Algebraische Dynamische Programmierung

Janis Voigtländer

Universität Bonn

13. Juli 2012

28. Bundeswettbewerb Informatik, 1. Runde, Aufgabe 2

Handytasten:

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

28. Bundeswettbewerb Informatik, 1. Runde, Aufgabe 2

Handytasten:

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

Gegeben: Buchstabenhäufigkeiten

28. Bundeswettbewerb Informatik, 1. Runde, Aufgabe 2

Handytasten:

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

Gegeben: Buchstabenhäufigkeiten

Gesucht: optimale Tastenbelegung

Musterlösung des BWINF

imperativer 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**

Musterlösung des BWINF

imperativer 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?

Modellieren in Haskell

Buchstabenhäufigkeiten:

$$\mathit{englisch} :: [\mathsf{Integer}]$$
$$\mathit{englisch} = [8167, 1492, 2782, 4253, 12702, 2228, 2015, 6095, \ldots]$$

Modellieren in Haskell

Buchstabenhäufigkeiten:

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

```
test :: [Integer]
test = [1..5]   -- Abkürzung für [1, 2, 3, 4, 5]
```

Modellieren in Haskell

Buchstabenhäufigkeiten:

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

```
test :: [Integer]
test = [1..5]   -- Abkürzung für [1, 2, 3, 4, 5]
```

Datentyp für Tastenbelegungen:

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

Modellieren in Haskell

Buchstabenhäufigkeiten:

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

```
test :: [Integer]
test = [1..5]   -- Abkürzung für [1, 2, 3, 4, 5]
```

Datentyp für Tastenbelegungen:

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

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.

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

Das kann man als ein Parsing-Problem auffassen!

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

Aufzählen des Suchraums — Parserkombinatoren

Das kann man als ein Parsing-Problem auffassen!

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

Zur Erinnerung:

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

Aufzählen des Suchraums — Parserkombinatoren

Das kann man als ein Parsing-Problem auffassen!

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

Verwendet, aus einer Haskell-Bibliothek:

```
type SubSeq  = (Int, Int)   -- z.B.: (0, length sequence)
type Parser a = SubSeq → [a]

parse    :: Parser a → [a]
listItem :: Parser Integer
empty    :: a → Parser a
(|||)    :: Parser a → Parser a → Parser a
(<<<), (~~~) :: ...
```

Aufzählen des Suchraums

Mittels soeben gesehener Definition von *phone*:

```
> head 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))))
```

Aufzählen des Suchraums

Mittels soeben gesehener Definition von *phone*:

```
> head 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 !! 9   -- 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))
```

Aufzählen des Suchraums

Mittels soeben gesehener Definition von *phone*:

```
> head 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 !! 9   -- 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))
```

Erkennt jemand 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         = empty EndP

    key c = Letter c <<< listItem ~~~ (key (c + 1) ||| empty EndK)
```

Berücksichtigen der Tastenanzahl

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

```
> head (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))
```

Bewerten/Analysieren aufgezählter Kandidaten

Offenbar müssen wir auf 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 „rechnen“, per Traversierung.

Bewerten/Analysieren aufgezählter Kandidaten

Offenbar müssen wir auf 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 „rechnen“, per Traversierung.

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         = empty endP

    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty endK)
```

Bewerten/Analysieren aufgezählter Kandidaten

Offenbar müssen wir auf 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 „rechnen“, per Traversierung.

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         = empty endP

    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty endK)
```

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

Bewerten/Analysieren

Konkrete Rechnungen durch alternative Wahl der Parameter:

$$
\begin{aligned}
&\mathit{evaluate} = \mathit{phone}\ (\mathit{entry}, \mathit{endP}, \mathit{letter}, \mathit{endK}) \\
&\quad \textbf{where} \\
&\qquad \mathit{entry}\ k\ p \;\; = k + p \\
&\qquad \mathit{endP} \qquad = 0 \\
&\qquad \mathit{letter}\ c\ i\ k = c * i + k \\
&\qquad \mathit{endK} \qquad = 0
\end{aligned}
$$

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]
[29, 23, 26, 22, 21, 23]
```

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]
[29, 23, 26, 22, 21, 23]
```

```
> :t phone
phone
  :: (Num a, Num b, Ord b) ⇒
    (c → d → d, d, a → Integer → c → c, c) → b → [d]
```

Alternative Ausgabeformate

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

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]]]
```

