Datenstrukturen und Algorithmen: Blatt 13

Bernhard Dietrich (6256800)
Lars Fernhomberg (6256030)
Sebastian Kniesburges (6257120)
Marcus Köthenbürger (6258550)

Übungsgruppe 11
Mittwoch, 11:00-13:00 Uhr in D1.303
Matthias Ernst

Aufgabe	1	2	3	4	Σ	Korrektor
Punkte						
von	6	6	6	4	22	

Aufgabe 1

Zeigen Sie, dass der gegebene Algorithmus DELETE-MST für ungerichtete, zusammenhängende, gewichtete Graphen (G, w) einen minimalen Spannbaum berechnet.

Die Kanten sind durch Zeile 1 in absteigender Gewichtung gegeben, so dass in der for-Schleife verschiedene Kanten, die zu einem Punkt führen, in der Reihenfolge ihres Gewichtes abgearbeitet werden. Ein minimaler Spannbaum wird nun dadurch erzeugt, dass sämtliche Kanten, die für den Zusammenhang des Graphen irrelevant sind (vgl. Zeile 4) aus der Kantenmenge gelöscht werden (vgl. Zeile 5). Da die zuerst gelöschten Kanten ein höheres (oder gleiches) Gewicht als sämtliche Kanten haben, die in der sortierten Reihenfolge nach Ihnen kommen, werden somit bei allen Knoten zuerst die Kanten mit den höchsten Gewichten entfernt, so dass am Ende nur die Kanten zu diesem Knoten führen, welche für den Spannbaum unbedingt erforderlich sind, gleichzeitig aber minimales Gewicht haben. Da dies für alle Kanten gilt, ergibt sich somit ein minimaler Spannbaum.

Aufgabe 2

Sie haben eine Menge von Münzen mit den Nennwerten $c^0, c^1, ..., c^k$ für $c \in \mathbb{N}$ und ein $k \in \mathbb{N}$. Für jeden der möglichen Nennwerte stehen Ihnen beliebig viele Münzen zur Verfügung. Sie sollen Betrag b mit Ihren Münzen bezahlen, dabei allerdings möglichst wenige Münzen benutzen. Entwerfen Sie einen effizienten gierigen Algorithmus, der dieses Problem löst. Zeigen Sie die Korrektheit Ihres Algorithmus und analysieren Sie seine Laufzeit.

Eingabe: Betrag b, der beglichen werden muss, Basis c der Münzen und der maximale Exponent der größten Münze (also c^k).

```
PAY(b, c, k)
1.    i ← k
2.    while i > -1 and b>0
3.          do a ← round_down(b/c^i)
4.             Gebe a mal die Münze c^i aus
5.             b ← b - a * c^i
6.             i ← i - 1
```

Der Algorithmus ist gierig, da immer die maximale Anzahl an größtmöglichen Münzen ausgegeben wird.

Invariante der while-Schleife: Die maximale Anzahl an Münzen des Typs c^i, die in den Restbetrag b passt und sofern dieser größer als 0 ist, wird ausgegeben, so dass am Ende der komplette Betrag bezahlt ist.
Initalisierung: Vor Schleifenbeginn ist noch kein Geld ausgegeben worden, sodass der offene Restbetrag b beträgt und die erste Münze, die ausgegeben wird, c^k ist.

Erhaltung: Es wird in Zeile 3 ermittelt, wie oft die Münze c^i in den Betrag b passt, um diese Anzahl an Münzen auszugeben (vgl. Zeile 4). Der Betrag wird anschließend um den ausgegebenen Betrag reduziert und eine neue Münze gewählt (Zeile 5-6).

Terminierung: Da jedes Mal die maximal mögliche Anzahl an Münzen des jeweiligen Typs ausgegeben wurden und durch $c^0 = 1$ sämtliche noch offene Restbeträge beglichen werden können, arbeitet der Algorithmus wie spezifiziert.

