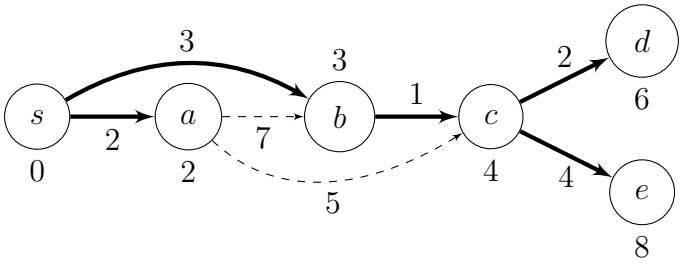
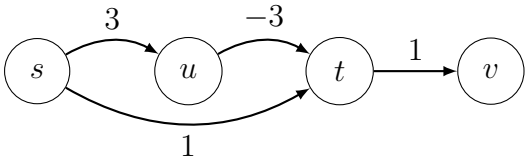
Aufgabe 1



Aufgabe 2

Wir verwenden folgendes Beispiel:



Dijkstras Algorithmus würde die Kante $[s, t\rangle$ mit Gewicht 1 und $[s, u\rangle$ mit Gewicht 3 relaxieren. Somit wird der Knoten t als nächstes aus der Prioritätswarteschlange entnommen und die Kante $[t, v\rangle$ relaxieren. Da t aus der Warteschlange entfernt wurde und auch keine weitere Kante mehr auf v zeigt, wird die Entfernung von 2 für v nicht mehr geändert. Allerdings ist die minimale Distanz von s nach v 1, da der Pfad von s nach t über u Gewicht 0 hat.

Für Graphen mit negativen Zyklen ist es das Versagen direkt offensichtlich, da Dijkstras Algorithmus niemals den Wert $-\infty$ vergibt oder solche Knoten anderweitig markiert.

Aufgabe 3

Wenn wir das Gewicht jeder Kante um 2 erhöhen, verändern sich die minimalen Spannbäume nicht. Da die Anzahl der Kanten aller Spannbäume identisch ist (die Anzahl der Knoten minus 1), gilt für jedes Paar von Spannbäumen T_A, T_B mit

$$\sum_{e \in T_A} w(e) \ \circ \ \sum_{e \in T_B} w(e), \circ \in \{<, =, >\}$$

auch

$$\sum_{e \in T_A} w'(e) = \sum_{e \in T_A} w(e) + 2(n-1) \ \circ \ \sum_{e \in T_B} w(e) + 2(n-1) = \sum_{e \in T_B} w'(e)$$

Anders betrachtet: Algorithmen wie Prim oder Kruskal verhalten sich in ihrer Ausführung nicht anders. Da alle Gewichte um 2 erhöht werden, verändern sich die minimalen Kanten (aus der Menge der betrachteten Kanten) nicht.

Aufgabe 4

Die kürzesten-Wege-Bäume müssen nicht übereinstimmen. Betrachten Sie dazu beispielhaft die kürzesten-Wege-Bäume zu dem folgenden gerichteten, gewichteten Graphen $G = (V, E, w)$ (links) im Vergleich zu $G' = (V, E, w')$ (rechts).

Der kürzeste Weg von s zu v in G führt offensichtlich über u, während in G' der direkte Weg der kürzeste ist.

Aufgabe 5

Zur Lösung des Problems verwenden wir Prim's Algorithmus mit einer Implementierung der Prioritätswarteschlange, welche die geringe Anzahl an möglichen Werten ausnutzt. Dazu nehmen wir zunächst an, dass die Kantengewichte die Werte 1 bis k annehmen. Sonst bestimmen wir zuerst die k verschiedenen Gewichte und sortieren diese in Zeit $\mathcal{O}(k \log k)$.

