GENIOMHE

Sequence algorithms

by Samuel Ortion

Prof.: Fariza Tahi

2024

Contents

Motifs algorithms

1 Motif

1.1 Searching a substring in a string

Algorithm 1 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
```

1.2 Using matrices to search motifs

Let S_1 and S_2 be two sequences.

S_1 = ACGUUCC S_2 = 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 **??**.

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 = ACGUUACGUU. Let's write the comparison matrix.

Algorithm 2 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
```

	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 1.1 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 3 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
```

Algorithm 4 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 5 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 6 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
```

1.3 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 $f : (s_1, c) \to s_2$

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 7 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 ∪ {(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 ∪ {((i − l, j − l, l))}
23:        end if
24:    end for
25:    return pos
26: end function
```

Algorithm 8 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 ∪ {((i_start, i − 1), (j_start, j − 1))}
16:            end if
17:        end for
18:    end for
19: end function
```

Algorithm 9 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
```

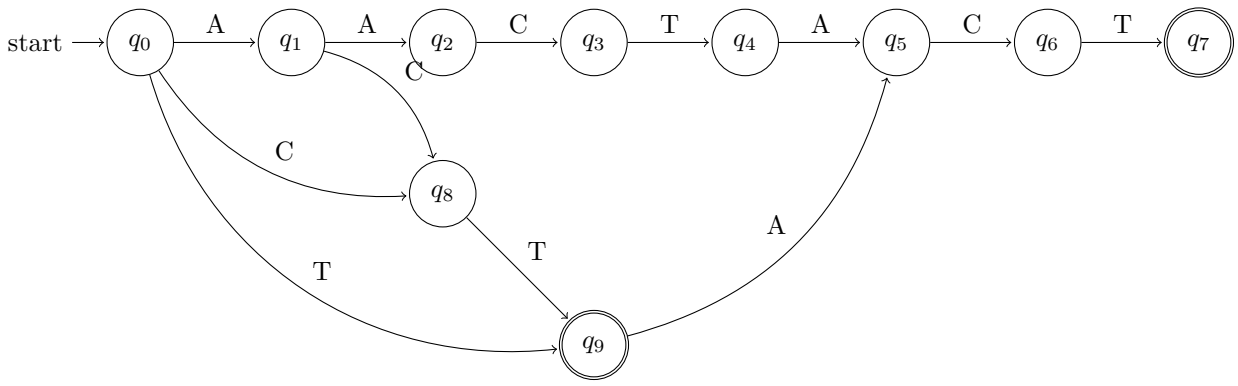

Figure 1.1 Suffix automaton for S = AACTACT

1.3.1 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 $\mathcal{O}(m)$

1.3.2 Automata for motif search

Let M be a motif M = ACAT.

The alphabet of motif is the same as the alphabet of the sequence. The search automaton is complete. If the there exists a letter c in the sequence that is not in the motif alphabet, we can make a virtual transition from each state to the initial state whenever we encounter an unknown letter.

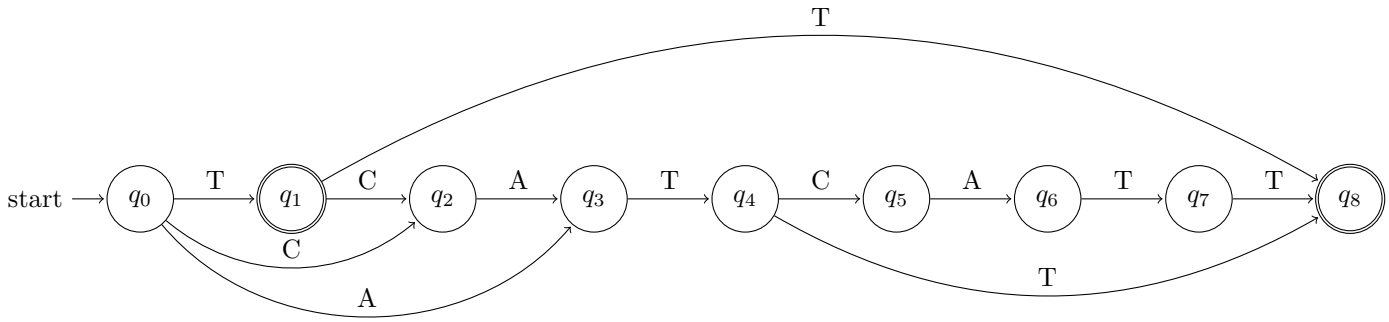

Figure 1.2 Suffix automaton for S = TCATCATT

Algorithm 10 Check if a sequences matches a motif, from a suffix automaton $\mathcal{O}(m)$, built from the automaton