Zeile 1 hat konstante Laufzeit $O(1)$, die Zeilen 3-6 haben pro Durchlauf der Schleife ebenfalls eine konstante Laufzeit. Die while-Schleife wird insgesamt für k Münzen durchlaufen, so dass diese die Laufzeit $O(k)$ hat. Es ergibt sich somit für den gesamten Algorithmus eine Laufzeit im Worst Case (es soll eine Summe bezahlt werden, bei der jede Münze mindestens einmal ausgegeben werden muss) von $O(k)$.

Aufgabe 3

Gegeben sind n Jobs $j_1, j_2, ..., j_n$, die alle auf einem Prozessor ausgeführt werden sollen. Jeder der Jobs benötigt genau eine Zeiteinheit, um auf dem Prozessor ausgeführt zu werden. Zu jedem Job j_i gibt es noch eine Deadline d_i und eine Belohnung b_i. Ist Job j_i spätestens zur Deadline d_i erledigt, wird die Belohnung b_i ausgezahlt. Wird der Job j_i später als zur Deadline d_i erledigt, so wird keine Belohnung ausgezahlt. Gesucht ist eine Reihenfolge, in der die Jobs abgearbeitet werden, die die gesamte ausbezahlte Belohnung maximiert. Entwerfen Sie einen gierigen Algorithmus, der dieses Problem löst und zeigen Sie seine Korrektheit.

Unser Ansatz sieht vor, dass die gegebenen Jobs zuerst nach der Belohnung in absteigender Reihenfolge sortiert werden. Anschließend werden die Jobs in eine „Zeitleiste“ (implementiert z.B. durch Array) eingeordnet, wobei versucht wird, die Jobs möglichst nahe an ihre Deadline zu setzen. Da jeder Job gemäß Aufgabenstellung genau eine Zeiteinheit benötigt, stellen in einem Array die Indizes der einzelnen Felder die Zeiteinteilungen da. Die Platzierung erfolgt nun so, dass verglichen wird, ob das Feld, welches vor der Deadline liegt (also Deadline-1), leer ist. Ist dies der Fall, wird das Element dort eingeordnet und es wird zum nächsten Job weitergegangen. Ist das entsprechende Feld bereits belegt, so wird die Zeitleiste von der Deadline zum Beginn der Zeitleiste nach einem leeren Platz durchsucht, auf dem der Job einsortiert wird. Findet sich kein freier Platz vor der Deadline, so wird der Job verworfen, da alle Elemente, die sich nun vor der Deadline des Jobs befinden, eine höhere Belohnung einbringen und somit, um den entsprechenden Job fristgerecht zur Deadline beenden zu können, ein bereits einsortierter Job entfernt werden müsste, welcher aber eine höhere Belohnung einbringt, so dass die Belohnung durch Überschreibung eines bestehenden Jobs durch den aktuellen Job nicht maximiert wird.

Hinweis: Die frühste Deadline die in unserem Ansatz realistisch möglich ist, ist eine Deadline nach dem ersten Zeitabschnitt (also bei 2), da ansonsten (bei Deadlone=1) der entsprechende Job bereits während der Einsortierungsphase nicht erfüllbar ist.

Eingabe: Array mit den Jobs $j_0, ..., j_n$

Ausgabe: Array mit der optimierten Ausführungsreihenfolge der Jobs (aka „Zeitleiste“)

```
OptimizeJobs(J)
1.    Jobliste J absteigend nach Belohnung sortieren
2.    for i ← 1 to length(J)
3.          do d ← deadline(J[i])
```

```
4.            for j ← d − 1 downto 1
5.                 do if Z[j] = NIL
6.                         then Z[j] ← J[i]
7.                              exit innere for-Schleife
8.    return Z
```

Der Algorithmus ist gierig, da immer versucht wird, Jobs mit einer möglichst großen Belohnung so einzusortieren, dass die maximale Anzahl an Belohnungen erreicht werden kann.

Invariante der inneren for-Schleife (Zeilen 4-7): Das Element $J[i]$ wird an eine leere Stelle der Zeitleiste zwischen 1 und $d-1$ eingeordnet oder, wenn es keine leere Stelle gibt, verworfen.

