Algorithmendesign
Lösungen zu Übungsblatt 4
WS 13/14

Gruppe 2
Max Bannach

Markus Richter (614027)

Aufgabe 4.1 Editierdistanz

Teilaufgabe 1)

$lev(s,t) = lev(t,s)$ gilt immer, angenommen man will s in t umwandeln und führt im optimalen Fall – also eine Umwandlung mit minimalen Kosten – an einer oder mehreren bestimmten Stellen s_i eine Ersetzung durch t_j aus, so dreht sich das im umgekehrten Fall um: t_j wird durch s_i ersetzt, die Kosten betragen also für das Ersetzen beide Male c_2. Für das Einfügen gilt: Hat man in einer Umwandlungsrichtung eingefügt muss man in der anderen Richtung löschen, d. h. es fallen beide Male kosten von c_1 an. Für das Löschen gilt das Umgekehrte. Daraus folgt also, dass die Kosten, also die Levenshtein-Distanz, für $lev(s,t) = lev(t,s)$.

Teilaufgabe 2)

$lev(\epsilon,\epsilon) = 0$, $lev(\epsilon,t) = |t|$, $lev(s,\epsilon) = |s|$

Teilaufgabe 3)

Es gilt $|s'| + |t'| < |s| + |t|$, daraus folgt, dass entweder $|s'| = |s| \wedge |t'| < |t|$ oder $|s'| < |s| \wedge |t'| = |t|$ möglich ist. Weiterhin definiere ich die Funktion Prefix $pref(x_i)$. Sie liefert das Wort $x_0 ... x_{i-1}$, d. h. $|pref(x)| < |x|$. Es ergeben sich die folgenden Möglichkeiten:

Löschen
$lev(s,t) = lev(pref(s),t) + 1$ Das bedeutet, man addiert zu der Distanz, die sich durch das Löschen ergibt, noch die Kosten für das Löschen selbst.

Einfügen $lev(s,t) = lev(t,pref(t)) + 1$ Es beschreibt also einen Übergang vo

Ersetzen $lev(s,t) = lev(pref(s),pref(t)) + 1$ (aber nur wenn verglichene Zeichen unterschiedlich). Das bedeutet, wenn eine Ersetzung stattfindet ist nur die Distanz beider Prefixe relevant zzgl. der Kosten für eine Ersetzung – sofern die verglichenen Zeichen unterschiedlich sind.

Da man damit alle Möglichkeiten betrachtet hat, da aber die Levenshtein-Distanz die minimalen Kosten zurück gibt, muss aus den ermittelten Werten das Minimum gewählt werden.

Teilaufgabe 4)

$$lev(s,t) = \begin{cases} \max(|s|,|t|) & \text{if } \min(|s|,|t|) = 0, \\ min \begin{cases} lev(prefix(s),t) + 1 \\ lev(s,prefix(t)) + 1 \\ lev(prefix(s),prefix(t)) + 1_{\text{(Zeichen ungleich)}} \end{cases} & \text{sonst.} \end{cases}$$

Der Algorithmus arbeitet rekursiv, allerdings unter Zuhilfenahme der dynamischen Programmierung, d. h. es wird eine Matrix $D_{i,j}$ erstellt, wobei $1 \leq i \leq |s|$ und $1 \leq j \leq |t|$. Die Zeilen $1 \ldots |s|$ repräsentieren die einzelnen Zeichen des Wortes s und die Spalten $1 \ldots |t|$ die einzelnen Zeichen des Wortes t.

Die Matrix wird initial mit Werten gefüllt, die auf der untersten Rekursionsstufe zurückgegeben werden. Das initiale Füllen läuft folgendermaßen ab:

- Die 0te Zelle $D_{0,0}$ wird mit 0 initialisiert und beschreibt die Distanz zwischen zwei leeren Worten.
- Die Zellen $D_{0,1}$ bis $D_{0,|t|}$ werden mit der Stelligkeit ihrer Position im Wort t initialisiert, da der Abstand zwischen dem leeren Wort und einem Wort t nur $|t|$ sein kann.
- Entsprechend werden die Spalten $D_{1,0}$ bis $D_{|s|,0}$ ebenfalls nach der Stelligkeit im Wort s initialisiert, da der Abstand eines Wortes s zu einem leeren Wort nur $|s|$ sein kann.

