Algorithmen und Datenstrukturen SoSe25

-Assignment 10-

Moritz Ruge
Matrikelnummer: 5600961
Lennard Wittenberg
Matrikelnummer: —

Juli 2025

1 Problem: Dynamisches Programmieren

Sei s eine Zeichenkette der Länge n. Sie vermuten, dass es sich bei s um einen deutschsprachigen Text handelt, bei dem die Leer- und Satzzeichen verloren gegan- gen sind (also zum Beispiel s = “werreitetsospaetdurchnachtundwind”), und Sie möchten den ursprünglichen Text rekonstruieren.

Dazu steht Ihnen ein Wörterbuch zur Verfügung, das in Form einer Funktion

$$dict : String \to Boolean$$

implementiert ist. dict(w) liefert true für ein gültiges Wort w, und false sonst (z.B. dict(“blau”) = true und dict(“bsau”) = false). Verwenden Sie dynamisches Programmieren, um einen schnellen Algorithmus zu entwickeln, der entscheidet, ob sich s als eine Aneinanderreihung von gültigen Wörtern darstellen lässt. Gehen Sie dabei folgendermaßen vor:

1. Definieren Sie geeignete Teilprobleme und geben Sie eine geeignete Rekursion an. Erklären Sie Ihre Rekursion in einem Satz.
2. Geben Sie Pseudocode für Ihren Algorithmus an.
3. Analysieren Sie die Laufzeit und Speicherplatzbedarf Ihres Algorithmus unter der Annahme, dass ein Aufruf von dict konstante Zeit benötigt.
4. Beschreiben Sie in einem Satz, wie man eine gültige Wortfolge finden kann, falls sie existiert.

1.1 Teilproblem Definieren und Rekursion

Teilproblem: Betrachte dp[i] als boolsche Tabelle, wobei dp[i] = true, wenn das Präfix der Länge i in gültige Wörter aufgeteilt werden kann.

Rekursion: Ein Präfix $s[0 \dots i]$ bis Index i ist gültig (dp[i] = true), wenn es einen Index $j < i$ gibt, so dass:

1. Das Präfix bis j gültig ist (dp[j] = true)
2. Die Zeichenkette von j zu i ein gültiges Wort bildet (durch Aufruf der Funktion dict)

1.2 Pseudocode:

```
def funktion(s: String, dict: String => Boolean): Boolean =
  val n = s.length
  Rval dp = Array.fill(n+1)(flase) // erzeugt Array mit False als default Value
  dp(0) = ture // leere Zeichenkette ist guetig

  for i <- 1 to n do
    for j <- 0 until i do
      if dp(j) && dict(s.substring(j, i)) then
        dp(i) = true

  dp(n)
```

1.3 Laufzeitanalyse

- **Laufzeit:**
  Die Äußere Schleife läuft von 1 bis n, also n mal, die innere Schleife bis i Mal → also insgesamt:

$$\sum_{i=1}^{n} i = O(n^2)$$

- **Speicherplatz:**
  Wie haben ein Array dp der Länge n+1 → also braucht der Speicher $O(n)$

1.4 Wortfolge

Man merkt sich bei jeder erfolgreichen Trennung die Position j und rekonstruiert die Wortgrenzen rückwärts von $dp[n]$ aus, um die Wortfolge zu erhalten.

2 Problem: Editierabstand

Der Editierabstand zwischen zwei Zeichenketten s und t ist die minimale Anzahl von Editieroperationen, um s nach t zu überführen. Es gibt drei Editieroperationen: (i) Einfügen eines Zeichens; (ii) Löschen eines Zeichens; und (iii) Ersetzen eines Zeichens durch ein anderes. Zum Beispiel beträgt der Editierabstand zwischen “APFEL” und “PFERD” drei: Lösche A, ersetze L durch R, füge D ein.

Beschreiben Sie einen Algorithmus, der den Editierabstand zwischen zwei Zeichenketten s und t in $O(k_l)$ Zeit berechnet, wobei s Länge k und t Länge l hat. Erklären Sie außerdem, wie man eine optimale Folge von Editieroperationen findet.

