c \in \mathit{coins}\} & \text{sonst} \end{cases}$$

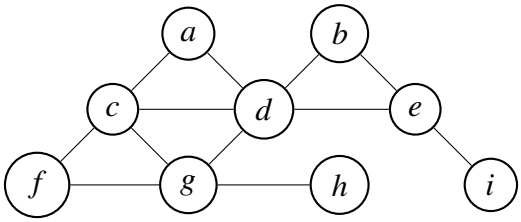
Aufgabe 3: Breitensuche, Tiefensuche

3.1

1. **Vorübung:** Ausgehend vom Startknoten wird eine kreuzende Kante bezüglich des markierten Bereiches ausgewählt, dabei sucht die Breitensuche vom Knoten, dessen Entdeckung em längsten her ist und die Tiefensuche vom Knoten, der als letztes entdeckt wurde. Dementsprechend geht die Breitensuche zuerst in die Breite des Graphen und besucht zunächst alle Knoten mit Kantenabstand 1 vom Startknoten, dann die mit Kantenabstand 2 etc. Die Tiefensuche geht zuerst in die Tiefe, rekursiv oder über einen Stack implementiert.
2. Entscheiden Sie für folgende Applikationen, ob DFS oder BFS oder beide besser geeignet sind.
   - topologische Sortierung – DFS
   - Bestimmung von Zusammenhangskomponenten in einem ungerichteten Graphen – beide
   - Entdeckung eines Zyklus in einem Graphen – beide (eher DFS)
   - Suche von Wegen, die möglichst wenige Kanten benutzen – BFS
3. Können BFS und DFS in einem Graphen gleich sein? Ja, ein mögliches Beispiel:



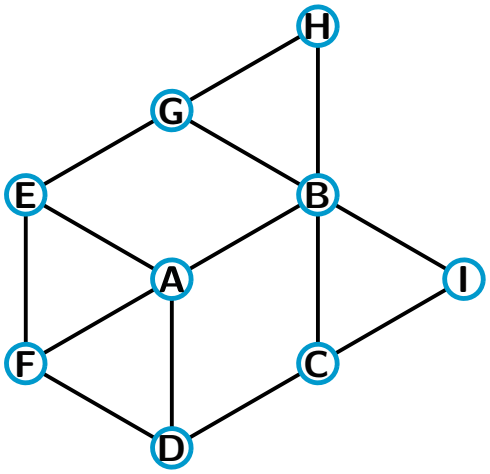
4. Gegeben ist der folgende Graph:



Wenn BFS und DFS auf den Graphen angewendet werden, hängt die Folge, in der die Knoten besucht werden davon ab, in welcher Reihenfolge die adjazenten Knoten durchlaufen werden. Welche der Folgen 1-6 können tatsächlich bei geeigneter Reihenfolge der Adjanzenlisten in der Durchsuchung mit DFS bzw. BFS vorkommen?

1. BFS: a, c, d, g, f, b, e, h, i - Korrekt
2. BFS: a, d, c, f, g, e, b, h, i - Falsch, der Knoten f muss nach den Knoten g, e und b entdeckt werden
3. BFS: a, d, c, e, b, g, f, i, h - Korrekt
4. DFS: a, c, d, g, f, h, b, e, i - Korrekt
5. DFS: a, c, d, b, e, i, f, g, h - Falsch, der Knoten g muss vor f entdeckt werden.
6. DFS: a, c, d, g, h, f, b, e, i - Richtig.

3.2 Führen Sie BFS und DFS auf dem unten gegebenen Graph mit Startknoten A aus und notieren Sie dabei die Reihenfolge, in der die Knoten besucht werden. Gehen Sie dabei davon aus, dass die benachbarten Knoten in alphabetischer Reihenfolge durchlaufen werden.



	0	1	2	3	4	5	6	7	8
BFS	A	B	D	E	F	C	G	H	I
DFS	A	B	C	D	F	E	G	H	I

3.3 Geben Sie die worst-case Laufzeiten von BFS und DFS auf einem Graphen $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ als Wachstumordnungen an.