Danach erfolgt das Befüllen rekursiv nach der oben angegebenen mathematischen Funktion $lev()$, wobei sich $D_{i,j}$ wie folgt berechnet: $D_{i,j} = lev(s_i, t_j)$; s_i, t_j beschreiben die Worte s,t bis zur Stelle i, j.

Korrektheit

Der Algorithmus sorgt dafür, dass die erste Zeile und Spalte initial gefüllt werden. Dadurch wird gewährleistet, dass das Ergebnis der Rekursion auf der untersten Stufe einen Wert zurück liefert, der auf der nächst höheren Stufe verwendet werden kann usw. Daraus folgt, dass der Algorithmus definitiv einen Wert zurück gibt. Dieser ist korrekt, da wie beschrieben sämtliche Möglichkeiten (Löschen, Einfügen, Ersetzen) stets in Betracht gezogen werden, dann jedoch nur das Minimum gewählt wird.

Laufzeit

Es wird mit einer Matrix gerechnet, jedoch mit dynamischer Programmierung, d. h. die Vergleiche (Einfügen, Ersetzen, Löschen) und die Additionen sind konstant. Es resultiert also eine Laufzeit von $\mathcal{O}(|s| \cdot |t|)$.

Teilaufgabe 4)

```
\lstinputlisting{}

Algorithmus Wechselgeld:
EINGABE:  n Anzahl der Muenzen, so dass gilt: Zur Verfuegung
    stehende Muenzwerte in Cent d_i sind d_1,...,d_n.
          m Das Wechselgeld als Cent-Betrag.

Initiales Fuellen der ersten Zeile mit 0, da C(0,x) = 0 und der
    ersten Spalte der nachfolgenden Zeilen mit 1, da C(j,0) = 1.

C := n × m-Matrix;

FOR i = 0 TO n DO
  FOR j = 1 TO m DO
    IF x - d_j < 0 THEN
      C(j,x) = C(j-1,x) + 0;
    ELSE
      C(j,x) = C(j-1,x) + C(j,x-d_j);
    ENDIF
  OD.
```

```
17  OD.
18
19  RETURN C(n,m);
```

Laufzeit
Zwei Schleifen mit Grenzen n bzw. m, wobei n dem j und m dem x aus der Aufgabenstellung entspricht, daraus resultiert: $\mathcal{O}(j \cdot x)$.

Teilaufgabe 2

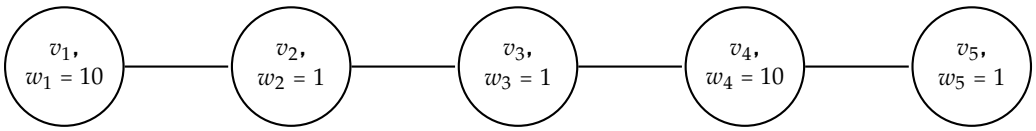
Weil der Algorithmus die Möglichkeiten für Wechselgeld x durch eine Anzahl an Münzen j einschränkt.

Aufgabe 3.2 Independent Set

Teilaufgabe 1

Der Graph wird zerlegt in die zwei Mengen S_1 und S_2. Jede Menge für sich ist ein Independent-Set, weil aufgrund des Modulo-Operators und der Tatsache, dass es sich um einen Pfad-Graph handelt, immer jeder zweite Knoten ausgelassen wird, sodass die verbleibenden Knoten ihre ursprünglich benachbarten Knoten – also Knoten zu denen sie eine Kante hatten – verlieren. Übrig bleiben also isolierte Knoten ohne Kanten dazwischen, demnach also ein Independent-Set. Da nun $S \in \{S_1, S_2\}$ gilt, ist S ebenfalls ein Independent Set.