Hinweis: Benutzen Sie dynamisches Programmieren analog zum LCS-Problem. Betrachten Sie das jeweils letzte Zeichen in s und t und unterscheiden Sie drei Möglichkeiten: (a) überführe s nach t' und füge dann ein Zeichen an; (b) überführe s' nach t und lösche dann ein Zeichen; (c) überführe s' nach t' und ersetze dann ein Zeichen, falls nötig. Hierbei bezeichnen s' und t' jeweils s und t ohne den letzten Buchstaben.

2.1 Implementierung

```python
def edit_distance(str1, str2):
  matrix = [[0 for i in range(len(str2)+1)] for j in range(len(str1)+1)]
  for i in range(len(matrix)):
    for j in range(len(matrix[i])):
      # i and j mark the current substring of the respective string. if i or j == 0 than the total amount of edits needed is
      # exactly j / i because "" needs i / j edits to reach str1[:i] / str2[:j]
      if i == 0:
        matrix[i][j] = j
      elif j == 0:
        matrix[i][j] = i
      elif str1[i-1] == str2[j-1]: # If the last characters in each string match then the required amount of edits to reach that substring is the amount of edits from the previous substring
          matrix[i][j] = matrix[i-1][j-1]
      else:  # If the last chars in both string dont match.
        matrix[i][j] = 1+ min(matrix[i][j-1], matrix[i-1][j], matrix[i-1][j-1])
        # matrix[i][j-1]: insert / append to str1, matrix[i-1][j] remove last char from str1: appending , matrix[i-1][j-1]: replace char in
  return matrix[len(str1)][len(str2)] # the bottom right index stores the minimal edits needed to convert str1 to str2.

def main():
print(edit_distance("apfel","pferd")) # converting apfel to pferd. The order of strings matters for the order of operations

main()
# this algorithmn creats a matrix (dictionary) that stores the amount of edits necesarry to convert every substring of str1 to every substring in str2,
# where index matrix[len(str1)][len(str2)] gives the amount of edits needed to convert str1 to str2. operations are always performed on the ast char of str1
# not performing a operation is equivalent to dropping last char from both strings.
# example run -> The goal is to convert str1 to str2
#[ ][i][a][p][f][e][l]
#[j][0][1][2][3][4][5]
#[p][1][1][1][2][3][4]
#[f][2][2][2][1][2][3]
#[e][3][3][3][2][1][2]
#[r][4][4][4][3][2][2]
#[d][5][5][5][4][3][3]
```

```
33 # matrix[0][j] and matrix[i][0] store how many chars need to be added to "" to reach
     str2 and str1
34 # e.g: "" to "ap" needs to edits the same counts for "" to "pf"
35 # e.g: "ap" to "pf" needs 1 edit.
36 # e.g matrix[3][1] ==> str1[2] = f != str2[0] = p -> matrix[3][1] = 1+min(matrix_left,
     matrix_top,matrix_top_left) = 1+1 = 2 matrix[3][1]
37 # check: "p" to "apf" -> insert(a), insert(f)
38
39 # this alogrithmn has O(kl) because in the worst-case of two completly different
     strings l operations have to be performed on k chars., where k = len(str1), l =
     len(str2)
40 # The reason this aproach is faster, is because it is using previous calculated edits
     -> recursion for every path no longer needed.
41 # for every index matrix[i][j] it is checking previous calculations -> lookup O(1)
42
43 # idea to track operations taken (this is an conceptual idea not an error proof
     implementation):
44 # add paramter list[] to track the path of operations  -> append,delete,swap ->
     edit_distance(str1,str2,opt_taken)
45 # the list starts of empty
46 # After every opertaion return the name of the operation taken is appended to the list
     -> done by checking which index in matrix in min() was chosen.
47 # cases:
48 # 1. nothing: str1[i-1] == str2[j-1] -> nothing
49 # 2. appending a char -> opt_take.append("append")
50 # 3. deleting a char -> opt_take.append("delete")
51 # 4. swapping a char -> opt_take.append("swap")
```

3 Problem: Finden von Senken in Graphen

Betrachtet man die Adjazenzmatrixdarstellung eines Graphen $G = (V, E)$, dann haben viele Algorithmen Laufzeit $|V|2$. Es gibt aber Ausnahmen. Zeigen Sie, dass die Frage, ob ein gerichteter Graph G eine globale Senke — einen Knoten vom Eingrad $|V| - 1$ und Ausgrad 0 − hat, in Zeit $O(|V|)$ beantwortet werden kann, selbst wenn man die Adjazenzmatrixdarstellung von G (die ja selbst schon die Größe $O(|V|2$) hat) verwendet. Beweisen Sie Korrektheit und Laufzeit Ihres Algorithmus.

