Practicum 3 Sorting & Searching

Substring search en regular expressions

Kostis Thanos
500739944
IS-204

Inhoudsopgave

Inleiding

In opdracht van de Hogeschool van Amsterdam moet ik (Kostis Thanos) voor het vak Sorting and Searching een aantal opdrachten maken en die vervolgens weer verwerken in verslagen. Dit verslag betreft opdracht 3, die bestaat uit die 3 subopdrachten.
Als eerst zal ik de source code van java en een andere programmeertaal bekijken om er achter te komen welk algoritme wordt gebruikt bij het zoeken naar een bepaalde String in een text (/code). Vervolgens vergelijk ik de algoritmes Knuth-Morris-Pratt en Boyer-Moore op efficiëntie door ze te gebruiken om woorden te zoeken in een lang stuk tekst (het gedicht Mei van Herman Gorter).
Als laatst zal ik met een pattern klasse uit java een java een methode schrijven die een aantal regels controleert. Veel plezier met lezen!

Programmeertaal 1: Java

Na even zoeken kwam ik er weer achter dat je op een string de methode “contains()” kan aanroepen om erachter te komen of een substring zich bevindt in de originele string. Als je de code in eclipse schrijft en klikt er vervolgens op met ctrl + linkermuisknop, kom je in de implementatie van de contins methode. De code verwijst een paar keer door naar andere methoden (dat wordt hieronder weergegeven).

De contains methode:

```
   public boolean contains(CharSequence s) {
        return indexOf(s.toString()) > -1;
    }
,stuurt ons door naar
```

Verwijst door naar de eerste indexOf methode

```
 * Returns the index within this string of the first occurrence of the
     * specified substring.
public int indexOf(String str) {
        return indexOf(str, 0);
    }
```

verwijst door naar De tweede indexOf methode

```
Returns the index within this string of the first occurrence of the
     * specified substring, starting at the specified index.
    public int indexOf(String str, int fromIndex) {
        return indexOf(value, 0, value.length,
                str.value, 0, str.value.length, fromIndex);
    }
```

En verwijst als laatst door naar de laatste IndexOf methode

```
    static int indexOf(char[] source, int sourceOffset, int sourceCount,
            char[] target, int targetOffset, int targetCount,
            int fromIndex) {
        if (fromIndex >= sourceCount) {
            return (targetCount == 0 ? sourceCount : -1);
        }
        if (fromIndex < 0) {
            fromIndex = 0;
        }
        if (targetCount == 0) {
            return fromIndex;
        }

        char first = target[targetOffset];
        int max = sourceOffset + (sourceCount - targetCount);

        for (int i = sourceOffset + fromIndex; i <= max; i++) {
            /* Look for first character. */
            if (source[i] != first) {
                while (++i <= max && source[i] != first);
            }

            /* Found first character, now look at the rest of v2 */
```

```
            if (i <= max) {
                int j = i + 1;
                int end = j + targetCount - 1;
                for (int k = targetOffset + 1; j < end && source[j]
                        == target[k]; j++, k++);

                if (j == end) {
                    /* Found whole string. */
                    return i - sourceOffset;
                }
            }
        }
        return -1;
    }
```

```
* @param   source       the characters being searched.(= de string)
* @param   sourceOffset offset of the source string. (= 0)
* @param   sourceCount  count of the source string. (= de string.length)
* @param   target       the characters being searched for.(= de substring)
* @param   targetOffset offset of the target string. (= 0)
* @param   targetCount  count of the target string (= de substring.length)
* @param   fromIndex    the index to begin searching from. (Index = 0)
```

first is het eerste karakter van de substring:

```
  char first = target[targetOffset];
```

Int max = String lengte – substring lengte:

```
      int max = sourceOffset + (sourceCount - targetCount);
```

als current string karakter niet de eerste van de substring is

```
            if (source[i] != first) {
```

zolang i++ kleiner of gelijk aan het maximum is, en de current string value niet de eerste waarde van de substring is

```
                while (++i <= max && source[i] != first);
```

i = de current geïtereerde waarde van de string

j = i + 1 (voor de tweede letter)

end = i + substring length (i waarde van het eind van substring)

Deze code wordt pas aangeroepen als de eerste letter is gevonden, waarna hij dus elke letter afgaat. Het laatste if statement controleert of j het einde van de substring bereikt (als i-waarde).

dan returnt hij i (want de sourceOffset is 0)

```
for (int k = 1; j < end && tweede letter == target[k]; j++ k++
     if (j == end){
           return i – sourceOffset                    (i – 0 = i)
```

De java code gaat zoekt dus steeds de eerste letter van de substring in de string en kijkt vervolgende voor de opvolgende letters van de substring tot hij hem vindt.