```
1:  function CheckMotifInSuffixAutomaton(W: Array(m), A: ⟨S, s_0, T, Σ, f⟩)
2:      returns Boolean valued to true if the motif is in the sequence
3:      s ← s_0
4:      i ← 0
5:      while i < m and ∃f(s, W[i]) do
6:          s ← 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
```

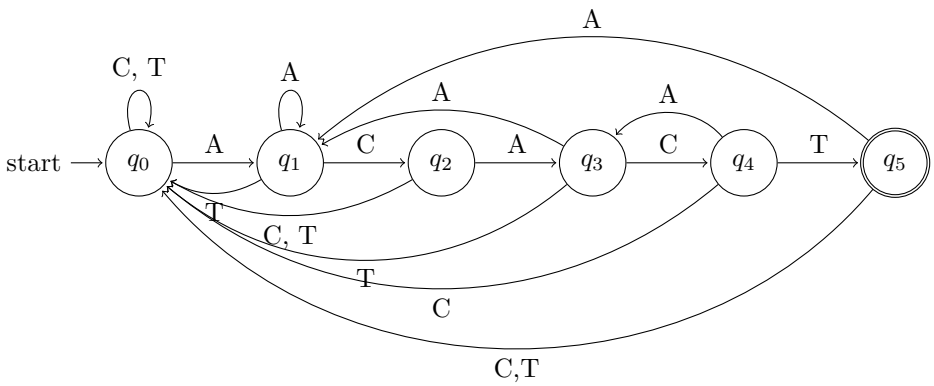

Figure 1.3 Motif search automaton for M = ACAT

Algorithm 11 Search a motif in a sequence with an automaton

```
1: function SearchMotif(S: Array(n), A: ⟨S, s_0, T, Σ, f⟩, P: Array(m))
2:     returns A set of positions where the motif has been found
3:     s ← s_0
4:     i ← 0
5:     pos ← {}
6:     while i < n do
7:         if s ∈ T then
8:             pos ← pos ∪ {i − m}
9:         end if
10:        s ← f(s, S[i])
11:        i + +
12:    end while
13:    return pos
14: end function
```

Algorithm 12 Check if the a motif automaton recognizes only the prefix of size $m-1$ of a motif P of size m

```
1: function SearchMotifLastPrefix(S: Array(n), A: ⟨S, s_0, T, Σ, f⟩, P: Array(m))
2:     returns A set of positions where the motif has been found
3:     s ← s_0
4:     i ← 0
5:     T_new ← {}
6:     for s ∈ S do
7:         for a ∈ Σ do
8:             for t ∈ T do
9:                 if ∃f(s, a) and f(s, a) = t then
10:                    T_new ← T_new ∪ s
11:                end if
12:            end for
13:        end for
14:    end for
15:    while i < n do
16:        if s ∈ T_new then
17:            return true
18:        end if
19:        s ← f(s, S[i])
20:        i + +
21:    end while
22:    return false
23: end function
```

Algorithm 13 Check if the a motif automaton recognizes only the prefix of size $m-1$ of a motif P of size m, knowing the sequence of the motif

```
1: function SearchMotifLastPrefix(S: Array(n), A: ⟨S, s_0, T, Σ, f⟩, P: Array(m))
2:     returns A set of positions where the motif has been found
3:     s ← s_0
4:     i ← 0
5:     while i < n and f(s, P[m − 1]) ∉ T do
6:         s ← f(s, S[i])
7:         i + +
8:     end while
9:     if f(s, P[m − 1]) ∈ T then
10:        return true
11:    else
12:        return false
13:    end if
14: end function
```

2 Longest common subsequence

Let $S_1 = \text{ATCTGAT}$ and $S_2 = \text{TGCATA}$. In this case the longest common subsequence of S_1 and S_2 is $TCTA$.

Algorithm 14 Construct a longest common subsequence matrix

```
1:  function LCSQ_Matrix(S1: Array(n), S2: Array(m))
2:      M ← Array(m + 1, n + 1)
3:      for (i = 0; i < n + 1; i++) do
4:          for j = 0; j < m + 1; j++ do
5:              if i = 0 or j = 0 then
6:                  M[i][j] = 0
7:              else
8:                  if S1[i] = S2[j] then
9:                      match = M[i − 1][j − 1] + 1
10:                 else
11:                     match = M[i − 1][j − 1]
12:                 end if
13:                 gap1 = M[i − 1][j]
14:                 gap2 = M[i][j − 1]
15:                 M[i][j] = max{match, gap1, gap2}
16:             end if
17:         end for
18:     end for
19:     return M
20: end function
```

Algorithm 15 Construct a longest common subsequence matrix keeping the path in memory