Anstatt einer „komplexen" Prioritätswarteschlange verwenden wir für jedes mögliche Kantengewicht je eine „normale" Warteschlange in Form einer Liste. Statt in die Prioritätswarteschlange einzufügen, fügen wir in $\mathcal{O}(1)$ in die dem Kantengewicht entsprechende Warteschlange ein. Für ein `deleteMin` entnehmen wir einen Knoten aus der ersten nichtleeren Warteschlange mit geringstem entsprechendem Kantengewicht. Dies ist in jedem Fall in $\mathcal{O}(k)$ möglich. Um die `decreaseKey` Operation auszuführen, speichern wir für jeden Knoten den aktuellen Eintrag in der entsprechenden Warteschlange. Wenn wir doppelt verkettete Listen als Warteschlangen verwenden, können wir den Eintrag in $\mathcal{O}(1)$ löschen und in eine andere Warteschlange einführen. Analog zur Vorlesung ergibt sich somit eine Gesamtlaufzeit von $\mathcal{O}(nk + m)$.

Wir erhalten also eine Laufzeitverbesserung, wenn nur wenige verschiedene Kantengewichte vorkommen, nämlich genau dann wenn $k \in \mathcal{O}(\log n)$.

Für den Fall $k = 2$ reicht es aus eine einzelne Liste zu verwenden bei der die Elemente mit Priorität 1 oben und jene mit Priorität 2 unten angehängt werden. Die Gesamtlaufzeit ist linear, $\mathcal{O}(n + m)$.

Aufgabe 6

Es gilt: Für einen Graphen mit Kantengewichten aus der Menge $\{1, 2\}$ befinden sich in der Prioritätswarteschlange des Dijkstra Algorithmus zu jedem Zeitpunkt höchstens 3 verschiedene Prioritäten. Besaß der letzte entnommene Knoten u die Priorität und Distanz $d[u]$, so gilt: Da $d[u]$ minimal war, existierten keine Elemente mit niedrigerer Priorität. Alle neu entdeckten Pfade die dem Pfad zu u mit Gewicht $d[u]$ eine Kante hinzufügen haben Gewicht $d[u] + 1$ oder $d[u] + 2$. Es existieren also auch nach dem verarbeiten von u keine niedrigeren Prioritäten. Gleichzeitig hatte kein vor u entnommener Knoten Priorität und Distanz größer als $d[u]$, sodass kein Knoten eine Priorität größer $d[u] + 2$ besitzen kann. Alle zu diesem Zeitpunkt in der Prioriätswarteschlange enthaltenen Knoten besitzen also Priorität $d[u], d[u] + 1$ oder $d[u] + 2$.

Dementsprechend ist es möglich ähnlich zu Aufgabe (5) die Prioritätswarteschlange durch Listen und einen Zeiger für jeden Knoten zu ersetzen, sodass `deleteMin` und `decreaseKey` nur konstante Zeit benötigen. Es genügt drei Listen L_0, L_1, L_2 zu verwalten, die jeweils Knoten mit Priorität $d[u], d[u] + 1$ und $d[u] + 2$ enthalten, wobei $d[u]$ die Distanz des zuletzt entnommenen Knoten u ist. Sobald L_0, die Liste mit geringster Priorität, leer ist ersetzt man L_0 durch L_1, L_1 durch L_2 und L_2 durch die leere Liste L_0. Für die `decreaseKey`-Operation speichern wir erneut den aktuellen Eintrag jedes Knotens um diesen in konstanter Zeit entfernen zu können.

Die n `deleteMin`-Operationen haben also eine Laufzeit in $\mathcal{O}(n)$. Die höchstens m vielen `decreaseKey`-Operationen haben eine Laufzeit in $\mathcal{O}(m)$. Es ergibt sich eine Gesamtlaufzeit von $\mathcal{O}(n + m)$.

Aufgabe 7

Hier wird der Algorithmus von Kruskal angewandt, d.h. in jedem Schritt wird eine minimale Kante ausgewählt die mit den bereits ausgewählten Kanten keinen Kreis bildet.