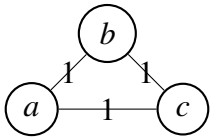
BFS	DFS

BFS	DFS
$O(V+E)$	$O(V+E)$

Aufgabe 4: MST

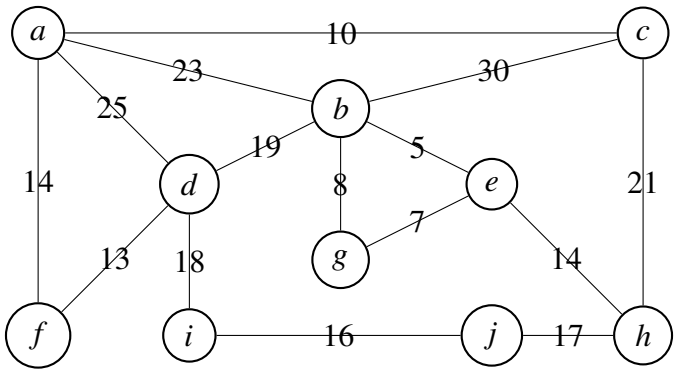
4.1 Grundlagen (Vorübung)

1. Was ist ein Spannbaum? - Der **Spannbaum** eines Graphen G ist ein Teilgraph von G, der alle Knoten von G enthält und ein Baum ist.
2. Was ist ein minimaler Spannbaum (MST)? - Ein **Minimaler Spannbaum** eines gewichteten Graphen ist ein Spannbaum des Graphen, der minimales Gewicht hat.
3. Ist der MST immer eindeutig? - Nein, der folgende Graph hat keinen eindeutigen minimalen Spannbaum:



4.2 Schnitteigenschaft

1. **Vorübung:** Wie ist ein Schnitt durch einen Graphen definiert? Was sind kreuzende Kanten? Was ist die Schnitteigenschaft?
2. Geben Sie für den folgenden Graphen vier verschiedene Schnitte an: Einen mit 2, einen mit 3, einen mit 5 und einen mit 7 kreuzenden Kanten.



3. Wie lautet der allgemeine Ansatz um einen minimalen Spannbaum zu finden?

Lösung:

1. Ein **Schnitt** durch einen Graphen $G = (V, E)$ teilt seine Knoten in zwei nicht-leere Teilgraphen. Konkret beschreiben wir einen Schnitt durch eine Teilmenge SV mit der Eigenschaft, dass (S, E_S) und $(V - S, E_{V-S})$ nicht-leere Graphen sind. Dabei bezeichnet E_S die Menge aller derjenigen Kanten von G, die in S liegen: $E_S := \{(v, w) \in E | v \in S \text{ and } w \in S\}$.
   Kanten mit einem Knoten innerhalb und einem Knoten außerhalb von S heißen **kreuzende Kanten**.
   Schnitteigenschaft: Sei ein beliebiger Schnitt durch einen Graphen gegeben. Jeder minimale Spannbaum des Graphen muss eine der minimal kreuzenden Kanten enthalten.
2. Schnitt: $\{f\} \in S$, $\{a, b, c, d, e, g, h, i, j\} \in V - S$. Kreuzende Kanten: $\{(a, f), (d, f)\}$.
   Schnitt: $\{c\} \in S$, $\{a, b, d, e, f, g, h, i, j\} \in V-S$. Kreuzende Kanten: $\{(a, c), (b, c), (c, h)\}$.
   Schnitt: $\{a, b, d, f, g, i\} \in S$, $\{c, e, h, j\} \in V-S$. Kreuzende Kanten: $\{(a, c), (b, c), (b, e), (g, e), (i, j)\}$.