```
1: function LCSQ_Matrix_Path(S_1: Array(n), S_2: Array(m))
2:     M ← Array(m + 1, n + 1)
3:     P ← Array(m + 1, n + 1)
4:     for (i = 0; i < n + 1, i + +) do
5:         M[i][0] ← 0
6:     end for
7:     for (j = 0; j < m + 1; j+) do
8:         M[0][j] ← 0
9:     end for
10:    for (i = 1; i < n + 1; i + +) do
11:        for (j = 1; j < m + 1; j + +) do
12:            if i = 1 or j = 0 then
13:                M[i][j] = 0
14:            else
15:                if S_1[i − 1] = S_2[j − 1] then
16:                    M[i][j] ← M[i − 1][j − 1] + 1
17:                    P[i][j] ←'↖'
18:                else if M[i][j − 1] ≥ M[i − 1][j] then
19:                    M[i][j] ← M[i][j − 1]
20:                    P[i][j] ←'←'
21:                else
22:                    M[i][j] ← M[i − 1][j]
23:                    P[i][j] ←'↓'
24:                end if
25:            end if
26:        end for
27:    end for
28:    return M, P
29: end function
```

Algorithm 16 Backtrack the longest common subsequence

```
1: function LCSQ(S_1: Array(n), S_2: Array(m))
2:     M, P ← LCSQ_Matrix(S_1, S_2)
3:     L ← Array(M[n][m])
4:     k ← 0
5:     i ← n
6:     j ← m
7:     while i > 0 and j > 0 do
8:         if P[i][j] ='↖' then
9:             L[k] ← S_1[i]
10:            i − −
11:            j − −
12:            k + +
13:        else if P[i][j] ='←' then
14:            j − −
15:        else
16:            i − −
17:        end if
18:    end while
19:    return L
20: end function
```

Sequence alignment

3 Definitions

A function d is a distance between two sequences x and y in an alphabet Σ if

- $x, y \in \Sigma^*, d(x, x) = 0$
- $\forall x, y \in \Sigma^*\ d(x, y) = d(y, x)$
- $\forall x, y, z \in \Sigma^*\ d(x, z) \leq d(x, y) + d(x, z)$

Here we are interested by the distance that is able to represent the transformation of x to y using three types of basic operations:

- Substition
- Insertion
- Deletion

Example:

- $sub(a, b) = \begin{cases} 0 & \text{if} a = b \\ 1 & \text{otherwise} \end{cases}$.
- $del(a) = 1$
- $ins(a) = 1$

Let $X = x_0 x_1 \dots x_{m-1}$, $Y = y_0 y_1 \dots y_{n-1}$
An alignment is noted as $z = \begin{pmatrix} \bar{x}_0 \\ \bar{y}_0 \end{pmatrix} \dots \begin{pmatrix} \bar{x}_{p-1} \\ \bar{y}_{p-1} \end{pmatrix}$ of size p. $n \leq p \leq n + m$
$\bar{x}_i = x_j$ or $\bar{x}_i = \varepsilon$ for $0 \leq i \leq p - 1$ and $0 \leq j \leq m - 1$
$\bar{y}_i = y_j$ or $\bar{y}_i = \varepsilon$ for $0 \leq i \leq p - 1$ and $0 \leq j \leq n - 1$
$X' = \bar{x}_0 \bar{x}_1 \dots \bar{x}_i \dots \bar{x}_{p-1}$ $Y' = \bar{y}_0 \bar{y}_1 \dots \bar{y}_i \dots \bar{y}_{p-1}$ for $0 \leq i \leq p - 1$, $\nexists i$, such that $\bar{x}_i = \bar{y}_i = \varepsilon$

4 Sequence alignment

Algorithm 17 Needleman-Wunsch Algorithm

