Fortgeschrittene Funktionale Programmierung

4. und 5. Vorlesung

Janis Voigtländer

Universität Bonn

Wintersemester 2015/16

Eine algorithmische Fragestellung

28. Bundeswettbewerb Informatik, 1. Runde, Aufgabe 2:

Szenario:

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MNO
7 PQRS	8 TUV	9 WXYZ
*	0 ␣	#

Gegeben: Buchstabenhäufigkeiten

Gesucht: optimale Tastenbelegung

Modellierung in Haskell

Buchstabenhäufigkeiten:

```
englisch :: [Integer]
englisch = [8167, 1492, 2782, 4253, 12702, 2228, 2015, 6095, ...]

test :: [Integer]
test = [1..5]
```

Datentyp für Tastenbelegungen:

```
data Partition = Entry Key Partition | EndP          deriving (Show, Eq)
data Key       = Letter Strokes Integer Key | EndK   deriving (Show, Eq)
type Strokes   = Int

example :: Partition
example = Entry (Letter 1 1 (Letter 2 2 EndK))
                (Entry (Letter 1 3 EndK)
                 (Entry (Letter 1 4 (Letter 2 5 EndK)) EndP))
```

Aufzählen des Suchraums

Von:

```
sequence :: [Integer]
sequence = test   -- könnte auch sein: englisch
```

zu:

```
example :: Partition
example = Entry (Letter 1 1 (Letter 2 2 EndK))
                (Entry (Letter 1 3 EndK)
                 (Entry (Letter 1 4 (Letter 2 5 EndK)) EndP))
```

und allen weiteren Möglichkeiten.

Das kann man als ein Parsing-Problem auffassen!

Aufzählen des Suchraums – Parserkombinatoren

```
phone :: [Partition]
phone = parse partition
  where
    partition = (key 1 ++> λk → partition ++>
                                  λp → yield (Entry k p)) |||
                (yield EndP)
    key c = item ++> λi → (key (c + 1) ||| yield EndK)
                          ++> λk → yield (Letter c i k)
```

```
phone :: [Partition]
phone = parse partition
  where
    partition = (Entry <<< key 1 ~~~ partition) |||
                (yield EndP)
    key c = Letter c <<< item ~~~ (key (c + 1) ||| yield EndK)
```

Aufzählen des Suchraums – Parserkombinatoren

$$\Downarrow$$

```
phone :: [Partition]
phone = parse partition
  where
    partition = (Entry <<< key 1 ~~~ partition) |||
                (yield EndP)
    key c = Letter c <<< item ~~~ (key (c + 1) ||| yield EndK)
```

wobei:

$(\lll) :: (a \to b) \to \text{Parser } a \to \text{Parser } b$
$f \lll p = p \mathbin{+\!\!+\!\!>} \lambda a \to \texttt{yield}\ (f\ a)$

$(\sim\!\sim\!\sim) :: \text{Parser } (a \to b) \to \text{Parser } a \to \text{Parser } b$
$p \sim\!\sim\!\sim q = p \mathbin{+\!\!+\!\!>} \lambda f \to q \mathbin{+\!\!+\!\!>} \lambda a \to \texttt{yield}\ (f\ a)$

Aufzählen des Suchraums – Parserkombinatoren

$$\Downarrow$$

```
phone :: [Partition]
phone = parse partition
  where
    partition = (Entry <<< key 1 ~~~ partition) |||
                (yield EndP)
    key c = Letter c <<< item ~~~ (key (c + 1) ||| yield EndK)
```

Zur Erinnerung:

```
data Partition = Entry Key Partition | EndP
data Key       = Letter Strokes Integer Key | EndK
```

Aufzählen des Suchraums

Mittels soeben gesehener Definition von `phone`:

```
> head phone   -- mit sequence = [1..5]
Entry (Letter 1 1 (Letter 2 2 (Letter 3 3 (Letter 4 4 (...)))))  EndP

> last phone   -- mit sequence = [1..5]
Entry (Letter 1 1 EndK)
(Entry (Letter 1 2 EndK)
 (Entry (Letter 1 3 EndK)
  (Entry (Letter 1 4 EndK)
   (Entry (Letter 1 5 EndK) EndP))))
```

Oder auch:

```
> phone !! 6      -- mit sequence = [1..5]
Entry (Letter 1 1 (Letter 2 2 EndK))
(Entry (Letter 1 3 EndK)
 (Entry (Letter 1 4 (Letter 2 5 EndK)) EndP))
```

Sehen Sie ein Problem?