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]
[[1,1,3],[1,2,2],[1,3,1],[2,1,2],[2,2,1],[3,1,1]]
```

Ausnutzen des Optimalitätsprinzips

Wie können wir Berechnung aller Zwischenergebnisse von

$> \mathit{minimum}\ (\mathit{evaluate}\ 3)$ -- mit $\mathit{sequence} = [1\,..\,5]$
21

vermeiden?

Ausnutzen des Optimalitätsprinzips

Wie können wir 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         = empty endP

    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty endK)
```

Ausnutzen des Optimalitätsprinzips

Wie können wir 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         = empty endP
    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty endK)
```

```
optimize = phone (entry, endP, letter, endK, choose)
  where
    (entry, endP, letter, endK) = ...   -- aus evaluate
    choose l = if 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         = empty endP
    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty endK)
```

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

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

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         = empty endP
    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty endK)
```

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

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]
```

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
```

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 etwa 3 Minuten
```

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 etwa 3 Minuten
```

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

Was sagt denn 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,

. . .

Was sagt denn 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 denn der Bundeswettbewerb Informatik dazu?

imperativer 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**

Dynamische Programmierung „bei uns“

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 (empty endP)
    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty endK)
```

Dynamische Programmierung „bei uns“

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 (empty endP)
    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty endK)
```

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

Dynamische Programmierung „bei uns“

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 (empty endP)
    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty endK)
```

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

Wiederverwendbar definiert:

```
type Table a = ...
tabulated :: Parser a → Table a
p         :: Table a  → Parser a
```

Backtracing?

Zurück zur Handytasten-Aufgabe:

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

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

Backtracing?

Zurück zur Handytasten-Aufgabe:

$>$ *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.

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“:

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

$$
\begin{aligned}
&opt\_prof = phone\ (algebra_1 \mathbin{***} algebra_2) \\
&\quad \textbf{where} \\
&\qquad algebra_1 = \ldots \quad \text{-- aus } optimize \\
&\qquad algebra_2 = \ldots \quad \text{-- aus } profile
\end{aligned}
$$

Backtracing?

Bei uns, allgemeine Kombination von „Auswertungsmodi“:

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

$$
\begin{aligned}
&opt\_prof = phone\ (algebra_1 \mathbin{***} algebra_2) \\
&\quad \textbf{where} \\
&\qquad algebra_1 = \ldots \quad \text{-- aus } optimize \\
&\qquad algebra_2 = \ldots \quad \text{-- aus } profile
\end{aligned}
$$

$$
\begin{aligned}
&> opt\_prof\ 8 \\
&[(164682, [2, 2, 3, 4, 2, 4, 2, 7])] \quad \text{-- ggfs. auch mehrere Profile}
\end{aligned}
$$

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 (empty endP)
    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty endK)
```

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 (empty endP)
    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty endK)
```

- Beschreibung des Suchraums — Parserkombinatoren

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 (empty endP)
    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty endK)
```

- Beschreibung des Suchraums — Parserkombinatoren
- flexible Anwendung des Optimalitätsprinzips (`... h`)

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 (empty endP)
    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty endK)
```

- Beschreibung des Suchraums — Parserkombinatoren
- flexible Anwendung des Optimalitätsprinzips (... *h*)
- flexible Tabellierung durch (orthogonale) Annotation

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 (empty endP)
    key c = letter c <<< listItem ~~~ (key (c + 1) ||| empty 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

imperativer Pseudocode aus der Musterlösung des BWINF:

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

$> \mathit{opt\_prof}\ 8$
$[(164682, [2, 2, 3, 4, 2, 4, 2, 7])]$

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

Anpassbarkeit unserer Lösung

```
> opt_prof 8
[(164682,[2,2,3,4,2,4,2,7])]
```

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

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

Anpassbarkeit unserer Lösung

```
> opt_prof 8 1 26
[(164682, [2, 2, 3, 4, 2, 4, 2, 7])]
```

Anpassbarkeit unserer Lösung

```
> opt_prof 8 1 26
[(164682, [2, 2, 3, 4, 2, 4, 2, 7])]
```

```
> opt_prof 8 1 6
[(165078, [4, 3, 4, 2, 4, 2, 3, 4])]
```

Anpassbarkeit unserer Lösung

```
> opt_prof 8 1 26
[(164682, [2, 2, 3, 4, 2, 4, 2, 7])]
```

```
> opt_prof 8 1 6
[(165078, [4, 3, 4, 2, 4, 2, 3, 4])]
```

```
> opt_prof 8 3 26
[(181222, [4, 3, 3, 3, 3, 3, 3, 4])]
```

Anpassbarkeit unserer Lösung

