GENIOMHE

Sequence algorithms

by Samuel Ortion

Prof.: Fariza Tahi

2024

Contents

1 Back to basics 3

2 Motif 5

3 Matrices 7

3.1 Automata 10

3.2 Suffix Automaton 12

1 Back to basics

Algorithm 1 Search an element in an array

```
1: function Search(A: Array(n), E: element)
2:     for (i = 0; i < n; i++) do
3:         if A[i] = E then
4:             return true
5:         end if
6:     end for
7:     return false
8: end function
```

Algorithm 2 Search an element in an array using a while loop

```
1: function Search(A: Array(n), E: element)
2:     i ← 0
3:     while i < n do
4:         if A[i] = E then
5:             return true
6:         end if
7:         i ← i + 1
8:     end while
9:     return false
10: end function
```

Algorithm 3 Search an element in an array using a while loop (bis)

```
1: function Search(A: Array(n), E: element)
2:     i ← 0
3:     while i < n and A[i] ≠ E do
4:         i ← i + 1
5:     end while
6:     if i = n then
7:         return false
8:     else
9:         return true
10:    end if
11: end function
```

Algorithm 4 Count the occurrences of an element in an array

```
1: function Search(A: Array(n), E: element)
2:     c ← 0
3:     for (i = 0; i < n; i + +) do
4:         if A[i] = E then
5:             c ← c + 1
6:         end if
7:     end for
8:     return c
9: end function
```

2 Motif

Algorithm 5 Brute-force search of a motif in a sequence

```
1:  function FindMotif(S: Array(n), M: Array(m))
2:      returns a list of position
3:      pos ← {}
4:      i ← 0
5:      while i < n − m + 1 do
6:          j ← 0
7:          while j < m and S[i + j] = M[j] do
8:              j + +
9:          end while
10:         if j = m then
11:             pos ← pos ∪ {i}
12:         end if
13:         i + +
14:     end while
15:     return pos
16: end function
```

Algorithm 6 Knuth-Morris-Pratt algorithm

```
1: function KMP_Search(S: Array(n), M: Array(m))
2:     returns Integer
3:     table ← KMP_Table(M)
4:     c ← 0                                   ▷ Count the number of matches
5:     i ← 0
6:     j ← 0
7:     while i < n do
8:         if S[i] = M[i] then
9:             i ← i + 1
10:            j ← j + 1
11:        end if
12:        if j = m then
13:            c ← c + 1
14:            j ← table[j − 1]
15:        else if j < n and M[j] ≠ S[i] then
16:            if j ≠ 0 then
17:                j ← table[j − 1]
18:            else
19:                i ← i + 1
20:            end if
21:        end if
22:    end while
23:    return c
24: end function
25: function KMP_Table(M: Array(m))
26:    Returns Array(m)
27:    previous ← 0
28:    table ← array of zeros of size m
29:    for i = 0; i < m; i + + do
30:        if M[i] = M[previous] then
31:            previous ← previous + 1
32:            table[i] ← previous
33:            i ← i + 1
34:        else
35:            if previous = 0 then
36:                previous ← table[previous − 1]
37:            else
38:                table[i] ← 0
39:                i ← 1
40:            end if
41:        end if
42:    end for
43: end function
```

3 Matrices

Let S_1 and S_2 be two sequences.

$S_1 = \text{ACGUUCC}$ $S_2 = \text{GUU}$

Let $n = |S_1|$, $m = |S_2|$ The complexity of this algorithm is $\mathcal{O}(n \cdot m)$ to build the matrix, and it requires also to find the diagonals and thus it is a bit less efficient than the Algorithm 5.

To find repetitions, we can use a comparison matrix with a single sequence against itself. A repetition would appear as a diagonal of ones, not on the main diagonal.

Let $S = \text{ACGUUACGUU}$. Let's write the comparison matrix.

Algorithm 7 Construct a comparison matrix

```
1:  function ComparisonMatrix(S: Array(n))
2:      M ← Array(n, n)
3:      for (i = 0; i < n; i++) do
4:          for j = 0; j < n; j++ do
5:              if S[i] = S[j] then
6:                  M[i][j] = 1
7:              else
8:                  M[i][j] = 0
9:              end if
10:         end for
11:     end for
12:     return M
13: end function
```

	A	C	G	U	U	C	C
G	0	0	1	0	0	0	0
U	0	0	0	1	1	0	0
U	0	0	0	1	1	0	0

Table 3.1 Comparison matrix

	A	C	G	U	U	A	C	G	U	U	G	U	U
A	1	0	0	0	0	1	0	0	0	0	0	0	0
C	0	1	0	0	0	0	1	0	0	0	0	0	0
G	0	0	1	0	0	0	0	1	0	0	1	0	0
U	0	0	0	1	1	0	0	0	1	1	0	1	1
U	0	0	0	1	1	0	0	0	1	1	0	1	1
A	1	0	0	0	0	1	0	0	0	0	0	0	0
C	0	1	0	0	0	0	1	0	0	0	0	0	0
G	0	0	1	0	0	0	0	1	0	0	1	0	0
U	0	0	0	1	1	0	0	0	1	1	0	1	1
U	0	0	0	1	1	0	0	0	1	1	0	1	1
G	0	0	1	0	0	0	0	1	0	0	1	0	0
U	0	0	0	1	1	0	0	0	1	1	0	1	1
U	0	0	0	1	1	0	0	0	1	1	0	1	1