Berücksichtigen der Tastenanzahl

```
phone :: Int → [Partition]
phone k = parse (partition k)
  where
    partition k | k > 0 = Entry ⋘ key 1 ~~~ partition (k − 1)
    partition 0         = yield EndP
    key c = Letter c ⋘ item ~~~ (key (c + 1) ||| yield EndK)
```

```
> last (phone 3)                    -- mit sequence = [1..5]
Entry (Letter 1 1 EndK)
(Entry (Letter 1 2 EndK)
 (Entry (Letter 1 3 (Letter 2 4 (Letter 3 5 EndK))) EndP))
```

```
> (phone 3) !! 2 == example    -- mit sequence = [1..5]
True
```

Bewerten/Analysieren aufgezählter Kandidaten

Offenbar müssen wir alle Kandidaten wie

Entry (Letter 1 1 (Letter 2 2 EndK))
(Entry (Letter 1 3 EndK)
 (Entry (Letter 1 4 (Letter 2 5 EndK)) EndP))

geeignet „auswerten“, per struktureller Rekursion.

Statt erst alle Kandidaten aufzuzählen und dann einzeln zu analysieren/traversieren, Parametrisierung des Parsers:

```
phone (entry, endP, letter, endK) k = parse (partition k)
  where
    partition k | k > 0 = entry <<< key 1 ~~~ partition (k - 1)
    partition 0         = yield endP
    key c = letter c <<< item ~~~ (key (c + 1) ||| yield endK)

> (phone (Entry, EndP, Letter, EndK) 3) !! 2 == example   -- ...
True
```

Bewerten/Analysieren

Konkrete Rechnungen durch alternative Wahl der Parameter:

```
evaluate = phone (entry, endP, letter, endK)
  where
    entry k p     = k + p
    endP          = 0
    letter c i k  = c * i + k
    endK          = 0
```

```
> evaluate 3   -- mit sequence = [1..5]
[23, 21, 22, 26, 23, 29]
```

```
> :t phone
phone :: Num a =>
         (b -> c -> c, c, a -> Integer -> b -> b, b) -> Int -> [c]
```

Alternative Ausgabeformate

```
compact = phone (entry, endP, letter, endK)
  where
    entry k p    = k : p
    endP         = []
    letter _ i k = i : k
    endK         = []

> compact 3   -- mit sequence = [1..5]
[[[1,2,3],[4],[5]],[[1,2],[3,4],[5]],[[1,2],[3],[4,5]],
 [[1],[2,3,4],[5]],[[1],[2,3],[4,5]],[[1],[2],[3,4,5]]]

profile = phone (entry, endP, letter, endK)
  where
    entry k p    = k : p
    endP         = []
    letter _ _ k = 1 + k
    endK         = 0
```

Alternative Ausgabeformate

```
compact = phone (entry, endP, letter, endK)
  where
    entry k p    = k : p
    endP         = []
    letter _ i k = i : k
    endK         = []
```

```
> compact 3   -- mit sequence = [1..5]
[[[1,2,3],[4],[5]],[[1,2],[3,4],[5]],[[1,2],[3],[4,5]],
 [[1],[2,3,4],[5]],[[1],[2,3],[4,5]],[[1],[2],[3,4,5]]]

> profile 3   -- mit sequence = [1..5]
[[3,1,1],[2,2,1],[2,1,2],[1,3,1],[1,2,2],[1,1,3]]
```

Ausnutzen des Optimalitätsprinzips

Wie können wir die Berechnung aller Zwischenergebnisse von

```
> minimum (evaluate 3)   -- mit sequence = [1..5]
21
```

vermeiden?

Schon über Zwischenlösungen minimieren:

```
phone (entry, endP, letter, endK, h) k = parse (partition k)
  where
    partition k | k > 0 = entry <<< key 1
                                  ~~~ partition (k - 1) ... h
    partition 0         = yield endP
    key c = letter c <<< item ~~~ (key (c + 1) ||| yield endK)

optimize = phone (entry, endP, letter, endK, choose)
  where
    (entry, endP, letter, endK) = ...   -- aus evaluate
    choose l = if null l then [] else [minimum l]
```

Ausnutzen des Optimalitätsprinzips

Schon über Zwischenlösungen minimieren:

```
phone (entry, endP, letter, endK, h) k = parse (partition k)
  where
    partition k | k > 0 = entry <<< key 1
                                ~~~ partition (k - 1) ... h
    partition 0         = yield endP
    key c = letter c <<< item ~~~ (key (c + 1) ||| yield endK)

optimize = phone (entry, endP, letter, endK, choose)
  where
    (entry, endP, letter, endK) = ...   -- aus evaluate
    choose l = if null l then [] else [minimum l]

> optimize 3   -- mit sequence = [1..5]
[21]

(...) :: Parser a → ([a] → [a]) → Parser a
```

Analysieren des Suchraums

```
count = phone (entry, endP, ⊥, ⊥, h)
  where
    entry _ p = p
    endP      = 1
    h l       = [sum l]
```

```
> count 3   -- mit sequence = [1..5]
[6]
```

Ab jetzt:

```
sequence = englisch
```

```
> optimize 8
[164682]   -- nach 2–3 Minuten
```

```
> count 8
[480700]   -- ...
```

Was sagt der Bundeswettbewerb Informatik dazu?