3.

```
ME := {}
while ME ist kein Spannbaum
  wähle einen Schnitt S, der keine kreuzende Kante in ME hat
  füge eine minimal kreuzende Kante (v,w) zu ME hinzu
end
// ME ist ein minimaler Spannbaum
```

4.3 Algorithmen von Prim und Kruskal

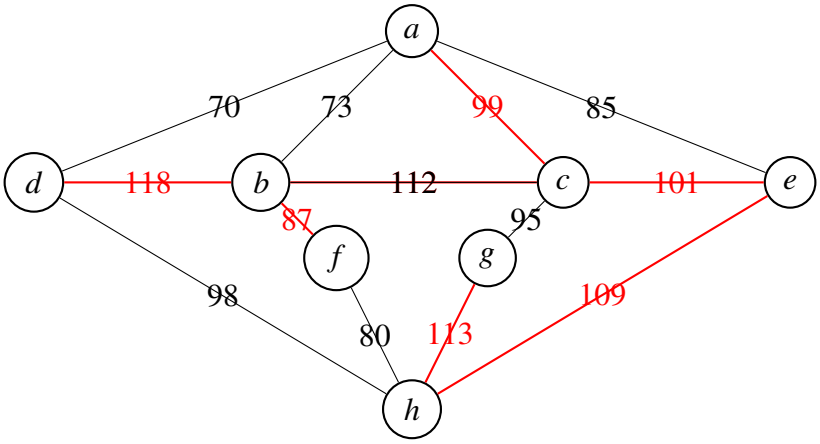
1. Was ist der Hauptunterschied in der Kantenauswahl zwischen den Algorithmen von Prim und Kruskal? - Prim's Algorithmus und Kruskal's Algorithmus gehen in der Reihenfolge der Kantenwahl sehr unterschiedlich vor.
   Prim's Algorithmus: Bilde einen Baum ausgehend von einem Startknoten s. Wähle unter den kreuzenden Kanten eine mit geringstem Gewicht.
   Kruskal's Algorithmus: Durchlaufe die Kanten nach aufsteigendem Gewicht. Füge eine Kante hinzu, wenn sie keinen Zyklus mit den bisher gewählten Kanten bildet. Die Zwischenergebnisse bei Kruskal sind ggf. unzusammenhängend.
2. Führen Sie die beide Algorithmen auf dem obigen Graphen aus. Schreiben Sie die Kanten in der Reihenfolge notieren, in der sie der Algorithmus auswählt.

   Prim: Der Startknoten ist a. Kanten, die MST von dem Graph enthält in der Reihenfolge, in der sie von Prim's hinzugefügt wurden:
   $\{(a,c),(a,f),(d,f),(d,i),(i,j),(h,j),(e,h),(b,e),(e,g)\}$. Gewicht von MST ist 114.
   Kruskal: Kruskal's Algorithmus: Kanten, die MST von dem Graph enthält in der Reihenfolge, in der sie von Kruskal's hinzugefügt wurden:
   $\{(b,e),(e,g),(a,c),(d,f),(a,f),(e,h),(i,j),(h,j)\}$. Je nach Implementierung kann (e,h) auch vor (a,f) hinzugefügt werden. Gewicht von MST ist auch 114.
3. Das Gewicht von jeder Kante wird mit sich selbst multipliziert. Ändert sich der minimale Spannbaum? - Nein, der MST ändert sich nicht, da die Reihenfolge von Gewichten der Kanten sich nicht ändert ($f(x) = x^2$ ist monoton).
4. Welche Datenstruktur sollte benutzt werden, um eine effiziente Implementation des Algorithmus von Prim zu erreichen? - IndexPQ

4.4 Maximaler Spannbaum

Ein maximaler Spannbaum ist ein Spannbaum mit maximalem Gewicht.

1. Wie kann der Kruskal Algorithmus geändert werden, um den maximalen Spannbaum zu finden? - Man Kann die Reihenfolge, in der die Kanten im Algorithmus ausgewählt werden umdrehen; oder alle Gewichte negieren (d.h. aus 2 -2 machen und aus -4 4) und dann den Algorithmus für einen minimalen Spannbaum nutzen.
2. Was ist das Gewicht des maximalen Spannbaums des folgenden Graphens?