Algorithm 8 Construct the top half of a comparison matrix

```
1:  function ComparisonMatrix(S: Array(n))
2:      M ← Array(n,n)
3:      for (i = 0; i < n; i++) do
4:          for j=i; j < n; j++ do
5:              if S[i] = S[j] then
6:                  M[i][j] = 1
7:              else
8:                  M[i][j] = 0
9:              end if
10:         end for
11:     end for
12:     return M
13: end function
```

Algorithm 9 Find repetitions (with a set of visited segments)

```
1:  function FindRepetions(S: Array(n))
2:      returns A list of start and end positions for repeated sequences
3:      M = ComparisonMatrix(S)
4:      pos = {}
5:      visited = {}
6:      for (i_start = 0; i_start < n; i_start++) do
7:          for (j_start = i_start + 1; j_start < n; j_start++) do
8:              if M[i_start][j_start] = 1 and (i_start, j_start) ∉ visited then
9:                  i = i_start
10:                 j = j_start
11:                 while M[i][j] = 1 do
12:                     i++
13:                     j++
14:                     visited = visited ∪ {(i, j)}
15:                 end while
16:                 pos = pos ∪ {(i_start, i), (j_start, j)}
17:             end if
18:         end for
19:     end for
20: end function
```

Algorithm 10 Find repetitions with an exploration of diagonals

```
1:  function FindRepetions(S: Array(n))
2:      returns A list of start and end positions for repeted sequences
3:      M = ComparisonMatrix(S)
4:      pos = {}
5:      for (diag = 1; diag < n; diag++) do
6:          j = diag
7:          i = 0
8:          while i < n and j < n do
9:              if M[i][j] = 1 then
10:                 i_start = i
11:                 j_start = j
12:                 while i < n and j < n and M[i][j] = 1 do
13:                     i++
14:                     j++
15:                 end while
16:                 pos = pos ∪ {((i_start, i − 1), (j_start, j − 1))}
17:             end if
18:             i++
19:             j++
20:
21:         end while
22:     end for
23: end function
```

3.1 Automata

An automaton is a tuple $\langle S, s_0, T, \Sigma, f \rangle$

- S the set of states
- s_0 the initial state
- T the set of terminal states
- Σ the alphabet
- f the transition function $f : (s_1, c) \rightarrow s_2$

Example Given the language L on the alphabet $\Sigma = \{A, C, T\}$, $L = \{A^*, CTT, CA^*\}$

Definition 1: Deterministic automaton

An automaton is deterministic, if for each couple $(p, a) \in S \times \Sigma$ it exists at most a state q such as $f(p, q) = q$

Definition 2: Complete automaton

An automaton is complete, if for each couple $(p, a) \in S \times \Sigma$ it exists at least a state q such as $f(p, q) = q$.

Algorithm 11 Find repetitions with an exploration of diagonals, without nested while

1: **function** FindRepetions(S: Array(n))
2:     **returns** A list of start positions for repeted sequences and match length
3:     M = ComparisonMatrix(S)
4:     $pos = \{\}$
5:     **for** $(diag = 1;\ diag < n;\ diag++)$ **do**
6:         $j = diag$
7:         $i = 0$
8:         $l = 0$
9:         **while** $i < n$ and $j < n$ **do**
10:             **if** $M[i][j] = 1$ **then**
11:                 $l++$
12:             **else**
13:                 **if** $l > 0$ **then**
14:                     $pos = pos \cup \{(i - l, j - l, l)\}$
15:                     $l = 0$
16:                 **end if**
17:             **end if**
18:             $i++$
19:             $j++$
20:         **end while**
21:         **if** $l > 0$ **then**
22:             $pos = pos \cup \{((i - l, j - l, l))\}$
23:         **end if**
24:     **end for**
25:     **return** pos
26: **end function**

Algorithm 12 Find repetitions

1: **function** FindRepetions(S: Array(n))
2:     **returns** A list of start and end positions for repeted sequences
3:     M = ComparisonMatrix(S)
4:     $pos = \{\}$
5:     **for** $i_{start} = 0;\ i_{start} < n;\ i_{start}++$ **do**
6:         **for** $j_{start} = i_{start} + 1;\ j_{start} < n;\ j_{start}++$ **do**
7:             **if** $M[i_{start}][j_{start}] = 1$ **then**
8:                 $i = i_{start}$
9:                 $j = j_{start}$
10:                 **while** $M[i][j] = 1$ **do**
11:                     $M[i][j] = 0$ ▷ *Ensure that the segment is not explored again*
12:                     $i++$
13:                     $j++$
14:                 **end while**
15:                 $pos = pos \cup \{((i_{start}, i - 1), (j_{start}, j - 1))\}$
16:             **end if**
17:         **end for**
18:     **end for**
19: **end function**