Aus der Musterlösung:

Da es also „nur“ $\binom{N-1}{K-1} = 480\,700$ mögliche Belegungen gibt, kann man diese durchprobieren, jeweils die Kosten berechnen und die optimale Belegung finden.

Dynamische Programmierung

Insbesondere für andere Konstellationen mit mehr Buchstaben und mehr Tasten wäre ein effizienteres Verfahren sehr wichtig. Es gibt netterweise ein Verfahren mit Dynamischer Programmierung, das die Lösung mit einem Berechnungsaufwand von $O(N^2K)$ findet. Man beachte,

. . .

Nun erstellt man eine Tabelle, in der man für jede Taste i und jeden Buchstaben j speichert, wie die optimalen Kosten wären, wenn es weder frühere Tasten noch frühere Buchstaben gäbe. Für ein solches Paar (i, j) berechnet man diese Kosten, indem man für jeden späteren Buchstaben l die Kosten berechnet, die entstehen, wenn l der erste Buchstabe auf der Taste $i+1$ wäre, und davon das Minimum wählt.

Was sagt der Bundeswettbewerb Informatik dazu?

Pseudocode:

1: **for** $j = 1$ to N **do**

2: $\quad COSTS[K][j] \leftarrow \left(-j \sum_{l=j}^{N} h_l \right)$

3: **end for**

4: **for** $i = K-1$ to 1 **do**

5: $\quad$ **for** $j = 1$ to N **do**

6: $\quad\quad COSTS[i][j] \leftarrow \min \left\{ COSTS[i+1][l] - \left(j \sum_{m=j}^{l-1} h_m \right) \middle| \, l > j \right\}$

