Flexible Pattern Matching

January 20, 2016

Testsätze und -pattern sowie Hilfsfunktion um Treffer in diesen zu rendern

```
In [1]: from IPython.display import display_html

        tests = [("Ich bin ein Moofoo der in Barfoo lebt.", "foo"),
                 ("And the magician said: 'abracadabra, simsalabim!'", "abracadabra"),
                 ("CPM_annual_conference_announce", "announce")]

        def print_highlighted_tests(search_fn):
            out = []
            for text, pattern in tests:
                start_idxs = search_fn(text, pattern)
                text = list(text)
                for idx in start_idxs:
                    text[idx] = '<strong>' + text[idx]
                    end_idx = idx+len(pattern)-1
                    text[end_idx] += '</strong>'
                display_html("".join(text), raw=True)
```

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 [2]: def simple_search(text, pattern):
            for text_idx in range(len(text) - len(pattern) + 1):
                for pat_idx, char in enumerate(pattern):
                    if char != text[text_idx+pat_idx]:
                        break
                if pat_idx == (len(pattern) - 1):
                    yield text_idx

        print_highlighted_tests(simple_search)
```

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 [3]: 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

        print(get_prefix_table('abracadabra'))
```

```
[-1, 0, 0, -1, 1, -1, 1, -1, 0, 0, -1, 4]
```

Table 1: Präfix-Tabelle für das Pattern abracadabra

0	1	2	3	4	5	6	7	8	9	10	match
a	b	r	a	c	a	d	a	b	r	a	
-1	0	0	-1	1	-1	1	-1	0	0	-1	4

```
In [4]: def kmp_search(text, pattern):
            prefix_table = get_prefix_table(pattern)
            text_idx, pat_idx = 0, 0
            while text_idx < len(text):
                while pat_idx >= 0 and text[text_idx] != pattern[pat_idx]:
                    pat_idx = prefix_table[pat_idx]
                text_idx += 1
                pat_idx += 1
                if pat_idx == len(pattern):
                    yield text_idx - len(pattern)
                    pat_idx = prefix_table[pat_idx]

        print_highlighted_tests(kmp_search)
```

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 [5]: from bitstring import BitArray
        from collections import defaultdict

        def get_bit_table(pattern):
            alphabet = set(pattern)
            # Every entry represents the *reversed* pattern, with a 1
            # at the positions with the character
            table = {char: BitArray('0b' + ''.join('1' if c == char else '0'
                                                   for c in reversed(pattern)))
                     for char in alphabet}
            return table

        get_bit_table('abracadabra')
```

```
Out[5]: {'a': BitArray('0b10010101001'),
         'b': BitArray('0b00100000010'),
         'c': BitArray('0b00000010000'),
         'd': BitArray('0b00001000000'),
         'r': BitArray('0b01000000100')}
```

Table 2: Bitmasken-Tabelle für das Pattern abracadabra

letter										
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 [13]: def shift_and_search(text, pattern):
             m = len(pattern)
             bit_table = get_bit_table(pattern)
             empty_vec = BitArray(length=m)
             # Vector that represents a full match on the pattern ('1000...')
             found_vec = BitArray('0b1' + '0'*(m-1))
             # Vector that adds a 'empty_word' transition on the first state
             # to itself ('...0001')
             init_vec = BitArray('0b' + '0'*(m-1) + '1')

             # In the beginning, all states in the NFA are inactive
             automaton = BitArray(length=m)
             for text_idx, char in enumerate(text):
                 automaton = ((automaton << 1 | init_vec)
                              & bit_table.get(char, empty_vec))
                 if automaton & found_vec != empty_vec:
                       yield text_idx - m + 1

         print_highlighted_tests(shift_and_search)
```

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

```
In [7]: def shift_or_search(text, pattern):
            m = len(pattern)
            empty_vec = BitArray('0b' + '1'*m)

            # Vector that represents a full match on the pattern ('0111...')
            found_vec = BitArray('0b0' + '1'*(m-1))

            # XORing with an all-one array of same length creates the complement
            bit_table = {char: vec ^ empty_vec
                         for char, vec in get_bit_table(pattern).items()}

            # In the beginning, all states in the NFA are inactive
            automaton = BitArray('0b' + '1'*m)
            for text_idx, char in enumerate(text):
                automaton = (automaton << 1 | bit_table.get(char, empty_vec))
                if (automaton | found_vec) != empty_vec:
                      yield text_idx - m + 1

        print_highlighted_tests(shift_or_search)
```

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 [8]: def get_horspool_table(pattern):
            alphabet = set(pattern[:-1])
```

```
            rev_pat = pattern[::-1]
            table = {char: rev_pat.index(char) or len(pattern)
                     for char in alphabet}
            return table

        get_horspool_table('announce')
```

Out[8]: {'a': 7, 'c': 1, 'n': 2, 'o': 4, 'u': 3}

```
In [9]: def horspool_search(text, pattern):
            # FIXME: There's still a bug in here, the 'abracadabra'
            #        example doesn't work!
            n = len(text) - 1
            m = len(pattern) - 1
            table = get_horspool_table(pattern)
            pos = -1
            while pos <= n -m:
                j = m
                while j > 0 and text[pos + j] == pattern[j]:
                    j -= 1
                if j == 0:
                    yield pos
                pos += table.get(text[pos+m], m)

        print_highlighted_tests(horspool_search)
```

7 Faktorbasierte Suche

```
In [ ]:
```