Algorithm 13 Check wether a word belong to a language for which we have an automaton

```
1:  function WordInLanguage(W: Array(n), A: ⟨S, s_0, T, Σ, f⟩)
2:      returns A Boolean valued to true if the word is recognized by the language automaton
3:      s ← s_0
4:      i ← 0
5:      while i < n do
6:          a ← W[i]
7:          if ∃f(s, a) then
8:              s ← f(s, a)
9:          else
10:             return false
11:         end if
12:         i++
13:     end while
14:     if s ∈ T then
15:         return true
16:     else
17:         return false
18:     end if
19: end function
```

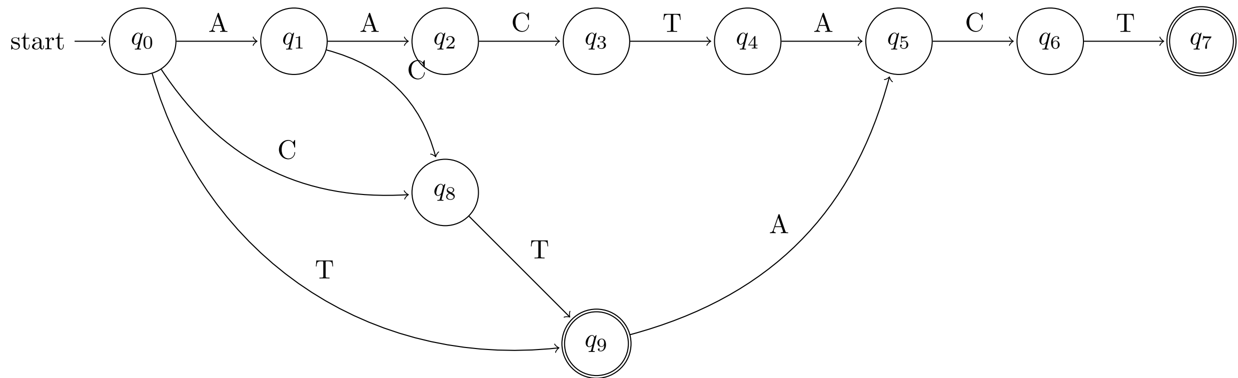

Figure 3.1 Suffix automaton for S = AACTACT

3.2 Suffix Automaton

Let S = AACTACT

A suffix automata recognize all suffix of a given sequence.

The suffix language of S is $\{S, ACTACT, CTACT, TACT, ACT, CT, T\}$.

The complexity of the pattern matching algorithm is $\mathcal{O}(n + m)$, because building the automaton is $\mathcal{O}(m)$

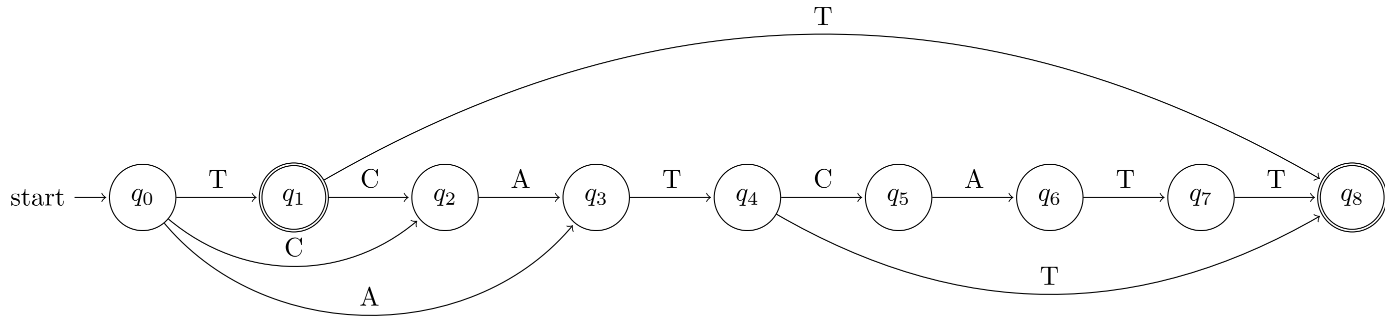

Figure 3.2 Suffix automaton for S = TCATCATT

Algorithm 14 Check if a sequences matches a motif, from a suffix automaton $\mathcal{O}(m)$

1: **function** CheckMotifInSuffixAutomaton(W: Array(m), A: $\langle S, s_0, T, \Sigma, f\rangle$)
2: **returns** Boolean valued to `true` if the motif is in the sequence
3: $s \leftarrow s_0$
4: $i \leftarrow 0$
5: **while** $i < m$ and $\exists f(s, W[i])$ **do**
6: $s \leftarrow f(s, W[i])$
7: $i{+}{+}$
8: **end while**
9: **if** $i = n$ **then**
10: **return** `true`
11: **else**
12: **return** `false`
13: **end if**
14: **end function**