7: $\quad$ **end for**

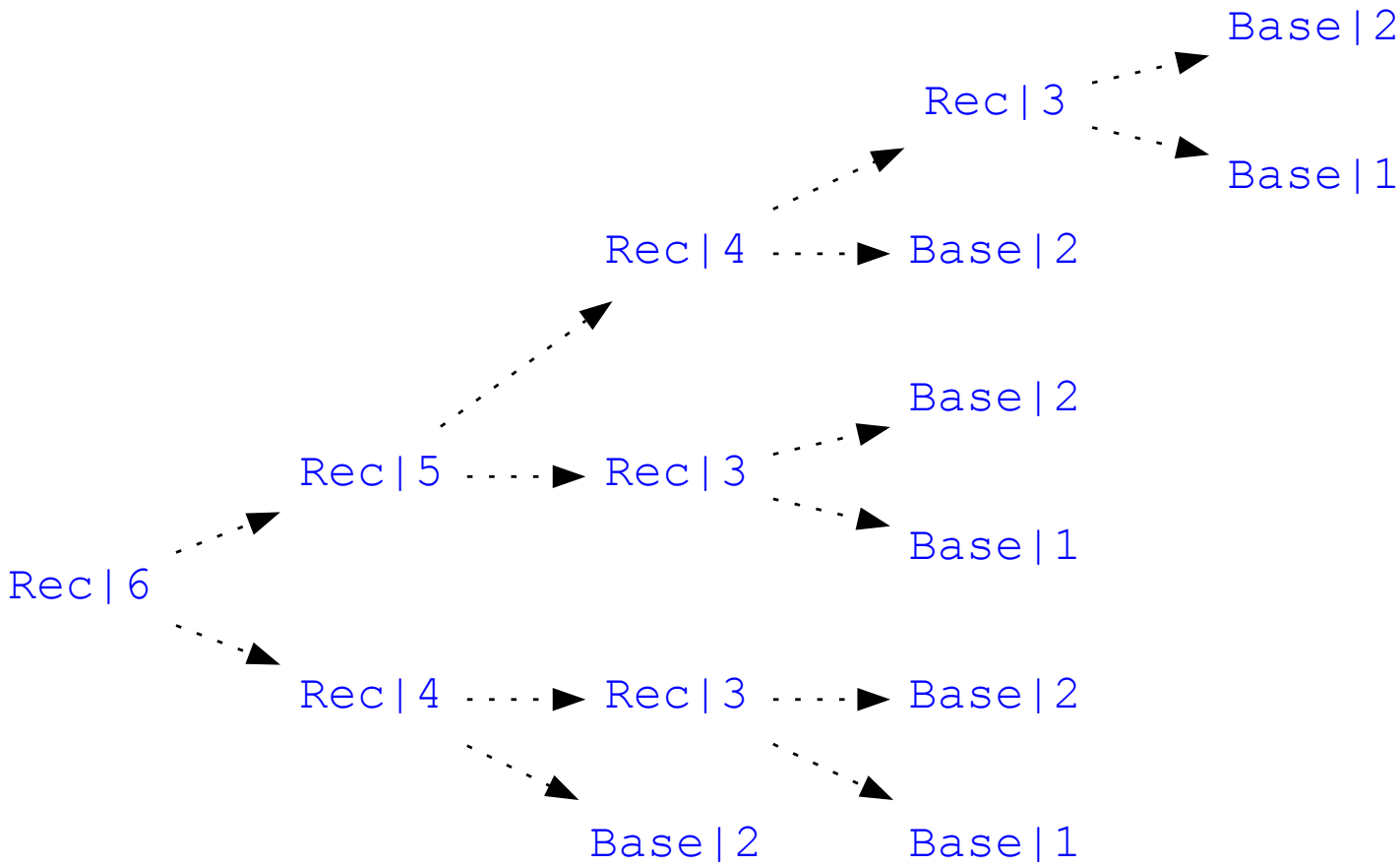
8: **end for**

Können wir (Haskell) das vielleicht besser/eleganter?

Zur Erinnerung

```
fib :: Int → Int
fib n | n < 2 = 1
fib n         = fib (n − 2) + fib (n − 1)
```

Rekursiver „Aufrufgraph“, für `fib 6`:

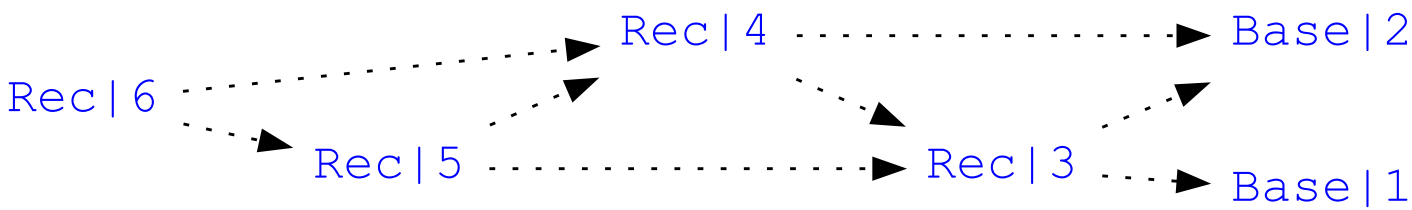


Zur Erinnerung – Memoisierung

```
mfib = memo fib

fib :: Int → Int
fib n | n < 2 = 1
fib n         = mfib (n − 2) + mfib (n − 1)
```

„Aufrufgraph“ jetzt:



Wäre doch schön, wenn wir ähnliche „Magie“ auch bei `phone` entfalten könnten!

Dynamische Programmierung „bei uns“ – Challenge

Minimal invasive Änderung des Programms:

```
phone (entry, endP, letter, endK, h) k = parse (p (partition k))
  where
    partition k | k > 0
      = tabulated (entry <<< key 1 ~~~ p (partition (k - 1)) ... h)
    partition 0 = tabulated (yield endP)
    key c = letter c <<< item ~~~ (key (c + 1) ||| yield endK)
```

```
> optimize 8
[164682]   -- nach weniger als 1 Sekunde
```

Wiederverwendbar definiert:

```
type Table a = ...
tabulated :: Parser a -> Table a
p         :: Table a  -> Parser a
```

Etwas Reflexion

Aktuelle Version:

```
phone (entry, endP, letter, endK, h) k = parse (p (partition k))
  where
    partition k | k > 0
       = tabulated (entry <<< key 1 ~~~ p (partition (k − 1)) ... h)
    partition 0 = tabulated (yield endP)
    key c = letter c <<< item ~~~ (key (c + 1) ||| yield endK)
```

- Beschreibung des Suchraums – Parserkombinatoren
- flexible Anwendung des Optimalitätsprinzips (... *h*)
- flexible Tabellierung durch (orthogonale) Annotation
- unabhängige Beschreibung verschiedener Analysen:

```
optimize = phone (entry, endP, letter, endK, choose)
  where
    entry = ...
    ...   = ...
```

Nochmal zum Vergleich

Pseudocode aus der Musterlösung des BW Informatik:

1: **for** $j = 1$ to N **do**
2: $\quad COSTS[K][j] \leftarrow \left(-j \sum_{l=j}^{N} h_l \right)$
3: **end for**
4: **for** $i = K - 1$ to 1 **do**
5: $\quad$ **for** $j = 1$ to N **do**
6: $\quad\quad COSTS[i][j] \leftarrow \min \left\{ COSTS[i+1][l] - \left(j \sum_{m=j}^{l-1} h_m \right) \middle| \, l > j \right\}$
7: $\quad$ **end for**
8: **end for**

Anpassbarkeit unserer Lösung

> `optimize` 8
[164682]

Angenommen, es gibt Vorgaben für Mindest-/Maximalanzahl an Buchstaben pro Taste... Übung

Anpassbarkeit unserer Lösung

> optimize 8 1 26
[164682]

> optimize 8 1 6
[165078]

> optimize 8 3 26
[181222]

Oder auch:

sequence = deutsch

> optimize 8 1 26
[158780]

> optimize 8 1 6
[162970]

Backtracing?

