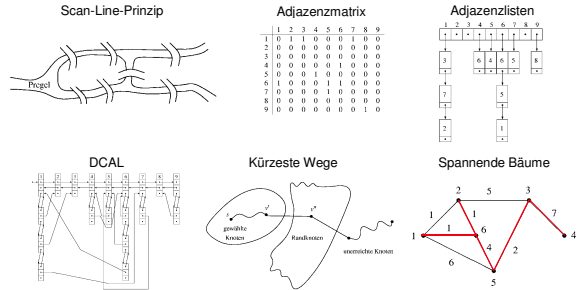


Textsuche

Thomas Röfer

- Naive Suche
- Verfahren von Knuth-Morris-Pratt
- Verfahren von Boyer-Moore
- Ähnlichkeitssuche
- Editierdistanz

Rückblick „Graphenalgorithmen“



Pl-2: Textsuche

2

Einleitung

- **Motivation**
 - Für viele Anwendungen ist das Durchsuchen von Texten notwendig
 - Texte können sehr groß sein, daher ist Geschwindigkeit ein zentraler Faktor
- **Ansätze**
 - Direktes Durchsuchen des Texts
 - Z.B. in Textverarbeitung
 - Aufbereitung des Texts z.B. durch Index
 - Z.B. Google
- **Definition**
 - Gegeben ist ein Zeichensatz Σ (Alphabet)
 - Gegeben ein Text $T \in \Sigma^n$ und ein Wort $w \in \Sigma^m$ mit $n \gg m$
- **Fragestellungen**
 - Kommt das Wort w im Text T vor?
 - Wo kommt w (das erste Mal) in T vor?
 - Wie oft kommt w in T vor?

Pl-2: Textsuche

3

Naive Suche

```
boolean naiveSearch(char[] text, char[] word)
{
    for(int i = 0; i <= text.length - word.length; ++i)
    {
        boolean matches = true;
        for(int j = 0; j < word.length; ++j)
            if(text[i + j] != word[j])
                matches = false;
        if(matches)
            return true;
    }
    return false;
}
```

- Aufwand: $O(n \cdot m)$

Pl-2: Textsuche

4

Naive Suche

```
boolean naiveSearch2(char[] text, char[] word)
{
    for(int i = 0; i <= text.length - word.length; ++i)
    {
        int j;
        for(j = 0; j < word.length && text[i + j] == word[j]; ++j)
            ;
        if(j == word.length)
            return true;
    }
    return false;
}
```

- worst-case Aufwand: $O(n \cdot m)$

Pl-2: Textsuche

5

Naive Suche

er sprach abrakadabra, es bewegte sich aber nichts
 aber
 aber
 aber ...
 aber
 aber ...
 aber
 aber

Pl-2: Textsuche

6

Suche nach Knuth-Morris-Pratt

Idee

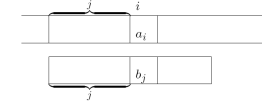
- Information aus vorherigen Vergleichen wiederverwerten
- Keine bekannten Zeichen des Textes erneut vergleichen
- Wort ggf. um mehrere Stellen verschieben

Beispiel

a b c a b c a b d
 a b c a b d

Vorverarbeitung des Wortes (Dynamische Programmierung)

- Betrachte $w_0 = b_0, \dots, b_{l-1}$
- Gesucht: größter Suffix von w_0 , der selbst Präfix von w ist.
- Verschiebetabelle $next(j)$



j	0	1	2	3	4	5	6
$char[j]$	A	N	A	N	A	S	
$next[j]$	-1	0	0	1	2	3	0

Suche nach Knuth-Morris-Pratt

Aufwandsabschätzung

- i wird nie kleiner
- $next(j)$ ist immer kleiner als j
- i und j werden immer gemeinsam inkrementiert
- j kann nur so oft kleiner werden, wie es vorher inkrementiert wurde

Daraus folgt

- $j = next(j)$ wird höchstens n -mal angewandt
- Aufwand in $O(n)$

Einschränkung

- $next(j)$ wird als bekannt vorausgesetzt

```
boolean kmpSearch(char[] text, char[] word)
{
  int i = 0,
      j = 0;
  int [] next = initNext(word);
  while(i < text.length && j < word.length)
    if(j == -1 || text[i] == word[j])
    {
      ++i;
      ++j;
    }
    else
    {
      j = next[j];
    }
  return j == word.length;
}
```

Suche nach Knuth-Morris-Pratt

Ansatz

- Muster über sich selbst schieben

Beispiel

j	0	1	2	3	4	5	6
$word[j]$	A	N	A	N	A	S	
$i=0, j=-1$	-1						
$i=1, j=0$	-1	0					
$i=1, j=-1$	-1	0					
$i=2, j=0$	-1	0	0				
$i=3, j=1$	-1	0	0	1			
$i=4, j=2$	-1	0	0	1	2		
$i=5, j=3$	-1	0	0	1	2	3	
$i=5, j=1$	-1	0	0	1	2	3	
$i=5, j=0$	-1	0	0	1	2	3	
$i=5, j=-1$	-1	0	0	1	2	3	
$i=6, j=0$	-1	0	0	1	2	3	0