Dit is dus de Brute-force substring search

Programmeertaal 2: Python

In de programmertaal C wordt gebruikt gemaakt van de functie:

```
str1.find(str2)
```

De implementatie van de code is als volgt:

```
20 Py_LOCAL_INLINE(Py_ssize_t)
21 fastsearch(const STRINGLIB_CHAR* s, Py_ssize_t n,
22             const STRINGLIB_CHAR* p, Py_ssize_t m,
23             int mode)
24 {
25     long mask;
26     Py_ssize_t skip, count = 0;
27     Py_ssize_t i, j, mlast, w;
28 
29     w = n - m;
30 
31     if (w < 0)
32         return -1;
33 
34     /* look for special cases */
35     if (m <= 1) {
36         if (m <= 0)
37             return -1;
38         /* use special case for 1-character strings */
39         if (mode == FAST_COUNT) {
40             for (i = 0; i < n; i++)
41                 if (s[i] == p[0])
42                     count++;
43             return count;
44         } else {
45             for (i = 0; i < n; i++)
46                 if (s[i] == p[0])
47                     return i;
48         }
49         return -1;
50     }
51 
52     mlast = m - 1;
53 
54     /* create compressed boyer-moore delta 1 table */
55     skip = mlast - 1;
56     /* process pattern[:-1] */
57     for (mask = i = 0; i < mlast; i++) {
58         mask |= (1 << (p[i] & 0x1F));
59         if (p[i] == p[mlast])
60             skip = mlast - i - 1;
```

```
    }
    /* process pattern[-1] outside the loop */
    mask |= (1 << (p[mlast] & 0x1F));

    for (i = 0; i <= w; i++) {
        /* note: using mlast in the skip path slows things down on x86 */
        if (s[i+m-1] == p[m-1]) {
            /* candidate match */
            for (j = 0; j < mlast; j++)
                if (s[i+j] != p[j])
                    break;
            if (j == mlast) {
                /* got a match! */
                if (mode != FAST_COUNT)
                    return i;
                count++;
                i = i + mlast;
                continue;
            }
            /* miss: check if next character is part of pattern */
            if (!(mask & (1 << (s[i+m] & 0x1F))))
                i = i + m;
            else
                i = i + skip;
        } else {
            /* skip: check if next character is part of pattern */
            if (!(mask & (1 << (s[i+m] & 0x1F))))
                i = i + m;
        }
    }

    if (mode != FAST_COUNT)
        return -1;
    return count;
}

#endif
```

n = de string

m = de substring

Hier haat het om Boyer-Moore omdat het algoritme hier skipt en ook steeds controleert of de laatste letter van de substring matcht met de waarde van de string die er tegenover staat, zodat hij weet of hij voor de volgende reeks letters die hij onderzoekt daar moet beginnen of juist één karakter verder.

Boyer-Moore geanalyseerd

Door de code van Boyer-Moore uit het boek ook iets aan te passen kom ik op de volgende code die het aantal keer voorkomen van het woord telt. Als skip nul is wordt er één toegevoegd aan de counter.

```
public int search(String txt) { // Search for pattern in txt.
    int N = txt.length();
    int M = pat.length();
    int skip;
    for (int i = 0; i <= N - M; i += skip) { // Does the pattern match the text
at position i ?
        skip = 0;
        for (int j = M - 1; j >= 0; j--) {
            if (pat.charAt(j) != txt.charAt(i + j)) {
                skip = j - right[txt.charAt(i + j)];
                if (skip < 1) {
                    skip = 1;
                }
                    comp++;
                break;
            }
        }
        if (skip == 0) {
            counter++;
            return i; // found.
        }
    }
    return N; // not found.
}
```

Knuth-Morris-Pratt geanalyseerd

Hieronder staat de code uit het boek waarin de dubbele int array wordt gebruikt die dfa heet.

```
      public KnuthMorrisPratt(String pat){ // Build DFA from pattern.
            this.pat = pat;
            int M = pat.length();
            int R = 256;
            dfa = new int[R][M];
            dfa[pat.charAt(0)][0] = 1;
            for (int X = 0, j = 1; j < M; j++){ // Compute dfa[][j].
                  for (int c = 0; c < R; c++){
                        dfa[c][j] = dfa[c][X]; // Copy mismatch cases.
                        dfa[pat.charAt(j)][j] = j+1; // Set match case.
                        X = dfa[pat.charAt(j)][X]; // Update restart state.
                  }
            }
      }
```

Door de code aan te passen in de search methode kan het algoritme tellen hoe vaak een woord dus in de tekst voorkomt. Als de integer a groter is dan de dfa is het woord gevonden en wordt er +1 gedaan voor de int komtAantalKeerVoor die uiteindelijk gereturnd wordt.

```
public int search(String txt) { // Simulate operation of DFA on txt.
      int i, j, N = txt.length(), M = pat.length();
      int komtAantalKeerVoor = 0;
      for (i = 0, j = 0; i < N && j < M; i++) {
           int a = dfa[txt.charAt(i)][j];
           if(a > 0){
               komtAantalKeerVoor ++;
           }
      }
      return komtAantalKeerVoor;
      }
```

Beide algoritmes geven hetzelfde aantal keer voorkomen van de woorden, wat betekent dat ze allebei goed werken (of allebei fout, wat we niet hopen)!
Ook worden het aantal vergelijkingen gegeven per algoritme per woord.