```
1:  procedure FillMatrix(S_1: Array(m), S_2: Array(n)) ▷ sub(a,b) is the substitution score, del(a) and ins(a) are the deletion and insertion penalty, in regard with the reference S_1 sequence
2:      M = Array(m + 1, n + 1)                ▷ Initialize the matrix first column and first row
3:      P = Array(m, n)                        ▷ Store the direction of the cell we chose to build the next cell up on.
4:      M[0][0] = 0
5:      for (i = 1; i < m + 1; i + +) do
6:          M[i][0] = M[i − 1][0] + gap_penalty
7:      end for
8:      for (j = 1; j < n + 1; j + +) do
9:          M[0][j] = M[0][j − 1] + gap_penalty
10:     end for                                ▷ Fill the remaining matrix
11:     for (i = 1; i < m + 1; i + +) do
12:         for (j = 1; j < n + 1; j + +) do
13:             delete = M[i − 1][j] + gap_penalty
14:             insert = M[i][j − 1] + gap_penalty
15:             substitute = M[i − 1][j − 1] + sub(S_1[i − 1], S_2[j − 1])
16:             choice = min{delete, insert, substitute}
17:             if substitute = choice then
18:                 P[i − 1][j − 1] =′↖′
19:             else if deletion = choice then
20:                 P[i − 1][j − 1] =′←′
21:             else
22:                 P[i − 1][j − 1] =′↑′
23:             end if
24:             M[i][j] = choice
25:         end for
26:     end for
27: end procedure
```

Algorithm 18 Needleman-Wunsch Algorithm (Backtrack)

```
1: procedure ShowAlignment(S_1: Array(m), S_2: Array(n))
2:     extend_1 =''
3:     extend_2 =''
4:     i = m
5:     j = n
6:     while i > 0 and j > 0 do
7:         if P[i-1][j-1] ='↖' then
8:             extend_1 = S_1[i-1] ∘ extend_1
9:             extend_2 = S_2[j-1] ∘ extend_2
10:            i − −
11:            j − −
12:        else if P[i-1][j-1] ='↑' then
13:            extend_1 = S_1[i-1] ∘ extend_1
14:            extend_2 =' −' ∘ extend_2
15:            i − −
16:        else
17:            extend_1 =' −' ∘ extend_1
18:            extend_2 = S_2[j-1] ∘ extend_2
19:            j − −
20:        end if
21:    end while
22:    while i > 0 do
23:        extend_1 = S_1[i-1] ∘ extend_1
24:        extend_2 =' −' ∘ extend_2
25:        i − −
26:        Insert(0, alignment,tuple)
27:    end while
28:    while j > 0 do
29:        extend_1 =' −' ∘ extend_1
30:        extend_2 = S_2[j-1] ∘ extend_2
31:        j − −
32:    end while
33:    print(extend_1)
34:    print(extend_2)
35: end procedure
36: FillMatrix(S_1, S_2)
37: ShowAlignment(S_1, S_2)
```

Algorithm 19 Needleman-Wunsch Algorithm, using proper notation