Aufwand: $O(m)$

```
int[] initNext(char[] word)
{
  int[] next = new int[word.length + 1];
  int i = 0,
      j = -1;
  next[i] = j;
  while(i < word.length)
    if(j == -1 || word[i] == word[j])
    {
      ++i;
      ++j;
      next[i] = j;
    }
    else
    {
      j = next[j];
    }
  return next;
}
```

Suche nach Boyer-Moore

Motivation

- Suche nach Knuth-Morris-Pratt nutzt bisher erfolgreiche Vergleiche
- Das kommt aber eher selten vor
- Man könnte größere Sprünge machen, wenn man das Wort von rechts durchsucht

Voraussetzung

- Wort $w = w_0 \dots w_{m-1}$ wird von links nach rechts gegen Text T verschoben
- Aber: Vergleich läuft von w_{m-1} nach w_0

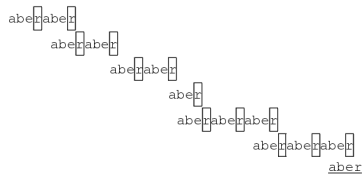
```
boolean naiveSearch3(char[] text, char[] word)
{
  int i = word.length - 1,
      j = word.length - 1;
  while(i < text.length && j >= 0)
    if(text[i] == word[j])
    {
      --i;
      --j;
    }
    else
    {
      i += word.length - j;
      j = word.length - 1;
    }
  return j < 0;
}
```

Suche nach Boyer-Moore

„Schlechtes Zeichen“-Strategie

- Bei Mismatch wird das Suchwort bis zum rechtesten Vorkommen des aktuellen Textzeichens verschoben
- Kommt das aktuelle Textzeichen gar nicht im Suchwort vor, kann das Wort um seine volle Länge verschoben werden

er sagte abrakadabra, es bewegte sich aber nichts



Suche nach Boyer-Moore – einfach

Methode

- Bestimme das rechteste Vorkommen jedes Zeichens im Suchwort
- Erzeuge die Tabelle $delta_i(c)$, in der für jedes Zeichen steht, um wie viel man das Muster weiterschieben darf
 - Abstand vom Ende
- Kommt ein Zeichen nicht vor, ist dies die Länge des Wortes
- Boyer-Moore (einfach)**
 - Wählt immer die größtmögliche Verschiebung aus „naiv“ und $delta_i(c)$

```
boolean bmSearchSimple(char[] text, char[] word)
{
  int[] delta1 = initDelta1(word);
  int i = word.length - 1,
      j = word.length - 1;
  while(i < text.length && j >= 0)
    if(text[i] == word[j])
    {
      --i;
      --j;
    }
    else
    {
      i += Math.max(word.length - j, delta1[text[i]]);
      j = word.length - 1;
    }
  return j < 0;
}
```

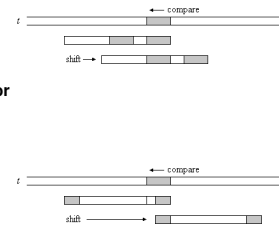
Suche nach Boyer-Moore – vollständig

- ▶ **Aufwand**
 - ▶ Im schlechtesten Fall: $O(n \cdot m)$.
 - ▶ Für großes Alphabet/kurze Muster $O(n/m)$.
- ▶ **Verbesserung: „Gutes Ende“-Strategie**
 - ▶ Tabelle $delta_2(j)$ zur Rechtsverschiebung
 - ▶ Konstruktion ähnlich der $next(j)$ -Tabelle beim KMP-Verfahren
 - ▶ Verhindert Verschieben nach links \rightarrow worst-case: $O(n+m)$

```
boolean bmSearch(char[] text,
                char[] word)
{
    int[] delta1 = initDelta1(word),
        delta2 = initDelta2(word);
    int i = word.length - 1,
        j = word.length - 1;
    while (i < text.length && j >= 0)
        if (text[i] == word[j])
        {
            --i;
            --j;
        }
        else
        {
            i += Math.max(word.length - j - 1,
                          delta2[j + 1],
                          delta1[text[i]]);
            j = word.length - 1;
        }
    return j < 0;
}
```

Suche nach Boyer-Moore – $delta_2(j)$

- ▶ **Suffix kommt nochmal vor**
 - ▶ a b a a b a b a c b a
 - c a b a b
 - c a b a b
- ▶ **Suffix kommt nicht nochmal vor**
 - ▶ a b c a b a b a c b a
 - c b a a b
 - c b a a b
- ▶ **Teil des Suffix ist Präfix**
 - ▶ a a b a b a b a c b a
 - a b a a b
 - a b b a b