Gut und schön:

$>$ `optimize` 8

$[164682]$ -- nach weniger als 1 Sekunde

Aber wie erhält man die optimale Belegung (statt ihrer Kosten)?

Wieder aus der Musterlösung des Bundeswettbewerbs Informatik:

...

> Damit erhält man die Kosten der optimalen Belegung. Die Belegung selbst erhält man, indem man in der Tabelle außer dem Minimum noch den Index des minimalen Elementes speichert. Dann kann man vom Feld $(1,1)$ ausgehen (der erste Buchstabe muss ja auf der ersten Position der ersten Taste stehen: $k_1 = 1$) und findet dort den optimalen ersten Buchstaben für Taste 2, k_2. In Feld $(2,k_2)$ steht dann der optimale erste Buchstabe für die dritte Taste usw.

Und wie machen wir das?

Backtracing?

Wieder aus der Musterlösung des Bundeswettbewerbs Informatik:

...

> Damit erhält man die Kosten der optimalen Belegung. Die Belegung selbst erhält man, indem man in der Tabelle außer dem Minimum noch den Index des minimalen Elementes speichert. Dann kann man vom Feld $(1,1)$ ausgehen (der erste Buchstabe muss ja auf der ersten Position der ersten Taste stehen: $k_1 = 1$) und findet dort den optimalen ersten Buchstaben für Taste 2, k_2. In Feld $(2,k_2)$ steht dann der optimale erste Buchstabe für die dritte Taste usw.

Bei uns, allgemeine Kombination von „Auswertungsmodi“:

$$
\begin{aligned}
&(entry_1, endP_1, letter_1, endK_1, h_1) \mathbin{***} (entry_2, \ldots, endK_2, h_2) \\
&\quad = (entry, endP, letter, endK, h) \\
&\qquad \textbf{where} \\
&\qquad\quad entry\ (k_1, k_2)\ (p_1, p_2) = (entry_1\ k_1\ p_1, entry_2\ k_2\ p_2) \\
&\qquad\quad \ldots \qquad\qquad\qquad\quad = \ldots \\
&\qquad\quad endK \qquad\qquad\qquad\quad = (endK_1, endK_2) \\
&\qquad\quad h\ l = [(x,y) \mid x \leftarrow h_1\ [x \mid (x,\_) \leftarrow l], \\
&\qquad\qquad\qquad\qquad\quad y \leftarrow h_2\ [y \mid (x',y) \leftarrow l, x' == x]]
\end{aligned}
$$

Backtracing?

Bei uns, allgemeine Kombination von „Auswertungsmodi“:

$$(entry_1, endP_1, letter_1, endK_1, h_1) \mathbin{***} (entry_2, \ldots, endK_2, h_2)$$
$$= (entry, endP, letter, endK, h)$$
where
$$entry\ (k_1, k_2)\ (p_1, p_2) = (entry_1\ k_1\ p_1, entry_2\ k_2\ p_2)$$
$$\ldots = \ldots$$
$$endK = (endK_1, endK_2)$$
$$h\ l = [(x, y) \mid x \leftarrow h_1\ [x \mid (x, \_) \leftarrow l],$$
$$y \leftarrow h_2\ [y \mid (x', y) \leftarrow l, x' == x]]$$

```
opt_prof = phone (algebra1 *** algebra2)
  where
    algebra1 = ...   -- aus optimize
    algebra2 = ...   -- aus profile, plus id für choose
```

```
> opt_prof 8
[(164682, [2, 2, 3, 4, 2, 4, 2, 7])]   -- ggfs. auch mehrere Profile
```

Einige Beobachtungen

- Spezifikation auf hohem konzeptionellen Niveau
- Abstraktionsmechanismen, hohes Wiederverwendungspotential
- Klare Separierung der für dynamische Programmierung relevanten Aspekte:
  - Beschreibung des Suchraums (durch Parserkombinatoren)
  - Optimalitätsprinzip von Bellman (selektiv anwendbar)
  - Tabellierung für Effizienz (ohne Änderung der Codestruktur)
  - Formulierung, Kombination verschiedener Analysen (durch Algebren)
- Exploration von Problemen/Algorithmen
- weitere Abstraktionen möglich, etwa generisches Abzählen, Aufzählen, Algebrenprodukt, ...