```
1: procedure FillMatrix(S_1: Array(m), S_2: Array(n))
2:     M = Array(m + 1, n + 1)
3:     P = Array(m, n)                ▷ Store the direction of the cell we chose to build the next cell up on.
4:     M[0][0] = 0
5:     for (i = 1; i < m + 1; i++) do
6:         M[i][0] = M[i − 1][0] + gap_penalty
7:     end for
8:     for (j = 1; j < n + 1; j++) do
9:         M[0][j] = M[0][j − 1] + gap_penalty
10:    end for
11:    for (i = 1; i < m + 1; i++) do
12:        for (j = 1; j < n + 1; j++) do
13:            delete = M[i − 1][j] + gap_penalty
14:            insert = M[i][j − 1] + gap_penalty
15:            substitute = M[i − 1][j − 1] + sub(S_1[i − 1], S_2[j − 1])
16:            M[i][j] = min{substitute, insert, delete}
17:        end for
18:    end for
19: end procedure
```

Algorithm 20 Needleman-Wunsch Algorithm, using proper notation (Backtrack)

1: **procedure** BacktrackAlignment(S_1: Array(m), S_2: Array(n))
2: $alignment = LinkedList$
3: $i = m$
4: $j = n$
5: **while** $i > 0$ and $j > 0$ **do**
6: **if** $M[i-1][j-1] = M[i][j] - sub(S_1[i-1], S_2[j-1])$ **then**
7: $tuple = \begin{pmatrix} S_1[i-1] \\ S_2[j-1] \end{pmatrix}$
8: $i--$
9: $j--$
10: **else if** $M[i-1][j-1] = M[i][j-1] - gap\_penalty$ **then**
11: $tuple = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix}$
12: $i--$
13: **else**
14: $tuple = \begin{pmatrix} \varepsilon \\ S_2[j-1] \end{pmatrix}$
15: $j--$
16: **end if**
17: Insert(0, *alignment*,*tuple*)
18: **end while**
19: **while** $i > 0$ **do**
20: $tuple = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix}$
21: $i--$
22: Insert(0, *alignment*,*tuple*)
23: **end while**
24: **while** $j > 0$ **do**
25: $tuple = \begin{pmatrix} \varepsilon \\ S_2[j-1] \end{pmatrix}$
26: $j--$
27: Insert(0, *alignment*,*tuple*)
28: **end while**
29: **end procedure**
30: FillMatrix(S_1, S_2)
31: BacktrackAlignment(S_1, S_2)

Algorithm 21 Backtrack a single alignment in a recursive way

1: S_1: Array(m), S_2: Array(n), M: Array($m+1$, $n+1$),
2: **function** BacktrackRecurse(i, j)
3: **if** $i > 0$ and $j > 0$ **then**
4: $substitute = M[i-1][j-1]$
5: $delete = M[i-1][j]$
6: $insert = M[i][j-1]$
7: $min = \min\{substitute, delete, insert\}$
8: **if** $substitute = min$ **then**
9: $z =$ BacktrackRecurse(S_1, S_2, M, $i-1$, $j-1$)
10: $z = \begin{pmatrix} S_1[i-1] \\ S_2[j-1] \end{pmatrix} \circ z$
11: **else if** $delete = min$ **then**
12: $z =$ BacktrackRecurse(S_1, S_2, M, $i-1$, j)
13: $z = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix} \circ z$
14: **else**
15: $z =$ BacktrackRecurse(S_1, S_2, M, i, $j-1$)
16: $z = \begin{pmatrix} \varepsilon \\ S_2[j-1] \end{pmatrix} \circ z$
17: **end if**
18: **else if** $i > 0$ **then**
19: $z =$ BacktrackRecurse(S_1, S_2, M, $i-1$, j)
20: $z = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix} \circ z$
21: **else if** $j > 0$ **then**
22: $z =$ BacktrackRecurse(S_1, S_2, M, i, $j-1$)
23: $z = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix} \circ z$
24: **else**
25: **return** []
26: **end if**
27:
28: **return** z
29:
30: **end function**
31: **function** Backtrack(S_1: Array(m), S_2: Array(n), M: Array($m+1$, $n+1$))
32: **return** BacktrackRecurse(S_1, S_2, M, m, n)
33: **end function**

Algorithm 22 Backtrack all the optimum alignments in a recursive way

1: **procedure** BacktrackRecurse(S_1: Array(m), S_2: Array(n), M: Array($m+1$, $n+1$), i, j)
2: **if** $i > 0$ and $j > 0$ **then**
3: $substitute = M[i-1][j-1]$
4: $delete = M[i-1][j]$
5: $insert = M[i][j-1]$
6: $min = \min\{substitute, delete, insert\}$
7: **if** $substitute = min$ **then**
8: $value = \begin{pmatrix} S_1[i-1] \\ S_2[j-1] \end{pmatrix}$
9: $z' = value \circ z$
10: BacktrackRecurse(S_1, S_2, M, $i-1$, $j-1$, z')
11: **end if**
12: **if** $delete = min$ **then**
13: $value = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix}$
14: $z' = value \circ z$
15: BacktrackRecurse(S_1, S_2, M, $i-1$, j, z')
16: **end if**
17: **if** $insert = min$ **then**
18: $value = \begin{pmatrix} \varepsilon \\ S_2[j-1] \end{pmatrix}$
19: $z' = value \circ z$
20: BacktrackRecurse(S_1, S_2, M, i, $j-1$, z')
21: **end if**
22: **else if** $i > 0$ **then**
23: $value = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix}$
24: $z' = value \circ z$
25: BacktrackRecurse(S_1, S_2, M, $i-1$, j, z')
26: **else if** $j > 0$ **then**
27: $value = \begin{pmatrix} \varepsilon \\ S_2[j-1] \end{pmatrix}$
28: $z' = value \circ z$
29: BacktrackRecurse(S_1, S_2, M, i, $j-1$, z')
30: **else**
31: print(z)
32: **end if**
33: **end procedure**
34: **procedure** Backtrack(S_1: Array(m), S_2: Array(n), M: Array($m+1$, $n+1$))
35: **return** BacktrackRecurse(S_1, S_2, M, m, n, [])
36: **end procedure**