Initalisierung: Vor der Initalisierung der Schleife besitzt die Variable j noch keinen Wert, sodass es keine Stelle gibt, an der das Element $J[i]$ eingeordnet werden könnte.

Erhaltung: Sollte die Stelle $Z[j]$ frei sein, so wird das Element $J[i]$ hier eingefügt, ansonsten wird weiter verglichen.

Terminierung: Es wurde entweder eine leere Stelle gefunden und das Element $J[i]$ eingefügt oder es gab keine leere Stelle, so dass das Element verworfen wird. Die Schleife arbeitet somit wie spezifiziert.

Invariante der äußeren for-Schleife (Zeilen 2-7): Die gegebenen Jobs werden effizient, d.h. Belohnungsmaximierend, in die Zeitleiste eingefügt bzw. verworfen, falls die Einfügeoperation die Belohnung nicht maximiert.

Initalisierung: Vor der Initalisierung der Schleife besitzt die Variable i noch keinen Wert, so dass noch kein Element eingefügt wurde.

Erhaltung: Der Job $J[i]$ wird durch die innere Schleife eingefügt, sofern in der Zeitleiste leere Plätze existieren, oder wird verworfen, falls diese nicht existieren. Wenn ein Element verworfen wird, so bedeutet dies, dass es zwischen dem ersten Feld der Zeitleiste und der Deadline kein leeres Feld gibt, so dass, um den Job $J[i]$ einzufügen, ein bereits belegtes Feld überschrieben werden müsste. Da die Jobs allerdings absteigend nach ihrer Belohnung sortiert wurden (vgl. Zeile 1) würde dieser Überschreibungsvorgang bedeuten, dass ein Job mit einer niedrigeren Belohnung, welches aufgrund der Sortierung später eingefügt wird, einen bereits vorhandenen, eingefügten Job mit einer höheren (oder gleichen) Belohnung überschreiben würde, welcher so zeitnah an seiner Deadline wie möglich liegt. Dieses würde keine Verbesserung der maximal zu erreichenden Belohnung bedeuten, so dass der Job verworfen wird, um die maximale Belohnung erhalten zu können.

Terminierung: Da alle Jobs effizient eingefügt (bzw. eben nicht eingefügt) wurden, arbeitet der Algorithmus korrekt.

Aufgabe 4

Für einen ungerichteten Graphen $G=(V,E)$ definieren wir das Paar $A_G=(S_G,I_G)$ folgendermaßen

1. $S_G = E$
2. $T \subseteq E$ liegt in I genau dann, wenn der Graph $G_T=(V,T)$ azyklisch ist.

Zeigen Sie, dass für jeden Graphen G das so definierte Paar $A_G=(S_G,I_G)$ ein Matroid ist.

Nachweis der drei Matroid-Eigenschaften:

1. Endliche Menge
   S endliche Menge gemäß Definition ✓
2. Vererbung
   $B \subseteq S, B \in I$, d.h. azyklisch.
   $A \subseteq B$, d.h. $G_A \subseteq G_B \Rightarrow$ G_A azyklisch und $A \in I$
   $\Rightarrow A \in S$ ✓
3. Austausch
   $A \in I$, $B \in I$, $|A| < |B|$, $B \setminus A$=Kanten in B und nicht in A
   $G_A (V, A)$ azyklisch und $|A| \leq |V| - 1$
   $G_B (V, B)$ azyklisch und $|B| \leq |V| - 1$
   Es folgt somit $|A| \leq |V| - 1$
   Da G_A azyklisch und $\exists v \in V$ ohne Kante (v, u) in G_A mit Kante (v, u) in G_B
   füge (u, v) zu A hinzu.
   $A' = A \cup \{(u, v)\}$ bleibt azyklisch womit folgt $A' \in I$. ✓

Durch den Beweis aller drei Matroid-Eigenschaften folgt, dass A_G ein Matroid ist.