Gegenbeispiel:



$S_1 = \{v_1, v_3, v_5\}$ mit einem Gesamtgewicht von 12.
$S_2 = \{v_2, v_4\}$ mit einem Gesamtgewicht von 11.
Daraus ergibt sich laut Algorithmus $S = S_1$ mit einem Gesamtgewicht 12. Optimal wäre jedoch die Menge $\{v_1, v_4\}$ mit einem Gesamtgewicht 20.

Teilaufgabe 2

```
1   Algorithmus Independent Set:
2   EINGABE:  Pfad-Graph G = (V,E) mit V = {v_1,...,v_n} und Gewichten
        w_1,...,w_n
3
4   Array W mit |W| = |V| + 2;  W[|V| + 1] := 0;  W[|V| + 2] := 0;
5   Mengen S_{|V|+1} = ∅ und S_{|V|+2} = ∅
6
7   FOR i = |V| TO 1 DO
8     IF w_i + W[i + 2] ≥ W[i + 1] THEN
9       W[i] = w_i + W[i + 2];
10      S_i = S_{i+2} ⋃ v_i;
11
12    ELSE
13      W[i] = W[i + 1];
14      S_i = S_{i+1};
15
```

```
    ENDIF
OD.

RETURN S = S_i;
```

Das Array $W[i]$ beschreibt die addierten Gewichte bis zum Knoten i rückwärts betrachtet – da der Algorithmus bei n beginnt.

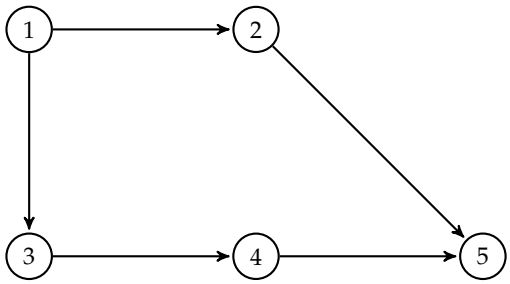
Laufzeit
Eine Schleife, die für jeweils einen Durchlauf konstant viele Schritte berechnet, daraus resultiert eine Laufzeit von $\mathcal{O}(|V|)$.

Aufgabe 3.3 Längste Pfade in sortierten Graphen

Teilaufgabe 1

Wir haben einen topologisch sortierten Graphen, was bedeutet, dass für einen Knoten v_i nur dann eine Ausgangskante existieren kann, wenn es einen Nachfolgerknoten v_{i+1} gibt, andernfalls würde die Bedingung $(v_i, v_j) \in E$ für $i < j$ nicht zutreffen, da v_j nicht existiert. Weil v_n keinen Nachfolgerknoten hat – der Knoten ist nach Definition der letzte – ist v_n immer der eindeutig bestimmte Knoten ohne ausgehende Kante.

Teilaufgabe 2



Dem Algorithmus nimmt nun den Pfad $1, 2, 5$ und erreicht damit eine Gesamtlänge $L = 2$. Optimal wäre jedoch der Pfad $1, 3, 4, 5$ mit einer Gesamtlänge von $L' = 3$.

Teilaufgabe 3

```
Algorithmus Laengster Pfad:
EINGABE:  Single Sink Graph G = (V,E) mit V = {v_1,...,v_n}

Array L der Laenge |V|. Initial gefuellt mit 0.
Leere Verkettete Liste K der Laenge |V|.

FOR jeden Knoten v w DO
  FOR jede Kante (v,w), von v nach w DO
    IF L[w] ≤ L[v] + 1 THEN
      L[w] = L[v_i];
      K[w] = K[v] ∪ w;
    ENDIF
  OD.
OD.

RETURN L;
```

Das Array $L[v_i]$ beschreibt die Länge vom Startknoten zum Knoten v_i. Die verkettete Liste K enthält den Pfad.

Teilaufgabe 4

Für beide Fälle ergibt sich eine Laufzeit von $\mathcal{O}(|V| + |E|)$.