Eine andere typische Anwendung von DP

Wie sieht es aus mit Problemen auf zwei Sequenzen?

Zum Beispiel `diff`/Edit-Distanz?

Beispiel: Eingabe „abcbcba“ und „acbda“
$\rightsquigarrow$ mögliche Erklärungen des Unterschieds:

a	b	c	b	c	b	a
a	c	b	d	a	–	–

a	b	c	b	c	b	–	a
a	–	–	–	c	b	d	a

a	b	c	b	c	b	a
a	–	c	b	d	a	–

Vergleichsmaß: Anzahl von Operationen zur Löschung, Einfügung oder Änderung eines Zeichens

Klassische Lösung

Idee:

- Speicherung aller Distanzen $d_{i,j}$ (zwischen `s[i]...s[n-1]` und `t[j]...t[m-1]`) in Tabelle
- Berechnung in geeigneter Reihenfolge

Beispiel:

	a	b	c	b	c	b	a	
a	3	3	3	2	2	3	4	5
c	4	3	2	2	1	2	3	4
b	5	4	3	2	2	1	2	3
d	6	5	4	3	2	1	1	2
a	6	5	4	3	2	1	0	1
	7	6	5	4	3	2	1	0

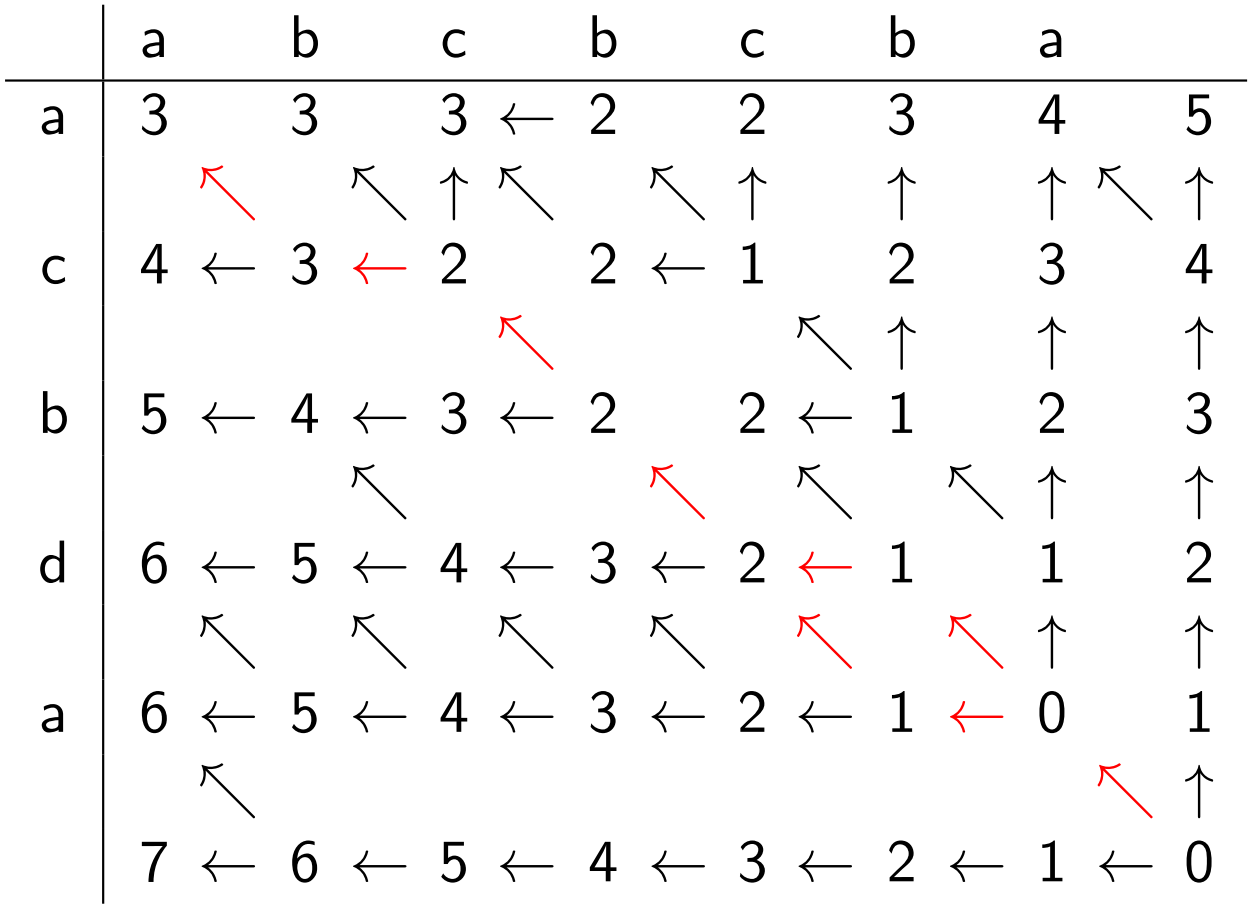
$$d_{i,j} = \begin{cases} m - j & \text{wenn } i = n \\ n - i & \text{wenn } j = m \\ min\{1 + d_{i+1,j}\,;\, 1 + d_{i,j+1}\,;\, d_{i+1,j+1}\} & \text{wenn } \texttt{s[i]==t[j]} \\ 1 + min\{d_{i+1,j}\,;\, d_{i,j+1}\,;\, d_{i+1,j+1}\} & \text{sonst} \end{cases}$$

