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**: $lev(s,t) = lev(t, pref(t)) + 1$
- **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

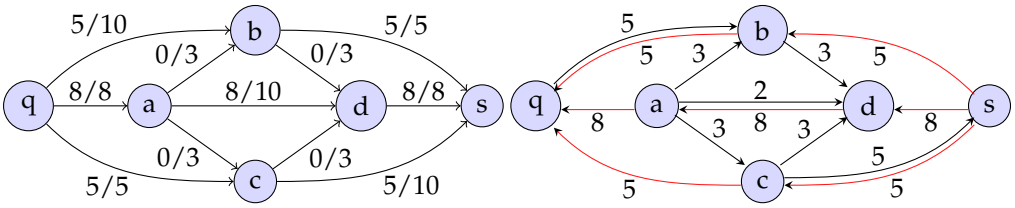
Es wird mit einer Matrix gerechnet, jedoch mit dynamischer Programmierung, d. h. die Vergleiche (Einfügen, Ersetzen, Löschen) und die Additionen sind konstant. Es resultiert also eine Laufzeit von $\mathcal{O}(|s| \cdot |t|)$.

Teilaufgabe 4)

```
public class LevDistance {

   //Kosten
   private static final double C1 = 1.0;
   private static final double C2 = 1.0;


   public double lev(String s, String t) {
      double D[][] = new double[s.length() + 1][t.length() + 1];

       //Initiales Fuellen

      //Zwei leere Woerter werden verglichen:
      D[0][0]=0;

      //Zellen D_{0,1} bis D_{1,|t|}
      for (int j = 1; j < t.length(); j++) {
         D[0][j] = j;
      }

      //Zellen D_{1,0} bis D_{|s|,0}
      for (int i = 1; i < s.length(); i++) {
         D[i][0] = i;
      }
      for (int i = 1; i <= s.length(); i++) {
         for (int j = 1; j <= t.length(); j++) {
            // Loeschen
            double delete = D[i - 1][j] + C1;
            // Einfuegen
            double insert = D[i][j - 1] + C1;
            // Ersetzen
            double replace = D[i - 1][j - 1];
            if (s.charAt(i - 1) != t.charAt(j - 1)) {
               replace += C2;
            } // Nur das Minimus wird aber ausgewaehlt!
            D[i][j] = Math.min(Math.min(delete, insert), replace);
         }
      } // Das Ergebnis befindet sich in Zelle D_{|s|}{|t|}
      return D[s.length()][t.length()];
   }
}
```

Weihnachtsfeier

Teilaufgabe 1)



Sei also $G = (V, E)$ der Graph, der die strikte Hierarchie des Unternehmens darstellt. Jeder Knoten $v_i \in V$ repräsentiert einen Mitarbeiter. Eine Kante $e_j \in V$ definiert zwischen zwei Mitarbeitern die Beziehung Vorgesetzter-Untergebener. Ein grüner Knoten repräsentiert hier einen Mitarbeiter, der teilnehmen kann, d. h. sein direkter Vorgesetzter erscheint nicht. Der Wert im Knoten repräsentiert den Wert $fun(v_i)$.

Es folgt aus der Modellierung, dass die Lösung ein **Independence Set** sein muss, andernfalls würde mindestens ein Mitarbeiter mit seinem direkten Vorgesetzten zur Party erscheinen, was nicht passieren darf. Die Lösung muss darüber hinaus die maximale Summe von Knotenwerten aufweisen.

Teilaufgabe 2)

Die Hauptidee hinter dem Algorithmus ist folgende: Man wendet eine rekursive Tiefensuche an und gibt zwei 2-Tupel zurück an den Aufrufenden: $((yes), \{v_i \dots v_j\}), ((no), \{v_k \dots v_l\})$. *yes* entspricht dem Spaßwert, falls dieser Mitarbeiter erscheint, *no*, falls nicht. Das zweite Tupelelement ist eine Menge, die die Mitarbeiter enthält, die mit dieser Entscheidung hinzukommen. Es handelt sich also auch um **Devide and Conquer**, da man das Problem auf jeder Höhe des Baumes durch die Anzahl der Kinder teilt, bis ganz unten an den Blättern nur noch einzelne Probleme gelöst werden müssen.

An einem Knoten – also einem gerade betrachteten Mitarbeiter w_i – kommt es nur zu folgenden Entscheidungsmöglichkeiten:

- Der Mitarbeiter w_i kommt zur Party: Bilde das *yes*-Tupel $(yes + fun(w_i), \{v_i \dots v_j\} \cup w_i)$, wobei sich der Wert *yes* aus der Summe der *no*-Werte seiner Untergebenen berechnet. Die Liste der Mitarbeiter $\{v_i \dots v_j\}$ ergibt sich aus der Vereinigung aller Mengen in den No-Tupeln seiner Untergebenen.
- Der Mitarbeiter w_i kommt nicht zur Party, man halt also die Wahl ob die Untergeben kommen können oder nicht: Summiere jeweils die *yes*- und die *no*-Werte der Untergebenen und wähle dann das Maximum. Gleichzeitig vereinige die Mengen in den *yes*- bzw. *no*-Tupel und bilde das *no*-Tupel mit dem maximalen Spaßwert und der Menge der dazugehörenden Mitarbeiter.

  Diese beiden Tupel werden zurückgegeben. Jeder Knoten ruft also diese Funktion auf jedes seiner Kinder auf, erhält die Tupel, bildet seine eigenen zwei Tupel entsprechend der Auswahlmöglichkeit oben und gibt sie zurück usw. Das geht solange bis die Rekursion an der Wurzel ist und ein endgültiges Tupel berechnet wurde, aus dem dann das Tupel ausgewählt wird mit dem maximalen Spaßwert.

  Korrektheit
  Der Algorithmus vergleicht in jedem Knoten zwei Fälle, nämlich den Fall, dass der Knoten in Betracht kommt und den Fall, dass nicht. Da der Knoten nur im zweiten Fall bei seinen Kindern den Fall heranzieht, dass diese in Betracht kommen, kann es nur gültige Ergebnisse im Sinne des **Independence Set** geben. Dazu kommt, dass

es in jedem Fall ein Ergebnis gibt, da die Rekursion in jedem Fall nach dem **Devide and Conquer**-Prinzip an den Blättern einen Wert zurück liefert – nämlich ein Tupel für eine Entscheidung mit diesem Knoten und eine ohne diesen Knoten v_i: *yes* und *no*-Tupel, wobei *yes*–Tupel:= $(fun(v_i), \{v_i\})$ und $no - Tupel := (0, \varnothing)$.

Laufzeit
In der Rekursion ruft man jeden Knoten genau einmal auf. Bei jedem Knoten gibt es eine Berechnung bzgl. der Summe und eines Maximalwertes. Hinzu kommen die Vereinigungen der Listen. Dieser Aufwand ist jedoch pro Knoten konstant. Abhängig von der Eingabe, angenommen eine Adjazenzliste, resultiert daraus also die Laufzeit $\mathcal{O}(n)$.