```
> opt_prof 8 1 26
[(164682, [2, 2, 3, 4, 2, 4, 2, 7])]
```

```
> opt_prof 8 1 6
[(165078, [4, 3, 4, 2, 4, 2, 3, 4])]
```

```
> opt_prof 8 3 26
[(181222, [4, 3, 3, 3, 3, 3, 3, 4])]
```

Oder auch:

```
sequence = deutsch
```

```
> opt_prof 8 1 26
[(158780, [2, 2, 3, 4, 2, 4, 2, 7])]
```

Anpassbarkeit unserer Lösung

$> \mathit{opt\_prof}\ 8\ 1\ 26$
$[(164682, [2, 2, 3, 4, 2, 4, 2, 7])]$

$> \mathit{opt\_prof}\ 8\ 1\ 6$
$[(165078, [4, 3, 4, 2, 4, 2, 3, 4])]$

$> \mathit{opt\_prof}\ 8\ 3\ 26$
$[(181222, [4, 3, 3, 3, 3, 3, 3, 4])]$

Oder auch:

$\mathit{sequence} = \mathit{deutsch}$

$> \mathit{opt\_prof}\ 8\ 1\ 26$
$[(158780, [2, 2, 3, 4, 2, 4, 2, 7])]$

$> \mathit{opt\_prof}\ 8\ 1\ 6$
$[(162970, [2, 2, 3, 4, 2, 4, 3, 6])]$

Einige bemerkenswerte Aspekte

- Mathematische Reinheit erlaubt Spezifikation auf hohem konzeptionellen Niveau.

Einige bemerkenswerte Aspekte

- Mathematische Reinheit erlaubt Spezifikation auf hohem konzeptionellen Niveau.
- Sehr gute Abstraktionsmechanismen,
  sehr hohes Wiederverwendungspotential.

Einige bemerkenswerte Aspekte

- Mathematische Reinheit erlaubt Spezifikation auf hohem konzeptionellen Niveau.
- Sehr gute Abstraktionsmechanismen,
  sehr hohes Wiederverwendungspotential.
- Im konkreten Fall dynamischer Programmierung,
  klare Separierung der relevanten Aspekte:

Einige bemerkenswerte Aspekte

- Mathematische Reinheit erlaubt Spezifikation auf hohem konzeptionellen Niveau.
- Sehr gute Abstraktionsmechanismen,
  sehr hohes Wiederverwendungspotential.
- Im konkreten Fall dynamischer Programmierung,
  klare Separierung der relevanten Aspekte:
  - Beschreibung des Suchraums
    (durch Parserkombinatoren)

Einige bemerkenswerte Aspekte

- Mathematische Reinheit erlaubt Spezifikation auf hohem konzeptionellen Niveau.
- Sehr gute Abstraktionsmechanismen,
  sehr hohes Wiederverwendungspotential.
- Im konkreten Fall dynamischer Programmierung,
  klare Separierung der relevanten Aspekte:
  - Beschreibung des Suchraums
    (durch Parserkombinatoren)
  - Optimalitätsprinzip von Bellman
    (selektiv anwendbar)

Einige bemerkenswerte Aspekte

- Mathematische Reinheit erlaubt Spezifikation auf hohem konzeptionellen Niveau.
- Sehr gute Abstraktionsmechanismen,
  sehr hohes Wiederverwendungspotential.
- Im konkreten Fall dynamischer Programmierung,
  klare Separierung der relevanten Aspekte:
  - Beschreibung des Suchraums
    (durch Parserkombinatoren)
  - Optimalitätsprinzip von Bellman
    (selektiv anwendbar)
  - Tabellierung für Effizienz
    (ohne Änderung der Codestruktur)

Einige bemerkenswerte Aspekte

- Mathematische Reinheit erlaubt Spezifikation auf hohem konzeptionellen Niveau.
- Sehr gute Abstraktionsmechanismen,
  sehr hohes Wiederverwendungspotential.
- Im konkreten Fall dynamischer Programmierung,
  klare Separierung der 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)

Einige bemerkenswerte Aspekte

- Mathematische Reinheit erlaubt Spezifikation auf hohem konzeptionellen Niveau.
- Sehr gute Abstraktionsmechanismen,
  sehr hohes Wiederverwendungspotential.
- Im konkreten Fall dynamischer Programmierung,
  klare Separierung der 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)
- Allgemein als Sprache sehr gut geeignet zur Exploration von Algorithmen, ohne auch praktische Relevanz aufzugeben.

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.