Backtracing

Problem: Auffinden der Anordnung für die minimale Distanz

Ansatz: „Herkunft“ der minimalen Werte in Tabelle festhalten

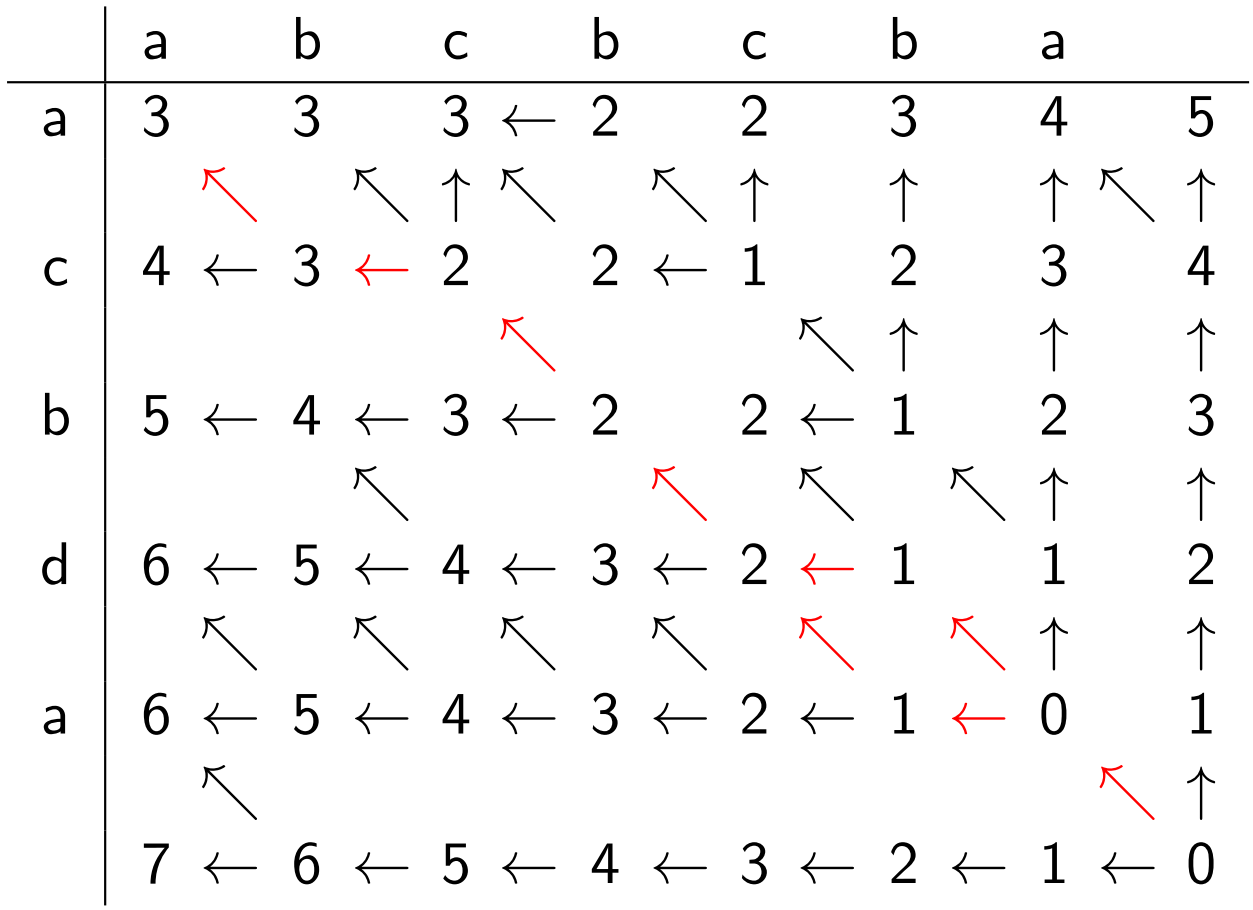
Beispiel:



Lösung: Verfolgen aller Wege von $d_{0,0}$ nach $d_{n,m}$

Backtracing

Beispiel:



Ablesen:

a	b	c	b	c	b	a
a	–	c	b	–	d	a

a	b	c	b	c	b	a
a	–	c	b	d	–	a

Und in unserer Haskell-Lösung?