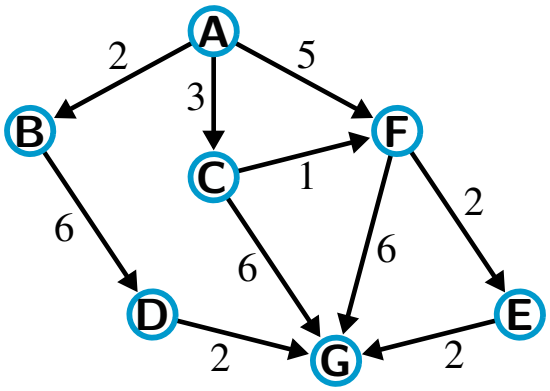


739

4.5 Geben Sie die worst-case Laufzeiten der MST Algorithmen auf einem Graphen $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ als Wachstumordnungen an.

Prim	Kruskal
$O(E \log(V))$ mit IndexPQ	$O(E \log(V))$ mit UnionFind mit Pfadkompression

Aufgabe 5: SSSP



5.1 Tragen Sie in die unten stehenden Tabelle die Distanzen (`dist`) der kürzesten Wegen von Knoten **A** zu allen anderen Knoten mit Hilfe des Dijkstra Algorithmus ein. Notieren Sie auch zu jedem Knoten den Vorgänger Knoten (`parent`) auf einem kürzesten Weg.

	A	**B**	**C**	**D**	**E**	**F**	**G**
`dist`	0	2	3	8	6	4	8
`parent`	-	A	C	B	F	C	E

5.2 Entfernen Sie eine Kante, damit die kürzeste Distanz von **A** nach **G** 9 beträgt.

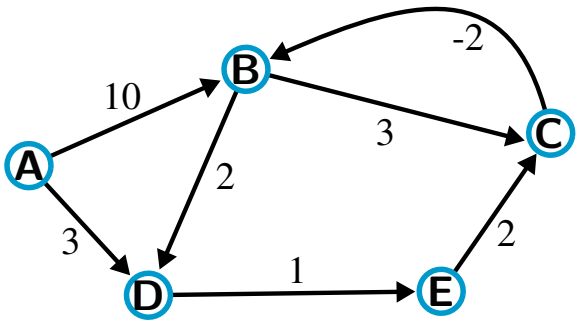
Eine der Kanten (**C**,**F**), (**F**,**E**), (**E**,**G**).

5.3 Beschreiben Sie, für welche Graphen der Dijkstra-Algorithmus nicht so gut geeignet ist und erläutern Sie den Grund.

Für Graphen, die (auch) negative Kantengewichte haben; Kanten werden u. U. mehrfach relaxiert, wodurch die Effizienz leidet. Im *worst-case* ist die Laufzeit exponentiell und damit deutlich schlechter, als die des Bellman-Ford Algorithmus.

(Graphen mit negativen Zyklen haben wir generell von der Frage nach kürzesten Wegen ausgeschlossen.)

5.4 Führen Sie auf folgendem Graphen den Bellman-Ford Algorithmus durch und notieren Sie den Ablauf (vergleiche Screencast Video).



#	Knoten	dist	parent
	A	0	–
1	B	10	A
1	D	3	A
2	C	13	B
2	E	4	D
3	B	11	C
3	C	6	E
4	B	4	C

5.5 Was muss für den Graph gelten, damit man topologische Sortiertung anwenden kann? Ist die topologische Sortierung (wenn sie existiert) eindeutig?

Der Graph muss gerichtet sein und darf keinen Zyklus enthalten (DAG = *directed acyclic graph*).
Die Sortierung ist nicht eindeutig. Bei dem Graphen mit den Kanten **A**→**B** und **A**→**C** sind **A B C** und **A C B** beides korrekte topologische Sortierungen.