Woord	Komt voor	Knut-Morris-Prat vergelijkingen	Boyer-Moore vergelijkingen
ik	318	145241	67904
van	438	145505	41450
Glimlachend	1	145549	1020
genie	3	145539	15999
in	1579	142888	66628
systematisch	0	145550	503
veldheer	1	145550	4219
het	552	145535	42434
geel	39	145512	26053
zat	58	145509	40911

Wat we uit de gegevens kunnen opmaken is dat Knuth-Morris-Prat (KMP) bijna consistent blijft aan de lengte van de tekst terwijl Boyer-Moore zwiepen maakt. De vergelijkingen zijn allebei in lijn zijn met de tabel “cost summary for substring search implementations” in het boek(Sedgewick & Wayne, 2011, p. 779). KMP is grof gezegd lineair en Boyer Moore is inderdaad sublineair met een factor M (de pattern length). Zo is het woord “van” 4 letters lang. Door vervolgens145512 te delen door 3 komen we op ongeveer 48501.
In de tabel is dat 41450. Als we het woord “Glimlachend” nemen (11 letters lang) en de KMP-waarde, 145549 delen door 11, komen we op 13140. In de tabel is dat 1020. Dit is wel ongeveer een factor 10 kleiner dan het hoort te zijn. Als je de rest van de waardes neemt wijken ze ook een beetje af (niet zoveel als voor Glimlachend) dus ik zou zeggen dat het wel klopt.

Reguliere expressies

Deze laatste subopdracht gaat over reguliere expressies. De bedoeling is dus ommet de java class "`java.util.regex.Pattern`" een methode "boolean checkURL (String url)" te schrijven die controleert of een url voldoet aan de volgende regels.

1. De URL begint met http(s) of ftp(s).
2. Het top-level domein is of nl of edu.
3. Er is minimaal één derde-level domein en als het derde-level domein www is dan is er minimaal een vierde-level domein.
4. Het second-level domein is nooit korter dan 3 karakters.
5. Tussen iedere slash zit minimaal 2 karakters en die begint nooit met een cijfer.
6. Als er parameters voorkomen in de URL dan eindigt de naam van de parameter altijd met een cijfer.

Ik heb aan de hand van de bovengenoemde regels de onderstaande code geschreven

```
public static boolean checkURL(String
    string) { Pattern pattern =
    Pattern.compile(""
```

1. **De URL begint met http(s) of ftp(s).**
   ```
   + "^(https?|ftps?)://"
   ```

2. **Het top-level domein is of nl of edu.**
   ```
   + "(nl|edu)"
   ```

3. **als het derde-level domein www is dan is er minimaal een vierde-level domein.**
   ```
   + "((www.([a-zA-Z0-9]+)+)"
   ```

4. **Het second-level domein is nooit korter dan 3 karakters.**
   ```
   + "[a-zA-Z0-9]{3,}(.|/)"
   ```

5. **Tussen iedere slash zit minimaal 2 karakters en die begint nooit met een cijfer.**
   ```
   + "(/[a-zA-Z][a-zA-Z0-9\\-]+)*"
   ```

6. **Als er parameters voorkomen in de URL dan eindigt de naam van de parameter altijd met een cijfer.**
   ```
   + "(/[a-zA-Z0-9]+\\?(&?[a-zA-Z0-9]*[0-9]=[a-zA-Z0-9]+)*)*");
   ```

```
    Matcher match =
    pattern.matcher(string);
    return match.matches();
}
```

Aan de hand van de bovengenoemde code volgen er een aantal tests:

https://www.jumia.com.gh/
kan niet want:
2. Het top level domein is niet nl of edu, voldoet niet.
3. Er is minimaal één derde-level domein en als het derde-level domein www is dan is er minimaal een vierde-level domein, voldoet niet.

https://www.google.nl/
kan niet want:
3. Er is minimaal één derde-level domein en als het derde-level domein www is dan is er minimaal een vierde-level domein, voldoet niet.

Ntp://nl.ea.nl
kan niet want:
4. Het second-level domein is nooit korter dan 3 karakters, voldoet niet.

http://html.nl/page.php?id=1254
Kan wel want:
- voldoet aan alle eisen (de parameter is er en de url eindigt met een cijfer)

https://us.stanford.edu/site/
Kan wel want:
- voldoet aan alle eisen

Als laatst nog de twee testgevallen uit het boek

http://lib.hva/nl
Kan wel want:
- voldoet aan alle eisen

ntp://www.hva.nl/a/b?tijd=UTC
Kan niet want:
- 5. Tussen iedere slash zit minimaal 2 karakters en die begint nooit met een cijfer, voldoet niet.
- 1. De URL begint met http(s) of ftp(s) , voldoet niet.
- 6. Als er parameters voorkomen in de URL dan eindigt de naam van de parameter altijd met een cijfer, voldoet niet.
- 3. Er is minimaal één derde-level domein en als het derde-level domein www is dan is er minimaal een vierde-level domein, voldoet niet.