Hinweis: Sei A die Adjazenzmatrix von G und $u, v \in V, u \neq v$. Was folgt über u und v, wenn $A_{uv} = 1$ ist? Was, wenn $A_{uv} = 0$ ist?

3.1 Beispiel: Dreieck

	A	B	C	D
A	0	1	0	1
B	0	0	1	1
C	1	0	0	1
D	0	0	0	0

Ein Knoten kann in diesem Fall keine Kante zu sich selbst haben (keine Kreise)
Darüber hinaus, kann es maximal nur eine globale Senke geben. Dies ergibt sich aus der Definition von globalen Senken.

directed Graph $matrix[v][*]$ row outgoing edges, $matrix[*][v]$ coloumn incoming edges. → we care about $matrix[*][v]$

```
i = 0 # row
j = 0 # cols
while i < len(matrix) and j < len(matrix):  # solange i und j kleiner gleich der Anzahl von Knoten in der Matrix sind.
  if matrix[i][j] == 0:     # wenn der Knoten i keine ausgehende Kante zum Knoten j hat gehe nach rechts in der Matrix.
    i += 1
  else:          # Der Knoten i hat eine ausgehende Kante, kann also keine globale Senke sein.
    j += 1
gol_snk = i          # moegliche globale Senke Befindet sich am Index i bzw. j, da diese in einer quadratischen Matrix, auf den identischen Knoten zeigen. Im ersten Squentiellen Loop werden alle ausser einem Knoten als moegliche globale Senke ausgeschlossen. Ist ein matrix[i][j] == 1, ist der Knoten matrix[i] als Kandidat ausgeschlossen, da dieser eine ausgehende Kante hat. Die Suche wird fuer den naechsten Knoten fortgesetzt. Der Knoten i, bei dem die Schleifen bedigung i < len(matrix) and j < len(matrix) verletzt wird ist der Kandidate fuer die globale Senke. Dies geschiet nur wenn matrix[i] nur aus 0 besteht oder wenn i = len(matrix)-1 und beim unten weiter suchen matrix[i][j] die erste 0 gefunden wird.

# Ueberpruefe erneut, ob matrix[gol_snk] keine ausgehenden Kanten hat
for k in range(len(matrix)):
  if matrix[candidate][k] == 1: # es gibt doch eine ausgehende Kante -> es gibt keine globale Senke.
    return -1  # std error code

# Ueberpruefe ob der Kandidat (matrix[*][gol_snk]) wirklich eine eingehende Kanten von jedem anderen Knoten hat, ausser von sich selbst.
for k in range(len(matrix)):
  if k != gol_snk and matrix[k][gol_snk] == 0: # alle Eintraege der Kandidatenspalte (ausser von sich selbst) muessen 1 sein.
    return -1  # Not a sink

```

```
20 return candidate  # wenn alle Ueberpruefungen durchlaufen wurden, und nicht vorher
       abgebrochen wurde, wurde die globale Senke gefunden.
```

3.2 Laufzeit

Laufzeitkomplexität im worst-case: $O(2|V| - 1 + |V| + |V|) \rightarrow$ dominate Klasse ist $O(|V|)$
Die Überprüfungen dauern: $|V|$
Die erste Suche kann $2|V| - 1$ dauern, da man $|V| - 1$ mal nach rechts und unten gehen kann. Es ist also mögliche die globale Senke in linearer Zeit zufinden. $O(|V|)$ worst-case: $O(n)$
Die globale Senke garantiert und korrekt in nur $|V|$ Iterationen / Schritten zu finden ist nicht möglich.