Suche nach Boyer-Moore – $delta_2(j)$

- ▶ **Bestimmung von $next(j)$, aber rückwärts $\rightarrow next'(j)$**
- ▶ **Entspricht**
 - ▶ Wort umdrehen
 - ▶ $next()$ ausrechnen
 - ▶ Für $j = 0 \dots m$: $next'(j) = m - next(m-j)$
- ▶ **Hinweis**
 - ▶ j müsste eigentlich Indizes im Bereich $[-1 \dots m-1]$ haben
 - ▶ Unpraktisch für Java, daher um 1 verschoben
- ▶ **Beispiel**
 - ▶ j : 0 1 2 3 4 5 6 7
 - w : a b b a b a b
 - $next'(j)$: 5 6 4 5 6 7 7 8

Suche nach Boyer-Moore – $delta_2(j)$

- ▶ **Fall 1: Suffix kommt nochmal vor**
 - ▶ j : 0 1 2 3 4 5 6 7
 - w : a b b a b a b
 - $next'(j)$: 5 6 4 5 6 7 7 8
 - $delta_2(j)$: 0 0 0 0 2 0 4 1
- ▶ **Fall 2: Teil des Suffix ist Präfix**
 - ▶ j : 0 1 2 3 4 5 6 7
 - w : a b b a b a b
 - $next'(j)$: 5 6 4 5 6 7 7 8
 - $delta_2(j)$: 5 5 5 2 5 4 1

```
for (int i = 1; i < word.length; ++i)
{
    int j = next2[i];
    while (j < next2.length)
    {
        if (word[i - 1] != word[j - 1])
            delta2[j] = j - i;
        j = next2[j];
    }
}

int j = next2[0];
for (int i = 0; i <= word.length; ++i)
{
    if (delta2[i] == 0)
        delta2[i] = j;
    if (i == j)
        j = next2[j];
}
```

Ähnlichkeitssuche

- ▶ **Motivation**
 - ▶ Manchmal möchte man ein Wort nicht exakt im Text wieder finden, sondern auch „ähnliche“ Zeichenketten akzeptieren, z.B. wenn man die Schreibung eines Begriffes nicht genau kennt
- ▶ **Ähnlichkeitsmaße**
 - ▶ k -Mismatch: Weicht eine Zeichenkette im Text an höchstens k Stellen vom gesuchten Wort ab?
 - ▶ Editierdistanz: Wie viele Ersetzen-, Einfüge- und Löschooperationen sind notwendig, um aus dem gesuchten Wort eine im Text vorkommende Zeichenkette zu machen?
 - ▶ Phonetische Ähnlichkeit: „Klingt“ das gesuchte Wort so wie eine Stelle im Text?
 - ▶ Sprachabhängig!
 - ▶ Synonymsuche: Kommt vielleicht ein Synonym des Suchwortes im Text vor?
 - ▶ Erfordert Synonym-Lexikon (genutzt z.B. von Google)
 - ▶ ...

k -Mismatch-Suche

- ▶ **Abwandlung der naiven Suche**

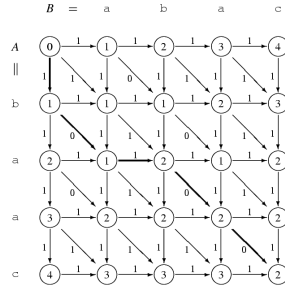
```
boolean kMismatchSearch(char[] text, char[] word, int k)
{
    for (int i = 0; i <= text.length - word.length; ++i)
    {
        int mismatches = 0;
        for (int j = 0; j < word.length && mismatches <= k; ++j)
            if (text[i + j] != word[j])
                ++mismatches;

        if (mismatches <= k)
            return true;
    }
    return false;
}
```

- ▶ **Nachteil: Fehlende oder überzählige Zeichen werden nicht erkannt**

Editierdistanz

- ▶ **Operationen**
 - ▶ Einfügen eines Zeichens
 - ▶ Waagrecht, Kosten 1
 - ▶ Löschen eines Zeichens
 - ▶ Senkrecht, Kosten 1
 - ▶ Ersetzen eines Zeichens
 - ▶ Diagonal, 1 wenn Änderung, sonst 0
- ▶ **Beispiele**
 - ▶ b a a c
 - ▶ a a c0 ersetzt
 - ▶ a b a c1 ersetzt
 - ▶ b a a c
 - ▶ a a c0 gelöscht
 - ▶ a b a c1 eingefügt
- ▶ **Spurgraph**
 - ▶ Startmuster steht links
 - ▶ Zielmuster steht oben
 - ▶ Finde kürzeste Spur



Fachgespräche

- ▶ **Ort**
 - ▶ MZH 3060, 31. Juli bis 11. August (Terminabsprache mit Tutoren)
- ▶ **Umfang**
 - ▶ ca. 20-25 Minuten pro 3er-Gruppe
- ▶ **Inhalt**
 - ▶ Suchen (KW Kap. 11 + OW Kap. 3)
 - ▶ Sortieren (KW Kap. 12 + OW Kap. 2)
 - ▶ Bäume (KW Kap. 13 + OW Kap. 5)
 - ▶ Hashing (WK Kap. 14 + OW Kap. 4)
 - ▶ Manipulation von Mengen (OW Kap. 6)
 - ▶ Geometrische Algorithmen (OW Kap. 7)
 - ▶ Graphenalgorithmen (OW Kap. 8)
 - ▶ Textsuche (OW Kap. 9)