Flexible Pattern Matching

January 19, 2016

1 Simple Pattern Matching

- eine Schleife, die über den Text itertiert (kann beendet werden, sobald der restliche Text kürzer wäre als das Pattern selbst)
- eine zweite Schleife, die an jeder Position des Textes über die nächsten Buchstaben und das Pattern iteriert und abbricht, sobald ein Buchstabe im Pattern nicht mit dem aktuellen Buchstaben im Text übereinstimmt
- wenn die zweite Schleife komplett durchlaufen wurde, wurde ein Match gefunden

In [ ]:
```
def simple_search(text, pattern):
    for i in range(len(text) - len(pattern) + 1):
        for j, char in enumerate(pattern):
            if char != text[i+j]:
                break
        else:
            yield i
```

2 Knuth-Morris-Pratt

- das selbe Prinzip wie beim Simple Pattern Matching
- das Pattern wird jedoch bei einem Mismatch von Buchstaben "weiter nach vorne geschoben"
- hierbei hilft eine Prefix-Tabelle (auch Next-Funktion) die die "Verschiebepositionen" speichert. (Die Verschiebeposition ist die Länge des längsten Suffix des Teils des Patterns der gefunden wurde, der gleichzeitig Präfix des gesamten Patterns ist)

In [ ]:
```
def get_prefix_table(pattern):
    i, j = 0, -1
    prefix_table = [-1] * (len(pattern) + 1)
    while i < len(pattern):
        while j >= 0 and pattern[j] != pattern[i]:
            j = prefix_table[j]
        i += 1
        j += 1
        if i == len(pattern):
            prefix_table[i] = j
        elif pattern[i] != pattern[j]:
            prefix_table[i] = j
        else:
            prefix_table[i] = prefix_table[j]
    return prefix_table
```

Table 1: Präfix-Tabelle für das Pattern abracadabra

0
a
-1

```
In [ ]:     def kmp_search(text, pattern, prefix_table):
                i, j = 0, 0
                while i < len(text):
                    while j >= 0 and text[i] != pattern[j]:
                        j = prefix_table[j]
                    i += 1
                    j += 1
                    if j == len(pattern):
                        yield i - len(pattern)
                        j = prefix_table[j]
```

3 Shift-And

- Automat wird mit Bitmasken repräsentiert
- Bitmasken für jeden Buchstaben im Pattern erstellen (alle anderen Buchstaben haben 0-Vektor als Bitmaske)
- Vektor der einen Automaten repräsentiert, dessen Anfangszustand immer aktiv ist, wird durch Shift-Operationen "durchlaufen" und mit der Bitmaske des aktuell gelesenen Buchstabens im Text "verundet"
- wenn der letzte Zustand des Automaten aktiv ist, wurde das Pattern gefunden
- Endianness der Vektoren ist zu beachten! (immer Big Endian?)

```
In [ ]:     from BitVector import BitVector
            def get_bit_table(pattern, alphabet):
                table = {}
                for char in alphabet:
                    table[char] = BitVector(size=len(pattern))
                for i, char in enumerate(pattern):
                    table[char] = table[char] | BitVector(size=len(pattern)-i-1) \
                                              + BitVector(intVal=1) \
                                              + BitVector(size=i)
                return table
```

Table 2: Bitmasken-Tabelle für das Pattern abracadabra

ltr											
a	1	0	0	1	0	1	0	1	0	0	1
b	0	0	1	0	0	0	0	0	0	1	0
r	0	1	0	0	0	0	0	0	1	0	0
c	0	0	0	0	0	0	1	0	0	0	0
d	0	0	0	0	1	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0

```
In [ ]:     def shift_and_search(text, pattern, bit_table):
                found = BitVector(intVal=1) + BitVector(size=len(pattern)-1)
```

```
        zero = BitVector(size=len(pattern))
        A = BitVector(size=len(pattern))
        for i, char in enumerate(text):
            A = (A<<1 | (BitVector(size=len(pattern)-1) \
                         + BitVector(intVal=1))) \
                & bit_table[char]
            if A & found != zero:
                yield i - len(pattern) + 1
```

4 Shift-Or

- gleiches Konzept wie beim Shift-And Verfahren
- hier repräsentieren 0en aktive und 1en inaktive Zustände, so kann der Schritt des "aktiv machens" des ersten Zustands des Automaten gespart werden, da beim shift automatisch eine neue 0 (aktiver Zustand) hinzugefügt wird
- alle Bitvektoren sind hier natürlich invertiert

5 Boyer-Moore

- Pattern wird wie zuvor von links nach rechts durch den Text geschoben, jedoch wird nun das Pattern von rechts nach links durchlaufen (in natürlichen Sprachen wird so üblicherweise früher ein Mismatch gefunden und das Pattern kann schneller verschoben werden)
- für die Verschiebung werden zwei Heuristiken angewandt
  - **Bad-Character Heuristik**
    bei einem Mismatch kann das Pattern soweit verschoben werden, dass der aktuell im Text gelesene Buchstabe mit dem letzten vorkommen dieses Buchstabens im Pattern aligniert ist, wenn dieser Buchstabe gar nicht im Pattern vorkommt, kann das Pattern um seine ganze Länge verschoben werden
  - **Good-Suffix Heuristik**
    Wenn das bis zum Mismatch gelesene Suffix des Patterns nochmals Infix des Patterns ist, kann das Pattern soweit verschoben werden, bis der gelesene Teil mit diesem Infix aligniert ist, kommt dieses Suffix kein zweites malim Pattern vor, kann das Pattern um seine ganze Länge verschoben werden
- es wird immer die maximale Verschiebung die sich durch diese Heristiken ergeben angewandt

6 Horspool

- Wie bei Boyer-Moore wird der Text von links nach rechts, das Pattern aber von rechts nach links durchlaufen
- sobald ein Mismatch erreicht wird, wird das Pattern soweit verschoben, dass das gerade gelesene Zeichen im Text mit dem letzen

In [ ]:
```
    def get_horspool_table(pattern,alphabet):
        table = {}
        for char in alphabet:
            try:
                table[char] = pattern.rindex(char)
            except ValueError:
                table[char] = len(pattern)
        return table

    def horspool_search(text, pattern, horspool_table):
```

```
pos = 0
while pos <= len(text) - len(pattern):
    j = len(pattern)-1
    while j > 0 and text[pos + j] == pattern[j]:
        j -= 1
    if j == 0:
        yield pos
    pos = pos + horspool_table[text[pos+len(pattern)]]
```

7 Faktorbasierte Suche