Wie gehen wir jetzt damit um, dass es sich um zwei Sequenzen als Eingabe handelt?

Wir haben ja „nur“:

```
sequence :: [Integer]
```

```
parse :: Parser a → [a]
item  :: Parser Integer
...
```

Ein einfacher „Hack“:

```
sequence :: [Integer]
sequence = sequence₁ ++ [-1] ++ reverse sequence₂
           where sequence₁ = ...
                 sequence₂ = ...
```

Und in unserer Haskell-Lösung?

Wir haben ja „nur“:

```
sequence :: [Integer]
parse :: Parser a → [a]
item  :: Parser Integer
...
```

Ein einfacher „Hack“:

```
sequence :: [Integer]
sequence = sequence1 ++ [-1] ++ reverse sequence2
           where sequence1 = ...
                 sequence2 = ...


delimiter :: Parser ()
...
```

Edit-Distanz algebraisch modelliert

```
data Alignment = Nil () | D Integer Alignment | I Alignment Integer |
                 R Integer Alignment Integer

diff (nil, d, i, r, h) = parse (p alignment)
  where alignment = tabulated
                      (nil <<< delimiter |||
                       d   <<< item ~~~ p alignment |||
                       i   <<< p alignment ~~~ item |||
                       r   <<< item ~~~ p alignment ~~~ item
                       ... h)

alg_optimize = (nil, d, i, r, h)
  where nil _  = 0
        (d, i) = (λa s → s + 1, λs b → s + 1)
        r a s b = if a == b then s else s + 1
        h l     = if null l then [] else [minimum l]
```

Edit-Distanz algebraisch modelliert

```
diff (nil, d, i, r, h) = parse (p alignment)
  where alignment = tabulated
                      (nil <<< delimiter |||
                       d   <<< item ~~~ p alignment |||
                       i   <<< p alignment ~~~ item |||
                       r   <<< item ~~~ p alignment ~~~ item
                       ... h)

alg_optimize = (nil, d, i, r, h)
  where nil _  = 0
        (d, i) = (λa s → s + 1, λs b → s + 1)
        r a s b = if a == b then s else s + 1
        h l     = if null l then [] else [minimum l]
```

Leider:

```
> diff alg_optimize
...   -- terminiert nicht
```

Ein kleiner Eingriff

```
diff (nil, d, i, r, h) = parse (p alignment)
  where alignment = tabulated
                      (nil <<< delimiter |||
                       d   <<< item -~~ p alignment |||
                       i   <<< p alignment ~~- item |||
                       r   <<< item -~~ p alignment ~~- item
                       ... h)
```

unter Verwendung spezieller Kombinatoren (-~~) und (~~-) als Spezialfälle von (~~~), die jeweils auf einer Seite genau ein Element der Sequenz zulassen.

Nun:

```
> diff alg_optimize   -- mit sequence_1 = [1,2,3,2,3,2,1]
                      --      sequence_2 = [1,3,2,4,1]
[3]
```

Diskussion möglicher Anpassungen

1. Mismatch zwischen zwei „Symbolen“ a und b kostet nicht immer gleich viel, sondern wird abhängig von a und b unterschiedlich stark „bestraft“.

2. Die „Bestrafung“ für eine Lücke hängt durch eine lineare Funktion von deren Länge ab, jedoch nicht rein proportional.

3. Es wird nicht geringste globale Abweichung sondern beste lokale Ähnlichkeit gesucht.
   Genauer: Statt Mismatches mit $+1$ und Matches mit 0 zu bewerten, werden Mismatches mit $+1$ und Matches mit -1 bewertet. Und statt zu fragen, was der minimale Wert bei Alignment der beiden Eingabesequenzen als Ganzes ist, wird gefragt, was der minimale (ggfs. negative) Wert bei Alignment irgendeiner Teilsequenz der ersten Eingabesequenz mit irgendeiner Teilsequenz der zweiten Eingabesequenz ist.

Literatur

Robert Giegerich, Carsten Meyer, and Peter Steffen.
Towards a discipline of dynamic programming.
In Sigrid E. Schubert, Bernd Reusch, and Norbert Jesse, editors, *GI Jahrestagung*, volume 19 of *LNI*, pages 3–44. GI, 2002.

Robert Giegerich, Carsten Meyer, and Peter Steffen.
A discipline of dynamic programming over sequence data.
Science of Computer Programming, 51(3):215–263, 2004.